Article
Mikhail Khomenko · Nov 25, 2020 18m read

InterSystems Kubernetes Operator Deep Dive: Introduction to Kubernetes Operators

Introduction

Several resources tell us how to run IRIS in a Kubernetes cluster, such as Deploying an InterSystems IRIS Solution on EKS using GitHub Actions and Deploying InterSystems IRIS solution on GKE Using GitHub Actions. These methods work but they require that you create Kubernetes manifests and Helm charts, which might be rather time-consuming.
To simplify IRIS deployment, InterSystems developed an amazing tool called InterSystems Kubernetes Operator (IKO). A number of official resources explain IKO usage in details, such as New Video: Intersystems IRIS Kubernetes Operator and InterSystems Kubernetes Operator.

Kubernetes documentation says that operators replace a human operator who knows how to deal with complex systems in Kubernetes. They provide system settings in the form of *custom resources*. An operator includes a *custom controller* that reads these settings and performs steps the settings define to correctly set up and maintain your application. The custom controller is a simple pod deployed in Kubernetes. So, generally speaking, all you need to do to make an operator work is deploy a controller pod and define its settings in custom resources.
You can find high-level explanation of operators in How to explain Kubernetes Operators in plain English. Also, a free O ' Reilly ebook is available for download.
In this article, we ' ll have a closer look at what operators are and what makes them tick. We ' ll also write our own operator.

Prerequisites and Setup

To follow along, you ' ll need to install the following tools:

kind

```
$ kind --version
kind version 0.9.0
```

golang

```
$ go version
go version go1.13.3 linux/amd64
```

kubebuilder

```
$ kubebuilder version
Version: version.Version{KubeBuilderVersion:"2.3.1"…
```

kubectl

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.11"...
```

operator-sdk

```
$ operator-sdk version
operator-sdk version: "v1.2.0"…
```

Custom Resources

API resources is an important concept in Kubernetes. These resources enable you to interact with Kubernetes via HTTP endpoints that can be grouped and versioned. The standard API can be extended with custom resources, which require that you provide a Custom Resource Definition (CRD). Have a look at the Extend the Kubernetes API with CustomResourceDefinitions page for detailed info.
Here is an example of a CRD:

```
$ cat crd.yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: irises.example.com
spec:
  group: example.com
  version: v1alpha1
  scope: Namespaced
  names:
    plural: irises
    singular: iris
    kind: Iris
    shortNames:
    - ir
  validation:
    openAPIV3Schema:
      required: ["spec"]
      properties:
        spec:
          required: ["replicas"]
          properties:
            replicas:
              type: "integer"
              minimum: 0
```

In the above example, we define the API GVK (Group/Version/Kind) resource as example.com/v1alpha1/Iris, with replicas as the only required field.

Now let ' s define a custom resource based on our CRD:

```
$ cat crd-object.yaml
apiVersion: example.com/v1alpha1
kind: Iris
metadata:
  name: iris
spec:
  test: 42
  replicas: 1
```

In our custom resource, we can define any fields in addition to replicas, which is required by the CRD.
After we deploy the above two files, our custom resource should become visible to standard kubectl.
Let ' s launch Kubernetes locally using kind, and then run the following kubectl commands:

```
$ kind create cluster
$ kubectl apply -f crd.yaml
$ kubectl get crd irises.example.com
NAME                  CREATED AT
irises.example.com    2020-11-14T11:48:56Z

$ kubectl apply -f crd-object.yaml
$ kubectl get iris
NAME   AGE
iris   84s
```

Although we ' ve set a replica amount for our IRIS, nothing actually happens at the moment. It ' s expected. We need to deploy a controller - the entity that can read our custom resource and perform some settings-based actions.
For now, let ' s clean up what we ' ve created:

```
$ kubectl delete -f crd-object.yaml
$ kubectl delete -f crd.yaml
```

Controller

A controller can be written in any language. We ' ll use Golang as Kubernetes ' " native " language. We could write a controller ' s logic from scratch but the good folks from Google and RedHat gave us a leg up. They have created two projects that can generate the operator code that will only require minimum changes – kubebuilder and operator-sdk. These two are compared at the kubebuilder vs operator-sdk page, as well as here: What is the difference between kubebuilder and operator-sdk #1758.

Kubebuilder

It is convenient to start our acquaintance with Kubebuilder at the Kubebuilder book page.

The Tutorial: Zero to Operator in 90 minutes video from the Kubebuilder maintainer might help as well.

Sample implementations of the Kubebuilder project can be found in the sample-controller-kubebuilder and in kubebuilder-sample-controller repositories.

Let ' s scaffold a new operator project:

```
$ mkdir iris
$ cd iris
$ go mod init iris # Creates a new module, name it iris
$ kubebuilder init --domain myardyas.club # An arbitrary domain, used below as a suff
ix in the API group
```

Scaffolding includes many files and manifests. The main.go file, for instance, is the entrypoint of code. It imports the controller-runtime library, instantiates and runs a special manager that keeps track of the controller run. Nothing to change in any of these files.

Let ' s create the CRD:

```
$ kubebuilder create api --group test --version v1alpha1 --kind Iris
Create Resource [y/n]
y
Create Controller [y/n]
y
…
```

Again, a lot of files are generated. These are described in detail at the Adding a new API page. For example, you can see that a file for kind Iris is added in api/v1alpha1/iristypes.go. In our first sample CRD, we defined the required replicas field. Let ' s create an identical field here, this time in the IrisSpec structure. We ' ll also add the DeploymentName field. The replicas ' count should be also visible in the Status section, so we need to make the following changes:

```
$ vim api/v1alpha1/iris_types.go
…
type IrisSpec struct {
        // +kubebuilder:validation:MaxLength=64
        DeploymentName string `json:"deploymentName"`
        // +kubebuilder:validation:Minimum=0
        Replicas *int32 `json:"replicas"`
}
…
type IrisStatus struct {
        ReadyReplicas int32 `json:"readyReplicas"`
}
…
```

After editing the API, we ' ll move to editing the controller boilerplate. All the logic should be defined in the Reconcile method (this example is mostly taken from mykindcontroller.go). We also add a couple of auxiliary methods and rewrite the SetupWithManager method.

```
$ vim controllers/iris_controller.go
```

```
…
import (
...
// Leave the existing imports and add these packages
        apps "k8s.io/api/apps/v1"
        core "k8s.io/api/core/v1"
        apierrors "k8s.io/apimachinery/pkg/api/errors"
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
        "k8s.io/client-go/tools/record"
)
// Add the Recorder field to enable Kubernetes events
type IrisReconciler struct {
        client.Client
        Log    logr.Logger
        Scheme *runtime.Scheme
        Recorder record.EventRecorder
}
…
// +kubebuilder:rbac:groups=test.myardyas.club,resources=iris,verbs=get;list;watch;cr
eate;update;patch;delete
// +kubebuilder:rbac:groups=test.myardyas.club,resources=iris/status,verbs=get;update
;patch
// +kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;up
date;delete
// +kubebuilder:rbac:groups="",resources=events,verbs=create;patch

func (r *IrisReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
    ctx := context.Background()
    log := r.Log.WithValues("iris", req.NamespacedName)
    // Fetch Iris objects by name
    log.Info("fetching Iris resource")
    iris := testv1alpha1.Iris{}
    if err := r.Get(ctx, req.NamespacedName, &iris); err != nil {
        log.Error(err, "unable to fetch Iris resource")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }
    if err := r.cleanupOwnedResources(ctx, log, &iris); err != nil {
        log.Error(err, "failed to clean up old Deployment resources for Iris")
        return ctrl.Result{}, err
    }

    log = log.WithValues("deployment_name", iris.Spec.DeploymentName)
    log.Info("checking if an existing Deployment exists for this resource")
    deployment := apps.Deployment{}
    err := r.Get(ctx, client.ObjectKey{Namespace: iris.Namespace, Name: iris.Spec.Dep
loymentName}, &deployment)
    if apierrors.IsNotFound(err) {
        log.Info("could not find existing Deployment for Iris, creating one...")

        deployment = *buildDeployment(iris)
        if err := r.Client.Create(ctx, &deployment); err != nil {
            log.Error(err, "failed to create Deployment resource")
            return ctrl.Result{}, err
        }

        r.Recorder.Eventf(&iris, core.EventTypeNormal, "Created", "Created deployment
```

```
 %q", deployment.Name)
        log.Info("created Deployment resource for Iris")
        return ctrl.Result{}, nil
    }
    if err != nil {
        log.Error(err, "failed to get Deployment for Iris resource")
        return ctrl.Result{}, err
    }

    log.Info("existing Deployment resource already exists for Iris, checking replica
count")

    expectedReplicas := int32(1)
    if iris.Spec.Replicas != nil {
        expectedReplicas = *iris.Spec.Replicas
    }

    if *deployment.Spec.Replicas != expectedReplicas {
        log.Info("updating replica count", "old_count", *deployment.Spec.Replicas, "n
ew_count", expectedReplicas)
        deployment.Spec.Replicas = &expectedReplicas
        if err := r.Client.Update(ctx, &deployment); err != nil {
            log.Error(err, "failed to Deployment update replica count")
            return ctrl.Result{}, err
        }

        r.Recorder.Eventf(&iris, core.EventTypeNormal, "Scaled", "Scaled deployment %
q to %d replicas", deployment.Name, expectedReplicas)

        return ctrl.Result{}, nil
    }

    log.Info("replica count up to date", "replica_count", *deployment.Spec.Replicas)
    log.Info("updating Iris resource status")

    iris.Status.ReadyReplicas = deployment.Status.ReadyReplicas
    if r.Client.Status().Update(ctx, &iris); err != nil {
        log.Error(err, "failed to update Iris status")
        return ctrl.Result{}, err
    }

    log.Info("resource status synced")
    return ctrl.Result{}, nil
}

// Delete the deployment resources that no longer match the iris.spec.deploymentName
field
func (r *IrisReconciler) cleanupOwnedResources(ctx context.Context, log logr.Logger,
iris *testv1alpha1.Iris) error {
    log.Info("looking for existing Deployments for Iris resource")

    var deployments apps.DeploymentList
    if err := r.List(ctx, &deployments, client.InNamespace(iris.Namespace), client.Ma
tchingField(deploymentOwnerKey, iris.Name)); err != nil {
        return err
    }
```

```
    deleted := 0
    for _, depl := range deployments.Items {
        if depl.Name == iris.Spec.DeploymentName {
            // Leave Deployment if its name matches the one in the Iris resource
            continue
        }

        if err := r.Client.Delete(ctx, &depl); err != nil {
            log.Error(err, "failed to delete Deployment resource")
            return err
        }

        r.Recorder.Eventf(iris, core.EventTypeNormal, "Deleted", "Deleted deployment
%q", depl.Name)
        deleted++
    }

    log.Info("finished cleaning up old Deployment resources", "number_deleted", delet
ed)
    return nil
}

func buildDeployment(iris testv1alpha1.Iris) *apps.Deployment {
    deployment := apps.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name:            iris.Spec.DeploymentName,
            Namespace:       iris.Namespace,
            OwnerReferences: []metav1.OwnerReference{*metav1.NewControllerRef(&iris,
testv1alpha1.GroupVersion.WithKind("Iris"))},
        },
        Spec: apps.DeploymentSpec{
            Replicas: iris.Spec.Replicas,
            Selector: &metav1.LabelSelector{
                MatchLabels: map[string]string{
                    "iris/deployment-name": iris.Spec.DeploymentName,
                },
            },
            Template: core.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: map[string]string{
                        "iris/deployment-name": iris.Spec.DeploymentName,
                    },
                },
                Spec: core.PodSpec{
                    Containers: []core.Container{
                        {
                            Name:  "iris",
                            Image: "
store/intersystems/iris-community:2020.4.0.524.0",
                        },
                    },
                },
            },
        },
    }
```

```
    return &deployment
}

var (
    deploymentOwnerKey = ".metadata.controller"
)

// Specifies how the controller is built to watch a CR and other resources
// that are owned and managed by that controller
func (r *IrisReconciler) SetupWithManager(mgr ctrl.Manager) error {
    if err := mgr.GetFieldIndexer().IndexField(&apps.Deployment{}, deploymentOwnerKey
, func(rawObj runtime.Object) []string {
        // grab the Deployment object, extract the owner...
        depl := rawObj.(*apps.Deployment)
        owner := metav1.GetControllerOf(depl)
        if owner == nil {
            return nil
        }
        // ...make sure it's an Iris...
        if owner.APIVersion != testv1alpha1.GroupVersion.String() || owner.Kind != "I
ris" {
            return nil
        }

        // ...and if so, return it
        return []string{owner.Name}
    }); err != nil {
        return err
    }

    return ctrl.NewControllerManagedBy(mgr).
        For(&testv1alpha1.Iris{}).
        Owns(&apps.Deployment{}).
        Complete(r)
}
```

To make the events logging work, we need to add yet another line to the main.go file:

```
if err = (&controllers.IrisReconciler{
                Client: mgr.GetClient(),
                Log:    ctrl.Log.WithName("controllers").WithName("Iris"),
                Scheme: mgr.GetScheme(),
                Recorder: mgr.GetEventRecorderFor("iris-controller"),
        }).SetupWithManager(mgr); err != nil {
```

Now everything is ready to set up an operator.
Let ' s install the CRD first using the Makefile target install:

```
$ cat Makefile
…
# Install CRDs into a cluster
install: manifests
        kustomize build config/crd | kubectl apply -f -
...
```

```
$ make install
```

You can have a look at the resulting CRD YAML file in the config/crd/bases/ directory.
Now check CRD existence in the cluster:

```
$ kubectl get crd
NAME                       CREATED AT
iris.test.myardyas.club    2020-11-17T11:02:02Z
```

Let ' s run our controller in another terminal, locally (not in Kubernetes) – just to see if it actually works:

```
$ make run
...
2020-11-17T13:02:35.649+0200 INFO controller-
runtime.metrics metrics server is starting to listen {"addr": ":8080"}
2020-11-17T13:02:35.650+0200 INFO setup starting manager
2020-11-17T13:02:35.651+0200 INFO controller-
runtime.manager starting metrics server {"path": "/metrics"}
2020-11-17T13:02:35.752+0200 INFO controller-
runtime.controller Starting EventSource
{"controller": "iris", "source": "kind source: /, Kind="}
2020-11-17T13:02:35.852+0200 INFO controller-
runtime.controller Starting EventSource
{"controller": "iris", "source": "kind source: /, Kind="}
2020-11-17T13:02:35.853+0200 INFO controller-
runtime.controller Starting Controller {"controller": "iris"}
2020-11-17T13:02:35.853+0200 INFO controller-
runtime.controller Starting workers {"controller": "iris", "worker count": 1}
…
```

Now that we have the CRD and the controller installed, all we need to do is create an instance of our custom resource. A template can be found in the config/samples/example.comv1alpha1iris.yaml file. In this file, we need to make changes similar to those in the crd-object.yaml:

```
$ cat config/samples/test_v1alpha1_iris.yaml
apiVersion: test.myardyas.club/v1alpha1
kind: Iris
metadata:
  name: iris
spec:
  deploymentName: iris
  replicas: 1
$ kubectl apply -f config/samples/test_v1alpha1_iris.yaml
```

After a brief delay caused by the need to pull an IRIS image, you should see the running IRIS pod:

```
$ kubectl get deploy
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
```

```
iris   1/1     1            1           119s
$ kubectl get pod
NAME                   READY   STATUS    RESTARTS   AGE
iris-6b78cbb67-vk2gq   1/1     Running   0          2m42s
$ kubectl logs -f -l iris/deployment-name=iris
```

You can open the IRIS portal using the kubectl port-forward command:

```
$ kubectl port-forward deploy/iris 52773
```

Go to http://localhost:52773/csp/sys/UtilHome.csp in your browser.
What if we change the replicas ' count in CRD? Let ' s make and apply this change:

```
$ vi config/samples/test_v1alpha1_iris.yaml
…
  replicas: 2
$ kubectl apply -f config/samples/test_v1alpha1_iris.yaml
```

You should now see another Iris pod appear.

```
$ kubectl get events
…
54s         Normal   Scaled                  iris/iris                  Scaled dep
loyment "iris" to 2 replicas
54s         Normal   ScalingReplicaSet       deployment/iris            Scaled up
replica set iris-6b78cbb67 to 2
```

Log messages in the terminal where the controller in running report successful reconciliation:

```
2020-11-17T13:09:04.102+0200 INFO controllers.Iris
replica count up to date
{"iris": "default/iris", "deployment_name": "iris", "replica_count": 2}
2020-11-17T13:09:04.102+0200 INFO controllers.Iris
updating Iris resource status {"iris": "default/iris", "deployment_name": "iris"}
2020-11-17T13:09:04.104+0200 INFO controllers.Iris
resource status synced {"iris": "default/iris", "deployment_name": "iris"}
2020-11-17T13:09:04.104+0200 DEBUG controller-
runtime.controller Successfully Reconciled
{"controller": "iris", "request": "default/iris"}
```

Okay, our controllers seem to be working. Now we ' re ready to deploy that controller inside Kubernetes as a pod. For that, we need to create the controller docker container and push it to the registry. This can be any registry that works with Kubernetes – DockerHub, ECR, GCR, and so on.
We ' ll use the local (kind) Kubernetes, so let ' s deploy the controller to the local registry using the kind-with-registry.sh script available from the Local Registry page. We can simply remove the current cluster and recreate it:

```
$ kind delete cluster
$ ./kind_with_registry.sh
```

```
$ make install
$ docker build . -t localhost:5000/iris-
operator:v0.1 # Dockerfile is autogenerated by kubebuilder
$ docker push localhost:5000/iris-operator:v0.1
$ make deploy IMG=localhost:5000/iris-operator:v0.1
```

The controller will be deployed into the IRIS-system namespace. Alternatively, you can scan all pods to find a namespace like kubectl get pod -A):

```
$ kubectl -n iris-system get po
NAME                                        READY   STATUS    RESTARTS   AGE
iris-controller-manager-bf9fd5855-kbklt   2/2     Running   0          54s
```

Let ' s check the logs:

```
$ kubectl -n iris-system logs -f -l control-plane=controller-manager -c manager
```

You can experiment with changing replicas ' count in the CRD and observe how these changes are reflected in the IRIS instances count.

Operator-SDK

Another handy tool to generate the operator code is Operator SDK. To get the initial idea of this tool, have a look at this tutorial. You should install operator-sdk first.
For our simple use case, the process will look similar to the one we ' ve worked on with kubebuilder (you can delete/create the kind cluster with the Docker registry before continuing). Run in another directory:

```
$ mkdir iris
$ cd iris
$ go mod init iris
$ operator-sdk init --domain=myardyas.club
$ operator-sdk create api --group=test --version=v1alpha1 --kind=Iris
# Answer two 'yes'
```

Now change the IrisSpec and IrisStatus structures in the same file – api/v1alpha1/iristypes.go. We ' ll use the same iriscontroller.go file as we did in kubebuilder. Don ' t forget to add the Recorder field in the main.go file.
Because kubebuilder and operator-sdk use different versions of the Golang packages, you should add a context in the SetupWithManager function in controllers/iriscontroller.go:

```
ctx := context.Background()
if err := mgr.GetFieldIndexer().IndexField
(ctx, &apps.Deployment{}, deploymentOwnerKey, func(rawObj runtime.Object) []string {
```

Then, install the CRD and the operator (make sure that the kind cluster is running):

```
$ make install
```

```
$ docker build . -t localhost:5000/iris-operator:v0.2
$ docker push localhost:5000/iris-operator:v0.2
$ make deploy IMG=localhost:5000/iris-operator:v0.2
```

You should now see the CRD, operator pod, and IRIS pod(s) similar to the ones we ' ve seen when we worked with kubebuilder.

Conclusion

Although a controller includes a lot of code, you ' ve seen that changing the IRIS replicas is just a matter of changing a line in a custom resource. All the complexity is hidden in the controller implementation. We ' ve looked at how a simple operator can be created using handy scaffolding tools.
Our operator cared only about IRIS replicas. Now imagine that we actually need to have the IRIS data persisted on disk – this would require StatefulSet and Persistent Volumes. Also, we would need a Service and, perhaps, Ingress for external access. We should be able to set the IRIS version and system password, Mirroring and/or ECP, and so on. You can imagine the amount of work InterSystems had to do to simplify IRIS deployment by hiding all the IRIS-specific logic inside operator code.
In the next article, we ' re going to look at IRIS Operator (IKO) in more detail and investigate its possibilities in more complex scenarios.

#DevOps #Kubernetes #InterSystems IRIS