The little book of Kubernetes Operators

Oz Tiram

Table of Contents

Introduction

Kubernetes is an open source platform, originally developed by Google, to schedule workloads on a cluster of computers. The most basic unit of work that kubernetes understands is a Pod, which is originally, one or more docker containers with dedicated amount of compute resources. These originally include CPU, RAM, network and disks. This unit of work is the source of viewing Kubernetes as a Container Scheduler. However, since the inception and birth of Kubernetes, work units have become more than just docker containers. In the meanwhile Kubernetes, can schedule a lot of things. For example LXC containers, Full blown virtual machines, lightweight virtual machines, and even FreeBSD Jails.

As such, we should view Kubernetes as a Workload Scheduler or as a distributed kernel, where we ask the system for compute resources to execute different tasks. Kubernetes then tries to fulfill our request by allocating compute resources and running our tasks.

This distributed kernel is composed of many controllers whose responsibility is fulfill a desired state and react to changes in the actual state of what ever resources they control. If you already work with Kubernetes, you are probably familiar with the built-in controllers `Deployment` and `StatefulSet`

The developers of kubernetes recognised the need for adding custom user controllers already in early versions of Kubernetes. Version 1.7 added the ability to define `ThirdPartyResource`, which allowed extending Kubernetes. These were later named `CustomResourceDefinition` in version 1.8 and onward.

Adding new resources to the kubernetes API opened the way for users writing their own controllers which watch these resources and perform actions based on changes in a `CustomResourceDefinition`.

Your first operator

We begin our journey for working with operators with a simple operator written in BASH. It does only one thing: Count the Deployments and Pods in the cluster. The operator Grafzahl [1] would output:

```
$ bash v1/main.sh
Deployments now 1, previously 0
Pods now 10, previously 0
Pods number increased! 👧 is 😄!
Deployments number increased! 👧 is 😄!
...
```

Note that the operator counts deployment and pods cluster in all namespaces. Hence, it starts the counting with numbers larger than 0.

In a different terminal we start a Pod:

```
$ kubectl run nginx --image docker.io/nginx
pod/nginx created
```

Now Grafzahl says:

```
Deployments now 1, previously 1
Pods now 11, previously 10
Pods number increased! 👧 is 😄!
```

When we delete the Pod:

```
$ kubectl delete pod nginx
pod "nginx" deleted
```

[1]
Graf Zahl is the German name for the Sesame Street character Count von Count.

Grafzahl should be sad:

Deep diving into building an operator

We begin with re-creating *grafzahl* using the pure go-client.

Basic Structure

To begin with a modern go project we create a layout for our CLI and package which is imported. We use go module to manage the dependencies:

```
$ cd grafzahl
$ mkdir -pv {cmd,pkg/operator}
$ go mod init gitlab.com/oz123/grafzahl
```

In `cmd/main.go`:

```
package main

import (
     "fmt"
     operator "gitlab.com/oz123/grafzahl/pkg/operator"
)

func main() {
     fmt.Println("Hello, Modules!")
     operator.PrintHello()
}
```

And in `pkg/operator/operator.go`:

```
package operator
import "fmt"

var VERSION = "0.0.0.dev"

func PrintHello() {
     fmt.Printf("Hello, Modules! This is Operator v%s speaking!\n", VERSION)
}
```

We also add a `Makefile` to help us manage the project:

```
DATE    = $(shell date +%Y%m%d%H%M)
VERSION = v$(DATE)
GOOS    ?= $(shell go env | grep GOOS | cut -d'"' -f2)
BINARY  := grafzahl

LDFLAGS := -X gitlab.com/oz123/grafzahl/pkg/operator.VERSION=$(VERSION)
GOFLAGS := -ldflags "$(LDFLAGS)"
PACKAGES := $(shell find $(SRCDIRS) -type d)
GOFILES  := $(addsuffix /*.go,$(PACKAGES))
GOFILES  := $(wildcard $(GOFILES))

.PHONY: all clean

all: bin/$(GOOS)/$(BINARY)

bin/%/$(BINARY): $(GOFILES) Makefile
     GOARCH=amd64 go build $(GOFLAGS) -v -o $(BINARY) cmd/main.go
```

Compile and run the program:

```
$ make
GOARCH=amd64 go build -ldflags "-X gitlab.com/oz123/grafzahl/pkg/operator.VERSION=6d31605 "
-v -o grafzahl cmd/main.go
command-line-arguments

$ ./grafzahl
Hello, Modules!
Hello, Modules! This is Operator v6d31605 speaking!
```

Note

The previously shown code is found in the v0.1 tag in the code repository.

Debugging your operator

After deploying the operator, you will, for sure, find some issues with it, which will require fixing. No software is born perfect.

Modifying the code, building a docker image, pushing it to a registry, and then deploying it to the cluster to watching the logs, only to repeat again a few moments later, is time consuming and frustrating.

You can run your code locally with a correct `KUBECONFIG` environment variable set. To do this, on Kubernetes versions 1.24 and later, you will have to create it explicitly, by create an explicit service account token, and then creating a KUBECONFIG file:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: grafzahl-sa-token
  namespace: grafzahl
  annotations:
    kubernetes.io/service-account.name: default
type: kubernetes.io/service-account-token
EOF
secret/grafzahl-sa-token created
```

Now, you can create a `KUBECONFIG` file:

```
NAMESPACE="grafzahl"
SECRET_NAME="grafzahl-sa-token"
KUBECONFIG_OUT="sa-kubeconfig.yaml"

APISERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')

TOKEN=$(kubectl get secret "${SECRET_NAME}" -n "${NAMESPACE}" -o jsonpath='{.data.token}' |
base64 -d)
kubectl get secret "${SECRET_NAME}" -n "${NAMESPACE}" -o jsonpath='{.data.ca\.crt}' |
base64 -d > ca.crt

CA_B64=$(base64 -w0 ca.crt 2>/dev/null || base64 ca.crt | tr -d '\n')

cat <<EOF > "${KUBECONFIG_OUT}"
apiVersion: v1
kind: Config
clusters:
- name: ${NAMESPACE}-cluster
  cluster:
    certificate-authority-data: ${CA_B64}
    server: ${APISERVER}
contexts:
- name: ${NAMESPACE}-context
  context:
    cluster: ${NAMESPACE}-cluster
    namespace: ${NAMESPACE}
    user: ${SECRET_NAME}-user
users:
- name: ${SECRET_NAME}-user
  user:
    token: ${TOKEN}
current-context: ${NAMESPACE}-context
EOF
```

Set the environment variable `KUBECONFIG` to point to this file, before running the operator:

```
export KUBECONFIG=$(pwd)/sa-kubeconfig.yaml
make all
./grafzahl
2025/07/17 10:07:28 Starting Grafzahl Operator v0.6.2-8-g0523ff9
2025/07/17 10:07:28 Starting Pod Watcher
...
```

This works, but there is a better alternative.

Debugging with devspace

`devspace` is like magic, and this section only scratches the surface. It allows you to create a container which syncs your local directory with a container running inside the cluster, where you can run your operator code directly or even run a debugger to which you can connect remotely.

To start with `devspace` first download the binary from the release page:

```
$ curl -L -o devspace "https://github.com/loft-sh/devspace/releases/latest/download/
devspace-linux-amd64" \
    && sudo install -c -m 0755 devspace /usr/local/bin && rm devspace-linux-amd64
```

Now, run the `devspace` command and configure it:

```
devspace init


     %########%
     %############%       ____                 _____
         %##########%    |  _ \   ___ __   __ / ___/  ____    ____   ____ ___
         %##########%    | | | | / _ \\ \ / / \___ \ |  _ \  / _  | / __// _ \
     %##############%    | |_| |(  __/ \ V /  ____) )| |_) )( (_| |( (__(  __/
     %##############%    |____/  \___|  \_/   \____/ |  __/  \__,_| \___\\___|
 %################%                                 |_|
 %############%


info Detecting programming language...

? Select the programming language of this project  [Use arrows to move, type to filter]
  c# (dotnet)
> go
  java-gradle
  java-maven
  javascript
  php
  python
```

The wizard is straight forward and in the end you should see:

```
$ devspace init

... snipped ...

info Detecting programming language...

? Select the programming language of this project go

? How do you want to deploy this project? kubectl

? Please enter the paths to your Kubernetes manifests (comma separated, glob patterns are
allowed, e.g. 'manifests/**' or 'kube/pod.yaml') [Enter to abort] k8s

? Do you want to develop this project with DevSpace or just deploy it?  [Use arrows to move,
type to filter] I want to develop this project and my current working dir contains the
source code

? Which image do you want to develop with DevSpace? docker.io/oz123/grafzahl:v0.6.2

? How should DevSpace build the container image for this project? Use this existing
Dockerfile: ./Dockerfile

? Which port is your application listening on? (Enter to skip)

done Project successfully initialized
info Configuration saved in devspace.yaml - you can make adjustments as needed

You can now run:
1. devspace use namespace - to pick which Kubernetes namespace to work in
2. devspace dev - to start developing your project in Kubernetes

Run `devspace -h` or `devspace [command] -h` to see a list of available commands and flags
```

Now, you can run the commands shown above. If your manifests have the namespace hardcoded in them, then you should choose the namespace hardcoded in them, in our case `grafzahl`. However, you might want to create a new namespace, and deploy your debug version inside this namespace, e.g. `grafzahl-dev`:

```
$ devspace use namespace grafzahl
info The default namespace of your current kube-context 'kvm2' has been updated to
'grafzahl'
...
done Successfully set default namespace to 'grafzahl'
```

Start the debugging session with:

```
$ devspace dev
info Using namespace 'grafzahl'
info Using kube context 'kvm2'
deploy:app Applying manifests with kubectl...
deploy:app clusterrole.rbac.authorization.k8s.io/grafzahl-rc configured
deploy:app clusterrolebinding.rbac.authorization.k8s.io/grafzahl unchanged
deploy:app customresourcedefinition.apiextensions.k8s.io/numbers.grafzahl.io unchanged
deploy:app deployment.apps/grafzahl-operator unchanged
deploy:app Warning: resource numbers/total is missing the kubectl.kubernetes.io/last-
applied-configuration annotation which is required by kubectl apply. kubectl apply should
only be used on resources created declaratively by either kubectl create --save-config or
kubectl apply. The missing annotation will be patched automatically.
deploy:app number.grafzahl.io/total configured
deploy:app clusterrole.rbac.authorization.k8s.io/grafzahl-rc configured
deploy:app clusterrolebinding.rbac.authorization.k8s.io/grafzahl unchanged
deploy:app Successfully deployed app with kubectl
dev:app Waiting for pod to become ready...
dev:app Selected pod grafzahl-operator-devspace-5b8d9d95cf-mml8h
dev:app ports Port forwarding started on: 2345 -> 2345
dev:app sync  Sync started on: ./ <-> ./
dev:app sync  Waiting for initial sync to complete
dev:app sync  Initial sync completed
dev:app ssh   Port forwarding started on: 11888 -> 8022
dev:app proxy Port forwarding started on: 11473 <- 10567
dev:app ssh   Use 'ssh app.grafzahl.devspace' to connect via SSH
dev:app term  Opening shell to main:grafzahl-operator-devspace-5b8d9d95cf-mml8h
(pod:container)

     %########%
     %############%       ____                  _____
         %##########%    |  _ \   ___ __    __ / ___/  ____    ____   ____ ___
         %##########%    | | | | / _ \\ \ / / \___ \ |  _ \  / _  | / __// _ \
     %##############%    | |_| |(  __/ \ V /  ____) )| |_) )( (_| |( (__(  __/
     %##############%    |____/  \___|  \_/   \____/ |  __/  \__,_| \___\\___|
 %################%                                  |_|
 %############%


Welcome to your development container!

This is how you can work with it:
- Files will be synchronized between your local machine and this container
- Some ports will be forwarded, so you can access this container via localhost
- Run `go run main.go` to start the application
```

Run your code inside the `devspace` Pod:

```
devspace ./app # go run cmd/main.go
go: downloading k8s.io/api v0.23.1
go: downloading k8s.io/apimachinery v0.23.1
go: downloading k8s.io/client-go v0.23.1
go: downloading github.com/gogo/protobuf v1.3.2
go: downloading github.com/google/gofuzz v1.1.0
...
2025/07/16 13:55:07 Starting Pod Watcher
2025/07/16 13:55:07 Successfully add handlers to Pod Watcher
2025/07/16 13:55:07 Starting Deployment Watcher
2025/07/16 13:55:07 Added all handlers to Deployment Watcher
2025/07/16 13:55:07 Starting Events Watcher
2025/07/16 13:55:07 Added all handlers to Events Watcher
2025/07/16 13:55:07 Grafzahl 0.0.0.dev is ready to count
There are 12 pods in the cluster
There are 4 deployments in the cluster
2025/07/16 13:55:07 Initialized total deployments: 4, pods: 12
2025/07/16 13:55:07 Deployments: 4, Pods: 12
```

In a different window you can edit the code, and the changes in your code, will immediately sync to the `devspace` container where you can run the code without having to run:

```
$ make build
$ make push
$ kubectl set image -n grafzahl main=docker.io/oz123:v<newversion>
$ k logs -n grafzahl -l app=grafzahl
```