



Waterford Institute of Technology

CLOUD COMPUTING 1

BACHELOR OF SCIENCE (HONS) APPLIED COMPUTING

CoreOS Operators

Ciaran ROCHE - 20037160

April 16, 2019

Plagiarism Declaration

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

Contents

1	Prerequisites	4
2	Introduction	5
2.1	etcd	5
2.2	Create/Destroy	6
2.3	Resize	7
2.4	Recover	8
2.5	etcd Operator Summary	11
3	Kubernetes	12
3.1	Overview	12
3.2	Custom Resources	12
3.3	ReplicaSet	12
3.4	Kubernetes Objects	13
3.5	Role-Based Access Control	13
4	CoreOS Operators	14
4.1	Operator Framework	14
4.2	Operator SDK	14
4.3	Operator Lifecycle Manager	15
4.4	Operator Metering	15
5	Building an Operator	17
5.1	Install Operator SDK	17
5.2	Memcached Operator	17
5.3	Initialize Operator	17
5.4	Adding Custom Resource Definition	18
5.5	Adding a Controller	18
5.6	Operator Layout	19
5.7	Manager	20
5.8	Custom Resource Definition	20
5.9	Controller	21
5.10	Building the Operator	22
5.11	Updating the Memcache Operator	25
5.12	Running the updated Memcache Operator	29
5.13	Memcache Operator Summary	30

6 Operator Lifecycle Manager	30
7 Summary	33
Appendices	34
A CoreOS etcd Operator	34
B Operator SDK	34
C Memcached Operator Example	34
D Memcached Operator Controller	34
E Operator Demo	34
F Full Operator Demo Youtube	34
Bibliography	35

1 Prerequisites

A number of prerequisites are needed in order to complete or follow any examples outlined in this paper.

- Dep version v0.5.0+.
- Git.
- GoLang version v1.10+.
- Docker version 17.03+.
- Kubectl version v1.11.0+.
- Kubernetes v1.11.0+ cluster. (Minikube)

2 Introduction

This report looks to answer the question, "What is a Kubernetes Operator?". To help answer this question we will look at an Operator in use throughout this introduction. Gaining the hands on experience of an Operator working now will aid in further understanding and explanations over the course of this paper.

2.1 etcd

etcd is a distributed key-value store, in fact the primary key-value store for Kubernetes. Its job is the storing and replicating of all Kubernetes cluster state. It is a critical component of a Kubernetes cluster, as it deals with maintaining quorum, cluster reconfiguration, creation of backups, disaster recovery etc. Normally to implement this kind of work requires a specific expertise. Through the use of an etcd Operator this work is now easier as the Operator handles lots of the heavy lifting which was once carried out by a sys admin (*CoreOS Blog, 2018a*). I feel this is summarized nicely in Figure 1 where it is defined that "An Operator represents human operational knowledge in software, to reliably manage an application."

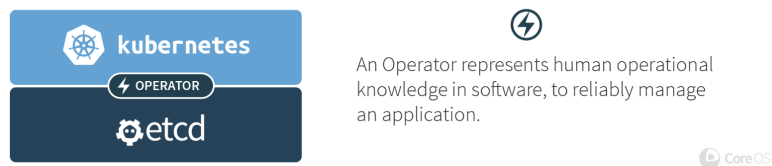


Figure 1: *Image from CoreOS*

The elements of the etcd Operator are as follows:

- Create/Destroy
- Resize
- Recover
- Backup
- Upgrade

Before running an example of the above mentioned elements, we need to gain some understanding into how the operator simulates human behaviors. It follows the common Operator pattern in that it watches for changes and then acts. CoreOS etcd summarizes it as three steps, Observe, Analyze

and Act. It observes the current state of the cluster through leveraging the Kubernetes API, It then analyzes the current state and if it notices a difference from the desired state it then acts to fix the difference and return the cluster to the desired state. With a high level view of how the etcd operator behaves lets dive into some practical examples.

2.2 Create/Destroy

To setup first I cloned the CoreOS etcd Operator onto my GOPATH, the github repo can be found in Appendix A. With the operator cloned I opted to take advantage of the included example. First was to deploy the operator by running the example deployment, this can be seen in Figure 2.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)  
[master [origin/master ]]  
→ kubectl create -f ./example/deployment.yaml  
deployment.extensions/etcd-operator created
```

Figure 2: Operator Create Command

With the operator deployed, we included a desired state of 3 replicas. This can be seen in Figure 3 showing we have 3 example pods running.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)  
[master [origin/master ]]  
→ kubectl get pods  
NAME                                READY    STATUS    RESTARTS   AGE  
etcd-operator-69b559656f-6wgdm      1/1      Running   0           1m  
example-etcd-cluster-nb7x82dc7d     1/1      Running   0           1m  
example-etcd-cluster-nw282jkvzt     1/1      Running   0           1m  
example-etcd-cluster-nzmhxk75h5     1/1      Running   0           1m
```

Figure 3: kubectl get pods after create

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✗)  
[master [origin/master ]]  
→ kubectl delete -f example/example-etcd-cluster.yaml  
etcdcluster.etcd.database.coreos.com "example-etcd-cluster" deleted
```

Figure 4: Operator Delete Command

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ])
→ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-operator-69b559656f-6wgdm	1/1	Running	0	3m
example-etcd-cluster-nb7x82dc7d	0/1	Terminating	0	3m
example-etcd-cluster-nw282jkvzt	1/1	Terminating	0	2m
example-etcd-cluster-nzmhxx75h5	0/1	Terminating	0	2m

Figure 5: *kubectl get pods after delete*

Due to Operators extending the Kubernetes API we can run as normal a `kubectl delete` command seen in Figure 4 to delete our pods. After issuing the command we can see in Figure 5 our pods being terminated.

2.3 Resize

Since we deleted our pods in the previous steps we can simply update the operator by running the command seen in Figure 6. The command will update the operator and set our desired state to 3 replica sets. This can be seen in Figure 7.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ])
→ kubectl apply -f example/example-etcd-cluster.yaml
etcdcluster.etcd.database.coreos.com/example-etcd-cluster created
```

Figure 6: *Apply Operator*

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (x)
(master [origin/master ])
→ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-operator-69b559656f-6wgdm	1/1	Running	0	6m
example-etcd-cluster-f7nsqdrqn	1/1	Running	0	1m
example-etcd-cluster-gjbdxfjdlkh	1/1	Running	0	1m
example-etcd-cluster-hzlpblvfj	1/1	Running	0	47s

Figure 7: *kubectl get pods after update*

If we refer to Figure 9 we see the config for our custom resource definition. If we update the size to 5, telling our operator we want a desired state of 5 pods in our cluster. With the CRD updated we run the `apply` command as seen in 8 to update the operator. Also seen in Figure 8 is an added pod as the operator begins to act on the current state of the cluster based on the recent update.


```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ])
> kubectl apply -f example/example-etcd-cluster.yaml
etcdcluster.etcd.database.coreos.com/example-etcd-cluster configured
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-operator-69b559656f-6wgdh	1/1	Running	0	7m
example-etcd-cluster-f7nsqdrqn	1/1	Running	0	2m
example-etcd-cluster-gjbdfjdlkh	1/1	Running	0	2m
example-etcd-cluster-hzlzpblvfj	1/1	Running	0	1m
example-etcd-cluster-z7tflz5jsz	1/1	Running	0	15s

Figure 8: *apply update and get pods*

```
example-etcd-cluster.yaml — cloud-midterm
Explorer (⌘E) ble-etcd-cluster.yaml x
1 apiVersion: "etcd.database.coreos.com/v1beta2"
2 kind: "EtcdCluster"
3 metadata:
4   name: "example-etcd-cluster"
5   ## Adding this annotation make this cluster managed by clusterwide operators
6   ## namespaces operators ignore it
7   # annotations:
8   #   etcd.database.coreos.com/scope: clusterwide
9 spec:
10  size: 5
11  version: "3.2.13"
12
```

Figure 9: *etcd custom resource definition*

2.4 Recover

So far the operations we have demonstrated could also be easily taken care of by the Kubernetes API, the recover element to the Operator will show some of the power behind an Operator and the added value it brings to deployments.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
> kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-operator-69b559656f-6wgdh	1/1	Running	0	1h
example-etcd-cluster-ck9t6h6mgh	1/1	Running	0	13s
example-etcd-cluster-gjbdfjdlkh	1/1	Running	0	1h
example-etcd-cluster-h4rg6rb8s6	1/1	Running	0	29s
example-etcd-cluster-lf2z8f6vq5	1/1	Running	0	1h

Figure 10: *get pods*

If we look at Figure 10 we can see that we currently have 5 example pods running within our cluster. We are going to simulate a failure but deleting one of these pods and watch out the operator acts to restore the state of the cluster.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
➤ kubectl delete pod example-etcd-cluster-dtjccx7ql6 --now
pod "example-etcd-cluster-dtjccx7ql6" deleted
```

Figure 11: *delete a pod*

As we can see in Figure 11, we delete one of our running pods, on running the get pods command after the successful deletion we can see in Figure 12 a new pod has be initialized and the desired state has been restored to our cluster.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✗)
(master [origin/master ] 1)
➤ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-operator-69b559656f-6wgdm	1/1	Running	0	1h
example-etcd-cluster-ck9t6h6mgh	1/1	Running	0	5m
example-etcd-cluster-gjbdfjdlkh	1/1	Running	0	1h
example-etcd-cluster-h4rg6rb8s6	1/1	Running	0	6m
example-etcd-cluster-lf2z8f6vq5	1/1	Running	0	1h
example-etcd-cluster-q7tb55d74f	1/1	Running	0	8s

Figure 12: *new cluster state*

We have now seen how the etcd operator manages the state of the cluster if a pod fails, the following steps shows how an operator recovers if it itself fails. For this we are going to simulate an operator failing and also an example pod failing. Note I have configured an alias for *kubectl*.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
➤ kc delete -f example/deployment.yaml
deployment.extensions "etcd-operator" deleted
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
➤ kc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-operator-69b559656f-6wgdm	0/1	Terminating	0	1h
example-etcd-cluster-ck9t6h6mgh	1/1	Running	0	22m
example-etcd-cluster-gjbdfjdlkh	1/1	Running	0	1h
example-etcd-cluster-h4rg6rb8s6	1/1	Running	0	22m
example-etcd-cluster-lf2z8f6vq5	1/1	Running	0	1h
example-etcd-cluster-q7tb55d74f	1/1	Running	0	16m

Figure 13: *Delete operator*

As can be seen in Figure 13 we have terminated the operator within our cluster. Figure 14 shows the deletion of a pod from our cluster

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
➔ kc delete pod example-etcd-cluster-ck9t6h6mgh --now
pod "example-etcd-cluster-ck9t6h6mgh" deleted
^C
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✗)
(master [origin/master ] 1)
➔ kc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
example-etcd-cluster-ck9t6h6mgh	0/1	Terminating	0	23m
example-etcd-cluster-gjbdfjdlkh	1/1	Running	0	1h
example-etcd-cluster-h4rg6rb8s6	1/1	Running	0	23m
example-etcd-cluster-lf2z8f6vq5	1/1	Running	0	1h
example-etcd-cluster-q7tb55d74f	1/1	Running	0	17m

Figure 14: Delete a pod

With both an operator and a pod deleted from our cluster Figure 15 shows the operator being redeployed to our cluster and if we look at Figure 16 we can see the operator running within our cluster and it has observed the current state not being what is desired and has acted to initialize a new pod to restore our cluster to its defined state.

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
➔ kc create -f example/deployment.yaml
deployment.extensions/etcd-operator created
```

Figure 15: Launch Operator

```
(~/Projects/go/src/github.com/ciaranRoche/cloud-midterm/etcd-operator) (✓)
(master [origin/master ] 1)
➔ kc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
etcd-operator-69b559656f-jqtkj	1/1	Running	0	37s
example-etcd-cluster-gjbdfjdlkh	1/1	Running	0	1h
example-etcd-cluster-h4rg6rb8s6	1/1	Running	0	24m
example-etcd-cluster-lf2z8f6vq5	1/1	Running	0	1h
example-etcd-cluster-q7tb55d74f	1/1	Running	0	18m
example-etcd-cluster-rld22j6c5l	0/1	Init:0/1	0	2s

Figure 16: new cluster state

2.5 etcd Operator Summary

We have looked at some of the basic operations of an etcd Operator to try gain an understanding of what is an Operator and the purpose of it. Through the introductory examples it should lead to a greater understand of the theory behind Operators explained in Section 4. If we reference Figure 17 to show the logic behind the etcd Operator and remember the actions, observe, analyze and act as we will delve into this logic in detail.

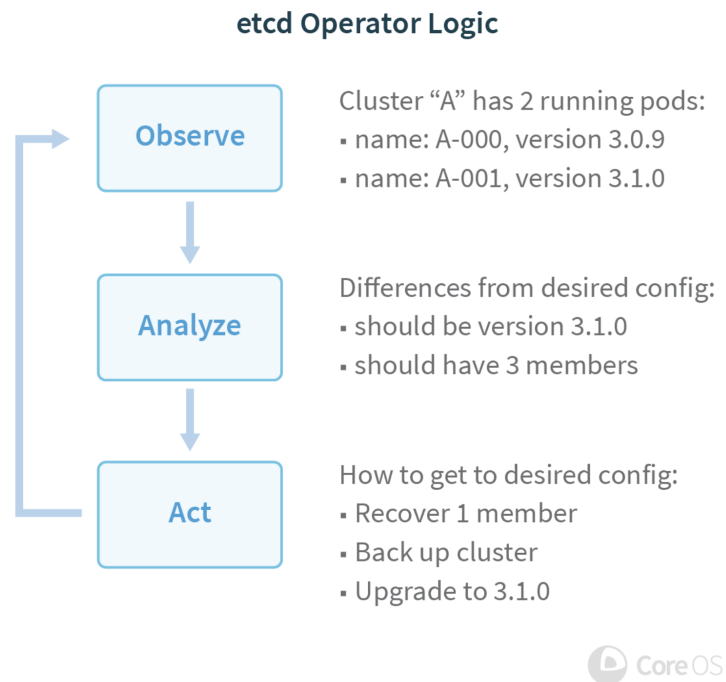


Figure 17: *etcd Operator Logic*

3 Kubernetes

An understanding of Kubernetes is needed and some of the components to which Operators build on top of or extend before we continue. Kubernetes is an open-source platform for managing containerized workloads and services. It provides a container-centric management environment that orchestrates computing, networking and storage infrastructure on behalf of a user. To help put it in perspective we can think of Kubernetes as a Platform as a Service with the flexibility of Infrastructure as a Service (*Concepts*, 2018).

Kubernetes uses a number of abstractions to represent a cluster, some of which are important to our understanding of Operators and will be mentioned throughout this paper.

3.1 Overview

To work with Kubernetes we use the Kubernetes API objects to set our desired state. A desired state consists of what applications we want to run, what container images, the number of replicas, what network and disk resources we want available etc. Normally we do this through a command-line interface, `kubectl`. Through the `kubectl` we can interact with the cluster directly, setting or modifying the desired state (*Concepts*, 2018).

Kubernetes contains a number of abstractions that represent the state of the system, these abstractions are represented by objects. It is some of these objects which Operators are built atop of or extend.

3.2 Custom Resources

A resource is an endpoint in the Kubernetes API that stores a set of API objects. For example the pods resource contains a set of Pod objects. A custom resource on the other hand is one that is necessarily not available to every cluster in that it represents a customization of a particular installation. Once a custom resource is installed we can create and access its objects through the `kubectl`, same as how we interact with a regular resource (*Custom Resources*, 2018).

3.3 ReplicaSet

A controller ensures that a specified pod replicas are running at any one time. A ReplicaSet is classed as a Replication Controller, these can be used independently but they are mainly used by Deployments as a mechanism to orchestrate pod creation, deletion and updates (*Controllers*, 2018).

3.4 Kubernetes Objects

For brevity we have left out a lot of Kubernetes Objects and only mentioned some that need a clearer understanding when it comes to Operators. Basic Kubernetes objects include:

- Pod
- Service
- Volume
- Namespace

Atop of basic object Kubernetes has a number of high-level abstraction called Controllers, these include:

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job

3.5 Role-Based Access Control

A summary and a high level understanding of RBAC is needed ahead of [Section 6](#), where we look at the Operator Lifecycle Manager, RBAC is mentioned.

RBAC is a method for regulating access to resources based on the roles of users. Throughout the practical elements we will be deploying roles and role bindings. The roles will connect API resources to operations, such as create, update, delete etc. The role binding then connects the roles to the users.

4 CoreOS Operators

From the example of an Operator working in Section 2.1 we can conclude that an Operator is a process for packaging, deploying and managing Kubernetes applications. To define what a Kubernetes application is is an application that is deployed on Kubernetes but also managed using the Kubernetes APIs and the Kubectl tooling (*CoreOS*, 2018).

Typically a Site Reliability Engineer (SRE) is one who manages an application by writing software. And also has the know how to develop software for a specific application domain. This piece of software now has operational domain knowledge built into it. Taken from this perspective we can look at Operators as an implementation of this concept, a piece of software that can create, configure and manage Kubernetes applications.

It does this by extension of the Kubernetes API and builds atop of the basic Kubernetes resource and controller concepts, but infuses domain and or application-specific knowledge to automate common tasks (*CoreOS Blog*, 2016)

4.1 Operator Framework

In order to make the development of Operators and the management of Kubernetes applications easier, Red Hat and the Kubernetes open source community created the Operator Framework. The framework is a toolkit designed to manage Kubernetes native applications in an automated and elastic way (*CoreOS Blog*, 2018b). The following sections looks at some of the tool kits within the Operator Framework.

4.2 Operator SDK

Creating Operators can be difficult due to the complexities of using low level APIs, writing your own boilerplate and the overall lack of modularity which often leads to duplication. With this in mind the Operator Framework added an Operator SDK toolkit. The SDK provides the tools to build, test and package Operators. The best practises and patterns are included in the SDK to help prevent reinventing the wheel, Figure 18 illustrates the development cycle for working with the SDK (*operator framework*, 2018c).

Typically Operators are developed in Go but can also be developed in Ansible the following outlines the workflow for a Go Operator:

- Use the SDK CLI to create a new operator project
- Define new resource APIs through a Custom Resource Definition (CRD)

- Define Controllers to watch resources
- Write reconciling logic for the Controller by utilizing the SDK and controller-runtime APIs
- Use the SDK CLI to build and generate the operator deployment manifest

This workflow will be demonstrated in Section 5.

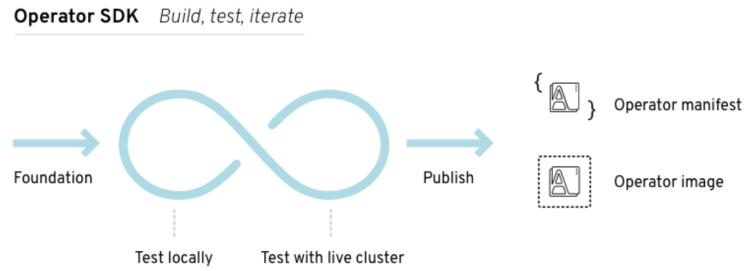


Figure 18: *Operator SDK workflow (Image from CoreOS)*

4.3 Operator Lifecycle Manager

Built operators need to be deployed on a Kubernetes cluster. It is the Operator Lifecycle Manager (OLM) facilitates the management of Operators on a Kubernetes cluster. True the Lifecycle Manager administrators are able to control what Operators are available in what namespaces and who is able to interact with them. It also gives the capabilities to trigger updates to resources and the Operator itself ([operator framework, 2018a](#)).

To put differently if we think of an Operator of a mechanism to manage our pods, how do we manage our Operator, does it require its own Operator? Well the Operator Lifecycle Manager extends Kubernetes to provide a declarative way to install, manage and upgrade operators and their dependencies within a cluster. As of writing this report the OLM currently supports the definition of applications as a single Kubernetes resource and encapsulates these requirements and the metadata. It allows for the install of applications automatically with dependency resolution, and finally it can upgrade applications automatically with different approval policies.

4.4 Operator Metering

As of writing this report, Operator Metering is not released yet and plans to be released in the coming months.

It will enable usage reporting for Operators that provide specialized services. It will do this through the recording of historical cluster usage, from these records it will generate reports showing usage breakdowns by pod or namespace over time periods. The overall project aim is to tie into the

Operator Lifecycle Manager *Install & update across clusters*

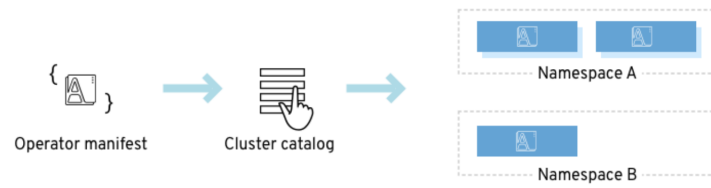


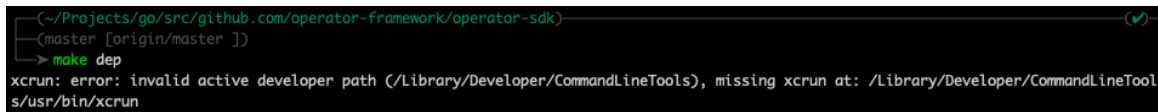
Figure 19: *Operator SDK Lifecycle Manager (Image from CoreOS)*

cluster CPU and memory reporting, this will allow for a calculation of Infrastructure as a Service costs ([operator framework, 2018b](#)).

5 Building an Operator

5.1 Install Operator SDK

To begin building an Operator it was decided to use the Operator SDK. In order to use the Operator SDK, GO is needed to be installed, and the operator-framework to be added to the GOPATH. Once added clone the Operator SDK, link can be found in Appendix B. With the SDK cloned it is a matter of running *make dep* and *make install*. Just to note after a recent update to osMojave it caused an error with xCode Command Line Tools, the error thrown can be seen in Figure 20. After a number of trouble shooting steps, to fix this error xcode-select needed to be reset, *xcode-select -reset*.

A terminal window with a dark background. The prompt is '~/.Projects/go/src/github.com/operator-framework/operator-sdk'. Below the prompt, it shows '(master [origin/master])' and then the command 'make dep' is entered. The output is an error message: 'xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun'.

```
~/Projects/go/src/github.com/operator-framework/operator-sdk (master [origin/master])  
➤ make dep  
xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun
```

Figure 20: Xcode error

5.2 Memcached Operator

For this section of the report we are going to work with the example Memcached Operator supplied with the Operator SDK in order to get a better understanding of what is happening under the hood of an Operator.

Memcached is a general-purpose distributed memory caching system. It is often used to speed up dynamic database-driven websites by caching data and objects in RAM. Memcached is free and open sourced (Dormando, n.d.). The operator built for this practical can be found at Appendix C.

5.3 Initialize Operator

With the error fixed the dependencies could be added and the Operator SDK installed. With that it can be seen in Figure 21, creating a new operator, it should be noted following with Go practises the operator was created in *\$GOPATH/src/github.com/ciaranRoche/* directory.

A terminal window with a dark background. The prompt is '~/.Projects/go/src/github.com/ciaranRoche'. The command 'operator-sdk new app-operator' is entered. The output shows a series of file creation messages: 'Create cmd/manager/main.go', 'Create build/Dockerfile', 'Create deploy/service_account.yaml', 'Create deploy/role.yaml', 'Create deploy/role_binding.yaml', and 'Create deploy/operator.yaml'.

```
~/Projects/go/src/github.com/ciaranRoche  
➤ operator-sdk new app-operator  
Create cmd/manager/main.go  
Create build/Dockerfile  
Create deploy/service_account.yaml  
Create deploy/role.yaml  
Create deploy/role_binding.yaml  
Create deploy/operator.yaml
```

Figure 21: Create a new Operator

5.4 Adding Custom Resource Definition

With a basic Operator created we will need to add a Custom Resource Definition. This can be seen in Figure 22. We call this Memcached and add it with an APIVersion of *cache.ciaranroche.com/v1alpha1*.



```
(~/Projects/go/src/github.com/ciaranRoche)
→ cd app-operator
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master)
→ operator-sdk add api --api-version=cache.ciaranroche.com/v1alpha1 --kind=Memcached
Create pkg/apis/cache/v1alpha1/memcached_types.go
Create pkg/apis/addtoscheme_cache_v1alpha1.go
Create pkg/apis/cache/v1alpha1/register.go
Create pkg/apis/cache/v1alpha1/doc.go
Create deploy/crds/cache_v1alpha1_memcached_cr.yaml
Create deploy/crds/cache_v1alpha1_memcached_crd.yaml
Running code-generation for custom resource group versions: [cache:v1alpha1, ]
Generating deepcopy funcs
```

Figure 22: Add Operator API

5.5 Adding a Controller

A new Controller is added and can be seen in Figure 23, the Controller will watch and reconcile the Memcached resource. This will be discussed further in Section 5.9.



```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 1 3)
→ operator-sdk add controller --api-version=cache.ciaranroche.com/v1alpha1 --kind=Memcached
Create pkg/controller/memcached/memcached_controller.go
Create pkg/controller/add_memcached.go
```

Figure 23: Add Operator Controller

5.6 Operator Layout

Figure 24 shows the folder structure to the Operator that has been created through the Operator SDK. As can be seen from the structure the amount of boilerplate created shows the value of the Operator framework when it comes to creating an Operator.



```
(~/Projects/go/src/github.com/ciaranroche/app-operator)
(master 1 5)
> tree -L 2
.
├── Gopkg.lock
├── Gopkg.toml
├── build
│   └── Dockerfile
├── cmd
│   └── manager
├── deploy
│   ├── crds
│   ├── operator.yaml
│   ├── role.yaml
│   ├── role_binding.yaml
│   └── service_account.yaml
├── pkg
│   ├── apis
│   └── controller
├── vendor
│   ├── cloud.google.com
│   ├── github.com
│   ├── go.uber.org
│   ├── golang.org
│   ├── google.golang.org
│   ├── gopkg.in
│   ├── k8s.io
│   └── sigs.k8s.io
└── version
    └── version.go

18 directories, 8 files
```

Figure 24: *Operator Tree*

The breakdown of Figure 24 is as follows:

- **Gopkg.lock Gopkg.lock** : The Go files that outline any external dependencies for the Operator
- **build** : Contains a Dockerfile which is used to build and dockerize the operator
- **cmd** : Contains the manager which is the entry point to initialize and start the operator using the operator-sdk API
- **deploy** : Contains the Kubernetes manifests for the deployment of the Operator on a Kubernetes cluster.
- **pkg** : This defines the APIs and Custom Resource Definitions.
- **vendor** : A GoLang folder that holds local copies to dependencies for the operator
- **version** : Contains version data of the Operator

5.7 Manager

The Operator main program within the `cmd/manager/` directory, initializes and runs the Manager. Its the Managers job to automatically register the scheme set for all custom resources that are defined under the Operator, and also run all controllers. The manager can restrict the namespaces this can be seen in Figure 26

```
// Setup Scheme for all resources
if err := apis.AddToScheme(mgr.GetScheme()); err != nil {
|   log.Fatal(err)
}

// Setup all Controllers
if err := controller.AddToManager(mgr); err != nil {
|   log.Fatal(err)
}
```

Figure 25: *Setting CRD's and Controllers*

```
// Create a new Cmd to provide shared dependencies and start components
mgr, err := manager.New(cfg, manager.Options{Namespace: namespace})
if err != nil {
|   log.Fatal(err)
}
```

Figure 26: *Manager function to restrict the namespace*

5.8 Custom Resource Definition

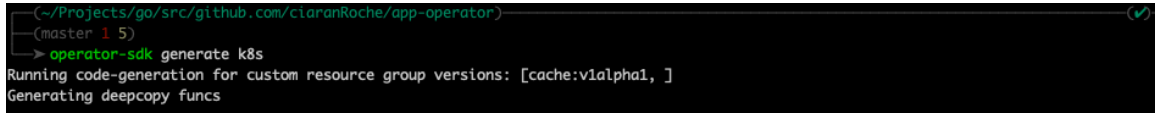
From the boilerplate of the CRD that was generated by the command in Figure 22 we need to define our own Custom Resource, this can be seen in Figure 27, where we define the size of the Memcached deployment along with the nodes being the name of the Memcached pods.

```
// MemcachedSpec defines the desired state of Memcached
type MemcachedSpec struct {
|   Size int32 `json:"size"`
}

// MemcachedStatus defines the observed state of Memcached
type MemcachedStatus struct {
|   Nodes []string `json:"nodes"`
}
```

Figure 27: *Memcache CRD*

It should be noted that it is needed to generate the code for the resource which we just defined that can be done by executing the *generates k8s* command, which can be seen in Figure 28



```
(~/Projects/go/src/github.com/ciaranroche/app-operator)
(master 1 5)
➔ operator-sdk generate k8s
Running code-generation for custom resource group versions: [cache:v1alpha1, ]
Generating deepcopy funcs
```

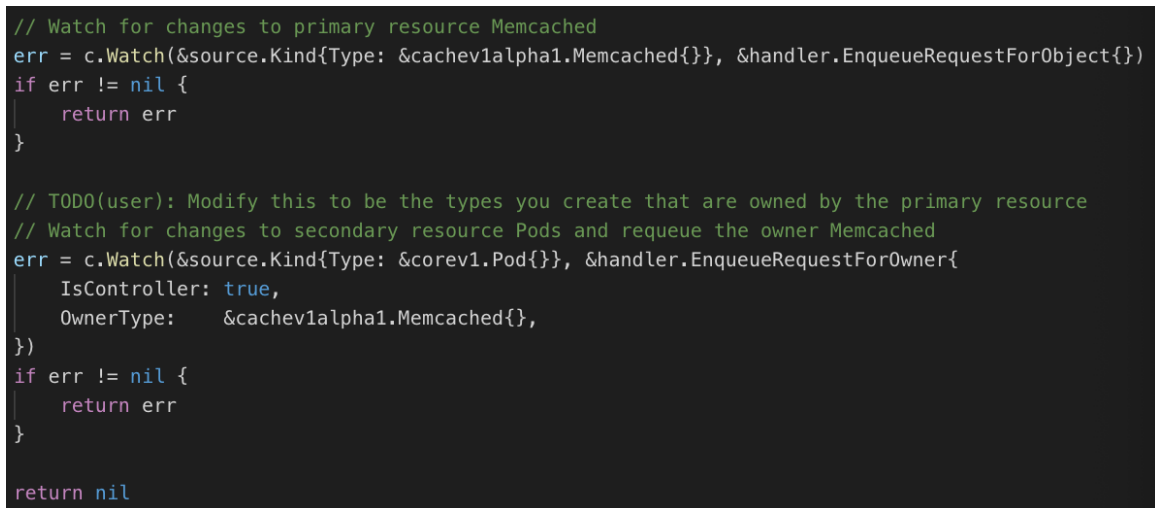
Figure 28: *Generate k8s*

5.9 Controller

A powerful aspect to operators is their *watch* function. In that is watches for events to happen and on the event happening the watch will trigger and act.

When we added the controller to our operator in Section 5.5 it scaffold a new Controller implementation for us in the *pkg/controller/memcached* directory. From this implementation we can see how the Controller watches resources and how it triggers the reconcile loop. If we refer to Figure 29 we can see two watch functions that were generated for us. The first watch function is for the Memcached type as the primary resource. For every event the reconcile loop is sent a reconcile request for the Memcached object. A reconcile request is made up of a namespace/name key.

If we look at the second watch function this is for Deployments. The event handler here maps each event to a reconcile request. Which in Figure 29 is the Memcached object for which the Deployment was created. This allows the controller to watch Deployments as a secondary resource.



```
// 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: &corev1.Pod{}}, &handler.EnqueueRequestForOwner{
    IsController: true,
    OwnerType:    &cachev1alpha1.Memcached{},
})
if err != nil {
    return err
}

return nil
```

Figure 29: *Controller watch functions.*

Every Controller has a Reconciler object that contains a reconcile function which implements the

reconcile loop. The loop is passed the reconcile request which I mention earlier contains a Namespace/Name Key, this is used to lookup the primary resource object. Figure 30 shows the reconcile function which was generated for us. Based on the result from this function the request may be requeued and then the loop triggered again.

```
func (r *ReconcileMemcached) Reconcile(request reconcile.Request) (reconcile.Result, error) {
    log.Printf("Reconciling Memcached %s/%s\n", request.Namespace, request.Name)

    // Fetch the Memcached instance
    instance := &cachev1alpha1.Memcached{}
    err := r.client.Get(context.TODO(), request.NamespacedName, instance)
    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
            return reconcile.Result{}, nil
        }
        // Error reading the object - requeue the request.
        return reconcile.Result{}, err
    }
}
```

Figure 30: *Controller reconcile function.*

5.10 Building the Operator

Before building and running the Operator we need to register our Custom Resource Definition with the Kubernetes apiserver. This can be seen in Figure 31. It should be noted when I first ran this command, I was running on Minishift, which is a tool to allow you to run an OKD cluster locally. OKD is the community edition of OpenShift. OpenShift is classed as enterprise Kubernetes. Due to the way OKD hands users I was unable to create a CRD and need to log in as an admin. To avoid running into any other problems, I shutted down Minishift and started up Minikube and ran Kubernetes locally through that.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master)
➔ kubectl create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.ciaranroche.com created
```

Figure 31: *Registering CRD*

Once we have registered the CRD, Figure 32 and 33 shows how to build an operator image using the operator sdk. As can be seen we then use Docker to push the image to Quay.io

It should be noted here to check your Quay.io repository to ensure the image is set to public visibility to avoid and unwanted behaviour later on.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master)
➔ operator-sdk build quay.io/ciaranroche/memcached-operator:v0.0.1

Sending build context to Docker daemon 140.1MB
Step 1/3 : FROM alpine:3.6
----> 94627dfbdf19
Step 2/3 : USER nobody
----> Using cache
----> 7b50e0239ca8
Step 3/3 : ADD build/_output/bin/app-operator /usr/local/bin/app-operator
----> 0f8271baf7a2
Successfully built 0f8271baf7a2
Successfully tagged quay.io/ciaranroche/memcached-operator:v0.0.1
```

Figure 32: *Building Operator Image*

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master)
➔ docker push quay.io/ciaranroche/memcached-operator:v0.0.1
The push refers to repository [quay.io/ciaranroche/memcached-operator]
db68bb3168fc: Pushed
05c2dea6380d: Mounted from ciaranroche/app-operator
v0.0.1: digest: sha256:a30a21aebb463580cc6bd0268975ae9d41a380e861c37ce02e82fcdf868a6ebe size: 3044
```

Figure 33: *Pushing Image to Quay.io*

Now with the image pushed to the repository we need to update our *Operator.yaml* file in order to reflect the current image which we have stored in our repo. The *Operator.yaml* file can be seen in Figure 34.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: app-operator
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        name: app-operator
10   template:
11     metadata:
12       labels:
13         name: app-operator
14     spec:
15       serviceAccountName: app-operator
16       containers:
17       - name: app-operator
18         image: quay.io/ciaranroche/memcached-operator:v0.0.1
```

Figure 34: *Operator.yaml* (Note: Some of the file has been left out)

With the *Operator.yaml* file updated we can begin running our Operator. The commands used are the same as those used in Section 2.1 with the etcd Operator. First we make sure the Custom Resource Definition is registered with the Kubernetes apiserver this can be seen in Figure 2

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (✓)
(master 1 1)
> kc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.ciaranroche.com created
```

Figure 35: *Registering the CRD*

Next we need to set up the Kubernetes Role Based Access Control, this can be seen in Figure 36.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (✓)
(master 1 1)
> kc create -f deploy/service_account.yaml
serviceaccount/app-operator created
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (✓)
(master 1 1)
> kc create -f deploy/role.yaml
role.rbac.authorization.k8s.io/app-operator created
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (✓)
(master 1 1)
> kc create -f deploy/role_binding.yaml
rolebinding.rbac.authorization.k8s.io/app-operator created
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (✓)
(master 1 1)
> kc create -f deploy/operator.yaml
deployment.apps/app-operator created
```

Figure 36: *Setup RBAC*

If we run the *get deployment* command we can see we have created our operator and have it running within our Kubernetes cluster. This can be seen in Figure 37

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (✓)
(master 1 1)
> kc get deployment
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
app-operator   1         1         1            1          47s
```

Figure 37: *Kubectl get deployments*

Finally in order to launch our memcached pods we need to create our Custom Resource. This file can be seen in Figure 38.

Figure 39 shows the creation of the Custom Resource.

```

1  apiVersion: cache.ciaranroche.com/v1alpha1
2  kind: Memcached
3  metadata:
4    name: example-memcached
5  spec:
6    size: 3
7

```

Figure 38: *Custom Resource*

```

(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 2 1)
> kc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
memcached.cache.ciaranroche.com/example-memcached created

```

Figure 39: *Create Custom Resource*

Now if we run the `get pods` command we can see that we have successfully launched our example pod and our app operator. This reflects the current state of our operator as our memcached pod is essentially a busy box. In Section 5.11 we will dive in deeper and update and create a working memcached pod.

```

(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 2 1)
> kc get pods

```

NAME	READY	STATUS	RESTARTS	AGE
app-operator-78d676d5c8-gbgx7	1/1	Running	0	4m
example-memcached-pod	1/1	Running	0	1m

Figure 40: *Kubectrl get pods*

5.11 Updating the Memcache Operator

This section will look at the Operator Controller in more detail. The entire code can be found at Appendix D. In Section 5.9 we covered the basis of the Controller in that it watches the resources and triggers a reconcile loop. We outlined some notable functions in particular the Reconcile Function, which can be seen back in Figure 30. We outlined that the reconcile function reads the current state of the cluster for our created objects and makes changes based on the current state. Currently the reconcile method will only launch a single busy box pod which we verified and used previously to ensure our operator was performing as expected

We will now edit this controller so that the reconcile function will create a Memcached Deployment for every Memcached Custom Resource. Reconcile reads that state of the cluster for a Memcached object and makes changes based on the state read.

Figure 41 shows the original function which created a Busy Box, this was called within our Reconcile Function.

```
// newPodForCR returns a busybox pod with the same name/namespace as the cr
func newPodForCR(cr *cachev1alpha1.Memcached) *corev1.Pod {
    labels := map[string]string{
        "app": cr.Name,
    }
    return &corev1.Pod{
        ObjectMeta: metav1.ObjectMeta{
            Name:      cr.Name + "-pod",
            Namespace: cr.Namespace,
            Labels:     labels,
        },
        Spec: corev1.PodSpec{
            Containers: []corev1.Container{
                {
                    Name:      "busybox",
                    Image:     "busybox",
                    Command: []string{"sleep", "3600"},
                },
            },
        },
    }
}
```

Figure 41: *Original Busy Box*

The following steps need to be completed, we must define our Memcached Deployment, we then need to add logic to our reconcile function to make decisions based on the current state of the cluster.

Figure 42 shows the deployment object. As can be seen the rough outline of the object definition follows a normal k8 file. We specify some Type and Object metadata. We outline the number of replicas we want and finally define our pod spec. As can be seen we are using a memcached alpine image.

The following code was added to the Reconcile Function, in Figure 43 we want to return our deployment object. Figure 44 is checking if the deployment already exists, if not we will create a new deployment, and then return to the reconcile queue. Figure 45 checks the deployment size, this is used to manage number of replicas within our cluster, and finally Figure 46 we want to update the current memcached status with the pod names.

These are typical scenarios and tasks which would normally require human intervention or with the aid of some sort of cron script.

```

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

    dep := &apps1.Deployment{
        TypeMeta: metav1.TypeMeta{
            APIVersion: "apps/v1",
            Kind:       "Deployment",
        },
        ObjectMeta: metav1.ObjectMeta{
            Name:      m.Name,
            Namespace: m.Namespace,
        },
        Spec: apps1.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",
                        }},
                    }},
                },
            },
        },
    }
}

```

Figure 42: *Memcached Deployment object*

```

// 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
        log.Printf("Memcached %s/%s not found. Ignoring since object must be deleted\n", request.Namespace, request.Name)
        return reconcile.Result{}, nil
    }
    // Error reading the object - requeue the request.
    log.Printf("Failed to get Memcached: %v", err)
    return reconcile.Result{}, err
}

```

Figure 43: *Getting Deployment Object*

```

// 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)
    log.Printf("Creating a new Deployment %s/%s\n", dep.Namespace, dep.Name)
    err = r.client.Create(context.TODO(), dep)
    if err != nil {
        log.Printf("Failed to create new Deployment: %v\n", err)
        return reconcile.Result{}, err
    }
    // Deployment created successfully - return and requeue
    return reconcile.Result{Requeue: true}, nil
} else if err != nil {
    log.Printf("Failed to get Deployment: %v\n", err)
    return reconcile.Result{}, err
}

```

Figure 44: *Checking Deployment Exists*

```

// 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 {
        log.Printf("Failed to update Deployment: %v\n", err)
        return reconcile.Result{}, err
    }
    // Spec updated - return and requeue
    return reconcile.Result{Requeue: true}, nil
}

```

Figure 45: *Checking Deployment Size*

```

// 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 {
    log.Printf("Failed to list pods: %v", err)
    return reconcile.Result{}, err
}
podNames := getPodNames(podList.Items)

```

Figure 46: *Updating status with pod names*

5.12 Running the updated Memcache Operator

The updated operator was built and run following the same steps as Section 5.10. Figure 47 shows when we run `get deployments` now we have our operator and our example memcached. As we can see the example memcached has a desired states of 3, current state of 3, up to date of 3 and with 3 available. This is also reflected in Figure 48 where when we ran `get pods` we have our operator pod running and 3 memcached pods running.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (master 1)
> kubectl get deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
app-operator        1         1         1            1           21s
example-memcached   3         3         3            3           7s
```

Figure 47: *Kubectl get deployments*

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (master 1)
> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
app-operator-7fc95b5ffc-b5899       1/1     Running   0          28s
example-memcached-6944856484-29t8q  1/1     Running   0          14s
example-memcached-6944856484-6cm1x  1/1     Running   0          14s
example-memcached-6944856484-n7g8f  1/1     Running   0          14s
```

Figure 48: *Kubectl get pods*

In figure 49 we are returning our example memcached pod config, as we can see from the output that it reflects what was set within the operator.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator) (master 1)
> kubectl get memcached/example-memcached -o yaml
apiVersion: cache.ciaranroche.com/v1alpha1
kind: Memcached
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"cache.ciaranroche.com/v1alpha1","kind":"Memcached","metadata":{"annotations":{},"name":"example-memcached"},"namespace":"default"},"spec":{"size":3}}
  clusterName: ""
  creationTimestamp: 2018-11-13T20:27:35Z
  generation: 1
  name: example-memcached
  namespace: default
  resourceVersion: "130782"
  selfLink: /apis/cache.ciaranroche.com/v1alpha1/namespaces/default/memcacheds/example-memcached
  uid: 8e78a8b9-e782-11e8-b307-0800275bb131
spec:
  size: 3
```

Figure 49: *Kubectl get memcached/example-memcached -o yaml*

5.13 Memcache Operator Summary

A demo of the steps taking to build and launch the Memcached example can be found at [Appendix E](#).

At face value it was quite intimidating when you see the sheer amount of boilerplate that is generated when you first bootstrap an operator. But when you get down to it, what you need to do in order to develop an operator is quite small. And it is quite easy to see the benefit that adding user logic that would normally require some sort of human intervention.

6 Operator Lifecycle Manager

Throughout [Section 5](#) we manually built and ran our memcached operator. In this section we will look at the Operator Lifecycle Manager (OLM), it is this tool that handles the install, updating and management of an operators lifecycle. The OLM runs as a Kubernetes extension and lets us use the *kubectl* for all the lifecycle functions with out the need for any additional tools.

Unfortunately I could not get the OLM to run as expected, the following documents the steps I took, as Operators are relatively new, there is currently a lack of resources online, with only the bare documentation to go off, I found it difficult to debug the issue.

Before proceeding I should explain the overall goal to what I set out to achieve in this section. As we have seen already, Operators are a way we can automate the management of our pods. But how to we manage our Operators, do we need to create an Operator for an Operator? This is where OLM comes in, the OLM manages a Kubernetes resource called *ClusterServiceVersion*. This CSV is applied to the cluster and through it the OLM manages our Operator. The CSV contains all the metadata needed to update the Operator.

So with that I set about adding a CSV to the memcached operator. [Figure 50](#) shows this, for the full CSV see [Appendix B](#).

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 3 1)
> cat memcachedoperator.0.0.1.csv.yaml
# This file defines the ClusterServiceVersion (CSV) to tell the catalog how to display, create and
# manage the application as a whole. If changes are made to the CRD for this application kind,
# make sure to replace those references below as well.
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: memcachedoperator.v0.0.1
  namespace: default
spec:
  install:
    strategy: deployment
    spec:
      permissions:
        - serviceAccountName: app-operator
        rules:
```

Figure 50: Operator CSV (image cut for brevity)

As per the documentation, Figure 51 shows the CSV being applied to the cluster.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 1)
> kc apply -f memcachedoperator.0.0.1.csv.yaml
clusterserviceversion.operators.coreos.com/memcachedoperator.v0.0.1 created
```

Figure 51: *Applying CSV*

Figure 52 shows the deployment of the of the Custom Resource Definition and the Role Based Access Control.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 1)
> kc apply -f deploy/
deployment.apps/app-operator created
role.rbac.authorization.k8s.io/app-operator created
rolebinding.rbac.authorization.k8s.io/app-operator created
serviceaccount/app-operator created
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 1)
> kc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.ciaranroche.com created
```

Figure 52: *Deploying CRD and RBAC*

Figure 53 shows the deployment of our Custom Resource to launch our pods.

```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 1)
> kc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
memcached.cache.ciaranroche.com/example-memcached created
```

Figure 53: *Deploy CR*

Figure 54 shows our operator and pods and running as expected.

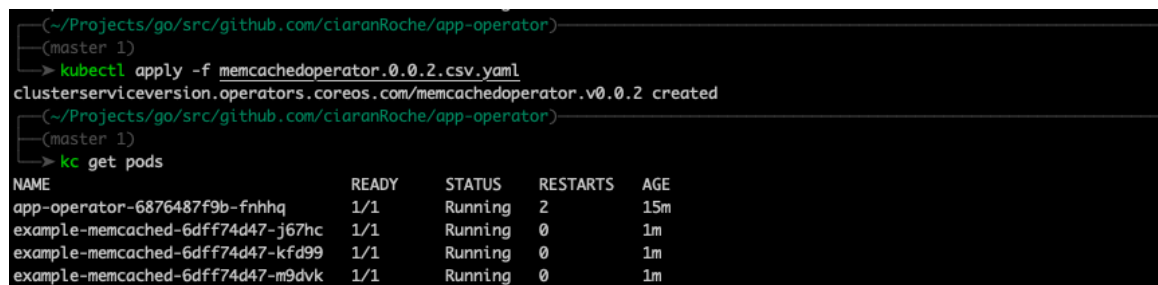
```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 1)
> kc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
app-operator-6876487f9b-fnhhq	1/1	Running	2	14m
example-memcached-6dff74d47-j67hc	1/1	Running	0	4s
example-memcached-6dff74d47-kfd99	1/1	Running	0	4s
example-memcached-6dff74d47-m9dvk	1/1	Running	0	4s

Figure 54: *Kubectrl get pods*

The next step was to apply a new CSV, this contained a different image for our operator. What we should see in Figure 55 is after our CSV has been deployed, our operator should be updated with the running memcached pods being unaffected. Unfortunately this does not happen. I spent quite some time trying to troubleshoot this problem. Due to a lack of documentation and tutorials online and with me experience with Operators limited to this paper I found it difficult. I suspect that I am missing some requirements needed for OLM as there is none in their documentation. I went to lodge an issue about this but noticed someone else had lodged one before me. As OLM provides a makefile, I was unable to run their make script due to missing requirements, the work around I found for this involved me having to manually apply the OLM manifests to my cluster. I am guessing that I may be missing something here.

Regardless I feel this section outlines where OLM fits with the Operator Framework.



```
(~/Projects/go/src/github.com/ciaranRoche/app-operator)
(master 1)
> kubectl apply -f memcachedoperator.0.0.2.csv.yaml
clusterserviceversion.operators.coreos.com/memcachedoperator.v0.0.2 created
~/Projects/go/src/github.com/ciaranRoche/app-operator
(master 1)
> kc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
app-operator-6876487f9b-fnhhq	1/1	Running	2	15m
example-memcached-6dffb74d47-j67hc	1/1	Running	0	1m
example-memcached-6dffb74d47-kfd99	1/1	Running	0	1m
example-memcached-6dffb74d47-m9dvk	1/1	Running	0	1m

Figure 55: Operator CSV (image cut for brevity)

7 Summary

In the Kubernetes world, Operators are very much a buzz word at the moment, as its ecosystem is growing quickly with a fast adoption rate for some major players. I first heard about operators on my internship at Red Hat, unfortunately the team I worked on didn't get much exposure to them until I was leaving. That said in the single sprint where we had an operator in our backlog I got to see the potential and power of them, away from the examples I have shown in this paper. We utilized the watch function of an operator to watch for changes within a cluster and through these changes it would trigger updates to a UI. Unfortunately I didn't get to work on the operator, I worked on the UI, but regardless I gained an appreciation for operators and my interest was sparked, with that I took this paper as an opportunity to gain some experience with them.

It was an experience exploring a new technology and there was many challenges presented in the lack of tutorials and blogs around the subject. The sole reliance on documentation left me with an overall greater appreciation I felt. At face value I failed to see what operators offered, it was only when I got into it and nearing the end of the paper, did I see how valuable and powerful it can be to be able to inject your business logic into your automation.

Seen the innovation that is happening in the community and the work going into the Operator Framework leaves me excited to see where operators will be in a years time. In my opinion I found them intimidating at the start due to the amount of boiler code. I would prefer to have some of this abstracted away, but that said I feel the boiler plate is probably needed as the complexity of an operator increases.

Operators bring a new level to Kubernetes, which is all ready extremely powerful. With the growth of Kubernetes and Containers in general it would be great to see a switch to treating the likes of AWS and Azure as a commodity, without the fear of vendor locking. Allowing us to choose a cloud provider based on performance and cost and not based on services.

A video was made to compliment this paper showing some of the work carried out throughout. It can be found at [Appendix F](#)

Appendices

A CoreOS etcd Operator

<https://github.com/coreos/etcd-operator>

B Operator SDK

<https://github.com/operator-framework/operator-sdk>

C Memcached Operator Example

<https://github.com/ciaranRoche/memcached-operator-example>

D Memcached Operator Controller

[https://github.com/ciaranRoche/memcached-operator-example/
blob/master/pkg/controller/memcached/memcached_controller.go](https://github.com/ciaranRoche/memcached-operator-example/blob/master/pkg/controller/memcached/memcached_controller.go)

E Operator Demo

<https://asciinema.org/a/211619>

F Full Operator Demo Youtube

<https://www.youtube.com/watch?v=Og-sCyPYG8o&t=1s>

Bibliography

Concepts (2018).

URL: <https://kubernetes.io/docs/concepts/>

Controllers (2018).

URL: <https://kubernetes.io/docs/concepts/workloads/controllers/>

CoreOS (2018).

URL: <https://coreos.com/operators/>

CoreOS Blog (2016).

URL: <https://coreos.com/blog/introducing-operators.html>

CoreOS Blog (2018a).

URL: <https://coreos.com/blog/introducing-the-etcd-operator.html>

CoreOS Blog (2018b).

URL: <https://coreos.com/blog/introducing-operator-framework>

Custom Resources (2018).

URL: <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>

Dormando (n.d.).

URL: <https://memcached.org/>

operator framework (2018a), ‘operator-framework/operator-lifecycle-manager’.

URL: <https://github.com/operator-framework/operator-lifecycle-manager>

operator framework (2018b), ‘operator-framework/operator-metering’.

URL: <https://github.com/operator-framework/operator-metering>

operator framework (2018c), ‘operator-framework/operator-sdk’.

URL: <https://github.com/operator-framework/operator-sdk>