João Pedro Luna Fernandes

.NET Operator SDK Developing Kubernetes Operators in .NET

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Science in
Software Engineering

Advisers: Joaquim Filipe
Full Professor, Instituto Politécnico de Setúbal
Filipe Mariano
Assistant Professor, Instituto Politécnico de Setúbal

Setembro, 2021

.NET Operator SDK Developing Kubernetes Operators in .NET

Copyright © João Pedro Luna Fernandes, Escola Superior de Tecnologia de Setúbal, Polytechnic Institute of Setúbal.
The Escola Superior de Tecnologia de Setúbal and the Polytechnic Institute of Setúbal have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

To everyone that helped me throughout this project, thank you.

Acknowledgements

I would like to thank the following people for helping with this project:

My ex-team at work for making this project possible, without whom there wouldn't have been the curiosity to create this solution. All my friends and colleagues that helped me test this project and reviewed this dissertation.

Speed is everything. It is the indispensable ingredient to competitiveness.

Abstract

Kubernetes, microservices architectures and cloud computing are rising trends in the software industry. Ever growing enterprises required more and more attributes such as availability, maintainability, security and reliability. They are turning to technologies that enable these and riding the wave is Kubernetes. Kubernetes aims to automate a lot of operational knowledge but does it know how to automate domain-specific operational tasks? To solve this problem there are several projects to develop these Kubernetes-native domain aware applications called Operators. The problem is that these are mostly developed in Go for Go developers and there are none to develop using .NET.

.NET is used worldwide across enterprises which makes it an interesting choice as the technology in which to build Operators. Enterprises should be allowed to keep consistency across their technologies of choice instead of implementing an Operator in a language they might not be familiar with or not be allowed by their clients.

To solve this problem the .NET Operator SDK was created. This document presents the current state of the .NET Operator SDK which is of a MVP.

With the .NET Operator SDK developers will be able to develop Operators in .NET more easily than they would using a simple client. It reduces a lot the boilerplate code and offers a baseline structure for developers to build onto. Having a more structured codebase will allow to build more complex Operators to address complex business needs.

Keywords: .NET, Kubernetes, Operators, C#, Operations, Docker, Kubernetes-native, DevOps

Resumo

Kubernetes, arquiteturas de microsserviços e computação em nuvem são tendências crescentes na indústria de software. As empresas em crescimento exigiam cada vez mais atributos como disponibilidade, capacidade de manutenção, segurança e confiabilidade. Eles estão se voltando para tecnologias que permitem isso e, na onda, está o Kubernetes. O Kubernetes visa automatizar muito conhecimento operacional, mas ele sabe como automatizar tarefas operacionais específicas do domínio? Para resolver esse problema, existem vários projetos para desenvolver esses aplicativos com reconhecimento de domínio nativo do Kubernetes chamados de Operadores. O problema é que eles são desenvolvidos principalmente em desenvolvedores Go for Go e não há nenhum para desenvolver usando .NET.

.NET é usado em todo o mundo em empresas, o que o torna uma escolha interessante como a tecnologia na qual construir Operadores. As empresas devem ter permissão para manter a consistência em suas tecnologias de escolha, em vez de implementar um Operador em um idioma com o qual eles podem não estar familiarizados ou não ser permitido por seus clientes.

Para resolver este problema, foi criado o .NET Operator SDK. Este documento apresenta o estado atual do .NET Operator SDK, que é de um MVP.

Com o SDK do Operador .NET, os desenvolvedores poderão desenvolver Operadores em .NET mais facilmente do que com um cliente simples. Ele reduz muito o código clichê e oferece uma estrutura básica para os desenvolvedores criarem. Ter uma base de código mais estruturada permitirá construir operadores mais complexos para atender às necessidades de negócios complexas.

Palavras-chave: .NET, Kubernetes, Operadores, C#, Operations, Docker, Kubernetes-native, DevOps

Contents

List of Figures **xvii**

List of Tables **xix**

List of Listings **xxi**

Glossary **xxiii**

Acronyms **xxv**

1 Introduction **1**
- 1.1 A Bit of History 2
  - 1.1.1 Containers 3
- 1.2 Orchestration 3
- 1.3 Cloud & Bare-metal 4
  - 1.3.1 NGINX Ingress Controller 4
- 1.4 .NET 5

2 Domain Driven Design **7**
- 2.1 Why Domain Driven Design? 7
- 2.2 Ubiquitous Language 7
- 2.3 Service Layers 8

3 Kubernetes **9**
- 3.1 What is Kubernetes? 9
- 3.2 Kubernetes Objects 9
  - 3.2.1 Pods 9
  - 3.2.2 Deployments 10
  - 3.2.3 Services 10
  - 3.2.4 Custom Resources 10
  - 3.2.5 Custom Resource Definitions 10
  - 3.2.6 Service Accounts 10
  - 3.2.7 Roles 10
  - 3.2.8 Role Bindings 11
- 3.3 etcd 11
- 3.4 A Case Study 11

4 Kubernetes-Native **13**
- 4.1 Controllers 13

4.2 Operators 14
4.2.1 KEDA 15
4.2.2 cert-manager 15
4.3 Operator Maturity Model 16
4.4 Shared Informers 16
4.5 Current Solutions 17
4.5.1 Kubernetes Clients 17
4.5.2 KUDO 18
4.5.3 Operator SDK 19

5 .NET Operator SDK **21**
5.1 The Current Problems 21
5.2 Guidelines 22
5.3 Bootstrapping 22
5.3.1 Folder: Application 22
5.3.2 Folder: Controller 23
5.3.3 Folder: Resources 23
5.3.4 Folder: deploy 23
5.4 Reconciliation 23
5.5 Error Handling 24
5.6 A Case Study 24

6 Operator Anatomy **27**
6.1 Reflection 27
6.2 Types of controllers 28
6.2.1 Watcher Controller 28
6.2.2 Informer Controller 29
6.3 Kubernetes client 29
6.4 Dependency Injection 30
6.5 The Operator Class 31
6.6 The Hosted Service 32

7 Results **33**
7.1 Development Environment 33
7.2 Tests 34
7.3 Architecture 34
7.3.1 Package Diagram 35
7.4 The Case Study 36

8 Conclusion and Future Work **39**
8.1 Future Work 39
8.1.1 Bootstrapping CLI 39
8.1.2 Further optimizations 39
8.1.3 Open Source Contributions 40
8.1.4 Further field testing 40
8.1.5 Unit and Integration Testing 40
8.1.6 CI/CD 40
8.1.7 Telemetry 40

Bibliography **43**

List of Figures

1.1 Docker architecture 2
1.2 Kubernetes Service Providers 4
1.3 NGINX's ingress controller model from NGINX's official website 4
1.4 Most used programming languages among developers worldwide, as of 2021 . . 6

4.1 Control loop. 14
4.2 Operator performing manual tasks to achieve a desired state 15
4.3 Operator executing its control loop to achieve a desired state 15
4.4 Operator Maturity Model 16
4.5 Conceptual view of a Plan with Phases which in turn have Steps 18
4.6 Architecure of KUDO 19

5.1 Operator project file structure 22

6.1 Informer controller's event optimization queue 30

7.1 Cluster built of 2 nodes and a persistence unit 33
7.2 Development environment of the .NET Operator SDK 34
7.3 Implementations being changed without affecting the context 35
7.4 .NET Operator SDK package diagram 35
7.5 Class diagram showing the relevant components of the case study's Operator . . 36
7.6 Operator deployed in a kubernetes cluster that manages internal and external subjects 37

8.1 Health checks and heart beats example 41

List of Tables

List of Listings

Glossary

cgroups cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. i, 2

computer An electronic device which is capable of receiving information (data) in a particular form and of performing a sequence of operations in accordance with a predetermined but variable set of procedural instructions (program) to produce a result in the form of information or signals. i

virtualization Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs) i, 2

Acronyms

API Application Programming Interface i

AWS Amazon Web Services i, 4

CD Continuous Delivery i

CI Continuous Integration i, 40

CR Custom Resource i, 10, 12, 23

CRD Custom Resource Definition i, 10, 12, 23

DDD Domain-Driven Design i, 1, 7, 23

DI Dependency Injection i, 31, 32

EKS Elastic Kubernetes Service i, 4

GCP Google Cloud Platform i, 4

SDK Software Development Kit i

SIG Special Interest Group i

YAML YAML Ain't Markup Language i, 23

1 Introduction

Kubernetes is a hot topic in IT and has been for x years now. While of those that have heard about it, most know it but only a fraction understands it. Its complexity comes from many fronts. Being written in Go, despite being developed with simplicity in mind, by arguably three of the best minds in computer science, Robert Griesemer, Ken Thompson and Rob Spike is one of those fronts.

This dissertation aims to answer the question "How can one develop domain-aware kubernetes-native applications using .NET?". A quite specific question, but wide nonetheless. Despite its specificity, this question cross-cuts many relevant concerns of today's enterprise ecosystems. To demonstrate this, the previous question can be separated into many parts.

Starting with *domain-aware*. This means being aware of the surrounding domain concepts and business logic. Domain-Driven Design (DDD) has whole chapter dedicated to it in this dissertation.

Being *Kubernetes-native* means making direct use of kubernetes. Most applications running on *Kubernetes* are not kubernetes native as they are running inside containers on kubernetes but could be running inside containers on top of a different infrastructure. Thus they do not make use of *Kubernetes* directly. There is a chapter dedicated to these applications but they will be referenced a lot across most of them as they are one of the archetypes of the .NET Operator SDK.

One might ask, "well aren't there already plenty of solutions that help building these "domain-aware kubernetes-native"applications?" There are some, but there are none when it comes to building them in .NET C#, if wrapped REST clients are excluded, which should be partially excluded as they are not a good solution to the problem. The Go language dominates this sector and it makes sense that it does so as most of the ecosystem is built on that technology. The *.NET* concern in the previous question will be the main constraint in the implemented solution.

The *How* is not entirely the scope of this dissertation in the sense that it will only give a superficial overview of the usage of the SDK. It will explore the concepts, introduce the ecosystem, present the solution and explore implementation details that are part of a MVP version of the MVP. Then it is fair to say that this dissertation does not answer the question, right? That's right. But it does *allow that question to have a positive*, *satisfying answer*.

This project started due to the need of implementing a domain aware scaling solution in an enterprise. The requirement was that when a large message A arrived to the system, it would need to upscale the number of services running in the cluster to process this large message without reducing the throughput of other smaller messages. What made message A different from the other messages was business concerns.

As when that message arrived could not be predicted, it would be unreliable to have an operation's team member wait for that message to arrive to then scale the whole application. It was also not a good solution to have a large number of services always running only because message A could arrive. This would be a waste of resources.

Thus a solution that knew enough about the domain and also automatically performed Kubernetes operations was required. The last constraint was that it had to be developed in .NET because it was the framework being used throughout the whole system and what every team built on.

This is how the idea for the .NET Operator SDK was born.

1.1 A Bit of History

The year is 1970 and virtualization is being developed to resolve the need for many users to share a machine's resources simultaneously. In the early days of virtualization there were terminals connected to a mainframe as to perform tasks in a centralized location.[12] If one of the users crashed the mainframe it would result in the system going down for everyone connected to the mainframe. This led to the need of isolating environments while sharing system resources at the same time.

At the end of the 70s came the development of `chroot` that enabled the change of the apparent root directory of a running process and all its running and subsequent children. As per the documentation [1]:

> chroot() changes the root directory of the calling process to that specified in path... The root directory is inherited by all children of the calling process... A child process created via fork(2) inherits its parent's root directory.

In the first decade of 2000 engineers at Google developed and released *process containers*, which purpose was to isolate and limit the resource usage of processes. *Process containers* were later renamed to *control groups*, or short, cgroups. *Control groups* will be explained in the next section.

In 2013 came Docker and with it the popularity of container technology grew. Docker leverages the amount of work on the host operating system by providing a layer between the host and the running containers. Only binaries and libraries are copied into the application container, removing the overhead of having a whole operating system and managing its resources. Because of this, containers are fast to start and destroy as need arises.

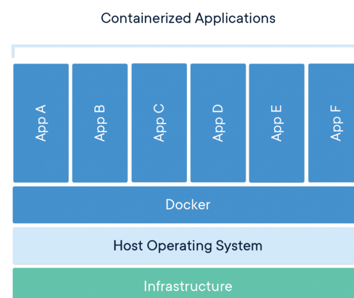

Figure 1.1: Docker architecture

1.1.1 Containers

A container is, at its core, an isolated process above the kernel. They are in essence like virtual machines, for the fact that they own process space, own network interfaces, can run services and can install packages. Its building blocks are namespaces that limit what "it can see", cgroups that limit what "it can do"and UFS. There are currently six types of namespaces implemented in the Linux kernel excluding cgroups as per the *namespace(7)* manual page. [2]

The mount (mnt) namespace which is the namespace that isolates mount points across processes. The process ID (pid) namespace can be used to provide processes with a set o process ID's independent from the other namespaces starting with PID 1. The Network (net) namespace takes care of isolating network devices, stacks, ports, etc. The inter-process communication (ipc) namespace for isolating inter-process communication. The UTS (uts) namespace isolates the hostname and NIS domain name. Finally the user ID (user) namespace isolates the user a group IDs.

The PID isolation can easily be demonstrated by simply running a docker container and printing all the processes withing the container.

```
$ docker run -it alpine /bin/sh
/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    7 root      0:00 ps
/ #
```

As can be seen, the main process (a shell) is isolated from the host and has its PID set as 1. The other process with PID 7 is the `ps` command that shows the current processes in the filesystem where the command was executed [3], in this case he container's filesystem.

One simple container can be codified in under 100 lines of code.

1.2 Orchestration

As microservice-based applications grow, so does the complexity of the system. Managing such environments is important to get the full potential of containers and their applications.

In a microservice-architecture the cluster has to ideally benefit from a wide number of characteristics. Availability, that is the degree to which the system is able to perform without interruptions. It should be fault tolerant. A cluster should be scalable as well due to the high number of services that will inevitably run on them. Reliable in the sense that these systems should perform consistently. Lastly it should benefit from security.

Container orchestrators, as in container orchestration platforms, simplify the deployment of multiple containers and the management of container lifecycle [10] to achieve the previously defined characteristics. Any container orchestrator no matter the vendor should be capable of providing the previously referred characteristics, such as availability and security. Enable integration with an organization's CI/CD practice, by making continuous deployment possible and relatively easy. Last but not least, should provide relevant insights of the system through monitoring and governance.

Some of the most known and used container orchestration platforms are, Kubernetes, Docker Swarm, Nomad and Mesosphere. An overview of Kubernetes and its native components will be presented in *Chapter 5*.

1.3 Cloud & Bare-metal

When it comes to these Orchestrators they can be served in different kinds of infrastructures. There can be a solution provided Off-Premises on any famous cloud provider such as, Azure Kubernetes Services on Azure, Amazon EKS on AWS or Google Kubernetes Service on GCP.

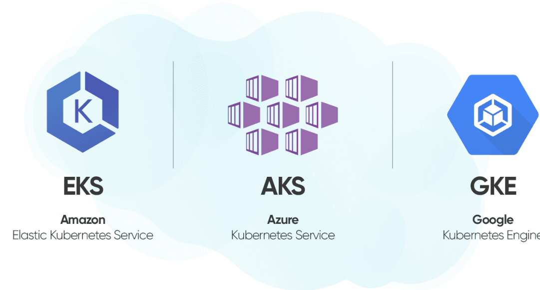

Figure 1.2: Kubernetes Service Providers

On the other side there are On-Premises Bare metal solutions. These require a deeper knowledge of Kubernetes administration and other related subjects such as Docker and Linux. It's harder to achieve a production ready environment on a bare metal setup but the result can sometimes save costs in the long term. Although they require more operational knowledge the tradeoff is a more fine-grained control over the whole environment.

For context, all the development and testing of the solution in this dissertation was done in an On-Premises bare-metal Kubernetes cluster. It was running with an High Availability configuration consisting of one master node and one worker node. The services were running behind a Bare-metal load balancing solution and an NGINX Ingress controller for hostname based routing on the exposed services.

1.3.1 NGINX Ingress Controller

NGINX and Ingress Controllers are an interesting topic but they are not the scope of this document. This subsection is only meant to invite the user to read about them as they are a useful toolkit in building distributed systems on Kubernetes. The model in figure 1.3 illustrates how an NGINX Ingress Controller can be used to expose several applications inside a Kubernetes cluster.

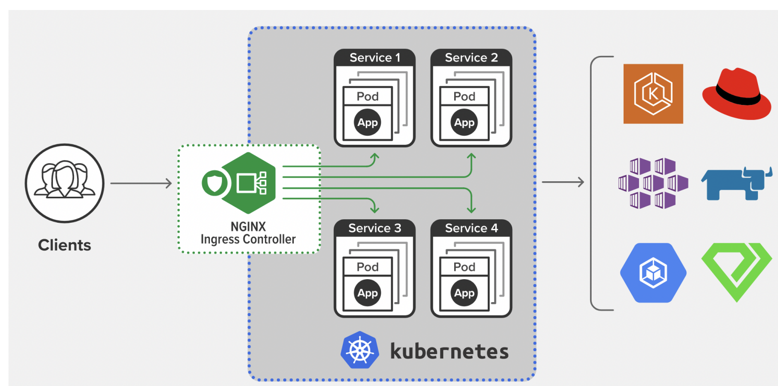

Figure 1.3: NGINX's ingress controller model from NGINX's official website

1.4 .NET

.NET [17] is a free, cross-platform, open source developer platform. .NET allows developers to build for web, mobile, desktop, games, and IoT. It is open source and under the .NET Foundation.

.NET can be used to develop for many operating systems [14], including:

- Windows
- macOS
- Linux
- Android
- iOS
- tvOS
- watchOS
- Lisp
- Perl

It also supports many processor architectures, including:

- x64
- x86
- ARM32
- ARM64

Which makes .NET extremely versatile as choice to develop containerized applications.

Another advantage of using .NET to build systems is the fact that it has a large community. This community offers users a great support in growth and adoption of this technology.

.NET can be used in different languages including C# which is one of the most widely used languages. Figure 1.4 shows the most used programming languages among developers worldwide and we can find C# there. This was taken from statista.com at the time of writing.

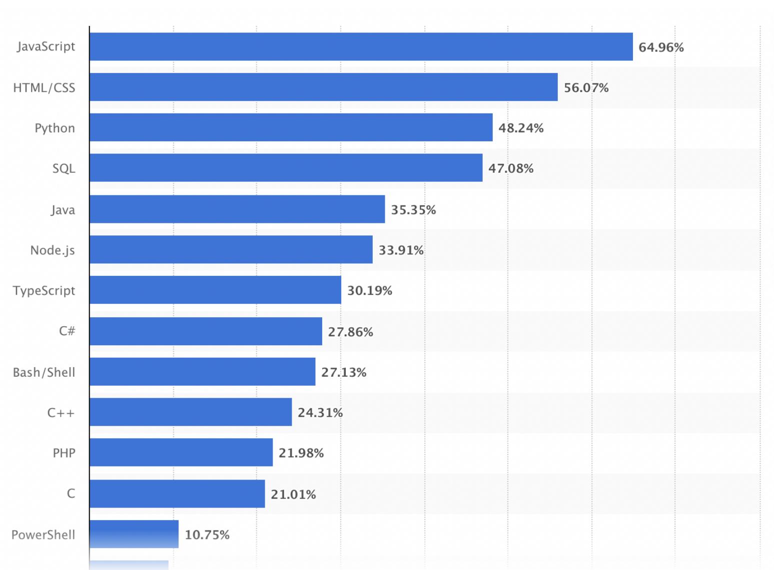

Figure 1.4: Most used programming languages among developers worldwide, as of 2021

2

Domain Driven Design

The work on this dissertation is tightly tied to the concept of Domain Driven Design. As such, these concepts, as well as how are they related to .NET Operator SDK, will be briefly discussed in this chapter. There are entire libraries dedicated to the subject which makes this chapter an overview rather than an in-depth tour to the world of DDD. Without going into much detail, as that is the purpose of future chapters, the ties between DDD and the systems built using the project presented in this dissertation.

2.1 Why Domain Driven Design?

To create great software, developers have to know what that software is all about. They should know its *domain*. Developers that do not have knowledge about energy lack the tools to make good energy grid software.

It is important to create a model which is deeply rooted in the domain, and should reflect its essential concepts with great accuracy to allow the people involved to communicate and express efficiently. Domain Driven Design provides the techniques and concepts that enable developers and business responsible to create accurate models of the problems at hand. [5] DDD is crucial when building operators using the .NET Operator SDK as it tries to pass human behavior into a programed controller. Both programmers and domain experts take part in developing an Ubiquitous Language that will help define the *Operator's* implementation.

Considering for instance when a *Custom Resource Definition* is created to define a new Kubernetes API object. Usually, what is happing at the a higher level is the introduction of some elements of the business domain into the scope of the cluster. The *Operators* make use of these models as part of their internal workings. They can be considered a "blueprint"of the operator that Kubernetes understands.

2.2 Ubiquitous Language

In a software project, developers must communicate to exchange ideas about the domain models, about the elements involve, how we connect them, what is relevant and what is not so relevant. Although this is not enough to guarantee the success of a project, it is a great start.

DDD has, as a core principle, the use of a language based on the domain model. This language should appear consistently in all the communication inside the team. This language is called Ubiquitous Language.

It should connect parts of the system and create the premise for the design team to work well. Domain experts should help in the creation of this language to avoid awkward or inadequate

terms or structures for domain understanding. They should be able to understand everything in the model, if not, something is most likely wrong with it.

The process of creating a Ubiquitous Language is quite verbose. A joint effort should be put into it and all team members should be aware of the need to created it. Everyone has to stay focused on the essentials and use it whenever necessary. One recommendation[5], is that developers should implement the main concepts of the domain model in the code. That is very helpful as it makes the code more readable and a more accurate representation of the domain model. Doing so will prove to be worth later in a project.

Lastly, how should this language be expressed? It can be through speech, writing, UML or documentation. One note though, do not try to model the whole project in a single diagram as it will most likely result in a hard to understand model due to being cluttered with information. Instead separate it into concerns as best fit to allow ease of read.

2.3 Service Layers

The .NET Operator SDK is built with DDD in mind. It offers developers the archetypes to build their operators and generates most of the boilerplate code that is not focused in the business domain. Mostly the Service layer is expected to be implemented by developers using the SDK. There might be the need to extend the Infrastructure layer of the *Operator* for more fine-grained control over Kubernetes operations. The layered structure provided by the .NET Operator SDK in the bootstrapping code will be explored in more detail in the next chapters.

3

Kubernetes

This chapter will be dedicated to providing an overview of the Kubernetes ecosystem and outline the core components around Operators. It will not be an introduction to kubernetes neither a guide on how to use it. It will focus on architectural aspects of the platform and briefly explore some of Kubernetes' first-class citizens.

3.1 What is Kubernetes?

You probably heard a lot of buzzwords surrounding the definition of Kubernetes such as "orchestrator"and "containerised applications". Those are the ones that come to mind. Kubernetes is all that and more.

It is an *application orchestrator* in the sense that Kubernetes deploys, scales, heals and does many more things with applications. [18] These applications can be fine-grained within the domain and are usually called microservices. But Kubernetes can be efficiently used for bigger services, it is not tied to fine-grained microservices.

These applications that Kubernetes run are packaged and run as containers, thus the "containerized applications"term. Containers were introduced in a previous chapter. The standard low-level technology used by Kubernetes is the famous *container runtime* Docker. Although Docker is the standard it is not limited to that specific *container runtime*. One common alternative is the lightweight containerd.

Kubernetes is the most widely used container orchestration solution in the market. Because of such widely spread use, the need rose to operate it efficiently.

In the next sections several of the building blocks of the Kubernetes ecosystem will be exposed. Note there are many more but only the most relevant will be in the next sections.

3.2 Kubernetes Objects

3.2.1 Pods

The smallest deployable units of computing that you can create and manage inside a Kubernetes cluster are called *Pods*.

A *Pod* can be made of one or more containers, which share storage resources, network resources and a specification for how to run those containers. A *Pod's* contents are always co-located and co-scheduled, and run in a shared context. A *Pod* models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

3.2.2 Deployments

In Kubernetes, a *Deployment* is used to create or modify instances of *Pods*. They are a high level abstraction to the deployment of pods. Deployments are responsible for many tasks such as scaling the number of replicated pods and enable the rollout of updated code in a controlled manner.

3.2.3 Services

A Kubernetes service is an high-level abstraction for a deployed group of identical pods in a Kubernetes cluster. Since pods are ephemeral, this means they are supposed to last for a short period of time, a service enables a group of pods to be assigned a name, unique IP address and/or a DNS name. These services can allow for a workload to be exposed outside of a cluster through the NodePort or LoadBalancer service type.

3.2.4 Custom Resources

Kubernetes is designed to be extended easily through the use of built-in mechanisms such as Custom Resources (CR). [18] These CRs can be interacted with just like any other native resource in a per-cluster basis. Normally CRs are watched by custom controllers that in turn create, update, or delete other cluster objects or even arbitrary resources outside of the cluster.[18]

3.2.5 Custom Resource Definitions

> A CRD is akin to a schema for a CR, defining the CR's fields and the types of values those fields contain.[18]

A Custom Resource Definition (CRD) is what defines a (CR) discussed previously. It is the blueprint of a specific CR that will be created in the cluster.

3.2.6 Service Accounts

All Kubernetes clusters have two categories of users: service accounts managed by Kubernetes, and normal users. service accounts are users managed by the Kubernetes API. They are bound to specific namespaces, and created automatically by the API server or manually through API calls.[16] Service accounts are tied to a set of credentials stored as *Secrets*, which are mounted into pods allowing in-cluster processes to talk to the Kubernetes API. These are useful when it is required that Pods interact with the Kubernetes API to perform some kind of operation.

```
1  apiVersion: v1
2  kind: ServiceAccount
3  metadata:
4     name: example-operator-sa
```

3.2.7 Roles

A role specifies the rules by which specific users, groups or service accounts will be bound. It specifies which *resources* from which *API groups* have which *permissions*. An important characteristic of a role is that it is scoped to a namespace. For instance consider this role definition:

```
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: Role
3  metadata:
4     namespace: default
5     name: pod-reader
6  rules:
7     - apiGroups: [""] # "" indicates the core API group
8     resources: ["pods"]
9     verbs: ["get", "watch", "list"]
```

This definition creates a role that allows for a user to perform read operations on *pods* on the default namespace. [16]

Another important concept to know about are *Cluste Roles*. These are not much different from *Roles*, with the biggest difference being that a *Cluster Role* is, as the name suggests, scoped to the whole Kubernetes cluster. There is no need to go further into detail about these.

3.2.8 Role Bindings

A role binding, as the name suggests, binds the permissions defined in a role to a user or set of users. It holds a list of subjects, these can be either users, groups, or service accounts, and defines a reference to the role being granted to those subjects. Below is a sample *ClusterRole* to bind the *Role* defined previously.

```
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: RoleBinding
3  metadata:
4    name: read-pods
5    namespace: default
6  subjects:
7     - kind: User
8     name: john
9     apiGroup: rbac.authorization.k8s.io
10 roleRef:
11    kind: Role
12    name: pod-reader
13    apiGroup: rbac.authorization.k8s.io
```

A *RoleBinding* grants permissions to any *Role* within a specific namespace whereas, similarly to what happens with *Roles*, a *ClusterRoleBinding* grants a cluster-wide access to a cluster roles. A *RoleBinding* can reference a *ClusterRole* and bind that *ClusterRole* to the namespace of the *RoleBinding*.[16]

3.3 etcd

Kubernetes stores some of its data and metadata such as deployment data inside etcd. etcd is a distributed key-value storage focused on consistency. A curious note about etcd, it was the first technology to have a corresponding Kubernetes Operator.

These were some of the building blocks of any Kubernetes cluster. They allow for our workloads to be executed in the cluster as well as manage their scope. There are many more concepts explained in the official documentation.

3.4 A Case Study

Starting from this chapter, at the end of every chapter there will be a section dedicated to a case study Operator. This will be a simple Operator to allow the reader to understand and

use practically the concepts explored previously. This operator will watch for a specific CR and act upon events on that resource. The subject of the Operator in this case study will be called a Mob.

As seen previously there are quite a lot of building blocks in a Kubernetes cluster. Our Operator will make use of many.

First, there needs to be a CRD, which will be called Mob. Below is a possible definition of the Mob resource.

```
1  apiVersion: apiextensions.k8s.io/v1beta1
2  kind: CustomResourceDefinition
3  metadata:
4    name: mobs.company.dev
5  spec:
6    group: company.dev
7    versions:
8      - name: v1alpha1
9      served: true
10     storage: true
11   scope: Namespaced
12   names:
13     plural: mobs
14     singular: mob
15     kind: Mob
```

After that, there needs to be a *Custom Resource* created in the cluster corresponding to the CRD created just now. Below is the configuration of that same CR.

```
1  apiVersion: "company.dev/v1alpha1"
2  kind: Mob
3  metadata:
4    name: mob1
5  spec:
6    replicas: 2
```

The Mob resource specification only contains an entry for the number of replicas. The replicas field in the spec will control the number of pods that need to available.

4

Kubernetes-Native

Kubernetes-native applications are applications that are both deployed in kubernetes and managed through the kubernetes API. The reason why kubernetes-native applications are relevant is because they enable the extension of kubernetes and empower systems with domain-driven cluster customization. Although orchestrators already know how to automate much of the tasks operation teams would do daily, there are still some business specific tasks that orchestrators are unable to perform due to limited knowledge about the domain.

4.1 Controllers

To understand what are operators and what they are capable of, there is the need to first know about controllers. Controllers act on resources. Core controllers act on core resources and are part of the control plane. User defined custom resources, through *Custom Resource Definitions* are managed by custom controllers.

Michael Hausenblas and Stefan Schimanski refer, in their book about Kubernetes operators [] three different solutions to implement custom controllers and operators. One is using the Kubernetes client in Go, which is also available in a number of different languages. The other is using the Kubebuilder, a tool designed to allow building Kubernetes APIs easily. And finally the Operator SDK written in Go, part of the Operator Framework developed by CoreOS which introduced the concept of Operators.

A Kubernetes controller is composed of a control loop that receives events from the API server, either through listing or watching. These events contain the state of a resource, which kind is being watched by the controller. The control loop, will attempt to reconcile and move the current state of the cluster towards the desired declared by the event's state.

The control loop normally is composed of three stages [13] no matter the complexity of the controller. (1) In the first stage the state of a resource is obtained through an event (in case of an event-driven approach). (2) The control loop will then try to reconcile the cluster (and any external resource) and move them to the desired state. Consider for instance that an event is received from the API server saying that a user defined custom resource was modified and now instead of having a *replicas* value of 3 has a value of 5. This stage is responsible for the creation of 2 more replicas. (3) Update the resource's status in the API server *etcd* storage. As the name suggests this is a loop and as such upon finishing the third stage the control loop cycle repeats itself again and again.

The problem here is that for building custom controllers and operators easily it becomes hard to escape from Go's grasp. Note that Go is an incredible technology that brought amazing new realities to the table. Consider Go routines for instance, the Go's user mode lightweight threads that solve old problems like the memory overhead problem from traditional threads, resizing initial stack size from 1MB (which in can reach 1GB in 1000 threads) to just 2KB and

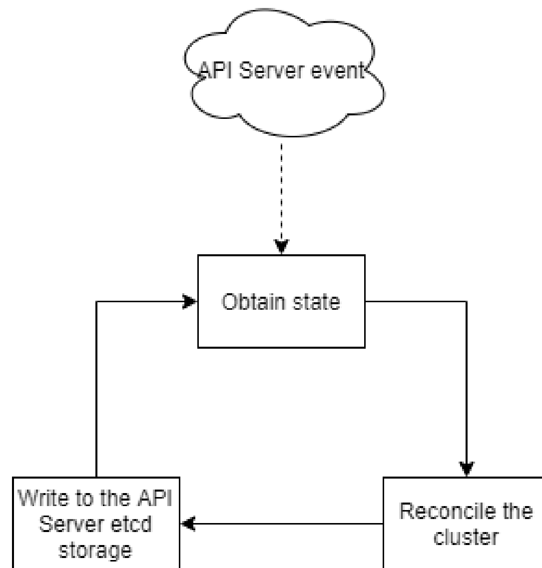

Figure 4.1: Control loop.

the problem associated with context-switching that Go solved by implementing a runtime scheduler to manage its user-mode threads. The point here is that developers are limited when it comes to writing controllers and operators in other programming languages. Not just to develop controllers and operators, developers are limited when it comes to extending the Kubernetes API without using Go.

4.2 Operators

> An Operator is a Kubernetes API extension in the form of a controller to automate domain-specific workflow actions through custom resource definitions.
> - Operator Framework

Operators are domain-driven controllers that manage custom resources with some operational and business knowledge. They allow clusters to reach new lengths in terms of customer and product value. The lifecycle of cluster resources are no longer managed by "dumb"entities without any business logic. Operators can, for example, enable scaling to be based on more than traffic and throughput. They allow for business rules to dictate the state of the cluster and to better adapt it to the needs.

Operators extend the Kubernetes API and the control plane through what is called a *custom resource* (CR). They allow developers to manage the applications they are delivering to their costumers and make them first-class citizens of the Kubernetes API. More and more operators are being developed to automate the operations of infrastructure engineers and operations teams.

An Operator is analogous to an infrastructure operator. An infrastructure operator is a person responsible of performing tasks to guarantee that a system's infrastructure is running according to the specifications the current demand and other factors. In the context of Kubernetes an operator would, as an example, manually increase the number of replicas of a given high demanded service at peak times.

With Operators (Kubernetes-native controllers) this process is no longer manual and prone to human error and delays. The Operator "watches"the current state of the cluster, executes its *control loop* to move the cluster (or external infrastructure, if the controller has access to outside resources) to its desired state.

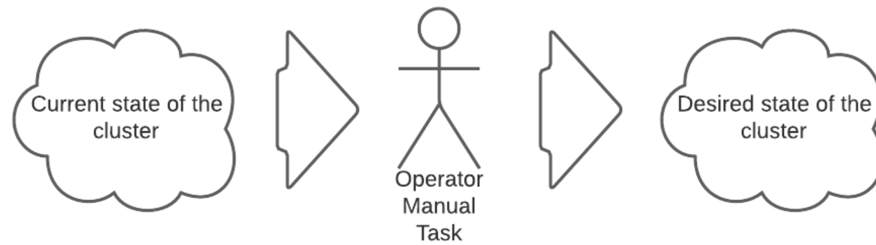

Figure 4.2: Operator performing manual tasks to achieve a desired state

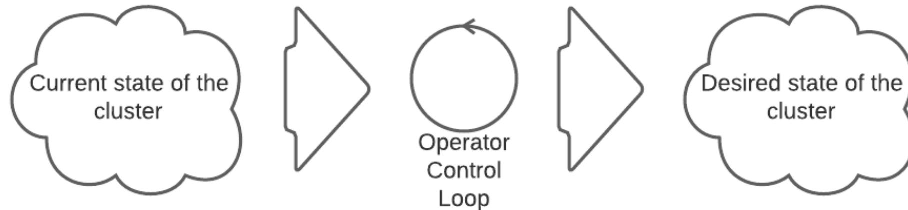

Figure 4.3: Operator executing its control loop to achieve a desired state

The next subsections will present some examples of existing Operators.

4.2.1 KEDA

From KEDA's official documentation [9], KEDA stands for Kubernetes-based Event Driven Autoscaler, and that is exactly what it is. With KEDA, you can drive the scaling of any container in Kubernetes based on the number of events needing to be processed. You can also map the apps you want to use event-driven scale, with other apps continuing to function.

KEDA supports many different event sources and scalers such as Apache Kafka, AWS Cloud Watch, Azure Blob Storage, Azure Service Bus, CPU, Cron, PostgreSQL, MongoDB, RabbitMQ and many more.

4.2.2 cert-manager

One of the most widely used kubernetes components is the cert-manager. cert-manager allows the automation of the management and issuance of TLS certificates from various issuing sources. It will make sure that the certificates issued are valid and up to date periodically and it will attempt to renew certificates at an appropriate time before they expire. In its essence, cert-manager is a Kubernetes operator as it comes with its own representation of SSL related entities, such as Certificates, Issuers, ClusterIssuers and more. It also ships with a controller to manage all this resources.

All these are somehow generic operators as they are used to address a subset of generic business needs. A subset of generic business needs is simply a set of common business needs that apply to many different domains such as reacting to Apache Kafka messages or react to the execution of PostgreSQL query. These kinds of needs are transversal to many domains and thus were generalized by the people who develop these operators.

It is obvious that the more specific a business need is for an Operator, less probable it is usable in another domain. An Operator that reacts to high volume of shoe sales in an e-commerce application in order to activate a shoe related service in the system will most certainly be of

no use in an hospital's IT system. This illustrates how not every Operator for every need will be available on catalogs.

4.3 Operator Maturity Model

Operators can have different sets of functionalities depending on the needs we have for the operator. These features can be generalized into what is called the Operator Maturity Model. This model is shown in figure 4.4:

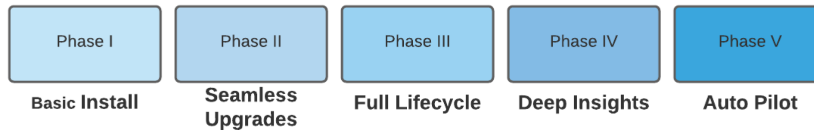

Figure 4.4: Operator Maturity Model

Both Phase I and Phase II of this model are capabilities present in simple Operators. That means Operators provide Automated application provisioning and configuration management and also support patch and minor version upgrades.[15]

More advanced Operators support application lifecycle and storage lifecycle (backup, failure, recovery) labeled Phase III.[15]

Operators that provide insights about applications they monitor are considered to be Phase IV Operators. Insights can be metrics about the applications that the Operator is monitoring or some form of analysis of its workflows.

The more mature Operators provide vertical and horizontal scaling of the systems they manage. This kind of Operators take full advantage of their environments and provide great domain knowledge to their automation that only someone from an operational team could. Now the overall systems can react to environmental changes and perform domain specific operational tasks.

4.4 Shared Informers

As per the kubernetes Go client source code, a *shared informer* "provides eventually consistent linkage of its clients to the authoritative state of a given collection of objects.", where shared means it can be consistently used by concurrent workers or "listeners". Each *informer* is responsible for a collection of objects from a particular API group and kind combination but can be further restricted to label, namespace, field selectors or any combination of them. They are important due to their ability to cache and create a eventually consistent reflection of the cluster's resources in memory. This cache is eventually populated on startup as a kind of "boot"work, to fill in the cache with the current state of the cluster.

If implemented in such a way that the *informer* is decoupled from the cache component, the same can be interchanged to meet the needs of individual applications. Consider for instance the need for a distributed in-memory cache such as Redis. If the *informer* logic is decoupled from the cache it uses we can extend it without modifying it, following the *Open Close Principle* of the SOLID Principles. It is important to note that when talking about decoupling the cache from the *informer* it is not implying that an *informer* and its cache component are two complete separate things. Consider an *informer* as the composition of the informing algorithm/structure with the cache that supports it and other components.

Following the same lines of good design every *informer* implementation should follow the same interface as the others. This allow for different implementations and optimizations to be used without requiring changes in the other components.

To wrap this overview of *informers* and tying it with this document's subject, an informer has another important component that has previously been described and that is a controller. A state arrives to this controller component, the *control loop* is executed as to move the system to the desired state, which makes it a controller by definition. This controller receives an event and will update its internal cache to reflect the cluster's state, thereby the "moving the system to the desired state".

4.5 Current Solutions

There are currently several solutions to help building *controllers* and *operators*. In this section some of these solutions will be briefly presented and discussed. These solutions help developers by removing most of the boilerplate code and allowing them to focus on the business logic. Michael Hausenblas and Stefan Schimanski [13] provide an interesting comparison between *kubernetes clients*, *Kubebuilder* and the *Operator Framework*. In addition to these, Metacontroller, another popular tool for building custom controllers, will be presented.

4.5.1 Kubernetes Clients

To write applications that make use of the Kubernetes REST API, there is no need to implement the API calls. There are available several client libraries for the programming language you are using. From Kubernetes' documentation there are currently six official clients and around 30 community-maintained clients, at the time of this writing.

The official clients are written for:

- Go
- Python
- Java
- .NET
- Javascript
- Haskell

These are maintained by *Kubernetes SIG API Machinery*, which is responsible for covering all aspects of API server, API registration and discovery, generic API CRUD semantics, admission control, encoding/decoding, conversion, defaulting, persistence layer (etcd), OpenAPI, CustomResourceDefinition, garbage collection and client libraries.

Some of the community maintained client libraries include:

- Clojure
- Go
- Python
- Ruby
- Python

- Haskell
- Node.js (TypeScript)
- Lisp
- Perl

And many more.

Building from simple client libraries that wrap the underlying REST client calls to the Kubernetes REST API lacks the desired bootstrapping for building *custom controllers* and *operators*. Fortunately, using the *client-go*, developers can make use of the *k8s.io/code-generator* to generate necessary functions such as typed clients, informers, listers and deep-copy methods.

When building operators following this approach there are two parts that need to be implemented in terms of business logic: The first part is *types.go*. Here is where the structure of the *CustomResource* is defined following the Kubernetes convention[18] (*spec* and *status*). The second part is the *controller.go* where the business logic will be implemented.

4.5.2 KUDO

KUDO stands for Kubernetes Universal Declarative Operator. It provides a declarative approach to building production-grade Kubernetes Operators covering the entire application lifecycle. [4] Unlike the other tools, KUDO utilizes a Universal and Declarative approach (the UD in KUDO), thus, it doesn't require any code.

KUDO has the concept of *plans*. *Plans* are documented procedures written in a way that allows KUDO to interpret and execute them. Each *plan* is made up of phases and each phase is made up of steps, which can be run either in series or parallel.

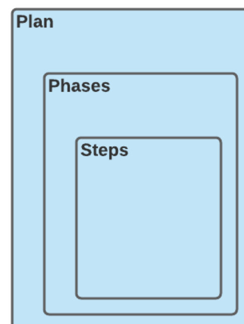

Figure 4.5: Conceptual view of a Plan with Phases which in turn have Steps

KUDO's official documentation [4] has a nice depiction which greatly describes its architecture. It's simplistic architecture provides valuable insights for other solutions. Put simply, KUDO defines three CRDs (Operator, Operator Version and Instance) and deploys one controller in the target cluster. They are responsible for offering the KUDO API to the clients and enable the deployment of the user's operators. At the low level an instance is an abstraction above the Kubernetes' *Pods* first-class citizens. The operators defined by the users are *packaged* inside a repository where the kubernetes CLI will fetch from.

These *packaged* operators follow the structure defined by KUDO:

```
1 .
2 |- operator.yaml
3 |- params.yaml
```

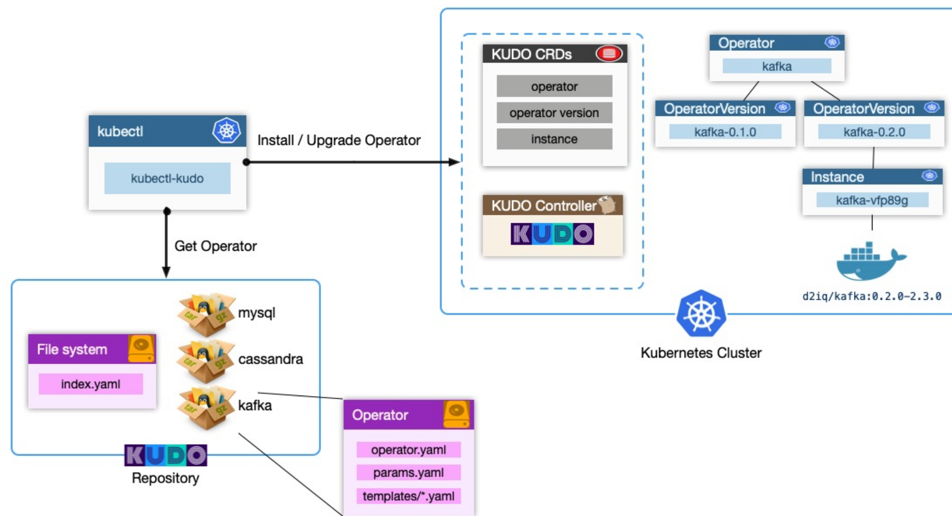

Figure 4.6: Architecure of KUDO

```
4 |- templates
5     |- deployment.yaml
6     |- ...
```

4.5.3 Operator SDK

The Operator SDK, developed by a team at CoreOS, is a set of tools for scaffolding, building, and preparing Operators for deployment. Although the SDK has plans to support more languages in the future it only supports Go, Helm charts and Ansible playbooks. [18] It is part of the Operator Framework, which is a toolkit for everything regarding Operators, from lifecycle management and metering to cataloging.

The Operator SDK comes with a CLI to help developers scaffolding its Operators avoiding all the boilerplate error prone code that come with Operators. Below is an example of the usage of the CLI to create an helm Operator.

```
$ operator-sdk new example-operator \
--api-version=example.com/v1 \
--kind=ExampleApp \
--type=helm
```

This command will create the folder structure required for an helm Operator, reducing greatly the effort required to bootstrap an Operator project. The Operator SDK will also generate all the RBAC resources it needs to operate in the cluster, controllers, CRDs and many other components.

All this, combined with a lot of other features, makes the Operator SDK the favorite tool for Operator development for a lot of developers in the community.

There are other projects that help the development of Operators mainly in Go. Its up to the developer to research and choose the best option according to their requirements.

What most of these solutions have in common is that their are used to develop in Go. This presents some challenges to enterprises as many have not adopted Go as their technology stack yet.

5

.NET Operator SDK

The .NET Operator SDK is an effort to expand the development of Kubernetes operators outside of the Go niche. [13] Even with the increasing popularity of Go, .NET has a much wider community as well as enterprise usage. The concept of operators was formalized by the teams at CoreOS that describe them as "a method of packaging, deploying and managing a Kubernetes application"[7] and was described in previous chapters.

Some of the concepts presented in this chapter are parallel to those already existent in solutions like the Operator Framework and Kubebuilder to keep consistency with these solutions.

5.1 The Current Problems

Most of the actual solutions to develop kubernetes operators are developed in Go. As previously stated, Go lacks some features that can reduce the amount of code written or hereby generated such as generics.

How does Go handle generalization? Through tooling such as code generation. This mindset is deep inside Go programming. Go projects make heavy use of tools to support development. This leads to having to generate the necessary code every time we make a change to our operators. [13] Consider for example, if a field is changed in a custom resource, it is required that we run the generator script. This code generation also happens with other Kubernetes clients such as the C# client. The Kubernetes project contains a repository for client generation. This leads to big interfaces to work with in most clients.

Generics could help with this issue, offering the ability to reuse the client code independent of the underlying type the client is handling. Reducing the amount of unnecessary code and regeneration needed.

Another problem with the current solutions was briefly mentioned previously and it is the fact that most existing tools for building operators are written in Go. Reality is that Go is still not widely used across enterprises. Those mostly use technologies such as .NET, java and nodejs and possibly using combinations of this same technologies and cannot afford to add another one to the stack. If you think about microservices, it is not absurd to agree that the maximum number of different languages in a microservices architecture should be 2 to 3. This will have several benefits down the line. If companies are already using, let's say C#, they should have the same ease to develop Kubernetes operators in that same language. It might even be a requirement imposed by the client. At the time of writing, someone that wanted to develop an Operator would have to use the generated C# client which is still in development and write all the boilerplate code to set it up.

The .NET Operator SDK provides flexibility by making it easy for developers to use .NET standard, including its external libraries, in their Operators.

5.2 Guidelines

The following guidelines were adapted to the current context and technologies from the guidelines written in the kubernetes community's github (https://github.com/kubernetes/community/blob/8cafef897
These guidelines will help make sure that good performance and desired results are met.[11]
This is a curated list of existing guidelines.

1. "Operate on one item at a time". For instance if developing an `InformerController` each worker is guaranteed to work on different items.
2. "Random ordering between resources. When controllers queue off multiple types of resources, there is no guarantee of ordering amongst those resources."
3. "Use `SharedInformers`". They also provide convenience functions for accessing shared caches and determining when a cache is primed.
4. "Operators should own only one CRD and one Operator should control only one CRD on a cluster".
5. "Operators shouldn't make any assumptions about the namespace they are deployed in".

5.3 Bootstrapping

The base repository offers an example Operator that can be used as the base for a simple *Watch Controller*. The generated Operator follows a simple structure that allow developers to be productive faster. Below in figure 5.1 is the structure of the example Operator:

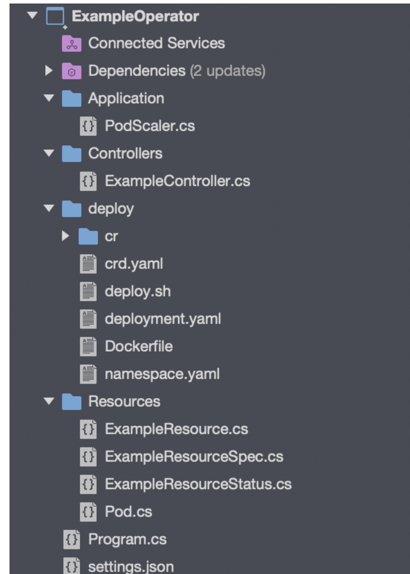

Figure 5.1: Operator project file structure

In the next sub sections the structure will be dissected to better understand the underlying components of an Operator.

5.3.1 Folder: Application

This folder includes only one file and it's here where everything starts. It contains the code that executes on the Operator's startup and stop routines. On startup it sets up the controller, subscribes itself as a worker in the *shared informer* and starts the shared informer. It also implements all the required methods.

5.3.2 Folder: Controller

Here is where the controller's logic begins. Similar to the controller's layer in the MVC pattern, this is the entry point for the business logic. All the controller's methods for each event (Create, Modify and Delete) are implemented in the controller using the corresponding method annotations *Create*, *Modify* and *Delete* respectively. There are also examples of how to use the *Exception handling* method annotation.

5.3.3 Folder: Resources

The last folder that is part of the Operator is the *Resources* folder. Here the Custom Resource that represents our Operator is translated into C# classes. There will be the base resource class that extends the *CustomResource<TSpec, TStatus>* generic class, the spec class, the status class and an implementation of *IDeltableObject* for the object we want to watch.

5.3.4 Folder: deploy

Finally, the deploy folder is where all the scripts and configurations required for the deployment of our Operator and its archetypes are located. There will be CRD and CR definitions, the deployment YAML configuration the deployment shell script and the Dockerfile to build the Operator's image.

This structure is what is provided in the example Operators in the .NET Operator SDK repository. It can be changed according to requirements and is by no means a static structure. It tries to use practices from DDD to allow for a more robust system that addresses the needs of the business and its stakeholders efficiently.

5.4 Reconciliation

As seen previously, the core of an Operator's logic runs inside its reconciliation loop. In the .NET Operator SDK we add logic to our reconciliation loop through the `SyncMethod` annotation for the *Shared Informer Controller*. For *Watch Controllers* there are three different annotations, each for different kinds of events. These are `CreateEventMethod`, `DeleteEventMethod` and `ModifyEventMethod`. There is another type of *Event Method* called `ErrorEventMethod` that will be explored in detail in the next section.

Consider for instance a controller the we want to log each time the reconciliation loop is called. The subject of this controller is an "Example"resource with a single spec field called "Example Field". The controller's method to log would look something like this.

```
1   public class ExampleController : InformerController<Example>
2   {
3       . . .
4
5       [SyncMethod]
6       public void Log(Example obj)
7       {
8           _logger.LogDebug($"This object has exampleField: {obj.Spec.ExampleField}]");
9       }
10
11      . . .
12  }
```

5.5 Error Handling

The .NET Operator SDK ships with its own annotations to handle errors in the reconciliation loop. Consider the following code snippet from a given Controller.

```
[ErrorEventMethod()]
[ExceptionHandler(typeof(OperatorException))]
public Task GenericOperatorExceptionResolver(OperatorException ex)
{
    _logger.LogError(ex.ToString());
    . . .
}
```

The `ErrorEventMethod` annotations informs the reconciliation loop that this is an error handling method. The loop will collect a reference to this method to execute when an error occurs.

This information alone is not enough for the reconciliation loop to have fine-grained control over the error handling. For this purpose the SDK offers a second annotation called `ExceptionHandler` which accepts an exception type. This annotation informs the reconciliation loop which exceptions should occur for this method to be executed. In the example above, the method `GenericOperatorExceptionResolver` is called when a generic `OperatorException` occurs. All method annoted with these receive an appropriate `Exception` instance object.

5.6 A Case Study

Back to the Mob Operator. The bootstrapped example informer Operator from the github repository has the structure necessary for the Mob Operator.

There needs to be a controller inside the `Controllers` folder. Consider a controller with the following structure:

```
public class MobController : InformerController<Mob>
{
   private readonly IKubernetes _client;
   private readonly CustomResourceDefinition _crd;
   private readonly ILogger<IKubernetesController> _logger;
   private readonly ISharedIndexInformer<Mob> _sharedIndexInformer;

   public MobController(
         IKubernetes client,
         ISharedIndexInformer<Mob> sharedIndexInformer,
         CustomResourceDefinition crd,
         ILogger<IKubernetesController> logger,
         IConfiguration configuration) : base(client, crd, sharedIndexInformer, logger,
             ↪ configuration)
   {
      _client = client;
      _sharedIndexInformer = sharedIndexInformer;
      _crd = crd;
      _logger = logger;
   }

   [SyncMethod]
   public void PerformMobLogic(Mob mob)
   {
      _logger.LogInformation($"{mob.Metadata.Name}␣-n␣{mob.Metadata.NamespaceProperty}␣[{
          ↪ mob.Spec.Replicas}]");

      // TODO: Implement some business logic here.
   }
}
```

The `MobController` is injected with an instance of an implementation of IKubernetes, an implementation of `ISharedIndexInformer` of type `Mob`, a `CustomResourceDefinition`, an `ILogger` and an `IConfiguration`. The constructor calls the parent's class constructor and stores the injected instances locally.

This is a very simple controller for demonstration purposes only. It has a single `SyncMethod` that logs some information and performs some business logic there.

That business logic can be anything as long as the controller has the access to the current and desired system's state through the event and the Kubernetes client. With this information the controller can perform its tasks that will integrate into the Operator's reconciliation loop. One example for such a logic could be comparing whether the replicas field corresponds to the number of nginx servers running a specific application. The method would have to compare those two states and see whether it needs to reduce the number of replicas in the cluster or whether it needs to spin a new one, all based on the triggered event and the current state of the cluster.

6

Operator Anatomy

This chapter will take a closer look at some of the inner workings of the .NET Operator SDK. To fully grasp the concepts in this chapter the reader should have some knowledge about data structures, object oriented programming, *reflection* and concurrency.

6.1 Reflection

At the heart of the .NET Operator SDK and the Operators developed with it is the use of *reflection*. Reflection is a feature of some programming languages which enable code to inspect and modify its own code or other code in the same system. As per the C# official documentation on reflection [6], "You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties". *Reflection* has to be used carefully as it relies a lot on casting and has performance problems associated. It is usually a good idea to implement some form of caching to reflection methods.

Thus reflection is very useful when we want to access some specific attributes in our program's metadata. This is what allows for the reconciliation loop to access user defined controller methods and error handling methods. Take for instance this snippet of code from the internal `GetExceptionResolvers` method.

```
1  protected IEnumerable<MethodInfo> GetExceptionResolvers(Type exceptionType)
2  {
3      return OfTypeUtils.GetMethodsWithAttribute(
4          this,
5          typeof(ExceptionHandler),
6          attribute => ((ExceptionHandler)attribute)
7              .ExceptionType.Equals(exceptionType)
8      );
9  }
```

This method uses one of the utility methods provided in the .NET Operator SDK to get the methods with a given attribute from the context which it is called. Inspecting the `OfTypeUtils` ↪ `.GetMethodsWithAttribute` and the `OfTypeUtils.GetAttributesOfType` static method we can see how it uses *reflection* to get the methods that have a given annotation, in this case, the `ExceptionHandler` annotation.

```
1  public static IEnumerable<MethodInfo> GetMethodsWithAttribute<T>(
2      T context,
3      Type attributeType,
4      Func<Attribute, bool> attributeFilter = null
5  ) {
6
7      IEnumerable<MethodInfo> methods = context.GetType()
```

```
8              .GetMethods();
9
10         if (attributeFilter != null)
11         {
12             return methods.Where(method => GetAttributesOfType(
13                 method,
14                 attributeType
15             ).Any(attributeFilter)).ToList();
16         }
17         else
18         {
19             return methods.Where(method => GetAttributesOfType(
20                 method,
21                 attributeType
22             ).Any()).ToList();
23         }
24     }
25
26     public static IEnumerable<Attribute> GetAttributesOfType(
27         MethodInfo method,
28         Type attributeType
29     ) {
30         MethodInfo ofTypeMethodInfo = typeof(Enumerable).GetMethod("OfType")
31             .MakeGenericMethod(attributeType);
32
33         return (IEnumerable<Attribute>)ofTypeMethodInfo.Invoke(
34             null,
35             new object[] { method.GetCustomAttributes() }
36         );
37     }
```

`OfTypeUtils.GetAttributesOfType` is an utility method that allows to retrieve all the attributes (annotations) of a specific type from the passed method. It executes the extension method `OfType` on the list of attributes of a given `MethodInfo` object. This method is used by `OfTypeUtils.` ↪ `GetMethodsWithAttribute`.

`OfTypeUtils.GetMethodsWithAttribute` checks if an attribute filter function is passed to it. If a function is passed it applies that same function to the list of attributes of the given type and returns the resulting list. If not, it simply returns the complete list of attributes of that type on the method.

This kind of logic can become really complex and that is why it is in a separate utility class. These will be used throughout the SDK several times. Keeping them in a separate static class reduces the amount of duplicated code.

6.2 Types of controllers

The .NET Operator SDK comes with different kinds of controllers to extend from based on the goal to achieve. For simple, lightweight work developers can use the *Watcher Controller* which is ideal for low activity jobs. Often the amount of work and activity in a cluster is too high to rely on simpler controllers and for such scenarios there is the *Informer Controller* which relies heavily on cache and indexing to achieve optimized results.

6.2.1 Watcher Controller

This type of controller simply uses a watch stream to get events from the *API Server*. A *Watch Controller* is the simplest controller regarding implementation out-of-the-box. It can

be configured to store a reflection of the cluster's custom resources the controller is watching active in the cluster and automatically synchronize with received *events*.

This solution is less performant than the Informer Controller but has a smaller footprint and works great for small scale operations. It is also simple to use and understand.

6.2.2 Informer Controller

The *Informer Controller* is composed of several components which include, a thread-safe FIFO-like structure that aggregates events to optimize the process of handling events, an indexer to serve as storage for keys identifying objects inside the controller, an informer and many more. Several concepts and solutions are adaptations of the code developed by the contributors of the Kubernetes ecosystem.

Its main component is the informer. Informers were described previously and the SDK's implementation follows that description.

The FIFO-like data structure has several characteristics that make it a solid fit for our informers. One is that it stores an accumulator that maps an identifier not to an object but rather to the *events* that arrived while the object sits in the queue awaiting processing. This accumulator forms a delta of the initial state of the object when it arrived to the informer and the final state. By storing all the intermediate values, ordered by arrival, a history is kept to provide value to the users. Depending on the objective the delta can be calculated as to have an insight in which properties changed by applying the delta formula.

$$\Delta O = O_f - O_i$$

Said accumulator can be optimized, and is by the current implementation, by checking if the new event is a *delete* event (or other considered terminal event) and the last received event is too such an event. In such cases the newly received event is discarded, as in the current implementation it doesn't make sense to store two terminal events sequentially. Other characteristic is the fact that the `Pop` method is blocking. This means the caller thread will block until an event arrives and is returned to the caller.

The figure 6.1 illustrates the behavior described previously when a delete event arrives to an object that already contains a delete event. Consider there is an informer controller running and it contains work to be done in the "work queue". While the controller's workers are working in their last dequeued object, an event arrives to an object already in the queue. That object has already three events in the "work queue". Two of those events are modify events and the other one is a delete event. The new event received for that object is also a delete event which is the same as the last event in that object's queue, thus it makes no sense to add it to the object's queue as it would be redundant.

This kind of optimization helps reduce the work on the controller by reducing redundant or useless operations.

Every *Informer Controller* makes use of a *Shared Index Informer* which is a singleton injected into the controllers that manages *events* in a optimistic concurrency strategy.

6.3 Kubernetes client

All this logic has to integrate with Kubernetes some how. For this, there is an open source client library for use with C#. The source code is maintained on Github under `kubernetes-client/csharp` and it's still in BETA version.

The client library allows to create clients using the default configuration file in a system which makes it really easy to get started.

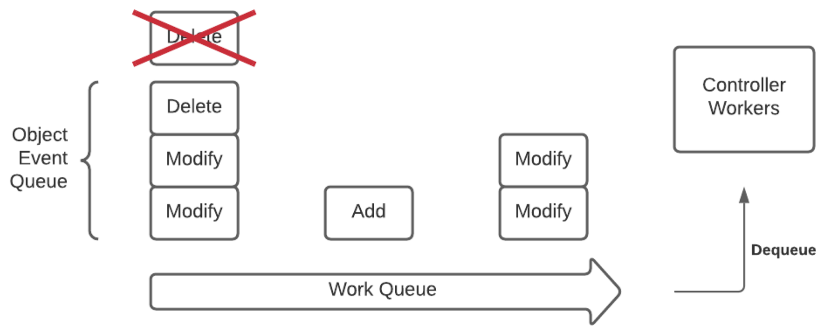

Figure 6.1: Informer controller's event optimization queue

```
var config = KubernetesClientConfiguration.BuildConfigFromConfigFile();

var client = new Kubernetes(config);
```

The .NET Operator SDK makes use of this library's simplicity to interact with the Kubernetes API. It is used as its infrastructure layer connecting the business domain layers of the Operator and abstracting the complexity of the infrastructure. The Operator's controller has a reference to the already instantiated client. This allows for our controller to react to an event and act upon that event by changing something in the cluster (this can be for example creating a *Pod*).

From the client's Github page:

```
var ns = new V1Namespace
{
    Metadata = new V1ObjectMeta
    {
        Name = "test"
    }
};

var result = client.CreateNamespace(ns);
Console.WriteLine(result);
```

This snippet shows how the client can be used to create a namespace in the cluster through the client.

If the controller is the "brain"of the Operator, then the kubernetes client is the "senses"that act as the "brain"commands.

6.4 Dependency Injection

The .NET Operator SDK follows several principles and patters. One of these is the Dependency Injection pattern which implements the famous Inversion of Control principle. Dependency Injection aims to make a class independent of its dependencies. This is specially important in complex applications such as Operators.

Operators can make use of Dependency Injections from .NET leveraged by the .NET Operator SDK. The `Operator` class provides a method called `ConfigureServices`. This method allows the registration of dependencies to the DI registry.

Consider the following snippet of code from an example controller.

```
var @operator = new Operator();

@operator.ConfigureServices((services) =>
{
    services.AddSingleton<IKubernetesController, SomeController>();
});
```

In the first line it initializes a new instance of `Operator`. After having the `Operator` instantiated, it calls the `ConfigureServices` method which takes a function as parameter. That function has an `IServiceCollection` as parameter. This collection will be added to the DI registry and those services will be injectable to our controllers and services as well.

This improves the overall extensibility of the Operator and makes its components more decoupled by following the Inversion of Control principle. By leveraging the .NET's dependency injection, the .NET Operator SDK lets developers take full advantage of this software pattern.

6.5 The Operator Class

The main class that represents the Operator per se is the `Operator` class.

We've seen briefly how we can initialize an Operator object previously. It is a simple operation designed to abstract any initialization complexity that would be otherwise bound to happen.

This `Operator` class offers two very important methods. The first interesting method is the `ConfigureServices` method. As seen previously it allows to configure services to be available to the Operator's DI.

The method's signature is as follows:

```
public void ConfigureServices(Action<IServiceCollection> configuration)
```

It takes a method as parameter that in itself receives an IServiceCollection as parameter. In turn, when the operator starts it will have all the services registered using this method available for injection.

An example of how this method can be used is shown below.

```
@operator.ConfigureServices((services) =>
{
    services.AddSingleton<ILogger, Logger>();
    services.AddSingleton<IConfiguration, Configuration>();
});
```

The second most important method available in the Operator class is the `Start` method. This method starts the Operator's lifecycle. Internally the operator will create an `HostedService` ↪ which is a custom implementation of the `IHostedService` interface from Microsoft. After creating an instance using the default builder methods, the hosted service is configured with the essential services for the Operator to function. Only after configuring the essential services are the user specific configurations applied. Those configurations were added by using the previously mentioned `ConfigureServices` method.

Finally, after having everything setup, the host is built and ran. As support for creating more detailed Operators this class provides more methods to support those tasks. An example of this is the `AddInformerServices`' generic method. This method takes a type parameter that extends `CustomResource` and `IDeltableObject`.

6.6 The Hosted Service

When talking about the Operator class in Section 6.5, it was briefly mentioned an instance of `HostedService`. The `HostedService` class implements the `IHostedService` interface from Microsoft. As per the official documentation, "The IHostedService interface is the basis for all long running services in .NET"[8]

This interface requires two methods to be implemented, `StartAsync` and `StopAsync`. The following code snippet is the complete definition of `IHostedService` interface.

```
1  using System.Threading;
2  using System.Threading.Tasks;
3
4  /// <summary>
5  /// Defines methods for objects that are managed by the host.
6  /// </summary>
7  public interface IHostedService
8  {
9      /// <summary>
10     /// Triggered when the application host is ready to start the service.
11     /// </summary>
12     /// <param name="cancellationToken">Indicates that the start process has been aborted.</
           ↪ param>
13     Task StartAsync (CancellationToken cancellationToken);
14
15     /// <summary>
16     /// Triggered when the application host is performing a graceful shutdown.
17     /// </summary>
18     /// <param name="cancellationToken">Indicates that the shutdown process should no longer
           ↪  be graceful.</param>
19     Task StopAsync (CancellationToken cancellationToken);
20 }
```

The .NET Operator SDK's implementation of the `IHostedService` is quite simple. It simply starts the controller registered in the DI when the Hosted Service starts. When stopped it sends a cancellation signal to the controller's process for it to start shutting down. This signaling is achieved through `CancellationToken` class.

This chapter offered an in-depth view of how the .NET Operator SDK works and how it is implemented. Note that the features presented in this chapter are still prone to change as this is an overview over a MVP. The last chapter will go through the future of the .NET Operator Framework and which of these features are planned to be improved.

7

Results

This document presents the current state of the .NET Operator SDK. Its available as an MVP closed to the public.

It will be distributed as an NuGet package. NuGet is the package manager for .NET the same way lstinlinenpm is the package manager for NodeJS.

7.1 Development Environment

The development of the .NET Operator SDK did not went well at first. Due to the high cost of maintaining a Kubernetes cluster and having no more free tier on the main cloud providers, the school was asked if they could provide any means to run a Kubernetes cluster such as a VM or a managed solution. The request was not answered and so, in order to develop in a production-like environment, a bare-metal Kubernetes cluster was built.

The build consists of 2 nodes and one persistence unit (a 256GB SSD). Figure 7.1 is a picture of the built cluster.

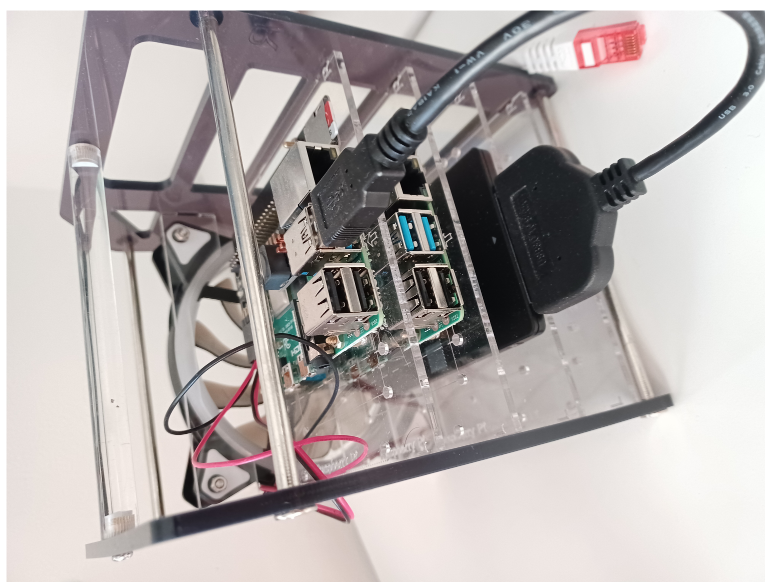

Figure 7.1: Cluster built of 2 nodes and a persistence unit

It has 8GB RAM across both nodes, Quad core 64-bit SoC at 1.5GHz each and Gigabit ethernet. The persistence unit is attached to the master node where workloads requiring more critical persistence are ran. Each node is provided 5V to operate through a PoE adapter. The power is provided to the nodes through a PoE supporting TP-Link switch.

Both nodes and the persistence unit are layered in simple server rack with an attached fan to dissipate the heat mainly produced by the PoE adapter. The whole hardware running is shown in figure 7.2.

Figure 7.2: Development environment of the .NET Operator SDK

7.2 Tests

The .NET Operator SDK follows a semi TDD approach. It makes use of unit testing to help ensure that the code is of quality and does not break with changes. It makes use of the famous NUnit package for unit tests.

There are around 25 unit tests focusing on particular features and data structures based on complexity and importance. They help guarantee that changes are retrocompatible with previous high-importance features.

Throughout the development of this project several "Smoke Tests"were performed as new features for the MVP were being added. What also helped in testing those features were the example controllers available in the repository. The On-Premises Kubernetes cluster introduced in Section 1.3 was also crucial to test that the Operators were working.

7.3 Architecture

This section describes the high level architecture of the .NET Operator SDK. It's a complex system developed with extensibility in mind.

One common pattern followed throughout the development is the use of interfaces as a mean of decoupling the system. Most pieces can be easily replaced or their functionality extended.

Other pattern found in the .NET Operator SDK is the Factory pattern. Consider the `KubernetesClientFactory` ↪ class. It provides static *factory methods* to create `IKubernetes` instances from environment variables or configuration files. It allows to move the client creation code into one place in the project, making the code easier to support. Finally it is possible to introduce new ways of creating a client without breaking existing code.

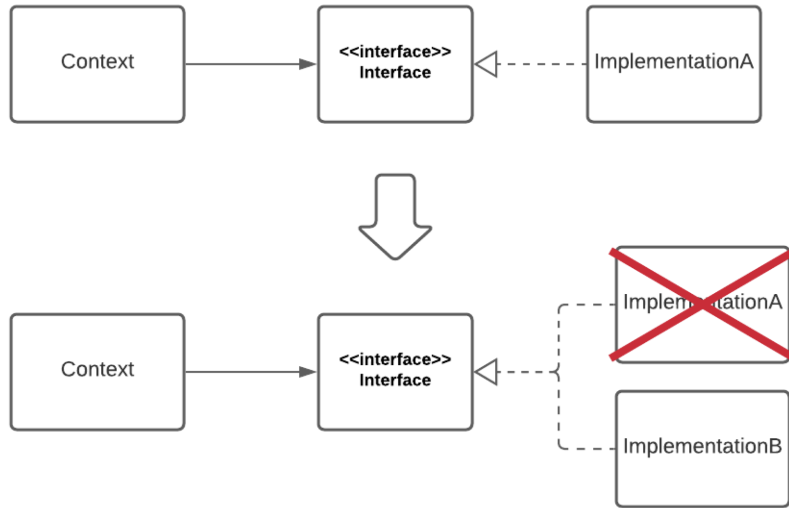

Figure 7.3: Implementations being changed without affecting the context

7.3.1 Package Diagram

This section introduces the package diagram of the .NET Operator SDK to show how its inter-dependencies are layed out.

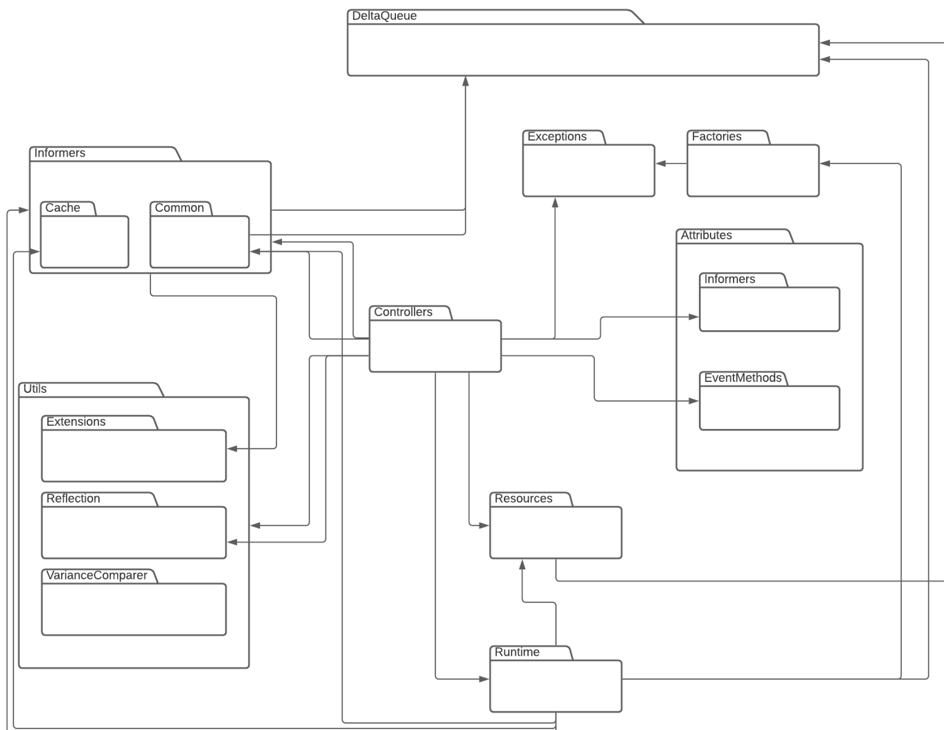

Figure 7.4: .NET Operator SDK package diagram

These are the different packages that together make up the SDK. The central piece is the lstinlineControllers package, where most things are put together and managed. The utilities package serves the purpose of aiding users with complex complementary tasks.

7.4 The Case Study

The purpose of the simple case study built in the previous chapters was to help the reader grasp the concepts behind the .NET Operator SDK and how to use it. The final result should look something like figure 7.5. It is a simplification of the components inside an Operator.

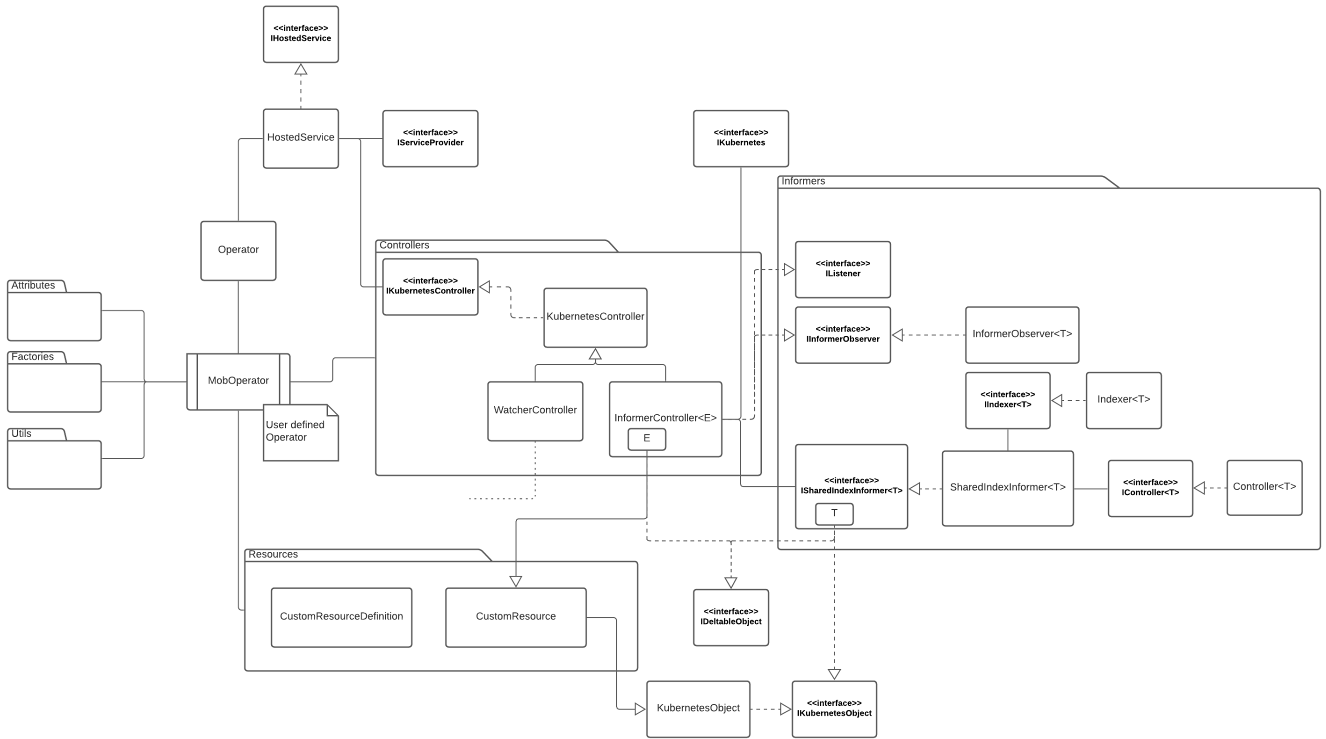

Figure 7.5: Class diagram showing the relevant components of the case study's Operator

The diagram shows how the different components and packages of the .NET Operator SDK are connected to give power to its Operators. One thing that can be visibly noted is the use of a lot of interfaces mentioned previously. This is what makes its components decoupled and allow for extending functionalities without great overhead. This modular architecture is leveraged to benefit the future of the SDK and to make the most out of future contributions. Components can easily be changed as long as they implement the proper interfaces.

The result is a flexible, future-proof, easy to maintain codebase.

It all starts with an `Operator` object, C# *attributes*, *factories*, a *controller* and some *resources*. It makes use of some utility functions as well.

The case study in previous sections can be used to help achieve a functional Operator. The diagram in figure 7.6 illustrates an Operator running in a Kubernetes cluster and managing both an internal and an external subject.

These are the building blocks that make the MobOperator built throughout the previous chapters.

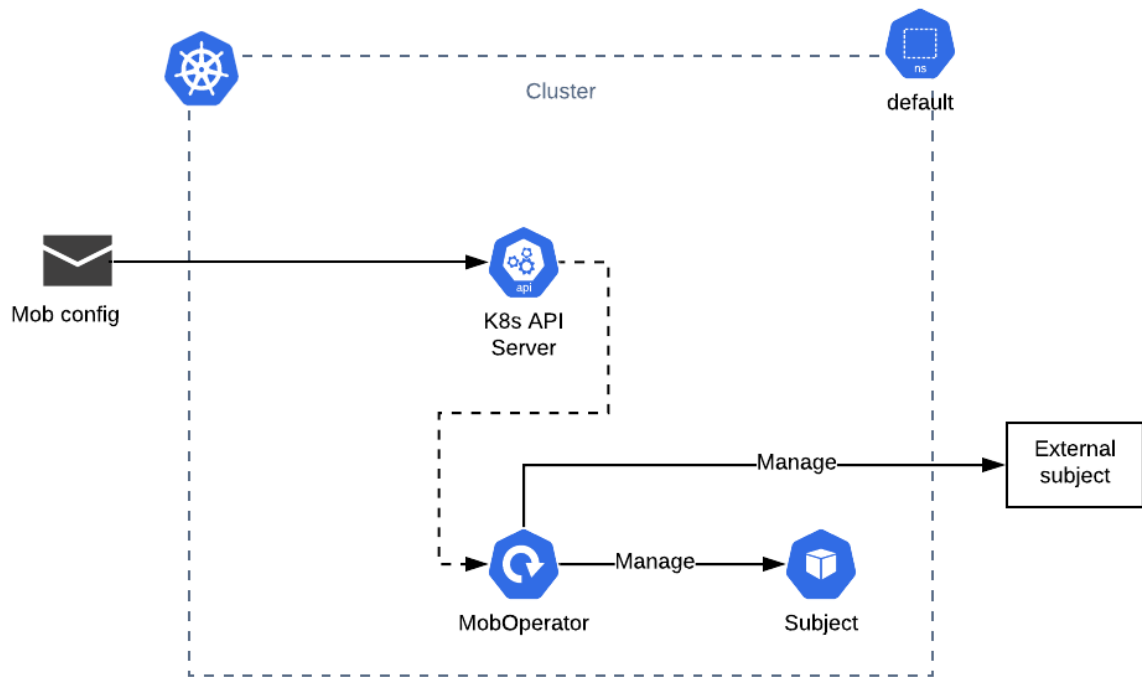

Figure 7.6: Operator deployed in a kubernetes cluster that manages internal and external subjects

8

Conclusion and Future Work

Enterprises have an ever growing need for automating its processes. Concepts like Continuous Integration and Continuous Delivery are two widely adopted techniques for managing and delivering code that focus on the automation (but not limited to) of building, testing and releasing code. Kubernetes automates the scheduling and execution of application (running inside containers) across a cluster in an efficient manner. With the .NET Operator SDK users are able to develop Operators to automate business operational tasks using .NET which is a framework widely used in enterprises. Developers should be able to create operators using the .NET Operator Framework MVP presented in this document.

8.1 Future Work

The .NET Operator SDK is still in development and is not released to the public. The main milestone to achieve is the release of an enhanced MVP open to the public for use.

To achieve this milestone there are some important features to implement that were not the scope of this MVP.

8.1.1 Bootstrapping CLI

One of the future features is the release of a CLI script to generate the complete project structure. The purpose of this CLI is to allow developers of Operators to get started faster and generate all the boilerplate code required to have the Operator up and running. This is more an utility and user experience feature and thus was not the focus of the this MVP.

Below is an example of the user of such cli:

```
$ dotnet-operator-cli new ExampleInformerController --controller-type=informer

$ ls
Controllers                             Resources
ExampleInformerController.csproj        settings.json
Program.cs                              deploy
```

As is exemplified, the command would create the necessary folders structure as well as all the boilerplate code. The user would then only need to implement the domain specific logic for the Operator to work.

8.1.2 Further optimizations

For a public release of the .NET Operator SDK the code needs to be optimized and refactored.

Good contributions can only be expected if the codebase is in a state which developers can easily grasp how it works and contribute easily. To achieve this the code must be of good quality and follow widely used and accepted patterns a principles. The .NET Operator SDK was developed with these principles in mind where possible.

To address the need for fast performing applications of enterprise systems, a publicly available .NET Operator SDK needs to be very optimized. That level of optimization was not the scope of this MVP. This does not mean that the MVP lacks optimized code, on the contrary, the code is already optimized in certain components.

8.1.3 Open Source Contributions

It is the vision of the .NET Operator SDK to be an open-source project to allow contributions from all over the world. This will give the project a life of its own and allow the community to have influence in its development. The project is hosted in a private Github repository and will be made public as soon as it is stable enough for a public release.

Upon public release the repository will be made public and pull requests will start to be accepted.

8.1.4 Further field testing

The .NET Operator SDK has to be further developed to be production-ready and be used in production enterprise applications safely. There are currently some users of the MVP that were hand picked to test the features and possibilities in their projects.

Future work includes testing the SDK by developing and deploying Operators in different projects and evaluating its performance and usage. One metric that can also be collected in such testing scenarios could be the development speed, the ease of debugging defects and how error prone it is.

Currently in the backlog for the public release are a few tasks that will be presented in this further subsections. The backlog is hosted on Github and is currently private.

8.1.5 Unit and Integration Testing

One of the things to improve in the future are the overall testing environment of the .NET Operator SDK. It lacks testing in part due to time and in part due to not this being an MVP. This will be important in the future as more and more contributors start pushing code to the repository.

8.1.6 CI/CD

Continuous Integration and Continuous Delivery, in short CI/cd, are essential for fast delivery of modern software products.

Currently the .NET Operator SDK NuGet package is deployed using a simple Github Actions pipeline. This does not represent the vision for the project's CI/cd pipelines. The aim is to use a third party SaaS such as Travis-CI or Jenkins.

8.1.7 Telemetry

Telemetry is one of the next features to be implemented before public release. The .NET Operator SDK will provide two different ways to offer insights into the health of Operators.

The first one is through health checks. Health checks usually are made to a specific endpoint with the single purpose of providing insights to external parties about the health of a system.

Kubernetes uses such mechanisms to address the liveness and readiness of its workloads so it makes sense that the .NET Operator SDK provides such functionality.

Another way that is planned to provide insights about the health of Operators is through what is usually called heartbeats. Heartbeats are signals sent by a system to a listening agent at a particular rate. When a system misses several heartbeats it might mean that something is wrong with it and something should be done to address what is not working properly. Usually the processes to handle these rely on external resources such as alerts or monitoring dashboards. Figure 8.1 illustrates both behaviors.

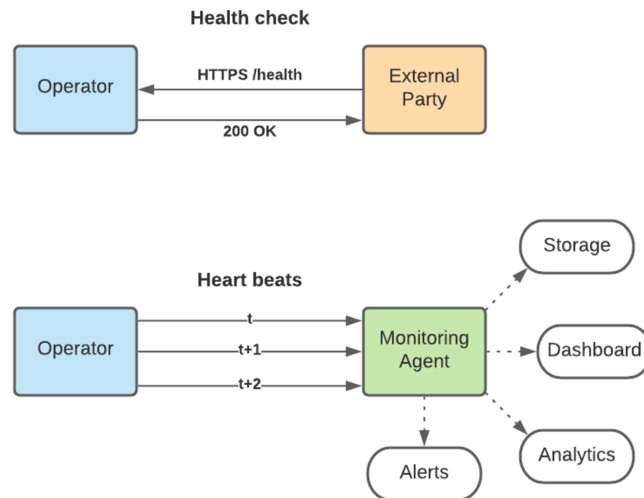

Figure 8.1: Health checks and heart beats example

Bibliography

[1] URL: http://man7.org/linux/man-pages/man2/chroot.2.html (cit. on p. 2).

[2] URL: http://man7.org/linux/man-pages/man7/namespaces.7.html (cit. on p. 3).

[3] URL: http://man7.org/linux/man-pages/man1/ps.1.html (cit. on p. 3).

[4] URL: https://kudo.dev/docs/ (cit. on p. 18).

[5] A. Avram and F. Marinescu. *Domain-Driven Design Quickly*. C4Media, 2006 (cit. on pp. 7, 8).

[6] BillWagner. *Reflection (C#)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection (cit. on p. 27).

[7] CoreOS. 2020. URL: https://coreos.com/operators/ (cit. on p. 21).

[8] IEvangelist. *Implement the IHostedService interface*. URL: https://docs.microsoft.com/en-us/dotnet/core/extensions/timer-service (cit. on p. 32).

[9] Kedaorg. *KEDA Concepts*. URL: https://keda.sh/docs/2.4/concepts/ (cit. on p. 15).

[10] A. Khan. "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application". In: *IEEE Cloud Computing* 4.5 (2017), 42–48. DOI: 10.1109/mcc.2017.4250933 (cit. on p. 3).

[11] Kubernetes. *kubernetes/community*. URL: https://github.com/kubernetes/community/blob/8cafef897a22026d42f5e5bb3f104febe7e29830/contributors/devel/controllers.md (cit. on p. 22).

[12] E. Marquez. 2018. URL: https://linuxacademy.com/blog/linux-academy/history-of-container-technology/ (cit. on p. 2).

[13] M. H. S. Schimanski. *Programming Kubernetes, Developing Cloud-Native Applications*. O'Reilly Media, Inc, 2019 (cit. on pp. 13, 17, 21).

[14] Tdykstra. *.NET introduction and overview*. URL: https://docs.microsoft.com/en-us/dotnet/core/introduction (cit. on p. 5).

[15] *Understanding Operators*. 2019. URL: https://docs.openshift.com/container-platform/4.1/applications/operators/olm-what-operators-are.html (cit. on p. 16).

[16] *Using RBAC Authorization*. 2021. URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (cit. on pp. 10, 11).

[17] *What is .NET? An open-source developer platform*. URL: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet (cit. on p. 5).

[18] J. D. J. Wood. *Kubernetes Operators, Automating the Container Orchestration Platform*. O'Reilly Media, Inc, 2020 (cit. on pp. 9, 10, 18, 19).