KubeSharper

An SDK for Building Kubernetes Operators in C#

Master Thesis

Valér Orlovský

AALBORG UNIVERSITY
STUDENT REPORT

Aalborg University
Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science
Aalborg University
www.aau.dk

Title:
KubeSharper: An SDK for Building Kubernetes Operators in C#

Theme:
Distributed Systems, State Reconciliation, Kubernetes, Infrastructure-as-Code, C#/.NET

Project Period:
Spring Semester 2020

Project Group:
ds109f20

Participant(s):
Valér Orlovský

Supervisor(s):
Brian Nielsen

Page Numbers: 90

Date of Completion:
June 11, 2020

Abstract:

The thesis addresses the lack of library support and overall low accessibility of implementing custom state reconciliation use-cases on Kubernetes when working within the confines of the C# language and the .NET platform. A solution in form of a C# Software Development Kit named KubeSharper was implemented and evaluated by building two representative applications. Results indicate that, while further research and testing is required, the delivered solution appreciably reduces the complexity involved in discussed implementations.

Summary

This thesis addresses the lack of supporting tools and libraries in the C#/.NET ecosystem when it comes to implementing integrations and leveraging the Kubernetes system as an engine for automatic state reconciliation, including but not limited to infrastructure automation purposes.

Chapter 1 (Introduction) established the initial motivation for this thesis. After reflecting on the nature of modern infrastructure practices, the concept of state reconciliation was introduced as the pattern of synchronizing a representation of a desired state with the actual, existing state of a system.

Afterwards, the benefits of using Kubernetes as a core engine for applying the pattern were presented, followed by pointing out the limited accessibility as well as the potential value of writing such implementations using C#/.NET. An initial problem statement was then formulated, related to exploring how these implementation may be made more accessible for C#/.NET developers.

An overview existing projects related to this problematic was given in chapter 2, followed by the first part of the report which aimed to provide the background and knowledge for the concepts explored in the thesis.

In chapter 3, the properties and challenges of modern infrastructures were expanded upon and chapter 4 dived into a specific, popular practice - Infrastructure-as-Code. Then, the definition and details of the state reconciliation pattern were given in chapter 5, followed by a deep dive into the Kubernetes system and the details regarding how it embraces, as well as enables state reconciliation.

In chapter 7, after presenting the background, several possibilities of addressing the initial problem statement were discussed and considered. Based on the discussions, a solution and a specific problem statement were proposed, making the main purpose of the thesis to design an implement a software development kit in C#, for implementing custom state reconciliation in form of Kubernetes controllers. Furthermore, the attempted solution was titled KubeSharper and the functional and non-functional requirements were described.

Next, Chapter 8 provided a design of the solution and its components, inspired by and party based on the design and concepts from an comparable solution - the Controller Runtime library written in Go. Then, based on the design an implementation of the solution was written and significant areas of the components and their code were discussed and presented in-detail.

Furthermore, the implemented solution was tested and evaluated in chapter 10, where it was applied to several representative use-cases. A qualitative review showed that the solution overall met the defined requirements by providing the necessary functionality, level of abstraction and developer experience. Several points were also made regarding shortcomings and areas of improvement of the solution.

The qualitative review was then additionally supported by a quantitative analysis of the code written for the evaluated use-cases. There, the number of lines of code not relating to the main use-case were contrasted with the lines of code needed to configure KubeSharper and Kubernetes integration. The relatively low amount of KubeSharper code further demonstrated the noticeably level of abstraction provided by the SDK.

Finally, the overall outcome of the thesis was concluded in chapter 11, deeming the attempted solution mostly successful, but at the same time expressing the need for and proposing potential areas of future work.

Preface

Acknowledgements

I would like to thank my supervisor, Assoc. Prof. Brian Nielsen, for the advice, guidance and direction that I have been given throughout the semester.

Valer Orlovsky
[vorlov18@student.aau.dk]

Table of contents

CHAPTER 1

Introduction

In the last decade, we have seen a radical shift in how software and IT systems are being developed, hosted, delivered and scaled. The ever increasing need to be able to support web-scale usage of applications, rapidly adapt to fluctuations in user traffic and minimize the cost of managing IT infrastructure has paved the way for widespread adoption of cloud technologies.

As exemplified by internet-scale tech companies like Netflix or Zalando, software development approaches and methodologies have transformed from plan-driven deliveries of large monolithic applications happening a few times a year to systems of hundreds of single-purpose services, each independently and continuously deployed, often several times a day.[30][10]

However, despite the inherent flexibility of cloud computing, the increased complexity of application stacks and platforms still warrants innovation of the underlying application infrastructure and the processes involved. Application and infrastructure operators and teams are facing challenges in managing the infrastructures that are supposed to support hundreds, even thousands, of applications at once. In such scenarios, traditional approaches to automation, such as using purpose-built, imperative scripts, are proving to be hard to manage and scale.

Rise of Infrastructure-as-Code and state reconciliation

Instead, automation tools that have recently been gaining popularity aim to follow the Infrastructure-as-Code (IaC) principle. According to this principle, the overall configuration and shape of the infrastructure and core services is to be defined using (typically declarative) code. Then, using clever tools, this code, representing the defined image of a *desired state*, can be automatically converted into the right instructions and API calls, resulting in provisioned resources that perfectly match the desired state.

This process of synchronizing a definition of the desired state with the *actual* state is called *state reconciliation* and several modern infrastructure and cloud automation tools embrace it, some of the most popular being Terraform[1] and the container-based platform Kubernetes[2].

Potential in Kubernetes

Kubernetes specifically, has started being favoured for infrastructure automation and state reconciliation use-cases. This technology, which is currently the most widely used platform for managing container-based applications, has been designed around the concept of state reconciliation from day one.

The entire architecture can be thought of as a system of small and specific reconciliation loops coordinating via a shared state store. Additionally, extensibility is a core aspect in the design of Kubernetes.

By combining these two features, Kubernetes is also being used as a generic state reconciliation engine, capable of reconciling custom, application-specific resources and states, which can even be external to Kubernetes itself, such as cloud provider resources or virtual data-center appliances.

Limited accessibility

In order to leverage these features, applications need to handle the often non-trivial details of communicating and integrating with the Kubernetes API correctly and efficiently. Furthermore, with Kubernetes being still a relatively young project, only a limited number of libraries attempting to address this complexity seems to be available.

Since Kubernetes itself, similarly to many other container and cloud-native technologies, is written in the Go language, so are the most mature integration and extension libraries. These can certainly and often significantly improve the development experience, that is to say, as long as the code in question can be written in Go.

While that may be a possibility for a decent portion of developers and/or companies, the fact still remains that there are also many who would benefit from more advanced Kubernetes abstraction libraries without having to sacrifice established standards and tools. In fact, according to most major programming language popularity rankings, despite an admirable adoption rate given the age of the language, Go's market share is only around 1.3% [44][2].

[1] https://terraform.io
[2] https://kubernetes.io

The case for C#/.NET

The C# language and the .NET platform are a good example of an ecosystem whose popularity exceeds that of Go. Yet, its accessibility story around Kubernetes integration is lacking. According to the rankings, the market share of C# is between 4-7%, which amounts to roughly 3 to 5 times the market share of Go [44][2].

This can be attributed to multiple factors. Historically, the then closed-source, single-platform .NET framework, has for a long time served as the main development platform of companies embracing the Windows operating-system and Microsoft technologies which remain prevalent in many enterprises.

Nowadays, the modernized, open-source and cross-platform .NET Core (soon to be named just .NET) provides a level of efficiency that is on-par with other popular platforms, while at the same time allowing teams and companies to benefit from the access to a large pool of .NET talent and pre-existing software.

Furthermore, C# includes several established, state-of-the-art language features such as a generic type system which many (especially enterprise) companies have grown to rely on and which are unfortunately as of now not present in the Go language. With Kubernetes starting to see adoption in the enterprise, there is therefore a valid case for improving the experience and ease of integration from C#/.NET.

1.1 Initial problem statement

Given the motivation presented above, the initial question that this thesis will explore is the following:

> *How can Kubernetes as a platform for implementing custom state reconciliation be made more accessible to teams and companies that rely on and/or want to leverage the C# language and the .NET platform?*

CHAPTER 2

Related work

This chapter introduces some of the existing projects in the Kubernetes ecosystem which have been researched for this thesis and have served as a basic and/or inspiration for design and implementation.

2.1 Kubernetes Go Client

The Kubernetes Go Client, also known as `client-go` i(based on the name of the open-source repository on GitHub), is the official Kubernetes client for the Kubernetes-native Go language.

The Go client represents the oldest and most mature Kubernetes client library, due to the fact that it is being used internally in the components of Kubernetes itself, which also means it is well-tested and reliable. [16]

The library encapsulates several abstractions and smaller packages that facilitate implementation of Kubernetes integrations and communication with the Kubernetes API. There are three packages that are relevant for basic Kubernetes API communication [16]:

- The *kubernetes* package, which provides static (and statically typed) client that can be used for performing operations against the Kubernetes API that involve native Kubernetes resource types
- The *dynamic* package, which provides a dynamic client capable of performing operations

genericallly against any resource types (native and custom)

- The *transport* package, which helps with low-level transport details when communicating with Kubernetes such as establishing a connection using valid authentication, etc

Additionally, since the client is being used throughout the Kubernetes codebase, including complicated scenarios, it also comes with additional tools, utilities, objects and abstractions for simplifying Kubernetes integration.

Given the subject of this thesis, which relates to state reconciliation and custom Kubernetes controllers, the most notable functionality of the client are implementations of the *Informer* pattern. [43]

Informers are abstractions over the real-time (watch) functionality of the Kubernetes API where the API can notify consumers with any change events regarding any object in the cluster. They provide an interface which allows developers to efficiently establish the mentioned change stream connections for a particular Kubernetes resource type.[43]

This represents crucial functionality for implementing custom controllers, which are based on constantly observing and reacting to resource-related changes.

2.2 Kubernetes Controller Runtime

As discussed in the previous section, the `client-go` library provides many abstractions which can simplify implementation of Kubernetes controllers and state reconciliation. Nevertheless, the library is meant to be a general client and it does not directly address this use-case.

Other projects, such as the Operator SDK[1] and Kubebuilder[2], provide an even higher level of abstraction. They are targeted specifically towards Kubernetes API extension developers and designed with custom controllers in mind.

While the two projects represent two fully-featured, opinionated frameworks/toolkits, they are both built on top of a common, core codebase, known as the Controller Runtime [13]. It is a set of libraries which together represent a common model for extending Kubernetes with custom reconciliation logic.[3][13]

The runtime builds on top of the `client-go` packages (section 2.1), extending it with the useful concepts and APIs for building controllers, which among others include[13]:

- A high-level *Client* for reading and writing Kubernetes objects

[1] https://github.com/operator-framework/operator-sdk
[2] https://kubebuilder.io/

- The *Cache* for efficient fetching of Kubernetes objects
- The *Manager* for sharing dependencies and starting controllers
- The *Controller*, which represents the core abstraction of listening and responding to Kubernetes API events in order to reconcile state based on declarative definitions stored in Kubernetes objects
- The *Webhook* for extending the object-admission process of the Kubernetes API
- The *Reconciler* representing the reconciliation function to execute based on events
- The *Source*, meant to facilitate and encapsulate Kubernetes events streams

The fundamental abstractions and concepts introduced by the Controller Runtime library have been a significant inspiration and basis for parts of the design and implementation of Kube-Sharper, which will be present further in the report.

2.3 Kubernetes C# Client

As mentioned in the Introduction and also illustrated by the previous two sections, writing custom controllers for Kubernetes is widely supported by multiple libraries in the Go language ecosystem.

When it comes to other languages, the state of libraries varies, but the support is generally more limited. Apart from Go, the Kubernetes project provides official client libraries for five other languages. In all five cases, the provided libraries are automatically generated based on the OpenAPI specifications [42] of the Kubernetes API. [23]

The client library provided for the C# language, which this thesis focuses on, is hosted in the *kubernetes-client/csharp*[3] repository on GitHub. According to the classification defined by the Kubernetes community [14], its level of capabilities is rated as *Silver*.

This means that the client support basic (bronze) capabilities, such as loading Kubernetes configuration files and performing basic authentication and HTTP communication, as well as the more advanced (silver) support for using the Kubernetes API WATCH.

2.3.1 Limitations

While the C# client provides useful APIs for basic integration with the Kubernetes API, including creating, fetching and modifying objects and even utilizing the watch functionality, it comes with several limitations.

[3]https://github.com/kubernetes-client/csharp

Low-level watch API

First of all, the watch functionality is implemented as a quite low-level abstraction. While the API facilitates establishing a watch connection with the Kubernetes API, many additional aspects must be managed by the developer.

For example, the watch API does not implement any mechanism for automatically re-establishing the connection in case it is dropped or closed. This has to be addressed and handled by the developer with custom code.

Lack of a generic client

The second and most significant shortcoming of the C# client is the lack of generic API. Generic types, i.e. higher-level types which accept type parameters, are a widely used language feature, first introduced with the version 2.0 of the language in 2006. They allow developers to implement data structures and services that can be reused with values of different types while still maintaining type-safety.

Due to the automatically generated nature, the C# client does not utilize generic types. To illustrate this, consider client APIs for retrieving two resource objects, one of type `Pod` and one of type `Service` from the Kubernetes API.

In a generic API, a method could be expose on the client such as `Get<T>(string name)` which could then be used in both cases, i.e. `Read<Pod>(...)` and `Read<Service>(...)`.

In the C# client's non-generic API, on the other hand, two separate methods must be used: `ReadPod(...)` and `ReadService(...)`. Given that Kubernetes supports hundreds of object types, this significantly impairs the development experience. Since a separate method must be used for each combination of operation and object type, more branching is required and therefore complexity is increased.

Despite these limitations, the C# client provides the basic transport logic, and therefore represents a valuable base library for C# Kubernetes integrations, which is why it is going to be revisited further in this project.

CHAPTER 3

Dynamic infrastructure

Since its inception, the IT industry and its related practices have undergone rapid and radical evolution. In the past, most software services used to be highly coupled to its infrastructure which was composed of a set of physical hardware devices, such as bare-metal server machines with dedicated hard-drives and network interfaces and appliances.

To initially provision, as well as operate such an infrastructure involved extensive planning and manual work which entailed careful and individual installation and configuration of each device and appliance. It was typical for such configuration to be highly custom, tailored specifically to the software workload that it had been planned to host. The custom, catered nature of such a system meant that the only feasible way to scale it was to add additional resources (to scale *up*).[34]

In other words, using the popular analogy originally used by Bill Baker, Distinguished Engineer at Microsoft [1], servers and infrastructure components have in the past been treated as *pets* - unique, indispensable, manually "raised" and individually cared for.

Nowadays, the underlying physical hardware tends to be more decoupled from applications and workloads, with multiple layers of abstraction in between. The infrastructures, platforms and services behind modern applications are highly *dynamic and automated*. Software is often deployed several times a day to a selection of virtualized compute resources, such as virtual machines or containers. These resources are homogeneous and generic. [34]

Provisioning and maintenance is handled automatically instead of requiring hands-on involvement from an administrator. When a server is faulty, there are no attempts to repair it. Rather, the faulty server is destroyed and a fresh, identical server is spun-up. Instead of adding resources to the dedicated server(s) (scaling up), the homogeneity allows applications to be scaled by adding more server instances (scale *out*).[34]

In Bill Baker's analogy, the IT systems of today are considered *cattle* - managed as a group, for all intents and purposes identical to each other and easily replaceable. In this thesis, we will refer to this kind of infrastructure using the term *dynamic infrastructure* as used by Kief Morris in the book *Infrastructure as Code*.[34]

3.1 Characteristics

The concept of dynamic infrastructure can be further defined by providing a set of characteristics that should be consistent across many if not all dynamic infrastructure providers and systems.

3.1.1 Cloud computing

First of all, when talking about dynamic infrastructure, it is essential to describe the concept of cloud computing. Although dynamic infrastructure is a slightly broader concept, cloud computing providers and platforms represent the majority of instances of dynamic infrastructure.

Cloud computing represents one of the technologies that allowed businesses and teams to move into the paradigm of treating infrastructure components as dynamic, replaceable "cattle" instead of static, unique "pets". As described by Barrie Sosinsky, *cloud computing represents a real paradigm shift in the way in which systems are deployed* and *makes the long-held dream of utility computing possible with a pay-as-you-go, infinitely scalable, universally available system*.[40]

Furthermore, National Institute of Standards and Technology (NIST) of the United States Department of Commerce provides the following five essential characteristics of a cloud[31]:

- *On-demand self service*. A customer has access to request infrastructure resources to be provisioned immediately, without requiring human interaction.
- *Broad network access*. The service provided by the cloud is accessible over the network and via standard protocols and mechanisms, enabling access from a multitude of computer clients.
- *Resource pooling*. The cloud provider or platform uses resources that are pooled together and can be reassigned or relocated dynamically without the user's knowledge or involvement. The cloud provider's system has to feature multi-tenant capabilities in order for resources to be provided securely and reliably.
- *Rapid elasticity*. Provided resources are elastic, can be manually or automatically added, removed or resized within a short time-frame and appear limitless to the customer.

- *Measured service*. The customer's usage of resources is precisely monitored, often at an hourly or lower granularity. This data is reported to the customer and used for billing purposes.

3.1.2 Dynamic infrastructure platforms

As already alluded to, the term dynamic infrastructure aims to further generalize cloud-like platforms. According to Morris, while clouds are naturally designed as dynamic infrastructure platforms, there can be other types of environments which do not strictly follow the characteristics of a cloud. Simpler virtualization or hardware based platform can still exhibit similar dynamic properties, but do not necessarily need to use, for example, service metering, or sometimes even resource pooling. Morris therefore defines an alternative, more general, yet similar set of characteristics, stating that a dynamic infrastructure platform which has to be *programmable*, *on-demand* and *self-service*. [34]

Programmable

A dynamic infrastructure platform needs to be *programmable*, meaning it needs to enable headless software and scripts to programmatically interact with it using a remote API with an (optional) accompanying set of one or more software libraries or development kits.[34]

This requirement is similar to the *broad network access* characteristic of cloud computing (section 3.1.1), which further emphasizes the use of standard protocols to implement such APIs.

On-demand

Contrarily to NIST's cloud computing characteristics, Morris recognizes more nuance in the on-demand and self-service aspects and separates them.

The *on-demand* requirement, similarly to NIST's definition, expresses the need for a dynamic infrastructure platform to *allow resources to be created and destroyed immediately* without resorting to costly and lengthy processes such as service tickets. [34]

Self-service

Finally, the *self-service* requirement serves as an extension to *on-demand* and emphasizes the need for resources to be highly customizable by the users of the platform, on top of being easy to create and destroy. Users should be able to use the platform to fully tailor relevant resources to their specific use-case. [34]

Both the characteristics of cloud computing provided by NIST, as well as the requirements of dynamic infrastructure platforms from Morris illustrate the nature of modern infrastructures, highly influenced by the cloud computing. For the purpose of this thesis, Morris' three requirements provide a more suitable framework for talking about dynamic infrastructure in relation to automation and management from the perspective of the users (or teams).

3.2 Challenges

Dynamic infrastructures are a necessary evolution in management of IT systems. The approach is powerful and makes it possible to build infrastructures for applications at a scale that would not be possible to manage using traditional, manual strategies. However, the dynamic nature of this approach combined with its reliance on safe and efficient automation inherently comes with added complexity and therefore new challenges for IT engineers.

Morris recognizes the following six closely-related challenges and problems that teams often encounter when starting with dynamic infrastructure and the automation involved [34]:

- *Server Sprawl*. Dynamic infrastructure providers and systems often strive to make it simple, often trivial, to provision new resources on demand. While this significantly improves provisioning, it can result in servers and resources being created at a pace that is too fast, making slower activities like resource organization, maintenance, patching, upgrades, etc. challenging.
- *Configuration drift*. As a result of server sprawl, servers and resources tend to end up in inconsistent states. For example, server packages may be manually and reactively updated on a specific server as a result of an incident without rolling out the update consistently on all servers.
- *Snowflake servers*. Gradually, if not addressed, configuration drift may build up on a server (or other resource), resulting in a similar situation as illustrated in the beginning of this chapter - a component which is unique ("pet") and hard to reproduce.
- *Fragile infrastructure*. As the snowflake server problem spreads throughout the majority of the inventory of servers and resources, different instances in the infrastructure start requiring individual knowledge and treatment. This makes managing the infrastructure exceedingly difficult and introduces risk in performing management activities.
- *Automation fear*. As the level of fragility of the infrastructure increases, the confidence and ability of the operating team to use automation tools decreases. This is due to the fact that automating common tasks on servers requiring specific and individual instructions increases the branching factor and therefore the overall complexity of the automation logic.
- *Erosion*. This term, also known by other names like *bit rot* or simply *software entropy* refers to the fact that even without intervention (as illustrated in the case of configuration drift), the

> states of different servers will still inevitably drift apart due to common entropic forces like software failures, maintenance and upgrades.

Ultimately, the challenges described by Morris revolve around the dichotomy between the so-called "Day 1" and "Day 2" operations on this type of platforms. Dynamic infrastructure and cloud computing enable modern environments and use-cases by abstracting away and simplifying the "Day 1" operations, i.e. the provisioning of resources and therefore allowing for elasticity and scalability. On the other hand, "Day 2" operations, i.e. the remaining activities in the lifecycle of an infrastructure component, are often only facilitated and automated by the platform to a limited extent, which can be detrimental to teams and environment without adequate processes, practices and tools in place.

To conclude this chapter, it is clear that the adoption of cloud computing and dynamic and comes with many benefits due to the provided potential and capabilities and can even be necessary for many use-case. However, the added complexity of these platforms introduces challenges. Strategic and efficient use of automation capabilities is therefore necessary in order to correctly utilize them. The following chapters will describe and explore some of such approaches.

CHAPTER 4

Infrastructure-as-Code

Having described the properties and inherent challenges of dynamic infrastructures in the previous chapter, this chapter will introduce the practice called infrastructure-as-code (IaC). The practice represents a viable and popular approach for efficiently utilizing the potential of dynamic infrastructure as well as tackling its complexity and challenges.

4.1 Definition

Infrastructure-as-Code stems from the realization that the lifecycles of modern infrastructures are becoming increasingly similar to software applications. Components of modern infrastructures are more abstract. It is possible to provision and change them on-demand and immediately, which means the rate of iteration and changes is increased as well.

Morris defines IaC as *an approach to infrastructure automation based on practices from software development*, focusing on *consistent, repeatable routines for provisioning and changing systems and their configuration*. [34]

Using IaC, every aspect of infrastructure is defined in one or more files using some form of code. With that rule in place, processes can be devised which leverage automation tools in order to automatically provision resources or apply changes to the infrastructure based on the specification defined in the code file(s).

In a typical infrastructure-as-code workflow, as depicted in fig. 4.1, in order to make a change,

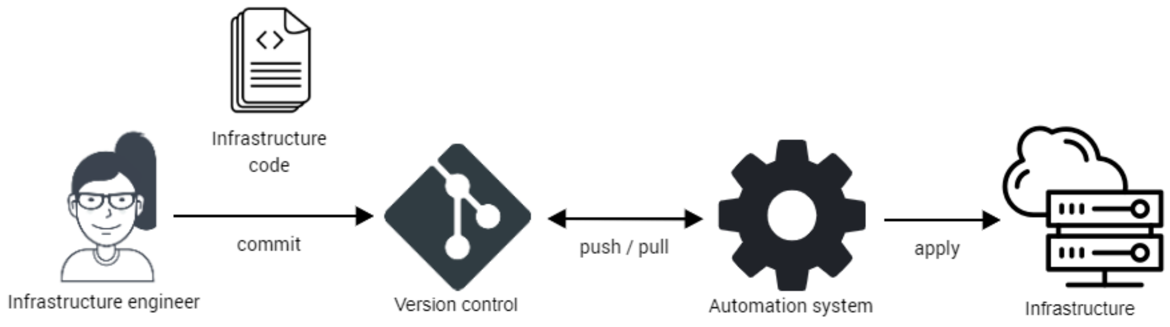


Figure 4.1: Illustration of infrastructure development workflow using IaC [7]

an infrastructure engineer would express the changes in a file containing the infrastructure code. Similar to application code, the code can then be committed to a repository in a version control system (VCS). Afterwards, the code is either pushed to or pulled by the automation system, which subsequently uses the file together with platform-specific integration functionality in order to apply the described changes in the infrastructure. [7]

4.2 Core practices

The infrastructure-as-Code approach encapsulates several practices inspired by software development. This section will introduce the three practices that are in this thesis considered as fundamental: *definition files*, *self-documentation* and *version control* and discuss their benefits.

4.2.1 Definition files

Using definition files to describe infrastructure is at the very core of infrastructure-as-code. Following more traditional approaches infrastructure resources are usually defined using an automation system's graphical interface and stored in its database or are not strictly defined at all and are simply documented in diagrams and specification documents. Following infrastructure-as-code, all aspects and resources of an infrastructure are defined as code, that is, as text files. [34]

There are several benefits gained by using definition files. First of all, they allow for describing the infrastructure resources precisely and in great level of detail. Moreover, changes to the definition can be done using a text editor of choice, which is often faster and simpler than using a graphical interface. Finally, definition files can help make the infrastructures more consistent and reusable, as the textual definitions can be copied with minimal adjustments to fit the new use-case. Alternatively, depending on the IaC tool and language used, higher-level programming constructs such as templates and functions can be used to simplify such process even further. [34] [7]

The usage of definition files directly enables the remaining two practices.

4.2.2 Self-documentation

Self-documentation can be considered as both a practice as well as a direct benefit of using infrastructure-as-code and definition files.

In traditional approaches, change implementation and documentation are two separate processes. This often results in outdated or non-existent documentation, as it starts proving challenging to keep the documentation up to date with frequent changes. By defining the infrastructure in a definition file using precise and detailed code, the code and files automatically act and can be used as documentation for the infrastructure. [34][7]

4.2.3 Version control

Finally, describing infrastructure using code and files enables one of the most widespread practices used in software engineering, namely, *version control* to be used for infrastructure.

Using a version control system (VCS) like Git, a codebase can be organized and versioned in a code repository. Every change made to the code has to be committed into the version control system which in return keeps track of the history of all the changes and supports operations for viewing different states (snapshots) of the codebase, at different points in the history. Thus, the code repository also serves as the source-of-truth for the code.

This ultimately provides several benefits to any codebase, including infrastructure code. First of all, having a single source-of-truth for all infrastructure definitions improves collaboration and general visibility of changes, as the history log can act an easy-to-use time-ordered overview of changes made. Furthermore, this also allows for traceability and auditability, as every change can be traced to a commit in the VCS, which usually holds information about the person making the change and also a description or justification for the change. Lastly, the history log and the operations of a VCS can also help in a rollback scenario where infrastructure needs to be reverted to some tested, safe state.

4.3 Tools and paradigms

In order to correctly and efficiently manage actual infrastructure according to definition files, automation tools or systems are required. This section presents and overview of infrastructure-as-code tools and discusses the two main programming paradigms used.

4.3.1 Types of tools

While the currently landscape of IaC and related tools is quite vast, we can can group them into the following four categories [7].

Scripting tools

Using common operating-system scripting tools and languages such as Bash, Powershell, Python, Perl, etc. represents the simplest approach for managing infrastructure using code. While they are often sufficient for simple tasks, these tools generally do not scale well in more complex scenarios. These tools inherently use an imperative approach.[7]

Configuration management systems

These represent a more fully-featured set of systems such as Chef[1], Puppet[2] or Ansible[3], which are usually used to manage servers in a generic way. These tools tend to directly communicate with servers using standard or specialized remote connection protocols and agents in order to install and configure software. In general, such tools tend to use specific concepts and terminology and tools with both imperative and declarative can be found.[7]

Provisioning tools

This type of tools usually provide a higher level of abstraction and allow users to create, modify and delete resources of dynamic infrastructure platforms. The most well-know examples of these are AWS CloudFormation[4] and Terraform[5]. Both embrace a declarative approach, with CloudFormation using JSON and Terraform using a custom, domain-specific configuration language (DSL) for their definition files. While CloudFormation is specific to the AWS platform, Terraform, on the other hand, can be used on numerous different platforms, as it can be extended with custom providers for virtually any platform that meets the dynamic infrastructure requirements (section 3.1.2). Alternatively, most dynamic platform also provide imperative-style provisioning tools in form of command-line tools or software libraries such as the AWS[6] or Google[7] SDKs.[7]

[1]https://www.chef.io/
[2]https://puppet.com/
[3]https://www.ansible.com/
[4]https://aws.amazon.com/cloudformation/
[5]https://www.terraform.io/
[6]https://aws.amazon.com/tools/
[7]https://cloud.google.com/sdk

Container-based systems and orchestrators

These are typically full dynamic infrastructure platforms designed with IaC principles in mind. Examples of these are Docker[8], Kubernetes[9], Nomad[10] and others. The core is to provide an infrastructure-level abstractions, like compute, storage and networking. However, the intrinsic features of containers enable IaC. Containers encapsulate the entirety of the a single application's environment, which can be defined using code (e.g. Dockerfile) and packaged into an immutable image. Furthermore, container orchestration systems (e.g. Kubernetes, Nomad) further embrace IaC by allowing for all resources in the system to be defined declaratively.[7]

4.3.2 Imperative vs. Declarative

As alluded to in the previous sub-section, there are two main programming paradigms generally used by IaC tools and systems: *imperative* and *declarative*.

Using the *imperative* paradigm (e.g. scripting or Chef), the code in the definition file is essentially a set of instructions to be executed in a specific order to achieve the desired state of infrastructure. In other words, imperative users require the user to describe not only the desired infrastructure resources but also, at least to a certain extent, how and in which order should the configuration be applied.

With the declarative approach, the code is meant to be simpler and specialized for defining a structure of resource objects, list and hierarchies. In this case, users are not required to know how resources should be provisioned on a particular platform and often not even in which order. Those concerns are abstracted away and delegated to the IaC tool itself.

Both styles come with advantages and disadvantages. To to the emphasis on the individual instructions to be performed, the imperative paradigm can be considered a more powerful approach that can support even the most complex, special configurations. The declarative approach, on the other hand, is less powerful and usually limited by the capabilities of the IaC system used. However, from the user-perspective, files written declaratively provide a simpler reading and writing experience by not requiring the user to have deep knowledge about how changes to the infrastructure should be performed. Furthermore, they also allow for differences between different states of infrastructure (files) to be spotted and analyzed faster and with less effort.

As described in section 3.2, having many specialized and inconsistent configurations is more akin to treating infrastructure as pets instead of cattle, making management more challenging and less scalable, and should generally be avoided if possible. This makes the value proposition of imperative tools far less appealing when compared with declarative tools which welcome the "cattle approach" and facilitate standardization, consistency and reusability.

[8]https://www.docker.com/

[9]https://kubernetes.io/

[10]https://www.nomadproject.io/

4.4 Example: Definition of an AWS network using Terraform

To illustrate using a declarative infrastructure-as-code tool, this section describes how to define a simple network configuration on AWS using Terraform. The network configuration that will be provisioned can be seen in fig. 4.2 and consists of an AWS VPC (virtual private cloud - a virtual network) with a set of three subnets located in different availability zones to ensure high availability.

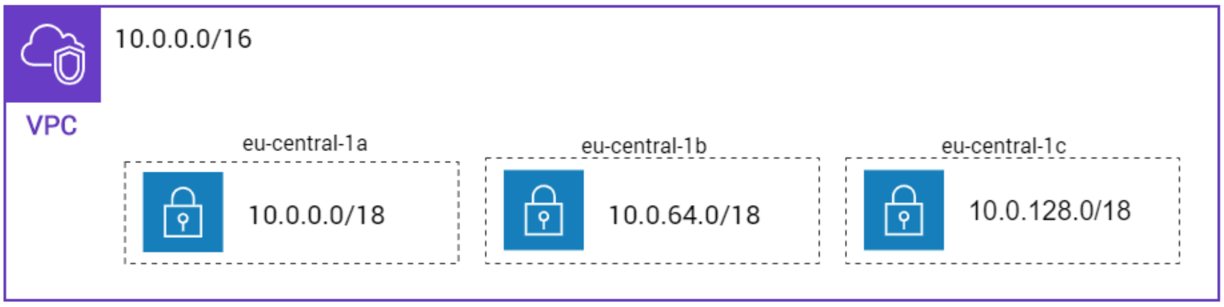


Figure 4.2: Sample AWS VPC configuration with three subnets located in three availability zones

In Terraform the network from fig. 4.2 can be defined using HCL, the HashiCorp Configuration Language, a declarative language built with configuration purposes in mind, similar in syntax to JSON (JavaScript Object Notation) but with additional declarative and functional features.

The relevant HCL/Terraform code may be seen in listing 1 on page 20. There, on lines 1-3, the AWS provider for Terraform is specified, which allows provisioning of AWS-specific resources such as, in this case, an AWS VPC (the `aws_vpc` resource) and an AWS subnet (the `aws_subnet` resource). Each resource instance is defined as a block that starts with the `resource <resource_type> <resource_identifier>` syntax, as seen on lines 10,15,21 and 27, where `<resource_type>` must be provided by one of the configured providers (in this case, only the `aws` provider is configured). Inside each resource block are assignments of resource attributes (specific to a given resource type).

The order of all the resources and other code blocks is purely for the purpose of readability and organization within the code and does not influence the actual order in which the resources will be created/updated. Terraform will automatically analyze the definition, build a directed acyclic graph of resource dependencies, as seen in fig. 4.3. In this case, each `aws_subnet` resource depends on the `aws_vpc` resource, due to the reference to its ID (lines 16,22 and 28).

The Terraform configuration may be applied to the infrastructure (an AWS account) using a `terraform apply` command. The implementation details of the AWS provider will inspect the existing infrastructure on the AWS account and ensure that the correct (imperative) AWS API operations are performed so that the desired infrastructure state, as defined in listing 1 on page 20, will be achieved.

As argued in section 4.3.2, the abstract and declarative approach of Terraform allows the user to simply define the set of resources and their attribute as required for the use-case, without having to be concerned about the intricacies and details of AWS tools and APIs.

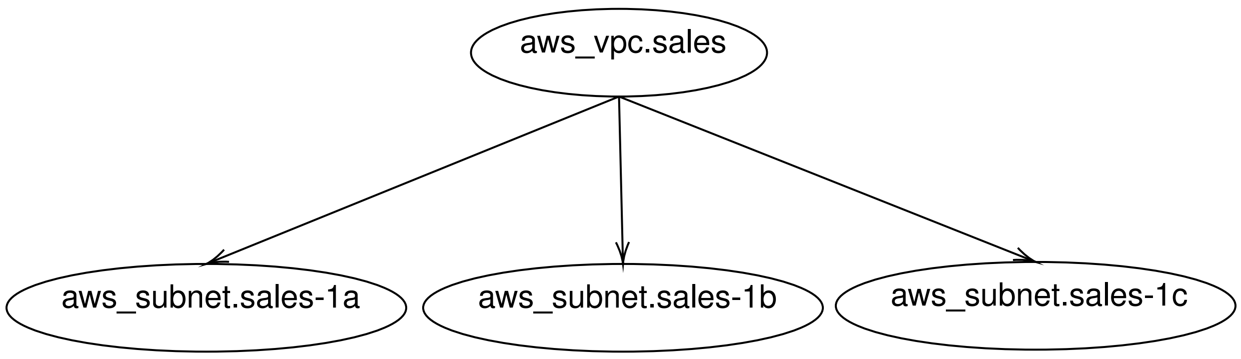


Figure 4.3: Terraform dependency graph for the code included in listing 1 on page 20

4.5 Summary

In this chapter, the concept of Infrastructure-as-Code (IaC) has been presented and described together with the involved practices and benefits. Different types of tools that support the practices were introduced and the imperative and declarative programming paradigms were discussed in the context of IaC, arriving at the conclusion that the declarative paradigm represents a more suitable approach for tackling dynamic infrastructure challenges. Finally, to illustrate declarative IaC and its benefit even further, a simple, yet practical example of using the Terraform tool in an AWS cloud environment was provided.

```
provider "aws" {
  region = "eu-central-1"
}
locals {
  tags = {
    "owned-by-team" = "sales"
  }
}
resource "aws_vpc" "sales" {
  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true
  tags = local.tags
}
resource "aws_subnet" "sales-a" {
  vpc_id = aws_vpc.sales.id
  cidr_block = "10.0.0.0/18"
  availability_zone = "eu-central-1a"
  tags = local.tags
}
resource "aws_subnet" "sales-b" {
  vpc_id = aws_vpc.sales.id
  cidr_block = "10.0.64.0/18"
  availability_zone = "eu-central-1b"
  tags = local.tags
}
resource "aws_subnet" "sales-b" {
  vpc_id = aws_vpc.sales.id
  cidr_block = "10.0.128.0/18"
  availability_zone = "eu-central-1c"
  tags = local.tags
}
```

Listing 1: Terraform (HCL) code definition for the network configuration from fig. 4.2

CHAPTER 5

State Reconciliation

In this chapter, we will explore a pattern that is common across most declarative infrastructure-as-code tools and systems and discuss its variations and uses from a general perspective. The pattern in question will be referred to as *state reconciliation* throughout this report.

5.1 From desired state to actual state

As was illustrated in the previous chapter, especially with the Terraform example in section 4.4, one of the essential features of declarative infrastructure as code tools is the ability to take a description of the *desired state* (e.g. particular set of infrastructure resources) and automatically modify the *actual state* (e.g. configured objects/services in an AWS account) so that it would reflect the desired state.

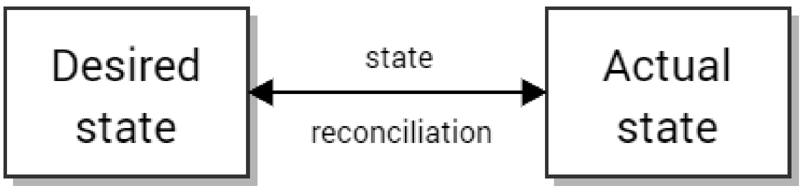
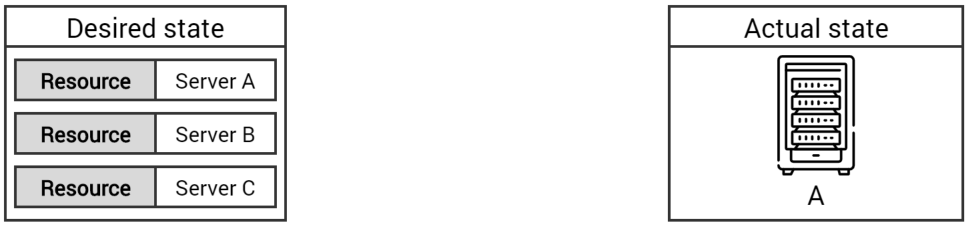


Figure 5.1: State reconciliation

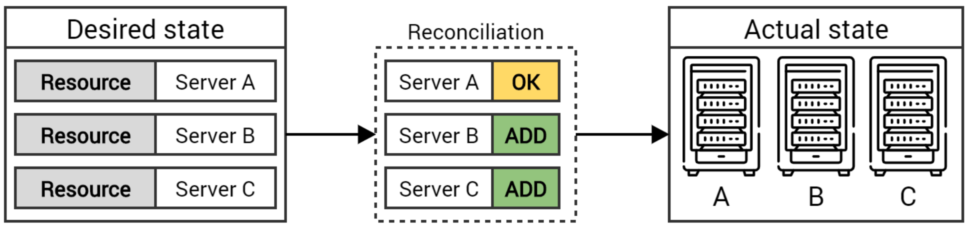
This pattern is in fact *state reconciliation*, which in general terms can be defined as a *the pro-*

cess of making a target state consistent with a source state. For the purposes of this thesis, state reconciliation will be considered mostly in the context of management of software applications and infrastructure, and therefore, in most cases the *source* state will be regarded as the desired, user-defined state, as per IaC, and the *target* state will be regarded as the state of the platform, infrastructure or application that is being managed.

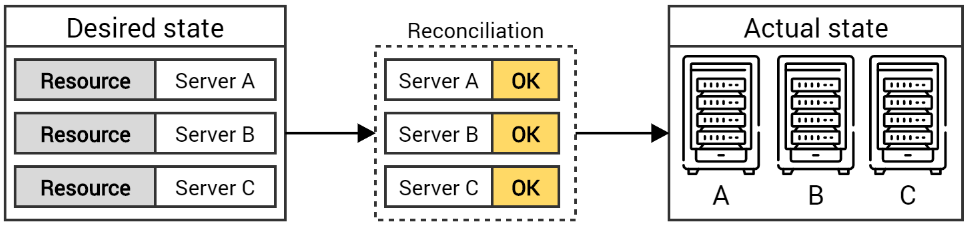
The concept can be further explained with some example scenarios of different instances of desired and actual state being reconciled. Scenarios using servers as example resources may be seen in fig. 5.2 below.



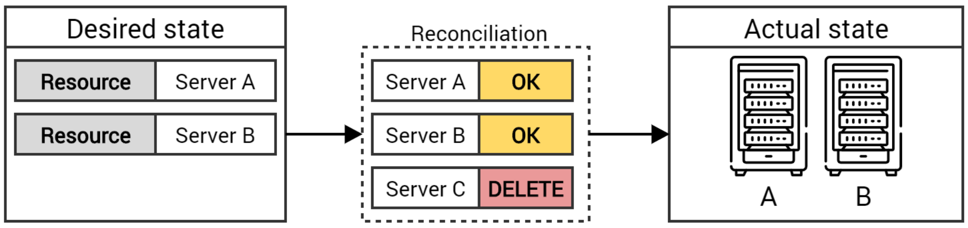
(a) Initial states. 3 servers configured, 1 provisioned.



(b) First reconciliation. Missing servers B and C. Server A untouched.



(c) Second reconciliation. No modifications needed. Actual state already consistent



(d) Third reconciliation. Server C removed due to change in desired state.

Figure 5.2: Example scenarios of state reconciliation

First of all, fig. 5.2a shows the initial states: three servers (A,B,C) have been defined in the desired state, while only server A exists in the actual infrastructure. Figure 5.2b illustrates the reconciliation process for the initial states. To make the actual state consistent with the desired one, the reconciliation process recognizes that server A already exists but servers B and C are missing and need to be created. After the discovered actions are performed (e.g. using the given platform's API) the actual state becomes consistent with the desired state.

Furthermore, fig. 5.2c illustrates the scenario where reconciliation is attempted again, while the states are still consistent. In this case, the reconciliation process recognizes full consistency and therefore no modifications are needed.

Finally, in the scenario from fig. 5.2d, server C has become redundant and has therefore been removed from the desired state. This constitutes an inconsistency, making the reconciliation process recognize that server C should not longer exist, resulting in only servers A and B remaining in the infrastructure.

5.2 Continuous state reconciliation

So far, the mentioned examples of state reconciliation such as the Terraform example from section 4.4 or the general one from the previous section have illustrated a particular flavour of state reconciliation. In these cases, the reconciliation process is triggered on-demand (e.g. manually by a user, or automatically as a result of a new change in infrastructure code) and performed once until completion. Many infrastructure-as-code tools, including Terraform, use this approach.

However, this has a shortcoming in that the achieved consistency of the two states (desired and actual) is only ever really true right after the reconciliation process finishes. At any point, certain events, such as manual changes in the infrastructure which bypass IaC, might lead to inconsistency and drift between the two states. With this one-off approach, the resulted inconsistency is not addressed until the next time the reconciliation process is triggered, which might not happen until the next time a change is made.

More advanced systems, such as the reconciliation features in Kubernetes (more on that in chapter 6), are able to continuously attempt to ensure consistency by constantly observing and reacting to the states in a *reconciliation loop*. This approach will be referred to in this thesis as *continuous state replication*.

5.2.1 Techniques

A basic technique is to simply execute a new one-off reconciliation process periodically, on each tick of a timer. While naive, this solution might be suitable, especially in environments with a lower rate of changes. However, the more frequent changes are in an environment, the shorter the reconciliation interval would need to be in order to react to changes in a timely manner. This may have load and performance implications at very tight intervals.

An improvement would be to execute the reconciliation process in an event-based manner, as a reaction to events indicating a change has happened. An event on the desired state side could be for example a new change in version control, if an IaC workflow such as the one from fig. 4.1 is being used. Moreover, the reconciliation system also needs to receive and react to events about resource changes in the infrastructure (the actual state). By reconciling as a reaction to events from both sides (the desired and actual state), the reconciliation process only runs when it is needed which can significantly improve performance and decrease load on APIs.

Whether the event-based approach can be implemented depends on the platform behind the actual state, which must support producing events about changes to its resources. The Kubernetes system supports this natively (see chapter 6), which is one of its main selling points.

5.3 GitOps

GitOps is a trending example of both continuous state reconciliation and infrastructure-as-code. Similar to some of the use-cases described earlier, it is an approach that uses the Git version control system as the source-of-truth. State reconciliation is used for quickly and continuously deploying applications, as well as infrastructure resources.

The Git repository in this case represents the desired state. In order to deploy changes to applications and infrastructure based on the resources in Git, a controller component is running, continuously watching definition files (section 4.2.1) in the repository and applying them to the platform (actual state). [8] [37]

5.4 Beyond infrastructure

While infrastructure management represents the main, original use-case for infrastructure-as-code tools and the state reconciliation mechanism, the application of these practices can be extended to management of other types of resources that are not normally thought of as IT infrastructure. The requirements of dynamic infrastructure platforms from section 3.1.2 are considered as both pre-requisites and enablers for IaC and state reconciliation. As long as these requirements are be met, other platforms can benefit from state reconciliation.

For example, continuous state reconciliation could be used for home automation, where states of different devices and appliances (e.g. light bulb color, room temperature) could be described in the desired state. The actual state of devices and appliances could then be read during the reconciliation process, as long as the home automation system provides exposes such information and control functionality via its APIs.

5.5 Summary

In this chapter, one of the core mechanisms used in infrastructure-as-code has been identified, generalized and described as state reconciliation. Additionally, continuous state reconciliation was introduced as a more powerful variation of state reconciliation and different techniques of implementation were discussed.

CHAPTER 6

Kubernetes

As hinted in the previous chapter, Kubernetes represents a system and a platform which heavily utilizes the pattern defined as state reconciliation in the previous chapter (chapter 5). This chapter introduces and further dives into Kubernetes, describing it's architecture, its usage of state reconciliation and, most importantly, the API extensibility features which enable and facilitates implementation of continuous state reconciliation loops for custom use-cases, which constitute part of the topic of this thesis.

6.1 Overview and architecture

This section is inspired by and partially based on a chapter from previous work in [36, section 4.3].

As stated in the documentation of Kubernetes, it is meant to be a *portable, extensible, open-source platform for managing containerized workloads and services*. It is also states that the platform *facilitates declarative configuration and automation* [22]. On a high level, Kubernetes acts as an open-source, vendor-agnostic platform for hosting applications and workloads in form of containers that are dynamically scheduled on a pool of nodes. To fully facilitate this use-case, it provides abstractions for networking, storage and other infrastructure-level concerns, as well as integration for provisioning external resources such as load-balancers or block storage in the cloud (or similar dynamic infrastructure platform).

The architecture of consists of a set of relatively small services which cooperate and communicate mostly by reading and writing data to a common metadata database. In a typical cluster, there are two sets of nodes (servers): the master nodes, which establish the control plane cluster and the worker nodes where actual applications are running.

Based on that, services can be further separated into two groups: the *control plane components* and the *node components*. In most configurations, the control plan components only reside on the master nodes, while node components are general and reside on all nodes. *control plane*, and the worker components running on worker nodes. An overview of the overall architecture may be seen in fig. 6.1. [18]

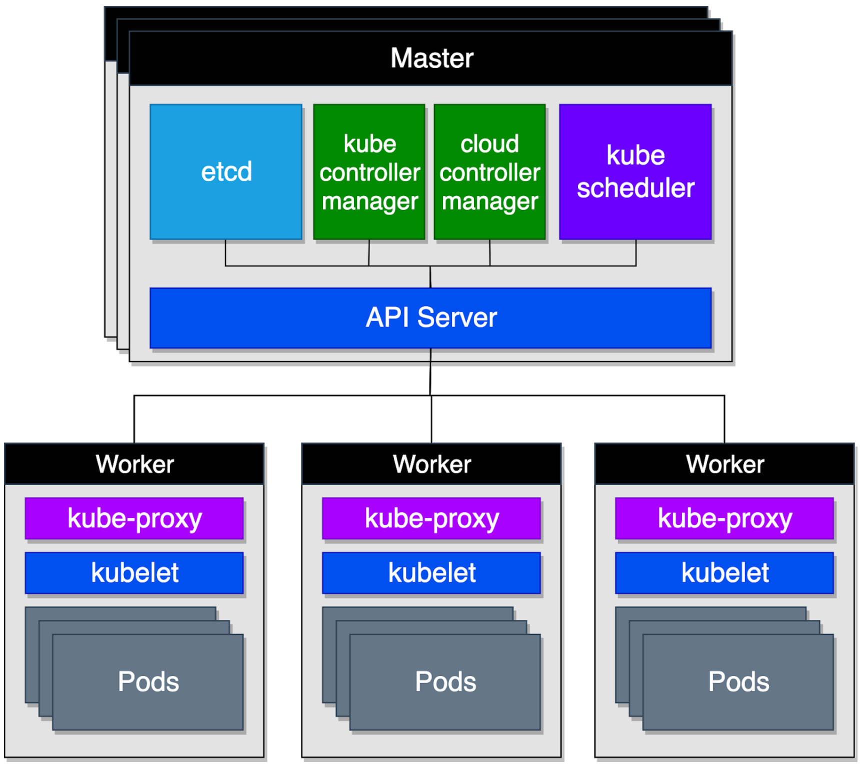


Figure 6.1: Architecture of Kubernetes

6.1.1 Control plane

As seen in fig. 6.1, the control plane consists of several services or components, namely *etcd*, *controller manager*, *cloud controller manager*, *scheduler* and the *API Server*.

etcd (metadata store)

As per the home page of etcd[1], it is *a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines* [6]. In Kubernetes, etcd is used as the backing store which holds all the metadata

[1] https://etcd.io/

of a Kubernetes cluster[18]. Examples of cluster metadata include definitions and statuses of node membership, container deployments, internal and external IP addresses and hostnames of applications, and many other infrastructure resources.

The design and capabilities of etcd help ensure consistency and reliability of the cluster metadata, which is critical for healthy operation of a Kubernetes cluster. This is mainly achieved by running a separate instance of etcd on each of the master nodes. The metadata is kept replicated among a the quorum of all etcd instances using the raft [35] algorithm.

Additionally, some features of etcd are also heavily leveraged in Kubernetes, such as the real-time API functionality for asynchronously receiving change events about the contents of the database (a.k.a the "watch" functionality, also touched on in section 5.2.1) [5].

API Server

While etcd is responsible for reliable storage of all the metadata, components do not manipulate this data directly.It is instead facilitated by the *API Server*. This component acts as a layer on top of etcd, hosting the Kubernetes API, which exposes operations for all Kubernetes resources and other domain logic of Kubernetes.[18]

The Kubernetes API is at the very core of all Kubernetes functionality and integration, due to the fact that it is used by both the internal components (as described in this section and in fig. 6.1) as well as external integrations for communication and orchestration purposes. The API design revolves around storage of domain-specific resources, following a model that is referred to as the *Kubernetes Resource Model*, which will be described in more detail in chapter 6.

Controllers and controller managers

In Kubernetes, a vast majority of the functionality is implemented by individual services which asynchronously collaborate via API Server and (indirectly) etcd. These services are generally referred to as *controllers*. Each controller is designed for and manages a narrow set of resource types via the API server. [18][41]

The *controller manager* and *cloud controller manager* are processes that package the core controllers of Kubernetes. Examples of controllers packaged in kube controller manager include the *node controller*, which manages server membership using *Node* resources, or the *replication controller* focused on ensuring that desired number of instances (replicas) of different applications and workflows are running.[18]

The *cloud controller manager* is only included if the cluster is running using a cloud provider and houses controllers that handle cloud-provider-specific logic, such as provisioning and managing load-balancer instances in the cloud, which are usually configured using the *Service* resource types.[18]

Scheduler

The *scheduler* (or *kube-scheduler*) is a component that effectively manages the resource pooling aspect of Kubernetes by handling the allocation of Pods to nodes, a Pod being the unit of deployment in Kubernetes (essentially a set of one or more container instances.

The scheduler watches the *Pod* resource type in the database. Whenever it encounters a pod which does not have a node assigned to it, the scheduler uses Kubernetes-specific scheduling algorithm to determine a suitable node for the new pod and assigns it to that node. The algorithm analyzes a variety of factors to make such decision related to, among others, the available hardware resources on each node as well as different constraints and policies, including user-specified ones.[20])

6.1.2 Nodes

There are three fundamental components that run on all Kubernetes nodes, whose role is to run and supervise containers as well as to support service discovery and network communication: *kubelet*, *container runtime* and *kube-proxy*.[18]

Container runtime

The *container runtime* (or container engine) is the component that runs and supervises containerized environments, typically Linux containers based on the LXC [29] technology but Windows nodes and containers are also supported [17]. Similarly to etcd, it is not a component developed as part of Kubernetes, but rather an existing container engine. Since Kubernetes uses a standardized interface called Container Runtime Interface (CRI), any runtime that supports it can be used. Examples include Docker[2], containerd[3] and CRI-O[4].[18]

kubelet

Kubelet can essentially be thought of as a Kubernetes agent. Each instance of the kubelet service is aware of the identity of the node it is running on and its role is to watch for Pods in Kubernetes that have been assigned this specific node. For each such pod, kubelet ensures that all the specified containers are started and running, by communicating with the container runtime using CRI.[18][11]

[2]https://www.docker.com/products/container-runtime
[3]https://containerd.io/
[4]https://cri-o.io/

kube-proxy

In Kubernetes, Services are virtual load-balancers that are assigned their own virtual IP address within the cluster which can be used to route and load-balance traffic to a multiple Pod instances. The kube-proxy component, on each node, ensures that the traffic from pods trying to communicate to these virtual services is routed correctly by maintaining a set of routing rules on the node. These rules are based on the IP addresses and endpoints specified by the resources of Service in etcd and are typically applied using operating-system-level capabilities such as iptables or IPVS (IP Virtual Server) on Linux, or similar. [18][21]

6.2 Kubernetes Resource Model

This section focuses on the Kubernetes API and the design decisions, that went into it in order to make the platform scalable and extensible and is based on a presentation done Daniel Smith, a software engineer working on the Kubernetes project, titled *Kubernetes-style APIs of the Future*[39].

The concept discussed here should lay the foundations for an in-depth understanding of the design of Kubernetes, how it embraces state reconciliation (further described in section 6.3) and how Kubernetes can be extended to custom use-cases (subject of section 6.4).

Kubernetes as a platform tries to provide abstractions for and facilitate many aspects of hosting applications managing infrastructure all the while enabling its users to adopt high levels of automation. This makes the the design of the Kubernetes API crucial to the success and adoption of Kubernetes. As time goes on and the platform matures, new functionality and abstraction need to be added, which constantly and inevitably increases the complexity of the system.

For example, according to a report from 2018, the number of exported API endpoints in the codebase of Kubernetes has increased from around 4000 in version 1 released in 2015 to around 16000 in version 1.12 released in 2019[9].

6.2.1 Complexity of APIs

Generally, the functionality an API consists of exposing different actions that can be performed in different contexts and on different entities. As illustrated in fig. 6.2, given the number of actions N and number of entities M, this means that the overall number of operations exposed by the API equals to $N \times M$. As the API matures and expands, both N and M may increase, resulting in polynomial growth of the overall number of operations.

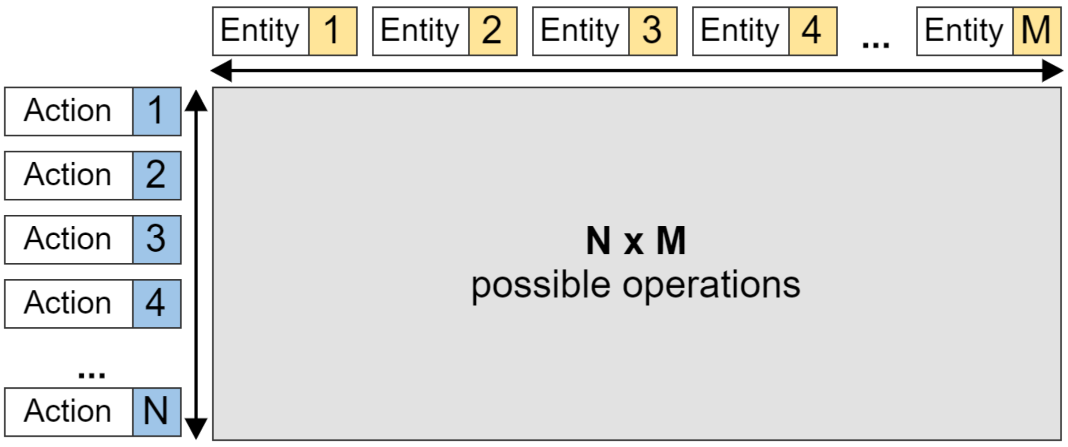


Figure 6.2: General case API operations complexity.[39]

REST APIs

There are approaches for API design that aim to reduce this complexity. For example, a popular approach is to design remote, HTTP-based APIs according to Representational State Transfer (REST). This architectural style introduces some constraints for how the functionality of an API should be exposed.[38]

One of those constraint is to regard the functional entities of the API as web resources whose types and attributes must be encoded as part of the URL address used for API requests, similarly to how websites and HTML documents are exposed. The action to be performed on a given resource is then inferred from the HTTP verb used for a request for a resource.[38]

As there are only five generally used HTTP verbs, `GET`, `POST`, `PUT`, `PATCH` and `DELETE`, the number of actions (N) in a RESTful API are technically reduced. However, in terms of usability and the perspective of a consumer/user of the API, this might not always be the case.

While REST enforces the interface of the API (combination of HTTP verb and a resource identifier), it does not restrict the actual implementation details of the actions, nor the shape of the entities. As a result, the usage details of issuing, for example, a `GET` request for a resource A might be significantly different from issuing a GET request for a resource B. Similarly, the structure and data of the returned response might differ as well. [39]

From a scalability perspective, even a RESTful API may pose maintenance challenges, due to, for instance, manual documentation being required for many operations in order to ensure correct usage and ergonomic experience on the side of the consumers.[39]

6.2.2 The Model

In the case of the Kubernetes API and the overall codebase of Kubernetes, a new approach was adopted that is generally referred to as the *Kubernetes Resource Model (KRM)*.This approach, similarly to REST, uses the combination of entities (resources) and a few possible actions (verbs) to model operations and API functionality. However, unlike REST, KRM adds further restrictions and standardization to limit complexity. [39]

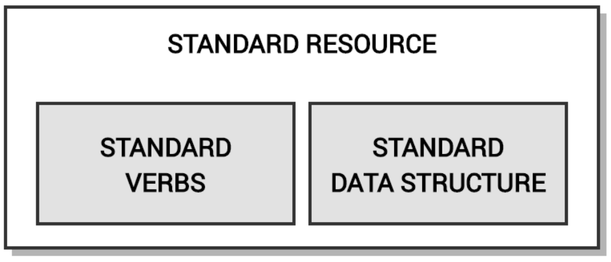


Figure 6.3: A standardized resource in Kubernetes.

Standard verbs

Currently, there are six possible actions (verbs) supported by the Kubernetes API: `POST`, `PUT`, `PATCH` `GET`, `LIST`, `WATCH`, `DELETE` and `DELETECOLLECTION`. While they appear to be a superset of the standard HTTP/REST verbs, these verbs constitute the Kubernetes API verbs, which are distinct and not to be confused with HTTP verbs. [28]

That being said, the semantic usage of most verbs is analogous to REST and HTTP, with the exception of the extra verbs. `LIST` is meant for retrieving collections of a given resource type, while `DELETECOLLECTION` is meant to delete one. `WATCH` is characteristic of Kubernetes (see section 6.3) and is used for subscribing to a change event stream (usually for a collection of a given resource type). [28]

On the transmission level, the Kubernetes API does make use of the HTTP protocol and its verbs. For example, a Kubernetes `GET` is performed using an HTTP `GET` request, but this does not necessarily imply a one-to-one mapping, as the same (HTTP `GET`) applies for Kubernetes `LIST` and `WATCH`.[28]

Additionally, contrarily to REST, all of Kubernetes verbs are generally required to be implemented for each resource.[39]

Standard data structure

Furthermore, resources in Kubernetes must all follow a standardized schema.

First of all, each Kubernetes object[5] must provide the `apiVersion` and `kind` fields, where `apiVersion` is a string composed of an API group identifier (logical grouping of API functionality) and a version (e.g. `v1`, `v1beta1`, etc) and `kind` is the group-unique name of the specific resource type.[12][27].

Moreover, each object must also have a valid `metadata` field. The metadata object itself has required values such as the *namespace*, which indicates membership of resource instance in a particular Kubernetes namespace (logical grouping for the purposes of separation and multi-tenancy) as well as a namespace-unique `name` and a globally unique `uid`. [12][27].

Finally, the required `spec` and optional `status` fields are meant to contain resource-type specific data and schema. This is where most resource types vary in structure.[12][27]

Standard resources

As depicted in fig. 6.3, the combination of the restrictions on verbs and the structure is how resources (entities) are standardized in Kubernetes. As a standard resource conforms to one standard structure used across all mandatorily supported standard verbs, the API consumption challenges described in section 6.2.1 are mitigated.

As represented in fig. 6.4, in practical terms, from the perspective of the API user/consumer, the general case of complexity described in section 6.2.1 and fig. 6.2 no longer holds. Instead, the KRM restrictions help ensure exactly $N = 8$ verbs per resource, i.e. linear . In addition to that, each resource type conforming to a common structure means that a user's knowledge (or a programmatic consumers existing implementation) of a resource type A is more likely to translate to the usage of a resource type B.

Additionally, the Kubernetes also provides *apply* functionality. What this allows is that a declaratively defined Kubernetes resource (typically in form of a YAML-formatted file) can simply be applied to to cluster, meaning the *apply* logic will determine the correct course of action to take in order to maintain the declared values of the object. This further reduces the complexity from the usability point-of-view, since users (and programmatic integrations) do not need to determine whether the resources exists and need to be updated, or just needs to be created, or which properties specifically need to be patched.

[5]In the context of Kubernetes, this thesis will use the terms resource and object interchangeably.

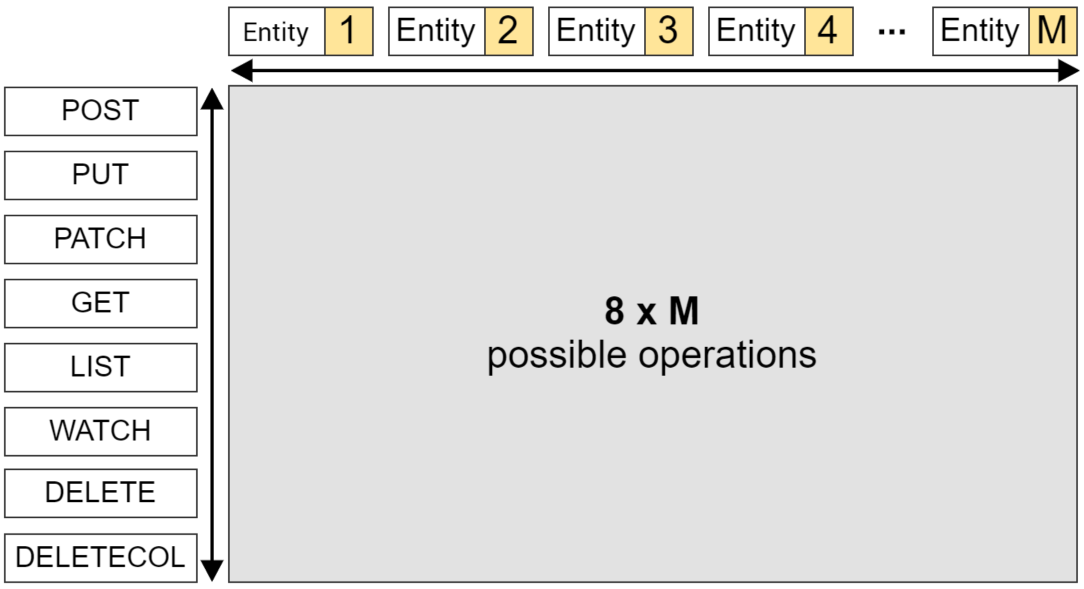


Figure 6.4: API operations complexity of the Kubernetes API (in accordance with Kubernetes Resource Model)

6.3 State reconciliation

As hinted previously, the majority of components and functionality in Kubernetes is implemented using small applications of the continuous, event-based state reconciliation pattern described in chapter 5. This section elaborates on this usage and illustrates the concept with an example which uses core Kubernetes resource types and components.

6.3.1 The Controller pattern

In Kubernetes, state reconciliation usages are contained in services referred to as *controllers*, named based on the idea of a control loop from control theory, which is a *non-terminating loop that regulates the state of a system*. [24]

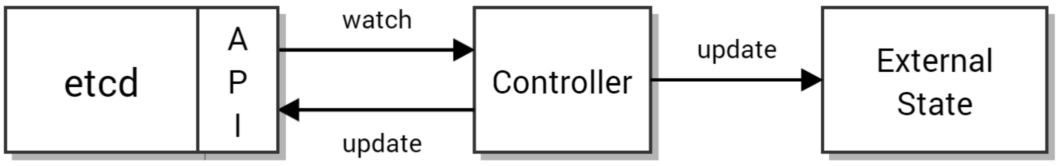


Figure 6.5: A *controller* in Kubernetes

As shown in fig. 8.3, a controller constantly watches some set of objects in the Kubernetes state for changes via the Kubernetes API. The watched objects, specifically their `spec` attribute, describe some desired state which the controller will attempt to reach in the relevant parts of the existing environment (actual state). [24]

In other words, the controller hosts the reconciliation loop that performs continuous state reconciliation, as defined in section 5.2. Figure 8.3 represents the general definition of a controller, illustrating that a controller may, as a result of an event, update some resource(s) in `etcd`, or update some external environment (e.g. cloud provider resource), or both. Therefore, the delimitation of the *desired* and *actual* state, as defined in chapter 5, varies between different implementations of controllers and the problems they have been designed to solve. fig. 8.3

The following subsection provides an example of this pattern by describing, on a high-level, the processes behind how Kubernetes ensures that containers run according to declared specifications.

6.3.2 Example: Pods, Scheduler and Kubelet

In Kubernetes, *Pods* are object used to deploy applications as containers. They are the smallest unit of deployment and provide a shared environment (e.g. network, IP, port-space, storage etc) to one or more encapsulated container. An interaction between the Kubernetes API, the kube-scheduler, as well as the Kubelet and the container engine ensures that any declared pod ends up running on one of the worker nodes. [20][18]

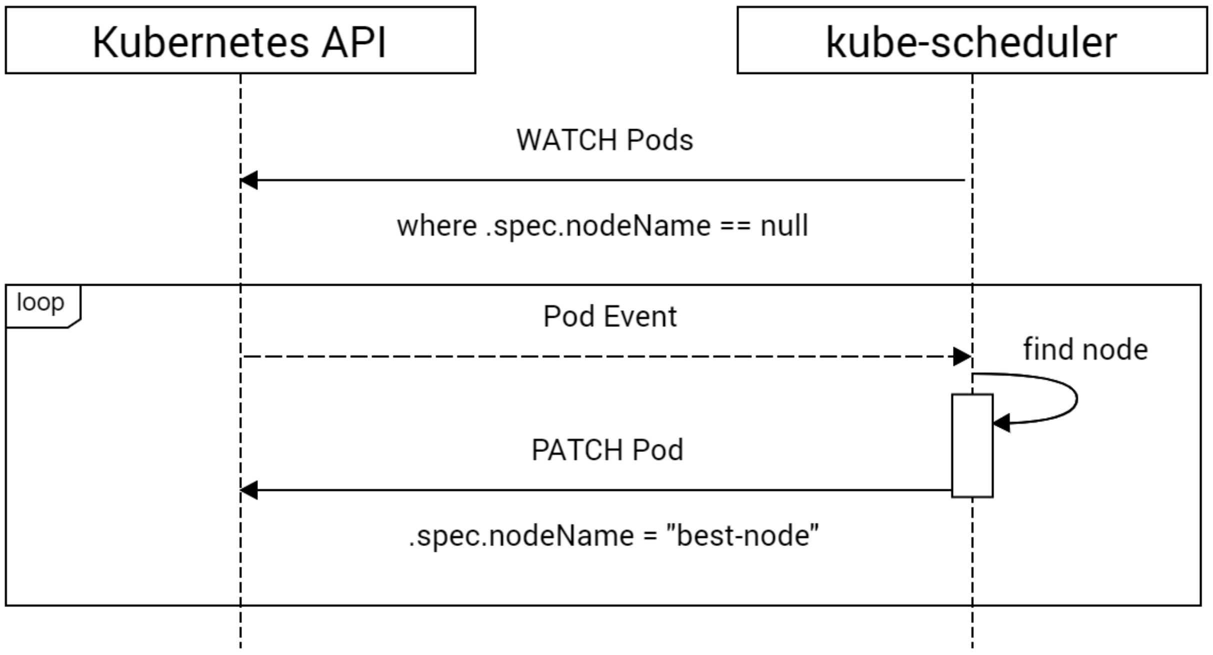


Figure 6.6: Interaction between the Kubernetes API and the kube-scheduler components

First, whenever a new Pod is created, the `nodeName` attribute of its `spec` is not defined. As illustrated in the diagram in fig. 6.6, the scheduler uses the watch functionality to receive events about Pods without a node assigned. Then, on each such event, the scheduler invokes its scheduling algorithm, which takes into account several factors (e.g. networking, storage dependencies, user-defined node preferences, etc) to determine a suitable node for the pod. The pod object is then updated, with the name of the selected node assigned (*best-node*, in this case).

Meanwhile, as shown in fig. 6.7, the kubelet component on the node named *best-node* also listens to events about pods, except in this case, only pods assigned to its own node (*best-node*). On any such pod event (e.g. when the scheduler assigns the node), kubelet will communicate with the container runtime on the node to start the container(s) as per the `spec` declared in the pod. Finally, it will then update the pod with status information about the containers.

In fact, kubelet does not only update the statuses after it react to a pod event. It periodically checks the statuses and health of containers and updates the information in Kubernetes. This detail has been omitted from fig. 6.7 for brevity and simplicity.

Desired and actual states

In the case of kube-scheduler (fig. 6.6), the Kubernetes state, in fact the individual Pod objects themselves, represent both the desired and the actual state. Another way to look at it would be to

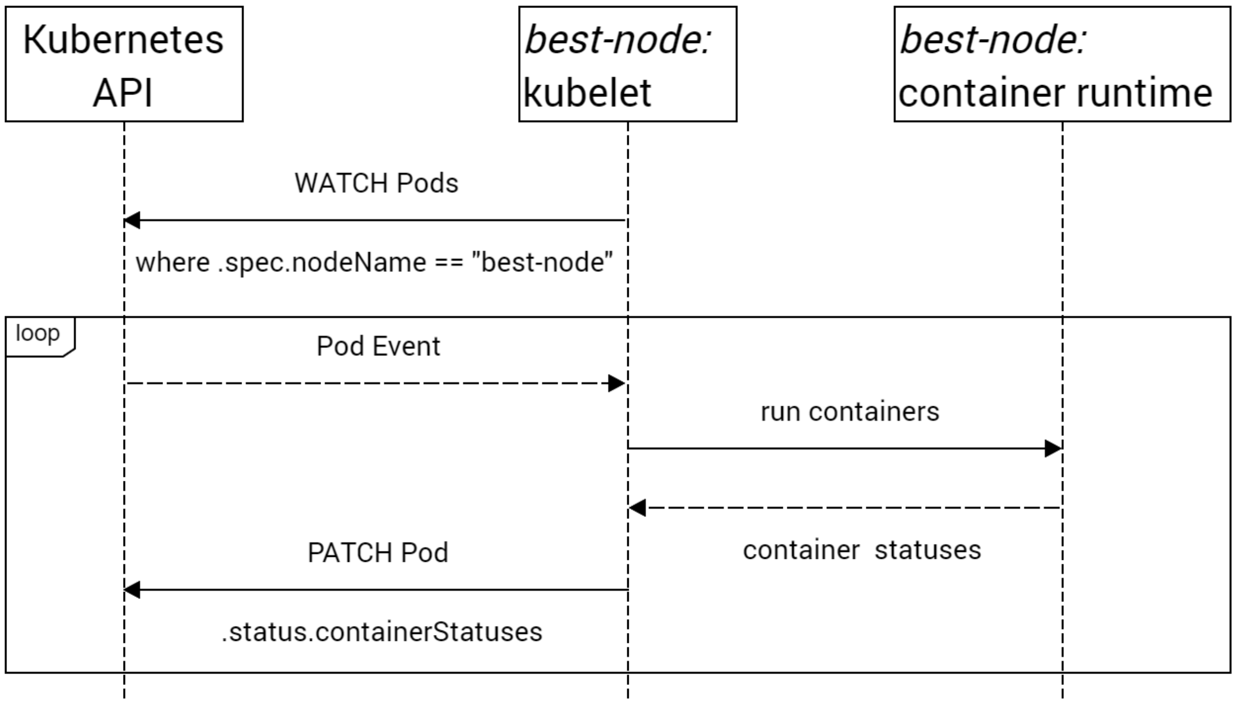


Figure 6.7: Interaction between the Kubernetes API, the kubelet and the container runtime

consider the `.spec.nodeName` part of the pod specification to be the actual (target) state, while the rest of the specification (and the existence of the Pod object itself) represents the desired (source) state.

In the case of kubelet (fig. 6.7), the Pod objects act as the desired state, while the the actual state in this case corresponds to the state of the container runtime, i.e. the state of the containers.

6.4 Custom state reconciliation

As discussed in this chapter, state reconciliation is at the core of Kubernetes and naturally, many aspects and capabilities of Kubernetes and its API facilitate the implementation of reconciliation loops and controllers.

However, it can be argued that the reconciliation-based continuous control approach represents a general pattern that can be useful for implementing use-cases beyond core Kubernetes functionality, including Kubernetes-specific automation/extensions, but also fully custom applications. This is why Kubernetes ships with extensibility features that enable implementation of reconciliation loops with custom logic. This section talks about and illustrates those features.

6.4.1 Custom Resource Definitions

In order to implement custom reconciliation, the possibility to define new and custom resources with individual custom schemas is necessary. In Kubernetes, this can be done using *Custom resource definitions*, or *CRDs*, which are a special resource type in Kubernetes (`CustomResourceDefinition`). [25]

The structure of a resource of type `CustomResourceDefinition` provides fields for specifying a custom resource that should be added to the Kubernetes API. Once a valid CRD is submitted, the Kubernetes API will be automatically extended with new endpoints, the URL structure of which depends on some of the fields. Similarly, the resource will also become available to query and manipulate in the `kubectl` Kubernetes command line client. The resource, via the API or `kubectl` will automatically support all the Kubernetes API verbs (section 6.2.2). [26]

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: lightbulbs.myhome.net
spec:
  group: myhome.net
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                on:
                  type: boolean
                color:
                  type: string
  scope: Namespaced
  names:
    plural: lightbulbs
    singular: lightbulb
    kind: LightBulb
    shortNames:
    - lb
```

Listing 2: Sample `CustomResoureDefinition` YAML manifest

Listing 2 contains a `CustomResourceDefinition`, which specifies a custom resource called

`LightBulb`, illustrating a possible use of custom reconciliation for home automation purposes.

API Group

Another attribute that needs to be specified for a new custom resource is the API Group name, as seen on line 6 in listing 2. The concept of an API group is used for the majority of Kubernetes resources (except the core ones like Pod) to group related resource types together and create a logical "partition" of the Kubernetes API.

The group becomes part of the URL of the API endpoints as well as part of the custom resource metadata and can also be used for permission control. It is common to name API groups using domain names to indicate ownership/authorship. [26]

Since `CustomResourceDefintion` is also itself a Kubernetes resource, we can see a reference to the API group it belongs to in listing 2 (line 1), namely `apiextensions.k8s.io`

Versions

The `spec` of a CRD further needs to specify at least one version for the new resource.

A version specifies the schema of the new resource structure, using the OpenAPI[6] schema specification format. For instance, the resource in listing 2 specifies that each `LightBulb` object should have a `spec` (lines 15-16) property, which should in turn be an object holding a boolean `on` property (lines 18-19), meant to indicate the state of the light bulb) and a string `color` property (lines 20-21, meant to indicate the color of a smart bulb).

Each version can be enabled or disabled (`served` attribute on line 9) and exactly one of the versions must be configured as the storage version (line 10). The storage version represents the structure that is actually going to be stored in etcd, while other version will simply be converted to the storage version. [26]

Versions are conventionally called for example `v1`, `v2`, etc. for stable versions, and for example `v1alpha1` or `v1beta2` for alpha/beta versions.

Names

To declare a new Kubernetes API resource, it is required to specify several forms of its name. The *plural* name (`lightbulbs` on line 24 in listing 2) is going to be used in the URL structure of the Kubernetes API endpoint for the resource, i.e. `/apis/myhome.net/v1/lightbulbs`. The *singular* name (line 25) is typically used in `kubectl` CLI commands (e.g. `kubectl get lightbulb`. It is

[6]https://swagger.io/specification

also possible to define shorter aliases for `kubectl` (e.g. `kubectl get lb`), as seen on lines 27-28. Finally, the last name that must be specified is the `kind`, which is essentially analogous to the name of the resource type and used in YAML manifests. [26]

Scope

The CRD must also specify how will the new resource be scoped. Generally in Kubernetes, a given resource type can either be *namespace*-scoped, meaning each object must belong to some namespace, or *cluster*-scoped, meaning each object is global. In a CRD, this is specified using the `scope` parameters, such as on line 22 in listing 2.

6.4.2 Custom controllers

Once the new custom resource definition is registered, respective custom resource objects with their specifications can be applied to Kubernetes. While that allows declaring and storing the desired state, a custom controller is required to be in place in order for the specifications to be applied to the real state/environment.

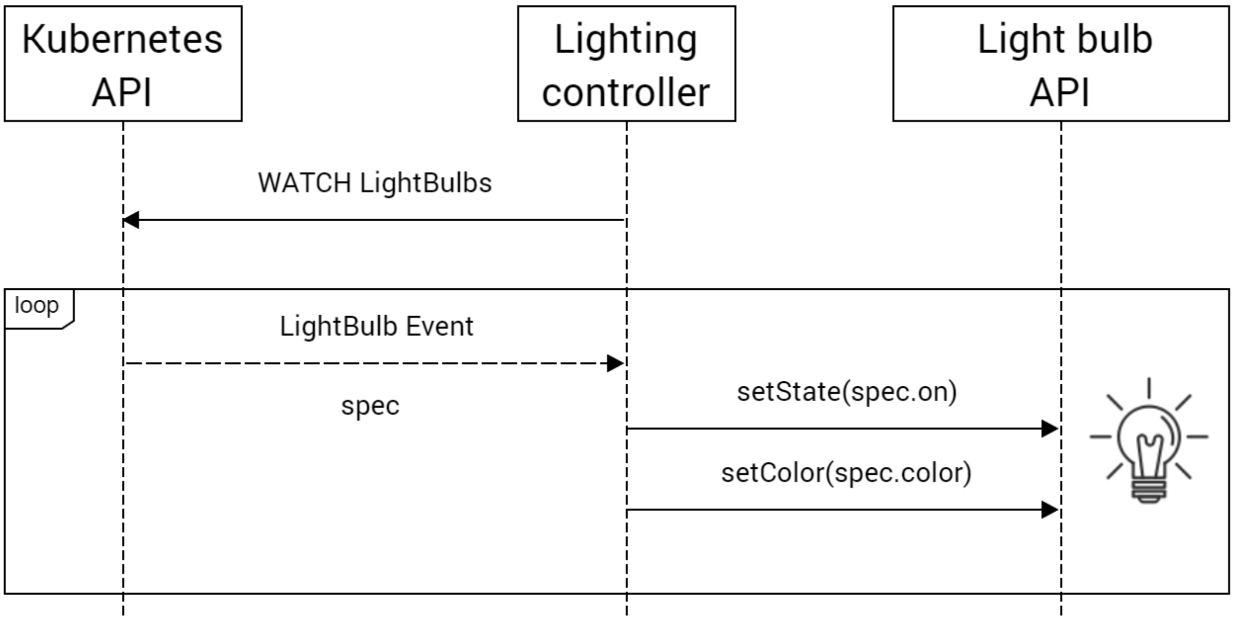


Figure 6.8: Possible custom controller for `LightBulb` custom resources

Continuing with the `LightBulb` example from listing 2, the diagram in fig. 6.8 illustrates a potential implementation of a custom controller for managing resources. As seen in the diagram, a *lighting controller* would be developed, which would use the watch API to listen for events regarding `LightBulb` custom objects, similarly to internal Kubernetes controllers.

Afterwards, on each event, such as when a new `LightBulb` is created, the custom controller can use the data included in the `spec` of the object in order to perform the necessary operations to reach the specified state. In this example, we assume there is a home automation API allowing to imperatively set the state (on/off) and color (e.g. "red") on a light bulb.

In this case, the controller is custom and does not ship as part of Kubernetes. Instead, the controller service needs to be installed into the cluster. Since all logic and dependencies are contained in the controller code, the installation can be done by simply deploying the controller in a container, using the same methods as for any other application, such as by adding a new Pod or a Deployment (set of replicated pods) object to etcd.

The Operator pattern

The approach that was just described is often referred to as *the operator pattern*. The term is used to essentially refer to the application of the controller pattern (section 6.3.1) to a some custom, domain-specific use-case. Services following this pattern are called *operators* and are paired with one or more custom resource definitions for custom resources whose specifications they are implemented to manage. [4]

The *awesome-operators*[15] GitHub repository lists hundreds of open-source operators created by the Kubernetes community. These are applied to use-cases which include, for example, managing infrastructure systems such as Cassandra database instances, configuring monitoring dashboards, or adding serverless (Functions-as-a-Service) capabilities to Kubernetes.

6.5 Summary

This chapter has delved into the details of the Kubernetes platform. First, a general overview of the system and its architecture has been provided, followed by an explanation of the Kubernetes Resource Model, which is used in the Kubernetes API to manage the complexity that comes with scaling and extending this vast platform.

Moreover, we have described how Kubernetes embraces declarative code and the pattern defined as continuous state reconciliation in section 5.2 in form of reconciling services dubbed as controllers.

Finally, this chapter also showed how Kubernetes provides first-class, API-level capabilities to easily extend its own API in order to leverage the platforms state reconciliation facilities, as well as how can these capabilities be used to implement custom reconciliation loops.

The background information provided in this chapter should have sufficiently illustrated and motivated the potential value and usefulness of implementing custom state-reconciliation-based applications and use-cases on the Kubernetes platform.

CHAPTER 7

Problem definition

In the beginning of the thesis, we have introduced the value of implementing custom state reconciliation by leveraging existing facilities provided by the Kubernetes platform, which was further investigated and reinforced in the background chapters (chapters 3 to 6).

As mentioned in chapter 1 and also illustrated further by chapter 2, while there is decent support for such implementations in the Kubernetes-native Golang ecosystem, that is not the case for many other languages, where integration with Kubernetes API can cost additional effort.

Moreover, this thesis recognizes the popularity and wide-spread adoption of the C# language and .NET platform, which is why, as per the initial question from chapter 1, the overall objective of this thesis is to explore how these state reconciliation implementations on Kubernetes can be made more accessible for use-cases where the C#/.NET platform is desired and/or required.

In this section, we will discuss and introduce the solution that will be attempted in this thesis and describe its requirements and scope.

7.1 Possible approaches

There are several potential approaches for improving the accessibility of mentioned implementations in the C#/.NET ecosystem.

Provide documentation

First of all, a descriptive documentation-driven approach could be chosen. In this case, the required details, example code and steps could be extensively described and documented.

While this approach would likely reduce the efforts of implementation to some extent, a significant amount of effort would need to be spent to replicate the documented process. Additionally, this would not enable code reuse in the community and hence potentially impair future evolution and maintenance of implemented projects.

For these reasons, a better approach would be to provide a general and reusable software development kit (SDK) including relevant tools and a code component (library) with a high-level interface for developing custom controllers and operators.

SDK from scratch

A potential approach for providing the SDK would be to implement it from scratch. This would first involve implementing low level functionality for communicating and interacting with the Kubernetes API in a reliable, secure and efficient manner, followed by implementing the necessary abstractions for custom state reconciliation.

The advantage of this approach is that the SDK could be tailored to this specific use-case. On the other hand, interaction with the Kubernetes API comes with significant amount of complexity due to, for instance, HTTP transport details such as handling certificate-based authentication, or handling real-time streaming for the watch functionality.

SDK based on C# client

Finally, to reduce the effort and scope of developing and maintaining the SDK, it could be based on the existing C# client mentioned in section 2.3. While, as already discussed, the client has limited functionality, it could be reused for basic operations against the API, including low-level watch functionality.

The state reconciliation SDK would be built on top of the basic functionality provided by the C# client and if appropriate, some of its functionality could eventually be accepted into the client project itself, further streamlining maintenance and future work.

7.2 Problem statement

Based on the presented overview of possible approaches, it is believed that building a Software Development Kit on top of the existing C# client provides a good balance of feasibility, quality and delivery in the context of the main goal of this thesis and has therefore been selected.

The specific problem of this thesis is therefore to *investigate, design and implement a Software Development Kit, based on the C# Kubernetes client, for developing Kubernetes-based custom state reconciliation use-cases with the aim of making such use-cases more accessible to C# developers*.

This proposed solution will be called and throughout the report referred to as *KubeSharper SDK*, or simply *KubeSharper SDK*. Functional and non-functional requirements will be further defined in the next section, while details regarding the design and implementation of the solution will follow later in the report.

To evaluate the SDK solution and the extent to which it solves the described problem, it will be used to build several representative applications, after which the functionality and the experience of the process will be reviewed against the stated requirements and the overall aim of the thesis.

7.3 Requirements

Based on the previous section, this thesis will attempt to answer the research question mentioned earlier, by implementing a Software Development Kit for implementing custom controllers and state-reconciliation-based applications on the Kubernetes platform and using the C# language.

The proposed software development kit, which throughout the report referred to as *Kube-Sharper*, should mainly consist of a library that can be imported in C# projects and contains useful and general abstractions for implementing discussed use-cases. Alongside the library, additional developer tools could be provided if relevant.

The upcoming subsections represent a list of requirements for *KubeSharper*.

7.3.1 Event configuration

First of all, the *KubeSharper* library should facilitate concise configuration of the event sources based on which reconciliation should be triggered. As discussed in section 6.4, the main approach for implementing custom reconciliation on Kubernetes is to encapsulate domain-specific desired state in one or more custom resources that a custom controller will watch.

For this reason, the *watch configuration* capabilities of KubeSharper should allow users of the library to initialize this resource observation process. To minimize boilerplate library code, the configuration interface should be minimal, only requiring the application-specific necessary arguments, such as information about the (custom) resource to be observed.

For example, a developer buidling the `LightBulb` use-case from fig. 6.8 on page 40 should be able to simply indicate that events about all resources with kind `LightBulb` should be watched.

7.3.2 Reconciliation interface

The reconciliation logic represents the core part and value of implementing the discussed custom reconciliation use-cases on Kubernetes. This is why, apart from being able to configure watched resources, users should also be able to easily provide a use-case-focused, custom reconciliation function.

When it comes to the reconciliation function, the developer should be able to focus on the custom business logic, while minimizing the amount of low-level, Kubernetes or KubeSharper specific logic that needs to be written and understood.

Thus, KubeSharper should provide a programming interface for accepting a custom reconciliation function that will be automatically wired the together with the provided event configuration. Furthermore, in order to better inform efficient reconciliation decisions in the user code, KubeSharper should expose context information, such as event metadata, to the custom code. Finally, the user-provided code should also be able to signal further actions based on the result of reconciliation.

7.3.3 Native custom resource representation

Working with custom resource definitions and object constitutes a central element in these implementations. C# is an object-oriented, and provides language-level features for defining and using custom, composable structures using classes and objects.

For KubeSharper to be idiomatic, letting developers to use established language features, it should be possible to define, pass and manipulate Kubernetes custom resources using native C# objects and classes.

To illustrate this again in the context of the `LightBulb` example from fig. 6.8 on page 40, the developer should be able to work with `LightBulb` Kubernetes objects in form of native C# objects, based on a custom class, such as in listing 3, with fields and properties representing the same structure as seen in listing 2 (`LightBulb` CRD).

```
public class LightBulb
{
        public LightBulbSpec Spec { get; set; }
}
public class LightBulbSpec
{
        public bool On { get; set; }
        public string Color { get; set; }
}
```

Listing 3: Possible C# class equivalent of the `LightBulb` custom resource from listing 2 on page 38

7.3.4 CustomResourceDefinition installation

Being able to use native C# representations for custom resource objects can enable quicker development and debugging. However, whenever structural details such as the schema of a custom resource is changed in the C# representation, this change needs to be reflected in the respective Kubernetes `CustomResourceDefinition` database object.

KubeSharper should therefore also ship with relevant tooling that would allow developers to use the native class representations in order to install and/or generate equivalent Kubernetes-native `CustomResourceDefintion` objects.

7.3.5 Controller co-hosting

When developing more advanced use-cases, it may be desired to bundle and deploy multiple custom controller services together in a single process, similarly to the kube-controller-manager and cloud-controller-manager Kubernetes components introduced in section 6.1.1.

For example, the familiar `LightBulb` use-case and the relevant `light-operator` (fig. 6.8, page 40) could be expanded into a more fully featured home automation service which would involve other custom resources, such as a `Thermostat` or a `SmartOutlet`. In such a case, it would likely be beneficial to maintain separation of concerns and reconcile the different resources and devices using separate controllers with different configurations. Overall, however, the collection of controllers still forms a cohesive home automation service, and could also share some dependencies and Kubernetes-specific configuration such as authentication etc.

Therefore, another requirement of KubeSharper is to support such a configuration and allow multiple controllers to be defined and registered, while sharing some common concerns and dependencies such as Kubernetes integration code.

7.3.6 .NET dependency injection support

It is a common practice in C#/.NET projects to utilize the dependency injection technique throughout the codebase. Dependency injection allows to decouple code components and classes by relieving developers from having to directly create new objects hierarchically. Instead, a dependency injection container can be used to register instances and interface implementations in a flat structure. The container can then be used to automatically resolve any desired type together with its dependent objects, which are usually passed in through class constructors. [32]

This pattern is supported by the .NET platform libraries that can be used in the setup boilerplate code of projects. It is also often used to facilitate the hosting of background services in long-running applications using the `IHosteService` interface. Any background code can be made to conform to this interface and registered into the dependency injection container in order to have it automatically hosted and managed by the runtime. [33]

As part of the effort to make KubeSharper feel native and idiomatic and hence also more accessible within the C#/.NET ecosystem, it should also integrate with the above mentioned platform features, allowing developers to use standard patterns to host custom controller services.

CHAPTER 8

Design

Based on the problem definition and requirements described in the previous chapter, this chapter will present the overall design for the KubeSharper solution that will be developed as part of this thesis. First, an overall, high-level overview of the system and its components will be provided, followed by more detailed overviews of the individual components.

8.1 Overview of components

The proposed solution design, illustrated by fig. 8.1 below, overall consists of six major components. As illustrated by the regions in fig. 8.1, these components can be separated into three categories.

Firstly, the *KubeSharper* components represent the internal abstractions and mechanisms of the KubeSharper library, as well as the *CRD Generator* tool provided together with KubeSharper. Furthermore, the *Reconciler* component represents *Application* domain and the custom reconciliation logic, that should be provided by the user and used by KubeSharper. Finally, the *Kubernetes API* component represents the external dependency on the Kubernetes platform and the point of integration via its API.

As illustrated by fig. 8.1, the user-provided reconciliation logic is meant to be integrated with the *Controller* component and invoked based on events gathered by the controller from the Kubernetes API via the *Event Sources*. The *Manager* component serves as the host and entry point for initializing and running one or more controllers.

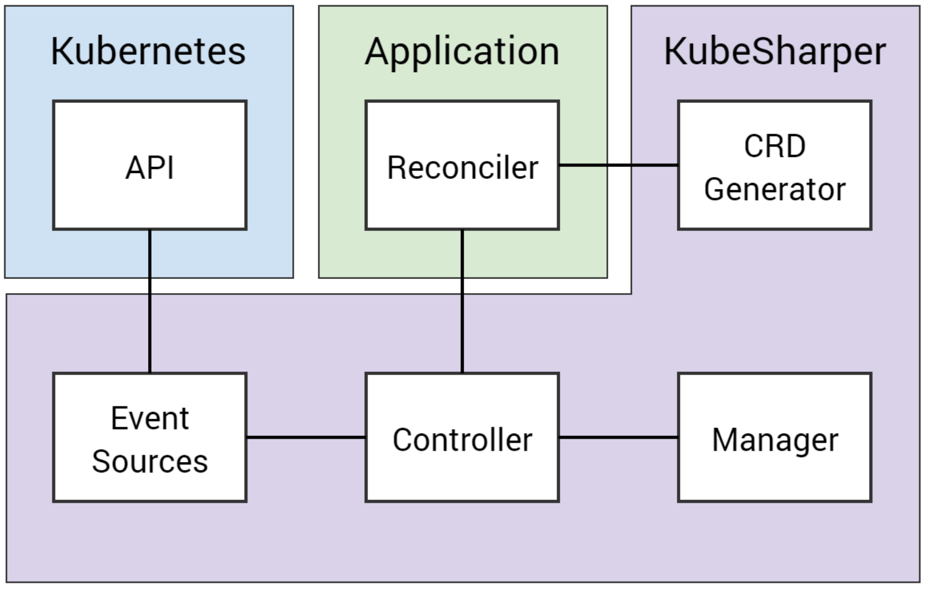


Figure 8.1: KubeSharper components overview

Additionally, the *CRD Generator* represents a tool for the users which allows generating *CustomResourceDefinition* object manifests (in YAML) based on definitions included in their own code.

This design has been influenced and inspired by the Controller Runtime library discussed in section 2.2 and reuses some of the concepts and terminology used there.

8.2 Event Sources

Event Sources are a lower-level part of KubeSharper which is meant to contain and encapsulate the integration with the Kubernetes API and the propagation of Kubernetes resource events which eventually result in the invocation of the user-defined custom reconciliation logic.

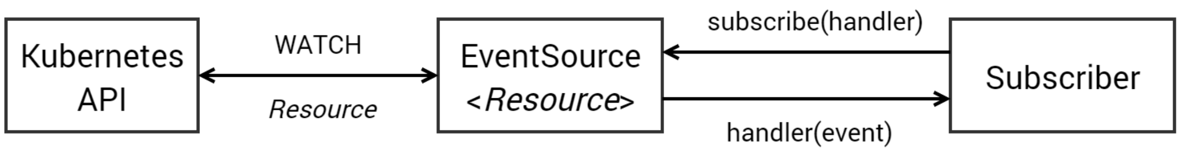


Figure 8.2: Overview of an Event Source in Kubesharper

Depicted by fig. 8.2, a single event source corresponds to a resource type in Kubernetes. The main purpose of an event source is to use the watch functionality of the Kubernetes API to listen to events regarding the respective resource type. For instance an Event Source for the resource type `LightBulb` would observe events related to any object of kind `LightBulb`.

Moreover, the event source has been designed so that it would also have the role of propagating these events to other components of KubeSharper. This is done by allowing a subscriber, representing an entity in the system that needs to directly receive and react to some events, to subscribe to a given event source by providing code to be invoked on each event, also referred to as a *handler*.

An Event Source then propagates events by being able to accept a subscriber's handler and invoke it on each event. This provides a generic interface for receiving and handling Kubernetes events in KubeSharper. Since all the Kubernetes integration is contained here, other parts of the system can be implemented in terms of resources and events instead of Kubernetes API details and idioms.

Additionally, by allowing any event handler to be passed in from outside the Event Source, the Kubernetes event stream is decoupled from its consumer, allowing the Event Source abstraction to be used in different parts of the system if necessary, using different approaches on how to react to events.

8.3 Controller

As mentioned earlier, the *Controller* represents a central part of the system, covering multiple use-cases. It represents an abstraction provided by KubeSharper for implementing the controller pattern (section 6.3.1) with custom reconciliation logic.

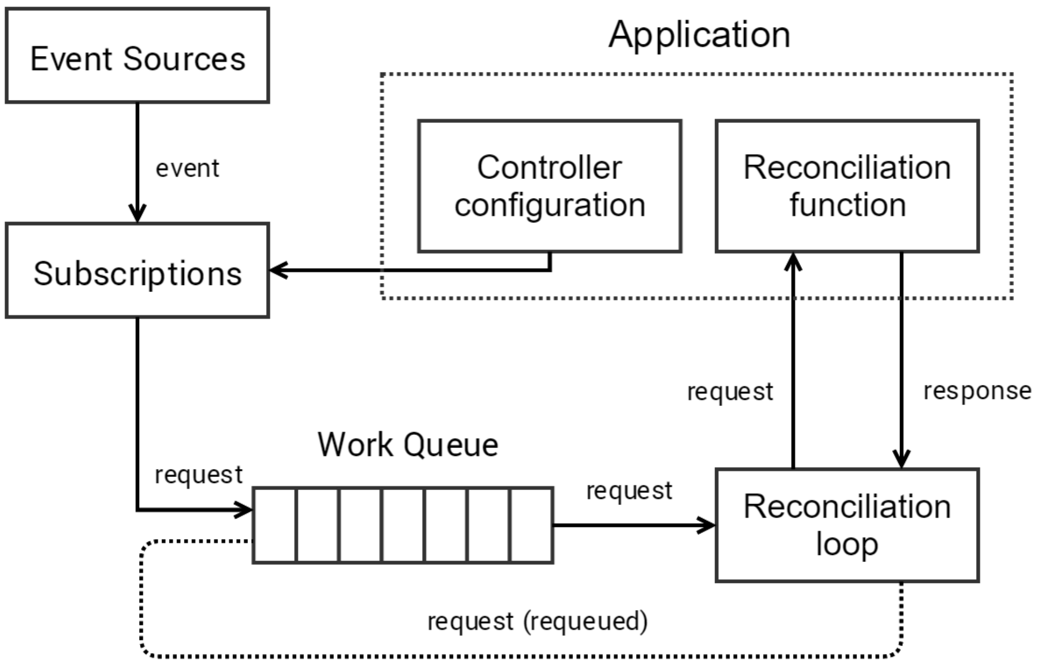


Figure 8.3: Overview of Controller component

As illustrated in fig. 8.3, Controller is responsible for initializing and managing event source subscriptions based on configuration provided by the application, allowing the developer to indicate details about which resource types are relevant and should be observed and reacted to for the particular use-case. This aspect of the Controller is meant to address the event configuration requirement described in section 7.3.1.

In the Controller, each observed event results in a *reconciliation request*. This request represents the intent to perform reconciliation of a particular object that has been created, modified or deleted.

As seen in the diagram (fig. 8.3), the Controller, using the event source subscription model described in the previous section, ensures that on each event a request is put into a queue (*Work Queue* in fig. 8.3). Since multiple events may be received from multiple event sources at any given time, putting requests into a queue ensures first-in first-out (FIFO) processing (reconciliation) of events.

The *reconciliation loop* is meant to consume requests from the queue during the entire lifetime of a Controller (and/or the application). Within the loop, each consumed request is propagated into the application layer and used to invoke the reconciliation function, which eventually indicates a *reconciliation result* to the loop. The result can be used by the application as a signal to KubeSharper (Controller) that the reconciliation failed and (optionally) that the request should be retried (requeued).

Apart from ensuring FIFO processing, incorporating a FIFO queue into the design of Controller enables centralized control of the request processing. For example, the queue could be implemented to limit the rate at which requests are enqueued. The Controller Runtime (section 2.2) library adopts a similar approach.

8.4 Reconciler

In a reconciliation application that uses KubeSharper, *Reconciler* represents the application-specific reconciliation code, which integrates with the Controller. It corresponds to the *reconciliation function* in the Controller design (fig. 8.3).

From the perspective of the KubeSharper library, the *Reconciler* is an interface according to which the custom reconciliation logic is provided by the application developer in order to be integrated with the event processing mechanisms facilitated by Event Sources and Controller.

Figure 8.4 contains a visual description of the *Reconciler* interface. Designed to be simple, the *Reconciler* needs to support a single *Reconcile* operation that takes a *reconciliation request* (as defined in the Controller design), together with a *reconciliation context* as input. The context input holds additional useful data and objects (see section 9.2.3) provided to the application by Controller.

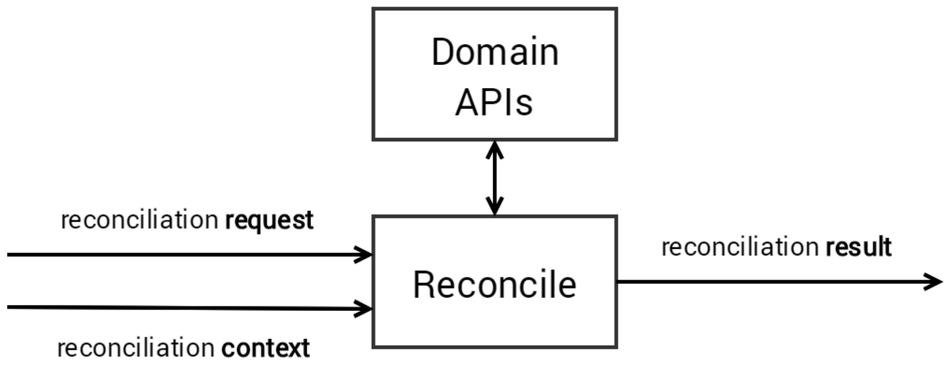


Figure 8.4: Reconciler interface inputs and outputs

This possibility of providing arbitrary reconciliation logic to be triggered by the controller using the *Reconciler* interface addresses the *Reconciler interface* requirement described in section 7.3.2.

8.5 Manager

So far, the previous components have been presented in terms of a single Controller scenario. However, as defined in the requirement from section 7.3.5, it should also be possible to co-host multiple Controllers, and therefore multiple reconciliation functions, in the same application.

The *Manager* component addresses this requirement and is based on an equivalent concept from the Controller Runtime (section 2.2) library. It acts as an container service for multiple Controllers, allowing a given set of Controllers to be managed (e.g. started and stopped) as a single unit.

Furthermore, by containing all Controllers, *Manager* can also serve as a repository for dependencies that can be shared and reused among controllers, such as user-provided Kubernetes credentials and configuration. This aspect will be further explored in the Implementation chapter (section 9.3).

8.6 CRD Generator

The *CRD Generator* component is an executable tool that is meant to address the *CustomResourceDefinition installation* requirement from section 7.3.4. As depicted in fig. 8.5, given a set of C# CRD representations within the application code (as per the requirement in section 7.3.3), the CRD Generator, when executed, will generate an equivalent set of Kubernetes `CustomResourceDefinition` object manifests (in YAML).

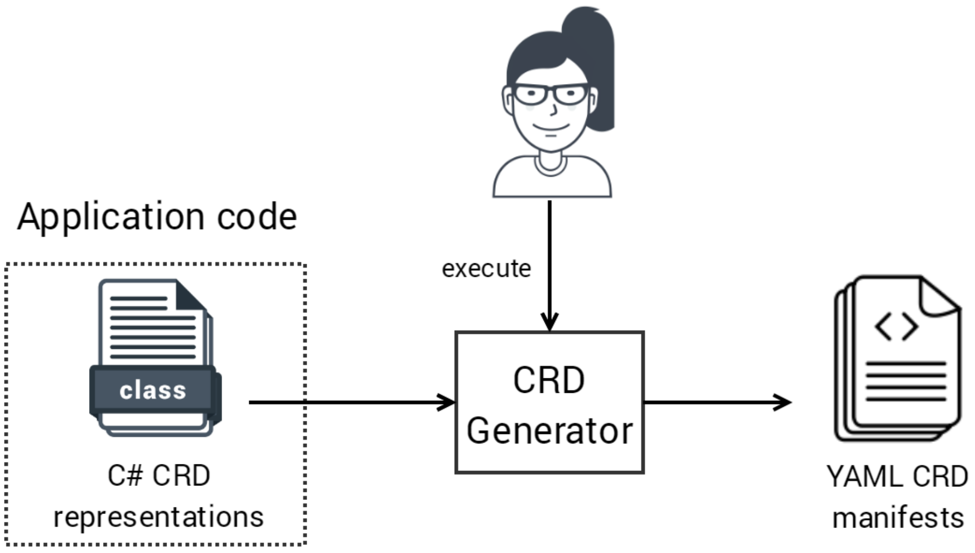


Figure 8.5: Overview of the CRD Generator tool

The tool was designed to provide a pragmatic and flexible way of facilitating the installation of CRDs to a cluster. While only generating the manifest files, instead of directly installing them to a given cluster, does not fully automate the process, it provides additional flexibility.

Firstly, the generated files are text files, which means they are subject to the Infrastructure-as-Code practices (and related benefits) described in chapter 4. For example, the files are human-readable and can also be easily compared with common tools like `diff` etc. The files can also be committed to a version-control repository and support practices like GitOps (section 5.3).

Furthermore, the generated files are standard Kubernetes manifests, which means that the tool does not impose any KubeSharper-specific practices. In other words, the tool allows developers to maintain familiarity with Kubernetes standards.

8.7 Summary

This chapter presented the design of KubeSharper and its following main components:

- the *Event Sources* which encapsulated real-time Kubernetes communication
- the *Controller*, a central component that processes resource events in order to efficiently trigger the reconciliation process when necessary
- the *Reconciler* component, which together with Controller enables users of KubeSharper to inject their custom reconciliation logic into the event processing flow
- the *Manager* component meant to facilitate composition and co-hosting of multiple Controllers

- and finally, the *CRD Generator* tool, which provides an executable tool for developers that should simplify installation of CRDs

Moreover, this design covers most of the requirements from section 7.3. The remaining *Native custom resource representation* (section 7.3.3) and *.NET dependency injection support* (section 7.3.6) are more related to providing developers with better, more idiomatic experience and usability when using the KubeSharper library. For this reason, these requirements will be covered directly by the implementation details of the KubeSharper library which will be the subject of the next section.

CHAPTER 9

Implementation

In this chapter, the details of the KubeSharper SDK implementation written as part of this thesis will be revealed. Implementation details will be presented for each of the components introduced in chapter 8 (Design) earlier, as well as additional aspects and features of KubeSharper that overall cover its purpose and requirements.

Throughout this chapter, the implementations will be mainly illustrated using C# code listings containing important parts of KubeSharper code. In order to improve readability and reduce verbosity, some non-essential code (such as logging, exception handling, etc) will be omitted from these listings.

9.1 Event Sources

Starting from the lowest layer of KubeSharper, this section will describe how the Event Sources (section 8.2) component was implemented and how Kubernetes API integration was handled.

As discussed in section 7.1, implementing Kubernetes API integration from scratch comes with significant complexity and effort required. It was therefore decided to build the Kubesharper SDK on top of the official C# client for Kubernetes. Despite the limitations of the client (as described in section 2.3), reusing its lower-level API reduces the overall scope of this project, allowing for focus to be put on state reconciliation utilities.

9.1.1 Providing a generic API

As explained in section 2.3, the main drawback of using the C# Kubernetes client stems from its mostly auto-generated code which does not provide a generic API for operations.

To provide good developer experience, KubeSharper itself needs to expose a generic API to developers. Consequently, the Event Sources component, which is directly tied to the C# client, needs to somehow provide a generic API as well.

While the ideal solution would be to improve the client and contribute those capabilities to it, a quicker solution was developed for this thesis. The solution involves creating the Event Sources API using code generation and reflection based on the C# client methods.

```
 1 private EventSource<V1Deployment> V1Deployment(
 2     IKubernetes operations, string @namespace, CancellationToken ct)
 3 {
 4     Watcher<V1Deployment> WatchMaker(
 5         EventSourceHandler onEvent, Action<Exception> onError, Action onClosed)
 6     {
 7         var list = operations
 8             .ListNamespacedDeploymentWithHttpMessagesAsync(@namespace, watch: true);
 9         return list.Watch(async (WatchEventType et, V1Deployment obj) =>
10         {
11             var metaObj = new KubernetesV1MetaObject
12             {
13                 ApiVersion = obj.ApiVersion,
14                 Kind = obj.Kind,
15                 Metadata = obj.Metadata
16             };
17             await onEvent(et.ToInternal(), metaObj);
18         }, onError, onClosed);
19     }
20     async Task<IList<V1Deployment>> Lister()
21     {
22         var list = await operations.ListNamespacedDeploymentAsync(@namespace);
23         return list.Items;
24     }
25     return new EventSource<V1Deployment>(WatchMaker, Lister, ct: ct);
26 }
```

Listing 4: Example of a Event Source factory method for the `V1Deployment` type (*generated/EventSources.cs*)

The code included in listing 4 represents a factory method that creates an *EventSource* (details in section 9.1.2) object for the type `V1Deployment`. In order to end up with a generic `EventSource<V1Deployment>` (line 25), the non-generic methods of the C# client (lines 8 and 22) are wrapped in generic functions that are passed to `EventSource`.

WatchMaker

To establish a Kubernetes watch using the client, a list operation must first be initialized with the `watch` parameter enabled (lines 7-8). Afterwards, the `Watch` extension method can be called on the result of the initial operation to establish a watch with a custom callback on every event (lines 9-18). This method returns a handle object (`Watcher<T>`, line 4) for the watch, which is generic.

Since the `EventSource` is not meant to have a watch running on initialization, it is given the `WatchMaker` function instead, which can be stored and invoked later to start a watch.

The `WatchMaker` function configures the Kubernetes watch so that each incoming event and associated Kubernetes object is first mapped to more fitting internal objects and then a callback (`onEvent` on line 17) is invoked on those. The callback is a variable, allowing it to defined in the `EventSource` object calling the `WatchMaker` function.

Lister

Furthermore, for purposes that will be discussed later, it is useful for an `EventSource` to be able to perform a one-time list of all objects. As this operation is also non-generic in the C# client, the operation is wrapped in a generic `Lister` function (lines 20-24) which returns all objects for the given resource type (in this case `V1Deployment` when called. The `Lister`, same as `WatchMaker` is passed to `EventSource` to be stored and invoked when relevant.

Code generation

The factory method for the `V1Deployment` presented only represents a single resource type. Since creating these factory methods manually would require a lot of time effort and complicate maintenance, a code-generation approach has been chosen instead.

A utility tool for populating the *EventSources* class with factory methods such as that from listing 4 has been created. The tool uses a templating library (Scriban[1]) to generate a partial definition of the `EventSources` class based on a textual template.

Looking at listing 4, the variable parts are: all the mentions of the `V1Deployment` type (lines 1,4,9,20,25) and the two calls to Kubernetes client operations (lines 9 and 22). To generate all factory methods, the template (with variables defined for the relevant parts) can be rendered given a list of all LIST operations in the C# client and the associated return (resource) types.

The developed code-generation utility determines this list by performing C# code reflection on the C# client assembly, listing all methods on the `IKubernetes` interface and filtering them according to the convention used in their naming (e.g. `ListNamespaced*WithHttpMessagesAsync`). The

[1] https://github.com/lunet-io/scriban

resource type (e.g. `V1Deployment`) is then derived from the return type of each of those methods.

As part of the code-generation process, a public and generic `GetNamespacedFor<T>` method on the *EventSources* class is populated as well, which internally makes sure to invoke the correct generated private factory method based on the type parameter `T`.

As mentioned earlier, a more elegant approach would be to extend the client with generic operations. Nevertheless, this code generation approach provides the required functionality. Additionally, because the code generation tool is based on reflection and methods in the C# client, the generated `EventSources` can be easily updated whenever anything changes upstream in the C# client.

9.1.2 IEventSource and EventSource<T>

The `EventSource<T>` objects mentioned previously conform to the interface included in listing 5. Its main functionality is represented by the `Start` method on line 5, which can be invoked to start the event source, meaning it will establish a Kubernetes watch and start propagating events.

```
1  public interface IEventSource : IDisposable
2  {
3      string ObjectType { get; }
4      bool IsRunning { get; }
5      void Start(EventSourceHandler handler);
6      Task<IList<KubernetesV1MetaObject>> ListMetaObjects();
7  }
```

Listing 5: `IEventSource<T>` interface (*EventSource.cs*)

As explained in section 8.2, the event propagation is facilitated by a subscriber providing a handler. This is represented in fig. 8.2 by the `Start` method requiring a `EventSourceHandler` delegate, whose signature dictates that any such handler must accept the event type and the object metadata related to the event being propagated. The signature can be seen in listing 6 below.

```
public delegate Task EventSourceHandler(EventType et, KubernetesV1MetaObject obj);
```

Listing 6: Signature of *EventSourceHandler* delegate (*EventSource.cs*)

The `IEventSource` is non generic, as resource type information is not necessary since in this implementation events only hold metadata of objects (such as name, namespace etc, as per section 6.2.2). This allows for even more general operations with event sources, such as putting them into a single collection and starting them all at once.

The generic API for creating new sources, contained in the `EventSources` (plural) class and described in previous section creates objects of class `EventSource<T>`. This class implements

IEventSource and provides additional functionality for event propagation on top of the Watcher<T> objects from the C# client.

```
 1  public void Start(EventSourceHandler handler) {
 2      InitWatcher(handler);
 3  }
 4  private void InitWatcher(EventSourceHandler handler)
 5  {
 6      void OnError(Exception _)  { InitWatcher(handler); }
 7      void OnClosed()            { InitWatcher(handler); }
 8      if (!IsRunning)
 9      {
10          _watcher = _watchMaker(handler.Invoke, OnError, OnClosed);
11      }
12  }
```

Listing 7: Implementation of `Start(EventSourceHandler handler)` in `EventSource<T>` (*EventSource.cs*)

The main functionality contained in the `Start` method of any `EventSource<T>` object is the initialization and management of Kubernetes watch. This is achieved using a helper method `InitWatcher`, seen in listing 7. There, the `_watchMaker` function (passed and stored as `WatchMaker` in the generated factory methods from section 9.1.1) is used to create a `Watcher<T>` object. To create it, the handler provided in `Start` is passed as the main event callback (line 10).

The `_watchMaker` function also accepts a callback for when an unrecoverable error happens or when the watch is closed. For those callbacks, functions defined on lines 6 and 7 are provided, which essentially attempt to restart the watch by re-creating using a recursive call to `InitWatcher`.

9.1.3 ISharedEventSource and SharedEventSource<T>

During the implementation process, it was discovered that it would be more efficient to minimize the amount of event sources (and therefore Kubernetes watches) created. In order to be able to reuse a single `EventSource<T>` (or `IEventSource`), it needed to be possible to fan-out events to multiple subscribers.

For this purpose, a new interface and class have been built, namely `ISharedEventSource` and `SharedEventSource<T>` respectively. As shown in listing 8, the interface is similar to `IEventSource` from listing 5.

The main difference is in the `Start` method not requiring any parameters. Instead, the `EventSourceHandler` is to be passed to the `Subscribe` method (line 5) by each subscriber.

The `Subscribe` method is implemented in the `SharedEventSource<T>` class, which is a wrapper for `EventSource<T>`. As seen in listing 9, each `Subscribe` assigns a unique id and stores the passed `handler` (lines 3-4).

```
public interface ISharedEventSource : IDisposable
{
    string ObjectType { get; }
    bool IsRunning { get; }
    IDisposable Subscribe(EventSourceHandler handler);
    void Start();
    Task<IList<KubernetesV1MetaObject>> ListMetaObjects();
}
```

Listing 8: `ISharedEventSource` interface (*SharedEventSource.cs*)

It returns a wrapper object which implements the `IDisposable` interface. This allows subscribers to cancel their subscription if necessary by calling `.Dispose()` on the returned object, which will result in the handler being removed from the `SharedEventsource<T>`.

In Listing 9 the `Start` method is implemented by calling the `Start(EventSourceHandler handler)` method on the underlying `EventSource<T>`, passing in a special handler (line 7). This handler (`PropagateEvent`) is responsible for invoking all the stored handlers (i.e. subscribers) in parallel on each event (lines 10-15).

```
 1  public IDisposable Subscribe(EventSourceHandler handler)
 2  {
 3      var id = Guid.NewGuid();
 4      AddHandler(id, handler);
 5      return new SharedEventSourceSubscription(() => RemoveHandler(id));
 6  }
 7  public void Start() { _source.Start(PropagateEvent); }
 8  private async Task PropagateEvent(EventType et, KubernetesV1MetaObject obj)
 9  {
10      var tasks = new List<Task>();
11      foreach(var handler in _handlers.Values)
12      {
13          tasks.Add(handler(et, obj));
14      }
15      await Task.WhenAll(tasks);
16  }
```

Listing 9: `Subscribe`, `Start`, and `PropagateEvent` methods of `SharedEventSource<T>` (*SharedEventSource.cs*)

9.2 Controller

Having presented the Event Sources component and related code, this section will describe the implementation of the most central component, the Controller. This will include details of the work queue implementation, the controller configuration, and the reconciliation loop. These details will also cover how the Reconciler interface (section 8.4) is used within Controllers.

9.2.1 WorkQueue

The particular queue implementation used in Controllers was based on and inspired by a similar queue implementation from the Go Kubernetes client (section 2.1), which is also used in the Controller Runtime (section 2.2). A custom queue implementation was needed due to properties and semantics that are specifically use-full when processing events in Controllers.

Firstly, because the queue is meant to be used for triggering reconciliation logic based on changed/added/deleted Kubernetes objects, it can be made to only hold *unique objects*. The reconciliation process is designed to be only triggered using object metadata, not the actual object data, i.e. the reconciliation function is only notified about which object needs to be reconciled, not how.

This implies that the reconciliation function must on its own fetch the current state of the changed object and reconcile it in an idempotent process (applying it multiple times should give the same results). For these reasons, it is wasteful for reconciliation to be requested for the same object multiple times.

Secondly, because the reconciliation function usually performs side-effects, the same object should never be in the process of being reconciled by more than one instances of the reconciliation function. This means that the queue needs to keep track of items being processed and not enqueue identical items until the processing has finished.

Therefore, the queue is defined by three main operations, `Add`, `Take`, and `MarkProcessed`, as well as supporting internal datastructures.

ValueSet

The `ValueSet` class was created as a set implementation that is based on object value comparison (as opposed to e.g. a hash-based set). The implementation is confined to the `WorkQueue` class and uses a C# dictionary under the hood and can be seen in listing 10. The `ValueSet` enforces uniquenes by tracking items as dictionary keys with arbitrary values (a single byte flag, line 3) and is used for the `_items` and `_processing` fields in `WorkQueue`.

Queue

The `BufferBlock`[2] from the `System.Threading.Tasks.Dataflow` .NET library has been selected to implement the FIFO queueing mechanism of items. This particular data structure has chosen

[2]https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.dataflow.bufferblock-1?view=netcore-3.1

```
class ValueSet<T>
{
    private const byte FLAG = 1;
    private Dictionary<T, byte> _dict = new Dictionary<T, byte>();
    public int Count => _dict.Count;
    public bool Has(T key) => _dict.ContainsKey(key);
    public bool Add(T key) => _dict.TryAdd(key, FLAG);
    public bool Delete(T key) => _dict.Remove(key);
}
```

Listing 10: `ValueSet` implementation (*WorkQueue.cs*)

due to its better support for asynchronous operations compared to the standard `Queue<T>` in .NET. In `WorkQueue`, it is represented as the `_queue` field.

TryAdd

The method for adding items to the queue can be seen in listing 11. Uniqueness is enforced on line 5, where a check is made against the `_items` set. If the item already exists in the set, it will not be added to the queue. If not already added, the item is always added to the `_items` set (line 6), however, it is only put into the queue if it's not already being processed by some queue consumer (exists in the `_processing` set).

```
 1  public async Task<bool> TryAdd(T item)
 2  {
 3      using(await _mutex.Use())
 4      {
 5          if (_items.Has(item)) return false;
 6          _items.Add(item);
 7          if(!_processing.Has(item)) _queue.Post(item);
 8          return true;
 9      }
10  }
```

Listing 11: `TryAdd` method of `WorkQueue` (*WorkQueue.cs*)

TryTake

The `TryTake` method can be used by consumers to dequeue items asynchronously. Additionally, it is designed to block (in the sense the consumer waits) until an item is available in the queue. This provides an easier and less error-prone way for writing consumers, as queue polling is already handled.

To wait for an item, the `ReceiveAsync` method of the `_queue` block is used (line 6). Once execution is resumed, the item is considered consumed and is then added to the `_processing` set and removed from the `_items` set, marking it as being processed and allowing an identical item to be submitted to the queue again (lines 9-10).

```
 1 public async Task<(bool,T)> TryTake(CancellationToken ct = default)
 2 {
 3     var item = default(T);
 4     try
 5     {
 6         item = await _queue.ReceiveAsync(ct);
 7         using(await _mutex.Use())
 8         {
 9             _processing.Add(item);
10             _items.Delete(item);
11         }
12         return (true, item)
13     }
14     // Thrown when _queue empty and completed
15     catch (InvalidOperationException) { }
16     return (false, item);
17 }
```

Listing 12: `TryTake` method of `WorkQueue` (*WorkQueue.cs*)

MarkProcessed

Once a consumer finishes processing an item, it must signalize it to the queue, by calling the `MarkProcessed` method from listing 13. This removes the item from the `_processing` set (line 5). If the item is present in the `items`, it means it must have been added during processing and since processing is finished, it can be processed again and can therefore be added to `_queue` (line 7).

```
 1 public async Task<bool> MarkProcessed(T item)
 2 {
 3     using(await _mutex.Use())
 4     {
 5         var success = _processing.Delete(item);
 6         // Requeue if the item was re-added while processing
 7         if(_items.Has(item)) _queue.Post(item);
 8         return success;
 9     }
10 }
```

Listing 13: `MarkProcessed` method of `WorkQueue` (*WorkQueue.cs*)

Concurrency

As may be seen throughout listings 11 to 13, a mutual exclusion object (mutex) in the _mutex field is used to ensure that the different data structure are never accessed concurrently by multiple threads.

The _mutex field is implemented as a semaphore object (using .NET SemaphoreSlim[3]) with the thread limit set to 1. The .Use() method is a convenience extension method, which facilitates acquiring the mutex and returning an IDisposable wrapper so that mutex usage can be written using a C# **using** block. The IDisposable returned implements the Dispose method by calling the Release method on the SemaphoreSlim object (_mutex field).

9.2.2 Controller configuration

The configuration and initialization of a controller is split into two functions. Firstly, the application-provided implementation of IReconciler (section 8.4) can be provided to a Controller object's constructor. Furthermore, resources that should be watched can be configured via a series of one or more calls of the AddWatch method, shown in listing 14.

```
1 public void AddWatch<T>(string @namespace, EnqueueingHandler handler)
2 {
3     var source = Cache.GetNamespacedFor<T>(@namespace);
4     _watches.Add(new WatchInfo(source, @namespace, handler));
5 }
```

Listing 14: AddWatch method of Controller (*Controller.cs*)

As seen in listing 14, watch configurations are tracked in a collection of WatchInfo (defined as private class in Controller) objects. As shown in listing 15, these contain an instance of ISharedEventSource (from section 9.1.3) as well as an EnqueueingHandler.

The ISharedEventSource instance is obtained on line 3 of listing 14. Based on the Kubernetes resource type provided in form of the type parameter T, the instance is fetched using a reference to a cache. Details about the cache used will be presented later in section 9.3.

EnqueueingHandler

The mentioned EnqueueingHandler is a delegate that is used to influence how reconciliation requests are enqueued based on a triggering event. As per the signature from listing 16, it supports functions which accept an event type, metadata about a Kubernetes object, and a work queue of items of type ReconcileRequest.

[3]https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphoreslim?view=netcore-3.1

```
class WatchInfo
{
    public ISharedEventSource Source { get; }
    public string Namespace { get; }
    public EnqueueingHandler Handler { get; }
    public WatchInfo(ISharedEventSource source, string @namespace, EnqueueingHandler handler)
    {
        Source = source; Namespace = @namespace; Handler = handler;
    }
}
```

Listing 15: `WatchInfo` class definition in `Controller` (*Controller.cs*)

```
public delegate Task EnqueueingHandler(
    EventType et, KubernetesV1MetaObject obj, IWorkQueue<ReconcileRequest> queue);
```

Listing 16: Signature of `EnqueueingHandler` delegate (*Handlers.cs*)

By allowing to pass an `EnqueueingHandler` to `AddWatch`, users may control how requests are added to the queue. However, most users are expected to simply want the request to be put on the queue without additional logic.

Alternatively, another common use-case might be to react to an event of object A by reconciling its parent (owner) object O_A. In that case, the handler should add a reconciliation request for object O_A instead.

For users who do not wish to specify a custom `EnqueueingHandler`, default handlers supporting two mentioned use-cases are available to be constructed with helper methods on the `Handlers` static class. These methods may be seen in listing 17.

`Handlers.EnqueueForObject()` returns a handler which simply constructs a `ReconcileRequest` object based on the event's object metadata (line 5) and adds it to the queue (line 6). On the other hand, `Handlers.EnqueueForOwner` checks the event object's metadata for owner references (line 15). If the object is owned by one or more different objects, a `ReconcileRequest` is created and added to the queue for each of those (lines 17-24).

AddWatch usage

To configure the controller to watch events for a specific resource, the `AddWatch` function can be called as illustrated in listing 18. There, a use-case is considered in which a resource represented by the `LightBulb` type is an owner of a resource represented by the `LightBulbColor` type. This could, for instance, mean that the reconciler watches `LightBulb` objects and in the process creates (children) objects of type `LightBulbColor`. In that case, `LightBulbColor` object changes should trigger reconciliation of the parent object to ensure the states have not drifted.

```
public static EnqueueingHandler EnqueueForObject()
{
    return async (et, obj, q) =>
    {
        var r = MakeRequest(obj);
        await q.TryAdd(r);
    });
}
public static EnqueueingHandler EnqueueForOwner(bool isController)
{
    return async (et, obj, q) =>
    {
        var ownerReferences = obj.Metadata.OwnerReferences;
        if (ownerReferences == null || ownerReferences.Count == 0) return;
        var requests = isController switch
        {
            true => ownerReferences
                .Where(r => r.Controller.HasValue && r.Controller == true)
                .Select(r => MakeRequest(obj, r)),
            false => ownerReferences.Select(r => MakeRequest(obj, r))
        };
        foreach (var r in requests) await q.TryAdd(r);
    });
}
```

Listing 17: Default `EnqueueingHandler` factories `EnqueueForObject` and `EnqueueForOwner` (*Handlers.cs*)

```
controller.AddWatch<LightBulb>("default", Handlers.EnqueueForObject());
controller.AddWatch<LightBulbColor>("default", Handlers.EnqueueForOwner(true));
```

Listing 18: Example usage of the `AddWatch` configuration method in `Controller` (*Controller.cs*)

9.2.3 Reconciliation

Once created and configured, a `Controller` can be started using the `Start` method shown in listing 19. First, all the configurations applied using `AddWatch` (tracked in `_watches`) are initialized by subscribing the provided `EnqueueingHandler` on the `ISharedEventSource` (lines 7-8).

Note that `Subscribe` in `ISharedEventSource` (listing 8, page 60) takes an `EventSourceHandler` (listing 6, page 58), while the handler provided in `AddWatch` and stored in `WatchInfo` is an `EnqueueingHandler` (listing 16, page 65). The call to `ToEventSourceHandler` (line 8) creates a compatible closure using the `EnqueueingHandler` and a reference to the queue.

Afterwards, one or more (depending on the value of `Concurrency` configured when creating the `Controller`) reconciliation loops are started (lines 9-11), followed by the initialization of an additional resynchronization loop on line 13. A description of both types of loops will follow later in this section.

```
public Task Start(CancellationToken ct = default)
{
    _cts = (ct == CancellationToken.None) ? new CancellationTokenSource()
        : CancellationTokenSource.CreateLinkedTokenSource(ct);

    foreach (var w in _watches) // Setup subscriptions
        w.Source.Subscribe(w.Handler.ToEventSourceHandler(Queue));

    _reconcileLoops = new List<Task>();
    for(int i = 0; i < Concurrency; i++) // Start reconcile loop(s)
        _reconcileLoops.Add(ReconcileLoop(_cts.Token));

    _resyncLoop = ResyncLoop(_cts.Token); // Start resync loop
    return Task.CompletedTask;
}
```

Listing 19: The `Start` method in `Controller` (Controller.cs)

ReconcileLoop

The mentioned reconciliation loop (`ReconcileLoop` method) is what consumes from the work queue and where reconciliation requests are passed outside of the library into the application code (via the `IReconciler` interface).

The code for the loop may be seen in listing 20. The loop essentially runs during the entire lifetime of a Controller once started (line 4) and waits for reconciliation requests to be available in the work queue in line 6 using `TryTake` (listing 12, page 63).

Once a request is available, it is passed to the application-provided `IReconciler` instance (line 8) together with a reconciliation context (line 3, currently only contains the configured C# Kubernetes client instance).

As shown in line 9 of listing 20, the result (`ReconcileResult`) returned by the application's `IReconciler` is then inspected. In case a request for a retry has been indicated, an asynchronous action is spawned to requeue the request after the indicated time (can also be immediately).

Finally, once the reconcilliation processing is done, the work queue is notified using the `MarkProcessed` operation (listing 13 on page 63), allowing requests for the same object to be added to the queue in case they have been submitted during processing.

```
private async Task ReconcileLoop(CancellationToken ct)
{
    var ctx = new ReconcileContext(Client);
    while (!ct.IsCancellationRequested)
    {
        var (success, req) = await Queue.TryTake(ct);
        if (!success) continue;
        var result = await Reconciler.Reconcile(ctx, req);
        if(result.Requeue) Requeue(req, result.RequeueAfter, ct);
        await Queue.MarkProcessed(req);
    }
}
```

Listing 20: The `ReconcileLoop` method in `Controller` (Controller.cs)

IReconciler, ReconcileRequest and ReconcileResult

As was just described, the controller (reconciliation loop) passes `ReconcileRequest` objects to the `Reconcile` method of the `IReconciler` which returns a `ReconcileResult` object. The interface and the two classes are shown in listing 21.

As shown in the listing, the `ReconcileRequest` passed to `Reconcile` uniquely identifies a to-be-reconciled Kubernetes object using its metadata (as introduced in section 6.2.2). The `ReconcileResult` must be created and returned by the application's `IReconciler` implementation, where it can be indicated whether the request should be requeued (i.e. retried). Optionally a time delay before the retry can also be specified.

```
public interface IReconciler
{
    Task<ReconcileResult> Reconcile(ReconcileContext context, ReconcileRequest request);
}
public class ReconcileRequest
{
    public string ApiVersion { get; set; }
    public string Kind { get; set; }
    public string Namespace { get; set; }
    public string Name { get; set; }
}
public class ReconcileResult
{
    public bool Requeue { get; set;}
    public TimeSpan RequeueAfter { get; set;}
}
```

Listing 21: `IReconciler`, `ReconcileRequest` and `ReconcileResponse` (IReconciler.cs, ReconcileRequest.cs, ReconcileResponse.cs)

Resynchronization

The resynchronization loop, first shown in line 13 of the controller's `Start` method (listing 19, page 67, periodically uses the one-time `LIST` operations implemented in event sources to get all the Kubernetes objects that are configured to be watched and, for each object, it enqueues a request for reconciliation.

Without this feature, reconciliation would only be triggered based on Kubernetes-side events. This would mean that out-of-band (not reflected in Kubernetes resources) changes to the actual state (e.g. platform configuration/APIs) would not be corrected until a change of the Kubernetes object triggers reconciliation.

With the resynchronization loop, on the other hand, this kind of state drift is corrected, as all relevant (watched) Kubernetes resource objects are pro-actively and periodically reconciled. For brevity, the relevant code will not be shown here, but may be seen in appendix A.1.

9.3 Manager

In this section, we will present the implementation details of the *Manager* component, which enables co-hosting of multiple KubeSharper controllers, as per the design described in section 8.5.

The manager acts as a host/container for controllers which allows them to be started/stopped together. Additionally, the manager allows certain dependencies and connections to be shared among controllers, which is meant to simplify the usage of KubeSharper while also making Kubernetes communication more efficient.

9.3.1 EventSourceCache

One of the main dependencies the `Manager` shares among controllers is a cache for `SharedEventSource` objects. As demonstrated by the `AddWatch` controller configuration method (listing 14, page 64), controllers use the cache whenever a watch for a new resource type or namespace is initialized.

Because a shared cache is used, then if two or more controllers require an event source for the same resource type and namespace, a single event source (and therefore a single Kubernetes connection) may be reused. This is also facilitated by the propagation of events to multiple subscribers in the `SharedEventSource` (section 9.1.3).

The `EventSourceCache` and `IEventSourceCache` interface (listing 22) expose the method `GetNamespacedFor<T>`, which is used in `AddWatch`. Whenever an event source for a given resource type and namespace pair is requested via the method for the first time, `EventSourceCache`

```
public interface IEventSourceCache : IDisposable
{
    ISharedEventSource GetNamespacedFor<T>(string @namespace);
    void StartAll();
}
```

Listing 22: `IEventSourceCache` interface (*EventSourceCache.cs)*

initializes the respective `SharedEventSource` instance using the `EventSources` class (section 9.1.1). Additionally, it stores a reference to the instance in a dictionary, under a key composed of the type `T` and the namespace.

Then, on any subsequent call to `GetNamespacedFor<T>` for the same `T` and namespace, the stored instance is reused by retrieving it from the dictionary instead of initializing it.

9.3.2 Kubernetes client configuration

Apart from the event sources and Kubernetes API connections, the `Manager` also shares an instance of the Kubernetes client with controllers. A path to the user-specific Kubernetes configuration file (kubeconfig) can be passed to the `Manager` when it's being created using the static `Manager.Create` method, shown in listing 23. The file is parsed and client created using the Kubernete C# library classes in lines 4 and 5.

```
1 public static async Task<Manager> Create(string kubeConfigPath)
2 {
3     var fi = new FileInfo(kubeConfigPath);
4     var config = await KubernetesClientConfiguration.BuildConfigFromConfigFileAsync(fi);
5     var client = new Kubernetes(config);
6     var sources = new EventSources.EventSources();
7     var cache = new EventSourceCache(sources, client);
8     return new Manager(client, cache);
9 }
```

Listing 23: Static method `Manager.Create` (*Manager.cs)*

9.3.3 Controller relationship via IStartable

To share the Kubernetes client and `EventSourceCache` instances, each controller must have a reference to and be added to a `Manager`. The manager maintains a collection of objects that satisfy the `IStartable` interface, which requires an implementation of a `Start` method and therefore is satisfied by `Controller` (via its `Start` method from listing 19, page 67).

```
 1 public Controller(Manager manager, ControllerOptions opts)
 2 {
 3     Client = manager.Client;
 4     Cache = manager.Cache;
 5     Reconciler = opts.Reconciler;
 6     ResyncPeriod = opts.ResyncPeriod;
 7     Concurrency = opts.Concurrency;
 8     Queue = new WorkQueue<ReconcileRequest>();
 9     manager.Add(this);
10 }
```

Listing 24: `Controller` constructor (*Controller.cs*

The `Manager` class exposes an `Add` method for this collection, which is used in the constructor `Controller`, as shown in line 9 of listing 24.

9.3.4 Starting controllers

Once one or more controllers have been created (constructor from listing 24) and configured (`AddWatch`, listing 14, page 64) and therefore added to a `Manager` object, its `Start` method can be used.

```
 1 public async Task Start(CancellationToken ct = default)
 2 {
 3     _cts = (ct == CancellationToken.None)  ? new CancellationTokenSource(),
 4         : CancellationTokenSource.CreateLinkedTokenSource(ct);
 5     // Start all controllers
 6     var tasks = _startables.Select(s => s.Start(_cts.Token));
 7     await Task.WhenAll(tasks);
 8     // Start all initialized event sources
 9     Cache.StartAll();
10 }
```

Listing 25: The `Start` method of `Manager` (*Manager.cs)*

As seen listing 25, `Manager.Start` starts all owned controllers (line via `Controller.Start` from listing 19 on page 67, which is followed by initializing all the configured event sources and Kubenretes connections via the cache (line 9).

The manager thus allows developers to have the entire KubeSharper layer of their application contained and controller via a single object, which is meant to improve the experience of using the library.

9.4 Custom resources

In order to keep the implementation idiomatic and provide a pleasant developer experience in the C#/.NET, effort has been put throughout this chapter to make sure that the KubeSharper library APIs are generic, allowing for configuration based on C# types and type system. While the C# Kubernetes client provides C# classes for native Kubernetes resource types, it handles custom resources dynamically.

In KubeSharper, custom resources can be instead handled in a type-safe manner, by allowing developers to extend the `CustomResource` class, whose details may be seen in listing 26. Per the listing, the class can be extended to create a domain-specific custom resource in the application by providing classes for the common `Spec` and `Status` properties of custom Kubernetes objects (section 6.2.2)).

```
public abstract class CustomResource<TSpec, TStatus> : KubernetesObject
    where TSpec : class, new()
    where TStatus : class, new()
{
    public V1ObjectMeta Metadata { get; set; }
    public TSpec Spec { get; set; }
    public TStatus Status { get; set; }
}
```

Listing 26: Base `CustomResource` class for defining custom resource objects in KubeSharper (*CustomResource.cs*)

Additionally, KubeSharper (and Kubernetes in general) requires some knowledge about the corresponding custom resource *definition* in order to be able to handle a custom resource object. In KubeSharper, this data can be easily expressed in the custom resource class by the application developer using a KubeSharper-provided `CustomResourceDefinition` class attribute.

```
[CustomResourceDefinition("databases.io", "v1", "databases", "database")]
public class Database : CustomResource<DatabaseSpec, DatabaseStatus> {}
public class DatabaseSpec
{
        public string Name { get; set; }
        public string InstanceSize { get; set; }
}
public class DatabaseStatus
{
    public string ConnectionString { get; set; }
    public bool IsReady { get; set; }
}
```

Listing 27: Example *Database* custom resource and CRD, defined using KubeSharper

Listing 27 contains a simple example of a possible `Database` custom resource, which could for example be used to implement a custom controller that manages internal database instances in a company via state reconciliation, Kubernetes and KubeSharper.

There, the class `Database` is created by extending `CustomResource`. `DatabaseSpec` and `DatabaseStatus` classes are defined and provided as the spec and status type parameters of `CustomResource`. Additionally, the `CustomResourceDefintion` is used in line 1 to define the API group, version, plural name and singular name attributes (section 6.4.1) of the custom resource respectively.

9.5 CRD Generator

The CRD Generator, described in section 8.3 is meant to serve as an executable tool for developers, which can be used to automatically generate Kubernetes manifests (in YAML) based on the type information of custom resources defined in the application code using C#.

In the current implementation, the tool accepts two arguments: path to the DLL file containing compiled .NET assembly of the application and an output path where YAML manifest files should be written.

```csharp
static void Main(string[] args)
{
    var (assemblyPath, outputPath) = (args[1], args[2])
    var typeCrds = Assembly.LoadFrom(assemblyPath).GetTypes()
        .Select(t => (Type: t, Crd: CustomResourceDefinition.For(t)))
        .Where(tuple => tuple.Crd != null);
    foreach(var (type, crd) in typeCrds)
    {
        var spec = type.GetProperty("Spec");
        var specSchema = GetSchemaFor(spec.PropertyType).ToObjectGraph();

        var status = type.GetProperty("Status");
        var statusSchema = GetSchemaFor(status.PropertyType).ToObjectGraph();

        var crdObj = MakeCrdObj(crd, type.Name, specSchema, statusSchema);
        var serializer = new SerializerBuilder()
            .ConfigureDefaultValuesHandling(DefaultValuesHandling.OmitNull)
            .WithNamingConvention(CamelCaseNamingConvention.Instance)
            .Build();
        var yaml = serializer.Serialize(crdObj);
        File.WriteAllText(Path.Join(outputPath, $"{crd.Singular}.yaml"), yaml);
    }
}
```

Listing 28: CRD Generator code (*Program.cs*)

As seen in listing 28, the DLL is first loaded (line 4) and scanned for all objects annotated using the `CustomResourceDefinition` attribute shown in listing 27 in the previous section. For each such type, the attribute data is extracted and a schema object is built for the `Spec` and `Status` properties based on the structure of their types.

To get the schema, a helper method `GetSchemaFor(Type)` is used (lines 10 and 13) which internally uses the *Newtonsoft Json .NET Schema*[4] library to build a schema objects based on .NET types.

Using the `MakeCrdObj` helper method (line 15), we build a C# representation of the `CustomResourceDefinition` Kubernetes YAML structure, based on the KubeSharper `CustomResourceDefinition` attribute data, as well as the schemas and the name of the type.

Finally, the combined and returned object is serialized into a YAML file using the YamlDotNet[5] library and the file is then written to the specified location.

9.6 Dependecy Injection and KubeSharper usage

As discussed in section 7.3, supporting the common pattern of using dependency injection to configure applications and dependencies in C#/.NET could make using KubeSharper more familiar in the ecosystem, and thus further improving the accessibility of custom reconciliation on Kubernetes.

To address this, KubeSharper provides helper extension methods on the `IServiceCollection` .NET dependency injection interface. As seen in the example in listing 29, these extensions enable configuration of the manager, as well as the controller and its watches, to be done using standard .NET dependency injection.

In listing 29, a simple controller is configured, which watches `LightBulb` (custom) resource objects (line 17) in the default namespace and reconciles them using the `LightBulbReconciler` (lines 10, 14).

Furthermore, the extension methods are implemented so that the configured `Manager` is registered in the `IServiceCollection` as an implementation of the `IHostedService`[6] interface. This delegates the management of services to the .NET platform, meaning the application developer only needs to use the dependency injection integration and the configured KubeSharper components will be embedded and started automatically with the application.

[4]https://www.newtonsoft.com/jsonschema

[5]https://github.com/aaubry/YamlDotNet

[6]https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.hosting.ihostedservice?view=dotnet-plat-ext-3.1

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args) .ConfigureServices((hostContext, services) =>
    {
        services.AddSingleton<LightBulbReconciler>();
        services.KubeSharperManager(@"C:\Users\vao\kubeconfig.yaml")
            .WithController((sp, ctrl) =>
            {
                ctrl.Options.Reconciler = sp.GetRequiredService<LightBublReconciler>();
                ctrl.Options.ResyncPeriod = TimeSpan.FromMinutes(60);
                ctrl.Options.Concurrency = 1;
                ctrl.AddWatch<LightBulb>("default", Handlers.EnqueueForObject());
            }).Add();
    });
```

Listing 29: Example configuration custom resource and CRD, defined using KubeSharper

The full implementation of the extension methods and the `IHostedService` .NET hosting integration has been included in appendix A.2.

9.7 Implementation Summary

This chapter presented the details of how the different components and aspects of KubeSharper design have been implemented using the C# language and .NET platform.

First and foremost, the implementation covered the main functional requirements, such as allowing developers to configure one or more custom controllers and integrate them with application-provided reconciliation functions and custom resource structures.

Furthermore, emphasis has also been put on developer experience and smooth integration with established .NET patterns. This included providing a fully generic API for configuring both native and custom Kubernetes resources, as well enabling KubeSharper dependencies to be injected as part of standard .NET application setup.

Overall, the implementation of KubeSharper developed as part of this thesis should provide the necessary abstractions to significantly improve the accessibility of custom reconciliation implementations on Kubernetes for C# use-cases. The next chapter will evaluate this in detail by building and reviewing two fully-contained use-cases.

CHAPTER 10

Evaluation

In this chapter, the KubeSharper SDK, whose implementation details was just presented in chapter 9, will be evaluated by implementing several representative state reconciliation use-cases.

Accessibility and ease-of-implementation is central to the problem of this thesis, which is why the method of evaluation will be mainly qualitative. Each of the built use-cases, including the development process and experience will be reviewed against the goals and requirements defined in chapter 7.

10.1 Use-case #1: Cloud Redis Operator

In the first example, the application uses a custom Kubernetes resource to define a Redis[1] in-memory key-value store instance using the *Memorystore for Redis*[2] service offered by the Google Cloud Platform.

10.1.1 Overview

The overview of the solution is depicted in fig. 10.1. The application extends Kubernetes with a custom resource type `CloudRedis` which holds the definition of a Redis instance in GCP. Fur-

[1] https://redis.io/

[2] https://cloud.google.com/memorystore

thermore, the application uses the KubeSharper SDK to configure a controller which, first of all, watches all `CloudRedis` objects in Kubernetes, and secondly, executes custom reconciliation logic via the `CloudRedisReconciler` (an implementation of `IReconciler`).

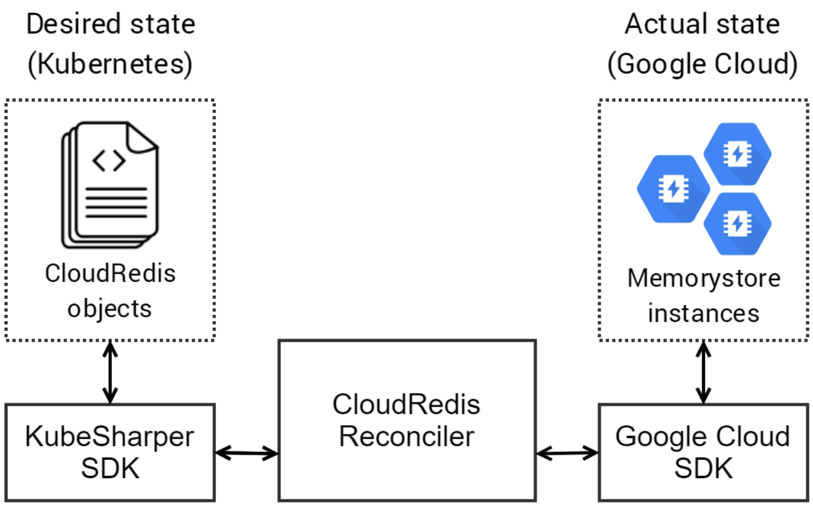


Figure 10.1: Cloud Redis Operator overview

10.1.2 Custom resource representation

The C# class used defined, with the help of KubeSharper SDK, to work with `CloudRedis` custom objects may be seen in listing 30 below.

```
[CustomResourceDefinition("operator.cloudredis.io", "v1", "cloudredises", "cloudredis")]
public class CloudRedis : CustomResource<CloudRedisSpec, CloudRedisStatus> {}
public class CloudRedisSpec
{
    public string Name { get; set; }
    public int MemorySizeGb { get; set; }
    public Tier Tier { get; set; }
}
public class CloudRedisStatus
{
    public string Id { get; set; }
    public string State { get; set; }
    public DateTime CreatedAt { get; set; }
    public string Location { get; set; }
    public string Host { get; set; }
    public int Port { get; set; }
}
```

Listing 30: `CloudRedis` class definition (*CloudRedis.cs*)

The `CodeRedis` class inherits from KubeSharper's `CustomResource` and provides the `CloudRedisSpec` and `CloudRedisStatus` types as the required spec and status types respectively. The spec type provides a basic set of attributes to define an instance in GCP (lines 5-7). While many more attributes can be specified, this keeps the example concise and does not affect evaluation.

The status class defines six properties (lines 11-16). These mainly hold information known after the real GCP instance is created and these information also represent what is generally needed to use and connect to the instance.

The KubeSharper `CustomResourceDefintion` attribute is used in line 1 to express the Kubernetes CRD metadata (API group, version, plural name and singular name).

Installation

To install the CRD into a cluster, a YAML manifest of kind `CustomResourceDefinition` corresponding to `CloudRedis` from listing 30 should be defined, including a schema definition for the entire structure of the custom resource objects (spec, status, etc) and applied via Kubernetes API. The full YAML definition has been included in appendix B.1.

Using the CRD Generator (section 9.5), the manifest may be generated given the application assembly to get a YAML file which can subsequently be applied using e.g. the Kubernetes command-line client, as shown in section 7.3.4.

```
> ./kubesharper-generate ./bin/Release/netcoreapp3.1/CloudRedisOperator.dll ./yaml/
> kubectl apply -f ./yaml/cloudredis.yaml
```

Listing 31: Generating and installing YAML manifest for `CloudRedis` using KubeSharper CRD generator and `kubectl`

10.1.3 Configuration

The application has been configured using the .NET dependency injection and hosting extensions described in section 9.6, seen in listing 32. First, application-specific implementations are registered such as the Google Cloud Redis client library (line 1), the reconciler itself (line 3) and a helper class (line 2).

The KubeSharper configuration can be seen in lines 4-11, where a manager is initialized with a kubeconfig file, and a single controller is created. The controller is setup to watch `CloudRedis` objects (line 7) and use the `CloudRedisReconciler` (line 8), with additional settings in lines 9-10.

```
services.AddTransient(_ => CloudRedisClient.Create());
services.AddTransient<RedisInstanceManager>();
services.AddSingleton<CloudRedisReconciler>();
services.KubeSharperManager(Configuration["kubeconfigPath"])
.WithController((sp, cfg) =>
{
    cfg.Watch<CloudRedis>("default", Handlers.EnqueueForObject());
    cfg.Options.Reconciler = sp.GetRequiredService<CloudRedisReconciler>();
    cfg.Options.Concurrency = 2;
    cfg.Options.ResyncPeriod = TimeSpan.FromHours(1);
}).Add();
```

Listing 32: Configuration of KubeSharper in the Cloud Redis Operator using dependency injection (*Program.cs*)

10.1.4 Reconciler implementation

The mentioned `CloudRedisReconciler` implementation can be seen in listing 33. The listing includes the top-level `IReconciler.Reconcile` method. There, first of all, the object to be reconciled is fetched according to the request (lines 3-5, 8).

Then, a helper class in the application (`_cloudRedis`) is used to either create a new Redis instance or modify and existing one according to the `CloudRedis` object fetched (line 20). Once the GCP SDK operations are finished, a helper method is used to update the `CloudRedis` object in Kubernetes so that its status reflects the instance data (line 21).

Finalizers

To handle cleanup whenever `CloudRedis` Kubernetes objects are removed, the concept of *finalizers* in Kubernetes is used. All `CloudRedis` are always updated to have a finalizer key in their metadata (ensured in lines 9-12). Whenever an object in Kubernetes is requested to be deleted, it will instead by only marked for deletion (by setting the `DeletionTimestamp` in metadata). Once the finalizer key is removed, the object will be removed automatically.

The reconciler implementation uses this mechanism and checks the deletion timestamp (line 13). If set, the reconciler destroys the Google Cloud (line 15) and removes the finalizer key using a local helper method (line 17). In case deletion fails, a delayed retry will be scheduled using KubeSharper's `ReconcileResult` (line 16).

```
public async Task<ReconcileResult> Reconcile(ReconcileContext ctx, ReconcileRequest r)
{
    var crd = CustomResourceDefinition.For<CloudRedis>();
    var response = await ctx.Client.GetNamespacedCustomObjectWithHttpMessagesAsync(
        crd.Group, crd.Version, r.Namespace, crd.Plural, r.Name);
    if(response.Response.StatusCode == HttpStatusCode.NotFound)
        return new ReconcileResult();
    var obj = ((JObject)response.Body).ToObject<CloudRedis>();
    if(!obj.Metadata.Finalizers?.Contains(FINALIZER) ?? true)
    {
        await PatchAddFinalizer(ctx.Client, obj);
    }
    else if(obj.Metadata.DeletionTimestamp.HasValue)
    {
        var deleted = await _cloudRedis.Delete(obj);
        if(!deleted) return ReconcileResult(TimeSpan.FromMinutes(1));
        await PatchRemoveFinalizer(ctx.Client, obj);
        return new ReconcileResult();
    }
    var instance = await _cloudRedis.CreateOrUpdate(obj);
    await PatchStatus(ctx.Client, obj, instance);
    return new ReconcileResult();
}
```

Listing 33: `CloudRedisReconciler` implementation (*CloudRedisReconciler.cs*)

10.2 Use-case #2: ACME Workload Operator

This second example is meant to illustrate a slightly difference reconciliation scenario that KubeSharper enables. This application represents a potential way to automate and enforce company-specific microservice deployment standards using reconciliation, Kubernetes and custom resources. The name uses a placeholder (ACME) for the company name, as this is meant to be a generic, yet realistic example.

10.2.1 Overview

When using Kubernetes as an internal platform for hosting applications and microservices, multiple Kuberetes resources often need to be created in order to deploy and host a single deployable unit (a microservice).

In order to automate the process and be able to influence the default or enforced values of such resources, the ACME Workload Operator enables defining a microservice using a single company-specific custom resource object, the `AcmeService`. As seen in fig. 10.2, the reconciliation process in this case is to the Kubernetes object necessary to deploy an application based on a single, company-standardized definition in form of the AcmeService custom object.

To keep this example concise, we consider a microservice to expose it's functionality over HTTP and the necessary Kubernetes resources to be Deployment[19], which facilitates hosting the running a set of application instances and Service[21], which enables those instances to be communicated with using a single, load-balanced endpoint.

In a different scenario, additional resources like ConfigMap, Secret, HorizontalPodAutoscaler, Ingress and event custom resources like a microservice-specific CloudRedis instance from use-case #1 (section 10.1) could be considered.

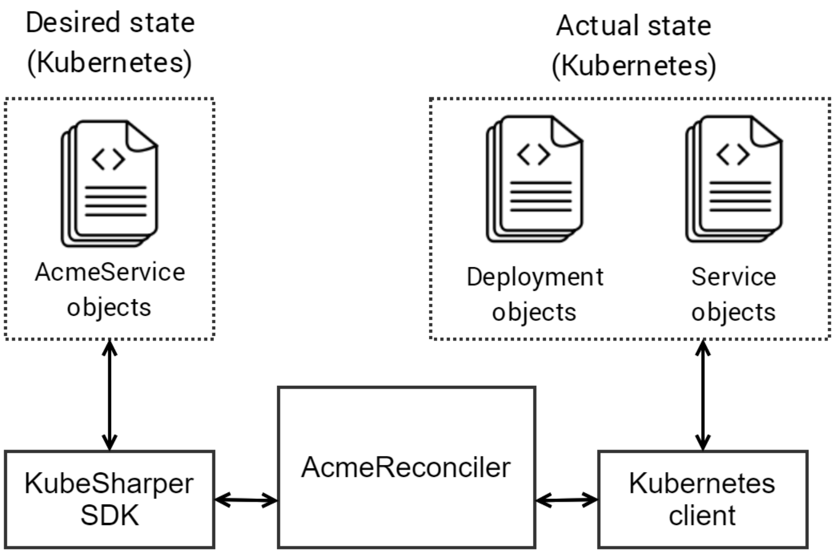


Figure 10.2: ACME Workload Operator overview

This means that, as illustrated by fig. 10.2, both the desired state and the actual state is contained within the Kubernetes cluster, similar to how native Kubernetes processes work (section 6.3). The AcmeReconciler observes custom AcmeService objects using KubeSharper SDK and then uses the Kubernetes client to create child Deployment and Service objects based on the desired definition.

The Deployment and Service objects are considered managed and owned by the parent AcmeService object and its respective controller (the ACME Workload Operator application).

10.2.2 Custom resource

The definition of the C# custom resource is similar to that of CloudRedis (listing 30, section 10.1.2). In this case, the class AcmeService inherits from CustomResource and provides its spec and status as the types AcmeServiceSpec and AcmeServiceStatus respectively.

The properties in the spec define several company-standard attributes necessary to deploy the service, such as the container image reference, number of replicas, exposed port, environment variables, labels, and others. The status type, similarly to the status from use-case #1, exposes the IP address and hostname of the service needed to connect to the service and known only after reconciliation.

For brevity, the full custom resource class as well as the corresponding YAML definition will not be presented here and are instead included in appendix B.2.

10.2.3 Configuration

The KubeSharper configuration for this use-case may be seen in listing 34. The main difference here, is that KubeSharper is configured to also watch `Deployment` and `Service` resources (line 6-7) apart from the custom `AcmeService` (line 5).

As per the usage of the `EnqueueForOwner` handler preset in lines 6-7, those additional resources are watched as children resources of `AcmeService`. This means that event when triggered by a `Deployment` or `Service` related event, the request propagated to the reconciler will be for the owning `AcmeService`.

```
1 services.AddSingleton<AcmeReconciler>();
2 services.KubeSharperManager(Configuration["kubeconfigPath"])
3 .WithController((sp, cfg) =>
4 {
5     cfg.Watch<AcmeService>("default", Handlers.EnqueueForObject());
6     cfg.Watch<V1Deployment>("default", Handlers.EnqueueForOwner(true));
7     cfg.Watch<V1Service>("default", Handlers.EnqueueForOwner(true));
8     cfg.Options.Reconciler = sp.GetRequiredService<AcmeReconciler>();
9 }).Add();
```

Listing 34: Configuration of KubeSharper in the ACME Workload Operator using dependency injection (*Program.cs*)

10.2.4 Reconciler implementation

For this use-case, as shown in listing 35 the reconciler performs a similar initial retrieval of the `AcmeService` object, based on the reconciliation request data (lines 3-4). As mentioned in the previous subsection, this request is always for a `AcmeService`, even when triggered by a change in a dependent `Deployment` or `Service` object.

Deployment and Service creation

The fetched `AcmeService` object holds data that influences the `Deployment` and `Service` object that will be created. To create the two dependent objects, helper classes `DeploymentManager` and `ServiceManager` were implemented as an abstraction over the Kubernetes C# client instance injected by KubeSharper into the reconciler (`ctx.Client`).

```
 1  public async Task<ReconcileResult> Reconcile(ReconcileContext ctx, ReconcileRequest r)
 2  {
 3      var crd = CustomResourceDefinition.For<AcmeService>();
 4      var response = await ctx.Client.GetNamespacedCustomObjectWithHttpMessagesAsync(
 5          crd.Group, crd.Version, r.Namespace, crd.Plural, r.Name);
 6      if(response.Response.StatusCode == HttpStatusCode.NotFound)
 7          return new ReconcileResult();
 8      var obj = ((JObject)response.Body).ToObject<AcmeService>();
 9      var (name, @namespace) = (obj.Metadata.Name, obj.Metadata.NamespaceProperty);
10      var deployment = await new DeploymentManager(ctx.Client).Apply(obj);
11      var service = await new ServiceManager(ctx.Client).Apply(obj);
12      await PatchAcmeServiceStatus(ctx.Client, obj, service);
13      return new ReconcileResult();
14  }
```

Listing 35: `AcmeReconciler` implementation (*AcmeReconciler.cs*)

Both helper classes have an `Apply` method, which, given an `AcmeService` object, creates or updates a corresponding `Deployment` or `Service` object, as used in lines 10 and 11 of listing 35. Afterwards, based on the created objects, the status of the `AcmeService` object is updated (line 12).

To make sure that the ACME Workload Operator can supervise the created objects and correct drifted state (e.g. incurred by manual changes), both the `Deployment` and `Service` objects are created with an `OwnerReference` to the parent `AcmeService` in their metadata.

Combined with the configuration from listing 34, this ensures that any external changes to `Deployment` or `Service` objects created by this reconciler a reconciliation, in which case the reconciler idempotently modifies their state back to the desired state defined in the `AcmeService` object.

Deletion events

In this case, since the actual/target state is purely kept within Kubernetes, no special logic is needed in case of a deletion. Because of the `OwnerReference`, whenever an `AcmeService` is deleted, so are the dependent objects.

10.3 Use-case #3: Co-hosting and cooperation

To demonstrate the manager and controller co-hosting capabilities of the KubeSharper implementation, consider a use-case where both operators (Cloud Redis and ACME Workload) would be used in the same company/environment.

In that case, it may be desired to have a single application act as both operators by *co-hosting* multiple controllers under a single KubeSharper manager. Furthermore, consider the two controllers should *cooperate*.The ACME workload operator would adopt a slightly different schema to also provision a Redis instance by creating an additional child object, in this case of type `CloudRedis`.

The configuration included in listing 36 represents this scenario. The separate configurations from listing 32 and listing 34 were combined using a pair of chained `WithController` (lines 6-12 and lines 13-19) calls on the same Manager configuration (line 5).

Additionally, to enable for the cooperation described above, an additional child resource to watch has been added for the controller with `AcmeReconciler`. With some adjustments in the reconciler code, a new `AcmeService` object will results in a `CloudRedis` resource being created as its child. Due to the configuration in line 8, this event will be propagated as a request to the `CloudRedisReconciler`, which will create the Redis instance in Google Cloud.

```
 1  // ...
 2  services.KubeSharperManager(Configuration["kubeconfigPath"])
 3  .WithController((sp, cfg) =>
 4  {
 5      cfg.Watch<CloudRedis>("default", Handlers.EnqueueForObject());
 6      cfg.Options.Reconciler = sp.GetRequiredService<CloudRedisReconciler>();
 7      cfg.Options.Concurrency = 2;
 8      cfg.Options.ResyncPeriod = TimeSpan.FromHours(1);
 9  })
10  .WithController((sp, cfg) =>
11  {
12      cfg.Watch<AcmeService>("default", Handlers.EnqueueForObject());
13      cfg.Watch<V1Deployment>("default", Handlers.EnqueueForOwner(true));
14      cfg.Watch<V1Service>("default", Handlers.EnqueueForOwner(true));
15      cfg.Watch<CloudRedis>("default", Handlers.EnqueueForOwner(true));
16      cfg.Options.Reconciler = sp.GetRequiredService<AcmeReconciler>();
17  }).Add();
```

Listing 36: Co-hosted configuration of KubeSharper (*Program.cs*)

10.4 Review and Discussion

Having presented the use-cases implemented for the purpose of evaluation, this section will review the functionality of KubeSharper as well as the development process of using it.

In section 7.3, several requirements were defined, describing the functionality and properties of KubeSharper that are necessary in order to develop the SDK and make state reconciliation in Kubernetes more accessible in C#ṄET environments.

10.4.1 Event configuration

First of all, as displayed in listing 32 and listing 34, the KubeSharper SDK allowed events about relevant resources to be configured concisely, by simply providing a C# native resource type without requiring much knowledge about Kubernetes API watch functionality or the Kubernetes C# client.

Additionally, in the configuration of use-case #2 (listing 34), we have seen that KubeSharper also supports more complex scenarios such as watching multiple resources as children of a main resource that should be reconciled.

At the same time, some issues, such as unnecessary reconciliations where an object was processed without undergoing significant changes have been observed on several occasions when testing the evaluation use-cases. Additional investigation, testing and potential introduction of caching mechanisms should be explored.

10.4.2 Reconciliation interface

As per the implementation described in section 9.2.3, and as seen for both use-cases in listing 33 and listing 35, it was possible to provide a custom reconciliation function to KubeSharper with little effort, as reconciliation code could simply be wrapped in a class that implements the single-method `IReconciler` interface provided by KubeSharper.

Furthermore, in the reconciler for use-case #1 (listing 33), we have also seen that the interface also allows a request for retry to be signalized from the application code by simply returning a relevant object to KubeSharper.

That being said, listings 33 and 35 also show room for improvement regarding this interface, especially when it comes to the passed in context. In both cases, the reconciliation code must handle some degree of Kubernetes integration that is not facilitated by KubeSharper (e.g. fetching/updating the custom resource object to be reconciled).

KubeSharper does include a configured instance of the C# Kubernetes client in the reconciliation context, which does simplify these operations to an extent. However, providing a more specialized, generically-typed client with abstractions for reconciliation-specific operations such as status patching etc could further improve the experience.

10.4.3 Controller co-hosting

As demonstrated by the third use-case and listing 36, given the current implementation of KubeSharper, it was also possible to co-host and combine multiple controllers in a single C# application/process.

As the use-case and the listing demonstrate, no special configuration or KubeSharper knowledge was needed to combine the two controllers, as their existing configuration code could simply be combined by chaining the provided .NET configuration methods.

10.4.4 CustomResourceDefinition installation

Usage of the CRD Generator tool was demonstrated in use-case #2 (listing 31). A YAML manifest for the CRD was automatically generated based on type information contained in the application's assembly. The manifest could then be used together with the Kubernetes command-line client to install the custom resource.

Additionally, this tool has proven useful several times during the implementation of these use-cases, as it allowed iterative changes made to the C# classes to be quickly applied in the Kubernetes and tested.

That being said, there have been some edge cases in which case the produced YAML output was incorrect or well formatted, which is why more extensive testing is needed.

10.4.5 Usability and Developer Experience

Overall, using KubeSharper allowed for the majority of the development of these use-cases to be done with C# code and using C# standards, practices and language idioms.

Native C# custom resources

One contributor to this was the ability to work with custom resources as native C# classes as mentioned in the requirements in section 7.3. This possibility allowed for easily writing custom helper classes and methods, which in turn enabled writing the reconciler in a more structured manner, making it more maintainable and also more readable.

On the other hand, the C# class representation imposed by the current implementation, as seen for example in listing 30, could be improved, as the enforced separation into spec and status classes with the main class bearing no properties is less elegant and harder to read.

Standard .NET startup

The second contributor was KubeSharper's integration with .NET dependency injection and configuration infrastructure, which represents a canonical way of implementing startup in .NET applications. As demonstrated by listings 32, 34 and 36, the KubeSharper setup could be done centrally, in the same place in the application where the rest of dependencies are initialized.

On top of that, the provided configuration methods have proven to be flexible enough to cover different scenarios of initializing a controller, including configuring multi-resource and parent-children events and even co-hosted controllers.

10.5 Lines-of-code analysis

Finally, to further demonstrate how KubeSharper facilitates the development of custom state reconciliation application on Kubernetes, we will perform a lines-of-code analysis on the codebases of the first two use-cases.

10.5.1 Methodology

The focus will be on the number of lines that can be considered as "KubeSharper boilerplate", that is, the part of the code that is necessary for KubeSharper configuration and Kubernetes event integration, but is not the core domain, reconciliation code which represents the business value of the application.

The code counted as *KubeSharper* ("boilerplate") code will include the elements like the KubeSharper configuration, code related to the reconciliation interface such as returned reconciliation objects, as well as KubeSharper specific attributes and parts of C# class definitions.

The code counted as *Use-case* code will include the remainder of the codebase, except some excluded lines such as C# namespace imports and definitions. Other Kubernetes-related code, such as updates/patching of the custom resource, etc, is considered specific to and dependent on the the use-case, and therefore will be included here as well.

10.5.2 Analysis

The lines-of-code analysis results may be seen in fig. 10.3 below. In the case of the ACME Workload use-case (#2), the KubeSharper code amounted to 19 lines versus 240 lines of use-case-specific code, representing 7.3% of the overall codebase.

In the case of the Cloud Redis use-case (#1), KubeSharper code consisted of 17 lines compared to the 274 lines of use-case-specific code, taking up 5.8% of the codebase in total.

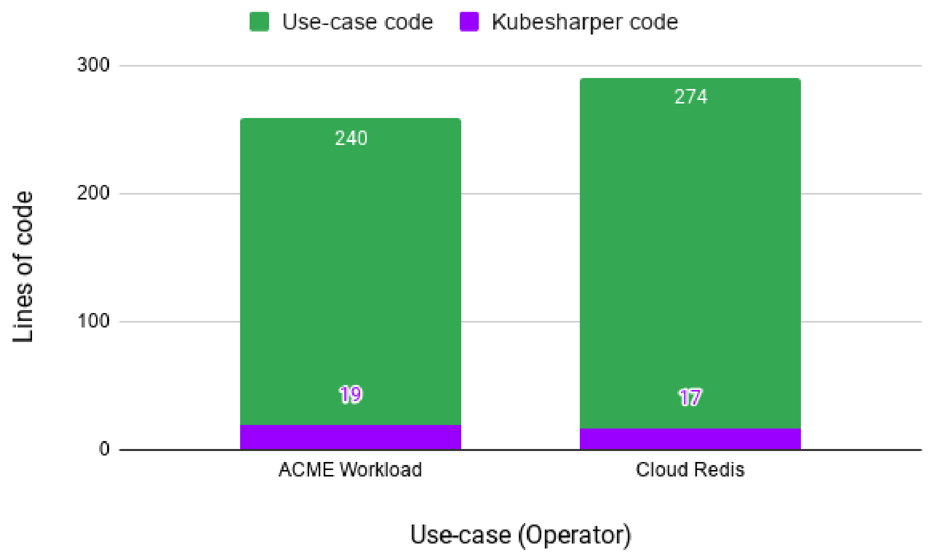


Figure 10.3: Comparison of *KubeSharper*-related vs *Use-case* lines of code

10.6 Results and Outcome

Given the review and discussion as well as the lines-of-code analysis presented in this chapter, the KubeSharper SDK has proven to be capable of simplifying the implementation of relevant use-cases.

Despite the previously mentioned shortcomings, mechanisms in need of improvement and an overall need for more testing, the SDK abstracts and automates aspects of implementing Kubernetes controllers and thus appreciably reduces the involved complexity.

CHAPTER 11

Conclusion

Recognizing a shortage of tools and a generally low accessibility in the C#/.NET ecosystem when it comes to implementing Kubernetes-based custom state reconciliation, this thesis explored designing and implementing a Software Development Kit in C# to address the problem.

The implemented solution, KubeSharper SDK, was built based on open-source software, designs and patterns from the Kubernetes community and delivers a C# library as well as an accompanying command-line tool in order to better enable implementation of the discussed use-cases for C# developers and community.

The solution has been evaluated in an examination and review of the implementation process as well as the resulting functionality of several representative applications built using the SDK. The results indicate that KubeSharper overall delivers the earlier stipulated functionality and noticeably reduces the complexity of developing the selected use-cases. Although, some implementation issues, inefficiencies and areas of improvement have also been identified.

Overall, it can be concluded that the work and product of the thesis provide a usable initial set of abstractions and tools for extending Kubernetes with C#-based custom reconciliation code. The work could also serve as a basis for future research further exploring this subject and/or building additional libraries and utilities within the area.

11.1 Future Work

For future work, multiple possibilities of further reviewing or extending the implementation could be considered.

First of all, the current implementation could undergo more extensive testing, including a performance and/or scalability analysis in comparison with the existing Go libraries . Additionally, considering parts of the evaluation results, additional focus could be put on improving the C# developer experience and abstractions provided.

Furthermore, research could also be made in relation to the Kubernetes C# client. For example, the foundation of the client could be extended with more utilities and abstraction for receiving and handling events, similarly to the official Go Kubernetes client. Parts of the functionality and code from KubeSharper could be further generalized and used as a basis for contributing similar functionality to the C# client.

APPENDIX A

Implementation

A.1 Resynchronization in Controller

```
private async Task ResyncLoop(CancellationToken ct)
{
    while (!ct.IsCancellationRequested)
    {
        await Task.Delay(ResyncPeriod, ct);
        var tasks = _watches.Select(w => ResyncWatch(w, ct, sw));
        await Task.WhenAll(tasks);
    }
}

private async Task ResyncWatch(WatchInfo watch, CancellationToken ct, Stopwatch sw)
{
    const double jitterFactor = 0.1;
    await Task.Delay(ResyncPeriod.GetJitter(jitterFactor));
    var objects = await watch.Source.ListMetaObjects();
    foreach (var o in objects)
    {
        if (ct.IsCancellationRequested) break;
        dynamic d = o;
        var metaObj = new KubernetesV1MetaObject
        {
            ApiVersion = d.ApiVersion,
            Kind = d.Kind,
            Metadata = d.Metadata
        };
        await watch.Handler(EventType.Resync, metaObj, Queue);
    }
}
```

A.2 Dependency injection extensions and IHostedService implementation

```
public static class HostingExtensions
{
    public static ManagerConfiguration KubeSharperManager(this IServiceCollection
    ↪  services, KubernetesClientConfiguration kubeConfig)
    {
        var mgr =
        ↪  Manager.Create(kubeConfig).ConfigureAwait(false).GetAwaiter().GetResult();
        return new ManagerConfiguration(mgr, services);
    }
    public static ManagerConfiguration KubeSharperManager(this IServiceCollection
    ↪  services, string kubeConfigPath)
    {
        var config =
        ↪  KubernetesClientConfiguration.BuildConfigFromConfigFile(kubeConfigPath);
        return KubeSharperManager(services, config);
    }
}

public class HostedManager : IHostedService, IDisposable
{
    private readonly Manager _manager;
    public HostedManager(Manager manager)
    {
        _manager = manager;
    }
    public Task StartAsync(CancellationToken cancellationToken) =>
    ↪  _manager.Start(cancellationToken);

    public Task StopAsync(CancellationToken cancellationToken) => Task.CompletedTask;
    public void Dispose() => _manager.Dispose();
}


public class ManagerConfiguration
{
    private readonly Manager _mgr;
    private readonly IServiceCollection _services;
    private readonly List<Action<IServiceProvider, ControllerConfiguration>>
    ↪  _configurators =
    new List<Action<IServiceProvider, ControllerConfiguration>>();
    public ManagerConfiguration(Manager mgr, IServiceCollection services)
    {
        _mgr = mgr;
        _services = services;
    }

    public ManagerConfiguration WithController(Action<IServiceProvider,
     ↪  ControllerConfiguration> configurator)
    {
        _configurators.Add(configurator);
        return this;
    }

    public IServiceCollection Add()
    {
        _services.AddHostedService(sp =>
        {
            foreach(var configurator in _configurators)
            {
                var config = new ControllerConfiguration(new ControllerOptions());
                configurator(sp, config);

                var controller = new Controller(_mgr, config.Options);
                foreach (var adder in config.WatchAdders)
                {
                    adder(controller);
                }
            }
            return new HostedManager(_mgr);

        });
        return _services;
    }
}

public class ControllerConfiguration
{
    internal readonly IList<Action<Controller>> WatchAdders = new
     ↪  List<Action<Controller>>();
    public ControllerOptions Options { get; }
    public ControllerConfiguration(ControllerOptions opts)
    {
        Options = opts;
    }

    public void Watch<T>(string @namespace, EnqueueingHandler handler)
    {
        WatchAdders.Add(ctrl => ctrl.AddWatch<T>(@namespace, handler));
    }
}
```

APPENDIX B

Evaluation

B.1 CloudRedis CRD Manifest

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: cloudredises.cloud.dev
spec:
  group: operator.cloudredis.io
  names:
    plural: cloudredises
    singular: cloudredis
    kind: CloudRedis
  scope: Namespaced
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
```

```
            name:
              type: string
            memorySizeGb:
              type: integer
            tier:
              type: integer
              enum:
              - 0
              - 1
              - 3
        status:
          type: object
          properties:
            id:
              type: string
            state:
              type: string
            createdAt:
              type: string
              format: date-time
            location:
              type: string
            host:
              type: string
            port:
              type: integer
```

B.2 AcmeService custom resource

B.2.1 C# classes

```
[CustomResourceDefinition("acme.dev", "v1", "acmeservices", "acmeservice")]
public class AcmeService : CustomResource<AcmeServiceSpec, AcmeServiceStatus> { }
public class AcmeServiceSpec
{
    public AcmeServiceSpec()
    {
        Labels = new Dictionary<string, string>();
        Environment = new Dictionary<string, string>();
    }
    public string Team { get; set; }
    public int Replicas { get; set; }
    public string ImageName { get; set; }
    public string ImageVersion { get; set; }
    public int Port { get; set; }
    public Dictionary<string, string> Labels { get; set; }
    public Dictionary<string, string> Environment { get; set; }
}
public class AcmeServiceStatus
{
    public string IP { get; set; }
    public string Hostname { get; set; }
}
```

B.2.2 YAML CRD manifest

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: acmeservices.acme.dev
spec:
  group: acme.dev
  names:
    plural: acmeservices
    singular: acmeservice
    kind: AcmeService
  scope: Namespaced
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              team:
                type: string
              replicas:
                type: integer
              imageName:
                type: string
              imageVersion:
                type: string
              port:
                type: integer
              labels:
                type: object
                additionalProperties:
                  type: string
              environment:
                type: object
                additionalProperties:
                  type: string
          status:
            type: object
            properties:
              IP:
                type: string
              Hostname:
                type: string
```

List of Figures

Bibliography

[1] Glenn Berry. Scaling sql server 2012. http://www.pass.org/eventdownload.aspx?suid=1902. Accessed: 2020-05.

[2] Pierre Carbonnelle. Pypl popularity of programming language index. https://pypl.github.io/PYPL.html, May 2020.

[3] Controller Runtime Documentation. https://godoc.org/github.com/kubernetes-sigs/controller-runtime/pkg, 2020. Accessed: 2020-05.

[4] coreos.com. Kubernetes operators. https://coreos.com/operators/, 2020. Accessed: 2020-05.

[5] etcd Authors. etcd docs | interacting with etcd. https://etcd.io/docs/v3.4.0/dev-guide/interacting_v3/#watch-historical-changes-of-keys, . Accessed: 2020-05.

[6] etcd Authors. etcd | home. https://etcd.io/, . Accessed: 2020-05.

[7] Tomas Fernandez. What is infrastructure as code? https://blog.stackpath.com/infrastructure-as-code-explainer/, 2020. Accessed: 2020-05.

[8] Simon Harrer Florian Beetz, Anja Kammer. Gitops | gitops is continuous deployment for cloud native applications. https://www.gitops.tech/. Accessed: 2020-05.

[9] Victor Coisne Francesc Campoy. An analysis of the kubernetes codebase. https://medium.com/sourcedtech/an-analysis-of-the-kubernetes-codebase-4db20ea2e9b9, 2018.

[10] Alex Giamas. From monolith to microservices, zalando's journey. *InfoQ*, 2016.

[11] GitHub Kubernetes Project. Cri: the container runtime interface. https://github.com/kubernetes/kubernetes/blob/242a97307b34076d5d8f5bbeb154fa4d97c9ef1d/docs/devel/container-runtime-interface.md, 2016. Accessed: 2020-05.

[12] GitHub: kubernetes/community. Api conventions. https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md, 2019. Accessed: 2020-05.

[13] github.com/kubernetes-sigs. Repo for the controller-runtime subproject of kubebuilder (sig-apimachinery). https://github.com/kubernetes-sigs/controller-runtime, 2020. Accessed: 2020-05.

[14] github.com/kubernetes/community. Kubernetes: New client library procedure. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/api-machinery/csi-new-client-library-procedure.md#client-capabilities, 2019. Accessed: 2020-05.

[15] github.com/operator-framework/awesome-operators. Awesome operators in the wild. https://github.com/operator-framework/awesome-operators, 2019. Accessed: 2020-05.

[16] kubernetes/client-go. Go client for kubernetes. https://github.com/kubernetes/client-go, 2020. Accessed: 2020-05.

[17] kubernetes.io. Intro to windows support in kubernetes - kubernetes. https://kubernetes.io/docs/setup/production-environment/windows/intro-windows-in-kubernetes/, . Accessed: 2020-05.

[18] kubernetes.io. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/, 2019. Accessed: 2019-12.

[19] kubernetes.io. Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, 2019. Accessed: 2019-12.

[20] kubernetes.io. Pod overview. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/, 2019. Accessed: 2019-12.

[21] kubernetes.io. Service. https://kubernetes.io/docs/concepts/services-networking/service/, 2019. Accessed: 2019-12.

[22] kubernetes.io. What is kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, 2019. Accessed: 2019-12.

[23] kubernetes.io. Client libraries. https://kubernetes.io/docs/reference/using-api/client-libraries/, 2020. Accessed: 2020-05.

[24] kubernetes.io. Controllers. https://kubernetes.io/docs/concepts/architecture/controller/, 2020. Accessed: 2020-05.

[25] kubernetes.io. Custom resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, 2020. Accessed: 2020-05.

[26] kubernetes.io. Extend the kubernetes api with customresourcedefinitions. https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/, 2020. Accessed: 2020-05.

[27] kubernetes.io. Understanding kubernetes objects. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/, 2020. Accessed: 2020-05.

[28] kubernetes.io. Authorization overview - kubernetes. https://kubernetes.io/docs/reference/access-authn-authz/authorization/#determine-the-request-verb, 2020. Accessed: 2020-05.

[29] Linux containers. Linux containers - lxc - introduction. https://linuxcontainers.org/lxc/introduction/, 2019. Accessed: 2019-12.

[30] Tony Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design. *nginx blog*, 2015.

[31] Peter Mell and Timothy Grance. The NIST definition of cloud computing - SP 800-145. *NIST Special Publication*, 2011. ISSN 00845612. doi: 10.1136/emj.2010.096966.

[32] Microsoft Docs. Dependency injection in asp.net core. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1, 2020. Accessed: 2020-05.

[33] Microsoft Docs. Background tasks with hosted services in asp.net core | microsoft docs. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-3.1&tabs=visual-studio, 2020. Accessed: 2020-05.

[34] Kief Morris. Infrastructure as code: managing servers in the cloud. *O'Reilly Media, Inc*, 2016.

[35] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014*, 2019. ISBN 9781931971102.

[36] Valér Orlovský. Uppaal cloud: Model-checking-as-a-service using kubernetes and uppaal. January 2020.

[37] Alexis Richardson. Gitops - operations by pull request. https://www.weave.works/blog/gitops-operations-by-pull-request, August 2017. Accessed: 2020-05.

[38] Roy Thomas Fielding. Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST), 2000.

[39] Daniel Smith. Kubernetes-style apis of the future. https://www.youtube.com/watch?v=S2U8GNHewpk, 2018. KubeCon + CloudNativeCon North America.

[40] Barrie Sosinsky. *Cloud Computing Bible*. 2010. doi: 10.1002/9781118255674.

[41] Michael Hausenblas Stefan Schimanski. Kubernetes deep dive: Api server – part 1. https://www.openshift.com/blog/kubernetes-deep-dive-api-server-part-1, April 2017. Accessed: 2020-05.

[42] swagger.io. Openapi specification. https://swagger.io/specification/. Accessed: 2020-05.

[43] Bobby Tables. Stay informed with kubernetes informers. https://www.firehydrant.io/blog/stay-informed-with-kubernetes-informers/, 2020. Accessed: 2020-05.

[44] TIOBE Software BV. TIOBE Index | TIOBE - The Software Quality Company. https://www.tiobe.com/tiobe-index/, May 2020.