University of Ljubljana
Faculty of Computer and Information Science

Boris Radovič

Analysis and usage of API gateways in Kubernetes

BACHELOR'S THESIS

UNDERGRADUATE UNIVERSITY STUDY PROGRAM
COMPUTER AND INFORMATION SCIENCE

Mentor: prof. dr. Matjaž Branko Jurič
Comentor: prof. dr. Basel Magableh

Ljubljana, 2020

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Boris Radovič

Analiza in uporaba prohodov API v okolju Kubernetes

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: prof. dr. Matjaž Branko Jurič
Somentor: prof. dr. Basel Magableh

Ljubljana, 2020

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Faculty of Computer and Information Science issues the following thesis:

Subject of the thesis:

Analyse microservices, cloud-native architecture and APIs and address key concepts. Make a detailed analysis of Kubernetes with special focus on Kubernestes services, such as ClusterIP, NodePort, LoadBalancer, ExternalName, and CoreDNS. Analyse the role of API gateways in Kubernetes with all the relevant aspects, such as routing, authentication and authorization and high availability. Review the Ingress controller. Address the event streaming concept for microservice communication and propose functionalities for routing events. Identify the most appropriate event bridge, such as Strimzi and extend its functionalities with content base routing and rate limiting. Develop a case study where you show the benefits of your extensions.

I am very grateful to have been given the opportunity to be mentored by prof. dr. Matjaž B. Jurič and dr. Basel Magableh, so the first thought here rightly goes to them. A sincere thanks also to assist. Zvone Gazvoda for lecturing this paper and for his precious advice that helped to shape the presented code.

However, all this guidance would have been fruitless if I hadn't had someone to spur me on: thanks to Peter and Devan for showing me the way, and to my whole family for their ever-present support and their unfailing belief in me. Finally, thanks to those who stayed close when paths were driving us apart, to those who taught me and those who made me forget, to those who had patience while I was learning English, to those who didn't complain when I was thinking aloud about my non-working code. Some call them friends...

Le storie d'amore sono irrazionali e folli, ma sopportiamo tutto perché in fondo ci piace il curry.

Povzetek

Naslov: Analiza in uporaba prohodov API v okolju Kubernetes

Avtor: Boris Radovič

Zaradi uvoda vsebnikov in nekaterih pomožnih tehnologij smo danes priča pravi prelomnici kar se tiče razvoja programske opreme. Vsebniki namreč omogočajo, da se funkcionalnosti aplikacij razdelijo na sorazmerno majhne enote, tako imenovane mikrostoritve, kar nasprotuje s tradicionalnimi aplikacijami, v katerih so vse funkcionalnosti združene znotraj ene same komponente. Kljub temu, da tak pristop k razvoju programske oprame izboljša fleksibilnost, razširljivost in še druge lastnosti na strani strežnika, le-ta prinaša tudi veliko novih izzivov. Za nekatere od teh obstajajo standardizirane ali celo skupne rešitve, kot so na primer API prehodi, medtem ko za druge, rešitve enostavno ni, saj porazdeljena narava aplikacij to prepreči. Vsebniki niso nikoli pridobili veliko pozornosti z vidika hranjenja podatkov, to dejstvo pa se sčasoma spreminja, saj želijo razvijalci imeti skupen pristop za upravljanje tako podatkovnega kot tudi logičnega sloja. V tej luči je tako nastal projekt Strimzi, t.j. projekt, ki omogoča postavitev Apache Kafka gruče v okolju Kubernetes. V tem izdelku bomo razširili Strimzi in dali možnost uporabmiku, da omeji dostop do Kafka posrednikov in upravlja usmerjanje sporočil na Strimzi Bridgeju, t.j. mikrostoritvi, ki omogoča komunikacijo med Kafka proizvajalci in Kafka posredniki potom protokola HTTP.

Ključne besede: API prehod, mikrostoritve, Kubernetes, Strimzi, Apache Kafka.

Abstract

Title: Analysis and usage of API gateways in Kubernetes

Author: Boris Radovič

Containerization has marked a turning point in the way software gets developed: applications' functionalities, instead of being bundled inside of single and potentially very large code bases as is the case in monolith architectures, are split into smaller units, the microservices precisely, and even though these latter improve the backends of applications when it comes to flexibility, scalability, and some other aspects too, they bring along a whole lot of new challenges. For some of these, an API gateway might represent the solution, while others, that are introduced by the distributed nature every MSA ("microservices architecture") application inherits, just have no way out, and force developers to make do. Furthermore, containerization never gained much attention when it comes to the data layer, but this aspect, mostly because of the developers' desire of having a standard way for managing both the data and the business layer, has been changing lately. In this light and with the aim to simplify the deployment of an Apache Kafka cluster inside of Kubernetes, the Strimzi project was born, and in this paper we describe two extensions that can be applied to the Strimzi Bridge, a microservice that decouples clients and Kafka brokers on one hand, and relieves these clients from the necessity of using the Kafka binary protocol on the other. These extensions, namely rate-limiting and content-based routing, aim to solve some conundrums that are in other situations dealt with by API gateways, so before proposing them, we shall carry out a comprehensive study of API gateways, after which it shall be clear, that MSA applications have common pitfalls, and that the solutions adopted to cope with them are almost standardized as well.

Keywords: API gateway, microservices, Kubernetes, Strimzi, Apache Kafka.

List of abbreviations

Abbreviation	Meaning
API	Application Programming Interface
AWS	Amazon Web Services
BFF	Backends For Frontends
CBR	Content-Based Routing
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CNI	Container Networking Interface
CRD	Custom Resource Definition
(D)DoS	(Distributed) Denial of Service
FQDN	Fully Qualified Domain Name
GCP	Google Cloud Platform
HA	High Availability
JAR	Java Archive
JVM	Java Virtual Machine
JWT	JSON Web Token
K8s	Kubernetes
L4/L7	Layer 4 / Layer 7
MSA	MicroServices Architecture
OSFA	One Size Fits All
POJO	Plain Old Java Object
REST	Representational State Transfer
RL	Rate-Limiting
SOA	Service Oriented Architecture

Contents

Razširjen povzetek

Mikrostoritve, cloud-native, API-ji

Tradicionalni pristop do razvoja spletnih aplikacij, v katerem so vse funkcionalnosti združene znotraj ene same komponente, se je izkazal za neprimernega v sodobnih časih, ko je zahteva po hitrosti v razvoju programske opreme vedno večja. Zaradi raznih pomanjkljivosti komaj navedene monolitne arhitekture je bil sčasoma uveden alternativni pristop, t.j. arhitektura mikrostoritev (ang. "microservices architecture" ali "MSA"), v kateri so funkcionalnosti aplikacije razdeljene med sorazmerno majhne komponente (mikrostoritve). V najbolj ekstremnem primeru je vsaka mikrostoritev zadolžena za eno samo nalogo, tako da ima aplikacija lastnost, ki prejme ime Princip Ene Odgovornosti (ang. "Single Responsibility Principle").

Mikrostoritveni pristop pri razvoju aplikacij ima določene pozitivne plati, med katerimi je vredno omeniti:

1. Skalabilnost (ang. "scalability"): posamezne mikrostoritve lahko skaliramo neodvisno od ostalih; na primer v spletni trgovini, ko veliko ljudi brska po katalogu, vendar malo ljudi zares kupuje, lahko postavimo veliko instanc tiste mikrostoritve, ki prikazuje katalog, in malo instanc tiste, ki skrbi za sam nakup;

2. Hitrost razvoja: razvoj mikrostoritev opravljajo majhne in fokusirane skupine, ki uspejo imeti boljše razumevanje tematike zaradi omejenosti problemske domene, obenem zaradi njihove okrnjene velikosti uspejo tudi dosegati hitrejši razvoj in hitrejšo nameščanje storitev;

3. Fleksibilnost: vsako mikrostoritev lahko razvijemo v drugačnem pro-

gramskem jeziku;

4. Odpornost do napak: v primeru, da prejmemo primerne varnostne ukrepe, izpad ene mikrostoritve ne povzroči izpad celotne aplikacije;

Tak pristop razvoja spletnih aplikacij se je uveljavil predvsem zaradi uvoda projekta Docker, ki omogoča, da se samo kodo in knjižnice, ki so potrebne za delovanje, združi v sorazmerno majhne enote, imenovane vsebniki (ang. "container"). Drugače povedano, vsebniki nudijo alternativo tradicionalnim navideznim strojem (ang. "virtual machine"), saj za njihovo delovanje ne potrebujejo, da se na novo naloži celoten operacijski sistem, temveč uporabljajo jedro že prisotnega operacijskega sistema.

Mikrostoritve nudijo funkcionalnosti potom API-jev, do katerih se v splošnem lahko dostopa z različnimi protokoli (HTTP, AMQP, gRPC, in drugimi).

V tem poglavju bomo torej poglobljeno spoznali mikrostoritveno arhitekturo, API-je in v splošnem cloud-native pristop, t.j. pristop, ki je bil uveden zaradi vse večje migracije infrastruktur proti javnemu oblaku (ang. "public cloud"). Obenem se bomo tudi seznanili z okoljem Kubernetes, ki je današnji de-facto standard za postavitev mikrostoritvenih aplikacij na aplikacijske strežnike.

Omrežni model v okolju Kubernetes

Kubernetes zahteva, da so vsebniki med izvajanjem oviti v tako imenovane stroke (ang. "pod"). Le-ti so postavljeni v ločeni naslovni prostor, tako da ima vsak strok iste sposobnosti kot tradicionalni navidezni stroj. Z druge strani tak pristop prepreči zunanjim odjemalcem dostop do strokov. To dejstvo lahko rešimo z Kubernetesovi NodePort Storitvami (ang. "NodePort Service"), ki so zvrst Kubernetes API objektov in omogočajo, da je promet iz zunanjih odjemalcev zmožen dosegati privatni naslovni prostor, v katerem se nahajajo stroki. NodePort ni edini tip Kubernetesovih Storitev: ostali so ClusterIP, ki služi za komunikacijo med stroki iz iste gruče, LoadBalancer, ki omogoči uporabniku avtomatično postavitev izravnalnika obremenitve (ang. "load balancer") v primeru, da je aplikacija postavljena v podprtem oblaku, kot na primer AWS in GCP, in ExternalName, ki preslika storitev zunanjemu CNAME imenu.

Seveda ima vsak strok svoj IP naslov, ki je unikaten skozi celotno Kubernetes gručo. Stroki, ki želijo komunicirati z drugim strokom, pošljejo temu API zahtevo. IP naslov ustreznega stroka pridobijo potom DNS strežnika, ki je postavljen znotraj same Kubernetes gruče - Kubernetesov uradni DNS strežnik se imenuje CoreDNS.

V tem poglavju bomo vse komaj navedene tematike obširno razložili. Seznanili se bomo z nizkonivojnim pogledom kako Docker in Kubernetes uporabljata osnovne gradnike Linuxa za delovanje omrežja, obenem bomo tudi pregledali, kateri pristopi so možni zato da stroki na različnih (navideznih) strojih lahko komunicirajo med seboj.

API prehodi v okolju Kubernetes

Pozitivnim platem MSA arhitekture se pridružijo tudi določene negativne. Ena od teh je upravljanje funkcionalnosti, kot na primer avtentikacija, ki jih potrebujejo različne mikrostoritve.

Seveda je možno implementirati avtentikacijo v vsaki mikrostoritvi, ki to storitev zahteva. Tak pristop prinaša celo vrsto težav in dodatnega bremena, ki izvirajo iz dejstva, da se krši zgoraj navedeni Princip Ene Odgovornosti. Specifično, glede na to, da so lahko mikrostoritve implementirane v različnih programskih jezikih, mora biti enak proces avtentikacije implementiran v vseh mikrostoritvah, ne glede na uporabljen programski jezik. Posledično to povečuje možnost za napake v kodi, kot tudi količino potrebnega dela. Poleg tega moramo tudi poudariti, da pristop, ki se uporablja pri monolitnih aplikacijah, kjer vsak strežnik hrani podatke uporabnikov, ki so mu dokazali identiteto, ni uporaben. Vsak strežnik bi namreč moral hraniti ne samo podatke uporabnikov, ki so se avtenticirali potom njega, temveč tudi tistih, ki so tak postopek izvedli na drugemu strežniku. Seveda lahko vpleteni strežniki stalno delijo te podatke, oziroma jih hranijo v ločeni podatkovni bazi, ki je dosegljiva vsem strežnikom. V MSA arhitekturi pogosteje srečamo drugačen pristop, v katerem so sami odjemalci dolžni, da hranijo podatke o uporabniku in te podatke posredujejo vsakič, ko pošljejo API zahtevo. Te podatke, ki imajo obliko žetona in enolično identificirajo uporabnika, avtentikacijski strežnik pošlje odjemalcu, potem ko je uporabnik uspešno dokazal identiteto.

Glede na to, da so API prehodi vstopna točka v infrastrukturo, t.j. vsaka API zahteva mora preko njih, so lahko le ti zadolženi za preverjanje veljavnosti posredovanega identifikacijskega žetona in posredovanje zahteve ustrezni mikrostoritvi.

API prehodom lahko dodamo vrsto dolžnosti, med katerimi lahko omenimo:

1. Usmerjanje (ang. "routing"): API prehodi lahko prejmejo vsa sporočila in jih nato dostavijo ustrezni mikrostoritvi, kar zagotovi, da bo en sam IP naslov zadostoval za delovanje aplikacije. API prehodi lahko upoštevajo celo vrsto parametrov za določitev, katero mikrostoritev naj bi določena zahteva prejela, kot na primer HTTP metodo (npr. GET oz. POST), glave HTTP, URI naslov, in tako dalje - različni API prehodi podpirajo različne načine usmerjanja;

2. API kompozicija (ang. "API composition"): podatki, ki jih odjemalec potrebuje, so v MSA arhitekturi večkrat razpršeni v različnih mikrostoritvah. Zato da ne bi odjemalec imel odgovornosti, da si samostojno priskrbi vse potrebne podatke, lahko API prehodi prejmejo zahtevo, jo distribuirajo (t.j. sprožijo celo vrsto novih zahtev), nato zbrane podatke združijo in jih končno vrnejo odjemalcu;

3. Reševanje problema "En Tip Ustreza Vsem" (ang. "One Size Fits All"): glede na to, da lahko API-je uporablja stotine, če ne celo tisoče različnih tipov naprav in da vsak potrebuje različne podatke (na primer ko brskamo po spletu na pametnih telefonih nam je pogosto prikazanih manj podatkov kot če iste strani obiščemo potom računalnika), so lahko API prehodi zadolženi za to, da vsakemu tipu naprave posredujejo prilagojene podatke;

4. Varnostne funkcionalnosti: med te lahko upoštevamo omejevanje dostopa, validiranje podatkov, enkripcijo sporočil, in druge.

API prehodi lahko nudijo še celo vrsto drugih funkcionalnosti. V primeru, da preveč teh združimo znotraj ene same komponente, v tem primeru API prehodov, se nekako spet približamo monolitni arhitekturi.

Za konfiguracijo API prehodov v okolju Kubernetes lahko uporabljamo bodisi Ingress API objekt kot tudi lastne objekte, ki jih uvede vsak API prehod - to so primerki tako imenovanih Kubernetesovih CustomResourceDefinition objektov. Večina sodobnih API prehodov (Kong, Ambassador, in drugi) nudi obe možnosti.

V tem poglavju bomo vse te koncepte podrobno analizirali, na koncu bomo tudi spoznali, da nudijo API prehodi in posredniki (ang. "proxy") veliko skupnih funkcionalnosti. Zaradi tega večina API prehodov sloni na obstoječih posrednikih, katerim lahko doda določene funkcionalnosti potom vtičnikov. Obenem je tudi postavitev in upravljanje API prehodov lažja v primerjavi s posredniki - na primer, nekatere API prehode je mogoče konfigurirati potom REST zahtev.

Apache Kafka

Apache Kafko lahko smatramo kot alternativo tradicionalnim podatkovnim bazam. Le-te namreč hranijo samo trenutno stanje, ki je v splošnem rezultat vseh dogodkov, ki so spremenili podatkovno bazo. V primeru, da imamo dostop do baze, nismo zmožni rekonstruirati dogodkov, ki so privedli do trenutnega stanja baze. Drži obratno: v primeru, da imamo na razpolago vse dogodke, jih lahko združimo in tako pridobimo stanje korespondenčne podatkovne baze.

Apache Kafka hrani trajni dnevnik dogodkov. Zaradi tega so hranjeni dogodki nespremenljivi, obenem je tudi njihov vrstni red zagotovljen, t.j. Kafka hrani dogodke v istem vrstnem redu kot so bili vstavljeni.

V tem poglavju bomo natančno analizirali Apache Kafko, njegovo arhitekturo tako na strani odjemalca kot tudi na strani posrednika (ang. "broker"), obenem bomo tudi spoznali pojem obdelave toka podatkov (ang. "stream processing") in pregledali, kako ga Apache Kafka podpira.

Dodatek funkcionalnostim Strimzi Bridgeja

Strimzi je odprtokodni projekt, ki omogoči postavitev Apache Kafka gruče v okolju Kubernetes. Sestavljen je iz več komponent, saj se tistim komponentam, ki so potrebne za običajno postavitev Kafke, dodajo še dodatne: to so Bridge, ki je mikrostoritev, ki dovoli komunikacijo med odjemalci in Kafka posredniki potom protokola HTTP, in razni Kubernetes operaterji (ang. "operator"), ki olajšajo postavitev in upravljanje Kafke v okolju Kubernetes.

V tem poglavju bomo Strimzi Bridgeju dodali dve funkcionalnosti:

1. Vsebinsko usmerjanje sporočil (ang. "content-based routing"): Bridgeju dodamo možnost, da samostojno določi temo (ang. "topic") prejetemu sporočilu. Trenutna verzija Bridgeja namreč zahteva, da je tema, kateri je sporočilo namenjeno, posredovana skupaj s samo vsebino sporočila. S funkcionalnostjo, ki smo jo dodali, lahko Bridge samostojno določi temo bodisi s tem, da upošteva samo glave HTTP zahteve, kot tudi poljubne parametre zahteve - v slednjem primeru mora razvijalec priskrbeti Java razrede, ki skrbijo za usmerjanje;

2. Omejevanje dostopa (ang. "rate limiting"): Bridgeju dodamo možnost, da omeji število pisalnih zahtevkov, ki jih lahko proži določeni odjemalec. Na primer, lahko določimo, da uporabniki, ki plačujejo za določene storitve, pošljejo največ x pisalnih sporočil določeni temi na minuto, medtem ko uporabniki, ki se poslužujejo storitev zastonj, v istem času pošljejo največ y sporočil.

Ustrezno bomo razširili krmilnik (ang. "controller"), da bo postal zmožen aplicirati ustrezno konfiguracijo Bridgeovim strokom, hkrati bo postal zmožen spremljati konfiguracije in v primeru sprememb, obstoječe stroke odstranil in jih nadomestil z novimi.

Primer uporabe: logiranje sporočil iz spletnih brskalnikov

V zadnjem poglavju bomo predlagali enostavni primer uporabe funkcionalnosti, ki so bile razvite v šestem poglavju. Razvili bomo spletno aplikacijo sestavljeno iz več mikrostoritev, ki omogoči uporabniku, da pošlje sporočila Kafka posrednikom, ki so nameščeni v Kubernetes gruči. Obenem bomo tudi predlagali enostavno avtentikacijo, za katero bo skrbel API prehod (Kong).

Spletno aplikacijo bomo postavili na GKE, t.j. upravljani Kubernetes storitvi.

Chapter 1

Introduction

1.1 Motivations

We are living in rather exciting times from the perspective of software development: the domineering entry of different cloud vendors into the market has caused - and still is - different companies, attracted by benefits such as trading capital expenses for variable expenses, eliminating the guessing capacity needs, the advantages of massive economies of scale, and others [98], to abandon their own on-premise infrastructure in favor of a hybrid or even all-in-cloud deployment model. At the same time, the just-listed benefits make the cloud the perfect solution for startups seeking some way to enter the ruthless software market without making unbearable initial investments - just consider Dropbox, a startup that in the origin focused on providing a more user-friendly way to store data in AWS S3 buckets, and just recently moved to a proprietary infrastructure [15]. On the other hand, the concept of the "cloud" goes much beyond that, and the CNCF company fosters all the cross-cutting approaches that were introduced along-with (and thanks to) this ever-growing paradigm: containerization, serverless computing, agile software development methodologies, and an almost blindfold reliance on open-source projects, are just some of these new technologies and approaches that are revolutionizing the way software is being developed and deployed.

In this paper we are mostly going to deal with containerized applications, i.e. those applications that are deployed by means of lightweight and portable units of code, the containers, precisely. Though the business layer has seen an almost unanimous acceptance of this new technology, developers have been, and largely still are, hesitant when it comes to adopting containers also for managing the data layer. This fact is rapidly changing in recent years, hence our desire to contribute to this revolution by extending and improving the features of Strimzi, a project that allows to deploy an Apache Kafka cluster inside of Kubernetes.

1.2 Goals

The main goal of this paper is to extend the capabilities of the Strimzi Bridge, which is a component that provides a RESTful interface that makes it possible for clients to communicate with the Kafka brokers without the need to use the Kafka binary protocol. Nevertheless, we will not fail to demystify some of the implementational details about Kubernetes, such as networking, the most common API objects, and so on, and thoroughly analyze the position that might be taken up by API gateways in a highly dynamic and rapidly evolving environment such as Kubernetes.

1.3 Structure of the paper

After a brief introduction into the cloud native revolution in chapter 2, we shall move to analyze in great detail the networking aspect of Kubernetes in chapter 3 and later on discuss all the possible challenges of implementing an application built using the MicroServices Architecture (hereinafter referred "MSA") and how can an API gateway help us with coping with such challenges - chapter 4. Before moving to the more practical part of the paper, in which we extend the capabilities of the Strimzi Bridge with some of the concepts introduced in the previous chapters - chapter 6 - and present a purely demonstrative use-case - chapter 7 -, we shall make in chapter 5 a brief discussion about Apache Kafka's benefits and limitations. This discussion, although not essential in order to understand the final two chapters, aims to clarify what Apache Kafka is, and why (and when) should we prefer this type of data storage to the traditional and more familiar databases.

Chapter 2

Microservices, cloud native and APIs

In this chapter, we will take a closer look at how is the microservices architecture changing the way software is being developed.

2.1 Key concepts of micro-services

In the past 10 years, software development has been going through some major changes, since the classical approach ("monolith applications") is gradually being replaced by microservices. Pioneers in this area are Netflix [12] and Amazon [11], that sooner than other software companies realized, that monolith applications cannot fit in a world where speed in software development may well be the key factor that distinguishes between a successful company and a company, whose future is doomed.

In particular, the classic, monolith applications consist of a large codebase in which all the functionalities of the app are piled, a property that directly causes the code base to get larger and larger as new functionalities are added to the app, and also results in difficult deployment, testing, scaling, and troubleshooting processes [32]. The apps, built with this architecture, suffer other limitations as well: for example, they behave poorly as the developer team increases in size and are not well suited for exposing business functionalities through an API [35]. Therefore, as a countermeasure, a new technique, in which software is decomposed into services that communicate over some Enterprise Serial Bus, was developed and took the name Service Oriented Architecture. Unfortunately, by nature, SOA applications could still be monolithic so this approach did not solve most of the problems we just listed. Eventually, microservices were introduced to solve these shortcomings and today, the just-mentioned and the SOA architectures coexist, given the fact that both these architectural patterns have their own positive aspects [5].

The Microservices Architecture (MSA) is a software development technique in which the functionalities of an application are split into services that are developed, deployed, and run independently. The aim of this approach is to develop software components that are as decoupled and as cohesive as possible [8], and at the same time let the components be independent of the underlying infrastructure: they might run on the same machine, run on different machines in the same LAN or on completely different networks. Given that microservices are usually not stand-alone entities, some communication mechanism has to be implemented - the development of this mechanism is arguably the most challenging part in the development of an MSA application [32]. Even if this communication can be achieved with several protocols (HTTP, AMQP, GPC, TCP, and others), the "smart endpoints and dumb pipes" approach is the preferred one [109]. No matter the protocol, the components expose their functionalities via an Application Programming Interface (API).

The main benefits we can point to the microservices architecture are the following [86, 40]:

- Scalability: a single component can be scaled in or out according to the demand on the component itself, as opposed to the monolithic approach, in which a whole instance of the application has to be deployed. For example, in an online shop developed with the MSA architecture, when a lot of people are browsing the catalogue but very few are actually buying goods, we can run multiple copies of the "catalogue" microservice and very few copies of the "billing" one;
- Speed of development: the development of a microservice is done by

small and focused groups, that thanks to their limited size can reach higher productivity. Furthermore, the testing of a single component can be automated, which allows companies to make use of CI/CD (continuous integration/continuous delivery) to deploy new software;

- Flexibility: each microservice can be implemented in a different programming language and can be completely modified, as long as the API through which it exposes its functionalities remains the same;

- Fault tolerance: the failure of a microservice does not cause the failure of the whole application, though a non-optimal design might undermine this property. Of course, when an instance of a microservice sends a request to another microservice, it cannot be sure that the request will be followed by a response. We can make here a distinction between immediate errors, which are good for they allow to immediately react to the disruption, e.g. by sending the same request to another instance of the requested microservice or by retrying to send the request after some time has elapsed, and timeout errors. The latter errors are considered unpleasant for there is no way to know whether they are happening because messages get lost during transmission, because the requested microservice is handling a lot of traffic so it just needs some more time to respond, or because of some other reason [58]. Given that we cannot know the cause why is such a disruption happening, we cannot know what is the best way to react to it either; in order to cope with such a conundrum, health checks and the circuit breaker pattern are often used - we discuss these topics in more detail in subsection 4.1.4.

It is said that nothing has only positive sides and the MSA architecture makes no exception. Indeed, this architectural style brings along a number of challenges and disadvantages which we will extensively analyze in section 4.1. Regardless of these shortcomings, microservices have increased in popularity in the last years, especially since 2013, when Docker was released as open-source [74].

Docker is a tool that allows us to containerize applications. This means that software is deployed in containers, that are by nature lightweight, inde-

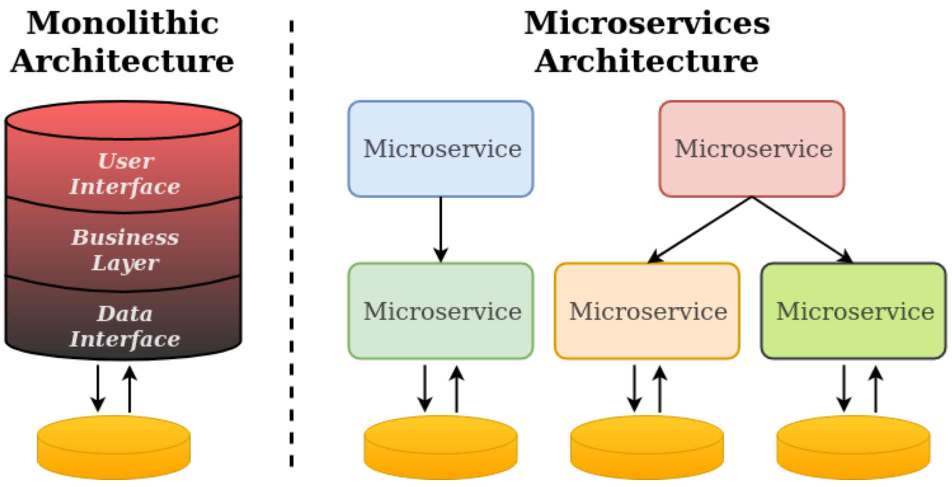

Figure 2.1: Monolith architecture vs MSA.

pendent of the hosting environment and efficient in system resource utilization. In fact, the concept of a container was not introduced by Docker [68], but it was this project that made working with containers neat and easy to the extent that today, even if microservices can be developed without Docker, this way of software development is by far the most applied.

2.2 Key concepts of APIs

An Application Programming Interface, commonly referred to as API, is an interface that establishes the ground rules that allow two software components to communicate. In the origin, an API was understood to be the interface by means of which an operating system provided functionalities to an application program, or alternatively a program put at disposal some functionalities to another, external program - this kind of APIs are still present and are usually available out-of-the-box when the software that provides the API is installed. The Java API and the Linux kernel API are just two of the many APIs of this kind that we use in our everyday life. On the other hand, with the advent of the Web, this concept has been extended since a new type of APIs was introduced, and from here on out we will shortly analyze the latter, Web APIs.

Web APIs are interfaces that allow two software components to interact over the internet. An optimal design of these interfaces is a crucial aspect in the design of a system, since a well-formed API allows the applications that use it to be independent, and it allows an optimal use of the underlying network as well [7].

The first Web APIs were of course web pages: clients, that at the dawn of the internet were only web browsers, made a request to a web server, which in turn responded with some HTML content that the browser displayed on the screen. We can see that the request is in such a case synchronous since web browsers can do nothing but to wait for the response of the server. Later on, in the early 2000 with the gradual integration of JavaScript, AJAX, and the notion of displaying some content to the user without the need of reloading the page, a new type of communication was introduced: a client invokes the API by sending a message to a server, which eventually responds with the requested data, usually either in a JSON or XML format - examples are the Google Maps API, the Twitter API and the Github API. The peculiarity of the just exposed process is in that the client does not necessarily need to wait for the response of the server, nor reload the page when the response is received. Either way, in order for this communication to happen, any type of protocol can be used, although the most common protocol for client to server communication is HTTP and its extension REST, which uses HTTP with some additional constraints for which the developer is responsible.

In the context of MSA, the API has the meaning we might expect: it is the interface that allows two microservices to communicate and exchange data. The communication itself can happen over a variety of ways, such as HTTP, messaging queues and event-driven communication [72], the latter two having the advantage that they allow and indeed promote decoupling between single microservices.

Whereas the provision of APIs has lately become a market per-se, with external connectivity managed by an API gateway with throttling, security, and other functions [103], the communication between microservices cannot be implemented with this kind of point-to-point (P2P) approach for this would lead to complications in terms of maintainability, operability, and deployment timings [25]. Better approaches are in use today, such as the End-to-End (E2E) technique, whose logic resembles the SOA one, and the API-led connectivity. The latter blueprint is an exponent of the layered architectural pattern, given the fact that it requires the decomposition of APIs into three layers, from the bottom up System, Process, and Experience layer, each layer providing services to the layer just above it. Benefits of the API-led connectivity are component reutilization, speed of development, and lower chances for bugs [25].

2.3 Key concepts of cloud native architecture

The shift we are witnessing in the last years from an on-premise to a hybrid or even cloud-based infrastructure results not only in a different way of managing hardware resources, but in a different approach to software development as well. The formal definition of "cloud native", provided by CNCF [108], describes it as a technique that empowers organizations to build scalable, resilient, manageable and loosely coupled systems, and among the approaches, that the cloud native paradigm expects us to use, one of the most important ones is the utilization of cloud native services, i.e. those functionalities, that are provided out-of-the-box by public cloud providers. Other (software and organizational) approaches that cloud native apps use are containerization, microservices, DevOps, i.e. a tight collaboration between the development and the operations teams, and, since recently, server-less computing [137], such as AWS Lambda and Google Cloud Functions.

We might say that this software shift has been driven, apart from the new technologies available, by a cultural shift as well: following the 2008 great recession, companies struggled to reduce their IT costs without renouncing to the innovation pace they used to have. The only way this target could have been achieved is by companies embracing the open-source philosophy and in fact, in those years an exponential growth of open-source solutions happened. This kind of solutions were adopted even by IT giants such as Google, that contributed firsthand to this shift with two major achievements: in 2015, Google released as open-source Kubernetes and launched, in association with the Linux foundation and other companies, the Cloud Native Computing

Foundation we mentioned above [77]. The idea behind this foundation is to create a neutral base that allows a widespread acceptance of the cloud native approach and fosters at the same time open-source projects. Among the projects that were developed under the wing of the CNCF foundation, let us here mention Kubernetes, Envoy proxy, CoreDNS, Prometheus and others. Today this foundation counts more than 400 members, among which we can find companies like Amazon, Google, Microsoft, Oracle and Alibaba, presences that we might interpret as a clear sign about how has the open-source technology been almost unanimously accepted.

One of the main pillars of the cloud native approach is Continuous Integration [94], which is the process of automating the software building process, and testing the product every time a developer makes changes to the code base. For this purpose, some framework such as Robot [64] might be used in order to automate the acceptance testing. Other benefits that cloud native apps have are the increased agility, rapid innovation, statelessness and others [137, 94].

Finally, let us point out that even though the cloud native technologies were developed in response to the migration towards the public cloud, they might be embraced in a private cloud as well.

2.4 Key concepts of Kubernetes

Kubernetes (sometimes abbreviated K8s) is an orchestration tool for managing containers and it is so popular nowadays, that we might consider it the de-facto standard for deploying containerized applications. As we briefly discussed in section 2.1, software is usually developed by means of Docker containers that, as a downside, have no built-in functionalities as for managing the pitfalls, that may happen while the containers are running - for example, if a container fails, it will not be restarted on its own. Furthermore, the Docker CLI is not provided with any tools that allow to automate the scaling process of the infrastructure, tools that are on the other hand present in the Kubernetes CLI. Let us here introduce the key components of the K8s infrastructure that we will be using hereinafter [122]:

- Pod: is a single instance of an application that wraps one or more containers (Docker, rkt or other) and gives them shared network stack and shared storage; usually, there is a one-to-one relationship between Pods and containers (even though, as a matter of fact, Kubernetes adds some control containers to each Pod [34]);
- ReplicaSet: API object consisting of a set of Pod replicas. It is managed by a ReplicationController, whose purpose is to maintain a certain number of Pods running at a given time;
- Deployment: is an abstraction over a ReplicaSet that, in contrast to the latter, allows a more flexible management of the Pods and a smoother transition from an old version of an application to a new one - this is the so-called rollout strategy;
- Service: is an abstraction of a set of Pods that run the same microservice. When a request is made to a Service, the Service proxies the request to one of the Pods it represents;
- Node: is a virtual or physical machine on which Pods are deployed. Sometimes referred to as "worker machine" or "minion", it accommodates a container runtime such as Docker, along with some Kubernetes specific software, namely kubelet, which is a piece of software that makes sure that the Pods are running as expected, and kube-proxy, that is responsible for the networking aspect of the node;
- Control plane: consists of one or more master nodes, on which software (such as scheduler, API server, controllers and others) for managing the correct operation of the cluster are deployed;
- Cluster: set of worker machines grouped with a set of master nodes. At the very least it consists of a worker node and its associated control plane as is the case with Minikube [126], though in a production environment there is the need to run multiple nodes in order to assure high availability ("HA").

Kubernetes API objects are defined via a YAML definition file. The request for creating such objects is issued to the API server residing on the master node. The latter, upon receiving such request, sends a message to the kubelet of some worker node, instructing it to create the desired objects.

In this paper, we will use capital letters to indicate Kubernetes objects. This is done to emphasize, for example, the differences between Kubernetes Services and services, intended as the contracted form of microservices.

Chapter 3

Kubernetes networking and routing

In this chapter, we will dive into how does Kubernetes handle network traffic and we will see how does it manage DNS through CoreDNS. This discussion will introduce a number of concepts that shall become of vital importance in chapter 3 and chapter 7, since it will clarify the constraints API gateways, and in general all microservices, are subjected to because of the networking model employed by Kubernetes. At the same time, we shall present different approaches about how to expose an application's functionalities to the Internet - one of these shall be used in chapter 7 to deploy an application on GKE.

3.1 Pod network

Kubernetes and Docker networking highly rely on the functionalities provided by the kernel of the operating system of the machine on which they run, so before diving into the networking aspect of Kubernetes, let us here introduce some key concepts - for simplicity, we limit ourselves to the Linux kernel. The Linux kernel has many networking building blocks, such as:

- Bridge: virtual switch. As such, it has the role of forwarding packets to the intended receiver by performing packet switching. It executes such an operation after inspecting the incoming frame, in the header of which the IP addresses of the sender and the receiver are imprinted, and after consulting its own table of known addresses, to which it adds records dynamically. In other words, whenever a switch handles a frame, it adds the IP addresses of the receiver and the sender, along with the respective MAC addresses, to the just-mentioned table. As we shall see shortly, *docker0*, which is one of the main components of the Docker networking model, is an instance of a Linux bridge.

- Network namespaces "netns": isolated network stack.
- Virtual Ethernet Device: usually referred to as *veth*, it is an interface that connects two namespaces.
- IPtables: is a firewall system integrated into the Linux kernel. The IPtables are composed of rules, that are organized into five tables, namely *raw*, *mangle*, *NAT*, *filter* and *security*.

The just exposed concepts are highly used by Docker and Kubernetes when setting up the network, to which the containers and the Pods are attached. Given that Kubernetes extends the networking model provided by Docker, let us first rapidly review how do Docker networks function.

If not otherwise specified, when a Docker container is launched, it is attached to the bridge network. This is an internal private network accessible only from within the network itself, with CIDR addresses usually in the range 172.17.0.0/16 [119]. It is composed by the *docker0* bridge and by the containers attached to it, each of which is created with a virtual ethernet device called *vethX* for some small number *X*. By creating the interface this way, i.e. with one end placed inside of the container and the other one connected to the Docker network, the container has the same capabilities of an actual virtual machine, and considering that all the containers are in the same network, they can all communicate with each other, at least as long as they are able to find out the IP address of the container they want to communicate with.

On the other hand, communicating with the outer world is not so straightforward, since a device, that is external to the Docker network, cannot issue commands directly to the IP address of the containers. In order to perform such an action, port publishing, a process that maps a port on the host machine to a port on the container, needs to be performed and is indeed performed by the Docker daemon by adding new rules to the IPtables of the machine, on which the daemon runs - a similar process happens when using other container runtimes, such as Apache Mesos Containers [128], which use the `mesos-cni-port-mapper` plugin for this type of configuration, and LXD Linux Containers [21], which require the user to manually configure the IPtables of the host machines or to leverage the capabilities of the LXD Proxy Device [81]. Given that the Docker network is, even when the container's ports are published, still completely isolated, inbound and outbound traffic use Source-NAT and Destination-NAT in order to forward packets inside and outside of the internal Docker network. Furthermore, for each container Docker launches, a *veth* pair that enables tunneling from the host namespace to the bridge namespace gets created [51].

Here is a quick example to show how do the just exposed concepts fit together. In a machine with Docker installed, we can issue the command:

```
docker run -p 8080:80 nginx
```

This command creates a container using the *nginx* image and makes the container accessible through port 8080 on the host machine. The Docker daemon needs therefore to update the NAT table of the IPtables so that whenever a packet is received on port 8080 by the host machine, a NAT operation is performed and the packet is forwarded to the *nginx* container. The actual NAT translation is performed by *netfilter* [10], which is a framework that runs in kernel space and whose operations are triggered whenever a message is received.

Docker supports other network types as well, such as the Custom Defined Bridges and the host network, in which the containers share the IP address of the host machine. The latter strategy should be used carefully though since in such a case, any kind of isolation of the containers is removed.

Taking a step further, the capabilities of Docker are limited to a single host machine. If we wish to run an application on multiple machines and do not want to configure all the networking aspects manually, some external tool such as Docker Swarm or Kubernetes is needed.

As we saw in chapter 2, the basic unit of an application in Kubernetes is the Pod, which is a set of containers that share network and storage capabilities. Even though, as we shall see shortly, communication between Pods can be achieved in different ways, Kubernetes imposes some ground rules regarding how this can happen [107]:

- as opposed to Docker, Pods must communicate without using NAT, even if they are on different machines;

- agents on a node, such as kubelet, must be able to communicate with all the Pods in the machine on which they are running;

- the IP a Pod sees as its own is the same as the IP the other Pods see.

Given the just stated conditions, we can quickly infer that the IP address of each Pod has to be unique across the whole cluster, and Kubernetes accomplishes that by assigning an overall address space to each node and later on assigning to each Pod an address within that range [37]. The IP addresses of all the Pods in the cluster are stored in the ETCD storage.

When a Pod is created, the pause container is added to it. This is done because of two main reasons: firstly, the pause container holds the network namespace for the Pod in the case that a container is shut down, and secondly, because it serves as the parent process of any other process, i.e. has a similar role as the init process in unix [34]. Luckily enough, as a user, we do not need to worry about all these details since they are all taken care of by Kubernetes.

Kubernetes configures other networking details, such as assigning to each Pod a virtual interface *eth0*, creating a custom bridge that substitutes the *docker0* bridge, and creating a *veth* pair between the Pod's *eth0* and the custom bridge [102].

The intra-node communication is now pretty simple: a Pod that wishes to send a packet to another Pod on the same host sends the packet to the default gateway, i.e. to *cbr0* via its own *eth0*. The custom bridge then performs an ARP request to find out the MAC address of the destination Pod, and since this Pod resides on that particular machine, it responds to the request. The *cbr0* ultimately adds the MAC addresses of both the Pods that participated in the communication to its own ARP table, and the packet is finally forwarded to the intended Pod. A simple visualization of the whole process is shown in Figure 3.1.

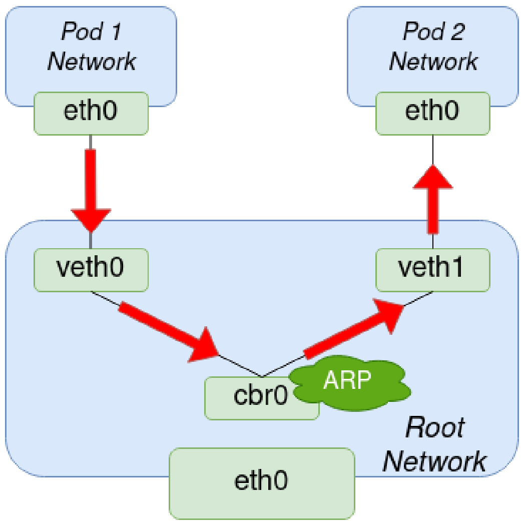

Figure 3.1: Intra-node communication.

On the other hand, the inter-node communication, i.e. the communication between Pods residing on different nodes, is more complex: in a similar situation as the one just exposed, no Pod would answer to the ARP request of the *cbr0* bridge for the destination Pod is not on the same machine, so the packet would be forwarded to the *eth0* interface of the host device. Therefore there is the need to provide a way for the packet to reach the intended Pod on the destination machine.

Different approaches exist in order to solve these difficulties and Kubernetes allows us to pick up any vendor's networking technology we prefer. So, to rectify what we stated previously, Kubernetes is not directly commissioned with the network connectivity because such a task is in fact delegated to special programs called Container Networking Interface (CNI) plugins. There are several CNI plugins among which we can freely choose the one that best suits our needs.

In general, we can group the strategies for inter-node communication into four groups [54]:

1. Layer 2 networking: this approach simulates a layer two connectivity

between Pods even when they are on different hosts. Pods can communicate this way through means of L2 switching and ARP requests only [31], but even though this solution is extremely simple, it is never used in production since even in the Kubernetes documentation it is stated that this solution "seems to work, but has not been thoroughly tested" [107];

2. Layer 3 networking: uses L3 routing to route the traffic to the intended Pod. Two different approaches can be taken in order to achieve that: the routing rules can be added to the default gateway of the cluster, in which case a packet is redirected to the intended node by the latter, or alternatively each node of the cluster can be populated with routes, so that there is no need to reach out for the default gateway. Either way, once the packet reaches the intended machine, the same mechanisms we explained for Docker are used for transmitting the packet from the machine's netns to the netns of the Pod.

   Calico, one of the most popular CNI plugins, known for the performance level it offers [55], belongs to this category. This CNI assigns to each Pod a public IP and configures on each node a vRouter, which makes the node act like an actual router. It uses BGP for routing and runs some daemons on the machine [63] in order to keep track of the changes in the network.

   A difficulty we might encounter using this approach is that some networks have anti-spoofing mechanisms that cause packets with a forged (in this case rightfully) IP address to be automatically discarded. Techniques to avoid this distress vary across different platforms; just to make an example, in GCE the virtual machines need to be configured as routers [23].

3. Overlay network: this approach provides an isolated L2 network that spans over the Pods spread across the cluster. The L2 connectivity between Pods is given by the fact that the packets traversing the underlay network are encapsulated in another packet and are decapsulated when they reach the destination machine.

The most popular CNI that implements this technology is Flannel. Flannel creates an additional Linux interface "flanneld", that is responsible for wrapping the messages it receives from *cbr0* into UDP packets, and configures a daemon ("flannel0") to listen to the Kubernetes API server in order to keep track of all the Pods running in the cluster [24].

Drawbacks of this approach are complexity overhead, which is caused by the encapsulation-decapsulation process to which the packets are subjected to, and an increased latency between nodes.

4. Cloud: cloud providers usually have their own Kubernetes service, e.g. AWS offers to their customers a fully managed service called EKS. These cloud providers use a combination of the strategies outlined in the preceding points.

As a user, we do not necessarily need to be aware of all of that, but it might become very useful when debugging a faulty application.

3.2 Kubernetes Services

Until now we discussed how can Pods communicate with each other, but by doing that, we assumed that they always knew the IP address of the Pod they wanted to communicate with. However, it transpires that Pods are ephemeral, which means that it happens quite often that they fail and have to be substituted by another instance of the same Pod. When this happens, i.e. when a Pod is destroyed and replaced with a new one, it is not guaranteed that the Pod that gets created in order to bridge the arisen gap will have the same IP address as the destroyed Pod, and in this kind of condition it is highly unpractical that every single Pod keeps track of all the changes happening in the cluster, hence the need for Kubernetes Services. A Service is an API object that, from a logical point of view, is placed in front of a set of endpoints and dispatches traffic across these endpoints. In other words, when a Pod (or an external client, as we shall see shortly), wants to access to an application, it makes a request to the Service associated to that application, which in turn forwards the request to one of the endpoints it represents. The tasks a Service needs to perform are very similar to the ones of a reverse proxy, so the characteristics we demand from it are the following [38]:

- it must be durable in the sense that its IP address does not change;
- it must have a list of endpoints it can forward requests to; since this list changes over time, it needs to keep updating it;
- it must have some way of knowing whether a particular Pod is healthy or not.

Before getting to know the Service types, let us quickly take a look at how does the ClusterIP, a Service designed for communication between Pods, work.

When launching a ClusterIP Service, we can immediately observe that the IP address assigned to it belongs to neither the Pod network nor the network the nodes are on - for example, on Figure 3.2, the address of the Service is *10.3.241.162*. It belongs to a network called service network, but in fact, there is no virtual or physical interface associated with it. Again, this IP address serves only to configure the IPtables of the nodes in the cluster in such a way, that the CNI plugin will be able to make the packets it handles reach their final destination. Let us see how this happens: when a packet reaches the interface of the node, the netfilter's functions are triggered and this causes the receiver's IP address and possibly port as well to be substituted with the ones of the actual destination Pod or machine. Therefore it is required that the IPtables in the kernel always have an updated list of Pods that correspond to a Service. This task is performed by kube-proxy, that as we discussed already, is a daemon that resides in each node of the cluster and communicates with the API server. As soon as a change happens in the cluster, kube-proxy is notified and as a result, the IPtables are updated.

The Services can run in either user-space proxy mode or in IPtables proxy mode - in fact, there is also a third mode we will shortly introduce. In the IPtables proxy mode, all the traffic is handled by the Linux kernel, so that the packets leaving the node already have the IP of the Pod they are intended

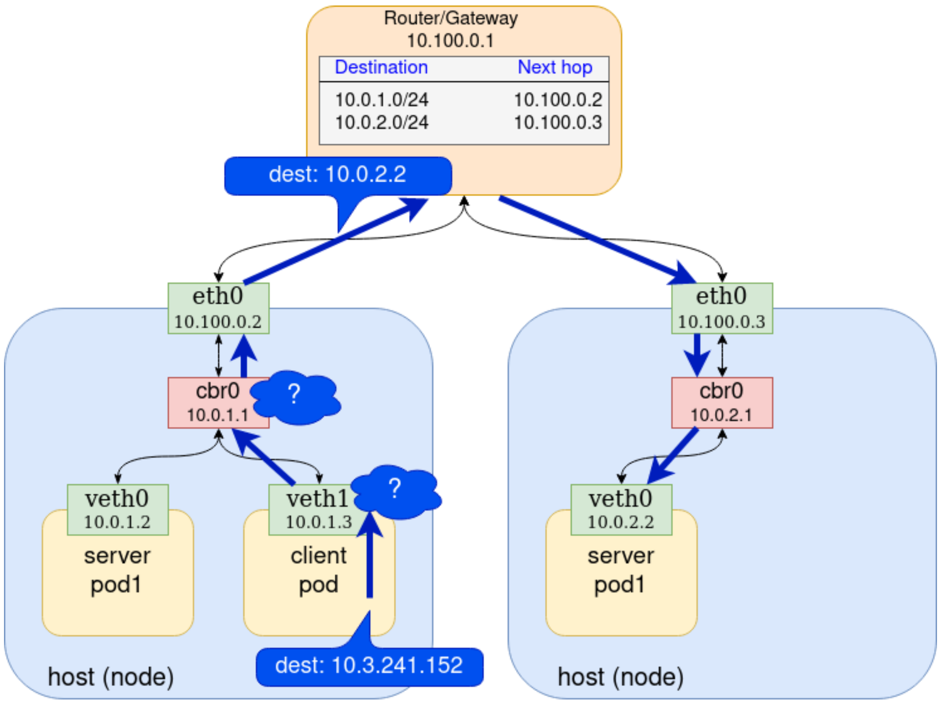

Figure 3.2: Example of ClusterIP Service.

to reach, rather than the one of the corresponding worker machine. On the other hand, in user-space mode, the outgoing packets have the IP address of the node in the cluster, on which the Pod the packet is meant to reach is running. That node, upon receiving the packet, passes it to kube-proxy, which has this way the additional task to forward the packet to some internal Pod that is running the required microservice. The user-space mode has worse performances, since the translation performed by kube-proxy happens in user space, and is considered less reliable than IPtables mode [133].

We can see now that even if a Service can be represented as a single object and is stored as a single object in the ETCD key-value store, it is in fact a distributed system that, as most of the other networking components of Kubernetes, highly relies on the features provided by the operating system of the nodes forming the cluster.

There are four kinds of Services, below we are going to take a look at each one of them.

3.2.1 ClusterIP

ClusterIP is the most basic Service type and is the Service that is created by default. It serves for communication between Pods, i.e. a Pod requesting services to an endpoint makes a request to the ClusterIP Service sitting in front of such endpoint. The configuration of which endpoints to forward traffic to is done in the YAML file that spawns the Service, namely in the `.spec.selector` field, and even if the Service is able to resolve the IP addresses on its own as discussed in section 3.3, there is still the need to configure the layer 4 networking aspect, i.e. we need to specify on which port does the Service accept requests on and to which port should the traffic be directed to.

By default, this kind of Service does not perform any kind of load balancing and distributes the incoming traffic with a round-robin or random algorithm, depending on whether it runs in user space or IPtables space.

3.2.2 NodePort

All the techniques discussed so far deal with communication between Pods. On the other hand, achieving that is pointless if we cannot expose the application to the outer world, and the NodePort Service deals precisely with that.

It is clear now, that in order to access an application or microservice deployed on K8s, we cannot use the bare endpoint's IP, since this IP is private. In order to solve this problem, NodePort Services were introduced. These Services listen to traffic on the each node of the cluster and forward the traffic to the Pod network.

In the YAML definition file used to create the NodePort Service, apart from selecting to which endpoints should the traffic be directed to, ports need to be configured as well: namely, the `nodePort` and the `port` parameters are used to specify which ports does the Service expose, and differ between them by the fact that whereas the `port` is reachable only from within the cluster, the `nodePort` is intended for external access to the app. Finally, the `targetPort` is the port inside the cluster to which traffic is directed to.

The NodePort Services have a number of downsides though, including the fact that the `nodePort` value can only be in the range 30000-32767, it is possible to have only one Service per host-port pair, NodePort Services might cause some problems if the IP of the node or virtual machine changes [56], they might cause a gaping hole in the security policies of the cluster for they allow to circumvent most network security policies set up in Kubernetes [36], each machine, on which a NodePort Service is running, needs to have a publicly accessible IP address and finally, the clients themselves are required to load balance requests among the available servers.

On the other hand, by using this kind of Service, the user can set up a custom load balancing policy by placing a user-defined load balancer in front of the NodePort Services, but this solution requires some effort to be set up and later on maintained.

3.2.3 LoadBalancer

An alternative to the rather primitive NodePort Service is the LoadBalancer Service, which is a Service that, when deployed in a supported cloud environment (e.g. AWS, GCP, Microsoft Azure and others), causes the Kubernetes Service controller to automatically provision a Load Balancer without the need for the user to invoke the Cloud Service Provider's API separately. For example, when we create a Service of type LoadBalancer in EKS (Kubernetes service provisioned by AWS), an instance of an Elastic Load Balancer (ELB) is automatically created and is thereafter responsible for load balancing across the worker nodes comprising the Kubernetes cluster. The configuration of the AWS ELB happens in the YAML file in which the Service is defined, namely in the annotation section: in those fields, the user can specify if he wishes to launch a Network or Classic ELB (it is Classic by default [125]), can make the ELB publicly accessible or private, and can make other configuration as well [127]. In general, the user has no control over how is traffic balanced: for example, the AWS ELB might use Round Robin algorithm, the least outstanding request routing algorithm, or a flow hash algorithm [118]. Either way, the external load balancer performs L4 traffic distribution.

The load balancer provisioned by the Cloud Service Provider has its own

IP address, so requests to access the app from the outer world have to be issued to the IP address of the Load Balancer. This Service is therefore a single point of failure [13], since if the balancer for any reason fails, it will not be possible to access to the services provided by the app. On the other hand, the cloud provider usually sets up a high availability policy by creating a failover instance in some other availability zone and making it ready to take on the traffic, so it is unlikely that the failure of the load balancer will result in a downtime.

3.2.4 ExternalName

When accessing to a ClusterIP Service from within the cluster, a DNS discovery mechanism is provided by Kubernetes, so that we can send traffic knowing the Service name, instead of the Service IP.

ExternalName is a Service that in contrast to that, allows the user to specify a DNS name to which the Service is mapped.

Unfortunately, the unavoidable fact is that the just exposed Services, based on IPtables, are the bottleneck that prevents us from creating large clusters with a very large number of Pods and Services [82]. Suppose for example a cluster with n Services, each one backed by m Pods: each node in the cluster would in this case have $n \cdot m$ entries in its IPtables, fact that becomes unpractical when both these parameters increase. As from Kubernetes v1.11, a new mode, called IPVS proxy mode, was introduced in order to face these problems. IPVS stands for IP Virtual Server and is just like IPtables a service incorporated into the Linux kernel, but has some peculiar functions that make it well suited for load balancing. In order to reach better performances when compared to IPtables, this service uses hash tables as the underlying data structure, and this results also in a better throughput [133]. When launching an IPVS Service we can specify the routing algorithm, which can be round-robin, least connection, shortest expected delay or some other. Standard, IPtables based Services are still default though, because they offer services such as packet filtering, SNAT and others, that IPVS Services do not [82].

3.3 CoreDNS

The last question we are going to answer in this chapter is how does a Pod find out the IP address of a Service. Of course, we could hardcode such IP address in the YAML file that spawns the Service in the `.spec.clusterIP` field and later on make the Pods use this address, but this is usually considered a bad practice. There are in fact two ways in which a Pod can find out the IP address of the Service it wants to communicate with.

The first is by using environment variables, that are stored inside the Pod itself and that can be accessed using the `printenv` command in Linux, which returns the IP, the port and other information about the Services. This approach has a major downside: since the environment variables are assigned to the Pod when it is created, information about only the Services created before the Pod are at disposal to the newly created Pod. We can see that by executing the following command:

```
kubectl exec <pod_name> -- printenv
```

After issuing such a command we get all the environment variables of the Pod, among which there is a pair of variables for each Service that was running when the Pod was launched. Such a pair contains the IP address as well as the port of the Service, and has a strictly defined format: specifically, for a Service named *foo*, the two variables are named `FOO_SERVICE_HOST` and `FOO_SERVICE_PORT` [110].

The limitations of this approach are the reason why DNS service discovery was introduced. From version 1.13, Kubernetes switched from KubeDNS to CoreDNS as the default DNS server, decision that was made because of the security vulnerabilities and the poor scaling performances, that are inborn in KubeDNS [42]. CoreDNS is, from the Kubernetes point of view, just another set of API objects: by default, when K8s is started, a Service named `kube-dns` and a Deployment named `CoreDNS` are launched in the `kube-system` namespace. The Pods of the just appointed Deployment act like a physical DNS server, since they are in fact the ones that take requests and resolve them by returning to the client the IP address of the requested Service.

CoreDNS is composed by a single executable file, which can be extended by plugins. It is this kind of flexibility, supported by memory safety and good performances in multiprocessor systems, that determines the popularity of this product. In contrast, the performances of CoreDNS are still not comparable with other DNS servers written in C/C++ [142]. All plugins are just like the main executable file written in go and can be easily added to the server by listing them in a Corefile. If the Corefile is missing, the `whoami` plugin is assumed [111] and this will cause the DNS server to respond with the IP of the client that made the request. So, just to wrap things up, the role of the DNS server in Kubernetes is performed by a set of Pods that have in their file system a Corefile that determines their behaviour. We can divide the plugins into three categories, the first one being configuration plugins, that handle errors, health checkings, monitoring, and other related features, followed by the middleware plugins, that are the ones that really manipulate the query, and finally the backend plugins [67].

The CoreDNS service is launched with a static IP address that is not changed even if the cluster is restarted. At the same time, each Pod is launched with the `cluster-dns` flag pointing to the address of the `kube-dns` Service. The IP address of this Service is then stored in the `resolv.conf` file.

From a high level perspective, service discovery can be done either client-side or server-side [84], the latter one characterized by a load balancing layer in front of the deployed Pods. The K8s Services we discussed so far perform exactly this operation, i.e. they are a layer between communicating Pods, in which load balancing is performed by kube-proxy. There is one more type of Services we did not mention so far though, the "headless" Services, that are identified by the fact that they have no Virtual IP address assigned to them and are therefore ignored by kube-proxy [142]. When a request to resolve the IP of this kind of Service is made to CoreDNS, the response will contain all the IPs of the endpoints targeted by it, thus it is the client itself responsible for load balancing. Ultimately, this is an example of client-side service discovery. Let us finally state, that it is possible to create a headless Service without selectors, i.e. without any Pods associated with it. In such cases, the DNS query will return either a CNAME record or the IPs of any endpoint that has the same name as the Service [133].

The DNS names are of course different from the ones we are used to when browsing the internet. The complete form, termed Fully Qualified Domain Name FQDN, is composed as follows:

`<resource_name>.<namespace>.<resource_type>.<cluster_domain>`

For example, the FQDN of a Service with the name `foo` in the Kubernetes `bar` namespace of the `cluster.local` domain is `foo.bar.svc.cluster.local` [112]. There is a shortcut to this naming convention though: if the Service is in the same domain as the Pod that is making the DNS request, we can simply refer to the Service by its name, and if it is in another namespace we can refer to the Service as `service_name.namespace`. This feature is made possible by the `/etc/resolv.conf` file, that is created in each Pod. This file has a set of suffixes that are added to the DNS search, e.g. this file might contain the suffix `default.svc.cluster.local` if the Service is located in the default namespace. It is worth mentioning that no matter what, the client itself tries all the suffixes listed in the `resolv.conf` file, and this might cause some additional latency, especially because DNS uses UDP on the transport layer [67]. This complication can be solved by the `autopath` plugin which, assisted by the `pods verified` policy declared in the `kubernetes` plugin, causes the CoreDNS server to resolve the IP address of the Pod making the request. By doing that, CoreDNS Pods automatically add only the suffix that is required, and since these operations are done server-side, the client Pod has to make one single DNS request. This strategy is not the default one because the considerably reduced latency comes at the expense of a much greater memory consumption by the CoreDNS Pods and an increased load to the API server [67].

Another vital plugin, that is required when running CoreDNS in K8s, is the `kubernetes` plugin, which by default causes the server to monitor all Services and endpoints in the cluster. Data about those objects is stored in an in-memory cache, that is this way always up-to-date [142]. Since CoreDNS does not need to query the API server, the responses to DNS queries are very fast. On the other hand, keeping track of all the endpoints can cause a lot of memory and CPU consumption, thus it is highly recommended to disable Pod monitoring if in the cluster there are no headless Services [142]. This is done by adding the option `noendpoints` to the `kubernetes` plugin policy.

Chapter 4

API gateways in Kubernetes

By now we have seen how is the microservices architecture structured, how to make microservices communicate with one another, and how can a client access an API by using a Kubernetes Service. Now, what other issues crop up when creating an application or API with the microservices pattern? In this chapter, we will debate that along with how can an API gateway assist us in improving the performance, scalability, and other key functions of our infrastructure. The principles we introduce in this chapter are going to guide us when extending the capabilities of the Strimzi Bridge in chapter 6, and at the same time, are going to give us the knowledge required to deploy an API gateway on the cloud, as we shall do in chapter 7.

4.1 Role of API gateways in Kubernetes

In the following pages, we are going to analyze some more challenges that the adoption of an MSA based app introduces. For most of these challenges, there is no unanimous consent on which ones are the best practices and techniques to solve them, so we will propose several ones. Among these solutions, we will see that API gateways will often turn up and in fact, API gateway providers have kept piling up the number of tasks delegated to such a component over the last few years, but nevertheless we shall always keep in mind throughout this discussion, that API gateways have their own limits and should not be expected to make miracles [41].

Before going on, let us shortly define what an API gateway is: it is a component, deployed usually as yet another (micro)service, that is placed in front of the microservice infrastructure and serves as the single entry point for API requests [45]. This definition is correct from a high-level perspective, although it might be argued that an API gateway should not be defined as a single point of entry, for this would imply that the monolith architecture has simply been pushed away from the business logic to the API gateway itself [41]. In fact, this component can be scaled in and out just like any other Kubernetes Deployment, so keeping that in mind, we will retain the above stated definition.

In the following sections we will examine the challenges introduced by the microservices architecture and see at the same time, how can an API gateway assist us when dealing with them - the sequence with which these challenges are listed adheres to no particular ordering.

4.1.1 Routing and API composition

As we mentioned already, the Client to Microservice Direct Communication requires each microservice to have a public endpoint with an associated load balancer, which is responsible to distribute the traffic across the Pods that comprise the service itself. Even if this option might be good enough for remote mobile apps and SPA web applications [90], it is worth noting, that with such a design decision, one single IP might not be enough to run the app, especially if more than one microservice needs to run on a specific port and if L7 routing is not an option.

One of the main tasks of an API gateway is therefore to act as a reverse proxy: when the gateway receives a request it consults a routing map and decides to which backend service to route the request to. The requests might be routed based on URL path, query string, HTTP headers, origin location or, in general, any parameter of the request.

This behavior makes the gateway encapsulate the internal structure of the application [136], which has some security benefits on one hand but is a prerequisite for achieving API composition as well. Data in an MSA application is usually distributed, often by applying what is known as the

Database per Service Pattern, in which the data associated with a service is kept private to the service itself and is made available only through an API. This ensures that the microservices are loosely coupled and furthermore that each service can choose the database that is best suited to its needs [130]. A number of downsides come into play though: server-side, ensuring consistency in transactions that span multiple services is not straightforward, but this can be solved by using the “Saga” method, in which transactions happen sequentially and can be undone if any local transaction fails [132]. At the same time, the database per service pattern might force the client to make multiple API calls to retrieve all the required data. Giving such a responsibility to the client is usually considered a bad practice for it leads, due to multiple API calls over the internet, to an increased latency and to some overhead in the client caused by the establishment of multiple connections [106]. The API composition pattern is a way of solving such a difficulty: in this pattern, an API composer is given the task of fetching data from multiple services and performing an in-memory join of that data in order to compose the response, that will be eventually delivered to the client. In simple terms, the API composer gives to the client the illusion that the API is coarse grained, even if the API is in fact the ensemble of multiple fine grained APIs.

The role of an API composer might be assumed by an API gateway, even if some gateways commonly used today, such as Kong [43], do not offer this kind of service. On the contrary, developers are able to perform API composition when using the Tyk API gateway: in this regard, a JavaScript function, named “virtual endpoint”, needs to be set up and associated with the route that, when invoked, will cause the function to be executed. Inside the function, a batch of API requests are triggered and later on composed into a single object, that will constitute the response [48].

An alternative to API composition done by the gateway is, of course, to set up a stand-alone service with this responsibility.

The API composition pattern faces some difficulties, namely an increased overhead, the risk of reduced availability, the lack of transactional data consistency, and possibly the inefficiency of queries [144]. Another approach is available, specifically the Command Query Responsibility Segregation pattern, in which data is read from a separate "view" database, that stores a replica of the data, that is otherwise distributed in multiple services. Changes in the services' local databases trigger events that update the view database - it is worth noting though, that changes will not be immediately visible in the view database because of the replication lag. Another downside of this strategy is an increased complexity of the architecture, but on the other hand, a better separation of concerns is achieved, since the services' databases are responsible for storing the data, while the view database is only expected to run queries.

4.1.2 Limits of OSFA

Early APIs were designed with the so-called "One Size Fits All" strategy: as the name suggests, with this kind of approach the API provider publishes an API and lets any device use it, as long as that device adheres to the constraints set up by the API itself. This results in the requests being treated in a generic manner, with no optimization done for the type of device that made the request. Devices might in fact differ in screen sizes, memory capacity, type of user interaction, and a whole lot of other ways and furthermore, since different devices might need different data, it is the client that, if the API is fine grained, has the responsibility to make multiple API calls to collect all the data it needs. On the other hand, the client might also be given unnecessary information as is the case with coarse grained APIs [80]. The OSFA approach is beneficial to the API provider for the server-side code is much simpler, but its detriments are not negligible: among others, let us here mention a more complex client code and an increased latency, caused by the multiple API calls that happen over the network [3].

Different approaches to solve these limitations were proposed: for example with the batching calls strategy, multiple API calls are embedded into a single HTTP request, fact that on one hand allows the server to optimize the queries, running the independent queries in parallel and the dependent ones sequentially [105], and on the other hand to reduce the overhead in the client caused by the multiple HTTP connections [106]. Another approach proposed was the introduction of query based APIs, that adopt a basic orchestration layer responsible for passing to the client only the requested parameters [6]. These approaches, although an improvement of the OSFA methodology, are still limited in that the client is still not given an optimized payload [141]. For example, neither of these strategies was suitable for the Netflix API, which as of 2012 provided the API services to more than 800 device types. Catering features to such a breadth of devices became too difficult to manage, so Netflix introduced the Client Adapter Code in between the Client Code and the Server Code [3]. In such a scenario, the client makes a request to the public endpoint that is specific to the device type, which results in an adapter receiving and exploding the request in an effort to gather together all the necessary data. Later on, the adapter has to combine together the data it receives and process it for delivery, i.e prune out the unwanted fields, format the data in the way that is most suited for the client, manage errors, and so on. This approach results in a simplified Client Code and at the same time improves the performance and efficiency of the transactions [3]. Since the adapter's code is specific to the single device, the role of designing it might be delegated to the UI team, as was the case in Netflix.

Today, even if some APIs allow the client to specify the fields they wish to receive, which by the way is a type of query based API and a fair choice when serving a broad range of external applications, some gateways are given the task of providing device-specific APIs. Such gateways are deployed with multiple endpoints, that are specifically tailored for a device type - the differentiation between devices might be made based on screen size, operating system or any other parameter that the application requires. When this kind of approach is taken, the gateway needs to change architecture, being now composed by two layers: an API layer, composed by several modules that contain what Netflix called the Client Adapter Code, and a common layer, which implements the functionalities that are common across all device types [136]. Of course, each module might be given different responsibilities: for example, a module might simply map one API operation to the corresponding service, while more complex modules might be given tasks such as the composition of different API calls into one single response for the client - this function is discussed more in depth in subsection 4.1.1.

Instead of having a single gateway with multiple modules, it is possible to deploy completely independent API gateways as well, each one being composed by a module and its own common layer. This technique, denominated Backends For Frontends (BFF) pattern, has a number of advantages: from a managerial point of view, the ownerships and responsibilities are always clearly defined, and at the same time, from an operational point of view, this pattern allows to improve reliability because of the isolation of the modules. Furthermore, when this pattern is adopted, a single gateway can be scaled according to the traffic it needs to handle, which is a benefit similar to the one encountered when switching from monolith to microservices. In fact, even Netflix eventually adopted this kind of structure for its own API gateways, with different containerized modules invoking a second layer API gateway called Netflix Falcor [136]. An example of the just exposed BFF pattern is shown in Figure 4.1.

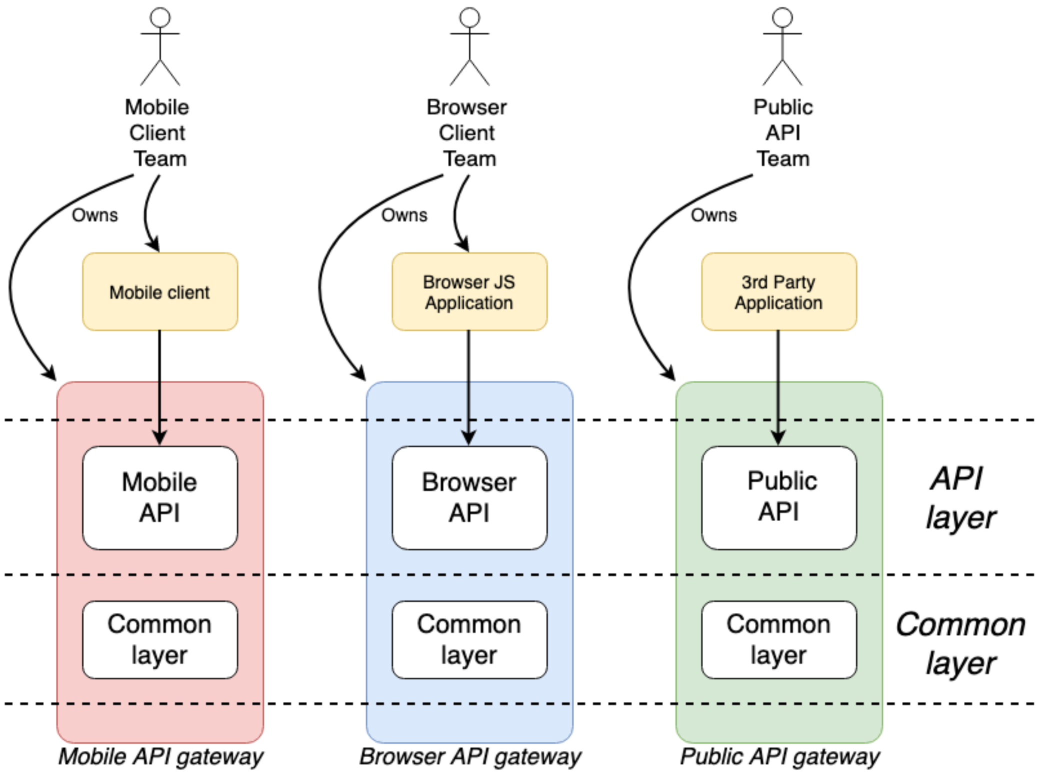

Figure 4.1: BFF pattern.

4.1.3 Authentication and Authorization

Authentication is the process that makes a system establish confidence in the identity of a user, while authorization is the following step, i.e. is the process by means of which the system grants (or not) to a user the privileges to perform some kind of operation on a resource [145]. It is a very common scenario in MSA that multiple microservices require these functions and in such cases, maintaining the principle of single responsibility is not a process without adversities. This principle expects us to implement the authentication and authorization checks separate from business logic or, in other words, these security checks should not be performed by the microservices in which the business logic resides.

In monolithic applications, in order to cope with the statelessness of the HTTP protocol, the strategy most commonly implemented is the use of sessions: when the user authenticates, a session is created and is associated with an ID, which is passed back to the client in the form of a cookie. In other words, the server is stateful, since the session resides only in the server that first responded. It is self-evident that this is not a feasible method for building an MSA application in Kubernetes because when multiple copies of the same Pod exist and are furthermore meant to fail, restart and scale, keeping a session alive in a single instance might lead to difficulties; for example, a load balancer might distribute incoming requests from the same client to different backend Pods. One solution is to configure the API gateway - or in general load balancer - in such a way, that all requests coming from a client are dispatched to the same node. Given that each node maintains a set of sessions, this method allows us to create sticky sessions similar to the ones used by monolith apps. As a downside, if the requests of a user need at some point to be redirected to another node, e.g. because the Pod, in which the session resides, has failed and is being restarted, the user would be required to log-in again and create this way another session in the new node. Given that Kubernetes Pods are by definition ephemeral, this solution is not the preferable one. Another option would be to replicate the session data across all the nodes in the cluster, but this would result in a bandwidth overhead, so a better approach is to use a centralized session storage, in which all session data is kept. This technique improves availability and scalability, but it needs to be implemented carefully as the session storage has to have appropriate security measures [59]. Another approach, that does not necessarily involve the adoption of an API gateway, is the use of client tokens that, as opposed to what was discussed so far, do not require the servers to maintain sessions: in the token, for example JSON Web Token (JWT), the required data about the user is gathered. It is up to the client to store the token, usually in the form of a cookie, and send it along with each request.

The tasks an API gateway performs in the authentication and authorization processes can vary a lot. For example, when using tokens, an API gateway might perform a very basic local validation, i.e. check that the token the client provided did not expire and remove junk API calls as well, and then pass the request to the appropriate upstream microservice, that is required in such a case to perform the authorization check [88]. Some difficulties are encountered in such a scenario though: if the token is granted to an application on behalf of a user, the user cannot revoke the permission until when the token expires. Again, an API gateway can play a fundamental role here, since it might be associated with an internal database, in which the tokens are stored; each token is given a unique reference, which is passed to the client instead of the token itself, so that the API gateway is given the task of translating the reference the client sends along with each request to the actual token. If such a strategy is used, a user can revoke the permissions to an app by simply issuing a command to the API gateway [33]. Figure 4.2 graphically shows how does this process happen. Alternatively, if such repudiation of tokens in an issue that is unlikely to arise, the API gateway might simply be given the responsibility to make the token opaque [59], i.e. transform it in a way that the client cannot extract the data from [99].

The API gateway might be given an active role in the authorization check as well. The privileges a user has are usually stored in an Authorization Server (AS) and the gateway, that stands in front of the app, might check whether a client is allowed to perform an operation before allowing the request to reach the appointed microservice [60]. This way the microservices are secured even further, since unauthorized requests are prevented from even reaching them.

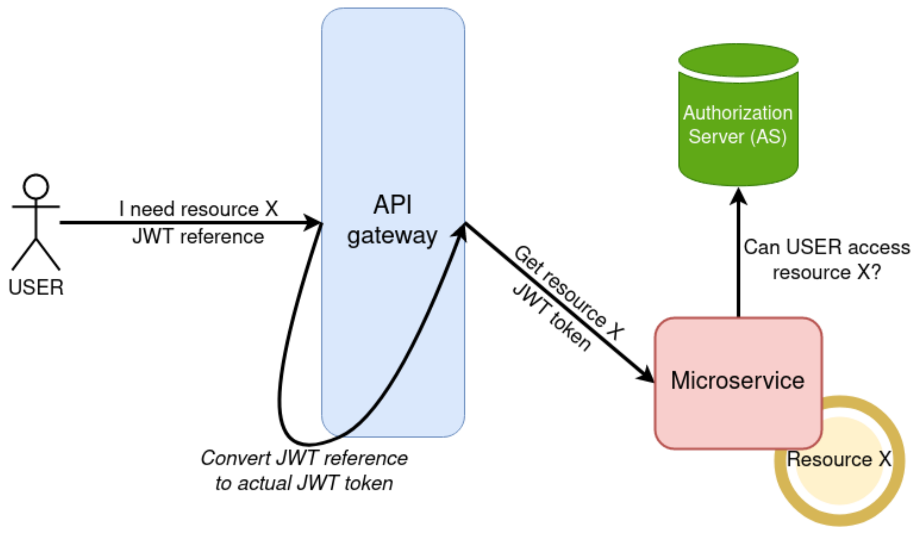
Authorization Server (AS)
I need resource X
JWT reference
USER
API gateway
Get resource X
JWT token
Can USER access resource X?
Microservice
Resource X
Convert JWT reference to actual JWT token

Figure 4.2: Role of API gateway in the authentication process.

Each API gateway may implement authentication and authorization in a rather unique way: for example, the AWS gateway makes the user configure the permissions with IAM or Incognito, while the actual check on whether the user is allowed to access a microservice is delegated to a Lambda authorizer function [139].

To conclude our discussion, let us clarify that the authorization and authentication processes are not only meant to block or forward requests, but might also be configured in such a way that requests are forwarded by setting some limitation to their extent. For example, we might provide a limited set of features to unsubscribed users and a more complete and better performing service to the subscribed ones, a scenario that the Kong API gateway documentation defines with the name "Anonymous authentication".

4.1.4 Ensuring high availability

Ensuring high availability in a distributed system such as an MSA application is more challenging than when running a single monolith application. The reason for that is the distributed nature of the system itself, since the disruption of a single service might, in the worst case scenario, undermine the correct responses of other services. Mathematically, this means that if n

services are running, each one with an availability expressed in percentage of a, the overall availability of the system is a^n [144]. A number of techniques should be implemented in order to ensure a satisfying behavior of the app and in the following lines, we are going to discuss how can an API gateway assist us in achieving that.

Every microservice should be highly available and resilient which means, at least in the case of Kubernetes, that multiple copies of the same microservice Pod should be running at the same time. This way, if one copy fails, as it often does, another copy will be at disposal, ready to take on requests. A load balancing mechanism is therefore needed: in Kubernetes, as we discussed thoroughly in chapter 3, this task is taken care of by Services. As a downside, Kubernetes Services offer this function in a rather primitive way in the sense, that kube-proxy operates on L4, which may not be an ideal option with today's application-centric protocols [85]. For example, when using gRPC, a L4 load balancer would not distribute requests coming from the same client to multiple backends because gRPC establishes a single keep-alive TCP connection across which the requests, sent by a client to a server, are multiplexed [49]. On the other hand, when using REST, a simple L4 load balancer would not be capable of distributing requests coming from a client to the same backend Pod, so sticky sessions cannot be established - this happens because REST, as opposed to gRPC, is neither multiplexed nor keep-alive.

In order to avoid such distresses, L7 load balancing is sometimes preferred, even if it introduces an increased latency due, among other reasons, to the decryption process of the incoming message [89]. This role can be assumed by API gateways that, in order to achieve an optimal load balancing strategy, might need to bypass Kubernetes Services altogether. For example Ambassador, an API gateway built on top of Envoy proxy, has two load balancing strategies: by default, a resolver is set up to perform Kubernetes Service-Level Discovery, so that the gateway only has to forward incoming traffic to the appointed Kubernetes Service - in such a case, load balancing is performed by the Service itself. If we want to use the load balancing features offered by the gateway, the resolver needs to be configured to use the Kubernetes Endpoint Resolver [134], which will make the Ambassador instances monitor the Kubernetes Endpoints API in order to keep track of all the Pods running. By doing so, the gateway will be able to forward requests directly to Pods and there will be no need to make use of Kubernetes Services [85]. Since the load balancing process happens in L7, parameters such as cookies and headers can be taken into consideration when deciding to which backend to forward the request to. This makes it possible to achieve sticky sessions [73], but it is still worth noting, that this kind of sessions are inherently fragile given the fact that Kubernetes Pods are ephemeral [28].

In general, API gateways are capable of performing load balancing with a number of algorithms, such as Ring Hash, Maglev and others [85].

Of course, the API gateway must guarantee the same level of availability as the backend services, otherwise all the efforts done will be vain. A single, "Edge" proxy API gateway suffers the fact that it is a single point of failure and might become a scaling bottleneck as well [28]. Therefore, the most common strategy to accomplish high availability of API gateways is to have multiple instances of the gateway running. These gateway Pods perform, among other tasks, L7 load balancing, while an upstream L4 load balancer is used to distribute traffic to these Pods [28].

Deploying multiple copies of the same microservice might not be enough though: in fact, when sending requests to a backend Pod, either from an external client or from another service, there is no way to be sure that the backend Pod is healthy. For example, a Pod might be incapable of handling requests because it ran out of database connections yet still be running [131]. Of course, Kubernetes tries to keep every component of the system always up to date with regard to which instances are healthy and which ones are not, but still, an additional security mechanism is usually used. The mechanism in question is a circuit breaker, that detects, by means of some health checks, whether a backend is healthy or not. If the service works as expected, the circuit breaker acts as a closed switch, allowing traffic to flow unhindered to the backends. On the other hand, if too many requests fail either because of timeout or immediate error, the switch opens for a certain amount of time. After the just mentioned time has elapsed, the circuit breaker tries to send some sample requests and, based on the response, either open for some time again or close itself. API gateways might be given such duty -

for example, Kong supports both active checks, in which the API gateway periodically probes the health status of upstream Pods with ping requests, and passive checks, in which the gateway only monitors the responses of various microservices [117]. If a Pod is detected to be faulty, traffic is load balanced between the remaining, healthy instances of the service, and the process we just described is initiated for the flawed Pod. The gateway can be configured to immediately return a default message if all the copies of the requested backend service are experiencing some problems. Finally, it is noteworthy that the circuit breaker pattern can help to prevent a backend Pod from being overloaded in the first place [115].

4.1.5 Security features

The API gateway can perform a number of important security features in addition to the authentication and authorization ones we discussed in subsection 4.1.3.

One of such functions is input validation, which involves the API gateway in the process of checking the validity of the request. Such checks, with the potential subsequent request discard, might include limiting the message size, blocking requests that may thread to perform some SQL injection attack, performing JSON thread protection, checking that the submitted fields have the correct format, and others [66]. When using the AWS API gateway, this is achieved by setting up a request validator, which is triggered whenever an API request is received [113].

API gateways may perform rate limiting in order to avoid DoS and DDoS attacks or, in general, to prevent a misuse of the system. Again, there is a lot of flexibility regarding how to implement this function, and Traefik API gateway allows to limit the number of requests with respect to the IP addresses, so that zombie machines can be prevented from making too many requests to the upstream services, and with regard to some other parameters as well [138].

Finally, we have seen in subsection 4.1.4, that in order to perform L7 routing and load balancing, the API gateway needs to decrypt the request. After performing such decryption, the API gateway can either encrypt the message again, and act this way as a SSL/TLS Forward Proxy, or forward the data in plain-text to the upstream microservice, behaving like a Termination Proxy. The pros and cons of both strategies are pretty obvious: when data is encrypted by the API gateway once again, there is the need to set up a new SSL/TLS session with the upstream Pod and furthermore, an additional encryption/decryption cycle is required, so that the whole process is slower but more secure at the same time. On the other hand, SSL/TLS termination allows, apart from a decreased burden on the upstream service, to manage certificates in fewer places. HAProxy allows both types of proxying by encrypting and decrypting data on the fly using OpenSSL [116].

4.1.6 Other features

Another important function API gateways are used for is caching, which allows better performances in terms of latency, network traffic and cost efficiency. In general, when dealing with web applications, it is preferred to cache the data as close as possible to the client so that the cost savings and other benefits are maximized [22], and since it is usually an API gateway the very first component of the MSA infrastructure that requests encounter, it makes sense that it is this element the one implementing the caching strategy. In general, the cache can be stored either internally in the API gateway or in some external memory-caching system, such as Redis or Memcached [26]. Of course, caching can be implemented in other levels as well, such as in the client - not convenient for the gateway would still have to invoke upstream services at least once for each client -, inside the microservice itself, or finally on the database level. When deploying the application on some cloud, there might even be the chance to cache the data on the very edge of the network by using a Content Delivery Network such as AWS CloudFront or Google Cloud CDN [22, 114].

The last function we are going to analyze a little deeper is protocol translation, a feature usually offered by API gateways which allows a client to use HTTP, WebSocket or some other web-friendly protocol to communicate with the gateway, while other protocols are used behind the scenes for the communication between microservices and the gateway. This requirement derives from the fact that, just to make an example, some protocols such as gRPC, whose popularity has grown dramatically over the last few years [46], are not supported on modern browsers [92].

API gateways can potentially offer a whole lot of other features we did not cover in our discussion for the sake of brevity. Such functions might include blocking a blacklist of IP addresses, detecting bots, serving static content (HTML, JS, ...), keeping track of the number of requests done by a client for billing purposes, transforming payload, canary releases, and others [27].

To sum up, an API gateway is a logical component of the systems and its functionalities are potentially almost limitless. Nevertheless, we should ask ourselves about the consequences of this kind of centralization, and in fact, a vivid discussion is still ongoing about whether the use of the API gateways for some of the functions we listed above should be considered a good practice or not. We will therefore conclude this section with the following phrases, that appeared only 2 years ago in the TechRadar Vol.17[65]:

> Vendors in the highly competitive API gateway market are [...] adding features through which they attempt to differentiate their products. This results in OVERAMBITIOUS API GATEWAY products whose functionality [...] encourages designs that continue to be difficult to test and deploy. [...] any domain smarts should live in applications or services.

4.2 Ingress and ingress controller

We discussed thoroughly in chapter 3 some ways that allow us to let external traffic reach the Kubernetes Pods. We saw the limits of the bare NodePort Service, spanning from the limited range of the nodePort field value to the additional requirement to keep track of all the nodes in the cluster, which makes it impossible to have a static IP associated with a microservice. At the same time, the LoadBalancer Service might be an alternative, even if in fact such a Service still utilizes NodePort Services under the hood [124], hiding them behind the mask provided by a load balancer deployed on some cloud. The LoadBalancer Service is therefore a step into the right direction, but then again it still has some non negligible downsides, the most prominent one arguabely being the one-to-one mapping between the load balancer and the service associated with it, which might result in the necessity of having multiple IP addresses at disposal and multiple load balancers running. Both of these requirements directly cause a price increase [56], but they can be avoided by exposing a single microservice that provides all the functionalities offered by API gateways in it, so that a single load balancer is needed to distribute traffic among the Pods forming the API gateway Deployment. This is a perfectly viable way to create an API gateway and in fact, as we shall see shortly, Ambassador is built using this very strategy [83]. On the other hand, Kubernetes offers a couple of API objects that we did not encounter so far, objects that were built specifically to route incoming traffic to the cluster. These tools are Ingress and Ingress Controllers, and today most API gateway vendors utilize such tools in the implementation of their products.

An Ingress API object, sometimes referenced to as Ingress Resource, is a single entry point to the Kubernetes cluster and essentially consists of a set of routing rules. Considering that it centralizes a number of tasks such as load balancing, SSL/TLS termination, and name-based virtual hosting, we might argue that Ingress is the most powerful and useful way to expose microservices in Kubernetes [52]. According to the type of routing it performs, we can break down the Ingress resources into the following categories:

- Single service: it is the most basic Ingress configuration, in which all the incoming traffic is routed to the very same upstream service. In such a case, there is no much profit from the usage of Ingress, since by the bare adoption of a Kubernetes Service we can achieve the same result;
- Simple fanout: the routing is done according to the URI of the request;
- Name-based virtual hosting: in such a case multiple hostnames, i.e. URLs, map to the same IP address. The routing is therefore performed

according to the value of the "Host" header field of the request, that by the way is required in both HTTP and Websocket protocols;

- Combination: of course, a combination of the simple fanout and name-based virtual hosting is easily achieved.

We can set up SSL/TLS termination in the Ingress definition file as well. In order to perform such a task, the Ingress resource must be associated with one or more Secret API objects, in which pairs of public certificates, along with the related private keys, are specified. The link with the Ingress resource is later on done by setting the `.spec.tls[*].secretName` field of the Ingress resource to have the same value of the `.metadata.name` field of the Secret. This approach causes SSL/TLS termination on a proxy level, i.e. the secure session is terminated on the proxy so that the traffic inside of the cluster flows unencrypted. If, in contrast to that, there is the yearning to pass encrypted data even between the Ingress Controller and the backend Pods, both the Ingress Controller and the Ingress resource need to be modified [57].

The Ingress resource is just like any other Kubernetes object defined using native primitives, that are independent of the Controller. This fact is particularly important because an Ingress resource on its own is not capable of performing any actions since it needs an Ingress Controller to fulfill the tasks it was entrusted with. Somehow surprisingly, by default, Kubernetes is not provided with an Ingress Controller, even if there are two Ingress Controllers that are officially supported by the Kubernetes community, namely the Nginx and the GCE ones. In other words, users are given the possibility to choose the Ingress Controller that best suits their needs, although it is worth noting, that a more detailed analysis of the available Controllers brings to light the fact that the differences among them are almost negligible [123]: some are based on Envoy instead of Nginx or HAProxy while others might allow more routing algorithms; some might be provided with different authentication methods while some others might use an external database instead of storing the data in the provided ETCD data store. Nevertheless, deep down, the functional differences between the most popular Ingress controllers, such as Kong, Gloo, Tyk, are almost hardly noticeable, as we can see in Table 4.1 [87, 123].

	Routing	Dashboard	Based on
`Kong`	Host, header, path, HTTP method	Grafana, Prometheus, Admin dashboard	Nginx
`Gloo`	Host, header, path, HTTP method, query parameter	Grafana, Prometheus, Admin dashboard	Envoy
`Tyk`	Host, header, path, HTTP method	Grafana, Admin dashboard	Tyk
`Nginx`	Host, path	Grafana	Nginx
`Ambassador`	Host, header, path	Grafana, Prometheus	Envoy

Table 4.1: Comparison of different Ingress Controllers

All this variety of the Ingress Controllers makes it very difficult to generalize how they actually operate under the hood: technically, an Ingress Controller can be any system capable of reverse proxying [14], but the internal structure of such components differs greatly among various vendors. Still, there are some common characteristics across different Ingress Controllers, such as the fact that they all have the responsibility to listen for changes in the Kubernetes API, and furthermore that they are always deployed as a Kubernetes Deployment. The structure of the Pods that make up the just-mentioned Deployment is unique to the vendor, and since we are unable to make a general assumption of such architectures, we will here make a couple of examples: the Nginx Ingress Controller Pods are composed by a single container, in which both the monitoring and the proxying tasks are embedded. On the other hand, Kong separates these two duties into two containers, one being the Kong controller, which monitors the Kubernetes API for changes and configures the Kong instance, and the other one being the Kong proxy itself, that is the component that indeed manages all the traffic. The changes of the Kubernetes API can be notified to the Ingress Controller using callback functions [129].

More differences between different Ingress Controllers crop up when implementing them: for example, different Controllers support different annotations, i.e. non-identifying metadata used by some libraries and tools to attach some behavior to the object. It is quite common though, that the user might not even be aware of such details, since Controller vendors might abstract those specifics by giving an easier way to manage the API gateway. It is the case for example of Kong and Tyk, that allow to configure APIs, plugins and similar by making REST requests to the API gateway itself.

As we stated above though, the usage of Ingress is not the only way to let external traffic reach the cluster, because while it is true that a lot of popular API gateways use these tools, some exceptions exist. It is the case for example of the Ambassador API gateway: the developers that created this piece of software opted for a completely different strategy because, they sustain, Ingress has a very limited set of features with respect to Envoy's capabilities and because they wanted to provide a way to control the API gateway that resembled more the Kubernetes fashion, avoiding the employment of REST APIs for configuration [44]. As for the internal architecture, the Ambassador Pod consists of a single container made up of the Envoy proxy and the Ambassador control plane - in section 4.3 we will take a look at how these two components cooperate.

It should be clear by now what the terms of the relation between an Ingress Controller and an API gateway are: an API gateway is a proxy that during its activity might perform a (sub)set of the tasks exposed in section 4.1; in Kubernetes, this can be achieved either by giving such responsibilities to an Ingress Controller, in which case, the terms Ingress Controller and API gateway have no fundamental differences [53], or by creating a standalone service. It is worth noting though, that the differences between those two concepts are quite blurry: for example, in the Kubernetes documentation it is stated that Ambassador is "an Envoy based ingress controller" [120], even though, as we just explained, Ambassador does not utilize Ingress for its configuration. Such a flexible use of the terms is made possible because the final outcome is the same regardless of the underlying implementation and because the user is not expected to know the details we just went through.

Note however that regardless of the underlying structure, the API gateway Deployment only lives inside of the cluster, so that the necessity of deploying a downstream Service, either type NodePort or type LoadBalancer, is inescapable. To summarize, the journey of a request done by an external client is the following: the request is issued to the IP of the most downstream load balancer, that forwards the request to a node of the cluster - it is possible to avoid this step and issue the command directly to some node, but this is not a feasible option in a production environment because of the downsides of the NodePort Service and because, needless to say, all the internet addresses of the nodes would be required to be public. On the appointed node, the NodePort Service takes the request and passes it to an API gateway Pod, which performs the tasks the gateway was entrusted with and then proxies the request to the intended Pod.

4.3 API gateways vs. L7 proxies

The last question that needs clarification is which ones are the differences between API gateways and API proxies.

In general, a proxy is a façade that is placed in between two endpoints, such as a client and a server. Given the fact that the vast majority of proxies perform load balancing functions, the terms "proxy" and "load balancer" are used roughly interchangeably in the industry [29]. The most top-level distinction we can make is by dividing proxies in forward proxies, that tend to be used in order to cache content and to bypass firewall restrictions, and reverse proxies, that are used by server admins with the purpose of taking incoming requests and forwarding them to some healthy server [69]. In other words, a reverse proxy is an intermediary for its associated server, while a forward proxy for its associated clients.

Because of their very nature, proxies are a common place where to implement load balancing and even though in the following lines we are going to focus on edge proxies, let us here just mention that the so-called side-car proxy topology, which is different from embedded client libraries in that the side-car proxy is a stand-alone, language-independent process, is gaining popularity in the last years for east-west traffic, i.e. for communication between services. Such a strategy is implemented for example by Istio, which interconnects services into the so-called service mesh [79].

In order to function properly, a proxy has to perform a number of additional tasks, including but not limited to [29]:

- Service discovery: determine the available backends. The techniques used by different vendors might differ greatly and are usually independent of the underlying infrastructure. For example, when running Envoy on Kubernetes, the load balancer is unaware of the fact that the endpoints are Pods rather than physical servers [50];
- Health checking: determine whether a backend is available to receive traffic, either with active or passive health checks. Possibly the circuit breaker pattern might be implemented as well;
- Load balancing: distribute traffic among the set of available endpoints;
- Sticky sessions: forward requests that belong to the same session to the same backend;
- Security: rate limiting, SSL/TLS termination, DoS mitigation, and others;
- Observability: load balancers are usually required to provide logs, stats and traces to help the debugging process of an application.

Even if this is just a subset of all the features a proxy might perform, it is already evident, that these features overlap with the ones of an API gateway a lot. So to answer to the question that opened this section, namely what the differences between an API proxy and an API gateway are, let us say that the disparities between these two components are somehow limited: from a functional perspective, proxies usually offer only a subset of the functionalities, that are otherwise provided by API gateways. This is why no API gateway vendor did, as the saying goes, reinvent the wheel, but rather extended the functionalities of an existing proxy in order to create the end product: for example, Kong utilizes Nginx, Ambassador and Istio run on top of Envoy, and HAProxy, Traefik and Nginx use their proprietary proxies. Therefore the main difference we may notice between gateways and proxies is on the operational level: API gateways are designed with the precise aim to allow an easy configuration of all the parameters of API proxies - it is no secret that proxies require a fairly high level of networking expertise in order to be configured [50] and API gateway vendors are trying to abstract and simplify the usage of such tools, so that developers can focus on business logic instead of the networking aspects of the infrastructure.

Again, we cannot make assertions that are valid for all API gateways, so let us just take a look at how Ambassador and Kong utilize their proxies, respectively Envoy and Nginx. At its core, Ambassador is a control plane that interacts with the data plane provided by Envoy. The control plane is the component with which the user interacts in order to set up the configuration, that has to be enacted by the data plane, and observe metrics. Given these definitions, we might say that Ambassador is primarily an engine performing text processing [76] with the following workflow:

1. The user modifies the Kubernetes configuration, either by updating annotations or by updating some Custom Resource Definition;

2. The Ambassador control plane is asynchronously notified of the changes and translates the new configuration into an abstract intermediary representation. This step is necessary because, by design, the Ambassador's conceptual model is very different from the Envoy's configuration;

3. The new configuration is applied to Envoy proxy via Envoy's Aggregated Discovery Service (ADS). In Envoy API v2, ADS replaced the hot restarts mechanism because the latter had some downsides, such as the sluggishness in applying the configuration changes and the fact that during the transition between two configurations, connections were sometimes dropped, especially when using keep-alive protocols;

4. Traffic starts flowing through the newly configured Envoy proxy.

We can see that Ambassador usually only employs features that are provided by the Envoy proxy, though it might be extended with a - as for now very limited - number of plugins, namely AuthService, LogService, RateLimitingService and TracingService [101].

Kong on the other hand utilizes the Nginx proxy, a lightweight server [87] that has some limitations, including limited observability and limited health checkings. These limitations are tempered in the Nginx Plus version, that as a downside is not open source [47]. The default configuration of Kong leaves the tasks assigned to the underlying proxy to a bare minimum, since the capabilities of the gateway are determined by the plugins, that the admins specifies. Here is a very concise description of how this happens: when running Nginx, each request goes through a number of phases, each one performing some processing on the request; for example, in the `NGX_HTTP_ACCESS_PHASE` it is verified whether a user is allowed to make a request or not. The plugins the user indicates hook into such phases by leveraging OpenResty, which is a web platform that glues together nginx core, LuaJIT, and other third party modules [17]. OpenResty allows Kong to run Lua modules (i.e. the Kong plugins) while the request is being processed. Since Kong configuration is done via an admin API, either by using the Kong API or Kong CLI, plugins can be added on the fly. Thus, there is no need to edit the underlying Nginx configuration files [18]. A more detailed treatment of the structure of Kong, including the way the Ingress Controller fits in the overall picture, is beyond the scope of this chapter.

Time to sum-up the findings of this chapter: API gateways are an extremely powerful way to expose APIs to the external world, providing a number of tasks that is potentially limitless. These gateways are built on top proxies because of the overlap of the functionalities that these two components have, and aim to provide a more user-friendly way to configure, or more in general customize, the behavior of the underlying proxies. Finally, when it comes to Kubernetes, API gateways are - sometimes improperly - automatically associated with the term Ingress Controller: while it is true, that most API gateways are implemented as an Ingress Controller, this is not always the case.

Chapter 5

Apache Kafka

In this chapter, we will provide a rapid overview of Apache Kafka and briefly discuss how does this tool foster stream processing, a paradigm that, due to the advent of big data, has been growing in popularity in recent years. Though not of the utmost importance for the goals of this paper, this discussion aims to clarify the advantages and limits of this type of data storage and at the same time, give the necessary information required in order to decide when to use an Apache Kafka cluster - and potentially the Strimzi project, which will be the subject of chapter 6 - instead of a traditional database. We will therefore examine the general principles behind Kafka rather than the actual coding details, that are required to create an app that uses this tool.

5.1 Kafka's internal architecture

So far we only considered scenarios in which two entities, be they microservices, physical devices, or any other type of software components, directly communicate with one another, implementing this way the so-called *request-response pattern*: the requesting component, e.g. a web browser, issues a request to the appointed backend service, such as an API gateway, and expects to get a response back in a reasonable amount of time. We can immediately observe that there are no intermediaries in between the two entities, so that the whole communication process happens with the lowest latency possible. On the other hand, this pattern has some undesirable characteristics too: the requested component needs to keep up with the pace the requests are issued, and furthermore, responses are subjected to the availability of the backend service. In other words, the communicating components are coupled over a number of dimensions and Apache Kafka, and in general the Event-Driven System design, addresses this kind of downsides, allowing to completely decouple the requesting and the requested component at the expense of an increased latency.

In the official documentation, Kafka is defined as a distributed streaming platform that implements the publish/subscribe messaging pattern [95]. From a high-level perspective, Kafka is a Java program that runs on a set of servers, denominated *brokers*, on which messages produced by a producer are stored, awaiting to be retrieved and processed by a consumer program. We can denote that there is an evident similarity between such a streaming platform and the traditional messaging queues such as IBM MQ, JMS, and AWS SQS, and in fact, the only difference between them is that whereas a traditional queue is meant to store messages and to delete them from the queue right after have they been forwarded to the appointed consumer program, Kafka's purpose goes beyond that: Kafka keeps all the messages, that in this context are also called events or records, even when they have been consumed by some app. The advantages of this strategy are pretty evident: a number of applications can consume the same data, and moreover, it is possible to add new consumer programs without making any changes to the set of existing ones or to the producers. Thus, Kafka stores a durable record of all transactions, so we might think that there is a correlation between this platform and traditional databases: usually, databases only store the most recent data, which is the result of all the transactions that have ever happened. This implies that if we capture all the transactions that modify a database with some Change-Data-Capture (CDC) mechanism, and store them into Kafka, we can later on *materialize* the stream of events into the resulting database table (or any other format the database uses). The resemblance between these two concepts is even more noticeable if we consider that Kafka allows to compact streams as well - a compacted stream is characterized by the fact that only the most recent inserted values for a given key are stored. Let us now take a look at what does Kafka actually offer.

As we mentioned already, Kafka runs on a set of brokers, that are grouped together into a cluster. Each broker has a number of tasks to perform, such as receiving messages from producers, assigning to each message a unique offset, committing messages to disk, and responding to fetch requests from consumers and other brokers [143]. Of course, some way to classify the messages is needed, and in fact, Kafka allows us to create topics in order to categorize similar messages - continuing the analogy Kafka-database, we might consider topics to be the equivalent of a table in a relational database. Because of horizontal scaling requirements, topics are in turn broken down into partitions, each of which can be hosted on a different broker: messages from the same topic are distributed between the existing partitions by assigning a key to the message being sent, and by using some partition strategy thereafter. For instance, if we set the key of the message to be the ID of a user, we ensure that the all the messages produced by the same user are stored inside the same partition. If, on the other hand, there is not such a requirement, we might simply fail to provide a key to the messages and distribute them this way with the round-robin algorithm. Eventually, messages might be deleted from the platform, fact that can happen either after the retention time has expired or when a certain amount of data has been exceeded. In order to improve the efficiency of such a pruning process, partitions are internally comprised of a set of segments, but this is an implementation detail we are not concerned about right now.

Each partition is owned by a single broker called *leader of the partition* or *leader replica*, that is the one that actually receives and processes all the requests for this given partition - for example, if a produce request is issued to a broker, that is not the leader of the partition to which the producer is trying to write, an error message will be returned to the client. Apart from the leader replica, in the cluster there is a set of follower replicas for each partition which are only required to replicate the data, that is otherwise stored inside the leader - a high-level view of the overall process is depicted in Figure 5.1. In Kafka, in order to avoid back-pressure, it is always the clients the ones that initiate connections and send requests, so that the follower replicas have to keep sending fetch requests to the leader replica. Such requests contain the offset of the last message the follower replica has received, and given that the offset is an increasing value, the leader replica is able to keep track of which follower brokers are up-to-date and which ones are not. In simple terms, if a broker sends a request with offset x, the leader can assume that the broker has received all the messages with offset up to x, and if the message being fetched has been written to the partition in less than y seconds (configurable parameter), the broker is considered up-to-date. Knowing which replicas are in-sync is important because when the leader replica fails, a follower replica needs to take on the role of the leader and, in order to avoid inconsistencies, only up-to-date replicas are allowed to do so. Inside the cluster one broker, called *controller*, is given some additional tasks, such as assigning partitions to brokers, designating the leaders of the partitions, monitoring for broker failures, and others [143].

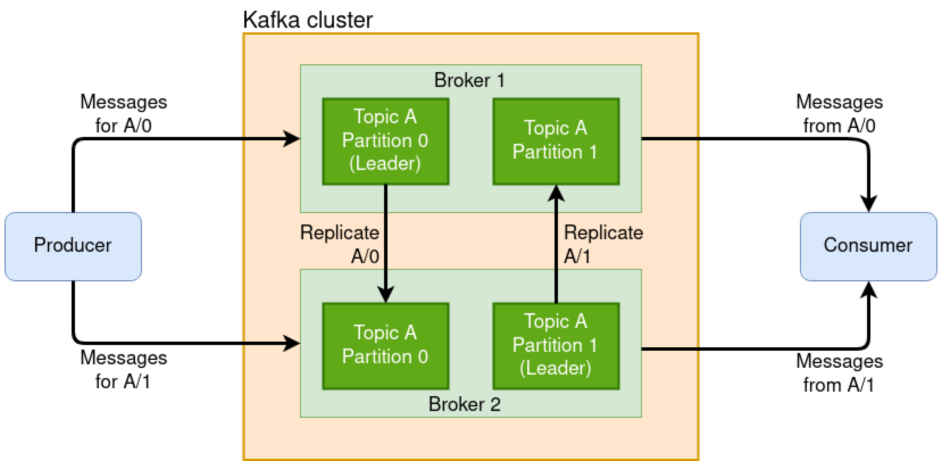

Figure 5.1: Simplified Kafka architecture.

Kafka requires a Zookeeper store to be associated with the cluster as well. This service is needed firstly in order to store metadata about the cluster along with the list of active brokers, and secondly, because it has an active role in maintaining the correct functioning of the cluster: namely, among other tasks, it keeps track of active brokers, consumers, and producers, by receiving periodical heartbeat messages from them, i.e. a client is considered no longer active if it does not send a heartbeat message to Zookeeper for a certain amount of time. Furthermore, when the controller broker fails, all the nodes try to become the new controller by sending a message to Zookeeper, but only one is appointed as the new controller, while all the others receive the "node already exists" exception raised by Zookeeper.

Kafka allows us to tune the characteristics of the system we are building in the way that best suits our needs, since, needless to say, some trade-offs between reliability, data consistency, availability, throughput, and latency need to be made when designing the system. For example, by setting the replication factor to 1, the partition will reside only in the leader replica, i.e. no follower replicas will be present. By doing that, we reduce the need for hardware resources along with the time required for a message to reach the consumer application, but as a downside, we acknowledge that a specific partition might not be available for a certain amount of time, e.g. because the broker is being restarted - by default, 3 replicas for each partition are run. Of course, there is a whole lot of other parameters that can be specified to better meet the requirements; on its own, all that Kafka guarantees are the following properties [143]:

1. The messages inside a partition are stored in the order they are received;
2. The messages Kafka receives are considered committed when they are written to all the replicas of a partition, so that committed messages are not lost as long as one replica remains alive;
3. Consumers can only read committed messages.

5.2 Kafka Producer API

The Kafka Producer API abstracts most of Kafka's architectural details and offers a neat way to write new messages to brokers without the need for manually managing the produce messages, that are part of the Kafka binary protocol.

In fact, when a Java producer wishes to send a message to Kafka, it first needs to create a `ProducerRecord` object, which is composed of a topic, a value, and optionally a key as well. The just-created object is later on serialized by a serializer and, after being grouped with other messages meant for the same partition into batches, sent to the leader of the partition. Usually, it is considered good practice to compress the batch before sending it for this causes a reduced overhead on the network as well as less storage consumption on the brokers - batches are stored on the servers as they travel on wire, with the only addition of some metadata fields. Of course, clients must first find out which one is the leader broker for the appointed partition, and in fact, they do so by issuing a metadata request to some node in the cluster. Metadata responses are usually cached by clients, however, these tasks are taken care of by the API, so that developers are relieved of such responsibilities. The leader broker receives this way a batch of messages and checks whether the request is valid - for example, the broker might verify whether the producer has enough privileges to write to the partition. At this point, the broker might return an error message, which might be either retriable, e.g. the `LEADER_NOT_AVAILABLE` error, or non-retriable, e.g. `INVALID_CONFIGURATION`. On the other hand, if the message is accepted and written to the partition, the server might not immediately return a response that states that the record was successfully posted. This happens because in the request message there is a parameter called `acks`, that is used in order to tailor the behavior of the broker to best suit the reliability needs of the application: it might happen for example that the leader broker fails before the message has been fetched by the other replicas. In such cases, a replica without the new message is elected as the new leader, and if the producer that sent the message is unaware that such a disruption has happened in the Kafka cluster, the message is irreversibly lost. Therefore, if the application we are developing cannot afford data loss, we should set `acks=all`, which will cause the leader replica to wait until all the follower replicas fetch the new message before responding to the client, that originally sent the produce request. Needless to say, in such cases, the producer needs to listen, either synchronously or asynchronously, for the response of the server, while on the other hand, when it is acceptable that some messages get lost, the sender might ignore the return message from the broker, implementing this way the strategy called "Fire and forget" [143].

Note that with the bare adoption of the just exposed techniques, we can achieve the *at least once semantics* or the *at most once semantics*, depending on the type of behavior the producer takes when timeout errors happen. As from version 0.11.x, Kafka supports the *exactly once semantics* too, which behind the scenes detects duplicate batches by attaching to each batch a sequence number and the ID, that is unique to a specific producer [70].

5.3 Kafka Consumer API

On the other side of the Kafka cluster there are consumers, that are the ones that finally read and process somehow the data. In order to achieve horizontal scaling, multiple consumers can cooperate while consuming a topic, and this is done by creating consumer groups that consist of multiple consumer instances, each of which is assigned a subset of the existing partitions of the topic. The distribution is carried out in such a way that the partitions assigned to each consumer form an exact cover of the set of the partitions in the topic. This implies that the number of partitions sets an upper bound to the number of possible consumers, so that if a topic consisting of n partitions is being read by a consumer group formed by $m > n$ consumers, $m - n$ consumers are bound to stay idle until some consumer fails - for obvious reasons, the number of partitions cannot be decreased, while it is possible to add more partitions [39]. In an attempt to reach high availability and resiliency, when a consumer fails or a new one is added, a rebalance process is triggered and the partitions are relocated to the current set of available producers, but unfortunately, this process has a number of concerns: for example, it causes a short window of unavailability of the whole group, in which consumers are unable to consume messages, and furthermore, if no specific strategy is in place, following the rebalance the consumers will lose their internal state. The latter might in fact be written to an internal and compacted change-log topic, but this does not guarantee the exactly-once read semantics because it might happen, that a consumer is restarted before being able to commit its internal state [96].

Each consumer instance subscribes to a list of topics and joins a consumer group - if such a group is not specified, a new one is generated and the consumer is assigned to it. Luckily enough, the memberships to the groups are handled by Kafka without the need for any further intervention: for example, Kafka causes the consumers to periodically send heartbeat messages to the broker, that has been designated as the group coordinator, and if they fail to do so, they are considered dead, so that the rebalance process is triggered. Apart from the just-mentioned one-time configurations, each consumer is required to perform two core tasks during its activity.

First, consumers need to pull data from the partitions they own, i.e. the partitions that are currently assigned to them. This is done by invoking the `poll(Timer timer)` method, which causes the producer to send a fetch request to the leader replica. When receiving such a message, the broker first checks whether the amount of new data, that has been written to the partition since the last poll request, is larger than the minimum size of the response, which was specified in the request. If the new messages combined fail to reach such a parameter, the broker does not immediately return the response. Instead, it waits until the minimum size is reached, yet when the time frame specified in the `timer` parameter expires, it sends the response back regardless of whether the minimum size has been reached or not. In the client code, the `poll` method is usually invoked in an infinite loop.

The other key function consumers need to perform is committing the current offset. As we mentioned already, each message stored in Kafka is associated with a unique and ever-growing integer value called offset, which is a piece of metadata that aims to avoid the same message from being processed multiple times. In fact, consumers retrieve messages in the same order as they have been written to the partition, and commit at the same time the value of the latest message they have processed. This way, if a rebalance process is triggered and the partition is assigned to a new consumer, the latter will be able to retrieve the offset associated with that partition and will therefore know which one is the next message that needs processing. In other words, if the committed value is x, the new consumer can safely assume that all messages with offset lower than or equal to x have successfully been processed by some consumer in the same consumer group as its own, and will therefore start fetching the messages with offset greater than or equal to $x+1$. The commits are stored inside a specific Kafka topic called `__consumer_offset`, and needless to say, if a consumer fails to commit some messages, which might happen because of some disruption, the exactly once semantics will very likely be violated, i.e. the not-committed messages might get processed multiple times. Given that the offset has such an important role, the default commit strategy, which commits the offset every *y* seconds (configurable parameter), might not accommodate the needs of some application, and in such cases, there is the necessity to explicitly implement a commit strategy in the consumer code. One strategy commonly used is the one that implements asynchronous commits, in which each processed message is individually committed and at the same time the outcomes of such commit messages are ignored - this is done to avoid an outdated offset value being stored. The synchronous committing strategy on the other hand might be used when the producer needs to be sure that a specific offset was successfully committed, e.g. before a repartitioning process.

5.4 Stream Processing in Kafka

Stream processing, also known as real-time analytics or event processing, is a paradigm that aims to solve some of the challenges that companies are facing in today's data-centric world [20], in which data is considered even more valuable than oil itself [19]. For example, businesses crave to get real-time analytics and need at the same time some mechanisms to tame the vast datasets at disposal in such a way, that some business value can be extracted from them [9]. By themselves, the Kafka APIs we discussed so far do not offer stream processing, but of course they put at disposal of developers all the tools required to achieve it. On the other hand, among the five Kafka core APIs there is also the Kafka Streams API, but before discussing it, let us explain what stream processing is.

A data stream is an ever-growing dataset of immutable events [143]. This definition is broad to such an extent that almost every business activity can potentially be considered a data stream: for example credit card transactions, API requests, package deliveries, and emails sent, could all be considered event streams. Following this introduction, we can define stream processing as the ongoing processing of one or more event streams. Even though messages are processed as soon as they are generated, some latency is still introduced, at least if comparing this pattern to the request/response one. On the other hand, the latencies that are in place with stream processing do not even come close to the ones of the batch processing paradigm, which could be in the order of hours. Therefore, stream processing might be considered a step in between the two alternatives we just listed, and at the same time their generalization [4].

In order to facilitate the creation of applications that require stream processing, Kafka developed the Streams API. The latter leverages the capabilities offered by both the Producer and the Consumer API, abstracting them in such a way, that the developers can focus on business functionalities rather than on managing the Kafka infrastructure: for example, the API offers stateful processing capabilities and manages all by itself the offset-committing task. It is important to note here that every streaming application takes on the role of a producer and a consumer at the same time, since the workflow it adheres to is generally the following: first, the application reads the data on a per-message basis from a topic, then it applies some processing to it, and finally writes the result back to some topic. Truth be said, this API uses a slightly different nomenclature from the one we used so far: topics are abstracted to form streams, messages are now called records or events, and the processing that alters messages is called topology.

In fact, the topology is the set of transformations that are applied to a stream, and is usually represented as a directed graph: in this digraph the nodes, called stream processors, represent the operation being applied to the stream, while the edges represent the streams being generated by the parent node of the arrow. For example, some node might perform a `filter` operation, so that the incoming stream would be mapped to a stream deprived of some events - in the corresponding topology, these streams would be depicted as the incoming and the outgoing edges to the node, which is performing the filtering. However, two nodes serve a special purpose: namely, the source processor is the one that has no upstream nodes and is given the task to produce an input stream by fetching data from one or more topics and, opposed to that, the sink processor is the one with no downstream nodes, so it is the one that receives the stream, on which all the transformations have already been applied, and writes it back to some Kafka topic.

The Streams DSL API explicitly models the stream-table duality with two abstractions, namely `KStream`, which represents an abstraction of a stream of unmutable and independent facts, and `KTable`, that abstracts a stream of evolving facts and that might therefore use behind the scenes some mechanism to compact the corresponding topic.

Chapter 6

Extending Strimzi Bridge's capabilities

In this chapter we are going to discuss whether some of the principles, that were thoroughly discussed in chapter 4, can be applied to Apache Kafka instances. We will at the same time implement the proposed solutions, extending the functionalities offered by the Strimzi Bridge.

All the code presented in this chapter can be found on GitHub in the following repositories: *borisrado/StrimziBridge* and *borisrado/Operator*.

6.1 Strimzi

Strimzi is an open source software that is being developed under the CNCF wing, currently as a sandbox project [71], and aims to simplify the deployment of an Apache Kafka cluster inside of the Kubernetes cluster, fact that, not surprisingly, is not as straightforward as deploying some stateless business logic. Though the application layer has seen a tremendous change because of containerization, the data layer has not gotten as much traction, but is still rather limited to use cases where some kind of data loss is acceptable, e.g. for caching purposes. Reasons for that might be sought in the fact that the likelihood of fail-overs of the database are much higher in Kubernetes than in traditional hosted solutions - keep in mind that Pods are ephemeral, so subjected to frequent restarts -, as well as the need for the containerized workloads to be resilient to scaling and other types of constraints. Nevertheless in recent years, due to the developers' desire of treating data infrastructure the same way as the application stack, the "run it on Kubernetes" approach for the data layer has been rapidly gaining attention [93], and the Strimzi project might be considered to be a direct consequence of that.

As we can see in Figure 6.1, the architecture of Strimzi is similar to the one a traditional Kafka deployment, with its Kafka brokers and its Zookeeper instances. There is one major addition in place though, namely the Kafka Bridge, which is a component that allows clients, that might be either internal to the same Kubernetes cluster as the Strimzi Deployment is in, or in any other way external to such cluster, to communicate with the Kafka brokers by making use of the HTTP protocol - as of today, the AMQP protocol is supported as well, but we are going to focus the attention to the HTTP protocol only. In other words, the Bridge provides a RESTful interface that allows clients to perform various actions, included but not limited to sending messages to topics, fetching data from topics, create new and delete existing consumers, and others [97], without the need to use the Kafka binary protocol that, as we discussed already, is needed in order to communicate with the Kafka brokers. The advantages of using the bridge can be identified quickly enough: separation of concerns, ability to code microservices even in languages that do not have libraries for dealing with the Kafka protocol, capability to send requests from web browsers, and others.

Needless to say, the task of the Bridge is taken on by a Kubernetes Deployment, so the necessity for having an upstream API gateway for north-south traffic is in no way reduced, though of course, Strimzi can be used for logging messages produced by microservices as well. Either way, after receiving an HTTP request, the Bridge extracts the necessary information from it, e.g. the topic and partition to which the messages are meant to be written to, the body of the messages, the consumer-groups that need to be updated, and so on, and starts the communication with the Kafka brokers - in fact, not all the routes to which the Bridge answers to require communication with the backend brokers. For instance, when making a `GET` request to the `/healthy` route, only the health status of the Bridge is checked. Eventually, when the Bridge concludes the tasks it was supposed to do, the original HTTP request is answered.

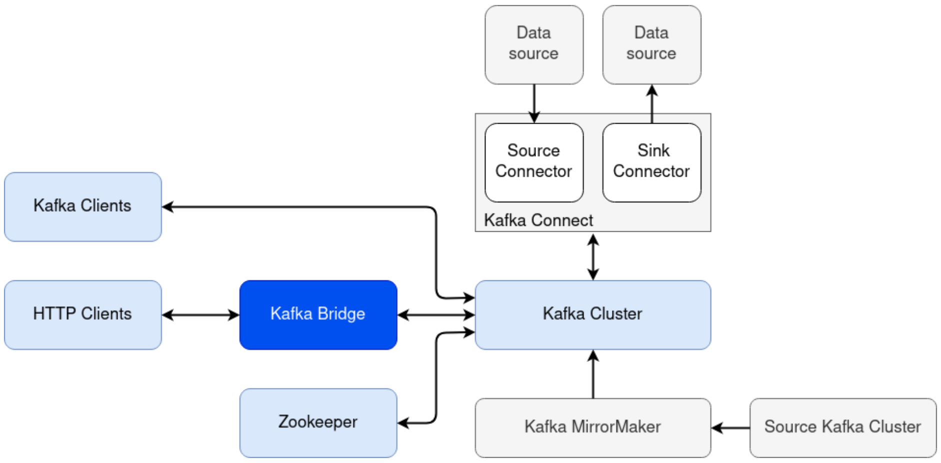

Figure 6.1: Strimzi architecture.

6.1.1 Strimzi Bridge analysis

Let us first shortly analyze how is the Strimzi Bridge built and the journey each request goes through before being answered. A basic understanding of these is required in order to appreciate the solutions we are about to present.

The Strimzi Bridge is build using Vert.x Java toolkit [135], which offers functionalities that permit to easily set up TCP and HTTP clients and servers as well as other components. This framework is somehow similar to the well renowned Node.js runtime (in fact, the original name Node.x has been changed in order to avoid naming disputes), and as the latter allows us to build scalable, non-blocking and concurrent applications. Another commonality they have is the concept of *reactive programming*, sharing the idea that an event loop should be responsible for delivering events to their appropriate handlers and for making sure at the same time, that during this process the succession with which the events are generated is maintained. Usually, although this is not an absolute requirement, the just mentioned handlers are implemented inside of independent chunks of code called *verticles*, each of which is assigned an event loop.

Vert.x applications run inside of a JVM and even if they are multi-threaded, with the maximum number of threads by default limited by the number of available CPUs, when running the Strimzi Bridge we might get the erroneous impression that the app is using one single thread: this is because by design choice, the Bridge is created with only one verticle, so that requests are always handled by the same eventloop-thread. As a consequence of that, it is extremely important not to block this thread with time consuming tasks, fact that is so crucial that in the Vert.x documentation is given the name *the golden rule*. Nevertheless, after the Bridge has successfully been deployed, several additional threads are running - a discussion of the purpose of each one of these is beyond the scope of this paper.

In order to create a verticle in Java, we must instantiate a class that implements the `Verticle` interface or extends one of its sub-classes, such as the `AbstractVerticle` abstract class - in our case, this is in fact done inside of the `HttpBridge` class, in which the `start()` method of the superclass is overridden so that the chores, required in order to bootstrap the server, are performed as the verticle is being deployed. For example, inside of this method, the Vert.x Web API Contract extension, that implements the OpenAPI 3 specification, creates a `Router` object by loading the `openapi.json` file, the content of which is later on used to dispatch requests to their appropriate handlers. More precisely, this JSON file contains the specification of the Bridge, so information such as at which route is which service available, which one is the expected format of a request, which ones are the possible responses a service might return, and others, are included - a truncated example of a JSON record contained in the file is shown in Listing 6.1. Just as importantly, each JSON record contains an `OperationId` field, which needs to match the Id of the handler that is added to the router after the `OpenAPI3RouterFactory` class has been instantiated inside of the `start()` method - the reason why the two need to be identical can be grasped quite intuitively. In the end, with this design pattern, the specification can be defined even before the services are actually implemented, hence the name "Contract-First Design", and among the benefits that we gain by using it, we can mention automatic request validation and automatic mount of security validation handlers [78], though in fact, no authentication or authorization mechanisms are currently in place for the Bridge.

Therefore, when the Bridge receives a request, the appropriate handler, which is in fact a standard Java method, is invoked and is passed an instance of a `RoutingContext` object. The latter contains all the information related to the request, such as the remote address of the connection, the headers, the query parameters, etc. Some processing of the request is done and eventually the `handle()` method, implemented either in the `HttpSourceBridgeEndpoint` or in the `HttpSinkBridgeEndpoint` class, is invoked, so that the response is finally built and sent to the client.

After this brief analysis of the existing code of the Strimzi Bridge, we can proceed to extend its functionalities.

6.2 Content-Based Routing

6.2.1 Overview and requirements

One of the core tasks an API gateway is usually entrusted with is routing. While it is true, that routing is usually the process of distributing the incoming traffic to different backends based on the URL of the request, this is not always the case: for instance we showed in subsection 4.1.2 that differentiation might also be performed based on other parameters of the request, such as the headers. Because of that, in our implementation we wish to offer a great degree of flexibility to the user, providing a standard routing solution as well as a flexible framework, that allows to implement any kind of routing the user needs to implement. Therefore, the main requirements we expect to meet are:

1. Content-based routing shall allow clients to write messages to some topic via `POST` HTTP requests. The functionality shall be made accessible on a separate route;

2. Routing rules, which will be used to dynamically determine to which topic messages will be sent to, shall be given to the Bridge by means of a configuration file when the Bridge is being started;

3. A possibly-empty set of rules, composed in turn by a non-empty set of conditions, shall be checked each time a request is received in order to determine the topic, to which a bunch of messages shall be directed to. If none of such rules apply, messages shall be written to a predetermined, default topic;

4. A basic routing shall be available out-of-the-box, but no limitation shall be given to the user as for the capabilities of the routing conditions;

5. The user shall be able to load custom routing classes by downloading all the necessary JARs. When such files are downloaded, some mechanism to prevent any kind of conflicts with the classes used by the Bridge shall be in place;

6. The topics specified in the configuration file shall be automatically created;

7. All the other functionalities and properties of the Bridge shall remain unaltered.

With the goals of this section in place, we can proceed to analyze the extension of the Bridge we introduced.

6.2.2 Configuration and Implementation

As we explained not long ago, the mapping between routes and Java methods is performed by a Vert.x `Router` object, the properties of which are set with the aid of a JSON file when the verticle is being started. Given the fact, that this file contains an actual contract between the service and the client, there is the need to carefully design the record we insert as we need to take into consideration what should the request look like as well as all the possible responses that might be returned by the Bridge, including the ones returned when some kind of error occurs. The most significant parts of the record we inserted are reported in Listing 6.1.

```
"/topics": {
    "post": {
        "tags": [
            "Topics",
            "Producer"
        ],
        "description": "Sends one or more records to a dynamically-
    determined topic",
        "operationId": "sendWithCBRRules",
        "requestBody": {
            "content": {
                "application/vnd.kafka.JSON.v2+JSON": {
                    "schema": {
                        "$ref": "#/components/schemas/ProducerRecordList"
                    }
                },
                "application/vnd.kafka.binary.v2+JSON": {
                    "schema": {
                        "$ref": "#/components/schemas/ProducerRecordList"
                    }
                }
            },
            "required": true
        },
        "responses": {
            "200": {
                ...
            }
            "400": {
                "description": "content-based routing is not enabled, so
    this route cannot be accessed",
                "content": {
                    "application/vnd.kafka.v2+JSON": {
                        "schema": {
                            "$ref": "#/components/schemas/Error"
                        }
                    }
                }
            }
        },
        ...
    }
}
```

Listing 6.1: /src/main/resources/openapi.json

With the addition of the above snippet and the appropriate changes to the Java source code (see `HttpBridge` class), clients become capable of sending `POST` requests containing either JSON or any kind of binary data type, to the `/topics` route. In the snippet we can also appreciate that a number of errors have been anticipated - as an example is reported the `400: BAD REQUEST` error, that happens if the client tries to access the content-based routing route when such option is not enabled by the configuration of the Bridge.

As for the format of the routing configuration, we choose to use a plain

JSON file. The choice falls on such format for several reasons, but the most prominent one is the presence of some well-performing and robust Java libraries that are able to validate the content of a JSON file against a JSON schema. By doing that, the configuration is assured to have the required format: for instance, following the schema validation, we can safely assume that strings have a determined length, that an array is non-empty, that a property is present, and a whole lot of other facts, and this tremendously simplifies the Java code, relieving us from manually performing any kind of similar checks. In fact, if the user provides a file that is not compliant with the schema, the `validate()` method of the Everit JSON-schema library raises a `ValidationException` exception and the Bridge is consequently started with the content-based routing option disabled. We might consider in such cases not to start the Bridge in the first place by calling the `System.exit(1)` method, but this would cause the Bridge to enter the `CrashLoopBackOff` state, fact that would cause a number of concerns to the Cluster Operator - the latter is discussed in section 6.4.

By taking a look at the JSON schema, we can see that the fields expected in the routing configuration file are almost a verbatim translation of the requirements we expressed above: the topmost object is therefore composed by a possibly empty array of routing rules - note that in JSON there is not the notion of a set -, and a default topic, which is in turn composed at its very minimum by a string representing its name. Each rule of the array is later on composed by a non-empty set of conditions, and it is precisely these the ones that provide the degree of flexibility we require. Sure enough, a routing condition can be either of the following types:

- Headers-only: in such a case, no additional work is expected from the user, who is only required to decide which header is going to be considered by the condition, whether this header must be present or not - this allows for example authenticated users, that are the ones who attach to their requests a header stating their identity, to send messages to different topics than anonymous users -, and optionally the value of the header, expressed as a regular expression;

- Custom-classes (downloaded from URL): when the user wishes to perform any other type of routing but the one depicted in the previous point, there is the need to use custom classes. In these classes, any kind of routing is possible, e.g. it is possible to inspect the body of the message and, by means of some mechanism, determine whether the message is meant to be for example a compliment or protest. The user is expected to group the required JARs into a ZIP file and make the latter available over the network to the Pods. Apart from the URL at which the ZIP should be downloaded, in the JSON record there is also the need to specify the name of the class that contains the `isSatisfied(Map<String,String> headers, String body)` method, which is called by the Bridge in order to determine whether the condition is satisfied or not;

- Custom classes (downloaded from Maven repository): this functionality works exactly as the previous one with the only difference that in this case, the classes are downloaded from the Maven repository, so the artifactId, groupId and version are required in place of the URL.

It is not surprising that also the Java classes that were introduced in order to keep at disposal the configuration during runtime faithfully reflect the content of the JSON file:

- `RoutingPolicy`: an instance of this class, that is stored inside of the Bridge configuration object `BridgeConfig`, contains all the necessary data needed in order to perform the content-based routing. For example, it contains data about the default topic as well as an array of `RoutingRule` objects;

- `RoutingRule`: consists of the name of the topic that messages are meant to be written to if all of the conditions set up by this rule are met. Because of the requirement 5. listed earlier, the class contains also an array of custom class loaders that, as opposed to the default class loading strategy, load classes with the child-first approach: this way, when a ZIP file containing all the necessary JARs is downloaded and its content placed inside of a separate folder (code available in the `DependencyLoader` class), the user-defined JARs are run inside of their own separate environment, i.e. if two classes have the same name, the class loaded by the user is going to have the precedence against the class available in the default classpath. This prevents any kind of dependency conflicts to happen. If the routing is based only on the value of the headers of the HTTP request, the method `isSatisfied()` of the `RoutingCondition` is going to be executed, while if the user uses custom classes for routing, the `isSatisfied()` method of the custom class is invoked instead. Even if the user is suggested to implement the `RoutingConditionInterface` in the custom class, this is not an absolute requirement: the reason for that is our decision to use custom class loaders, fact that prevents us from being able to cast the custom classes to the just-mentioned interface - in Java, the class identity consists of the fully qualified class name as well as the class loader name;

- `RoutingCondition`: when using the headers-only routing, which, as we just explained, routes the messages based solely on the value of the headers, an instance of this class represents the most basic check that can be performed on a request. Based on the attributes contained in this object and the value of the headers available in the `RoutingContext` object, the `isSatisfied()` static method returns either true if the condition is satisfied and false otherwise;

- `ContentRouter`: this class has only one static method that applies the logic contained in the `RoutingPolicy` object to check whether some `RoutingRule` applies to the current request. This check is in fact performed by a basic for loop that verifies whether all the conditions of a rule are satisfied, and if they are, returns the corresponding topic name. On the other hand, if none of the rules apply, the name of the default topic is returned. Note that messages are going to be written to the very first `RoutingRule` topic that is satisfied, so the order with which the rules are specified in the configuration file plays a potentially fundamental role.

In essence, if we compare the JSON configuration file and the just-listed Java object, we should immediately denote that the parallelism is quite evident.

The way the configuration file is provided to the Bridge is described in section 6.4.

The last requirement we need to address is how to create the topics that were specified inside of the configuration file without the need for the user to manually do so. The Kafka API offers a number of utility classes that allow us to do exactly that:

```java
try {
    /**
      * @props contains information such as the bootstrap servers
      * @allTopics is a map containing information about all the
      *     topics that were specified inside of the configuration
      */
    adminClient = AdminClient.create(props);
    ArrayList<NewTopic> newTopicsList = new ArrayList<NewTopic>();
    for (Topic t : allTopics.values()) {
        newTopicsList.add(new NewTopic(t.getTopicName(),
            t.getNumOfPartitions(),
            (short) t.getReplicationFactor())
        );
    }
    CreateTopicsResult res = adminClient.createTopics(newTopicsList);
    res.all().get(); // wait for the future to complete
    log.info("Topics from the routing policy successfully created.");
} catch (Exception e) {
    if (e.getCause() instanceof TopicExistsException) {
        log.info("Some topics from the routing policy exist already");
    } else {
        throw e;
    }
}
```

Listing 6.2: RoutingPolicy.java

The API does not allow to change the number of partitions or the replication factor of the existing topics, so we decide not to offer such capabilities either. Other than that, such operations should not be performed light-heartedly: the user needs to be aware of their consequences and if there is the need to update the replication factor or the number of partitions the same, it is better if the user manually does so. By forcing the user to explicitly update such parameters we avoid any kind of inconsistencies caused by some mistake in the configuration file (e.g. messages meant for a partition A were placed inside of partition B because the number of partitions is not the expected one), as well as any kind of data loss caused by an erroneous and involuntary decrease of the number of partitions of a specific topic.

It is worth noting here, that no specific messages are returned when it comes to which topics did exist before the above snippet executes. We could do that by issuing multiple requests to create single topics, but we choose the above strategy for a rather simple reason, namely that the Bridge Pods are completely independent entities, that is, no leader is elected inside of the Deployment, leader that could be given the task to create the new topics alone. Instead, all the Bridge Pods try to create the new topics, but only the very first that accomplishes to deliver the API request succeeds in doing so, while all the others are bound to get the `TopicExistsException` exception. In other words, when we deploy multiple Bridge Pods, all but one Pod print the message that some topic exists already, even if the latter has just been created by another Bridge instance. Any kind of additional logic to provide topic metadata seemed not to have any concrete benefits - a user can easily run multiple checks on a topic to get the needed information -, so none was implemented. This strategy is somehow similar to the one that is adopted when the leader of a Kafka partition fails: in such cases, all the brokers on which that particular partition resides try to become the new leader of the partition by sending a message to Zookeeper: the first triumphs, while the others get the message, that the leader exists already.

6.3 Rate-Limiting

6.3.1 Overview and requirements

Rate-limiting is the process of controlling the pace at which a component is allowed to perform some operation or, in other words, it sets an upper bound to the number of requests a client is allowed to perform in a given amount of time. This process is performed mainly because of three reasons: to prevent any kind of (voluntary or unintentional) resource exhaustion which might, in the worst-case scenarios, cause the backend to become unavailable, to limit the impact of cascading failures, and last but not least, to meter the resource usage [61]. As we shall see shortly, which of these three is the purpose why is rate-limiting adopted is going to play a fundamental role when choosing the rate-limiting strategy to use: quite intuitively we can assert, that if the goal is to limit requests because every transaction counts, the accuracy level needs to be much higher than when the rate-limiting is performed just to protect the backend from (Distributed) Denial of Service attacks [121].

Though in principle rate-limiting can potentially be implemented in multiple places, e.g. at the source and at the sink, which are, in our case, the producers and the Kafka brokers respectively, the only option we have is to rate limit at a middleware level - this is because we might not have control over producers, and limiting at the sink would require to change the Kafka source code. The same middleware strategy is in fact the one used also by API gateways that, as was discussed in subsection 4.1.5, throttle requests after they have been sent but before they have reached the intended backend microservice: when a request is allowed to continue its journey, the client might not even note that the API gateway performed some rate-limiting check whatsoever, although some headers containing the number of requests the client has left could be added to the response. On the other hand, if the request upper bound has been reached already, the request is rejected, returning to the client a `429: Too many requests` HTTP response.

Due to the distributed nature that Kubernetes applications inherently have, a well-thought rate-limiting solution is a goal that is not as straightforward to pursue as the content-based routing we just discussed. The main difference between the two is that whereas Bridge Pods can operate completely independently of one another when performing routing, when implementing rate-limiting we might be required to provide some kind of coordination between the relevant Pods, for the lack of doing so might lead to a non-optimal rate-limiting solution. In regards to that, a first high level distinction we can make is to divide rate-limiting strategies into the following categories:

1. Local rate-limiting: with local rate-limiting in place, each Pod keeps the data concerning rate limiting private to itself and thus, by design choice, the outcome of the throttling process might differ among the various Pods performing it. Another consequence this brings is that if no kind of coordination is in place, when the rate-limiting Deployment is, in order to meet demand, scaled from one to two Pods just to make an example, a client might be allowed to make up to twice as much API calls than what it was intended to. If the upstream load balancer or

API gateway is able to maintain sticky sessions this problem is unlikely to arise, but still a single Pod might fail and as a direct consequence of that, the upper bound might be maxed out. Another possible solution is to dynamically adjust the number of allowed requests according to the number of Pods performing rate-limiting, i.e. if n Pods are running and the allowed number of requests a client is allowed to make is m, each Pod should allow m/n requests to be processed. Again, the performances of such a strategy depend on the upstream load balancer and its ability to dispatch requests evenly among the available Pods. We can therefore conclude that regardless of the actual choices we make, a local rate-limiting strategy will not lead to a precise rate-limiting, condition that is particularly true when working in a highly dynamic, "cloud native" environment such as Kubernetes. In conclusion, this strategy should be used only when accuracy is of lesser importance, e.g. for backend protection purposes;

2. Global rate-limiting: in contrast to the above-mentioned strategy, with global rate-limiting in place, the data concerning rate-limiting is shared among all the Pods performing such a task, so that the final throttling outcome is the very same regardless of which Pod is performing it. In order to achieve that, some form of centralized data store to keep track of the request count can be used. Of course, in order to keep the results consistent, there is the need to ensure the atomicity of the transactions - such requirements can be met for example in Redis with the usage of some Lua script [62].

Kong API gateway offers both types of rate-limiting strategies, with the global rate-limiting coordinated by either a Redis cluster or by using the bare underlying datastore of each Pod, which is periodically replicated among the nodes in the cluster. On the other hand, developers at Datawire opted for a different approach for their Ambassador API gateway: namely, they decided to delegate the rate-limiting task to an external service, which is invoked over a gRPC connection each time the gateway receives a request. Needless to say, each of these strategies has its own bright and dark sides: for example, when using Ambassador, the developers are given complete freedom of choice over which rate-limiting algorithm to use and furthermore, the single responsibility principle is better met. At the same time, though, latency might be undermined and the complexity increase of the cluster, including the difficulty of the first installation, is not negligible.

For our own implementation of the rate-limiting policy we decided to lean towards what is already in place in Kong: we shall offer the possibility to use either global or local rate-limiting and the rate-limiting check shall be performed by the Strimzi Bridge itself - this will give less freedom to the user when it comes to implementing the preferred algorithm, but on the other hand will hopefully simplify the deployment of the Bridge and at the same time guarantee that good performances are not compromised. Furthermore, in order to coordinate the data relevant for the rate-limiting purposes, we chose not to use a separate Redis cluster as this would have increased the complexity of our Kubernetes infrastructure, and rather decided to opt for a data-grid such as Hazelcast. Our requirements therefore are:

1. Rate-limiting shall set an upper bound to the number of write requests a client is allowed to perform to some topic in a given amount of time - no limits shall be in place for retrieving messages from topics;

2. If the request quota has not been exceeded, the client shall get in the response, in addition to the data used to identify the just-produced messages, a header stating the number of allowed requests left and the limits the client is subjected to; otherwise, after this very same quota has been reached, a `429: TOO MANY REQUESTS` response shall be returned to the client and messages shall not have any follow-ups;

3. The rate-limiting data shall be given to the Bridge by means of a configuration file when the Bridge is being started;

4. The user shall be able to choose to use either global or local rate-limiting;

5. Clients shall be identified by either their IP address or by the value of some header;

6. Different limits shall be in place at the same time. In other words, there shall be the possibility to differentiate quotas based on the identity of a client and at the same time, different quotas shall be in place for different topics;

7. All the other functionalities and properties of the Bridge shall remain unaltered.

6.3.2 Configuration

Taken for granted that the configuration shall be provided as with content-based routing by means of a JSON file, there is at first the need to design how this file should look like.

As we just stated, there is the need to tell clients apart so that based on the somehow-established identity, a specific limit might be applied to a client or not. The identity of a client might be determined by using either its IP address or the value of some header contained in the request, and the reason why we want to offer both these options is soon said: if the clients that use the Bridge are internal to the Kubernetes cluster, we can assure firstly that each client has a unique IP address, and secondly that no authentication is happening in between the producer-microservice and the Bridge. On the other hand, when exposing the Bridge services to the outer world, the IP strategy might not be adequate - just consider what would happen when different clients would be sending requests from the same IP address because of NAT translation, or even what would happen if the same user would be sending requests from multiple devices. It is pretty evident that in both of these situations, identifying a client with an IP address might cause some perplexities; furthermore, we might even argue that malicious outside users have potentially the ability to spoof their IP address. Anyway, in such cases an API gateway performing authentication might come to the rescue: the gateway, that as we well know by now, is placed as an entry point to the cluster, authenticates the client and after a successful outcome of this process, appends a new header to the request and finally proxies the latter to the appointed upstream microservice, i.e. the Bridge in our case - just to make example, the Kong API gateway's authentication plugin appends the `X-Consumer-ID` header to the request after the identity of the user has successfully been determined.

In our configuration, in order to manage this aspect of the rate-limiting policy, we offer the possibility to specify the `groupByHeader` property that, in simple terms, specifies the header whose value uniquely identifies a client. In other words in here, the user points to the header that contains the identity of the user, so in the above-mentioned example of Kong, this parameter could have the value `X-Consumer-ID`. If the parameter is set and the request does not contain the header, the limit does not apply to the request and the next limit in the configuration is hence checked, while on the other hand, if this parameter is omitted by the configuration, the clients are identified by the remote address of the HTTP connection.

In order to give the possibility to apply to different clients different quotas - suppose the application distinguishes between users in the free and the premium tier -, two additional fields can be specified in the configuration file and again, in order attach the appropriate quota to some client, only header values are going to be considered. Specifically, inside of the configuration, we can specify the `header-name` value, which determines the name of the header, that is going to be considered by the appointed limit, and the `header-value`, the purpose of which is readily intelligible. The reason why we decided to use the headers here as well is that, as above, some downstream API gateway might attach some additional HTTP headers containing supplementary information about the client, information that might be later on user by the Bridge in order to hand out the appropriate quota to clients. If we consider Kong yet again, we can note that this gateway can be extended with the ACL plugin, which executes after the client has been authenticated by some other plugin, and attaches to the request the `X-Consumer-Groups` header containing the Comma Separated Value (CSV) representation of the groups, to which the client (in the Kong docs, the client is called consumer) belongs to. ACL stands for Access Control List and broadly speaking, it is simply a list containing information that links the client - the correct nomenclature would require us to call it the *subject* - with the resource to which the access is controlled, i.e. the upstream microservice, so the Strimzi Bridge in our case. Nevertheless, the Bridge needs not to be aware of any of this: if a limit contains the `header-name` parameter with the value `header-value`, and such parameters are reflected in the request, it applies the limit, whereas if they do not match, it checks the next limit in the stack.

Let us here make a simple example to show how all these parameters can be combined to form a simple rate-limiting strategy:

```
"topic": "quotationRequest",
    "strategy": "local",
    "limits": [
        {
            "header-name": "X-Tier",
            "header-value": "premium",
            "minutes": 20,
            "groupByHeader": "X-Identity"
        },
        {
            "header-name": "X-Tier",
            "header-value": "free",
            "minutes": 10,
            "groupByHeader": "X-Identity"
        },
        {
            "minutes": 5
        },
    ]
```

Though it is quite intuitive, let us go through the above snippet and see what is going on: authenticated users (i.e. those, with the `X-Identity` header in their request) with the value of the `X-Tier` header set to `premium`, are allowed to write up to 20 messages per minute to the `quotationRequest` topic. On the other hand, if the `X-Tier` header has the value `free`, the number of allowed requests decreases to reach only 10 requests per minute. Finally, all the other clients, i.e. the not authenticated ones as well as those with the `X-Tier` header value that does not match either of the above values, are given a default, 5 requests per minute quota - note that without this final limit, unauthenticated users would have no boundaries set. Let us finally emphasize here once again that the order with which the limits are inserted in the configuration file does matter: if the third limit was placed to the top, all clients would be given the 5 requests per minute quota - the limit is determined by a `for` loop, and the first limit that applies to the request is returned.

6.3.3 Implementation

At first, in order to enable rate-limiting, the workflow the Bridge takes when being started is the very same as the one taken when dealing with content-based routing: the configuration file provided is checked to see whether it is compliant to the JSON schema definition file and is later on parsed into a set of POJOs. The high-level logic behind such objects is the following:

- `RateLimitingPolicy`: an instance of this class contains all the data required for rate-limiting purposes so, apart from a Map containing `RateLimitInstance` objects, that are the ones that are created in order to limit access to a specific topic, and a default `RateLimitInstance` object containing the limits that are applied to all the topics but the ones with their own limits stored in the just-mentioned map, contains two booleans as well. These allow us to know which strategy was indicated to be used inside of the configuration - for example, if only using the local strategy, there is no need to spin up a Hazelcast cluster, while if just the global strategy is used, starting a new thread cleaning the unused `localBuckets` map would unnecessarily consume computing resources - more on this later in this section;

- `RateLimitInstance`: the non-empty list of `RateLimitSingleLimit` objects contained by these objects is optionally flanked by a topic name, but if this is missing, no harm is done and the limits are applied to all the topics but the ones with their own set of limits provided. Finally, whether the global or local strategy is to be used, is determined by a boolean field - therefore, all the limits in place for a specific topic as well as the default limits are bound to use the same strategy;

- `RateLimitSingleLimit`: this object represents a concrete limit, composed by the number of allowed requests per hour, per minute and per second (or a subset of these three), along with the values used to determine the identity of the user and the data required to determine the actual quota. In other words, it contains references to the `header-name`, `header-value`, and `groupByHeader` parameters we discussed above. All the fields are optional, only one among the limit values needs to be present - this is also easily observable by taking a look at the rate-limiting schema.

For simplicity, we decided to employ a ready-to-use library that allows us to set up rate-limiting, namely *bucket4j*. The latter uses the token bucket algorithm, which derives its name from the analogy with a fixed capacity bucket into which tokens, that in our case represent the number of messages the client is still allowed to write to the specified topic, are added at a fixed rate. In this regard, we decided to refill the buckets intervally, though we could have opted for a more gradual and greedy refilling strategy that, instead of adding for example 6 buckets every minute as it happens with the interval refill we adopted, adds 1 bucket every 10 seconds. The interval refill allows some request spikes to happen immediately after the buckets have been refilled, which is arguably a drawback, but uses less computational resources.

The concept of a physical bucket is translated rather literally into code with the `Bucket` class: objects of this class are instantiated with a unique name that, because of our requirements, is composed by the IP address of the client or its identity, i.e. the value of some header, followed by the name of the topic, to which the limit is being applied - if the limit applies to all the topics but the ones with their own limit, the topic name is simply left out. Additionally, when creating a bucket, that is going to be later on stored inside of some map - which type of map differs based on whether the bucket is stored locally or on Hazelcast -, we need to provide a configuration that contains the number of requests the client is allowed to perform, i.e. the bucket capacity, as well as the refilling strategy. As an example, let us take the code of the local strategy:

```java
/**
 * @bridge is an instance of HttpSourceBridgeEndpoint class containing
    all data of the request
 * @bucketName is the name of the bucket with format "identity"+"topic_name"
    as explained above
 * @headers is a map containing the headers of the request
 * @limit is an Optional object wrapping a RateLimitSingleLimit object,
    that applies to the request
 */
BucketDate bucketDate = Bridge.getLocalBuckets().computeIfAbsent(bucketName,
    key -> buildNewBucket(limit.get(), headers));
bucketDate.setLocalDate();
```

Listing 6.3: RateLimiting.java

As we can see, the buckets are in such a case stored inside of a regular Java map object - specifically we used a `ConcurrentHashMap` as is suggested in the library's documentation. It is worth noting though, that `Bucket` objects are not stored inside of this map as they are: in fact, they are wrapped inside of the `BucketDate` objects which contain, apart from the actual bucket, the time when the latter was last used. These objects were introduced because whereas in Hazelcast it is possible to set up an eviction policy that is automatically managed, when using a relatively primitive data structure such as a map we need to manually make sure that its size does not grow out of proportion. Therefore, in the Bridge we start a daemon that periodically goes through the map and evicts the records, that have not been used for a certain amount of time - because of the time windows the Bridge is capable of keeping track of, each time the eviction is run, the records that have not been used for more that 60 seconds in the case no `RateLimitSingleLimit` object uses an hour window, or 3600 seconds otherwise, are evicted, as we can see in the following snippet:

```java
synchronized (locaBuckets) {
  // @localBuckets is the local map containing all the BucketDate objects
  // @evictionTime is the time (in seconds), after which records get evicted
  for (Map.Entry<String, BucketDate> bucketDate : locaBuckets.entrySet()) {
    Duration difference = Duration.between(
        bucketDate.getValue().getLastAccessTime(),
        LocalDateTime.now()
    );
    if (difference.getSeconds() > EVICTION_TIME) {
      locaBuckets.remove(bucketDate.getKey());
    }
  }
  locaBuckets.notifyAll();
}
```

Listing 6.4: CleanLocalMapThread.java

The check is done in a `synchronized` block in order to avoid inconsistencies, and the `notifyAll()` method is called at the end, so that the threads awaiting for the eviction process to end are wakened up and are therefore able to pick up where they left.

On the other hand, when using the global strategy, data is replicated across the cluster by leveraging the capabilities of the Hazelcast in-memory data grid that, in simple terms, allows us to create an easily scalable distributed cache with auto-discovery functions. In other words, the cache automatically adjusts itself as new nodes - that in our case are KafkaBridge Pods - are added or removed. The only required actions in this regard we need to take is to create a Service and to place a reference to it inside of the XML configuration file of the cluster, available in `/config/hazelcast-cluster.xml`. Finally, we also need to give to the Bridge Pods the permission to monitor the API server, which is done by creating a ClusterRoleBinding, granting the `view` ClusterRole to the Bridge Pods.

```java
/**
  * @BUCKET_MAP is the name of the map
  * @globalBuckets is a ProxyManager object that offers an abstraction over
  *     the Hazelcast IMap. In other words, it simplifies the process of
  *     accessing storage outside of the current JVM
  */
Config hazelcastConfig = ConfigUtil.loadConfig();
hazelcastConfig.getMapConfig(BUCKET_MAP)
    .setTimeToLiveSeconds(EVICTION_TIME)
    .setEvictionPolicy(EvictionPolicy.LRU);

HazelcastInstance hazelcastCluster = Hazelcast
        .newHazelcastInstance(hazelcastConfig);
IMap<String, GridBucketState> map = hazelcastCluster.getMap(BUCKET_MAP);
this.globalBuckets = Bucket4j.extension(io.github.bucket4j.grid.hazelcast
        .Hazelcast.class).proxyManagerForMap(map);
```

After the configuration file is loaded, some additional parameters are set - note that we simply declare the eviction time and the eviction strategy (Least Recently Used in this case), and after that, Hazelcast takes over and automatically looks after this aspect. Hazelcast comes with a whole set of benefits, such as high availability, resilience to single node failures as partitions are by default replicated, and others, but a more in-depth treatment of it goes beyond the scope of this paper.

Even if Hazelcast is renowned to be extremely fast, the bucket object that the Bridge Pod is trying to access might be stored on a different JVM than its own, and when such relatively slow remote memory accesses happen over the network, we ought to make use of some Java classes, that allow us to achieve asynchronous, non-blocking programming, as shown here below:

```java
/**
  * @numOfTokes is the number of messages contained by the HTTP request
  * @requestBucket is a proxy to the actual bucket - this might therefore be
     stored outside of the JVM in which the code is being executed
  */
```

```java
CompletableFuture.supplyAsync(
    () -> requestBucket.asAsync().tryConsumeAndReturnRemaining(numOfTokens))
    .thenAccept(result -> {
        try {
            processRateLimitingResults(result.get(),
                routingContext,
                topic,
                Bridge
            );
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
            HttpUtils.sendResponse(routingContext,
                HttpResponseStatus.INTERNAL_SERVER_ERROR.code(),
                BridgeContentType.JSON,
                new JSONObject().toBuffer()
            );
        }
    });
```

Listing 6.5: RateLimiting.java

With the usage of the `CompletableFuture` class we ensure that the main thread will not ever be blocked - remember the golden rule we talked about not long ago. Apart from that, the way the above code snippet works is rather simple: the result of the lambda function that is first executed, i.e. the method passed to the `supplyAsync()` method, is given as a parameter to the next method in the stack, i.e. to the lambda function of the `thenAccept()` method, the execution of which is triggered right after the first method has completed. If some error happens, a `500: INTERNAL SERVER ERROR` response with an empty body is immediately returned to the client, while if everything ended for the best, the `processRateLimitingResults()` method is invoked. The latter is common to both the local and the global rate-limiting strategies and simply determines whether the requests shall be allowed to continue their journey (`if` block) or, if the maximum number of requests has been reached, rejected (`else` block):

```java
// @probe is the object returned by the tryConsumeAndReturnRemaining method
if (probe.isConsumed()) {
    Bridge.sendMessagesToTopic(routingContext, topic, probe.
    getRemainingTokens());
} else {
    HttpUtils.sendResponse(routingContext,
        HttpResponseStatus.TOO_MANY_REQUESTS.code(),
        BridgeContentType.JSON,
        getErrorMessage(topic, probe.getNanosToWaitForRefill())
    );
}
```

Rate-limiting should now have all the components required to properly work, so we can finally move to the final stage: extending the Strimzi operator so that the changes we introduced become available for use at last.

6.4 Wrapping up with CRDs

Strimzi operates the Kafka cluster by leveraging a couple of operators, namely the *Entity Operator*, that manages topics and users, and the *Cluster Operator*, which is responsible for managing the Kafka Cluster, the Zookeeper service, the Kafka MirrorMaker, the Entity Operator itself and, more importantly to us, the Kafka Bridge. Using Kubernetes terminology, the cluster operator is an instance of a *Custom Operator*, which is in essence yet another Deployment, and even though such API objects are not strictly necessary in order to run an application, they allow us to take full advantage of the Kubernetes capability that permits us to extend the set of API objects with new Custom Resource Definitions - the KafkaBridge introduced by Strimzi is an example of such a resource.

Broadly speaking, custom operators work in a rather similar fashion as standard Deployments and most other API objects, so before going on, let us review one of the most important concepts in Kubernetes, the control loop, which is greatly used both by the controllers that ship with the Kubernetes Controller Manager as well as by the operator we are about to modify. As an example, let us consider what happens when we create a Deployment with the `kubectl` command. In fact, quite a few operations happen under the hood: firstly the YAML file, in which the metadata and the specs of the object to be created are reported, is serialized and sent to the API server, which, after receiving the request, adds the new Deployment to the ETCD key-value store. The deployment controller, that is part of the kube-controller-manager, detects that some Deployment has been added - or, in general, modified - and tries to update the state of the Kubernetes cluster in such a way, that the actual state and the desired state (i.e. the state that was specified in the YAML file by the user) match. Therefore, the deployment controller takes over and adds a new ReplicaSet to ETCD - remember that a Deployment is nothing but an abstraction over a ReplicaSet. At this point, another controller detects the change caused by the deployment controller and in turn creates a set of Pods, that currently only live inside of the ETCD memory. Finally, one last stand-alone controller, the scheduler, detects that some Pods are meant to be running and schedules them to some node, node that becomes aware thanks to kubelet of the new assignment it was given and generates the required new Pod(s). At this point, the Deployment has successfully been deployed - of course, for simplicity we reported only to the most relevant steps, but there is really much more that could be said [140]. We can now appreciate that all controllers have a quite similar behaviour: first, they watch some resources, e.g. Deployments, RepicaSets and Pods in our case, awaiting for some change to happen to them, and when some change does happen, they react via an asynchronous callback handler. The handler does the tasks it is supposed to do - e.g. the deployment controller creates a new ReplicaSet -, after the termination of which, the controller gets back to the watching state - we can now see why the name *control loop.*

To sum up, Kubernetes has a whole lot of controllers, but nevertheless Strimzi developers decided to write their own operator - an operator is the composition of a controller and its associated CRD. The main reason for that is that the standard controllers have no specific insight into what they are managing, whereas by creating a new API object type and associating it with a specific controller, we are able to determine how the system ought to behave, how it ought to react to changes and/or errors, and so on. In this specific case, when we want to create a Strimzi cluster, we first need to deploy the cluster operator along with a whole lot of CRDs, ClusterRoles and ClusterRoleBingings - the latter two are required for without them the operator would not be allowed to monitor the API server in order to detect the changes happening to the objects it is meant to deal with. With the cluster operator running, we can deploy a Kafka CRD object, and as a consequence of that, the operator takes on the responsibility to bring up a Zookeeper cluster, once this is up it reacts and creates a set of Kafka brokers, and finally deploys the entity operator as well - note that without the cluster operator, we would need to manually create each API object separately, paying attention to deploy the components with the correct order - for instance, the Zookeeper service needs to be running before the Kafka brokers.

What is relevant for the purpose of this section is what happens when a KafkaBridge API object gets created. Without diving too deep in all the tools and frameworks that the cluster operator uses - let us here just mention that the Fabric8 Kubernetes Client plays a fundamental role -, the high level view of the workflow the operator sticks to is broadly the following: as usual, changes to the KafkaBridge CRD are detected by the controller and cause the `eventReceived()` method, implemented in the `OperatorWatcher` class, to be triggered. After a brief excursus on the final method `reconcile()` of the `AbstractOperator` interface, the `createOrUpdate()` method of the `KafkaBridgeAssemblyOperator` class is invoked. In the meanwhile, the specification of the newly created (or updated) Bridge has been deserialized by a model class, namely the `KafkaBridgeSpec` - under the hood, the Jackson API is used to perform this task. The important thing here is that the changes we need to apply to the operator, so that it will be able to cope with our new Bridge, are mainly the following:

- We need to update the CustomResourceDefinition of the KafkaBridge in such a way, that some manner of providing the configuration files for content-based routing and rate-limiting to the Pods becomes available;
- The additional aforesaid parameters the user provides in the YAML file need to be deserialized inside of the `KafkaBridgeSpec` class;
- The extended configuration needs to be applied to the KafkaBridge Deployment as this is being started;
- Changes applied to the rate-limiting and content-based routing configurations need to cause the existing Pods to be shut down in favour of a set of new Pods, which are launched with the new configuration in place.

The first design choice we need to make is how to provide to the Bridge Pods the configuration files. For that purpose, we decide to use yet another Kubernetes API object, the ConfigMap: as the name suggests already, this API object is structured as a regular map, i.e. it consists of a set of key-value pairs, and broadly speaking, there are two ways ConfigMaps can be used by a Pod in Kubernetes: they can be either mounted as a volume in the Pod, or be set as environment variables to be later on accessed with the following command:

```java
String envVarValue = System.getenv().get(ENVIRONMENT_VARIABLE_NAME);
```

Though using ConfigMaps as environment variables has its own drawbacks [91], we decide to use this way for providing the configuration to the Pods. The main consequence that such a decision has is that when the ConfigMaps are changed, instead of changing the configuration of the already existing Pods, new Pods are launched - this happens because environment variables cannot be changed at runtime. On the other hand this should not be a problem, since routing rules, as well as rate-limiting rules, are supposed to be seldom changed and in such cases, a rolling release shall update the Deployment with no downtime. The bright side of this strategy is that the Pods need not to be concerned with changes of the ConfigMaps and can therefore safely assume that the configurations will never change. The Custom Operator will be appointed to keep track of such changes instead and take the required actions in order to keep everything up-to-date.

With the updates of the resource definition, the user is now required to provide a `.spec.cbrConfigMap` and a `.spec.rlConfigMap` strings, that contain the name of the ConfigMaps, in which the actual JSON configuration is stored - for convenience, it is recommended that each ConfigMap contains only one key-value pair with the value storing the actual configuration. When deserializing the JSON object, using the fabric8 `DefaultKubernetesClient` class functionalities, we retrieve the content of the ConfigMap from the API server:

```java
DefaultKubernetesClient client = new DefaultKubernetesClient();
try {
    // @name is the name of the configmap that the user specified
    Resource<ConfigMap, DoneableConfigMap> configMapResource = client
        .configMaps()
        .withName(name);
    if (configMapResource == null || configMapResource.get() == null) {
        log.error("...");
    } else {
        Map<String, String> data = configMapResource.get().getData();
        if (data.size() == 1) {
            return data.entrySet().iterator().next().getValue();
        } else if (data.containsKey(optionalKey)) {
            /**
```

```
                 * if the configmap contains multiple entries, the key of the
                 * configurations must be 'rl-config' and 'cbr-config'. If the
                 * ConfigMap contains only one entry, the key is irrelevant
                 */
                return data.get(optionalKey);
            }
        }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    client.close();
}
```

At this point, the content of the ConfigMap is at disposal and can be set as an environment variable to the Bridge when this is being started (see `KafkaBridgeCluster` class).

With this in place, we have everything to start the KafkaBridge, but still no mechanism is available for updating the Deployment when the configuration files are changed. As we mentioned above, we are going to do so by using rolling releases, that are going to start up new Pods, and as they do so, terminate the old and obsolete ones:

```
kafkaBridgeServiceAccount(namespace, Bridge)
    // other operations (scale up and down, react to pod failures, etc...)
    .compose(i -> {
        if (configFilesChanged) {
            return deploymentOperations.rollingUpdate(namespace,
                Bridge.getName(),
                operationTimeoutMs
            );
        }
        return Future.succeededFuture();
    })
    .onComplete(reconciliationResult -> { /* handle the results */ });
```

Listing 6.6: KafkaBridgeAssemblyOperator.java

Of course, there is the need for the operator to store the existing configurations so that changes happening to them are correctly treated: suppose for example that the user changes the name of the employed ConfigMap but the configuration stored in the newly appointed ConfigMap is the very same as the one stored in dismissed one, or if just some spaces are added or removed from the ConfigMap's value, which causes the content, intended as a plain String, to be different, but the behaviour that comes with it to be the very same - in such cases, there is no need to perform a rolling update. These and other situations are dealt with in the `KafkaBridgeAssemblyOperator` class.

Chapter 7

Case-study: logging messages from web browsers

The topics we wanted to discuss in this paper have almost been exhausted. In this final chapter we want to present a simple case-study that groups together all the concepts we encountered, so that the concepts, that were approached from a more theoretical point of view, may become more tangible. This discussion will hopefully clarify the way an API gateway might cooperate with the Ingress resource, how does an API gateway concretely perform the tasks we described in chapter 4, how does an MSA application look like, and at the same time, let us test in a real world example the Bridge's extensions we just introduced. Beware, that we shall not focus on creating something intended for "production", but rather prefer simplicity and clarity over complexity and robustness.

7.1 Overview

As we stated in the previous chapter, one of the main benefits that the Strimzi Bridge brings along is the ability to log messages from web browsers, so the natural choice we make is to create a website, that allows users to write messages to the Kafka brokers running remotely on Kubernetes in some cloud. An API gateway placed at the entry point of such Kubernetes cluster shall perform routing in order to proxy the messages to the microservices they are intended to reach - in our simple application, only two Deployments are reachable from outside of the cluster -, as well as authentication, and in this regard, in order to fully demonstrate the capabilities of the rate-limiting extension we provided the Strimzi Bridge with, the authentication process shall be extended by assigning each user to some group, namely either the "free" or the "premium" one. The names are of course chosen in order to mimic a real world situation, in which users might be registered in the free tier and benefit from the most basic set of services available, or could be paying for the usage of the application and be therefore offered an improved and more complete user experience. Of course, given the simplicity we demand from the application, the users shall be allowed to choose the desired group without any constraints being in place.

To summarize, our goals in this chapter are the following:

- Create the frontend of the website and make a microservice serve such static content - section 7.3;
- Code a microservice responsible for registering and de-registering users - section 7.3;
- Deploy an API gateway - specifically, we decide to employ the Kong API gateway - with the appropriate Ingress resources and the required configuration - section 7.4;
- Finally, all the components shall be deployed on GKE - section 7.5.

As we asserted in chapter 4, sometimes the API gateways might be given the task of serving static content, though this functionality, at the present time, still seems quite uncommon and is offered for example by the Amazon API gateway which, when backed by the AWS S3 service, is able to serve static content such as HTML pages, CSS style-sheets, images, and so forth, on its own. On the other hand, the Kong API gateway is not able to do so, so we need to delegate this task to a microservice and, though we could set up a separate Deployment and give it this responsibility, we decide to reuse the microservice that is already in charge of registering users, and make it serve the HTML page.

The high level view of how should the application look like once completed is depicted in Figure 7.1.

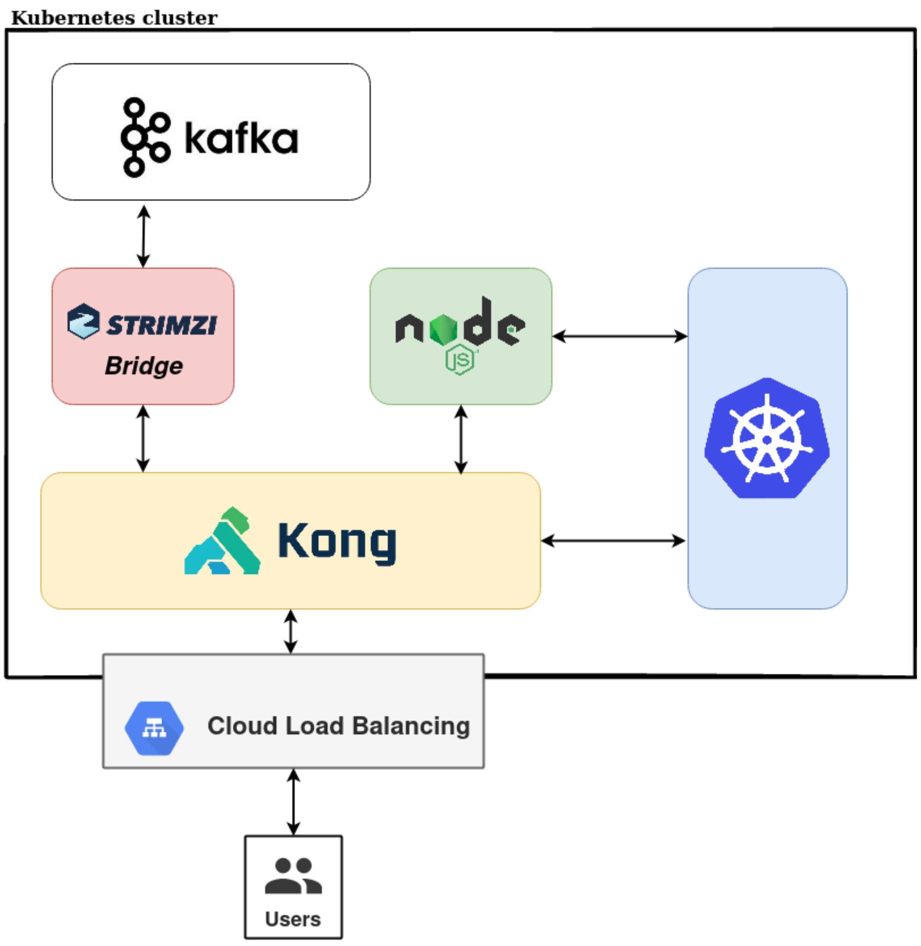

Figure 7.1: High level view of the application

The above schematic architecture has been created in an attempt to group both the networking and the implementational aspects of our application at the same time: given the fact, that the app is deployed on the cloud, the creation of a LoadBalancer type of Service, which is the default type of Service that is specified in the Kong's installation YAML file, causes the automatic deployment of an "actual" load balancer on the cloud. Needless to say, even as developers we have no insights into how is such a load balancer made up - remember that cloud computing allows us to stop thinking of the infrastructure as hardware, and think of it (and use it) as software instead [98]. The *Cloud Load Balancing* rectangle therefore represents both a downstream GCP load balancer with a static IP address and a set of NodePort Services running on the virtual machines that comprise the Kubernetes cluster.

Moving forward, each rectangle inside of the Kubernetes cluster represents a microservice: the blue rectangle on the right represents the Kubernetes API server which, as we already explained, is the frontend of the Kubernetes control plane, so it is used by Kong in order to fetch the API objects that determine its behavior (Ingress, different CRDs, and so on) as well as data about the application's users, data that is created by the Node.js microservice - the latter takes the name after the runtime in which its code runs. The purpose of all the other microservices should be immediately clear.

Let us now analyze each microservice in more depth.

7.2 Frontend description

The frontend of the application is rather simple: apart from the required HTML and the bare minimum of CSS that makes the page pleasing to the eye, it is equipped also with a very simple jQuery script, which is responsible for making a number of REST API calls to the backend - for example, when the user selects a tier, the script causes a request to be sent to the backend, specifically to the "/login" route, requesting the user to be registered. Later on, upon receiving a successful response to this request, the jQuery script makes it so that the API key, that has been generated and associated with the new user, is persisted to the `localStorage` memory and used in all the subsequent requests, that the user makes to the backend:

```
/*
 * @tier contains the name of the group the user chose, either 'free'
    or 'premium'
 */
$.get("/login", {tier}, (data, status, xhr) => {
  if(status === "success") {
    apikey = xhr.getResponseHeader("Set-Apikey");
    userNum = xhr.getResponseHeader("User-Number");
    localStorage.setItem("apikey", apikey);
    localStorage.setItem("userNum", userNum);
    alert("You have been authenticated in the " + tier + " tier")
    $("#userIdentity").html("Welcome, user" + userNum + " ...");
  } else {

    // process the error. The user is treated as an anonymous user

  }
}).fail(() => {

  // process the error. The user is treated as an anonymous user

}).always(() => $("#sending-messages").css("display", "inline"));
```

This type of approach would cause a number of security concerns if utilized in a real-world situation: for example, in the above snippet, a HTTP `GET` request, hidden behind the `$.get()` method, is used in order to login the user to the page - note that the "/login" route serves both for registering and for login purposes. The problem with such an approach is that this type of requests embeds the login data, such as the username and the password, inside of the URL, and this fact might cause a number of concerns: just to name a few, the URL is always stored as plain-text by web browsers, is subjected to being logged by some firewall logging mechanism [30], and furthermore might even be sent unencrypted during the DNS resolution process. In conclusion, for authentication requests, it is always recommended to use the HTTP `POST` method [75], possibly flanked by a more advanced challenge-response authentication algorithm [145]. This method is more secure because the data, that is in `GET` requests sent inside of the URL, is embedded inside of the body of the request. In our case, since no user data is exchanged during the authentication process, the usage of a `GET` request is a perfectly viable solution and simply gave us the pretext to make these few security considerations.

Another potential security flaw visible in the above snippet is the location where the authentication tokens are stored, that is the `localStorage` memory. The latter was introduced in 2014 with the launch of the fifth generation of the HTML language, and allows us to store data locally inside of the browser. It offers, apart from an improved API, also a far larger storage limit if compared to cookies (4kB vs. 5MB), but as a downside, it potentially exposes users to Cross-Site Scripting (XSS) attacks: the reason behind is that the data stored in this type of memory can be accessed by any JS script running in the same domain, i.e. any code present on the website has the ability to access the value of the token. Even if there are a number of countermeasures that can - and should - be taken in order to prevent this kind of attack from happening, it is usually recommended to use cookies for storing the access tokens - note that in our case we are dealing with rather primitive API key tokens, but the very same considerations can be repeated for JWTs as well. Someone playing the devil's advocate role might argue that neither cookies are immune to attacks, since they are the main culprits for a number of Cross-Site Request Forgery (CSRF) attacks that happened in the past: such attacks involve a malicious script to cause the browser to secretly perform some actions on a website in which the user is authenticated, actions that might span from requesting services on behalf of a user, as was the case with Netflix [1], to illicit money transfers [2]. Also here, a number of countermeasures can be taken in order to prevent such attacks from happening, so the assertion we just made about cookies being preferred to the `localStorage` memory for storing access tokens remains valid [16].

Both after a successful or a flawed authentication, as a result of which, no API key is returned so that the user is treated as an anonymous user, two text input boxes are displayed: in one of them the visitor is expected to write the content of the message, that is going to be written to some Kafka topic, while in the other is expected to type the value of the made-up `X-Custom-Header` header, which is attached to the request and is used by the Bridge in order to determine to which topic does the message belong to.

```
// @header and @body contain the values that have been inserted by the user
let requestHeaders = {"X-Custom-Header": header,
        "content-type":"application/vnd.kafka.json.v2+json"};
if(localStorage.getItem("apikey")) {
  requestHeaders.apikey = localStorage.getItem("apikey");
}

$.ajax({
    url: "/topics",
    headers: requestHeaders,
    type: "POST",
    data: buildRequest(body)
}).done((data, status, xhr) => {

  let allowedMinute = xhr.getResponseHeader("X-Limit-Minutes")
  let remainingMinute = xhr.getResponseHeader("X-Limit-Remaining");
  let millisUntilRefil = xhr.getResponseHeader("X-Millis-Until-Refill");
  if (millisUntilRefil != null && millisUntilRefil > 0) {
    let secondsUntilRefil = Math.ceil(millisUntilRefil / 1000);
    startCountDown(secondsUntilRefil);
  }
  let topic = data.topic;

}).fail((data, status, xhr) => {

  $("#errorResults").html("Error: <strong>" + xhr +"</strong>");
  let millisUntilRefil = data.getResponseHeader("X-Millis-Until-Refill");

});
```

Note that the above snippet has been stripped off of all the code that manipulates the DOM, for adding it would increase the length of the code without bringing any additional value to our discussion.

As we can see, the API key, that was generated when the user registered, is attached to the request inside of the `apikey` header each time a message is being sent to the Strimzi Bridge. Without this header, the user would be treated as an anonymous user and because of the configuration set up for the Kong API gateway, the request would be rejected before even reaching the Strimzi Bridge with a `401: Unauthorized` HTTP response.

Finally, let us point out how does the website make use of the extensions we introduced to the Bridge: given the fact, that rate limiting is in place, the responses to the "/topics" requests contain additional data such as the number of allowed requests per minute (see the `X-Limit-Minutes` header), the number of allowed requests remaining (`X-Limit-Remaining` header), and possibly the time left until the bucket is refilled (`X-Millis-Until-Refill`), and these values are shown to the user as they are. In other words, no further action is taken based on such values, while in a more serious situation, these parameters might be used to slow down the pace at which the requests are issued by the client, or stop such requests for a certain amount of time. Note ultimately that the frontend has no insight into how is the data assigned a topic, and furthermore that the routing rules might be dynamically updated without making any changes to the frontend.

7.3 Node.js microservice

For the implementation of the microservice responsible to register the users - note that Kong can only verify the identity of the user, but it cannot generate new users - we chose to use the JavaScript language and in particular the Node.js runtime. This choice was made with no particular reason but the one to demonstrate that in an MSA application, each microservice can be implemented in a different language and that this kind of decision does not influence in any way the other microservices. In fact, the tasks we require from this service are so basic that we could have used any language provided with a client library that facilitates the communication with the Kubernetes REST API, which is served by the Kubernetes API server.

In order to make our lives easier, we used the express framework, and equipped it with only three routes: the “/” route causes the microservice to return to the client the frontend, i.e. the HTML content and the scripts, that were described in the previous section, while the “/login” and “/logout” routes are invoked when a user wishes respectively to create a login token and to destroy it.

```
async function generateUserData(tier) {

    try {
        // get data that identify the user
        let apiKeyName = getApiKeyName(userCount);
        // the access token returned to the client has the value @secret
        let secret = makeid(8);

        //create a secret
        let keyBase64 = Buffer.from(secret).toString("base64");
        let newSecret = secretBaseImageJson
            .replace("__key_placeholder__", keyBase64);
        newSecret = newSecret.replace("__name_placeholder__", apiKeyName);
        const newSecretObject = JSON.parse(newSecret);
        client.api.v1.namespaces("kafka").secrets
            .post({ body: newSecretObject })
            .then(response -> {

                /**
                 * similarly we now create a new KongConsumer API
                 * object and associate it with the appropriate tier
                 */

                return [secret, userNum];
            });

    } catch(e) {
        /**
         * following an error, the user is not given an access token, so
         * is treated as an anonymous user
         */
        console.log(e);
        userCount++;
        return null;
    }
}
```

The code presented above leverages the capabilities of the GoDaddy library, which is an open-source, community library that makes invoking the Kubernetes API a piece of cake. Analyzing the code a little more in-depth, we can see that the `post()` method is used in order to create a new API Object, a Secret in this case, and because of the fact, that the method involves a potentially slow REST request over the network, it is provided with a callback method that executes when the Pod is notified that the Secret has successfully been created. The reason why a Secret - and, as we shall see shortly, a KongConsumer API object - is required, is explained in section 7.4.

After the code of the above snippet has successfully completed its execution, the user is returned an identification number as well as the value of the variable `secret`, which represents the value of the API key that the browser needs to send along in each subsequent request since it is used by the Kong API gateway to identify the user.

7.4 Kong API gateway

Though we already briefly introduced the structure of the Pods, that comprise the Kong Deployment in Kubernetes - section 4.3 - let us here recap the ideas behind the Kong API gateway and the way the latter can be configured. Being developed with the declared goal of being completely infrastructure-agnostic, this gateway went through some major changes in order to adapt itself to the Kubernetes environment and to its well-established practices of managing the cluster. Indeed, instead of configuring the gateway by means of REST requests made to the Kong Admin API, available on port 8001 - fact that is still now possible also in Kubernetes, though by default this port is closed for security reasons -, when deploying Kong on Kubernetes it is usually recommended to make use of the CRDs that ship with the Kong installation file. These object are monitored by the Kong Ingress Controller, which syncs the configuration from Kubernetes to the Kong gateway itself.

The first aspect of the gateway we need to configure is routing. As we discussed in section 4.2, the mapping between routes, or more in general URLs, and backend services, needs to be set up inside of an Ingress API object, which is a standard Kubernetes API object and is independent of the API gateway that is using it. In other words, the same Ingress can be reused by any Ingress Controller and the behavior of the gateway would be the very same, at least when it comes to the purely routing aspect. For example, the following YAML file causes the gateway to proxy all the requests with the "/topics" route to the `strimzi-bridge-service` Service, which is a ClusterIP Service whose endpoints are the Pods comprising the Bridge's Deployment:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-kafka
  namespace: kafka
  annotations:
    konghq.com/plugins: key-auth-strimzi-plugin, acl-strimzi-plugin
spec:
  rules:
  - http:
      paths:
      - path: /topics
        backend:
          serviceName: strimzi-bridge-service
          servicePort: 8080
```

This snippet is for the most part self-explanatory, though we still need to point out the role of the `metadata.annotations` field: inside of it, additional properties, that are unique to the Kong gateway, can be set in order to tailor the gateway's behavior to our needs. Indeed, the YAML configuration file of an Ingress Object has a clearly defined structure and can only be extended by adding new annotations, whose purpose is in fact the one to attach arbitrary and non-identifying metadata to the object - in our example, we instructed the gateway to use the plugins with the name "auth-strimzi-plugin" and "acl-strimzi-plugin" so, as these names already suggest, for the authentication of the users the key authentication plugin is used and at the same time, some Access Control List inspection is going to take place. After creating the above Ingress resource, users are going to receive the `401: Unauthaized` response if they try to access the "/topics" route without provisioning a valid API key in the request - indeed, this is the behavior we might expect from the key authentication plugin.

Therefore, each time a user registers, a couple of Kubernetes API objects need to be created. Among these objects there are Secrets, which are a type of API object we did not encounter so far, but just to give an idea about them, we can say that they are very similar to ConfigMaps with the difference, that they are obfuscated with the `base64` encoding - recall that in Listing 7.3, we used the `base64` encoding in the `toString()` method -, and were introduced for storing sensitive data such as passwords and keys. Needless to say, there is much more to the story: by default, Secrets are just `base64` encoded - they are not encrypted -, so that they are no more secure than the familiar ConfigMaps. Kubernetes v1.10 introduced therefore the concept of a Key Management System, which is a plugin in charge of encrypting the Secrets with some type of envelope encryption algorithm. This service is offered for example by plugins such as GCP Cloud KMS, AWS Encryption Provider, or HashiCorp Vault, but a more detailed discussion of Secrets and their management strategies exceeds the scope of this paper.

The objects we need to create when a user wishes to register are the following:

1. Secret: a brand new Secret, whose purpose is to store the API key linked to a user, is created each time a user registers;

2. KongConsumer: each KongConsumer CRD object represents a consumer of the application, so it is linked with a Secret containing the API key of the user, as well as with a Secret representing the ACL group to which the user belongs to.

Just for the sake of completeness, let us take a look at the `handler.lua` script, that performs the key authentication process: first, the request is searched for the API key, that, as we know, might be contained either in the headers or in the query parameters. Next it is checked whether the provided key is valid, i.e. if it is contained either in the Kong's cache or in the database associated with the gateway, and upon a successful outcome of this check, the consumer linked to the API key is retrieved and the appropriate headers are set to the request [104]. The ACL plugin executes after the user has been authenticated and, as we might expect, it simply retrieves the consumer groups associated with the KongConsumer CRD object and appends them to the request, that can be afterwards proxied to the upstream Service [100].

In the rate-limiting configuration file, that is used by the Strimzi Bridge, we need to take into consideration the headers that are introduced by the Kong API gateway, namely the `X-Consumer-Username`, that stores the username of the user - in our case, such parameter has a standard format, e.g. *user-X* for some number X - , and the `X-Consumer-Groups` header, containing in our case either the value "free-acl-tier" or the value "premium-acl-tier". Here is an extract of the rate-limiting policy we set up for the application:

```
[
  {
    "topic": "weather_discussion",
    "limits":
    [
        {
            "minutes": 3,
            "group-by-header": "X-Consumer-Username",
            "header-name": "X-Consumer-Groups",
            "header-value": "free-acl-tier"
        },
        {
            "minutes": 6,
            "group-by-header": "X-Consumer-Username",
            "header-name": "X-Consumer-Groups",
            "header-value": "premium-acl-tier"
        },
        {
          "minutes": 1
        }
    ]
  }
...
```

Since the strategy to be used is not specified, the local one is assumed.

7.5 Deployment on GKE

Running an application on some managed Kubernetes service turned out to be very similar to running the application on Minikube as we did throughout the paper.

An extremely nice feature that such managed services offer is the one that we do not have to manage any networking aspect of the infrastructure: when we generate a cluster, which is by default composed by three virtual machines, the different nodes are automatically connected to form a Kubernetes cluster and are given the appropriate roles. As a result of that, we have no direct insight into implementational details such as which networking strategies are in place or whether an overlay network is preferred to the L3 routing - we discussed these topics in chapter 2 -, and this simplifies our lives a lot.

Chapter 8

Conclusions and further work

Our journey through the wonders and pitfalls of the fascinating cloud-native world has drawn to a close. In the paper, we thoroughly analyzed how is networking dealt with in Kubernetes and, to some extent, we also discussed how does this container-orchestration system work under the hood. Likewise, we reasoned both from a theoretical and a more practical perspective which ones are the tasks that might be delegated to API gateways as well as how do these components carry out such tasks - in this regard, one of the main notions we acquired during our discussion is that even if API gateways have been given more and more responsibilities in the last years, mostly because vendors are trying to differentiate their products, we should stay distrustful of those asserting that gateways are the remedy of all the evils of MSA applications: API gateways are indeed very helpful and can simplify the development process of an application, but we should always keep in mind that centralizing too many functionalities inside of them returns us back to some kind of monolithic architecture.

We then successfully moved to extend the Strimzi open source project with some of the concepts that were introduced in the first chapters, namely rate-limiting and content-based routing. These extensions, that are as for now available to be used by either downloading the images available on Dockerhub (*borisrado/Operator* and *borisrado/Bridge*) or by building the images from the source code available on GitHub (*borisrado/StrimziBridge* and *borisrado/Operator*), might be included in a future official release of

Strimzi.

Hereafter there is a lot of room for additional improvements and new extensions: just for the sake of argument, we might consider giving the possibility to the admins to authorize the writes that happen to some Kafka topic with the OAuth 2.0 protocol.

We conclude and part with the wish that the extensions we introduced, flanked by further and unavoidable improvements of the already existing code, will hopefully guide the Strimzi project on its path towards the CNCF graduation.

Good luck, Strimzi!

Bibliography

[1] Netflix fixes cross-site request forgery hole , 2006. [Online; accessed 06-August-2020].

[2] Cross-Site Request Forgeries: Exploitation and Prevention, 2008. [Online; accessed 05-August-2020].

[3] Embracing the Differences : Inside the Netflix API Redesign, 2012. [Online; accessed 15-March-2020].

[4] The log and the stream , 2013. [Online; accessed 28-May-2020].

[5] Is REST losing its flair - REST API Alternatives, 2013. [Online; accessed 11-March-2020].

[6] The future of API design: The orchestration layer, 2013. [Online; accessed 18-March-2020].

[7] A history and future of Web APIs, 2014. [Online; accessed 11-March-2020].

[8] Microservices: a definition of this new architectural term, 2014. [Online; accessed 06-March-2020].

[9] Streaming 101: The world beyond batch , 2015. [Online; accessed 28-May-2020].

[10] A Deep Dive into Iptables and Netfilter Architecture, 2015. [Online; accessed 08-March-2020].

[11] What Led Amazon to its Own Microservices Architecture, 2015. [Online; accessed 06-March-2020].

[12] Why You Can't Talk About Microservices Without Mentioning Netflix, 2015. [Online; accessed 06-March-2020].

[13] 7 Things That Nobody Told You About Load Balancers, 2016. [Online; accessed 11-March-2020].

[14] Kubernetes Ingress, 2016. [Online; accessed 26-March-2020].

[15] The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire, 2016. [Online; accessed 07-August-2020].

[16] Where to Store your JWTs – Cookies vs HTML5 Web Storage, 2016. [Online; accessed 04-August-2020].

[17] API management with Kong, 2017. [Online; accessed 28-March-2020].

[18] Kong Architecture Overview, 2017. [Online; accessed 28-March-2020].

[19] The world's most valuable resource is no longer oil, but data , 2017. [Online; accessed 28-May-2020].

[20] Why Are We Moving Towards A Data-Centric World? , 2017. [Online; accessed 28-May-2020].

[21] 5.1 LXC Advanced Networking - Exposing Containers to the Network, 2017. [Online; accessed 27-August-2020].

[22] All you need to know about caching for serverless applications, 2017. [Online; accessed 22-March-2020].

[23] An illustrated guide to Kubernetes Networking [Part 2], 2017. [Online; accessed 10-March-2020].

[24] An illustrated guide to Kubernetes Networking [Part 2], 2017. [Online; accessed 10-March-2020].

[25] API-Led Connectivity, 2017. [Online; accessed 12-March-2020].

[26] AWS re:INVENT - cache me if you can, 2017. [Online; accessed 23-March-2020].

[27] How To Choose an API Gateway, 2017. [Online; accessed 22-March-2020].

[28] Introduction to modern network load balancing and proxying, 2017. [Online; accessed 21-March-2020].

[29] Introduction to modern network load balancing and proxying, 2017. [Online; accessed 28-March-2020].

[30] Is it bad practice to use GET method as login username/password for administrators?, 2017. [Online; accessed 05-August-2020].

[31] Kubernetes Networking, 2017. [Online; accessed 10-March-2020].

[32] Microservices, APIs and Integration, 2017. [Online; accessed 06-March-2020].

[33] Securing Microservices: The API gateway, authentication and authorization , 2017. [Online; accessed 18-March-2020].

[34] The Almighty Pause Container, 2017. [Online; accessed 07-March-2020].

[35] The History of APIs and How They Impact Your Future, 2017. [Online; accessed 06-March-2020].

[36] Think Before you NodePort in Kubernetes, 2017. [Online; accessed 11-March-2020].

[37] Understanding kubernetes networking: pods, 2017. [Online; accessed 07-March-2020].

[38] Understanding kubernetes networking: services, 2017. [Online; accessed 11-March-2020].

[39] Adding Partitions to a Topic in Apache Kafka , 2018. [Online; accessed 30-May-2020].

[40] 5 Major Benefits of Microservice Architecture, 2018. [Online; accessed 06-March-2020].

[41] An API Gateway is not the new Unicorn, 2018. [Online; accessed 22-March-2020].

[42] An Introduction to the Kubernetes DNS Service, 2018. [Online; accessed 13-March-2020].

[43] API aggregation, 2018. [Online; accessed 20-March-2020].

[44] Building Ambassador, an Open Source API Gateway on Kubernetes and Envoy, 2018. [Online; accessed 26-March-2020].

[45] Building Microservices: Using an API Gateway, 2018. [Online; accessed 22-March-2020].

[46] Building scalable microservices with gRPC, 2018. [Online; accessed 22-March-2020].

[47] Envoy vs NGINX vs HAProxy: Why the open source Ambassador API Gateway chose Envoy, 2018. [Online; accessed 28-March-2020].

[48] Example Batch Processing Aggregate Function, 2018. [Online; accessed 20-March-2020].

[49] gRPC Load Balancing on Kubernetes without Tears, 2018. [Online; accessed 21-March-2020].

[50] Intro: Envoy - Matt Klein and Constance Caramanolis, Lyft, 2018. [Online; accessed 28-March-2020; Minute 26:00].

[51] Introduction to Linux interfaces for virtual networking, 2018. [Online; accessed 07-March-2020].

[52] Kubernetes Ingress - comparison spreadsheet of different Ingress Controllers, 2018. [Online; accessed 26-March-2020].

[53] Kubernetes Ingress 101: NodePort, Load Balancers, and Ingress Controllers, 2018. [Online; accessed 26-March-2020].

[54] Kubernetes Networking, 2018. [Online; accessed 08-March-2020].

[55] Kubernetes Networking: Achieving High Performance with Calico, 2018. [Online; accessed 10-March-2020].

[56] Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?, 2018. [Online; accessed 11-March-2020].

[57] Kubernetes: Using Ingress with SSL/TLS termination and HTTP/2, 2018. [Online; accessed 27-March-2020].

[58] Microservices Aren't Magic: Handling Timeouts, 2018. [Online; accessed 30-April-2020].

[59] Microservices Authentication and Authorization Solutions, 2018. [Online; accessed 18-March-2020].

[60] Microservices Authentication and Authorization Using API Gateway , 2018. [Online; accessed 18-March-2020].

[61] Part 1: Rate Limiting: A Useful Tool with Distributed Systems, 2018. [Online; accessed 06-July-2020].

[62] Part 1: Rate Limiting: A Useful Tool with Distributed Systems, 2018. [Online; accessed 06-July-2020].

[63] Project Calico, the CNI way, 2018. [Online; accessed 10-March-2020].

[64] Setting Up CI/CD in Robot Framework, 2018. [Online; accessed 07-March-2020].

[65] Technology radar Vol.17, 2018. [Online; accessed 20-March-2020].

[66] The Role of API Gateways in API Security, 2018. [Online; accessed 22-March-2020].

[67] Understanding CoreDNS in Kubernetes, 2018. [Online; accessed 13-March-2020].

[68] What is Docker and why is it so darn popular?, 2018. [Online; accessed 07-March-2020].

[69] Differences Between Forward Proxy and Reverse Proxy, 2019. [Online; accessed 28-March-2020].

[70] Kafka Exactly Once Semantics, 2019. [Online; accessed 27-May-2020].

[71] Strimzi Apache Kafka Operator joins the CNCF, 2019. [Online; accessed 11-June-2020].

[72] 3 methods for microservice communication, 2019. [Online; accessed 12-March-2020].

[73] Advanced Load Balancing and Sticky Sessions with Ambassador, Envoy and Kubernetes, 2019. [Online; accessed 21-March-2020].

[74] API Management Microservices beyond HIP, 2019. [Online; accessed 07-March-2020].

[75] Auth: Why HTTP POST?, 2019. [Online; accessed 05-August-2020].

[76] Building an Edge Control Plane with Kubernetes and Envoy, 2019. [Online; accessed 28-March-2020].

[77] Cos'è la CNCF o Cloud Native Computing Foundation e qual è il suo ruolo?, 2019. [Online; accessed 30-March-2020].

[78] Designing a REST API — What Is Contract First?, 2019. [Online; accessed 18-July-2020].

[79] Gentle Introduction to the Envoy Proxy and Load-balancing, 2019. [Online; accessed 28-March-2020].

[80] How Granular Should You Design APIs?, 2019. [Online; accessed 18-March-2020].

[81] How to use the LXD Proxy Device to map ports between the host and the containers, 2019. [Online; accessed 27-August-2020].

[82] IPVS-Based In-Cluster Load Balancing Deep Dive, 2019. [Online; accessed 12-March-2020].

[83] K8s-based API Gateway - Steve Flanders, Omnition, 2019. [Online; accessed 26-March-2020; Minute 7:00].

[84] Kubernetes Patterns : The Service Discovery Pattern, 2019. [Online; accessed 13-March-2020].

[85] Load balancing strategies in Kubernetes, 2019. [Online; accessed 21-March-2020].

[86] Microservices Disadvantages and Advantages, 2019. [Online; accessed 06-March-2020].

[87] My experiences with API gateways..., 2019. [Online; accessed 28-March-2020].

[88] Securing Your APIs with OAuth 2.0 , 2019. [Online; accessed 18-March-2020].

[89] TCP vs HTTP(S) Load Balancing., 2019. [Online; accessed 21-March-2020].

[90] The API gateway pattern versus the Direct client-to-microservice communication, 2019. [Online; accessed 15-March-2020].

[91] The ConfigMap Pattern, 2019. [Online; accessed 14-July-2020].

[92] The state of gRPC in the browser, 2019. [Online; accessed 22-March-2020].

[93] To run or not to run a database on Kubernetes: What to consider, 2019. [Online; accessed 16-July-2020].

[94] Why Cloud Native is a must for Agile Development, 2019. [Online; accessed 07-March-2020].

[95] Kafka - Introduction, 2020. [Online; accessed 27-May-2020].

[96] Kafka Streams Internal Data Management , 2020. [Online; accessed 28-May-2020].

[97] Strimzi Overview guide (0.18.0), 2020. [Online; accessed 11-June-2020].

[98] Academy Cloud Foundations (ACF) Module 01 Student Guide , 2020. [Online; accessed 07-August-2020].

[99] Access Tokens, 2020. [Online; accessed 18-March-2020].

[100] ACL plugin - handler.lua, 2020. [Online; accessed 06-August-2020].

[101] Ambassador: Available Plugins , 2020. [Online; accessed 28-March-2020].

[102] An illustrated guide to Kubernetes Networking, 2020. [Online; accessed 09-March-2020].

[103] API Gateway: Why you need flexible deployment, 2020. [Online; accessed 12-March-2020].

[104] API key authentication - handler.lua, 2020. [Online; accessed 06-August-2020].

[105] Batch Requests, 2020. [Online; accessed 17-March-2020].

[106] Batching Requests, 2020. [Online; accessed 17-March-2020].

[107] Cluster Networking, 2020. [Online; accessed 09-March-2020].

[108] CNCF Cloud Native Definition v1.0, 2020. [Online; accessed 07-March-2020].

[109] Communication in a microservice architecture, 2020. [Online; accessed 06-March-2020].

[110] Container Environment, 2020. [Online; accessed 20-May-2020].

[111] CoreDNS Manual, 2020. [Online; accessed 14-March-2020].

[112] DNS for Services and Pods, 2020. [Online; accessed 13-August-2020].

[113] Enable request validation in API Gateway , 2020. [Online; accessed 22-March-2020].

[114] Google Cloud for AWS Professionals, 2020. [Online; accessed 22-March-2020].

[115] Guide to Microservices Resilience Patterns, 2020. [Online; accessed 22-March-2020].

[116] HAProxy SSL Termination, 2020. [Online; accessed 22-March-2020].

[117] Highly Available Microservices with Health Checks and Circuit Breakers, 2020. [Online; accessed 22-March-2020].

[118] How Elastic Load Balancing Works, 2020. [Online; accessed 11-March-2020].

[119] How to Change the Default Docker Subnet, 2020. [Online; accessed 07-March-2020].

[120] Ingress Controllers, 2020. [Online; accessed 27-March-2020].

[121] Kong Rate Limiting plugin, 2020. [Online; accessed 06-July-2020].

[122] Kubernetes concepts page, 2020. [Online; accessed 07-March-2020].

[123] Kubernetes Ingress - comparison spreadsheet of different Ingress Controllers, 2020. [Online; accessed 26-March-2020].

[124] Kubernetes Service Helm Chart, 2020. [Online; accessed 26-March-2020].

[125] Load Balancing, 2020. [Online; accessed 11-March-2020].

[126] Monolithic vs Microservices Architecture (MSA), 2020. [Online; accessed 14-August-2020].

[127] Network Load Balancer Support in Kubernetes 1.9, 2020. [Online; accessed 11-March-2020].

[128] Networking support in Mesos, 2020. [Online; accessed 27-August-2020].

[129] NGINX Ingress Controller - How it works, 2020. [Online; accessed 27-March-2020].

[130] Pattern: Database per service, 2020. [Online; accessed 20-March-2020].

[131] Pattern: Health Check API, 2020. [Online; accessed 30-April-2020].

[132] Pattern: Saga, 2020. [Online; accessed 20-March-2020].

[133] Service - Kubernetes, 2020. [Online; accessed 11-March-2020].

[134] Service Discovery and Resolver configuration, 2020. [Online; accessed 21-March-2020].

[135] Strimi Kafka Bridge - Github page, 2020. [Online; accessed 14-August-2020].

[136] The API Gateway Pattern, 2020. [Online; accessed 17-March-2020].

[137] The definitive guide to cloud native, 2020. [Online; accessed 07-March-2020].

[138] Traefik - Global Configuration, 2020. [Online; accessed 22-March-2020].

[139] Use API Gateway Lambda authorizers, 2020. [Online; accessed 18-March-2020].

[140] What happens when ... Kubernetes edition!, 2020. [Online; accessed 19-July-2020].

[141] Why REST Keeps Me Up At Night, 2020. [Online; accessed 17-March-2020].

[142] John Belamaric. *Learning CoreDNS: Configuring DNS for Cloud Native Environments.* O'Reilly, 2019.

[143] Neha Narkhede and Todd Palino. *Kafka - The Definitive Guide.* O'Reilly, 2017.

[144] Chris Richardson. *Microservices Patterns.* Manning, 2019.

[145] Laurie Stalling, William; Brown. *Computer Security: Principles and Practice.* Pearson, 2015.