AALBORG UNIVERSITY
DENMARK

API Gateways and Microservice Architectures

Master's Thesis

Richard Trebichavský

Copenhagen, Spring 2021

AALBORG UNIVERSITY

STUDENT REPORT

Department of Electronic Systems,
Innovative Communication
Technologies and Entrepreneurship
Aalborg University Copenhagen
A.C. Meyers Vænge 15,
2450, København SV
http://www.aau.dk

Title:
API Gateways and Microservices

Theme:
Master's Thesis

Project Period:
Spring Semester 2021

Project Group:
1-4.7

Participants:
Richard Trebichavský

Supervisor:
Henning Olesen

Copies: 1

Page Numbers: 69

Date of Completion:
August 6, 2021

Abstract:

This thesis aims to explore the significance of API Gateways in the context of microservice applications. We start with a theoretical background necessary to understand what is the right way of designing microservices and APIs. We follow by looking into the available API gateways solutions, determine what functionalities are required from them based on analysis of literature, case studies and interview. Finally, we choose a set of API gateways that best represent the current state-of-the-art, deploy them on the Kubernetes cluster in front of the demo application service and assess their functionalities and performance.

Contents

1 Introduction **4**

1.1. Problem definition . . . 5

1.2. Methodology . . . 6

2 Extended State of the Art **7**

2.1. API Gateways . . . 7

2.1.1 Case studies . . . 9

2.1.2 Literature . . . 13

2.1.3 Interview . . . 16

2.1.4 Functional requirements . . . 17

2.1.4.1 Authentication and authorization . . . 17

2.1.4.2 Monitoring and logging . . . 17

2.1.4.3 Routing . . . 17

2.1.4.4 Transformation of data, protocols and formats . . . 18

2.1.4.5 Service discovery . . . 18

2.1.4.6 Rate limiting and throttling . . . 18

2.1.4.7 Caching and compression . . . 19

2.1.4.8 Aggregation and response manipulation . . . 19

2.1.4.9 Circuit breaker . . . 20

2.1.4.10 Load balancing . . . 20

2.1.4.11 Support of GraphQL . . . 21

2.1.4.12 Declarative configuration . . . 21

2.1.4.13 Support for optional plug-ins and middleware . . . 21

2.2. Evolution of architectures: from Monoliths to Microservices . . . 21

2.2.1 Monolithic Architecture . . . 22

2.2.2 Service-Oriented Architecture . . . 23

2.2.3 Microservices Architecture . . . 25

2.3. Design principles of microservice applications . . . 25

2.3.1 The Twelve-Factor App . . . 25

2.3.2 Circuit Breaker pattern . . . 31

2.4. APIs in microservice architecture . . . 32

2.4.1 Representational State Transfer (REST) . . . 32

2.4.1.1 Resources, representations, URIs and URLs . . . 33

2.4.1.2 HTTP methods . . . 34

2.4.1.3 HTTP Status codes . . . 36

2.4.1.4 JavaScript Object Notation (JSON) . . . 37

2.4.1.5 Problem with JSON — no implicit semantics . . . 38

2.4.1.6 Best practices for REST APIs . . . 39
2.4.2 Backend-for-Frontend pattern (BFF) . . . 41
2.4.2.1 Netflix . . . 42
2.4.2.2 SoundCloud . . . 44
2.4.2.3 Conclusion . . . 44
2.4.3 GraphQL . . . 46

3 Implementation **48**
3.0.1 Tyk . . . 50
3.0.2 Kong . . . 51
3.0.3 KrakenD . . . 52
3.0.4 Other considered solutions . . . 52
3.0.4.1 Nginx . . . 52
3.0.4.2 Apollo Gateway . . . 53
3.1. Cloud-native microservices demo application . . . 54
3.1.1 Adding persistent storage for orders . . . 54
3.1.2 Adding REST endpoint for orders . . . 54
3.2. Installing API Gateways on Google Cloud Kubernetes cluster . . . 55
3.2.1 Installing Tyk . . . 55
3.2.2 Installing KrakenD . . . 56
3.2.3 Installing Kong . . . 56
3.3. Designing performance tests . . . 57

4 Results **59**
4.1. Comparison of functionality . . . 59
4.2. Comparison of performance . . . 60

5 Discussion **62**
5.1. Features . . . 62
5.2. Performance . . . 63
5.3. Recommendations and Future Work . . . 63

6 Conclusion **64**
6.1. Appendix . . . 69

Chapter 1

Introduction

The 21st century is a century of information. The introduction of the first mass-produced personal computer and the invention of the internet started a revolution in how the current private businesses and public institutions operate. As the internet-connected devices found it's way to the general public's homes and later into their pockets, since the Dot-com boom in the 90s, practically every business was pressured to declare its online presence to benefit from the new marketing and sales channel.

Digital transformation is an ongoing, society-wide trend that improves efficiency, unlocks new business models and creates new digital experiences [6, p. 6] and affects both the private and public sector. The promises of digital transformation include faster, more efficient processing of a large amount of information, all while reducing costs and sparing resources. Online services of businesses and institution are available to their customers at any time whenever they are. The efforts like Open data and development and standardisation of Application Programming Interfaces - APIs, make the information easier to access, interconnect and use in the context of other systems. Such reduction of friction and costs of connecting multiple systems enables rapid development of a new type of "digital-savvy" enterprises, creating a so-called API Economy.

As an example, we can take a restaurant and look at how digital services change the way it runs its business. Traditionally, a restaurant needs to run a kitchen, manage staff, operating a website and mobile app, run marketing campaigns, employing drivers and run delivery service and so on [6, p. 2]. However, most companies are in business because they do something very well [6, p. 6] — in the case of the restaurant, it would be preparing authentic and quality food. Instead of trying to secure every necessity of the business by themselves, the restaurant can choose to offload these burdens to the ecosystem partners and focus on preparing the food. So they can simply rent ready-made kitchen space and pair up with meal-delivery apps that have already optimised logistics and will offer an effective promotion as they already have a solid user base.

The enabler of these synergies is a well-designed and developer friendly API [6, p. 5]. According to research done by Google, the APIs are perceived as important when designing better digital experiences and products, accelerate innovation and are viewed as an essential cornerstone of the systems integration [6, p. 6]. It is clear that if modern businesses want to leverage most from the API Economy, they must quickly and cheaply integrate with other services and offer an API to allow other services to integrate with them. A question of how to create and manage an API layer in the best way stands as a central motivation for this work.

Throughout recent history, the notion of optimal application architectures undergoes some rather dramatic changes. What gains a lot of traction nowadays is cloud computing, microservice architecture, server-

less platforms and container technologies. It is a significant shift from the historically prevalent monolithic and service-oriented architecture. Talking about application architecture is important because it influences the task of designing and managing an API in a fundamental way.

In this project, we will examine API Gateways. API Gateways are pieces of software that are used to facilitate an interface between external clients and the collection of backend services. It acts as a reverse proxy that accepts, aggregates, manipulates and distributes API calls from clients to the internal services. The services in the microservice architecture are of different types. Some are abstract, high-level are coarse-grained that define core business operations, some provide a concrete implementation of some segment of the business functionality, and some are only infrastructure services without providing any business functionality [56, p. 14]. Typically, mutual visibility and internal communication of the services is happening in a protected subnet without direct access from the outside, although some argue that perimeter level security is not sufficient and promote a zero-trust approach [62]. Because microservice architectures could be internally quite complex, this creates a need for a thorough separation of the internal and external space to shield the external clients from the complexity and preserve the security of the inner space. All the communication between the external clients and internal services should be controlled and monitored. API Gateway offers a central place where to perform these tasks. We elaborate more on the API Gateways' functionalities and present a list of its functional requirements in section 2.1.4.

As we mentioned earlier, the API Economy is based on the effective exchange of information between services. To achieve this goal, there are a number of questions that need to be addressed: how to structure information, how to control what is exposed to whom, how to prevent misuse of the information, and last but not least, how to monetise it efficiently. API Gateway is a valuable tool that helps to achieve control over the access control and monitoring of the exposed resources. There are currently few popular different approaches to architect the APIs, for instance, SOAP, REST or GraphQL. We will discuss REST in section 2.4.1 and GraphQL in section 2.4.3.

1.1 Problem definition

The main problem that API Gateways help to solve could be formulated as following research question:

> **RQ:** *"How to successfully leverage API Gateway to efficiently implement and manage the API layer between the external clients and the system based on microservice architecture?"*

that could be further specialized by extending the question with following additions:

> **RQ.1:** *"... from the perspective of features?"*
> **RQ.2:** *"... from the perspective of performance?"*

In this project, we compare contemporary API Gateway solutions and provide a comparison of their functionalities, configuration capabilities, supported protocols, web technologies and monitoring tools, supported API architectures and the basic performance metrics.

We limited our choices to strictly only to open-source and self-hosted solutions. However, many vendors offer an easy upgrade to a more powerful paid version. Therefore, in the comparison part, we decided to also take into account claimed capabilities of the paid version. This is due to the fact that in need of extra features, it is typically much easier to upgrade to a paid version of the same API Gateway than to swap to an entirely different implementation.

1.2 Methodology

The methodology for testing and evaluating the Gateways consists of the following few steps. As the first step, in section 2.1, we define the API Gateway functional requirements and prioritise them using the MoSCoW prioritisation method. The process of identifying the functional requirements is based mainly on the literature research but also on the interviews with the actual API Gateway customers from the industry. In the second step, we deploy a simple application consisting of several microservices that will allow testing all important functionalities of the API Gateway. The application must provide an API endpoint that could be connected to the API Gateway. We use an application where the services run inside containers, as this is a common practice in the microservice architecture world. More details about the application itself and the technological decisions we made are in section 3. The third step of the methodology is to design a framework for evaluating the API Gateway functionality and performance. The first part of the evaluation process is the evaluation of the gateway's functionality. It builds on top of the API Gateway requirements from section 2.1.4 and breaks each requirement into one or more specific features, which are then comforted with the respective API gateway and evaluates whether they are supported by the gateway or not. The second part of the evaluation is performance testing. In section **??**, we design a performance test suite that we, under the same conditions, run against the API gateway and also directly against the API endpoint bypassing the gateway to provide a baseline for the comparison.

Chapter 2

Extended State of the Art

2.1 API Gateways

In the section 1, we mentioned how important it is nowadays, in the era of API Economy, to provide an API layer that enables seamless integration with a wide range of the order services the customers already use. In the following sections, we further discuss how a good API and its designing process should look like in order to yield optimal results for both API clients and providers. In this section, we try to answer why having an API Gateway could be an important component in providing a well-designed API, in which areas it could be helpful, and how does it make designing and provisioning of API layer more flexible and manageable.

Perhaps one of the major challenges of public APIs is that they need to preserve backwards-compatibility **??**. When a large number of external clients build their code and workflows around an existing public API, any changes to that API could potentially break this existing code and must be made with caution. This need for supporting existing clients could negatively affect future attempts to change, improve, and innovate the API. While making changes to API in a safe way is a general concern of API's proper strategy design and evolution, API Gateway could be helpful to make this process easier to control. When API Gateway serves as a point of contact for external clients, what it does is that it effectively separates and hides any underlying internal service infrastructure. This holds true regardless of the application's architectural style, either microservice architecture or a monolith. Once the direct coupling between clients and internal services is removed, the respective upgrades of the system's internal components are easier to do and safer to manage. As clients still connect directly to the API Gateway only, this allows to re-shuffle a rewire internal components behind the scenes without even giving external clients a chance to notice them and preserve a seamless and uninterrupted operation system whole. With API Gateway functioning as a reverse proxy for mediating application's external connections, the desired behaviours like *Zero down-time upgrades* or *Phased release* could be achieved.

Zero down-time upgrades. Suppose a client communicates directly with the internal service. When the service is upgraded, typically, all active client's connections are dropped, and the service is not able to process any further requests until its upgrade is fully completed. With the API Gateway maintaining the connection with the client instead, a new, upgraded version of the service could be started in the background while simultaneously keeping the old one still oper-

ational. After the new upgraded service completes its starting process and becomes fully operational, the gateway could just start re-routing new requests to the upgraded service instance while keeping the old active connections or sessions communicating with the old instance. Once all the old connections or sessions are closed and finalized, the old instance could safely be taken down without causing any noticeable service interruption.

Phased release and A/B testing. Any software upgrade poses a risk of causing bugs, performance issues or any other problems that negatively impact the operation of the service or other systems. To lower and better manage this risk, we could use a strategy called a phased release. The idea of phased release is that instead of serving a new, upgraded version of the application to all clients at once, we will stretch this process in time and roll the upgrade in phases, progressively increasing the number of affected clients. As an example, Table 2.1 shows how phased release could look like. In early phases, only a small part of the clients is initially served with the upgraded version of the application, while most of the other clients are intact. This significantly lowers the damage caused by production bugs when discovered in the early stages of the phased release. It also allows monitoring of the real production performance or any other behaviour that could be difficult to mimic in testing environments. Using this approach, a more stable and robust product has a higher chance of being delivered to later stages, where the cost of the production errors could be much higher. Similarly to a Phased release, a situation where different clients are served with a different version of the application at the same time also occurs during A/B testing. While this aspect is similar, the goal of A/B testing is not to eventually migrate all clients to a different version, but rather to conduct an experiment and collect data about which version of the feature, A or B, works better, either from a technical or business perspective.

Day	1	2	3	4	5	6	7
Release %	1%	2%	5%	10%	20%	50%	100%

Table 2.1: Example of *Phased release* schema used by Apple's App Store.

Although using API Gateway in front of monolithic application is already useful and brings some benefits, the full potential of the API Gateway will show up when used in front of many small independent components. Nowadays, we are primarily referring to applications using microservice architectural style. Generally, in distributed architectures, we differentiate between two types of traffic. First, an internal, service-to-service or east-west traffic. Second, the external, service-to-client or north-south traffic. A distinction between these two types is important because there exist a number of significant differences between these two types of traffic, including, for instance, different network throughput capabilities, different level of trust and security risks or legal issues with the transmission of certain types of data to other geographical locations. API Gateway could serve as a borderline between internal and external communication when all external communication needs to flow through the gateway. When API Gateways provides the single point of contact for external clients, compared to the distributed style, it makes it easier to implement, manage and control appropriate security policies, provide centralized authentication and authorization, restrict what information is exposed to whom. A single place through which all the traffic needs to pass thru also makes it an appropriate place to monitor it, which could help identify Quality-of-Service-related issues. API Gateway is also an appropri-

ate place to implement measures preventing the service overload caused by a sudden increase of legitimate traffic or targeted DDoS attacks.

While the benefits of API Gateway handling the north-south traffic are probably more apparent, it should not be forgotten that it could be successfully used to intermediate east-west traffic as well. If we take a closer look at the different services of a single microservice=based application, we will most probably identify some pieces of logic that repeat over and over again across all of them. These are called cross-cutting concerns, and they typically include, for example, authorization and authentication, logging, service discovery or configuration management. We could use API Gateway to abstract these concerns, reducing the complexity of individual microservices as a result. This will make it easier and more convenient for the developers to focus more on the core purpose of the service, its business logic, instead of some infrastructure technicalities.

2.1.1 Case studies

At this point, it should be apparent that the API Gateway is software that lies on a perimeter of the application's internal space and outer world and somehow mediates the communication between them. This description is, however, very general. To understand what specifically API Gateway should do and what are its most requested functionalities, we analyzed a number of API Gateway case studies published on the website of Tyk, one of the popular open-source API Gateways. This was done in addition to the papers, blog posts we read and interview we conducted.

On a Tyk's web page `https://tyk.io/why-tyk/case-studies/`, there is a list of 19 case studies that could be filtered according to the region, sector or use case. The use cases used in each case study company are shown in Table 2.2. Each case study description begins with a short description of the company or institution and its business. Next follows the motivation why the company needed an API Gateway, what they considered and why they decided particularly for Tyk. The final part of the case studies is devoted to describing what features of Tyk the company uses and in which context. Eventually, the company's future plans for major expansion, migration or adoption of new technologies are also mentioned.

	Branch[1]	VINCI[1]	Lotte	Fortune-500	MiQ	Modulr	areeba	Sun Energia	Scout24	IGA Bahrain	HeiGIT	PA Digital (PA)	Coliquio	QUT	HMCTs	true Corp	Cho Tot	SK Techx
Authentication & Authorization	✓	✓	✓	✓	✓	✓				✓					✓	✓		✓
Central API Management					✓									✓				
Digital transformation							✓			✓	✓	✓	✓		✓		✓	✓
Enhance existing or new product			✓					✓			✓		✓			✓	✓	
Manage external APIs	✓		✓						✓		✓		✓	✓				
Manage internal APIs	✓			✓			✓		✓	✓		✓		✓		✓	✓	
Rate limiting & quotas						✓		✓				✓						✓
Security		✓		✓	✓	✓	✓		✓						✓			

Table 2.2: Use cases of API Gateways used by respective companies.

[1]Implicit categorization of use cases for these companies was missing, so we categorized them manually based on the keywords analysis.

We analyzed these case studies with a goal in mind to get an overall impression of the most popular and most demanded features of API Gateways and, most importantly, the motivations that lead companies to decide to employ API Gateway in the first place. While reading each case study, we focused on identifying keywords related to the following two questions:

Question 1: *What were the problems the company want to address with API Gateway before they began using it?*

Question 2: *What are the features or capabilities of the API Gateway company ends up using after they started using it?*

After we collected keywords for both questions from all case studies, we grouped them based on the frequency of their occurrence throughout texts. The results are summarized in figures 2.1 and 2.2. We can conclude that they clearly highlight some most common motivations and demanded functions that companies expect from API Gateways. The most popular reasons why companies decide to start using API Gateways could be summarized in the following points:

- Centralize and simplify overall API management and control
- Improve performance and user experience
- Enhance and centralize security, authorization & authentication and token management
- Unify logging, collect metrics and observe system status using an advanced dashboard

The most valued attributes and features of API Gateways that are most important for companies when deciding for API Gateway could be summarized in the following points:

- Easy to install and provision, intuitive, seamless integrations with existing company's systems
- Provides central authentication, authorization, token management and access control to APIs, as well as rate limiting and throttling
- Cost-efficient and with a wide range of deployment options
- Provides tools for logging and monitoring, or enables seamless integration with existing logging and monitoring stack
- Open-source is preferable to avoid vendor lock-in; however, rich features and enterprise support are required as well

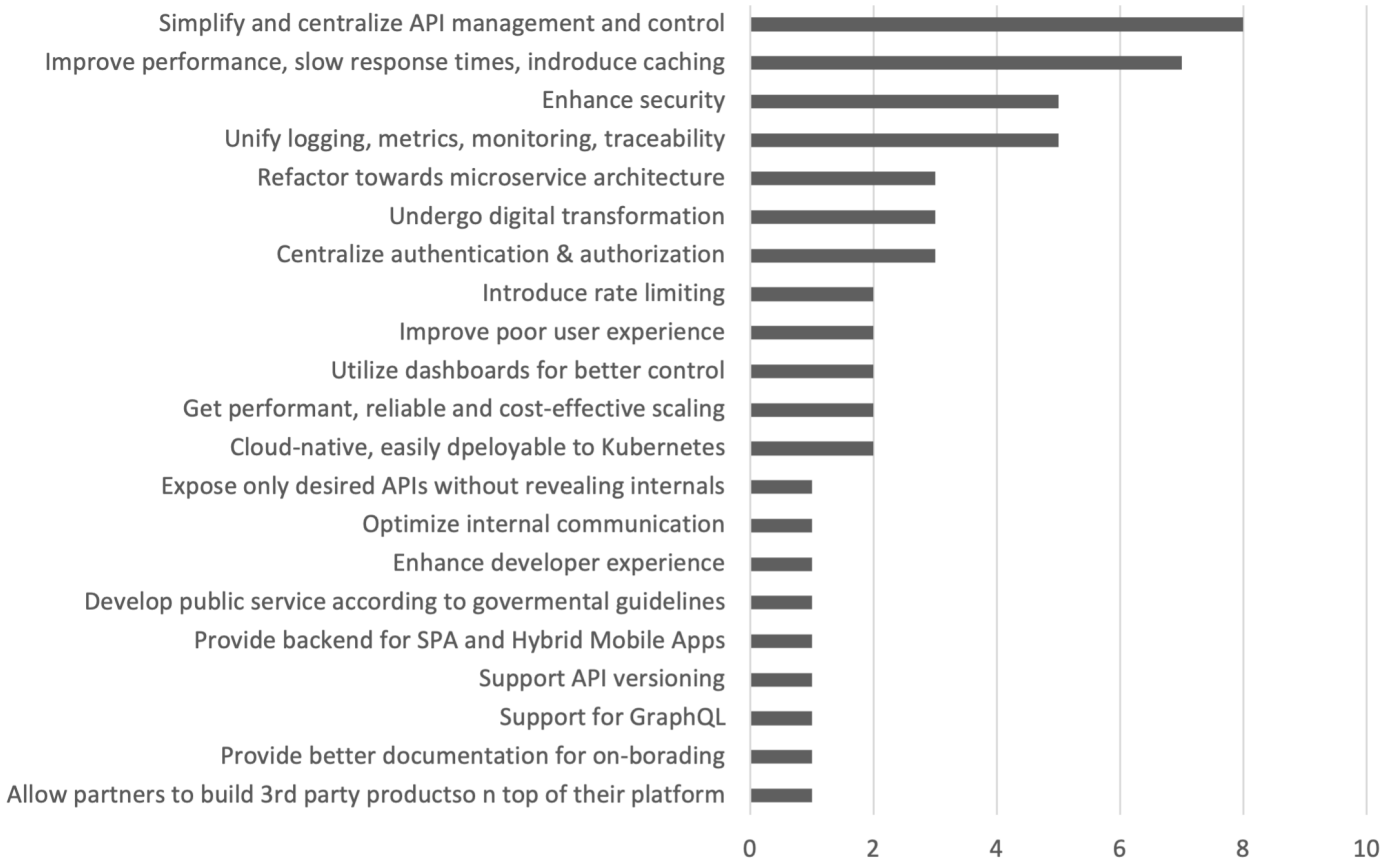

Figure 2.1: *Problems* companies wanted to address with API Gateways sorted by number of mentions in case studies.

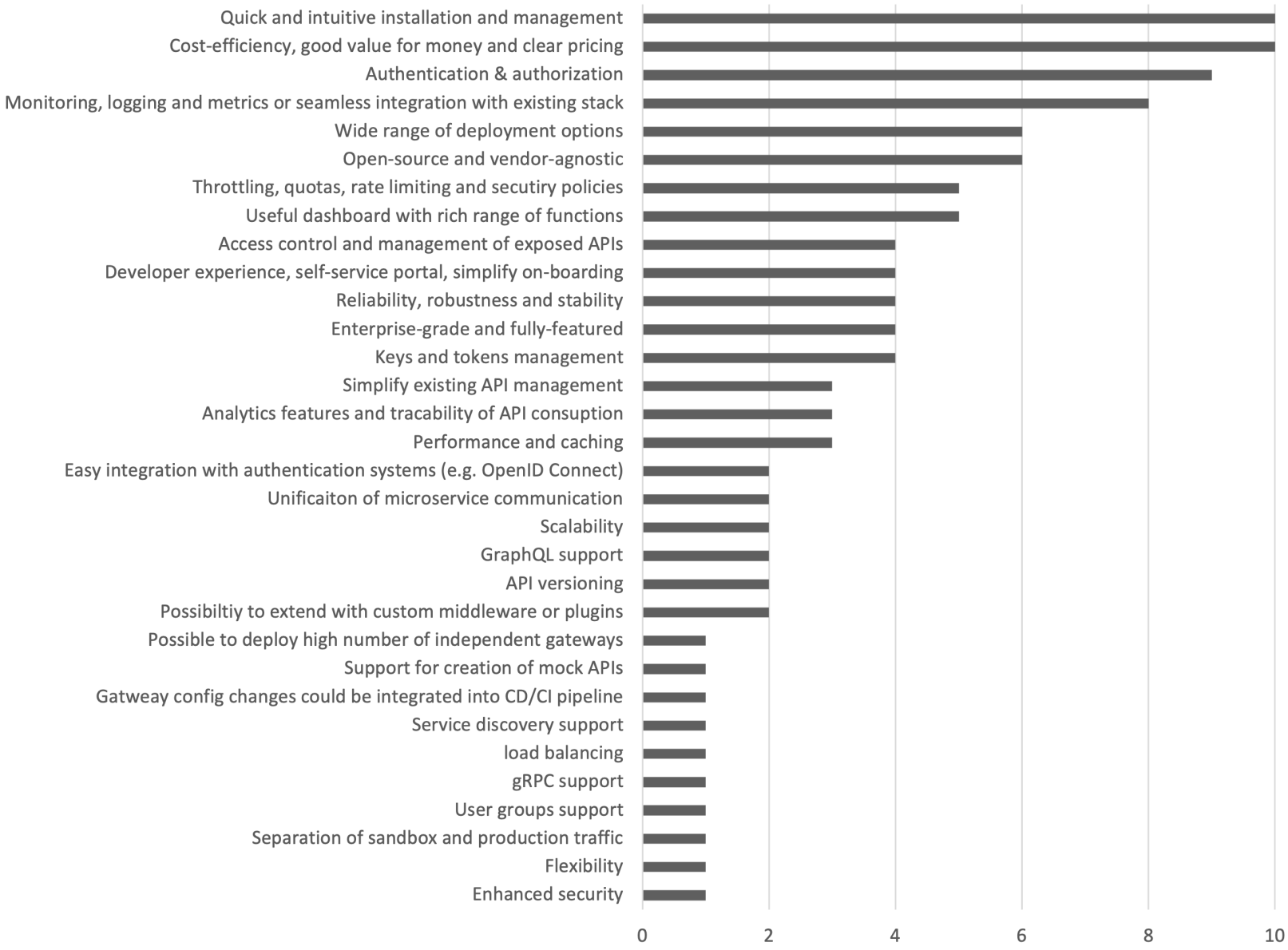

Figure 2.2: *Features* of the API Gateway companies used, sorted by number of mentions in case studies.

2.1.2 Literature

Another important resource for gathering the requirements for API Gateways was series of written articles and blog posts. When it comes to papers published in scientific journals regarding the API Gateways, there is certainly a number of them focusing on some specific technical aspects that API Gateway could provide. However, for collecting requirements about what an API Gateway should and shouldn't do, they showed up not being that useful. Instead, there is a number of blog posts and e-books published either under some technological company brand or independently from industry experts. Although blog posts are usually biased, subjective, and generally, they do not adhere to the high scientific standards, we believe it could still serve as a valuable source of information, especially when is it more of a practical nature. In Table 2.3, we provide a summary of the main resources we used for creating a list of API Gateway requirements.

Type	Author	Title	Description
Expert's Blog Post	Kevin Sookocheff	Overambitious API Gateways	• Reasons why to use an API Gateway • Three main categoreis of API Gateway functions: routing, offloading, aggregation • Avoid: extensive data transformation and aggregation
Opinionated Commercial Periodical	ThoughtWorks Technology Radar	Overambitious API Gateways	ThoutWorks warns about overambitious API Gateway products, as there is lot of risk involved with the dangers of putting too much logic into middleware
Commercial White Paper	Sanjay Gadge and Vijaya Kotwani from GlobalLogic	Microservice Architecture: API Gateway Considerations	• Definition of API Gateway as reverse-proxy in forn of microservices • Features: Security, Service discovery, Orchestration, Data transformation, Monitoring, Load balancing and Scaling
Commercial Blog Post	Albert Lombarte from KrakenD	An API Gateway is not the new Unicorn	• API Gateway role in context of MSA and BFF • Problems API Gateway should be used to solve • Problems API Gateway SHOULDN'T be used to solve • Dangers of "overambitious" API Gateways
Commercial E-Book	Liam Crilly	NGINX PLUS as an API GATEWAY	• Multiple backend servers - MSA • Manual how to set up NGINX to serve as API Gateway • Features: Authentication, Rate limiting, Access control, Validating requests, Health checks

Type	Author	Title	Description
Commercial Blog Post	Microsoft	The API gateway pattern versus the Direct client-to-microservice communication	• Definition of API Gateway • Main features of API Gateway → reverse proxy, routing, requests aggregation, cross-cutting concerns • Benefits of using API Gateway opposed to leaving client's to directly access microservices
Expert's Blog Post	Philip Calcado	Series of blog posts about GraphQL, Microservices and BFF	
Scientific Journal Paper	Fabrizio Montesi and Janine Weber	Circuit Breakers, Discovery, and API Gateways in Microservices	• Circuit breaker pattern • Service discovery • API Gateways
Expert's Blog Post	Chris Richardson	Pattern: API Gateway / Backends for Frontends	• "How do various clients of MSA access different services?" • Differencies of clients in their requirements and capabilities • Compares One-Size-Fits-All API vs. multiple BFF's

Table 2.3: Overview of literature used for creating a list of API Gateway requirements.

2.1.3 Interview

We conducted a single interview with the engineer of the company responsible for implementing an IT solution that is used on the internal hospital's network. The interview had an informal character and relatively short, approximately 30 minutes. The main focus was to find out the reasons the company decided to use API gateway, list the features of the gateway they used, learn a bit about the application itself and the context in which it is used, and last but not least, why they decided for the specific gateway and what other alternatives they considered.

The application is based on a microservice architecture composed of several Docker services which deployment is realized using Docker's Swarm orchestration toolkit. The API exposes a number of REST endpoints that are available only through the institution's internal network.

The primary motivation for conducting the interview was to confront the finding from the literature research on how accurate are they in practice. We did not intend to make interviews a primary material for gathering information but rather a supplementary method to make sure the findings from literature and case studies could be reasonably applicable to real situations.

The main reason why the company praised their decision to use API Gateway were:

- The ability for simple and centralized configuration of authorization, access control, JWT management and rate-limiting
- The simple basic setup, with the possibility to add advanced features later
- Monitoring, especially having an overview of requests that failed with 500 status code and also the response times
- Integration of logging with Greylog
- It is open-source, deployable on-premise without the need to be connected to the outside world

Things they like:

- Possibility to manipulate JSON responses — grouping and aggregation — declaratively, using lambda functions
- Possibility of caching responses

Things they say they need to be aware of:

- Configuration could become messy if there are many endpoints
- Not easy to test that changes in configuration are correct before they go live

Overall, we can conclude that the functionalities of API Gateway this company uses and perceives as most important are in agreement with findings from the case studies and literature research.

2.1.4 Functional requirements

API Gateway Functional Requirements		
1.	**Authentication and authorization**	Priority: **Must**

Unifies authentication and authorization across all microservices and implements access control policies, [60].

Authentication determines who the user is, whereas authorization is about verifying whether the user can do some action or access some resource. When we decide to implement authentication and authorization in API Gateway, it is important to consider whether this is the only way of accessing the data for the outside world. If it is not, then we would have to copy over and keep in synchronization the authorization rules in that other way of accessing data, which can become cumbersome very quickly. Therefore, routing all the external traffic through the API Gateway for authentication and authorization purposes can avoid these complications. To effectively manage authorization, the system should support group policies and token management, preferably using some widely adopted standards like JWT, OAuth 2.0 or OpenID.

2.	**Monitoring and logging**	Priority: **Must**

Provide real-time monitoring of all incoming and outgoing traffic, health checks, unify logging and preferably enable seamless integrations with third-party monitoring and logging stacks [19, p. 10], .

The most critical metrics of ingress and egress traffic the monitoring tools should provide are the statistics of response times, failure rate and resource utilization. There should be an ability to set up thresholds and alerts when thresholds are exceeded. There should be a single place that aggregates logs and errors from all services and allows advanced searching and filtering inside them, as this will make it easier and much faster to diagnose potential issues and bugs as opposed to the situation where developers need to collect logs from numerous locations of different services manually. Many popular third-party solutions exist for logging and monitoring, like Greylog, ELK Stack, New Relic, Sentry, Bugsnag and many more. If the application has already well-established monitoring and logging stack like those mentioned, the difficulty of integrating an API Gateway with this existing stack should be expected to be an essential factor.

3.	**Routing**	Priority: **Must**

Route request to appropriate internal services without exposing them directly to the outside clients.

An important function that API Gateways provides, especially when deployed in front of the microservice architectured application, is an isolation of the external clients from the internal structure of the application's microservice architecture. This prevents clients from creating direct dependencies on the application's services, as this would make any future changes in the service's structure much more difficult to do. Therefore, the service's internal structure should be considered as an implementation detail and not exposed to clients directly, where an API Gateway should serve as a mediator of traffic between the external clients and internal services. To achieve this, API Gateway must have knowledge of how to process each request, more specifically, to which services a specific request should be routed to.

4. Transformation of data, protocols and formats Priority: **Must**

API Gateway is a place to transform data payload formats, headers, and protocols to serve a wide range of different front-end clients and provide them with a consistent API tailored for their needs [19, p. 9].

The clients on the front-end are different and have different needs regarding the ideal way of receiving data. Also, given that the possible number of different microservices managed by the different teams could be high, it is reasonable to expect that some inconsistency in the data formats and communication protocols might arise. If this kind of manipulation is needed, an API Gateway is a suitable place to do it. This could be done either within a single API Gateway instance or by using multiple API Gateways following the Backend-for-Frontend pattern. It is important to note that this data and protocols manipulation should be of a rather general fashion, as it is important to avoid placing any specific business logic on the API Gateway layer.

5. Service discovery Priority: **Should**

API Gateway should provide server-side service discovery if a service discovery mechanism is needed [19, p. 7], [44].

The elasticity of the microservice architecture that seamlessly adapts to the amount of traffic is one of the architecture's strongest points. This aspect, however, means that the services locations cannot be static and are changing dynamically over time. For such a system, a service discovery mechanism is needed. To avoid putting the burden of service discovery on the clients and thus making them more complex, which is not desirable, an API Gateway should utilize Server Side Discovery, as this reduces the number of calls over the internet and allows to have only a single place that implements service discovery logic.

6. Rate limiting and throttling Priority: **Should**

Control limits for the number of requests and eventually requests complexity for each client [60], [19, p. 6], [39].

Each API exposed publicly is at risk that some of its clients would not always have noble intentions and would try to exploit it. Some might try to expose the security vulnerabilities to access data or execute actions they are not supposed to. However, even the best security policies do not prevent a DDoS attack — an attempt to overwhelm service with the massive amount of requests that will stress the servers and infrastructure to the point that it will severely downgrade the quality of service for the legitimate clients, or worse, bring the service down completely. However, it is not only the attackers that could cause excessive stress to the system. It could also be the case that the client contains a bug or is simply not well optimized. The method of guarding against these situations and ensuring the system's stability and robustness even in case of misbehaving clients is to limit how many requests could clients execute and how many resources they are allowed to consume. API Gateway is an excellent place to set limits for the number of requests per time unit clients are allowed to execute, which will prevent any request exceeding these limits from reaching and stressing the internal services.

7. Caching and compression Priority: **Should**

API Gateway should provide server-side and compression to increase the application's performance and reduce response times.

As the rate of the requests on the system increases, even the smallest tweaks and optimizations could add up to significant savings in resource usage, costs and improvements in response times and overall quality of service. Functionalities falling into this category that are suitable to implement in the API Gateway include, for example, response compression, typically using Gzip format, and simple forms of server-side caching, like HTTP caching via `max-age` header.

8. Aggregation and response manipulation Priority: **Should**

When approached carefully, aggregation of the responses and their enhancement could boost performance and customer experience.

Another set of methods that can boost the performance and provide a better developer's experience is to intercept, manipulate, and aggregate the communication between the clients and the target services. Examples of such manipulations include changing HTTP headers, translation to newer and more efficient protocols the service does not natively support, like HTTP/2, HTTP/3, WebSockets, gRPC or QUIC or translation of the request body from XML to JSON. An API Gateway might provide the functionality to define custom API endpoints that will in the background be resolved to a set of multiple calls to different services, and the results will be aggregated and returned to the client at once. This might be helpful as it is not only easier for a client to issue a single request instead of many of them, but it also reduces the amount of traffic sent through the internet, and the response times will be faster. However, any aggregation or request and response manipulation should be proceeded with a lot of caution, as it introduces a risk of putting too much logic into API Gateway, which might make it a bottleneck of the system in the future.

9. Circuit breaker Priority: **Should**

The circuit breaker pattern prevents stressing of the system in case of cascading errors chain [5], [6].

One of the drawbacks of distributed architectures, like microservice architecture, is that it often requires multiple services to communicate to serve a single request. As internal communication requires more overhead and it is slower than in-memory communication, a distributed architecture that is designed too fine could experience significant performance and response times. If some failure in a system happens due to the cascaded timeouts and retires each service implements, this could lead to excessive stress of the system and make the whole application unresponsive from the consumer's perspective. This could be prevented by implementing a Circuit breaker pattern, which we explain more in detail in section 2.3.2.

10. Load balancing Priority: **Could**

In some situations, it might be practical if an API Gateway provides an easy-to-configure load-balancing service.

In the situation where there is only a single instance of the API Gateway deployed in a set of replicated services, API Gateway could also provide load-balancing functionality. It might be simpler and more convenient to use its build-in load-balancing support instead of setting up a dedicated service, especially when the API Gateway is already configured and integrated into the system. However, there could also be deployments where the API Gateway itself needs replication, and in that case, a dedicated load-balancer service would become a necessity.

11.	**Support of GraphQL**	Priority: **Should**

Support for GraphQL might be desired, as it is a prospective technology.

Even though most APIs are still based on REST, GraphQL API is rapidly gaining in popularity. GraphQL introduces a substantially different philosophy of designing APIs compared to REST. It comes with specific needs and functionalities the API Gateway should provide to utilise this emerging technology's potential fully. We write more in detail about GraphQL in section **??**.

12.	**Declarative configuration**	Priority: **Could**

Configuration should be easy to reason about and testable before it goes to production.

The configuration should be manifested as a set of configuration files, typically in YAML or JSON format, that describes the desired state of the API Gateway. Opposed to imperative style configuration, it does not require an administrator to figure out what commands and which order needs to be executed to bring the system from the current state to the desired state, which could often be challenging to assess and introduces the possibility that something will be missed or executed in an inappropriate order. The declarative configuration also makes it easier to integrate automatic validation and testing into the CD/CI pipeline before the changes go live, which further reduces the possibility of errors caused by typos introduced by human mistakes.

13.	**Support for optional plug-ins and middleware**	Priority: **Could**

API Gateway should allow to easily plug in additional, custom functionality if needed.

Sometimes it is necessary to introduce custom processing logic that the API Gateway does not natively support. Examples could include support for a specific protocol or encoding and decoding of the streamed content, like video. API Gateway should provide a mechanism to enhance its functionality by allowing plugging in custom-made code modules. As part of this effort, they should document how to make such plugins on their own and provide an ecosystem supporting the creation and distribution of community-made plugins and middleware.

Table 2.4: API Gateway Functional Requirements

2.2 Evolution of architectures: from Monoliths to Microservices

The simplest and the most obvious style of architecting the software application is a lack of it — no architecture at all. The history of writing computer programs starts with punched cards, goes through assembly code,

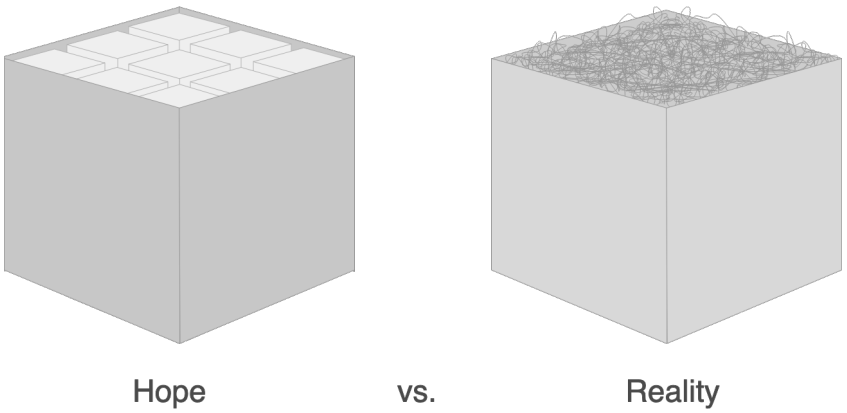

Figure 2.3: The internal structure of a typical monolithic application. Source: `https://martinfowler.com/articles/dont-start-monolith.html`

series of programming languages with the necessary use of GO-TO statements. Through the decades, the languages evolved into more abstract and structured enabling writing and managing large code bases more easily.

2.2.1 Monolithic Architecture

For a long time, distributed computing was, to a large degree, a domain of scientific applications. Typical businesses were typically running all their software on a few on-premise stations, i.e. on dedicated hardware machines placed in a server room or a data centre. If the application started consuming more resources than the on-premise machine could manage, the usual solution was to upgrade the hardware components or buy a stronger machine. In the circumstances like this, there was only little motivation to think about designing application in a distributed manner. Therefore, the applications were written as single units that needed to be entirely re-compiled whenever a single small change was made into their code. It was no exception that large applications grown into outrageous size, making it very difficult to maintain and improve. Because this application must always be operated with as a single unit and typically, it is impossible to re-use or reparate any subpart of the application because of the tangled application-wide dependencies, they are called monoliths and this way of designing the applications is referred to as monolithic architectural style.

Monolithic applications start to show problems as the application size grows, and there is a need for contribution from several development teams simultaneously. Similarly, as it is not possible to manage a large corporation without distinct departments and a hierarchy of organizational structures, it is impossible to maintain and develop an extensive application without splitting it into smaller modules with well-defined responsibilities.

But what does it mean precisely to split into smaller pieces and define responsibilities well? In object-oriented programming, there is exist a widely accepted set of design principles known under the acronym SOLID. The "S" in SOLID means the Single-responsibility principle, and it states that each class should have only one responsibility and only one reason to change. Is simply adhering to this principle sufficient enough? While it is theoretically possible to create a well-structured monolithic application, the reality is, unfortunately, very often different. The rules are easy to break and shortcuts too tempting to take.

The typical situation regarding monolithic applications is that they are extremely difficult to break down into smaller pieces, as the internal components have a massive amount of dependencies tangled be-

tween each other. The internal components often lack a clear definition of responsibilities causing unexpected side-effects all around the application, and even if they declare some contracts or conventions, there are often found to be broken as they are not strictly enforced.

Large server applications also typically need to handle heavy traffic that requires a substantial amount of computational resources. Often, the periods of heavy peak traffic endure much shorter than the periods of relatively low traffic. If the computational resources cannot be re-assigned dynamically in accordance with these peak periods, the system must provide enough resources to handle the peak traffic, which will cause inefficient use of the resources as in the low periods there would be unutilized. When the application is able to dynamically allocate and de-allocate the resources based on the processed workload, we say it is scalable. The application could be scalable in two ways: vertically or horizontally. Vertical scalability is the ability to reallocate the computational resources from the perspective of a single machine, e.g. CPU power, memory or storage. However, this approach has its clear limitations given by the power of available hardware components. On the other hand, horizontal scalability is the application's ability to spawn and shut down the copies of its own that run simultaneously on multiple servers and divide the workload between each other. Horizontal scaling does not have the strict hardware limits of vertical scaling. However, it demands that the application design is ready for it, as it introduces a new range of challenges like ensuring data consistency on distributed storage or dealing with unreliable network communication.

We mention scalability in the context of monolithic architecture because, without a doubt, it is a desired property of software applications. The problem is that especially horizontal scalability is difficult to achieve using monolithic architecture. To do that, we need to take a different approach — a distributed architectural style.

In distributed architectural style, the individual components of the application are self-contained entities that allow being developed, tested and deployed separately. The mutual communication of the components is provided using remote-access protocols — for example, Representational State Transfer (REST), Advanced Message Queuing Protocol (AMQP) or Simple Object Access Protocol (SOAP). This approach where the individual components are independent, almost to a degree as if they would be separate applications, has the consequence that it promotes and improves several important non-functional attributes of the application like better scalability, modularity, easier maintenance, and loose coupling [56, p. 1]. Components are loosely coupled when they are designed in a way, so they have the least possible amount of mutual dependencies. Compared to the monolithic architecture, where the components share the same programming libraries and often the same process, the barrier to introducing programming calls all across the application is low, which can easily lead to a tangled mesh of dependencies — a tight coupling. On the other hand, introducing dependencies between independent and self-contained components requires much more work, which naturally nudges the design of the components in a desired loosely coupled way. Unfortunately, the benefits of the distributed architecture do not come free of charge, and the trade-offs associated with the advantages are increased complexity, cost, time to develop or dealing with problems connected to unresponsive, unavailable services or issues with the network connection [56, p. 2].

2.2.2 Service-Oriented Architecture

OASIS [**OASIS**] reference model for Service Oriented Architecture [56] defines service as a mechanism to enable one or more capabilities with a well-defined interface and contract. The service can either implement business capabilities as well as non-business capabilities, like logging, auditing, security or monitoring. Based on this classification, it defines two types of services:

1. Functional services
2. Infrastructure services

In SOA, there is a standard formal taxonomy that defines into four basic types, which differ in terms of the granularity and abstraction level:

1. Business services
2. Messaging middleware
3. Enterprise services
4. Application services and infrastructure services

The *Business services* are abstract, high-level and coarse-grained services that define core business operations. On the other hand, the *Enterprise services* implement the concrete functionality defined by the business services. The *middleware component* bridges the previously mentioned types of services.

Application services are bound to the specific application context and implement specific business functionality that is not captured in the higher-level services. The final type, **infrastructure services**, implement non-business functionality as security, monitoring or logging and can be called from any higher-level services. Finally, the *middleware* is used to facilitate all communication between these services.

It is not a strict requirement to follow this standard scheme, and it is possible to define your own that better fits the specific application requirements. What is important is to well-define and document this taxonomy of the services so they have clearly defined responsibilities. This is needed as the different types of services usually have different owners. The business services are owned by the business users, whereas enterprise services are owned by shared service teams and architects. The distinct application development teams own the application services, and finally, the middleware is owned by the integration team.

The Service-Oriented architecture is not addressing the impact of the granularity of the services in the design. Therefore their size could range from very small and fine-grained to large enterprise services. This is because there is no universal answer, and the optimal service granularity varies from application to application, and it is a challenging tasks for architects to get it right.

The communication and coordination between the services are facilitated by the messaging middleware layer. It provides features including mediation, routing, message enhancement and transformation. While it is not a strict requirement, SOA usually uses messaging protocols like JMS, AMQP, MSMQ and SOAP as remote-access protocol.

When it comes to component sharing, the concept of Service-Oriented Architecture is to "Share-as-much-as-possible". In order to avoid replicating processing logic in different contexts, SOA solves this by creating a single enterprise service that is used in all different contexts. However, different contexts often need to use a different representation of the same data, and they might store that in separate databases. Because enterprise service is shared, it must be smart enough to know to accompany all the context it is used from, access the right databases and propagate updates to the different representations. Using the enterprise services achieves the goal of reducing duplication but imposes a risk of tight coupling with too many distinct parts of the application, which makes it difficult to change.

Overall, SOA fits best the large enterprise applications with a high amount of shared components. The smaller projects or projects that could be well partitioned will not be able to benefit from capabilities offered by SOA, and the drawbacks arising from the complexity and need for coordination will likely outweigh them.

2.2.3 Microservices Architecture

Microservice architectural style is an approach for developing a single application as a suite of small services, where each of them runs in a separate process and communicates using lightweight remote access mechanisms [41].

Microservices are independent components encapsulating a business or non-business functionality and are independently replaceable, upgradable, deployable and scalable. Compared to the libraries that are linked to the program and use in-memory calls, services communicate using some remote access protocols, like HTTP or RPC. This also makes it easier to achieve the level of encapsulation between the components, as the remote calls require clearly defined contracts and API to make them possible.

Microservice architecture also has an impact on the organizational structure. While in traditional enterprise applications, there are usually siloed teams of, e.g. UI specialists, middleware specialists or database admins, microservices allow to create smaller, cross-functional teams around the specific business capabilities. Because single business functionality is usually encapsulated within one service and one team, only minimal cooperation between teems is needed, and new business functionality could be implemented and deployed cheaper and faster than in SOA [56, p. 16].

Share-as-little-as-possible is a concept favoured in MSA, which is in sheer contrast to the share-as-little-as-possible concept of SOA [56, p. 22]. This means that achieving loose coupling is prioritized before creating a service on which all other services depend, and therefore it is a single point of failure and difficult to change at the same time.

"Smart endpoints, dump pipelines" [17] is a concept discouraging any complicated logic in the communication channels. Microservices typically use simple REST or RPC calls without any extra functions offered by the middleware in SOA like message enhancement, protocol translation, authentication and authorization. This makes it more difficult to implement security [56, p. 7], but again reduces the amount of cooperation needed.

Because microservices typically operate in a dynamic cloud environment, the lifespan of their instances could be pretty short - easily only a few hours. As the services independently scale up, down, are re-deployed with upgrades, it could easily happen that some services became unresponsive or even unavailable. The services must therefore be designed in a way that tolerates such failures and responds to the client as gracefully as possible [41].

Even after nearly a decade, the microservices architecture is still growing in popularity, along with the trend where more and more businesses are moving their IT to the cloud environment, and the need for cloud-native applications is rising.

2.3 Design principles of microservice applications

2.3.1 The Twelve-Factor App

The Twelve-Factor App [63] is a methodology for building modern software-as-a-service apps created by Adam Wiggins in 2011 and updated last time in the year 2017. The methodology identifies twelve essential aspects, or as they name it — factors — of the SaaS applications and specifies requirements and responsibilities that, if followed, should lead to a well-designed application. Chris Stetson further builds upon this methodology in The Nginx's Microservice Reference Architecture [43] and adapts it to the context of Microservice

Applications. In this section, we list and explain these Twelve Factors, as they do an excellent job in pinpointing the areas of the microservice architecture design that need to be addressed with extra care.

Factor 1: Codebase

> "*There is a one-to-one relationship between codebase and app, where the app could have many deployments.*"

Each service source code should be captured using a single repository in a version control system (Git, Subversion). In contrast to having all services in a single repository, this approach supports better isolation and independent development cycles of the services.

Factor 2: Dependencies

> "*Dependencies must be explicit, declared and isolated.*"

As the services could run on heterogenic execution environments, they should never assume that certain tools and libraries in specific versions are available and ready to use. All the software the services depend on should always be explicitly declared using either language-specific package managers like npm in Node.js, pip in Python or Dockerfile when using containers.

Factor 3: Configuration

> "*Configuration must be strictly separated from code, stored in the environment.*"

Any information that could change in runtime or could be different between the deployment and execution environments should not be stored in a source control system, but in environment variables managed by the execution environment. A good test of whether the source code complies with this rule is asking a question: "Can we make the source code public right now without compromising the security?" The Twelve-Factor methodology encourages the use of environment variables over configuration files and advocates against grouping and naming the configurations like "development configuration", "staging configuration", and "production configuration". Such an approach creates a need to track a vast amount of similar but slightly different configurations as the number of deployments grows. Environmental values should be stored in a secure space with a strictly controlled access policy, e.g. using Valut [40].

Factor 4: Backing Services

> "*Services treat each other as network-attached resources, where the attachment and detachment are controlled by the execution environment.*"

The app consumes the services over the network by connecting to the exposed ports or using their APIs. The services could be either controlled by the development team (e.g. PostgreSQL, Redis, RabbitMQ, SMTP, Memcached) or by a 3rd parties (e.g. Bugsnag, Sentry, New Relic, MailGun). In the microservices architecture,

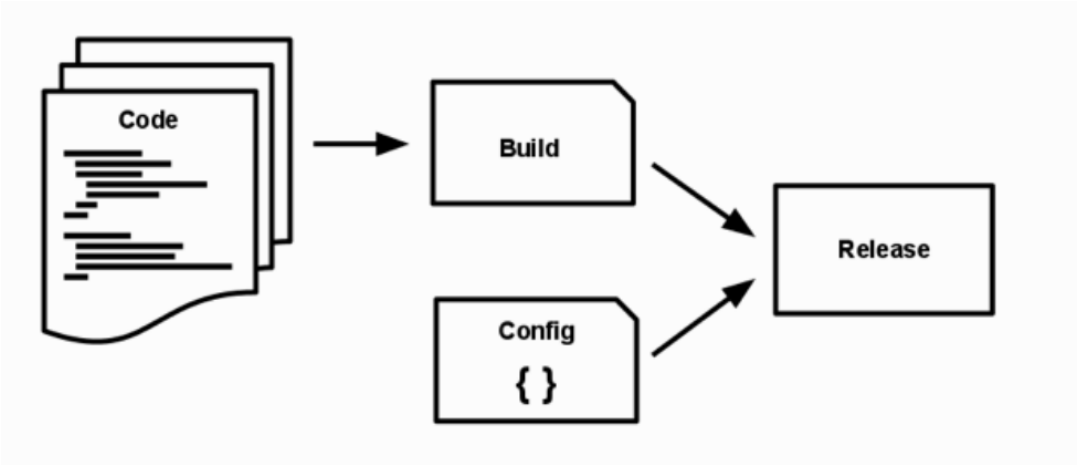

Figure 2.4: Build and release phases. Source: [63]

every service is treated as an attached resource. This encourages loose coupling and makes the development of the services more effortless, as the developers of one service do not need to change other services.

Factor 5: Build, Release, Run

> "*Build, Release and Run stages must be strictly separated.*"

The Twelve-Factor methodology requires strict separation of the build, release and run stages of the deployment:

1. First, in the build stage, the code is linked with its dependencies and assembled into executables.
2. Then, in the release stage, the executables are combined with the specific environment configuration and prepared to be executed.
3. Finally, in the run stage, the configured executables are started as the respective app processes.

Factor 6: Processes

> "*Application consists out of one or more stateless processes, where all persistent data are stored on a backing service.*"

In general, services should be designed to be resilient to the sudden termination of their instances. This means that they need to be stateless, and any persistent data should be stored in a separate stateful service. The service's filesystem and memory should be used only as a temporary, single-transaction cache, and it should never be assumed that anything that is stored there should be available in the future. The "Sticky sessions", which is a mechanism that ensures that the same instance of the service will process the future requests from the same client, are essentially a violation of the Twelve-Factor principle and should be avoided.

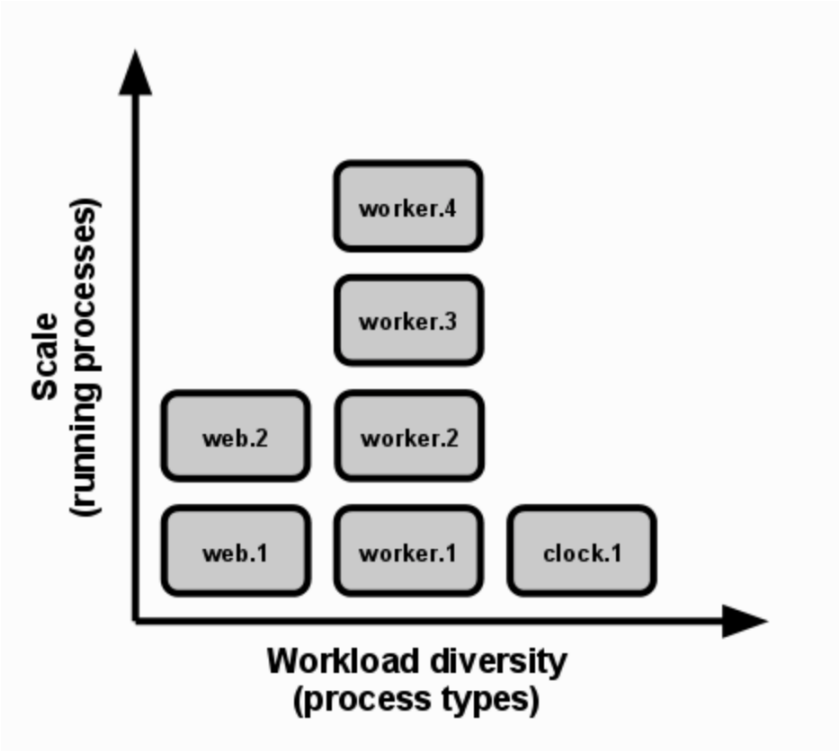

Figure 2.5: Workers specialized for different types of workload on scale. Source: [63]

Factor 7: Data Isolation and Port binding

"*Services are self-contained and communicate strictly only through their APIs.*"

Services in a microservice architecture should be self-contained. This means that they should not rely on any specific services or software to be available in the execution environment during runtime. The services should export a port to which other services can connect and use the API layer to communicate. It is highly recommended that the persistent data owned by the service would be available to other services only through the owing service API. For instance, if the service uses PostgreSQL as a backing service for persistent data storage, the direct to this backing service should be allowed only for the owning service. The other services should be prohibited from connecting to the database directly and execute custom SQL calls. This approach prevents the creation of implicit contracts between microservices and ensures that they preserve loosely coupled.

Factor 8: Concurrency

"*Different workloads are processed by different process types that can scale independently.*"

Every running computer program is represented by one or more processes. The program should be architected so that each different type of workload is handled by a different type of process. For example, HTTP requests should be handled by the webserver process, where long-running tasks should be offloaded to the separate background worker processes. This way, only the necessary processes could be scaled up if the amount of a particular type of workload increases, leaving the other parts of the app intact and responsive.

Factor 9: Disposability

> "*Services should be designed to be short-lived, thus featuring a quick startup and shutdown times and resiliency against sudden terminations to prevent any data loss.*"

One of the huge advantages of a cloud environment is that the computational resources can be assigned and released dynamically and automatically based on the current workload. This, however, means that the resources are not allocated permanently, and the services running on them could be terminated at any time when they are not needed any longer. Therefore, services should be designed for fast startup and shutdown and should be robust against "sudden kills", i.e. unexpected non-graceful termination and prevent any data loss even in these cases. The short lifetime of the services — hours or even only minutes — must not be a problem. If the service gets terminated in the middle of the operation, the queueing system that assigns tasks to the workers must ensure that the processed task would be required. To avoid any consistency issues using this approach, the operations must be either: a) atomic or wrapped in a transaction — means that unless the operation is completed as a whole, it has no effect at all, or b) idempotent — the fact that the operation is repeated multiple times does not change the result.

Factor 10: Dev/Prod parity

> "*Reducing time, personnel and tools gap between Dev and Prod environment as much as possible.*"

The gaps between the Development and Production environment are 3-fold:

Time gap. The time between the moment when the code is created on the development environment and when it is published to production could, historically, span from several days to even a few months. Nowadays, modern workflows try to reduce this time to just hours or even minutes. This allows a more agile approach where the businesses could roll out the updates and evaluate their impact much faster than it was previously possible. To achieve a high velocity while still maintaining sufficient code quality, a large number of quality checks and processes should be automated and integrated into the CD/CI pipeline.

Personnel gap. Traditionally, the moment when the developer would make a change to code would be only the beginning of the long process he would not be a part of. The code would undergo quality checks by quality assurance personnel, and later the operations team would deploy it. If any issue occurs during this process, for example, it would be discovered that the change has a negative impact on the performance or is not compatible with another change would restart this process from the beginning. To speed up this process, the developers should cooperate with the operations and be closely involved in the deployment process. To achieve such a high degree of cooperation between different roles, the traditionally siloed teams has to be split into smaller, cross-functional units dedicated to managing a specific part of the application throughout its whole life-cycle.

Tools gap. The development usually occurs on the developer's local machines or servers that are far less powerful than the production machines. There is a temptation for developers to use more lightweight alternatives to the ones in production, such as using SQLite instead of PostgreSQL. Also, there could be substantial differences caused by the different operating systems the developers use (macOS or Windows) to the one that

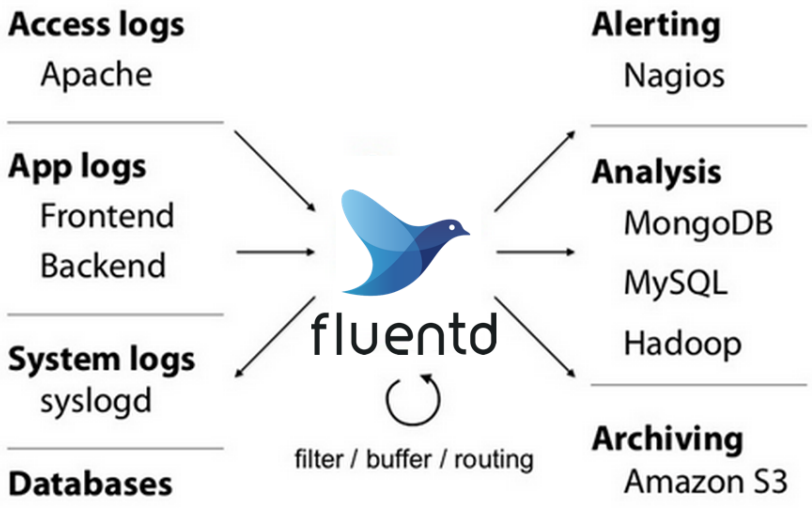

Figure 2.6: Unification and aggregation of logs by the service provided by environmental service Fluentd. Source: `https://github.com/fluent/fluentd`

is used in production — Linux. All of these, combined with the possibility of different versions of the installed software, can cause a risk of some appearing only in a single of the environments, making them difficult to debug. This risk could be minimized by making the Development and Production environments as similar as possible, which is achieved to a very high degree, for instance, when the application runs in containers.

Factor 11: Logs

> "*Process logs from all services as a stream of events using an external tool provided by the environment.*"

Logs should be treated as a stream of aggregated, time-ordered events. A good logging mechanism should provide these three functions:

1. Ability to find a specific event in a past
2. Large-scale graphing of trends in the system as a whole
3. Active alerting according to heuristics and thresholds

The potentially high number of microservices can create a considerable amount of diverse logging information. Letting each service manage the logging on its own will cause the logging logic to be re-implemented in each service and make it difficult to obtain a general overview of the system state. Instead, all log information should be fed into a central logging system that unifies the logging layer, for example, Fluentd. This approach leaves the microservices free of logging logic, so they could ideally just write their logging information to their standard output.

Factor 12: Admin and management processes

> "*Admin and management tasks should be in source control and packaged within the application.*"

Sometimes, developers or administrators would like to run a one-off or periodically repeated administrative tasks on the servers, like running database migrations, one-time scripts or triggering manually tasks that are usually triggered automatically. These operations should be run in the same environment as the regular long-running processes of the app, ideally implemented as commands of the application CLI and versioned in source control.

2.3.2 Circuit Breaker pattern

Microservices use remote network calls to communicate with each other. Contrary to the in-memory calls in monolithic applications, the networks calls could fail or hang unit time-out limit is reached. The network intercommunication between the microservices could be very busy, and when some connection channel runs into problems, it could cause cascading failures across multiple systems.

As the clients find out that the calls are unresponsive, they could repeat them, putting yet another stress on the already unresponsive system and causes consumption of an unnecessarily excessive amount of computational resources that could lead to a complete shutdown of the service. The Circuit Breaker pattern [16, 50, 44] is designed to prevent such catastrophic scenarios. The goal of the Circuit Breaker pattern is, similarly to the circuit breaker in electrical engineering, to minimize the damage when things go eventually wrong.

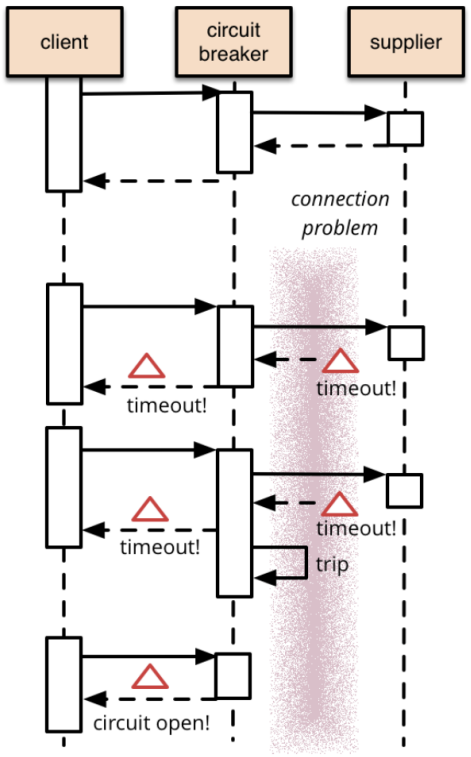

Figure 2.7: The Circuit Breaker in action. *Source:* [16]

The basic idea of the Circuit Breaker pattern is to wrap a potentially erroneous call into a protected block that will monitor the failures inside it. When the calls operate normally, the Circuit Breaker keeps itself in a `closed` state. The rate or number of failures of the protected call are recorded. When they exceed some given threshold, the circuit breakers switches to an `open` state, where it skips the execution of the

protected call entirely and instead serves the error message to the clients straight away. This reduces the stress to the underlying unhealthy service and gives it an opportunity to recover. To make Circuit Breaker capable of resetting itself back to the `closed` when the underlying call starts working again without the need for any external intervention, it must implement some self-resetting mechanism. Such a mechanism could be that after a reasonable interval, it switches to a `half-open` state, in which a real call is executed and if successful, it resets back to the `closed` state. If not, the `open` state is prolonged for another period.

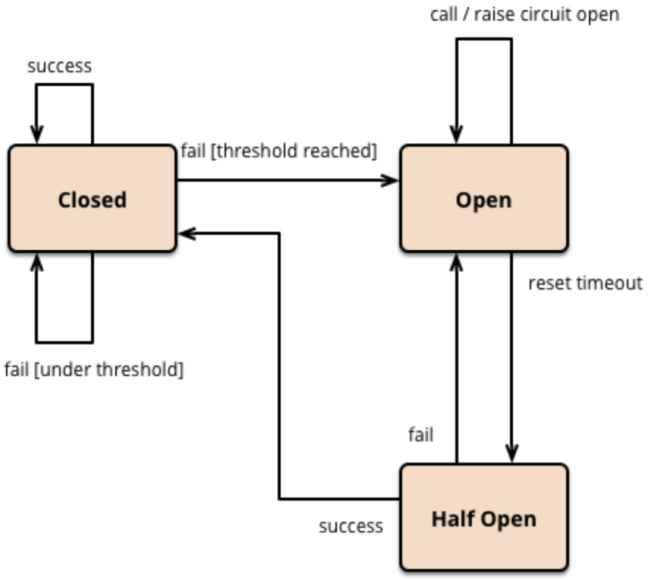

Figure 2.8: States of the The Circuit Breaker. *Source:* [16]

2.4 APIs in microservice architecture

In this section, we will describe the REST architecture, a Backend-for-Frontend pattern and briefly introduce GraphQL.

2.4.1 Representational State Transfer (REST)

REpresentational State Transfer (REST) was introduced in Fielding's dissertation [15] from year 2000 as an architectural style for distributed hypermedia systems. REST is based on for basic principles [52]:

1. Resource identification through URIs
2. Uniform interface for all resources
3. Self-Descriptive message representations
4. Hyperlinks to define relationships between resources and valid state transitions

In this section, we will describe the basic principles of the REST, subsequently, how the APIs based on this architectural style look like nowadays.

2.4.1.1 Resources, representations, URIs and URLs

In REST, a resource is an abstraction of information. That could be anything, for example, a virtual object like a document or image, a non-virtual object like a person or a temporal service like today's weather in Los Angeles [15]. Resources are the targets of the HTTP requests and could either be static or change and evolve over time.

Resource representation is used for purposes of HTTP to reflect past, current or desired states of the resource, consisting of both representation metadata and data. A single resource could have multiple representations.

Each resource is addressed by its unique identifier called URI, *Uniform Resource Identifier*. REST leaves it up to the author to choose the identifier that is best suited for a particular resource. If it is a book, a suitable identifier could be, for example, ISBN. If it is a product in an e-shop, URI could be, for instance, it's GTIN, EAN or barcode number.

A URL, *Uniform Resource Locator*, is a special form of URI that not only uniquely identifies the resource, but also provides infromation how to access it. Examples of URL's, taken from RFC 3986 [3], are:

- `ftp://ftp.is.co.za/rfc/rfc1808.txt`
- `http://www.ietf.org/rfc/rfc2396.txt`
- `ldap://[2001:db8::7]/c=GB?objectClass?one`
- `tel:+1-816-555-1212`

In the context of REST APIs, to allow clients to interact with resources effectively, it is required to provide them with the necessary information on how to access them. For this reason, it is more accurate to refer to the REST resource identifiers as URLs rather than just URIs.

As shown on figure 2.9, an URL typically consists of these parts:

- *Scheme*: A communication protocol, nowadays almost exclusively `https`
- *Domain*: Consists form the first, second and sometimes also from the third-level domain names
- *Path*: Identifies the resource or collection. Often, all API paths contain a common prefix containing an API version, like `/api/v1`
- *Query parameters*: Starts with the presence of `?` symbol. Used to specify how the response should look like e.g. for filtering, searching or pagination

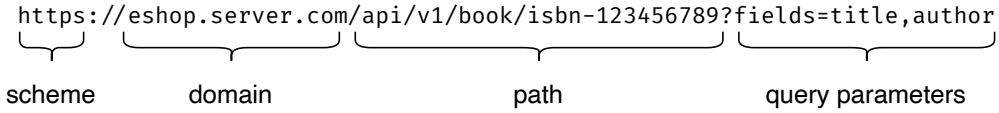

Figure 2.9: Description of the URL parts of a typical URL used in REST API's to access a resource.

Additionally, URL could also contain a "fragment" specified after `#` symbol following the query parameters part. This is typically used in web pages to point to a specific anchor in a longer text so that the

browser will automatically "scroll down" to that desired section when the page loaded. In APIs, fragments are rarely used because there is no practical advantage of using them over adding an additional parameter to the query.

2.4.1.2 HTTP methods

REST APIs offer a uniform interface of manipulating with the resources through the collection of HTTP methods, introduced initially in RFC 2616 [49] that was later replaced by a set of RFCs 7230-7237, where the methods are defined in RFC 7231 [14].

GET It is used to retain the information from the server but without causing any significant side effects or change the server's state. While it is OK for the server to, for example, increment the visits counter or logging the GET request, it should never cause any loss of data. This is the reason why we say the GET method is "safe". Being safe means that the client could experiment and explore the API by sending GET requests without prior knowledge of the consequences - the GET requests should guarantee that nothing bad, like unexpected data loss, will happen. In reality, there is no guarantee that GET requests behave this way. For example, legacy SOAP APIs typically don't rely on the HTTP methods at all, and it is not uncommon for SOAP APIs to allow GET requests to modify er even destroy the data. However, REST APIs should and are expected to respect the semantics of HTTP methods properly. The GET request could request a single item, the whole collection of items, or its subset by issuing a range or search request. A successful response to this request should have status code 200 (OK). GET requests are the most common types of requests, and the responses could be cacheable.

POST This method is used to create new content that is not yet identified by the server. It could be creating a new order on a webshop by submitting a web form, creating a new blog post or a new comment to it, or appending a new item to the list of products, etc. A server should send status code 201 (Created) when the processing of POST request was successful.

PUT Replaces a current resource representation or creates a new one. If the resource representation exists, it should be updated, and the successful response must be denoted with status code 200 (OK) or 204 (No Content), like if it would be in the case of a PATCH request. If it does not exist, a new representation should be created based on the supplied payload, and the request must return status code 201 (Created), as it would be in the case of the POST request.

DELETE Removes all current representations of the resource.

PATCH This method is widely popular in APIs, however, is not defined in RFC 7231[14], but in a separate RFC 5789 [11]. At first sight, it might look very similar to the PUT method as it is used for a similar purpose, but it is important to know the differences between them to avoid confusion. Both PUT and PATCH are used to update or create a new resource representation. However, the strategies they use to achieve it are different. A payload of the PUT request must always contain a complete version of the desired resource representation, and in case it exists, the existing version would completely be replaced by the new one. On the contrary, the payload of PATCH is different. It contains information on modifying the existing resource representation; we can say it as a form of "diff". Therefore, PATCH is no longer idempotent like the PUT requests, but it can be a

better choice for concurrent requests. Because PUT and PATCH could be often confused, it is a good practice to mark partial payloads with the `Content-Range` header and configure the server to refuse all PUT requests containing this header. This would prevent situations where incomplete representations would be mistakenly treated as complete ones and stored on the server.

An example of what can go wrong with the PUT request and the way how the PATCH version would fix it on figure 2.10. The example shows a situation where Client A would like to add 10 € to the account balance, while at the same time, Client B would like to withdraw 5 €. If clients use PUT, they would first need to fetch the current account balance using GET the request. However, because the clients have no way of knowing about each other, they will not realize that the actual account balance changes in the meantime before they send the PUT request, leading to an incorrect account balance in the end. A better way would be to allow clients to use a single PATCH request instructing the server how to modify the balance without any prior GET request. The root cause of this problem that GET and subsequent PUT request are not together treated as an single atomic operation, whereas PATCH, as a single request, is.

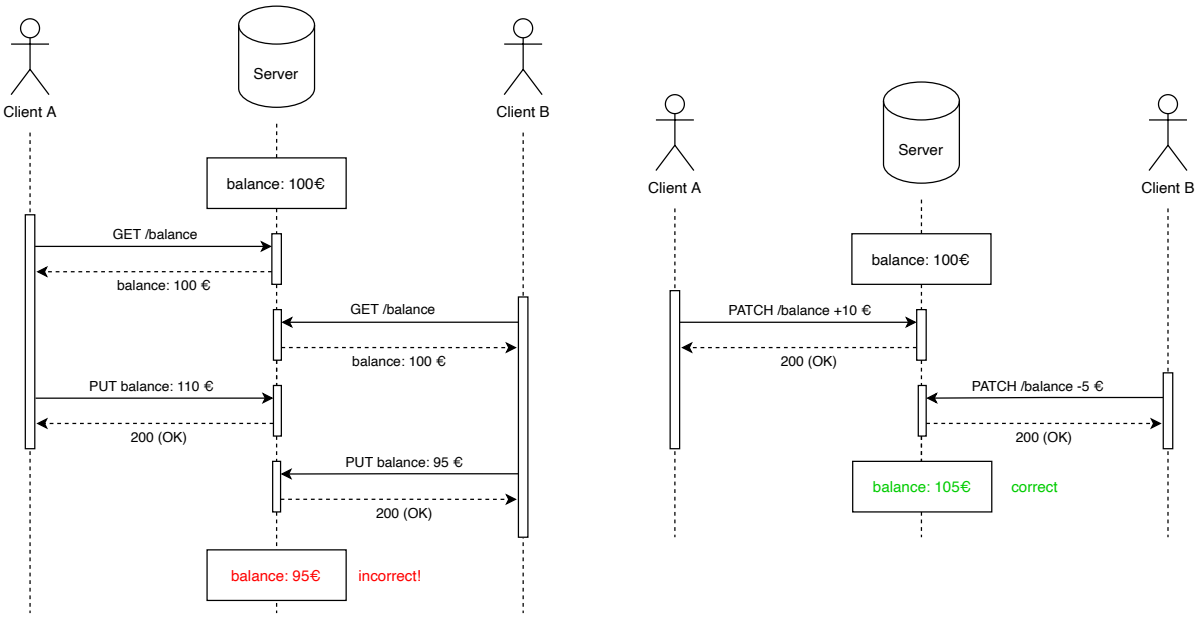

Figure 2.10: GET and PUT vs. PATCH. An example of how usage atomicity of the operation could be easily achieved using a PATCH method (on the right), contrary to the combination of GET and PUT (on the left).

Method's safety Safe methods are the ones that are not destructive. Those are the ones that only retain or read information but do not change them. Out of all HTTP methods, it is only the GET method that is fulfilling this criterion.

Method's idempotency A property that guarantees that the repeated call of the same method with the same data will not affect the result is called idempotency. This property is convenient in environments of unstable network connection, where the requests or responses could randomly be not delivered. When a client does not receive a response from the server after it sends the request, this could be because of two reasons: either a) request did not arrive at the server, or b) response from the server did not arrive at the client. In case of situation a), a re-transmission of request is necessary to achieve the desired result. However, in case of b), no re-transmission is necessary as the data are already on the server. The problem is that a client does not have a way to know which of these two scenarios actually happened when it did not receive a response and thus

cannot unequivocally decide whether to re-transmit the request or not. If the client decides incorrectly and re-transmits in situation b), the server will process the request a second time that will result in data duplication; and if it does not re-transmit in situation a), the server will contain no data at all. The solution to this dilemma is to make the Method idempotent — so that it guarantees that even in case of re-transmission in situation b), the final state of the data stored on the server stays the same. Idempotent methods, therefore, allow the clients to safely re-transmit any requests every time they did not receive a proper response from the server, keeping the data consistent and making the system robust to the occasional network outages.

HTTP method	**Safe**	**Idempotent**
GET	✓	✓
POST	✗	✗
PUT	✗	✓
DELETE	✗	✓
PUSH	✗	✗

Table 2.5: Summary of the properties of safety and idempotency of the most common HTTP methods.

2.4.1.3 HTTP Status codes

In a previous section dedicated to describing HTTP methods, we already mentioned what status codes are expected as the response to certain types of requests. While the methods provide a uniform interface for requests sent from client to server, the status codes provide a uniform interface for responses sent from server to client.

Below is a list of the most common standard HTTP status codes, organized into four groups: Success, Redirection, Client error and Server error.

Success

- *200 OK*: Standard response for successful HTTP requests.
- *201 Created*: Request was fulfilled, and a new resource was created as a result.
- *204 No Content*: Request was successfully processed, and no content is returned as the response.

Redirection

- *301 Moved Permanently*: Moved to a new location. All future requests should be directed to the new URL.
- *303 See Other*: A response to the request could be found by issuing an additional GET request to the given URL.
- *304 Not Modified*: A resource was not modified since the client downloaded it the last time (denoted by headers `If-Modified-Since` or `If-None-Match`), so there is no need to re-transmit it as the client already have the latest version.
- *307 Temporary Redirect*: Request should be repeated using another URL. However, future requests should still use the original URL.

Client error

- *400 Bad Request*: Malformed request. Could mean, e.g. invalid body content caused by syntax error.
- *401 Unauthorized*: Authentication of the client was not provided.
- *403 Forbidden*: Client does not have access rights to the resource.
- *404 Not found*: Resource was currently not found on the server but might become available in the future.
- *418 I'm a teapot*: Server refuses to brew a coffee because it is a teapot. This could lead to a serious problem. As we all know, the unavailability of freshly brewed coffee usually has a negative impact on the productivity of the entire team of software engineers. A reference to an April Fool's joke that defined Hyper Text Coffee Pot Control. Protocol [42].
- *422 Unprocessable Entity*: Request was syntactically correct but cannot be processed because of the semantic errors.
- *429 Too Many Requests*: Indicates that the rate limit for the current user was exceeded server is refusing to process any further requests at this moment.

Server error

- *500 Internal Server Error*: Unspecified server error.
- *502 Bad Gateway*: Server that is acting as a gateway or proxy received an invalid response form the upstream server.
- *503 Service Unavailable*: Server is currently not ready to handle the request, usually down for maintenance or overloaded.
- *504 Gateway Timeout*: Server that is acting as a gateway or proxy did not received any response form the upstream server.

2.4.1.4 JavaScript Object Notation (JSON)

JSON (JavaScript Object Notation) is a lightweight data-interchange format [4] is the most popular format used by modern REST APIs. As shown in listing 1, each JSON is an object build from the following structures:

- *value*: could be number (integer `42`, number with a fraction `4.2`, or even with exponent `1.2e+12`), boolean (`true` or `false`), `null` value, *string* escaped with double quotes `"text"`, *object* or *array*
- *object*: an unordered set of name/value pairs, where name must be a *string*
- *arrays*: an ordered list of *values*

The advantages of JSON are that its format is simple and minimal, requiring only a little overhead to add structure to the data. It is also easily readable by humans, as well as with machines, as the parser is simple to implement and available in about every existing programming language.

```
{
    "books": [
        {
            "isbn": "978-1-23-456789-7",
            "pages": 465,
            "weight": 457.7,
            "available": true
        },
        {
            "isbn": "978-2-12-345680-3",
            "pages": 590,
            "weight": 681.21,
            "available": false
        },
    ]
}
```

Listing 1: Example of valid JSON document.

2.4.1.5 Problem with JSON — no implicit semantics

While JSON is surely a way how to give the data some structure, it does nut any semantics to it, i.e. there is no implicit way to understand what the parts of the JSON document actually mean. This might not be a significant issue when there is a human looking at the data. However, if the data has to be processed autonomously by the machine, a strict specification of the document semantics is a necessity. Authors of RESTful Web APIs [57] call this issue a *semantic gap* or a *semantic challenge* and claim that this is the reason why there exist so many REST APIs that look, at first sight, very similar to each other, but in reality, they are completely incompatible with each other. The fact that JSON became a de-facto standard of the REST API's is not of any help because JSON, in its pure form, does not address this semantic issue at all.

In the efforts of addressing the drawbacks of JSON-based APIs, there have been developed numerous specifications or standards building on top of JSON, trying to add some meta-data and semantics to the JSON documents that follow this standard.

JSON Schema [26] Contrary to the DTD in XML, JSON does not implicitly let the client know what structure of JSON it should expect. JSON Schema bridges this gap by describing a JSON data format, specifying what objects, key names, arrays and values are permissible in a specific location or whether they are required or not. Shipping JSON Schema with the API documentation, for both requests and responses, is a great way to help a client to boost its productivity significantly with working the API, as it always knows how a valid request should look like and what to expect as a response. Without a schema, a client is often required to guess, and it is forced to try out a number of requests to simulate undocumented edge cases to see how the API behaves. Using a JSON schema to validate the client's requests also simplifies the server's validation logic.

JSON-LD (JSON Linked Data) [28] An extension of the JSON format that provides a structured way to connect the data and describe relationships between them, so they are easily processable by humans and ma-

chines. It is a suitable format for connecting the data within unstructured databases like MongoDB or Apache CouchDB or for creating RDF documents for the semantic web.

HAL (JSON Hypertext Application Language) [29] Another way how to add support of links to the JSON documents, simpler than JSON-LD.

Collection+JSON [1] When response contains in `Content-Type` header a value equal to `application/vnd.collection+json` instead of `application/json` that is specified in JSON RFC [4], it is in a special Collection+JSON format. The document is still valid JSON parsable with the original parser, but when treated as Collection+JSON format, additional constraints requiring each data object to be published with the corresponding URL make it a standardized way of publishing a searchable list of HTTP resources on the internet.

JSON:API While JSON Schema is only limited to describe and validated single JSON documents, it is not opinionated in any way on how the documents should look like, nor does it address the relationships between them. A JSON:API was developed as an opinionated approach on how to address the most common problems and issues almost every JSON API needs to deal with, to provide a uniform and standardized way, preventing the "re-invention of a wheel" with each new API. JSON:API specifies both how the client's and server's requests and responses should be formatted. The specification touches on various different areas and provides an opinionated solution to a problem like how to structure and format data resource objects, how to identify them, how to create collections, how to denote relationships between the objects, how to implement pagination, how to specify requests for sorting and filtering, provides a standardized way how to format error messages and many more.

Open API v3 Going beyond the rules and conventions on how to format a collection of JSON documents, Open API aims to describe the HTTP API as a whole. This means that the Open API definition document contains not only schema defining a structure of requests and responses, but, for instance, also examples of such calls, their possible response codes, documentation with rich text formatting, and details about the authentication of the endpoints. The Open API document should be a complete description and documentation of the whole API. There exist a number of tools that consume the Open API specification file and generate nicely formatted documentation as a set of HTML files that could be straight away published on the developer's portal.

2.4.1.6 Best practices for REST APIs

There are numerous guidelines, conventions, and best practices on designing REST APIs, and almost every significant technological company has its document on creating APIs. The authors of [45] analyzed 32 such guidelines, including global technology giants like Amazon, Atlassian, GitHub, Cisco or Microsoft. They found out what were the most common best practices that appeared across their REST API guidelines. In this section, we will provide an overview of a few categories the authors of the study picked for the in-depth analysis to examine whether there was a consensus among the different guidelines.

Backwards compatibility and versioning As backwards-compatibility is a very important feature of an API, it was strongly encouraged to extend API only in a backwards-compatible fashion and discourage versioning.

Only if the backwards-compatible changes were not possible, there was a consensus that some form of versioning has to be implemented. However, the guidelines showed a lack of consistency on how to do it. The differences were, for example, whether to use a single integer or some form of semantic verioning [54], whether to put the version into URL as prefix like `/api/v2`, as a suffix as a query parameter like `?version=3` or include the version information in the request headers only.

Naming Most guidelines recommend consistent and intuitive naming, consistent in the usage of singulars and plurals. Some suggest a set of words to avoid in naming. A slight majority of the guidelines prefer `camelCase` for naming fields instead of `snake_case`, as this is consistent with JavaScript conventions.

URL Structure A URL in REST APIs identifies a resource. It could be either collection of all instances of that resource using URL `/resource` or a specific instance of the resource denoted by URL `/resource/identifier`. In some cases, it might also be practical to put related sub-resources into URLs. Most guidelines suggest including only one level of sub-resource, so the URL looks like `/resource/identifier/sub-resource`. One guideline discourages sub-resources altogether, as it argues that it is better for backwards compatibility and promotes always using filters instead. Guidelines also do not discuss which types of relationships are suitable as sub-resources in the URL. For example, one-to-many relationships, like accessing all posts belonging to a single user using URL `/user/<user_id>/posts` might be appropriate, but if the relationship type is one-to-one or many-to-many, using sub-resources in URL should be discouraged.

Standard and custom HTTP methods The standard set of HTTP methods that most of the guidelines recommend using are GET, POST, PUT, and DELETE and some also add PATCH, HEAD and OPTIONS methods. It is strongly encouraged to respect methods properties like safety from side effects (GET, HEAD, OPTIONS), idempotency (GET, HEAD, PUT, DELETE) and ability to cache (GET, HEAD, POST). When it comes to custom HTTP methods, only to guidelines mention them. One advises against it, but the other one suggests that methods like `BatchGet`, `Cancel`, `Move`, `Search` or `Undelete` might be possible.

Response structure and format There is a general consensus that JSON should be used as a standard format. However, some suggest using extensions of the JSON format as a default option, for example, HAL [29] that adds linking capability. The rules also mention formatting all and time in standardized formats like ISO-8601 or RFC-3339. Some guidelines provide the methods for nesting and linking objects, while some do not address them.

Error responses Around half of the guidelines mention that the error response should always contain at least three fields: *code*, *error* type and *message*. Some also go further and suggest differentiating the error message for end-user and developer, sending two error codes: one internal and one HTTP status code, or adding a link to the documentation where developers can find more information.

Status codes In a general sense, to indicate both success and failure, guidelines suggested using commonly understood status codes while being as specific as possible. For example, as mentioned in section 2.4.1.3, more specific like 201 or 204 should be preferred over the general status code 200, when a new resource was created or removed as a result of the request, respectively. Following this rule is even more important in the case of the client's errors, as an appropriate status code and a well-formed and precise error message will help the client identify what is wrong with the request and how to fix it.

Documentation While there is no doubt good API documentation is essential, they cannot agree on if it is better to use the tools for automatic generation of the documentation and, if so, whether to manually decorate it afterwards or not. Automatic documentation generation is possible when the API is described using the OpenAPI specification file, as there exist a number of generators, e.g. SwaggerUI [58], Redoc[55] or DapplerDox [10], capable of transforming the specification file into nicely formatted documentation.

2.4.2 Backend-for-Frontend pattern (BFF)

In microservice architectures, the business logic in the backend is partitioned into several fine-grained microservices. Each of them is a single, well-defined purpose, and each of them exposes a different API. The microservices are owned by the backend teams responsible for meeting defined functional and non-functional requirements and providing an API that will serve the purposes of the API consumers. The API consumers could be either external third party services or the frontend teams creating the user interface for the application on various platforms and devices.

For the sake of this section, we neglect the fact whether the underlying backend infrastructure is microservice-based oFigure: A structure of the downstream services should not be public and visible for the clients, as they should be communicating only through the exposed API layer. illustrates how the underlying downstream architecture should not be visible to the API clients, as they should be communicating solely though the exposed API layer.

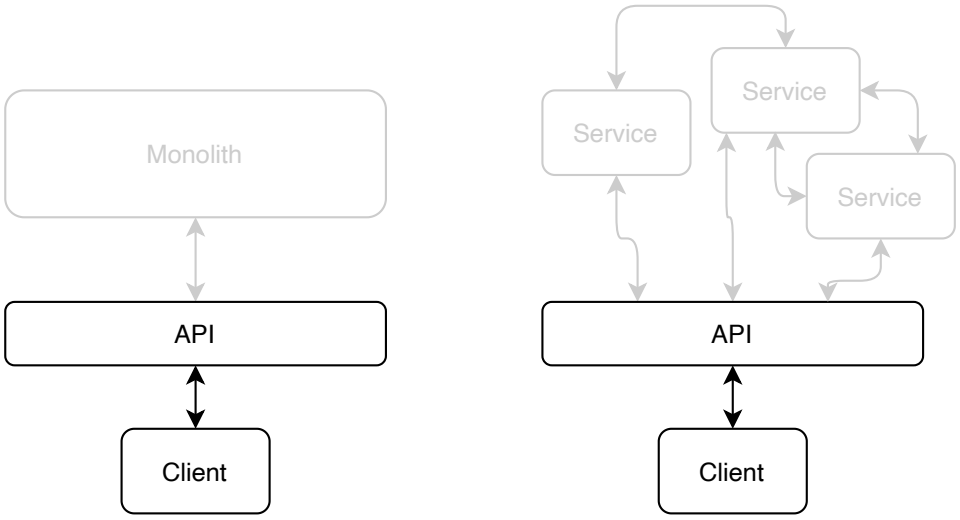

Figure 2.11: No difference form the client's perspective. A structure of downstream services should be hidden and irrelevant for API clients.

API consumers have different needs, require the API to fulfil different use-cases, and ideally want the API specifically tailored to their needs. However, the responsibility for API design lies on the backend teams' shoulders, and for them, it is easiest to provide a universal, One-Size-Fits-All (OSFA) API to all of their clients. However, while OSFA API is convenient for API providers, it could be cumbersome for API consumers. Different consumers have different needs and capabilities. For example, a product page of a web frontend of an e-commerce application would display a more detailed version of the products and images, while the mobile version would use a lower resolution image and would omit the product details. The inconsistency of the clients' needs emerges from their variable properties. For example, for the mobile devices, it would be

reduced screen size, limited computing resources that need to optimize for the battery life and usage of slow and unreliable mobile networks and limited data plans. With the increasing number of devices connected to the Internet of Things, the number of different clients with different needs increases even further. Having a single OSFA API that tries to answer all use cases of all different clients leads to a situation where the API might be too complex and challenging to understand and use by the clients. Single API that servers many different clients creates a high degree of coupling of the API, making it a single point of failure difficult to change and maintain.

The problem of serving different clients could be addressed in various ways. One could be, for example, adding a new endpoint for each device.

`/api/web/products`

`/api/mobile/products`

or

`/api/products?client=web`

`/api/products?client=mobile`

However, one can imagine that for a really large number of devices, this would generate an enormous amount of endpoints that need to be maintained.

Other solution could be to make use of some "query language convention" that is capable of filtering and requesting related entities, like `JSON:API` [27], or conventions used by well-known internet companies like Google or Facebook.

`/api/products?include=image&fields[products]=name,sku,price&fields[image]=title,url` (JSON:API)

`/api/products?fields=name,sku,price,image(title,url)` (Google)

2.4.2.1 Netflix

In 2012, the problem of single OSFA API and the fact that it has to communicate with more than 800 different devices was the problem they faced [46] in Netflix, a popular video streaming platform. All of these devices, including Smart TVs, set-top boxes, tablets, smartphones or gaming consoles, had different memory capacity and processing power; therefore had different requirements for the video quality, bandwidth, formats and encodings they were able to process.

For the reasons mentioned above, they had a difficult time serving all their clients using their OSFA API and achieve an optimal user experience on all their devices. They approached the problem with an interesting solution - they stopped trying to adapt all the different devices to use a single API and turned the whole situation upside-down - they ditched the OSFA API and started to embrace the differences by building a separate endpoint for each of their devices. Interestingly enough, Netflix patented this approach in a patent called "Api platform that includes server-executed client-based code". Because the frontend/device teams were the ones that knew the best how the API they would like to communicate with should look like, they came up with a model where the frontend teams could create an "adapter" residing on a server that would be a point of communication with their frontend devices and would mediate the communication to the downstream backend services.

The philosophy of the Netflix approach could be summarized with following key points:

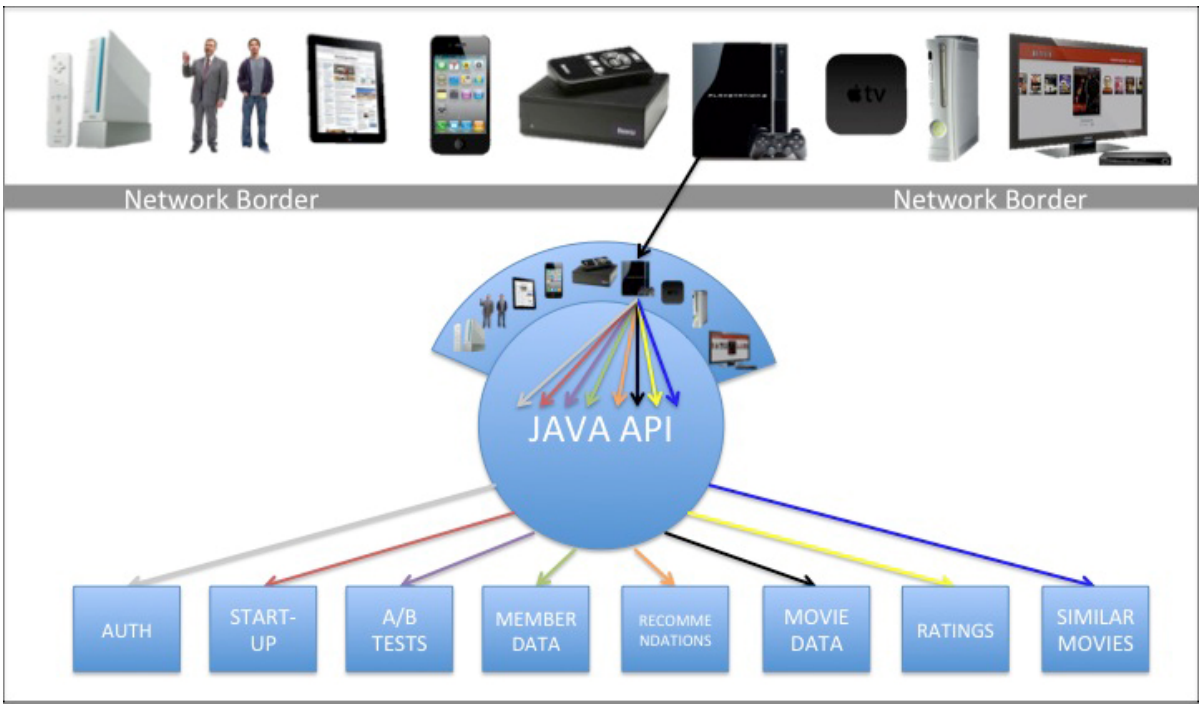

Figure 2.12: Netflix adapters architecture. *Source:* [46]

Embrace the fact that frontends have differences. Frontend teams could tailor their API endpoints to their exact needs. This includes the formats used for communication, the structure of the data, and the exact subset of the data itself that was actually used on the frontend devices.

Separate data gathering from data formatting and delivery. When frontend devices communicate with OSFA API, or worse, directly with the number of different services, the data they receive is often redundant and not in a format that is ideal for them to consume. Therefore, the frontend code must often contain non-trivial logic that needs to parse, join, reformat and sanitize the data, resolve their inconsistencies and handle appropriately various edge-cases, missing data or retransmission of the failed requests. With the adapters model, all this logic could be moved to the server, providing the clients with the data exactly as they need them, leaving their code as thin as possible and focused solely on the display logic.

Move the responsibility boundary between backend and frontend teams closer to the server. Traditionally, the responsibility boundary between the backend and frontend teams would be identical to the network boundary. However, when the frontend teams are responsible for part of the code that resides on the server, this also means that the responsibility boundary moves accordingly — closer to a server.

Distribute innovation. Because frontend teams gained control over their backend API adapter, any changes or experiments in this API would be much faster as the need for communicating with the backend teams responsible for the downstream services was minimized. Moreover, because the adapters were relatively isolated from each other, the bugs that could have been introduced with the change to the API adapter would only affect the single device for which the same team was responsible for and, therefore it would be speedy to isolate and fix bugs when they eventually arise.

While the approach of server-based adapters owned by fronted teams certainly brings a large palette of benefits, it comes with some trade-offs that need to be considered. First, the frontend teams expertise is, obviously, frontend. This includes technologies like HTML, CSS and JavaScript with web frontends or Swift and Kotlin for iOS and Android mobile apps developers, respectively. Now the frontend teams need to potentially learn a new language and paradigms that are used on the server-side. Unless the services adapters are thoughtfully isolated from each other, it could also pose a risk of introducing bugs into the server codebase by non-experienced frontend teams creating, for example, infinite loops and subsequently stressing the backend infrastructure inadequately. Adding a new server service, a new layer of abstraction, that the data needs to flow through also naturally increases an overall latency. However, since this extra communication is happening within the server network that is usually much faster and reliable than the communication over the internet, the added latency is arguably negligible compared to the savings in latency that are enabled because of the tailored optimizations between the API layer and the frontends are now possible.

2.4.2.2 SoundCloud

Another company, called SoundCloud, was going through the migration of their monolithic application towards the microservice architecture. [53]. The monolith exposed a single API serving multiple clients like web applications, Android apps, iOS apps, and external partners and mashups. As API grew in features, it started to suffer from the limitation of the OSFA API, as, for example, it did not consider the needs for smaller payload sizes and reduced request frequency for the mobile apps. Any change or a new feature of the API was needed to be coordinated with the backend teams, having poor knowledge of the needs of the mobile devices, resulting in a lot of friction, communication overhead and delay in implementing any changes to the API.

To solve this issue, SoundCloud allowed its frontend teams to implement customized API endpoint, so they were in control of where to get the required data from, how to aggregate and format them and how to transmit them to frontend clients and making there the owners of this new layer. They called this approach the Backend-For-Fronted pattern [5, 53].

In section 2.2, we wrote about how applications move from monoliths to microservice architectures. However, we do not discuss any strategies of how to achieve such transition in an existing system. Exhaustive discussion about this problem is out of the scope of this thesis, so we only mention that as mentioned in [5], adopting the Backend-for-Frontend pattern could be incorporated as a stepping stone in a transition strategy. As also illustrated in **??**, decoupling API layer from underlying downstream backend, whether it is a monolith or set of microservices, allows to freely perform incremental changes and refactorings between API layer and downstream backend without any need for changing the frontend.

2.4.2.3 Conclusion

The main idea of Backend-for-Frontend is having "different backends for different frontends" [5] owned by the frontend teams, as illustrated on 2.13. This enables to move all the logic that is not pure display logic and thus does not belong to the frontend to the edge service residing on the backend. Consequently, it increases productivity as it enables faster iteration cycles, minimizing the need for coordination between teams and implementing new features faster. In mobile applications, the increased velocity is also supported by the fact that with minimal logic in the frontend code, the approval process of the application required by the stores is faster and needed less frequently. Backend-for-Frontend also enables specific optimizations of the API, improving performance leading to better user experiences. Adopting the Backend-for-Frontend pattern sim-

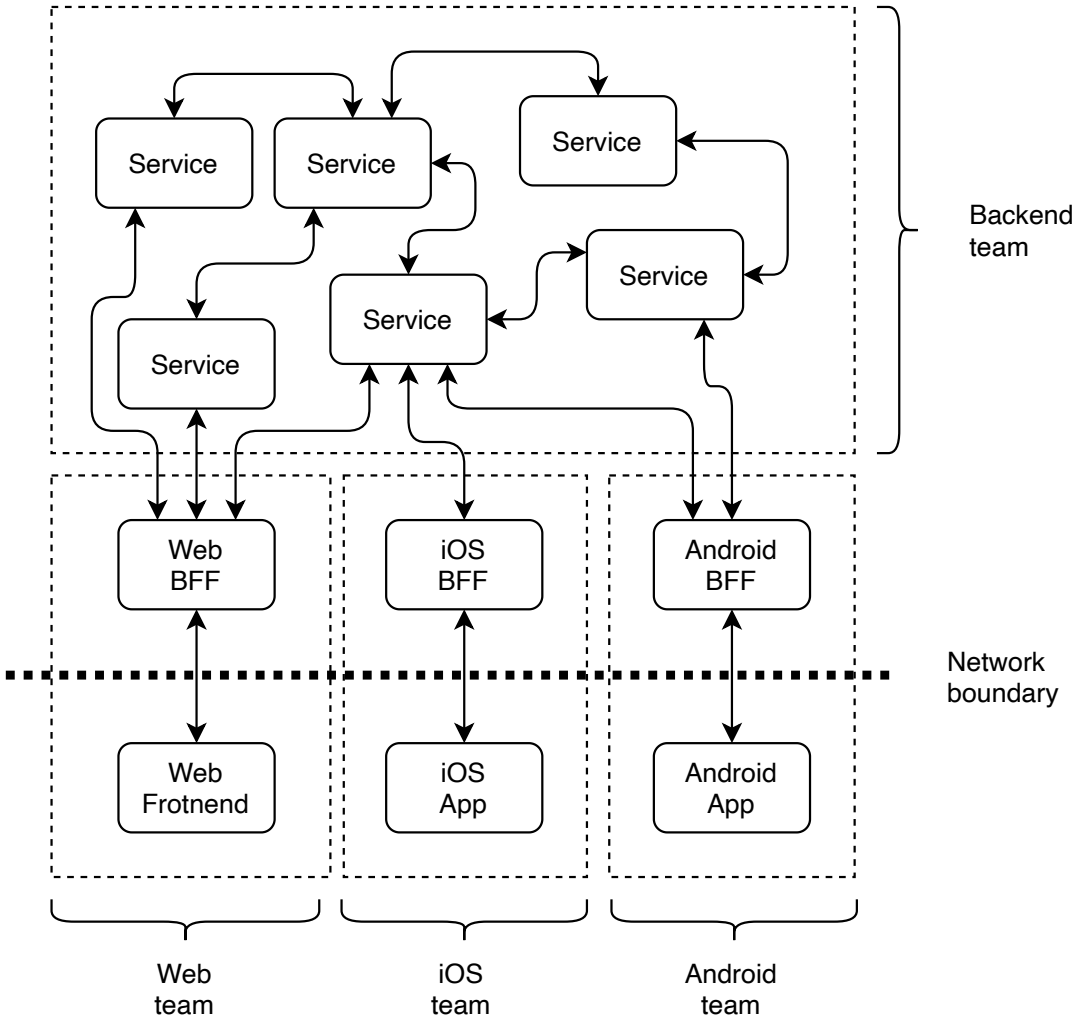

Figure 2.13: An illustration of Backend-for-Frontend pattern.

plifies the transition from monolith to microservices, as it allows to incrementally split the monolith without any need to change frontends.

On the other hand, giving frontend teams responsibility for backend service does indeed sound like it could lead to some problems. Indeed, frontend teams specialise in technologies like HTML, CSS, JavaScript or languages like Kotlin and Swift for mobile development. The backend services usually use different languages that frontend teams need to learn to use. Implementing a backend service could also require adopting different tools, workflow and different design principles than frontend. Since it is to be expected that frontend developers would have minimal experience with this, the expectations for the quality of their backend code cannot be set too high. Forcing frontend teams to deal with technologies they are unfamiliar with could cause them frustration and consume a lot of their time. Therefore, as they did in SoundCloud [53], the backend teams should support the frontend teams and their life as easy as possible by, for example, by creating a library or SDK that makes it simple to create and connect the API adapter to the existing environment and already solves the cross-cutting concerns, like authentication or logging. What also needs to be taken with caution is that possible bugs in backend services that, for instance, could cause infinite loop calls to other services could quickly consume their resources and make them unavailable, causing the entire backend malfunction. Allowing inexperienced frontend developers inexperienced to make changes in the backend could raise the risk of causing these situations.

When implementing Backend-for-Frontend, questions like the following would inevitably arise:

- *How many APIs should we have? When are the clients different enough that they deserve a different API?*
- *As new features and devices are added, when is the right time to split the existing API into separate ones?*
- *Does it bring enough benefit to provide different BFFs for different clients even at the expense that features would be duplicated, or would it be better to provide the APIs based on the features rather than clients?*

These questions do not have universally applicable answers. It needs to be considered that both premature separations and avoiding refactoring for too long are both non-optimal, and it is up to developers of concrete application to find the sweet spot of doing so. The rule of thumb "Three strikes and refactor" from Fowler's *Refactoring* book [18] could be a good starting point.

To highlight the main differences between both Backend-for-Frontend and Ons-Size-Fits-All approaches, in **??** we provide a comparison of their benefits and disadvantages.

	Backend-for-Frontend	One-Size-Fits-All
Pros		
	+ Increases velocity and time to implement new features	+ One API is easy to maintain for backend teams
	+ Enhances user experience of frontend	+ API experience is consistent
	+ Enables more experimentation and faster changes on the frontend	+ Easy to control authentication, rate limiting etc.
	+ Helps with the transition from monolith to MSA	
Cons		
	– Frontend teams need to learn backend languages and processes	– Difficult to learn and use for clients
	– Experience across different APIs is not consistent	– It could quickly become complex and overengineered, as it is trying to provide every possible use case
	– Repetition and overlap of the logic and concerns between different APIs	– Any changes to API are difficult to do because so many clients depend on it

2.4.3 GraphQL

In Facebook, there were also facing frustrations with their API, and the discrepancy between the data they required on frontend ability of servers to efficiently provide them and the considerable amount of code needed to prepare and parse the data on both server and client sides, respectively. They realised that the mental model of the data based on resource URLs, foreign keys and joins was not the best way how to think about the data model. They would prefer to think about it as a graph, where vertices would be the data objects and the edges the relations between them.

In 2015, Facebook introduced [12] GraphQL. Just to avoid any confusion, GraphQL is not a database, nor it has anything to do with the graph theory[1]. Rather than that, GraphQL is a specification for an API query language and a server engine capable of executing such queries. [21].

The discrepancy between the One-Size-Fits-All API and Backend-For-Frontend could create an impression that the engineers would always be trapped into guessing where to draw a fine line between these approaches that are best suited for a particular application. However, guesswork is not a sound engineering approach. What if there is a way that can take the best from both worlds — providing a single API, but the one that fits the needs of all different clients? A GraphQL is an attempt to do exactly that.

In this project, we did not experiment with GraphQL in the context of API Gateways, so we are not going to explain its principles here more in-depth. However, it is an interesting technology that is becoming more increasingly popular and have the potential of replacing the currently dominant position of REST APIs in the future. To start exploring the GraphQL, we recommend starting with the official documentation [23] and to get a more comprehensive overview of the technology; we recommend reading an ebook from Marc-André Giroux: Production Ready GraphQL [21]

[1] a discipline of discrete mathematics and computer science

Chapter 3

Implementation

When searching the internet for some examples of the implementations of the API Gateways, at first glance, there seems to be an overwhelming number of possible options. For example, searching in the GitHub using following URL:

> "`https://github.com/search?q=API+Gateway`"

the results sorted by "Best match" yield these top 10 repositories, as shown in table 3.1.

Position	Repository name	Repository description
1	Kong/kong	The Cloud-Native API Gateway
2	apache/apisix	The Cloud-Native API Gateway
3	ThreeMammals/Ocelot	.NET core API Gateway
4	fagongzi/manba	HTTP API Gateway
5	apache/incubator-shenyu	ShenYu is High-Performance Java API Gateway
6	aliyun/api-gateway-demo-sign-java	aliyun api gateway request signature demo by java
7	gravitee-io/gravitee-gateway	Gravitee.io - API Management - OpenSource API Gateway
8	wehotel/fizz-gateway-community	An Aggregation API Gateway
9	spinnaker/gate	Spinnaker API Gateway
10	ExpressGateway/express-gateway	A microservices API Gateway built on top of Express.js

Table 3.1: Top 10 GitHub's API Gateway repositories by "Best match".

When re-ordering the results by popularity, i.e. "Most stars", the list of top 10 changes to the one shown in table 3.2.

Rank	Repository name	Repository description
1	Kong/kong	The Cloud-Native API Gateway
2	vuestorefront/vue-storefront	The open-source frontend for any eCommerce. Built with a PWA and headless approach, using a modern JS stack.
3	TykTechnologies/tyk	Tyk Open Source API Gateway written in Go, supporting REST, GraphQL, TCP and gRPC protocols
4	ThreeMammals/Ocelot	.NET core API Gateway
5	apache/apisix	The Cloud-Native API Gateway
6	apache/incubator-shenyu	ShenYu is High-Performance Java API Gateway.
7	luraproject/lura	Ultra performant API Gateway with middlewares. A project hosted at The Linux Foundation
8	dherault/serverless-offline	Emulate AWS Lambda and API Gateway locally when developing your Serverless project
9	vendia/serverless-express	Run Node.js web applications and APIs using existing application frameworks on AWS #serverless technologies
10	claudiajs/claudia	Deploy Node.js projects to AWS Lambda and API Gateway easily

Table 3.2: Top 10 GitHub's API Gateway repositories by "Starred".

Even though the list of API Gateway repositories on GitHub is quite extensive, it does not represent all available gateways — but only those who have their code publicly available. The API Gateways that are proprietary, either stand-alone or as a module of some larger system, ale likely to not be found here.

For the purposes of this thesis, we had to decide how to navigate in this abundance of the API Gateway options and narrow down the choices and pick a few that represent well the current state-of-the-art of the free to use, general-purpose, stand-alone API Gateways. These are the main factors, in no particular order, that we considered when choosing the appropriate candidates:

- Reasonably popular
- Stand-alone and for general-use, not bind to specific framework or language
- Open-source
- Easy to deploy on Kubernetes cluster
- Provide an easy-to-use management dashboard

We finally decided on the following three API Gateway implementations:

- Tyk
- Kong
- KrakenD

In the following section, we further explain the reasons why they made it into our list, and we provide a general description of each of them.

3.0.1 Tyk

We chose Tyk [61] because it is the 3rd most starred API Gateway repository on GitHub, provides a wide range of features, including support for GraphQL and gRPC, and last but not least, we used its case studies as a base for analyzing the API Gateway requirements.

The Tyk Gateway itself, or as they call it, Tyk Community Edition, is open-source and available to install and use by anyone freely. However, it is headless — it means that it does not provide any graphical user interface for managing the gateway, and all the configuration must be done using an admin API. However, this free version does not lack any features and does not contain any limitations in terms of the gateway functionality, like a reduced number of supported protocols or limitations for the number of API endpoints that the gateway could manage.

The TykTechnologies, a company behind the Tyk Gateway, offers also paid, enterprise version of their gateway. The options they provide are to use either a fully managed solution residing on their cloud service, a solution self-managed by the client, or a hybrid between the two. The fully-managed cloud edition is the most comfortable for the client to use, as it takes off the burden of setting up and updating the gateway itself, as well as it could be easily scaled to provide the required performance. The Pro version incluide additional components that support the core API gateway functionality, namely Tyk Pump and Tyk Dashboard. The Tyk Pump is a service that is responsible for transferring the data stored in the temporary Redis storage by the gateway into the permanent storage. The Tyk Dashboard is a visual GUI for the management and analytics of the Tyk Gateway.

Because we wanted to test API gateways that provide a dashboard GUI, we opted for free of charge 14-day trial of the self-managed Tyk Pro.

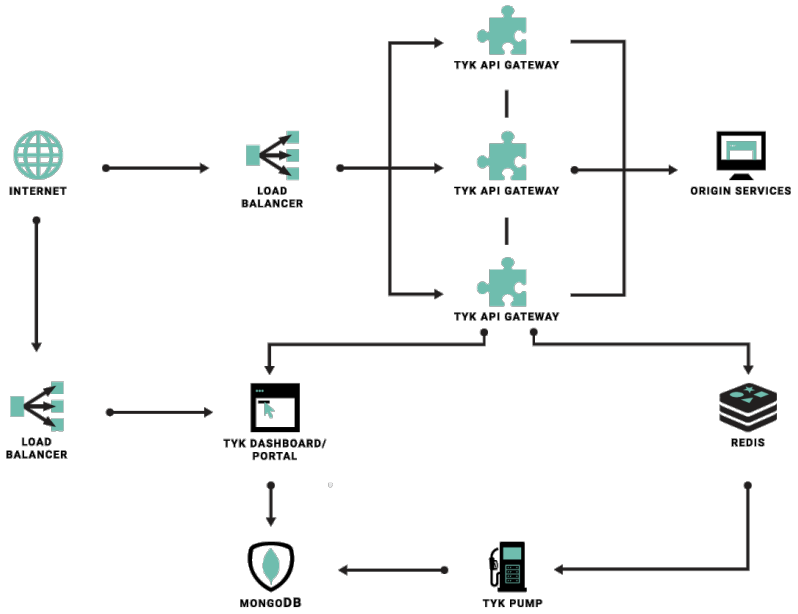

Figure 3.1: The setup of Tyk Pro Gateway. *Source:* `https://tyk.io/docs/tyk-pump/`

3.0.2 Kong

We picked Kong into our list because it is arguably the most popular API Gateway on the market. This statement, which they also claim themselves on their homepage, is supported by the fact that Kong showed up in the first place for API Gateway repositories on GitHub, but when sorted by relevance and popularity.

A Kong Gateway [30] is built on top of Nginx [48], a very popular open-source and high-performance tool usually used as a web server or load balancer. The Kong Gateway (OSS), or Community Edition [31], an open-source version of their gateway that provides basic functionality and is limited only to the community plugins in their Kong Hub [32] repository. A Community Edition does not come with bundled management dashboard. However, there exists an unofficial service called Konga [51] that can be deployed alongside the Community Edition of the Kong Gateway and configured to connect to it through its admin API, thus effectively providing a GUI even for the Community Edition for free.

An Enterprise edition provides a wider range of functionality and enhanced security, either build-in in the form of a native dashboard or as the enterprise-edition plugins available on the Kong Hub. These include, for instance, support for Kafka traffic or GraphQL, advanced rate limiting and caching, more advanced authentication options like OAuth 2.0 or OpenID Connect, or build-in monitoring and alerting.

For this project, we have decided to the open-source Kong Gateway Community Edition with the unofficial Konga dashboard.

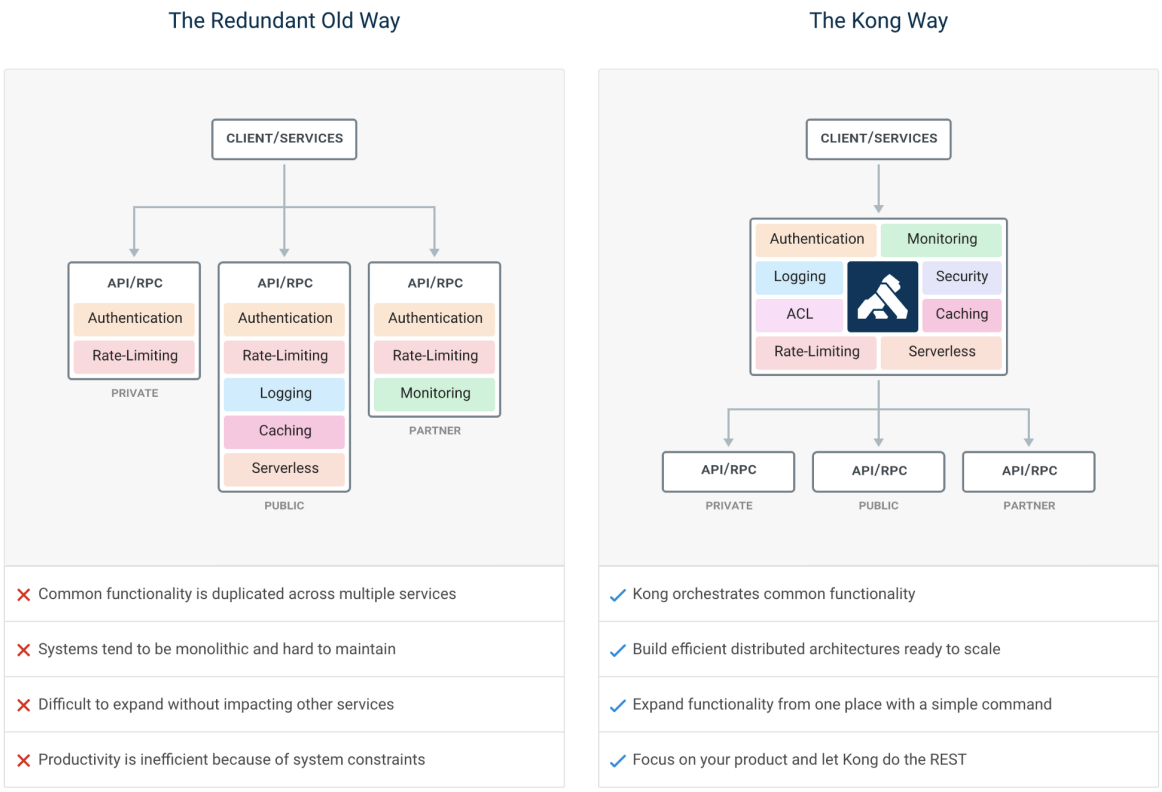

Figure 3.2: "The Kong Way" promises to centralize the management of the cross-cutting concerns of the services in a microservice architecture. *Source:* [13]

3.0.3 KrakenD

KrakenD [34] is the newest out of all gateways we chose. Out of these three, it has unique architecture, as it completely stateless, with a declarative configuration that provides superior performance. Moreover, in their performance benchmarks where they compare it with other gateways, it significantly outperforms both Tyk and Kong [7]. In May 2021, the KrakenD core engine was also donated to the Linux Foundation [37] of as a Lura project [59], so now we can say it became a standard way how to do an API Gateway on the Linux platform.

As the KrakenD is stateless and all the configuration is done using a single configuration file, it does not have a management GUI. Instead, it provides a GUI called KrakenDesigner [38] that is capable of loading, visually editing and exporting the configuration file. KrakenDesigner does not need to be deployed anywhere, as it is sufficient to just run it locally by a person who wants to change the configuration file.

KrakenD Enterprise edition [35] offers a cloud, self-managed or hybrid deployment and in addition to the free version and it comes with the dashboards to monitor and log metrics from the API gateway as well as upstream services to get better insight into what is happening "behind the curtains" and being able to faster debug and fix potential issues. The Enterprise edition capabilities might be extended by the number of enterprise plugins, assing, for instance, support for gRPC. A complete comparison matrix of the free and enterprise edition features is available on their website [36]. We did not find any mention of GraphQL support in either the free or enterprise edition feature list, nor an appropriate section in the KrakenDesigner GUI.

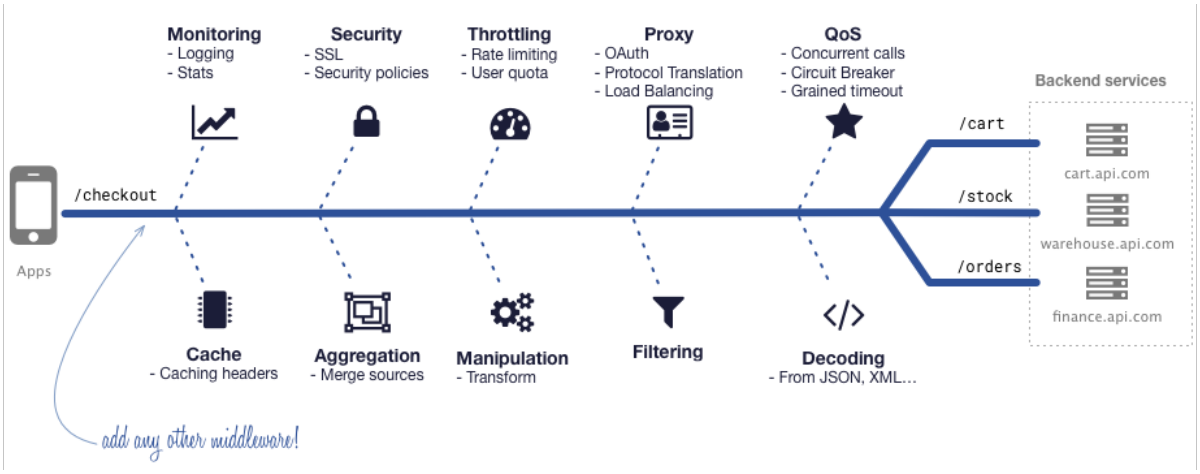

Figure 3.3: A diagram demonstrating the capabilities of the KrakenD gateway that is available on the front page of their website. *Source:* [34]

3.0.4 Other considered solutions

3.0.4.1 Nginx

When developers deploy a REST API, it is likely that they already are using an Nginx [48], a popular and highly performant web server to serve the API. However, Nginx use is not limited be used as a webserver only and could be used in various other scenarios, like a load balancer, content delivery cache or an API Gateway. In fact, the previously mentioned world's most popular API Gateway — Kong uses Nginx in its core. Nginx published a series of blog posts [8], and an ebook [9] that guides through how Nginx, or their enterprise version Nginx Plus, could be configured to function as an API Gateway. The open-source version of Nginx already pro-

vides support for a wide range of protocols, including gRPC and HTTP/2, but for instance, it lacks the support for authentication using JWT that only the Nginx Plus provides. The full feature comparison of open-source Nginx and Nginx Plus is available on their website [47].

The strong argument they state about preferring Nginx over a third-party, standalone API gateway is the importance of the so-called converged approach. Taking a convergent approach means that in the situation when the application is already using Nginx in some other scenario, for example, like web server, cache or reverse proxy, it would be better to utilize Nginx also for another use case — API Gateway — instead of introducing another technology into the stack, as it will increase its complexity unnecessarily. As the API gateway functionality feature set is a subset of what Nginx could provide, it can replace the standalone API gateway.

We did not choose the Nginx to test, as we were mostly interested in the "all in one" easy to deploy and use solutions. The configuration of the Nginx as an API Gateway is done only through the configuration files and arguably requires more expertise than the configuration GUIs the other gateways provide. However, we are mentioning it here because in a lot of scenarios where Nginx is already present in the technology stack and used in other use cases, it might be the optimal approach to leverage it also as API Gateway.

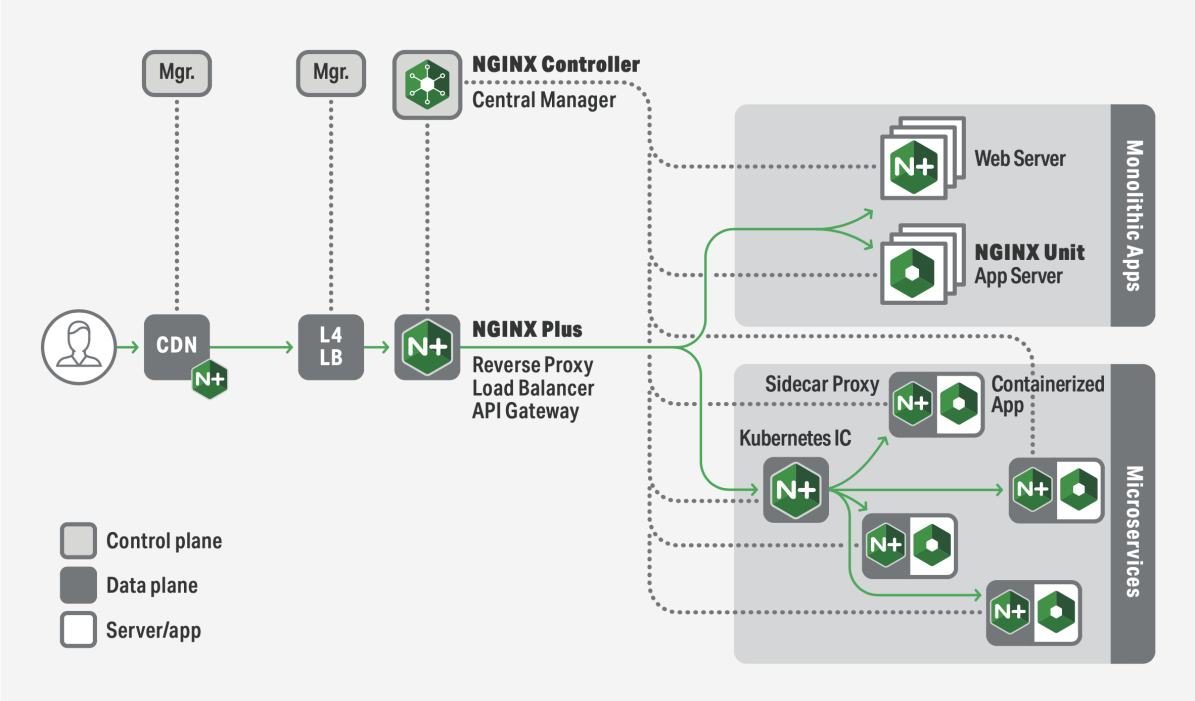

Figure 3.4: Example application architecture using Nginx Plus as an API gateway. *Source:* [9]

3.0.4.2 Apollo Gateway

An Apollo Gateway is an API gateway that was built from scratch to primarily support GraphQL instead of REST. As we mentioned earlier, in a microservice architecture, each service has distinct and separate functions. For example, one service could handle product catalogue, while the other service handles orders, and they both provide a GraphQL API to access their data — either products or orders. However, the fact that orders and products are handled by two separate services is irrelevant for the outside client that accesses the system as a whole and would like to have all resources, including both orders and products, in a single GraphQL graph instead of two — because, as we mentioned in Section 2.4.3, that is a core advantage of GraphQL and that is how it should be used. It turns out that this is a non-trivial problem, where the attempts

trying to address, like schema stitching [24] came with a serious trade-off and turned out to not be universally practical. An Apollo Gateway comes with the Federation design [25], which seems to be superior to the stitching [20, 22].

Apollo Gateway is an interesting project build to support an alternative approach to REST APIs in the form of GraphQL and is capable of exposing only the GraphQL endpoints. However, it could communicate with both GraphQL and REST API upstream services in the backend, making it an appropriate solution when introducing a new customer-facing GrapQL layer on top of existing microservices that have only REST API endpoints.

In the end, we did not end up trying an Apollo Gateway as we did not configure the GraphQL endpoints.

3.1 Cloud-native microservices demo application

As a representative of the backend application, we decided to use a Demo Online Boutique Shop, as it is reasonably complex and was used for demonstration by Google itself, so there is a chance that our readers will already be familiar with it. The application is available in the following GitHub repository:

"`https://github.com/GoogleCloudPlatform/microservices-demo`"

However, while this application was is simulating the complex creation of the orders, it does not have permanent storage where it saves the permanent order, nor does it have an API that could be used to retrieve a list of created orders. Therefore we had to enhance the application with the functionalities.

3.1.1 Adding persistent storage for orders

The `checkoutservice` is written in go. First, we had to create a new Postgres service in the Kubernetes manifests files, and then we changed the code of the `checkoutservice`, so in the last step, when order is created, it is stored to this Postgres instance.

3.1.2 Adding REST endpoint for orders

The second step of the application enhancement was to add a REST API that will be able to retrieve the orders stored in the Postgres database. We decided to create a new `restapiservice` that encapsulates a Node.js Express server that pulls the data from the specified Postgres tables.

The created API endpoints were following:

a) Collections of resources, each endpoint supports limit and offset query parameters:

- `GET /orders`
- `GET /order_items`
- `GET /addresses`
- `GET /shippings`

b) Endpoints to retrieve a single instance of a resource, identified by id:

- `GET /order/:id`

- `GET /order_item/:id`
- `GET /address/:id`
- `GET /shipping/:id`

We used a local minikube cluster when enhancing the application. Then we deployed it to the Google Cloud Kubernetes cluster, where we continued with installation and testing of the API Gateways.

3.2 Installing API Gateways on Google Cloud Kubernetes cluster

In the following section we provide a series of steps of how to install each Gateway on a cluster. Each guideline consist of the successions of steps that must be executed in respective order, whereas the steps are mainly commands that need to executed in console and changes that need to be made to the configuration files.

3.2.1 Installing Tyk

1. Install Tyk from using helm:

```
# Add TYK repo to helm and create "tyk" namespace in k8s
helm repo add tyk-helm https://helm.tyk.io/public/helm/charts/
helm repo update
kubectl create namespace tyk

# Install dependencies - Redis and MongoDB (these simplified packages are NOT
# for PROD usage!)
helm install redis tyk-helm/simple-redis -n tyk
helm install mongo tyk-helm/simple-mongodb -n tyk

# Generate configuraiton file
helm show values tyk-helm/tyk-pro > values.yaml
```

2. Set values in `values.yaml`:
   (a) `dash.license` → paste Tyk 14-day trial license here
   (b) `dash.license` → paste Tyk 14-day trial license here
3. Complete the installation:

```
# Install Tyk Pro (includes both gateway and dashboard)
# Note: "--wait" is for some reason necessary, so don't omit it and be patient,
# it takes some time to start
helm install tyk-pro tyk-helm/tyk-pro --version 0.9.1 -f values.yaml -n tyk --wait

# Expose and open up dashboard in the browser
minikube service dashboard-svc-tyk-pro -n tyk
# or using kubectl way
kubectl port-forward service/dashboard-svc-tyk-pro 3000 -n tyk
```

4. Log in to dashboard using username: `default@exmaple.com` and password: `password`

3.2.2 Installing KrakenD

1. Get a KrakenD config file:

   (a) Design you own using KrakenDashboard

   ```
   docker run --rm -p 8080:80 devopsfaith/krakendesigner
   ```

   (b) Or get a sample one form e.g. here: `https://github.com/devopsfaith/krakend-ce/blob/master/krakend.json`

2. Create `Dockerfile` with following content:

   ```
   FROM devopsfaith/krakend:1.4.1
   COPY krakend.json /etc/krakend/krakend.json
   ```

3. Build it:

   ```
   docker build . -t gcr.io/YOUR_REPO/krakend:1.4.1
   ```

4. Push it to the Kubernetes image repository (to make this work, it requires some one-time set up using `gcloud` utility):

   ```
   docker push gcr.io/YOUR_REPO/krakend:1.4.1
   ```

3.2.3 Installing Kong

1. Install Kong gateway:

   ```
   kubectl create namespace kong


   # $ clone https://github.com/Kong/charts as kong-charts
   # $ cd kong-charts/charts/kong
   # in file values.yaml, make following changes:
   #  -> enable postgres database: set "env.database" to "postgres"
   #  -> enable automatic installation of Postgres service: set "postgresql.enabled" to "true"
   #                                                         and uncomment the section under it
   #  -> enable Admin API: set "admin.enabled" to "true"
   #  -> enable plain HTTP: set "admin.http.enabled" to "true"


   helm install -n kong -f values.yaml kong ./
   ```

2. Install Konga Dashboard:

   ```
   git clone https://github.com/pantsel/konga konga-charts
   cd konga-charts/charts/konga
   helm install -n kong -f values.yaml konga ./

   # forward port to local machine
   ```

```
kubectl port-forward service/konga -n kong 8080:80

# Konga Dashboard will be at: http://127.0.0.1:8080/
```

3. Configure Konga Dashboard:
   (a) Create admin user and choose a password
   (b) Add and activate connection to the Kong Gateway with the following data:
      - Kong Admin URL: `http://kong-kong-admin.kong.svc.cluster.local:8001`
   (c) Create Service with following data:
      - Protocol: `http`
      - Host: `restapiservice.default.svc.cluster.local`
      - Port: `5000`
   (d) Add a route to service with following data:
      - Paths: `/api` (don't forget to press enter when adding)

3.3 Designing performance tests

When creating an API, each additional layer that requests and responses need to go through consumers additional resources and adds extra latency to the communication. As we deployed the API Gateways in front of the REST API, we wanted to measure whether the added latency might be a concern that would affect the experience of the API consumers or whether the extra consumed resources would make a significant difference in how much clients could be served at once.

To verify whether our concerns were valid or not, we decided to create a test that will load our REST API endpoints with the number of simultaneous requests, and we will measure whether there is any significant difference in latency of the processed requests.

The rationale behind setting a test for the API endpoints this way is to discover how the experience of the API consumers will be affected when API gateway will be introduced into the system compared to the previous situation without an API Gateway. In our test scenario, the gateway does not perform any additional authentication or request manipulation, so what we should be testing is only how much extra resources and time it takes to re-transmission of the request and response through the gateway itself.

For creating a test, we used a tool called Apache JMeter [2] that allows us to easily design and run performance tests for the REST APIs. We created a test will the following characteristics:

- each thread call each of our four endpoints for collections: `/orders`, `/addresses`, `/order-lines` and `/shippings`
- each thread repeats the call of all the endpoints 10-times
- there is a configurable number of how many simultaneous threads are run. We used values: 10, 50 and 200

We tried to make sure to compare the request with and without passing through the gateway as free of external factors as possible, so for each test, we followed the same procedure consisting of the following steps:

1. We configured and published the API Gateway API on the cluster
2. We ran the test suite against the API without the gateway to create a baseline
3. Right after finishing the baseline run, we ran tests against the gateway

This way, we ensured that the following was true to minimize the effect of the external effects on the test results:

- There was the same set of active pods deployed and active on the cluster
- Tests were run shortly after each other, always from the same computer using the same internet connection

For each gateway, we ran the performance tests using the following set of commands:

```
# Run baseline test
jmeter -n -t api_test.jmx -Jusers=5 -l results/API_GATEWAY_5_x10_baseline.jtl
jmeter -n -t api_test.jmx -Jusers=50 -l results/API_GATEWAY_50_x10_baseline.jtl
jmeter -n -t api_test.jmx -Jusers=200 -l results/API_GATEWAY_200_x10_baseline.jtl

# modify the test file, so the IP it calls points to the gateway

# Run tests for API_GATEWAY (either tyk, krakend or kong)
jmeter -n -t api_test.jmx -Jusers=5 -l results/API_GATEWAY_5_x10.jtl
jmeter -n -t api_test.jmx -Jusers=50 -l results/API_GATEWAY_50_x10.jtl
jmeter -n -t api_test.jmx -Jusers=200 -l results/API_GATEWAY_200_x10.jtl
```

Chapter 4

Results

In this chapter, we will present the results of the comparison of the selected API Gateways, from both functional and performance perspective.

4.1 Comparison of functionality

The data for table 4.1 were collection as the combination of information from the official documentation and observation of the available option when trying the API gateways in practice.

Funcitonal Requirement	**Sub-requirement**	**Tyk**	**KrakenD**	**Kong**
Authentication & Authorization	In general	✓	✓	✓
	Groups	✓	✓	✓
	JWT	✓	✓	✓
	OAuth	✓	✓	✓
	OpenID Connect	✓	✗	*paid*
Monitoring & Logging	Out-of-box	*paid*	*paid*	*paid*
	Third party	✓	✓	✓
Routing	In general	✓	✓	✓
Transformation of data, protocol and formats	Requests	✓	✓	✓
	Responses	✓	✓	✓
	HTTP/2	✓	✓	✓
	WebSockets	✓	*paid*	✓
	gRPC	✓	*paid*	✓
Service discovery	Support?	✓	✓	✗
Rate limiting and throttling	In general	✓	✓	✓
Caching and compression	Response caching	✓	✓	*paid*
Aggregation / Composition	Of requests	✓	✓	✗
Circuit breaker	Supports?	✓	✓	✗
Load balacing	Between bakcend services	✓	✓	✗
Supprot for GraphQL	In general	✓	✗	*paid*
	REST backends	✓	✗	*paid*
Declarative configuraiton / stateless	An option	✗	✓	✓
Support for plugins and middleware	Support plusings	✓	✓	✓
	Official plugin repo	✗	✗	✓
	Extra enterprise plugins	✓	✓	✓

Table 4.1: Comparison table of the supported function of the tested API Gateways.

4.2 Comparison of performance

We provide two tables, in the table 4.1, the data are based on the overall the request took to process. In the second table , we subtracted the connection time, so we that was occasionally very high and caused increased the whole request-response time significantly. When we subtracted the connection time, we got the "cleaner" results where we removed these occasional outliers.

Geteway	Threads	Simultaneous API calls	Total API Calls	Baseline Average	Baseline Median	Baseline std. dev.	Gateway Average	Gateway Median	Gateway std. dev.	Difference the gateway adds Average	Difference the gateway adds Median	Difference the gateway adds std. dev.
Tyk	5	20	200	149,2	144	23,576	149,43	143	33,762474	0,23	-1	10,18602
	50	200	2000	168,407	155	81,694	168,194	155	86,561867	-0,213	0	4,867398
	200	800	8000	327,8301	170	1257	337,287	172	1246,3468	9,45662	2	-10,6575
KrakenD	5	20	200	136,385	134	21,594	147,165	141	26,018118	10,78	7	4,423945
	50	200	2000	143,061	133	86,533	162,708	147	84,775122	19,6465	14	-1,75759
	200	800	8000	419,8213	196	1351,7	321,663	150	1413,8295	-98,1588	-46	62,13449
Kong	5	20	200	150,925	138	43,57	143,475	138	28,490312	-7,45	0	-15,0794
	50	200	2000	153,2215	141	71,521	152,291	140	92,28008	-0,9305	-1	20,75936
	200	800	8000	320,6389	146	1450,9	328,867	152	1245,915	8,22813	6	-205,007

Figure 4.1: Performance results for sample time.

Geteway	Threads	Simultaneous API calls	Total API Calls	Baseline Average	Baseline Median	Baseline std. dev.	Gateway Average	Gateway Median	Gateway std. dev.	Difference the gateway adds Average	Difference the gateway adds Median	Difference the gateway adds std. dev.
Tyk	5	20	200	147,275	142	22,227	145,865	140	33,2393	-1,41	-2	11,01207
	50	200	2000	161,7955	153	28,009	158,793	151	35,740311	-3,0025	-2	7,731475
	200	800	8000	178,8973	167	45,512	184,142	166	55,854498	5,24462	-1	10,34217
KrakenD	5	20	200	134,57	132	20,019	144,2	139	25,380917	9,63	7	5,361452
	50	200	2000	136,235	131	21,334	153,43	143,5	30,492982	17,1945	12,5	9,158752
	200	800	8000	218,1785	182	147,75	157,487	146	35,432567	-60,6911	-36	-112,322
Kong	5	20	200	145,87	136	35,256	140,95	137	21,100323	-4,92	1	-14,1561
	50	200	2000	146,475	139	23,359	145,056	139	24,444042	-1,4195	0	1,085206
	200	800	8000	153,4495	144	33,896	175,162	149	70,953957	21,7128	5	37,05818

Figure 4.2: Performance results for latency without connection time.

Chapter 5

Discussion

In this discussion chapter, we will summarize the main points of our findings, interpret them with regards to the research questions, draw implications from them, and denote their limitations.

At the beginning of this thesis, we explained what the microservice architecture is, why is it an important trend, but we also pointed out what are its typical issues. One of the approaches that are available to address the mentioned issue is to use an API Gateway, so we decided to examine in depth the role of API Gateways in microservice architectures by asking the following research question:

> **RQ:** *"How to successfully leverage API Gateway to efficiently implement and manage the API layer between the external clients and the system based on microservice architecture?"*

that could be further specialized by extending the question with following additions:

> **RQ.1:** *"... from the perspective of features?"*
> **RQ.2:** *"... from the perspective of performance?"*

5.1 Features

As a starting point, we constructed a list of functional requirements of an API Gateway from examining the series of case studies and conducting one interview. This served as a basic framework that helps us see what functionality is important in an API Gateway and which is not, thus providing an answer to the *RQ.1* sub-question.

As there were many options of which API Gateway to choose from, we needed to make a decision on selecting a few representatives of the current commonly used API Gateways for evaluation, and we chose Tyk, Kong and KrakenD, mainly based on their popularity, user-friendliness and open-source code.

What we found out when comparing the features of the gateways was that for the majority of the requirements, there were fulfilling them. However, neither of them supported all of them completely, where a Tyk was getting the closest, just without an option of the stateless deployment based on declarative configuration.

In the process of choosing the best API Gateway for the specific application, what would be the best choice is highly dependent not only on the requested features but also on the deployment options, existing

application stack, estimated workload and in the case of an enterprise solution, also on pricing. In some scenarios, neither of our tested API Gateway would be the best solution — it could easily be the Nginx if the application already uses Nginx as a web server or Apollo Gateway if the application's microservices provide GraphQL API instead of REST.

5.2 Performance

To test the API Gateways in practice and verify their list of features and how well they perform, we needed to deploy a sample microservice-based application that will provide a backend service for the API Gateway. We chose the microservice application demo of the Online Boutique Shop created by Google. We chose this application as it is a good representative of a reasonably complex microservice application. It is easily deployable to the Kubernetes cluster and was used repeatedly in various Google showcases, so there is a chance that our readers will be already familiar with it. The drawback of this application was that it did not expose any REST API endpoints, nor it did not store any persistent data — so we extended it and implemented these missing parts, so it was possible to actually configure and test the API Gateways and receive data generated by this application.

When deciding on how to test an API Gateway performance, we were already aware that there already exist the benchmarks comparing how many requests can an API Gateway process at the same unit of time [7] and according to them, in the context of our three tested gateways, a clear winner was KrakenD, Kong a runner-up and Tyk as the least performant one. Instead of trying to repeat these benchmarks, we decided to test something different — how is the processing time of a single request affected when facilitated by the gateway, compared to a situation without gateway, at various loads. We did not observe any added latency of processing requests in each of the tested gateways. However, we observed that with the Tyk Gateway deployed, during even the lower loads, the average and median time of the request processing was about ten milliseconds longer than when Kong or KrakenD were deployed. An interesting fact was that this extra latency was added regardless of whether the request was going through the Tyk gateway or not. Our interpretation of this observation is that as Tyk consumes the most resources out of all three tested gateways, it left fewer resources for the bare REST API service that took longer to respond to the requests.

5.3 Recommendations and Future Work

The fact that Tyk fulfilled most of the requirements and the requirements were in vast majority obtained from analysis of Tyk case studies points to the fact that more case studies from different sources of different gateways should be incorporated into the analysis, for example, the case studies published on Kraken website [33].

When doing performance testing on the highest tested load, 800 simultaneous requests, we experienced that the backend service itself was getting into the problems of responding to the load. What could make the results of the test more relevant would be to prevent a backend service from being a bottleneck by providing it more resources, load-balance or autoscale it. This would allow us to further scale up the number of simultaneous connections in the test and thus potentially find out where the limitations of how many simultaneous requests the API Gateways instances could handle lay and verify if this is in alignment with the already published performance comparison benchmarks.

Chapter 6

Conclusion

The goal of this thesis was to explore the field of API gateways and Microservices. In the Introduction section 1, we started by pointing to a trend of growing API Economy and how it will be increasingly important in future for services to provide and manage APIs. We defined a research question and explained a methodology of how are we going to evaluate API gateways in the context of microservice applications. In the Extended State of the Art section 2, we analyzed the contemporary literature, use cases and conducted an interview to determine the functional requirements of the API Gateways, presented in section 2.1.4. In the remaining part of the ESOTA, we provided a deep insight into the comparison of monolithic and microservice architectures and the design principles and best practices of microservice applications and REST APIs, wrapping up the chapter with a description of the Backend-for-Frontend pattern and a very brief introduction of GraphQL. The Implementation chapter 3 started with the selection of the API Gateways we are going to try out, the reasons why we chose them and their brief description. The remaining part of the chapter was a technical description of how we had to enhance the upstream microservice application, detail how to deploy the gateways into the Kubernetes cluster and last but not least, a description of the performance test suite we created. In the Results 4 and Discussion 5 chapters, we presented the obtained data, interpreted and discussed them in accordance with the research question and suggested possible future improvements.

Bibliography

[1] Mike Amundsen. *Collection JSON - Document Format*. May 2011. URL: http://amundsen.com/media-types/collection/format/.

[2] *Apache JMeter*. URL: https://jmeter.apache.org/.

[3] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. DOI: 10.17487/RFC3986. URL: https://rfc-editor.org/rfc/rfc3986.txt.

[4] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. Mar. 2014. DOI: 10.17487/RFC7159. URL: https://rfc-editor.org/rfc/rfc7159.txt.

[5] Phil Calçado. *The Back-end for Front-end Pattern (BFF)*. 2015. URL: https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html.

[6] Google Cloud. *The State of API Economy 2021 Report*. 2021. URL: https://pages.apigee.com/api-economy-report-register/.

[7] *Comparison of KrakenD vs other products in the market (Benchmark)*. Oct. 2016. URL: https://www.krakend.io/docs/benchmarks/api-gateway-benchmark/.

[8] Liam Crilly. *Deploying NGINX as an API Gateway, Part 1*. Feb. 2021. URL: https://www.nginx.com/blog/deploying-nginx-plus-as-an-api-gateway-part-1/.

[9] Liam Crilly. *Deploying Nginx Plus as an API Gateway*. 2015. URL: https://www.nginx.com/resources/library/nginx-api-gateway-deployment/.

[10] DapperDox. *DapperDox/dapperdox: Beautiful, integrated, OpenAPI documentation.* URL: https://github.com/DapperDox/dapperdox.

[11] Lisa M. Dusseault and James M. Snell. *PATCH Method for HTTP*. RFC 5789. Mar. 2010. DOI: 10.17487/RFC5789. URL: https://rfc-editor.org/rfc/rfc5789.txt.

[12] Facebook. *GraphQL: A data query language*. 2015. URL: https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/.

[13] Faren. *KONG - The Microservice API Gateway*. Jan. 2019. URL: https://medium.com/@far3ns/kong-the-microservice-api-gateway-526c4ca0cfa6.

[14] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. June 2014. DOI: 10.17487/RFC7231. URL: https://rfc-editor.org/rfc/rfc7231.txt.

[15] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[16] Martin Fowler. *Circuit Breaker*. 2014. URL: https://martinfowler.com/bliki/CircuitBreaker.html.

[17] Martin Fowler. “MonolithFirst”. In: (2015). URL: https://martinfowler.com/bliki/MonolithFirst.html.

[18] Martin Fowler. *Refactoring : improving the design of existing code*. Addison-Wesley, 1999. ISBN: 978-0201485677.

[19] Sanjay Gadge and Vijaya Kotwani. *Microservice Architecture: API Gateway Considerations*. 2017. URL: https://www.globallogic.com/wp-content/uploads/2017/08/Microservice-Architecture-API-Gateway-Considerations.pdf.

[20] Gunar Gessner. *GraphQL Federation vs Stitching*. Nov. 2019. URL: https://medium.com/@gunar/graphql-federation-vs-stitching-7a7bd3587aa0.

[21] Marc-André Giroux. *Production Ready GraphQL*. 2020. URL: https://book.productionreadygraphql.com/.

[22] *GraphQL Stitching versus Federation*. URL: https://seblog.nl/2019/06/04/2/graphql-stitching-versus-federation.

[23] *GraphQL: Schemas and Types*. URL: https://graphql.org/learn/schema/.

[24] *GrapQL Tools - Combining schemas*. URL: https://www.graphql-tools.com/docs/schema-stitching/stitch-combining-schemas.

[25] *Introduction to Apollo Federation*. URL: https://www.apollographql.com/docs/federation/.

[26] *JSON Schema is a vocabulary that allows you to annotate and validate JSON documents*. URL: https://json-schema.org/.

[27] *JSON:API*. URL: https://jsonapi.org/.

[28] *jsonld.js*. URL: https://json-ld.org/.

[29] Mike Kelly. *JSON Hypertext Application Language*. Internet-Draft. Work in Progress. Internet Engineering Task Force, Apr. 2014. 11 pp. URL: https://datatracker.ietf.org/doc/html/draft-kelly-json-hal-06.

[30] *Kong API Gateway*. July 2021. URL: https://konghq.com/kong/.

[31] KongHQ. *Kong Gateway (OSS) - A lightweight open-source API gateway*. URL: https://docs.konghq.com/gateway-oss/.

[32] KongHQ. *Kong Plugin Hub - Extend Kong Konnect with powerful plugins and easy integrations*. URL: https://docs.konghq.com/hub/.

[33] *KrakenD - Case-studies*. URL: https://www.krakend.io/case-study/.

[34] *KrakenD - Open source API Gateway*. URL: https://www.krakend.io/.

[35] *KrakenD Enterprise*. Oct. 2018. URL: https://www.krakend.io/enterprise/.

[36] *KrakenD Enterprise Edition (EE) and Community Edition (CE) comparison*. URL: https://www.krakend.io/assets/KrakenD-EE-vs-CE--feature-matrix.pdf.

[37] *KrakenD framework becomes a Linux Foundation project*. May 2021. URL: https://www.krakend.io/blog/krakend-framework-joins-the-linux-foundation/.

[38] *KrakenDesigner*. Oct. 2016. URL: https://www.krakend.io/designer/.

[39] Albert Lombarte. “An API Gateway is not the new Unicorn | by Albert Lombarte | DevOps Faith | Medium”. In: (2018).
. URL: https://medium.com/devops-faith/an-api-gateway-is-not-the-new-unicorn-303a3863f2a6.

[40] *Manage Secrets and Protect Sensitive Data*. URL: https://www.vaultproject.io/.

[41] Fowler Martin. “Circuit Breaker pattern”. In: (2014). URL: https://martinfowler.com/bliki/CircuitBreaker.html.

[42] Larry M Masinter. *Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)*. RFC 2324. Apr. 1998. DOI: 10.17487/RFC2324. URL: https://rfc-editor.org/rfc/rfc2324.txt.

[43] *Microservices Reference Architecture*. Nov. 2019. URL: https://www.nginx.com/resources/library/microservices-reference-architecture/.

[44] Fabrizio Montesi and Janine Weber. *Circuit Breakers, Discovery, and API Gateways in Microservices*. Provides comparison of 3 patterns found in microservice world - Circut Breakers, Discovery and API Gateways
. 2016. URL: https://arxiv.org/pdf/1609.05830.pdf.

[45] Lauren Murphy et al. “Preliminary Analysis of REST API Style Guidelines”. In: *Ann Arbor* 1001 (2017), p. 48109.

[46] Netflix. *Embracing the Differences : Inside the Netflix API Redesign*. 2012. URL: https://netflixtechblog.com/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d.

[47] *Nginx - Compare Models*. July 2021. URL: https://www.nginx.com/products/nginx/compare-models.

[48] *Nginx - High Performance Load Balancer, Web Server, Reverse Proxy*. July 2021. URL: https://www.nginx.com/.

[49] Henrik Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://rfc-editor.org/rfc/rfc2616.txt.

[50] Michael Nygard. *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007. ISBN: 978-0978739218.

[51] Pantsel. *pantsel/konga: More than just another GUI to Kong Admin API*. URL: https://github.com/pantsel/konga.

[52] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. “Restful web services vs." big"'web services: making the right architectural decision”. In: *Proceedings of the 17th international conference on World Wide Web*. 2008, pp. 805–814.

[53] Lukasz Plotnicki. *BFF @ SoundCloud*. 2015.

[54] Tom Preston-Werner. *Semantic Versioning 2.0.0*. URL: https://semver.org/.

[55] Redocly. *Redocly/redoc: OpenAPI/Swagger-generated API Reference Documentation*. URL: https://github.com/Redocly/redoc.

[56] Mark Richards. *Microservices vs. Service-Oriented Architecture*. 2015. ISBN: 978-1-491-95242-9.

[57] Leonard Richardson et al. *RESTful Web APIs: Services for a Changing World*. " O’Reilly Media, Inc.", 2013.

[58] Swagger-Api. *swagger-api/swagger-ui: Swagger UI is a collection of HTML, JavaScript, and CSS assets that dynamically generate beautiful documentation from a Swagger-compliant API*. URL: https://github.com/swagger-api/swagger-ui.

[59] *The Lura Project*. URL: https://luraproject.org/.

[60] ThoughtWorks. *Overambitious API Gateways*. 2015. URL: https://www.thoughtworks.com/radar/platforms/overambitious-api-gateways.

[61] *Tyk - API Gateway, API Management Platform, Portal Analytics*. Aug. 2021. URL: https://tyk.io/.

[62] Rory Ward and Betsy Beyer. “BeyondCorp: A New Approach to Enterprise Security”. In: *;login:* Vol. 39, No. 6 (2014), pp. 6–11. URL: https://research.google/pubs/pub43231/.

[63] Adam Wiggins. “The Twelve-Factor App”. In: (2017). URL: https://12factor.net/.

6.1 Appendix

The code including the enhanced demo microservice application, gateway's configuration files and performance test suite is available in the public repository on this URL:

"`https://github.com/trebi/thesis-impl`"