University of Minho
School of Engineering

Hugo Manuel Coelho de Oliveira

Patterns and development strategies used on a microservices architecture

July 2021

University of Minho
School of Engineering
Department of Informatics

Hugo Manuel Coelho de Oliveira

Patterns and development strategies used on a microservices architecture

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
António Nestor Ribeiro

July 2021

Copyright Notice

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of Universidade do Minho.

Braga, 27th July 2021

Abstract

Microservices are a modern architecture style that divides a single application into small, independently deployable services, each running in its own process and communicating through lightweight mechanisms. However, there is still a lack of research on the design and development of microservices applications.

The development of applications using microservice-based architectures requires a variety of essential factors that must be kept in mind to achieve good and future proof results.

Given the growing demand for scaling applications and the growth of cloud infrastructures, microservices emerged as one of the most prominent architectural advancements in recent years. They are still in their early stages of integration, and for that reason this architecture style has yet to be widely studied.

With that in mind, this dissertation aims to close this gap by providing the key elements that should be considered when designing and building solutions based in microservices. It begins by researching and studying these architectures and finishes with a implementation of microservices based on a case study.

Keywords: Microservices, Software Architectures, Patterns, Strategies, Scalability

Resumo

Os microsserviços emergiram recentemente como um estilo arquitetural moderno que divide uma única aplicação em vários serviços de forma independente, cada um executando o seu próprio processo e comunicando através de mecanismos simples. No entanto, existem ainda falhas sobre o estudo e desenvolvimento de aplicações baseadas em microsserviços.

O desenvolvimento destas aplicações requer uma variedade de fatores essenciais que devem ser tidos em conta para que seja possível obter bons resultados a longo termo.

Com a necessidade de escalar aplicações e com o crescimento de infraestruturas na *cloud*, os microsserviços surgem como um dos avanços arquiteturais mais importantes nos últimos anos. Ainda se encontram nas fases inicias de integração e, por essa razão, este estilo arquitetural necessita de ser amplamente estudado.

Neste sentido, o objectivo desta dissertação é colmatar esta lacuna, através do estudo dos elementos chave que devem ser considerados durante a concepção e construção de soluções baseadas em microsserviços. Inicalmente procede-se à pesquisa e estudo destas arquiteturas e no fim efetua-se a implementação de uma arquitetura de microsserviços baseada num caso de estudo.

Palavras-chave: Microsserviços, Arquiteturas de Software, Padrões, Estratégias, Escalabilidade

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 19

ACL Access-Control List. 17, 59

API Application Programming Interfaces. 12–17, 21, 23, 24, 33, 34, 37, 39, 46–48, 57–64

AWS Amazon Web Services. 38–43, 47, 64, 65

BFF Backend For Frontend. 15, 16, 61

CSV Comma-separated Values. 27, 28, 32

E2E End-to-end. 8

GPS Global Positioning System. 31, 32, 73

HTTP Hypertext Transfer Protocol. 6, 22, 23, 36, 37, 40, 46, 50, 57–59, 61, 64

IDE Integrated Development Environments. 8

IDL Interface Definition Language. 28

IoT Internet of Things. 5, 6, 31

IT Information Technology. 1, 63

JSON JavaScript Object Notation. 18, 27, 28, 45, 47

JWK JSON Web Key. 47, 55, 58, 59

JWKS JSON Web Key Set. 47

JWT JSON Web Token. 18, 47, 48, 58–60, 64

MA Monolithic architecture. 7–9, 11

MSA Microservices architecture. 9, 10, 12–14, 16, 23, 25, 31, 36, 38, 61, 63

ORM Object–Relational Mapping. 38

RBAC Role Based Access Control. 17, 52, 62, 64

REST Representational State Transfer. 10, 13, 22–24, 37, 46–48, 61, 62, 64

RFC Request for Comments. 18, 41, 47, 64

RPC Remote Procedure Call. 23

SDK Software Development Kit. 41

SMTP Simple Mail Transfer Protocol. 50

SOA Service-Oriented Architecture. 10

SOAP Simple Object Access Protocol. 10

SQS Simple Queue Service. 39–44, 65

SSL Secure Sockets Layer. 34

TST Taxi Service Trajectory. 72

TTL Time To Live. 56

UI User Interface. 7, 15

URL Uniform Resource Locator. 42, 58

UUID Universally Unique Identifiers. 41, 46, 47, 49, 53

VCS Version Control System. 45

WSGI Web Server Gateway Interface. 46

XML eXtensible Markup Language. 27, 28

Introduction 1

The world has become increasingly digital and technology has been spread into our everyday lives. Digital devices such as mobile phones, tablets, or laptops are being used to collect different types of data. The exchange of information has been expanding in all fields of science and engineering and it was propelled by the rapid evolution of networking, data storage, and the ability to collect large amounts of data [54]. Nowadays, better ways of building systems have been found. New technologies are being adopted and new architectures of Information Technology (IT) systems are being designed [54, 35].

There are several challenges in how to collect, handle and process all this information. In many cases, the greatest challenge is to learn how to extract useful information or knowledge to be considered in the future [54]. Therefore, it is necessary to know and apply the right patterns to build an architecture that is able to collect and process information.

Driven by customer requirements, many companies need to develop new features to meet their needs. Such new features need to be introduced quickly in order to keep their customers happier and loyal to their applications. For that reason, well-known companies are breaking their monolithic applications in a group of independent services, known as microservices [34, 35].

Microservices emerged as one of the most popular architectural developments in recent years, given the increasing need for scale applications as well as the rise of cloud infrastructures [8]. This type of architecture allows the continuous development of applications, as the developers are able to work independently and work with different technologies that promote an agile and flexible development of new features and more scalable, modular, and versatile systems [34, 8].

When it comes to building an architecture, selecting the right patterns is crucial. Some patterns have significant implications for the final system. As a result, understanding the advantages and disadvantages of architectural patterns may lead to a stronger and future proof architecture design.

With this in mind, the aim of this dissertation is to research the strategies used to build a microservice-based architecture. The development of microservices requires a variety of important factors that must be kept in mind in order to achieve good results. It is important to consider the expertise and understanding of other architectures and specialists in this field. Ignoring the common patterns can have long-term consequences, and ultimately it may become difficult to manage the individual services in the architecture. Thus, the goals of this dissertation are: (1) to explore and study the design of microservices architectures, their advantages and disadvantages and the most common patterns used to build this architectural style, including the study of communication, persistence and caching mechanisms; and (2) to expose and describe the development of a microservices-based architecture that meets previous study, giving some ideas and clues to help others understanding how to apply and use the most common patterns of this architectural style. This has a special focus on giving a general architecture example that can be applied on different use cases and should be supported by the best practices and patterns studied to help other engineers and companies building scalable systems.

To help with the creation of a microservice-based architecture, a case study was used. It is about a taxi business that installed sensors on its vehicles to capture real-time data, allowing the company to dispatch taxis efficiently, identify better routes, and allow taxi sharing. Based on the data gathered and further studies, a microservice architecture was built around the context of a taxi management platform, enabling the conceptualisation of a microservice-based architecture.

1.1 Document structure

The present dissertation is organised as follows:

- **Chapter 2: State of the art**

  This chapter shows the review of software architectures with an emphasis on microservices and the common patterns used to develop systems using this architectural style. It shows a comprehensive study of microservices-based architectures and the key elements that should be considered in the creation of these solutions.

- **Chapter 3: The problem and its challenges**

  In this chapter it is described the problem and its challenges, as well as the design of the proposed architecture based on a case study. The case study is introduced to help evaluating and validating the research that was previously conducted and to assist those introducing an microservices-based architecture.

- **Chapter 4: Development**

  Throughout this chapter, it is highlighted the technologies and the decisions during the development of the proposed architecture. It is also described the patterns that were applied to enabled a careful and well-structured architecture.

- **Chapter 5: Conclusion**

  The last chapter gives a final review and conclusions of the work developed as well as the dissertation's future work.

State of the art 2

This chapter starts with the review of a mapping study in microservice architectures and how platform was designed using this architectural style. Then it is introduced the concept of a software architecture and the distinction between a monolithic and a microservices architecture, which enables a clear understanding of how this architectural style is being adopted. It will then be presented the study of microservice architectures patterns and some solutions that are used to allow the development of modern and scalable applications.

2.1 A systematic mapping study in microservice architecture

The evolution of hardware and software and the increasing demand of having applications accesible trough the network has introduced new software development methods and software architectural styles. Recently, microservices architecture has emerged to address the maintenance and scalability demands of online services.

The available studies on microservices architectures and the relevant architectural challenges that microservices systems face are being discussed. Multiple researchers were able to explore all the published articles and studies that highlighted the gaps in microservices research and the solutions and initiatives made so far to address some of the challenges related to microservice systems.

Systematic mapping studies are in-depth analyses of specific research questions in an field or subject with the aim of identifying gaps in the published literature. Three key research questions were addressed based on the published articles and studies. The firs question focuses on the architectural challenges that microservice systems face. The second investigates which architectural diagrams or models are used to represent microservices architectures. The last research question identifies the possible quality attributes associated with microservices.

Leading software consulting companies have considered the microservices approach to be an attractive architecture that helps teams and software organizations to be more productive and create profitable software applications more often. This architectural style has also been tried and tested by many companies outside of the traditional software industry, and it has proven to be extremely beneficial. Microservices are also considered as an appropriate architecture for systems deployed on cloud infrastractures, as it can take advantage of elasticity and on-demand provisioning.

It has been identified some keywords for microservices challenges, including: communication strategies, services discovery, performance, fault-tolerance, security, tracing and logging, and deployment operations.

The observed research results indicate that microservices architectural style is still very new. Most of the solutions published in papers were found to be either proposals or validations. Apart from that, the literature shows microservices offer scalability, reusability, maintainability, fast and agile development and high performance.

2.2 Designing a Smart City IoT Platform with Microservice Architecture

The Internet of Things (IoT) is being used in a variety of applications and is considered one of the main enablers of the Smart City vision. Despite the great efforts adopting web standards and cloud computing technologies, building scalable Smart City IoT platforms remains challenging [25].

A platform named DIMMER is designed and presented to explain how microservices can be applied when designing a Smart City IoT platform capable of increasing the energy efficiency of a city. The main vision is to increase the quality of services offered to citizens while reducing the operational costs. To provide access to data and services, a large-scale IoT system is required. Moreover, it is important to consider the platform progression, which means the designed platform needs to be able to support new standards and services in the future.

One of the main challenges that current web and cloud technologies are able to address is the design of large-scale distributed systems that adapt as the underlying technology and requirements evolve. The simplicity of interfaces, the loose coupling of individual components and the elasticity provided by the cloud build the foundation of modern distributed applications. Microservices emerged as a pattern from a real-world experience of building distributed applications, and for that reason it does not have a formal definition.

The DIMMER Smart City platform architecture is present on Figure 2.1.

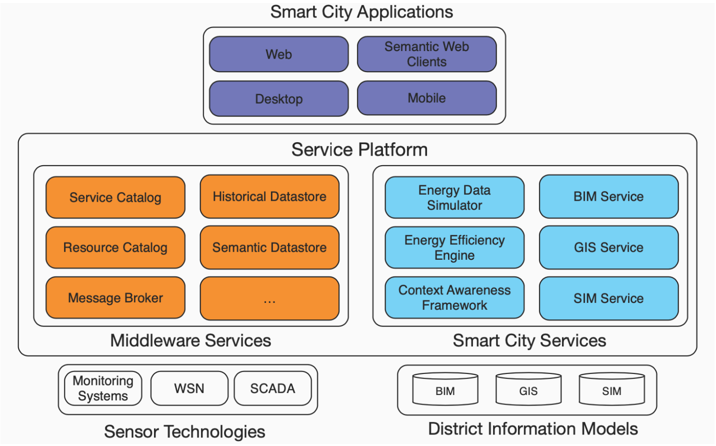

Figure 2.1: DIMMER platform architecture. Source [25]

The platform contains IoT services gathering and processing data from different sensor systems (Middleware Services), as well as other services for Smart City applications (Smart City Services). Then, the applications using the platform represent a large variety of web, desktop and mobile applications.

The Middlerware Services use both lightweight queries over Hypertext Transfer Protocol (HTTP) and SPARQL to search and discovery of IoT devices. The DIMMER platform also uses a decentralised data management approach of microservices to manage the IoT devices metadata, storing it on their own storage backend, i.e. timeseries databases and document-based database. The services also use a message broker for publish/subscribe communications. Each services is independently deployable, which means all of them can be scaled and updated separately.

Applications and services accessing the IoT data stored separately need to preform several queries to different microservices, which leads to a more complex client implementation. For that reason, an application gateway was implemented. The gateway provides a convenient high-level interfaces for different type of clients.

The Smart City Services were built on top of IoT middleware services and they implement the core functionality of the platform. These services include services that manage domain-specific data models, services that provide optimisation algorithms and strategies for energy consumption, services that simulate IoT data, etc.

The platform was built with interdisciplinary and international team, which aligned with microservices approach, allowed them to work highly independently, i.e. each team concentrated on their work while maintaining the overall system compatibility. This was also possible after defining the services interfaces. Moreover, each team was able to choose any technology to implement their own services.

Microservices make it easier to design and implement individual services, but they also add the complexities of distributed systems. Currently, the eventual consistency is considered a reasonable trade-off for the grained benefits that microservices brings.

2.3 Software architecture

Every software system has a software architecture which results as a set of design decisions about the system. All design decisions can potentially be made at any time during the system's lifetime and they contribute to a high-level structure of the systems - $e.g$ the components, connectors, configurations, system's deployments and so on. It is important to mention that a good architecture design is well correlated with good software quality [47] and it plays a vital role towards improving the software process associated with large and complex information systems [40].

In contrast to design, the term architecture is used to invoke notions of codification, abstraction, standards of software architects and style [40].

Any software architecture has to be at the very heart of design and development of the product. It must be in the foreground more than everything, so that the growth of the system and its long-term evolution can be effective and efficient. A design of any system architecture should be compared with the design of a large building, which requires to be considered before construction [47].

On the following sections, it is introduced the monolithic architecture and its benefits and drawbacks. Then the concept of microservices and why they are getting the attention of companies is well described.

2.4 Monolithic architecture

The rapid growth of the internet has significantly increased the number of requests for on-demand services, and the growing complexity of software systems makes them extremely difficult to handle. Traditional enterprise applications are referred to Monolithic Architecture (MA) and they are typically built in three parts: a client-side user interface (UI), a server-side component running the application logic, and a database component hosting the application data [9, 28].

A MA can be defined as a single, autonomous unit [28]. It has a single large codebase that provides multiple services using various interfaces [51]. Even if this solution contains several services or components internally, this architectural style consists of deploying a system as a single logical executable [28].

2.4.1 Advantages and disadvantages

Some of the benefits of a monolithic solution are presented bellow:

- applications are easier to develop and at the beginning of a project it is much easier to go with this architecture. In particular, the integrated development environments (IDEs) they are designed to help building, deploying, debugging single application, since the codebase is all together [44, 28];
- the communication between different modules is implied in-memory, since all services are a part of the same systems [24];
- it is easier to deploy once is a single package of code. The deployment times can vary, but copying a single archive to one directory is generally straightforward [44, 28, 34];
- monitoring the application is easier because there is a single unit of code [9];
- testing the entire application is simpler by using end-to-end (E2E) tests, validating the integration between the various components of the application [44]. There is no need to wait for extra dependencies or components as everything is already deployed [28];
- scaling horizontally is also simpler as it is possible to run multiple copies of the application behind a load balancer [35, 34, 44].

However, no matter how modular the MA is, it will eventually start to break and few problems will be raised. It is listed below some of the disadvantages of using this architectural style:

- the developers need to deal with situations where the number of users exceeds the server's capacity [9];
- a lot of effort is usually needed to understand and break up the application to something more useful when the application gets bigger. Moreover, it is also difficult to add new developers or to replace old team members [34, 35];
- throughout the year the applications codebase becomes large, complex and the quality of the code starts to decline. The product delivery rate slows and the frameworks used start to become obsolete [34];

- changing the technology stack is very difficult since all components will be sticking to the technology chosen at the beginning (*.e.g* change the the development framework or adopt a newer technology) [34]. This also means the team must use the same programming language, persistence stores, messaging systems and other tools to keep the developers aligned [28];
- the components themselves soon become tightly connected, preventing developers from working independently. There are fewer opportunities for developers to step up and own some component [35, 34];
- new modifications requires rebuilding and redeploying a new version of the application [35, 34, 28, 24]. These changes must guarantee that all other components and services continue to work [51];
- different application components may have different needs (CPU bound, memory bound, *et al*), which means that each component cannot be scaled independently [35];
- the architecture is not very resilient - if a component fails everything stops working [35, 51];
- even if the data model has a strong structure, this style ties the entire data management to the application domain and does not scale horizontally - if several instances run at same time and an increase in data volume is considered, each instance will access the same data source, leading to an increase in the volume of transactions [34, 7, 28].

2.5 Microservices architecture

The concept of microservices architecture (MSA) is being used to describe some of the actual architectures that are being adopted by well-known organisations such as Google, Amazon and Netflix. They have found that by using this type of architectures, they can deliver software faster and embrace newer technologies [35]. Different techniques, strategies and technologies were used to overcome the limitations and issues of a MA [51].

Lewis and Fowler define the microservice architectural style as *an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API* [27]. Therefore, the approach to the design of microservices aims to build an application as a set of small independent services where each service could be deployed on a separate platform and use an entirely independently technical stack [34, 5].

A microservice should only concern on a very specific functionality. By keeping the service focused on an explicit business boundary, it is possible to avoid the temptation for making it too large. The work micro is related with the business scope to which it is related and not the size or complexity of the service [35, 33].

2.5.1 Comparing to SOA

There is always some debate around Service-Oriented Architecture (SOA) and MSA. Both are architectural styles that structure a system as a set of services that work together to perform some set of tasks and communicate with one another via well defined interfaces [44, 35].

SOA emerged as a solution to the problems presented by monolithic architecures. It's promotes software reusability, i.e. multiple end-user applications may use the same services. It aims to make it easy to manage and rewrite applications by replacing one service with another without no one noticing, as long as the semantics of the service do not change dramatically [35].

Despite the efforts around SOA, there is a lack of consensus on how to do SOA well. Many of the concerns are related to communication protocols (e.g. Simple Object Access Protocol (SOAP)), middlewares, and a lack of awareness on how to break the system in different bits [46, 35].

MSA emerged from real-world use after taking a better understand of systems and architectures to do SOA well [35]. Some companies using microservices have already implemented SOA, which means they have embraced the concept of modularity and different communication mechanisms [33].

However, there are some variations between SOA and microservices: SOA use heavyweight technology like SOAP and generally connect services using an Enterprise Service Bus that contains business and message-processing logic to integrate the services. MSA, on other hand, use lightweight and open source technologies. The services communicate using message brokers or lightweight protocols like REST or gRPC (discussed later in Section 2.8) [44]; SOA typically share databases. In MSA each service owns a database (discussed later in Section 2.7) [35, 44]; and SOA is typically used to integrate large and complex monolithic applications. Even if in MSA the services may not be always tiny, they are almost always much smaller [44].

2.5.2 Advantages and disadvantages

Moving on from some of the previous ideas, it is described bellow some of the advantages of this architecture in order to better understand the concept of microservices and why they are getting the developers attention:

- each service can be written in different programming languages and use different data models, which contribute to technology heterogeneity. Specific technologies might be better to achieve the necessary performance levels [35, 24];

- it allows the integration of new technologies during development and maintenance in a more efficient way. Sometimes the right tool must be used for the right job [44, 35]. There are several places to test out a new piece of technology with multiple services. So, there is the opportunity in selecting a service with lower risk and test the technology there [35];

- each service can easily be scaled, which allows the system to become much resilience to partial failure [44]. It is possible to run other parts of the system on smaller and less powerful hardware by replicating the resources that need scaling. For the services that need it, it might also be possible to scale on demand [35];

- each service can run on different platforms with different hardware specifications [35];

- having multiple services can increase resilience. If one component fails, but that failure doesn't cascade, the problem can be isolated and the system remains operational. Systems can be created to handle the complete failure of services [35];

- deploying a given service independently of the others could result in faster implementation of new features. This also means that if a problem occurs a fast rollback can easily be executed [35];

- when writing small and individual services, it will be much easier to handle the expense of replacing them with a better implementation, or even removing them entirely [35]. It is easier to manage smaller code pieces than to update robust monolithic source code [24];

- there is a more flexible development approach by allowing smaller teams to develop and maintain the services, allowing them to move forward independently and much faster than they would in a MA. This will also provide an agile development of new features and independent releases of these services [44, 35]. Service's code has higher maintainability and understandability, since each team can track independently each service and react to any anomalies that can occur [5].

The developers may face some challenges when designing an architecture of microservices. It is described bellow some of them:

- compared to monolithic systems the learning curve isn't very high. Nonetheless, more effort is required to create the basic architecture. In systems with a small number of users, the monolithic solution may be quicker to develop and setup [50, 1, 49];
- as the number of services rises, it becomes necessary to automate things to deploy services. *DevOps* are typically an essential part of microservice architectures and they should be part of the development team [5, 1, 44];
- managing multiple services will require a strong coordination between team of developers [34, 44];
- the deployment might be complex as different services need to be managed [34]. Selecting the right platform to deploy a service is critical and may be the key for scaling microservices [1]. The container orchestration tools are generally used to make the deployment more trivial [5];
- the integration of automated tests is extremely important to identify service faults. However, due to the large number of components they can be very complex and may be more difficult to determine which functionality has broken since they are in a distributed system [34];
- fault tolerance mechanisms should be taken into account to provide some means of recovering or to stop the system failures [44];
- defining a proper inter-service communication strategy can be a problem. It usually requires the proper protocol and a good design of the Application Programming Interface (API) [34, 44];
- issues related to the network are common. Network latency or low bandwidth can increase the communication between two services. To prevent errors the network should be reliable [44];
- some architectures demand distributed data management. Therefore, distributed transactions are therefore needed which lead to very complex implementation [34, 44].

There is a significant increase in the use of microservices architecture. However, there seems to be a lack of information on how to build a system based on this style. Such knowledge will help decision-making on migrating applications to microservices or even begin to develop a full application [50, 49].

Design patterns exist to address such common challenges and provide solutions to make the architecture more efficient and more stable.

The common patterns used to design and build a MSA are explored in the following sections.

2.6 API gateway

Usually monolithic application expose a single interface to serve external services or mobile/web clients. However, that can become quite intensive and fairly inefficient when dealing with multiple services, since there is the need of requesting each service directly [35, 44].

The implementation of a service that is able to act as the entry point into the system from the outside world is common in the context of microservices architectures [44, 33].

A standard pattern observed in MSA implementation is securing the services interfaces with an API gateway [33], as it is shown in Figure 2.2.

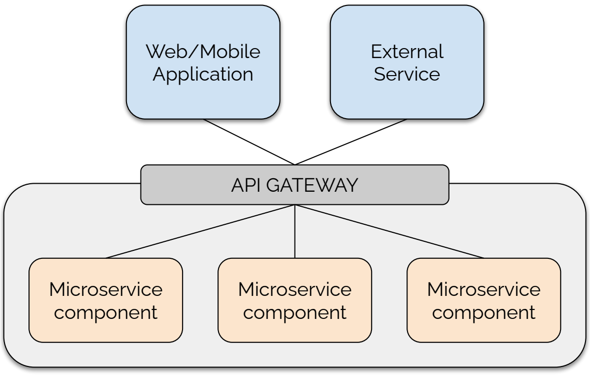

Figure 2.2: API gateway - using a single interface to handle requests from clients.

The API gateway pattern provide additional features required by microservices. This pattern emerged as a solution to: routing the requests to the appropriate microservice. All requests from external clients first go to the API gateway, which route them to the corresponding service [49, 44, 56]; securing the internal services. Certain services are deemed internal and the API gateway can provide some security to prevent external access to any microservice [33, 44]; invoking multiple services and aggregate the results to serve clients [49, 44]; translating protocols between clients and services. The gateway might provide a REST API to clients, even though the services use a mixture of protocols internally [44, 33]; sharing logic implementations like authentication and rate-limiters [49, 44]; and monitoring, handling static response and load balancing [49].

An API gateway could provide a single ”one size fits all” API. However, different clients often have different requirements, i.e. a third-party service might require the full details of an order, whereas a mobile client only needs a subset of the data. Therefore, an API gateway might have the capacity to provide each client with its own API, specifically designed to meet each requirements and optimising and encapsulating the communication between clients and services [44]. This property allows microservices updates without affecting the clients, as the only update required is on the API gateway [49]. This pattern is similar to the Facade pattern from object oriented design [44]. Another important feature of this pattern is that for each client only sees the information that is needed, which simplifies the development of end-user applications without adding the complexity of exposing and parsing useless information [49]. However, adopting a client-specific API inside an API gateway might require some considerations: if a separate team is responsible for the entire API gateway any change on the underline microservices API will require teams to submit a request and wait until the changes were approved and applied [49]; and if each team is responsible to the development of their own API module inside the API gateway there will be the problem of several teams contributing to the same code base, which is contradictory to the philosophy of microservice architecture: *"if you build it, you own it"* [49].

Based on these considerations, a new pattern called Backend for frontends take this concept of an API per client even further, by defining a separate API gateway per each client based on different platforms or distinct business domains [42, 49]. This patter is later defined on Section 2.6.1.

The API gateway pattern is indeed a common solution in MSA. It encapsulates internal structure of the system and it is able to provide client-specific APIs, which reduces the number of requests between client and services. However, the pattern has some drawbacks:

- the gateway is the only entry point for all requests. If not properly planned, this could be a potential bottleneck [49, 44];
- it may eventually get bloated, since it requires implementation of several interfaces for each service [42] [49];
- reuse of the APIs must be carefully considered. Keeping track of cases where different types of clients use the same API to be able to make changes that work for both is required [49];
- it requires the development, deployment and management of one more component of the entire architecture [44];
- it increases the response time due the additional network hop through the gateway [44];
- when the number of microservices in a network increases, there will be a need for a more effective and scalable routing mechanism for routing traffic through service APIs and better configuration management to dynamically configure and implement system changes [49].

2.6.1 Backends for frontends

This pattern is a variation of the API gateway pattern, where a separate API gateway is implemented for each type of clients. This also emerged has a solution when different teams end up managing the same gateway and changing the same code base inside the API gateway [44, 35].

A general-purpose API gateway is perfectly fine when clients have the same behaviour and make the same backend calls. However, each platform has different needs and different purposes [36]. Mobile devices might display just a few information and it's best to have smaller payloads and less requests for the APIs. On the other hand, web application might need an interface that provides all the data needed [36].

As Lukasz Plotnicki mentioned in [41], the developers of SoundCloud application didn't take into consideration the existence of multiple platforms while they were developing a monolithic API. Furthermore, when a new API endpoint was needed, the frontend team needed to convince the backend team to develop it and, when a team needed to change an existing endpoint they needed to make sure that the changes would not break any of their existing applications [41].

Once the apps begin to grow and new features begin to be introduced or modified, the API gateway becomes too dynamic and is eventually managed by separate teams. This leads the teams to throw everything into a giant layer, losing isolation [35, 41].

The solution to these problems has become with the idea of having different backends for different frontends, which ends up being called Backend for Frontend (BFF) pattern. Figure 2.3 shows one backend API per client context, instead of a general purpose API [35].

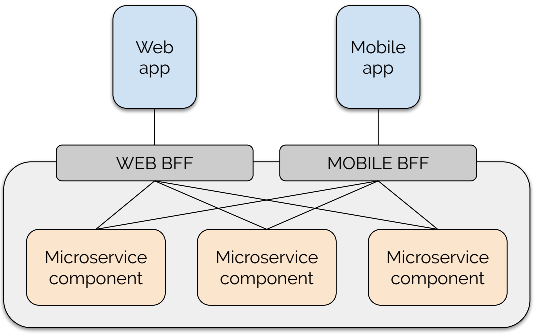

Figure 2.3: Backends for frontends pattern.

Each BFF should be coupled to specific context and should be maintained by the same team that owns the services, allowing each team to easily adapt the API to the user interface (UI) requirements [35, 36]. For example, if there is one application for both iOS and Android that share the same user experiences, a single BFF may exist. However, it might be necessary to split a single BFF in multiples BFFs when each application starts having experiences that diverge greatly. Each company should decide how many BFFs are needed [36].

The main advantage over the API gateway is the potential of allowing each team to independently build and deploy its own gateway. This will also improves their BFF's reliability as each API is separated from each other - if an error happens, the other gateway is probably not affected. This pattern also allow each BFF to scale independently [44].

Both API gateway and BFF patterns may cause some problems, i.e. each gateway can start taking a lot of business logic. This should remain in the services themselves and each gateway should only address specific behaviours to provide a particular user experience and context [36].

2.6.2 Implementing security

In order to develop secure applications, a wide variety of security issues must be addressed: including hardware security, data encryption, authentication and authorisation. These issues are the same regardless of using a monolithic or microservices architecture [44].

In a monolithic architecture, a user logs in with their passwords, the service verifies them, and returns a session token that can be used for future requests. However, on MSA, the user needs to interact with multiple services. Implementing security in a microservice architecture brings another challenges.

Before describing how security can be implemented in MSA, it is important to mention that each service must implement some aspects of security. For instance, a service must make sure each user only access their information. This type of security is based on each application use case [44].

As it was discussed before, the API gateway serves as a intermediary between external clients and microservices, offering a secure network environment that allows for private data exchange between them. In this section it is discussed how authentication and authorisation is handled a microservice architecture using the API gateway.

Let's start by exploring how to handle **authentication** in MSA. The authentication refers to verify ”who you are” and usually credentials are required. There are different ways to handle authentication:

- one option is for the individual services to authenticate the user. The problem is that this approach permits unauthenticated requests to enter in the internal network, and each service needs to correctly implement security in all services, increasing security vulnerabilities. Another problem is related how different clients interact with services: some may use basic authentication to log in and others might first log in and then use a generated session token on each request [44];

- another option is for the API gateway to authenticate a request before forwarding it to the internal services. The big advantage is to have one centralised place to get authentication and security right. It also allows the service to be less complex, removing security concerns from them [44, 56];

- if the previous options doesn't fit the system requirements, it is also possible to use a single service to authenticate the user. In that way a service can validate credentials and generate a session token to be used on the following requests to other services. This token can then be validated using the API gateway, so only requests with valid tokens are allowed [56].

Authentication alone is insufficient to implement security in microservices. For that reason, **authorisation** must be taken in place, which validates if an user can perform the requested operation [44]. There are different ways to handle authorisation:

- one solution to handle application-specific authorisation is by implementing this in the API gateway. Restricting requests to specific endpoints become in this way possible. If a user is not allowed to access a particular path, the API gateway can reject the request before forwarding it on to the service. As with authentication, centralising authorization within the API gateway reduces the risk of security vulnerabilities. The drawback of implementing authorisation in the API gateway is that it can be only role-based access to endpoints. Implementing additional checks on access to specific domain objects would need to create specific domain logic inside the API gateway. It's also not practical for the API gateway to implement an Access-Control List (ACL) that control access to individual domain objects, because that requires detailed knowledge of a service's domain logic [44];

- another solution is to implement authorisation inside each service. They can implement Role Based Access Control (RBAC) and ACL for controlling service functionalities. The API gateway should pass the session token along with the request towards the microservice. This token should contain the roles assigned to the user. The microservice will then determine if the user is allowed to make the request. The most significant disadvantage is that authorisation would be more dispersed across various services. If roles change very often this becomes difficult to manage [44];

When dealing with authentication and authorisation it is common the use of a transparent token that contains the information about the user. One popular standard is the JSON Web Token (JWT). This token consists of a header, payload, and signature, as specified in Request For Comments (RFC) 7519: the header contains the algorithm used to sign the token. It is generally signed with `HS256` or `RS256` [20, 44]; the payload is essentially a JSON object with additional properties about the user, such as its identity, roles and other metadata (issuer, expiration, etc); and the signature is the result of combining the header, payload and a secret and sign them using the algorithm specified in the header. Since the token is signed by the identity server using a secret, the information can be validated and trusted by the consuming application. The application can validate the token against the public key of the certificate used by the identity server for signing the token [20, 44].

One problem with JWT is that a token is self-contained and after checking the JWT's signature and expiration date, the application can execute the request process by default. As a consequence, there's no realistic way to revoke the token that's been compromised by a malicious third party. The alternative is to use short expiration periods. However, one disadvantage to short-lived JWTs is that the application must reissue tokens continually to keep the session alive [44]. These problems are solved using standards protocols for centralised authentication and authorisation like `OAuth2`.

As presented before, a service can be built to authenticate a client and obtain a session token. However, there are cases where this generic functionality is not required. In that case it is possible to use an off-the-shelf service called `OAuth2`, as specified in RFC 6749 [16]. This is an authorisation protocol that allows a user of an external service to grant a third-party application access to their data without having to disclose their password [44].

2.7 Data storage patterns

Normally a system design begins by identifying and establishing the data storage models. This is an effective way to design centralised systems, but this might not be a good way to implement on a microservices architecture [33].

In the following sections, it is described three different data storage patterns which are used during system design.

2.7.1 Shared database

Typically used on monolithic applications, this approach is represented as a single database shared by multiple services [45, 1].

Each service freely accesses data owned by other services using transactions, following the Atomicity, Consistency, Isolation, Durability (ACID) properties to enforce data consistency [45]. There is another variation where each service has a private database schema [38].

Figure 2.4 shows two microservices accessing the same database.

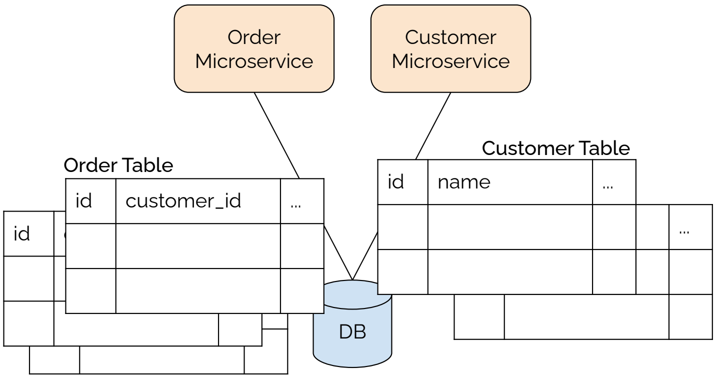

Figure 2.4: Shared database pattern.

Some of the advantages of this pattern are listed bellow:

- the use of ACID principles ensure that concurrently transactions will not violate the consistency of the data [45, 1];
- it is easy to move from monolithic applications as the same schema can be reused without any modification. The current code base can also be migrated without major modifications to the data access layer [1].

Using this pattern will bring some problems during the development and production of microservices:

- schema changes need to be coordinated with developers of other services [45];
- since all services use the same database, they could conflict with one another. For instance, long run transactions will hold a table lock [45]. The data can be accessed concurrently and in some cases without any data isolation [38].

2.7.2 Database cluster

This pattern is very similar to the previous one and it proposes storing data on a database cluster. The key difference is that this pattern improves scalability and allows each data to be replicated across services' databases.

It is usually recommended for services with huge data traffic. However, complexity and risk of failure increases due to cluster architecture and the distributed mechanism [38].

Figure 2.5 shows different microservices accessing the database cluster.

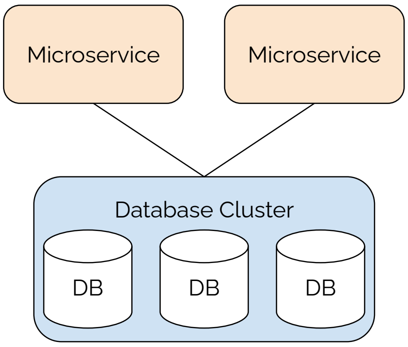

Figure 2.5: Database cluster pattern.

2.7.3 Database per service

This is a widely used approach that involves decentralising data management [27] where each service owns and accesses its private database [44, 1]. Figure 2.6 shows how each service interacts with its own database.

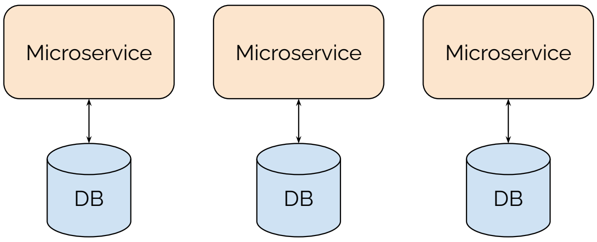

Figure 2.6: Database per service pattern.

It is described as the easiest approach when implementing microservices-based systems [1] and its advantages are listed below:

- each service can use different database technologies - an approach called polyglot persistence [27];
- it is possible to scale a database by using a database cluster when a service needs to scale [1];
- development teams can work independently on a service without influencing other teams' work in the event of changes in the database schema [1]. The developers can change the schema of a database during the development without coordinating with other developers [44];

- each database is secured against improper access and corruption of data from other microservices [1];

Using this pattern will introduce some issues when implementing queries that requires the use of multiple services. The use of API composition is a possible solution to this problem [44]. In order hand, distributed transactions are difficult to implement. As a consequence, microservice architectures emphasise transactionless coordination between services since consistency may only be eventual consistency [27].

2.8 Communication strategies

Moving away from traditional monolithic applications to highly available distributed microservices, may require some communication between them over a network [22, 33]. Every service needs to have an interface to achieve a high degree of software code isolation, independence and modularity. Otherwise it won't be able to independently deploy two microservices [33].

There are different ways of organising the communication between two services over a network. Considering the clients and the servers is the most common arrangement. The servers will expose an API over the network, and the clients will be able to connect to the servers by making requests based on the API. It is also important to mention that a server can itself be a client to another service [22].

Different communication solutions are used in a microservice architecture and they can affect the applications availability, so it should be considered the options carefully, as there are no silver bullets. Existing mechanisms can be divided into synchronous communication and, alternatively, asynchronous communication protocols [44].

2.8.1 Defining APIs

One advantage of microservices is allowing a more modular development approach: services can be performed by smaller independent development groups. To allow the communication between two services both groups must standardise how these services can be accessed [7].

Usually the clients request microservices through an API gateway that distributes incoming calls to different services with the help of discovery and routing tolls. To ensure long-term stable interaction between services and clients, it is important to establish a proper API with appropriate documentation [7].

2.8.2 Synchronous communication

On a synchronous communication there are two clients involved. One of them makes a request to another and waits for a response. It is expected that this response arrive between a certain time interval. If not, this client may block until the response arrive. This interaction usually results in a tightly coupled operation. There are two request/response mechanisms that are generally used on HTTP-based synchronous communication [44] and they are addressed on the following sections.

2.8.2.1 REST

Communicating using a synchronous approach is generally related to Representational State Transfer (REST) over HTTP protocol [44].

A key concept in REST is the resource, that usually represents a single business entity, such as an "user" or a set of users. REST uses the HTTP verbs to access resources referenced via a URL, such as *GET*, *POST*, *PUT* and *DELETE*. These verbs means that it is possible to retrieve, create, update and remove resource, respectively [44].

REST is a very simpler and familiar approach. However using it might bring some problems:

- generally the service will block until the response arrive, as described before [44];
- availability is reduced since both services communicate directly without an intermediary and both must be running for the duration of the exchange [44];
- each service must know the locations (IPs or URL) of other services to be able to send requests. Sometimes a service discovery mechanism is used to locate service instances [44];
- the overhead of HTTP for each request may also be a concern for low-latency requirements. If extremely low latency or small message size is important, it should not be considered this approach [35];
- the consumption of the payloads may require advanced serialisation and deserialisation mechanisms [35].

Despite these drawbacks, REST over HTTP is a reasonable default choice and also the preferred way for service-to-service communication [44, 35].

2.8.2.2 gRPC

Remote Procedure Calls (RPC) is a technique used for inter-process communication between processes in different computers across the network. It is used to make a local call and execute it on a remote service [33].

This technology relies on having an interface definition (discussed later on Section 2.9.2.2), which can then be used to create client and server stubs in various programming languages [33, 55].

When using RPCs, a client creates requests in the form a procedure to the remote server. The remote server receives the request, processes it and sends the response back to the client, giving the idea as if it was running inside the client. It is important to keep in mind that local calls are usually faster and more reliable than remote calls [33, 55].

As mentioned before, on challenge with using REST is that HTTP only provides a limited number of verbs, and it might not be straightforward to design a REST API using them. For that reason, using RPC seems to be a good implementation to use on MSA, due to transparency in communication and the possibility of allowing services to call methods from others easily [55, 44].

gRPC[1] is a binary message-based protocol from Google that simplifies the building of connected systems. It was designed to be as efficient as possible and this lies in the way the serialisation is handled. It is based on Protocol Buffers, an open source mechanism for serialising structure data [52, 46, 44], discussed later on Section 2.9.2.2.

gRPC builds on top of HTTP/2 protocol, which allows for faster and long-lived connections, reducing the network latency [52, 46, 44].

The Protocol Buffer compiler generates client-side stubs and service-side skeletons, which enables services to know exactly how to call other services and what is expected in the responses. Moreover, it generates a strongly-typed cross-platform client and server bindings for a variety of programming languages [44].

gRPC also has some drawbacks:

- clients (and especially JavaScript clients) are used to consume REST APIs, and more work is required to start consuming gRPC-based APIs [44];
- it has a more strict specification since it depends upon Protobuf. Other content types are not supported out-of-the-box as with standard HTTP REST-based APIs [44];
- tools and older services might not support HTTP/2 [44]

[1] https://grpc.io/

gRPC is a convincing alternative to REST, but it suffers from the same partial failure problem as REST since it is a synchronous communication mechanism [46, 44].

gRPC APIs can offer huge performance improvements and reduced response time as compared to REST APIs, but which approach to choose depends on particular use cases and system requirements.

2.8.3 Asynchronous communication

Previously it was studied the various ways in which data flows synchronously from one service to another. In this section it is described how to communicate asynchronously using event-based mechanisms.

With asynchronous communication, a service sends a request to another service, which may replies later asynchronously. Unlike the previous pattern, the first service will not block while waiting for the response. This is suitable for distributed systems due to the asynchronous nature [44, 34].

Many organisations using microservices architectures, such as Netflix, listed the notion of messaging and events as a key design practice [33].

In event driven systems, an intermediary, also known as message broker, is used to exchange messages from one service to another. These events are just messages shared between different services [35, 44]. A producer service uses a well-defined API to publish events to the broker. The broker then handle the subscriptions and delivers the events to the consumer service [35]. This is known as publish/subscribe communication.

2.8.3.1 Publish/Subscribe

Publish/subscribe is an asynchronous communication with multiple receivers. The services in general don't know each other and they communicate via a specific topic. The publisher defines the message topic while the subscriber registers for one or more topic of interest [23, 53, 44].

The message brokers are able to buffer events until the consumer is able to process them [44]. They are also able to keep track of which event each consumer has seen before. For that reason, they are designed to be scalable and resilient [23, 35].

There are some message brokers to choose from, such as `ActiveMQ`, `RabbitMQ`, `Apache Kafka` and even cloud-based brokers like `Amazon EventBridge` or `Amazon Kinesis` [35, 44].

Each one will have different trade-offs, including latency, ordering, delivery guarantees, persistence mechanisms, etc. The best option will be determined by the system requirements [44]. All of them will try to handle two problems: a way to each service send messages, and a way to a service notice and receive those messages [33].

Event-driven architectures generally lead to significantly more decoupled and scalable systems. However, there are some downsides:

- there is a risk that the message broker could be a performance bottleneck in the system [44, 35, 23];
- the message broker must be highly available, otherwise becomes a potential point of failure in the system [44];
- the broker is another element in the system that needs to be managed, leading to a more complex architecture [44, 23].

In a MSA, especially architectures using the database per microservice pattern, might need to exchange data between services. This can be done asynchronously by exchanging events [11, 44]. Domain events are generally used to implement side effects across different services [29].

An event usually represents a state change in the system. It should be published when something has been created or updated so that other services can consume it. [44, 29].

Domain events are useful because other services are often interested in knowing about a state change. Each domain refers to the change that was performed.

Each event has its own domain and properties that express the event's significance. Each property is either a single value or an object. It may also contains some metadata, such as the event identifier, a timestamp and the user that made the change. These values are usually important for auditing [23, 53, 44].

There are some use cases for publishing events, as described below:

- notify different services via a message broker or via webhooks to trigger the next step in a business process [44];
- notify different components inside the service to update a text database such as ElasticSearch or for sending notifications like emails [44];
- analyse events to check the user behaviour [44];
- use events for auditing and check if the system is behaving correctly. If systems get into a strange state it is possible to debug using the full log [11, 44].

To reliably publish events the service must use a transactional mechanism to ensure that they are published as part of the transaction that updates the information. This can be achieved using the transactional outbox pattern [44].

2.9 Encoding data

In the context of microservices communication, encoding data, also referred to serialisation or marshalling data [22], is the process of translating a state into a format that can be transmitted over the network and reconstructed later on other services [48].

Applications continue to change and evolve over time. Most cases happened when new functionalities are required, the business circumstances change or a there is a new understanding of the user requirements. Communication and data storage formats usually experience some changes during this process [22].

The choice of data serialisation format for an application depends on factors such as data complexity, human readability, disk space reduction or other bandwidth constraints [21].

Applications typically work with two different representations of data: in memory, *e.g.* when the data is kept in objects or well defined structs; and in some kind of self-contained sequence of bytes, *e.g.* when the data is saved on a file or sent over the network [22]. With this in mind, in this section several types of serialisation that are used between services for communication are described.

2.9.1 Language-specific formats

Many common programming languages have support for serialisation and deserialisation included in the core language or in standard libraries [15] - *i.e.* Java has `java.io.Serializable` [37], python has `pickle` [43], and so on. These different techniques allow the content of objects to be saved and restored with minimal additional code [30, 15]. This approach present some problems, described bellow:

- the encoding is often related to a specifc programming language [22], which is not portable [15]. Transmitting data in such an encoding can require the use of the same programming language over time, preventing the integration with other services developed in different programming languages [22];

- using systems with different processors architectures (big-endian, small-endian) contribute to a different binary representation of numbers and other objects [15];

- in these libraries, versioning data is often an afterthought: since it is intended for fast and simple data encoding, which can be a problem if the data format change during time - *i.e.* forward and backward compatibility problems [22];

- the serialisation is generally not efficient - *e.g.* Java's built-in serialisation is renowned for its bad performance (CPU time taken and the size of the encoded result) [22].

2.9.2 Language-independent formats

Moving to widely known, well supported and language-independent formats is a wise choice, as they have support for sharing information between modules built in different programming languages or executed on different architecture systems. [15, 22].

It has become increasingly difficult to choose the correct data serialisation format when building services [48]. They can be divided into two different types: text-based formats (*i.e.* a stream of text characters) and binary formats (.*i.e* a stream of bytes) [48, 15].

2.9.2.1 Text-based formats

JavaScript Object Notation (JSON), eXtensible Markup Language (XML) and Comma-separated values (CSV) are well known and well documented text-based data formats [48].

XML is generally known as being overly verbose, too complicated [22] and not suitable for mobile environments [48]. JSON, by comparison, is considered to be a simpler, more versatile, lightweight and efficient alternative to XML. It is also supported by the web browsers [48, 22]. CSV is another popular format, however less powerful [22].

These text-based formats have some drawbacks:

- there is much controversy around numbers encoding - endianness and size of memory representation. XML and CSV can't distinguish between number and string. JSON can do it, but it does not differentiate numbers and floating-point numbers and does not define a precision [22] [15]. Programming languages have different "basic integer type" - *e.g.* Java define `int` as 32-bit variable and C use the platform definition of `int` and at least 16-bit [15];

- JSON and XML support Unicode character strings, however they don't support binary strings, so the solution is to encode the binary data as text using `Base64`, which increase the data size [22];

- JSON is also not extendable, does not support namespaces and lacks on input validation [48]; CSV does not support schemas (optionally available for both XML and JSON), so it's application responsibility to define the meaning of each row and column. If an application changes a row or column the need to manage the adjustment would be present [22].

The overhead of parsing text is another drawback, especially when the messages are large. If efficiency and performance are important, it might be better to consider using a binary format [44].

2.9.2.2 Binary formats

In lieu of these common text-based formats, Protocol Buffers [14], Apache Thrift [3] and Avro [2] are the more recently used binary data serialisation formats [48, 22]. Some binary text-based formats (BJSON, WBXML, and others) have also been created, but none of them are widely adopted as textual versions or the formats previously listed. [22].

The binary formats were designed to be extremely lightweight and fast to serialise and deserialise data [48]. Compared to text-based formats, the data volute is less, since the data is further compressed to minimise the size and to improve the speed of the response during the transmission process [10].

Usually there is no need to use the most common encoding format for internal use within an organization, as there are more compact and faster formats to parse. Using a lighter and faster format on large datasets will have big impact. This observation led to the development of binary encodings [22].

The protocols mentioned before require a schema to encode any data. They provide a typed Interface Definition Language (IDL) to describe the schema. The formats come with a code generation tool (a compiler) that takes the defined schema and produce the code that serialises and deserialises records [44] [22]. There are several variations in how serialisation and deserialisation are treated by these formats - *i.e.* Protocol Buffers uses tagged fields, while for reading messages, Avro only needs to know the schema to interpret messages. Handling the schema evolution is easier with Protocol Buffers or Thrift than with Avro [44].

Textual data formats are widespread, however, schema-based binary encoding is a viable option, too. These formats are simpler to use and have a great variety of programming languages support [22]. They share some nice properties - *e.g.* they omit the fields names on the encoded data, so they can be much more compact; some formats allow the integration with databases of schemas where shema changes can be tested forward and backwardly for compatibility; it is also very useful to generate code from the schema to a specific programming language, which enables type checking at compile time [22].

Even if the binary formats looks better than text-based formats, there are some small drawbacks:

- these formats are not as adaptable as text-based since the data sent in a binary format can not be interpreted unless the receiver has the schema files [48];
- changing schemas overtime can lead to lack of forward and backward compatibility. Sometimes it is not possible to make a new fields as required and sometimes it is only possible to remove an optional field [22];
- some formats doesn't support a wide range of programming languages [48, 22];
- the encoded data must be decoded before being readable by humans [22].

2.10 Caching

Today's web applications rely on caching to reduce latency and system load, using services such as *Redis* , *Memcached* or other key value stores [6]. Cache usually helps to reduce the stress on a databases by positioning itself as an intermediate layer between the database and the end users. In general, when making a request, the returned data will be stored on the cache in such a way that it will be easier to access in the future. Any future query will check if the request is on cache and if it misses, it will fall down on the database [17].

If the data layer is constantly queried, latency will result in high strain and poor user experience. To address this, data that is regularly accessed can be temporarily stored in memory for fast future access. When cost and speed is considered, the cache size growth may be limited [17].

The needs of each application differ, and significant performance improvements can be achieved by adding a caching strategy to the application or service. Caching performance is determined by the workload and the caching algorithm when the cache is full. For instance, it is important to define a good strategy to select what item need to be removed from the cache [6].

There are some challenges when using a caching:

- a fundamental problem with caching is a limited cache size. Available storage may be limited mainly for price reasons. Cache management has two primary responsibilities: the cache decision and the cache replacement strategy [39];

- the cache decision strategy specifies which data should be stored for the system optimisation. There is not much work on approaches to cache decisions [39];
- replacement strategies are necessary if the size of a new data object exceeds the amount of free space left in the cache. Delete objects that are less likely to be needed again in the future is desirable. Items are replaced based on a usage prediction [39];
- consistency is another concern. Cached data is inconsistent when its source is updated but the cached copy is not. There is the need of creating mechanisms to ensure that cache is consistency [39].

3 The problem and its challenges

As it was shown throughout the Chapter 2, new technologies are being adopted. The use of microservices has been evident in several areas, including the digital world and IoT. However, designing and adopting an microservice architecture has not yet been well established and it is still a common problem that businesses need to deal with.

It was noticed that microservices have been adopted as a natural solution in the replacement of monolithic systems. Companies tend to embrace microservices architectures for different use cases, and with the goal in mind of mitigating monolithic architecture problems and making sure the investment of building MSA brings long-term benefits.

Microservices architecture offers some advantages to software applications, including smaller development teams, flexible programming languages, scability, etc. At the same time, architectural problems are also introduced. For that reason it is important to study and apply the best practices and patterns when adopting a MSA. Microservices are only in their early stages of integration, and the study of this architecture is yet to be widely studied in universities and adopted in businesses, in contrast to monolithic systems, which have a well-defined 3-tier architecture that is widely used.

Therefore, this dissertation aims to present the elements that should be considered when designing and implementing solutions based in microservices. The MSA study was presented on Chapter 2, and the following chapters will look at a particular use case to validate the conception of a microservice-based architecture. The use case will be used to assist the creation of a microservice architecture in order to help others understand how an MSA should be designed and developed using the common patterns.

Systems are continuously generating data that should be treated and analysed in order to produce vital information for businesses. This is the case of taxi companies which require valuable information in order to make decisions in real time. Sensors are being installed in each vehicle to provide intelligent transportation mechanisms, such as: efficient taxi dispatching, better route discovery and taxi sharing. Taxi dispatching services have replaced the conventional very high frequency radio dispatch system by installing mobile data terminals in taxis that usually provide GPS location information and the taximeter's state [32].

Thus, in this dissertation a microservices architecture will be created to collect and process data produced by 441 vehicles of a fleet running in the city of Porto, Portugal. The data under study was collected over a one year period and contains information related to the taxis GPS position and the taximeter's state. A CSV dataset was then created to help to predict taxi-passenger demands and to improve taxi-driver mobility intelligence [32]. It is presented on Appendix A.

Based on the taxi company study and the generated dataset, some requirements were established to be possible to define an architecture around a specific use case. Once again, it is important to mention that this particular use case will only be used to evaluate and validate the architecture being developed and prototyped. The proposed architecture should easily be adapted to different case studies and it should provide a conceptual structure for future projects.

In the system, there are three distinct intervenients: the administrator that manages the taxi's platform, the taxi drivers, and clients.

The functional requirements are listed below:

- all users should be able to login in the platform using its credentials;
- the administrator should be able to
  - add, retrieve, list and delete taxis and drivers in the system;
  - retrieve and list users;
  - retrieve and list trips;
  - add, retrieve, list and delete permissions and associate them to users;
- the client should be able to
  - retrieve and list its trips;
  - rate a given trip and driver after being notified by email at the end of it;
- the taxi driver should be able to
  - define its position based on coordinates;
  - create and update a trip. Each trip should include all taxi's positions and when it was finished.

To conveniently capture the functional requirements of the proposed platform, a use case diagram was created, and it is presented on Figure 3.1.

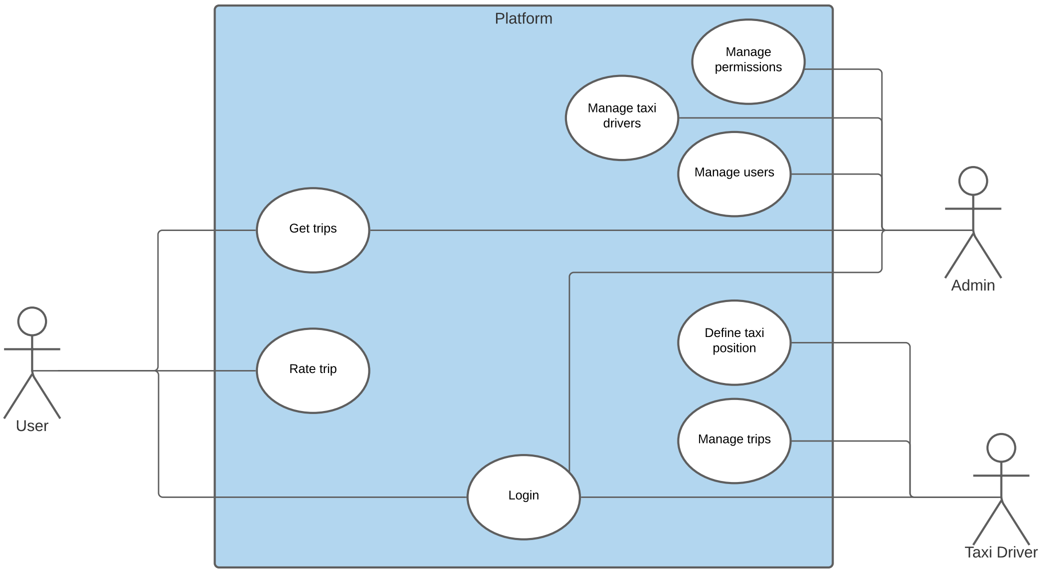

Figure 3.1: Platform's use case diagram.

The word "manage" in the diagram above refers to adding, retrieving, and removing items from the system. As previously stated, the system's actors are: the administrator, the taxi driver, and the client of the taxi platform.

Each actor has a small set of use cases to keep the system simple, making it easy to understand the key elements necessary to construct a microservices-based solution without fumbling by the complexity of the requirements.

3.1 Proposed architecture

It is important to identify an architecture capable of providing the core capabilities defined previously. As a result, the system general architecture is presented on Figure 3.2 using a components diagram.

The proposed architecture contains an API gateway, an event bus and four different microservices that are able to work together to form a platform.

As presented in Figure 3.2, the API gateway will serve as the platform's initial entry point, providing some security and forwarding requests to each microservice.

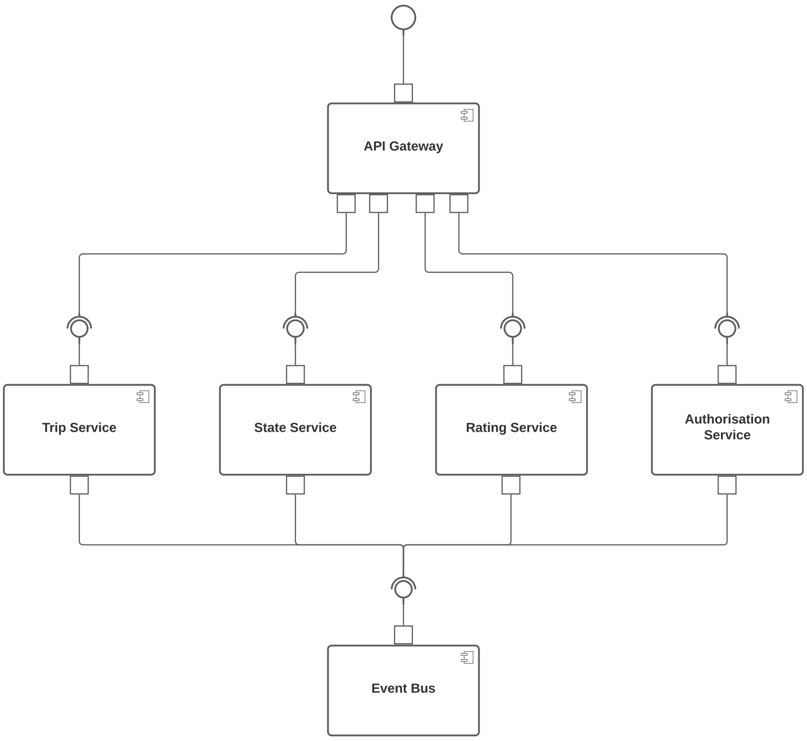

Figure 3.2: Components diagram of system general architecture.

On Chapter 2, the API gateway was presented as an essential component in the architecture to act as an intermediary between external clients and internal services, establishing a private network environment which allows for private data sharing between these parties. In this architecture, the API gateway is intended to:

- provide a single and unified API entry point across all internal APIs;
- act as a reverse proxy, routing the requests to the appropriate microservice;
- provide authorisation in the platform by validating access tokens.

Other factors can be consider when integrating an API gateway. It may be used to encrypt all incoming requests with Secure Sockets Layer (SSL) technology and used to enforce rate-limiting to control network traffic.

The event bus will be used to enable loosely coupled integration between the proposed microservices. The events will serve as notification and will enable services to react to changes in the platform. Each service will be able to consume as well as publish events. Each event contains set of identifiers that allow the change to be retrieved. These events are meant to be used as notifications rather than events with payload, which avoids distributed consistency problems.

The event bus will operate on a publish/subscribe model, with events being published and received by any service that has subscribed to specific type of events. This means only a set of services will benefit from the updates published onto the bus.

The proposed architecture will also contains four microservices that match specific business domains. The microservices reflect the four separate contexts within the system, namely: trip management, users and taxi management, rating management and finally, authorisation management. They were carefully selected to serve a purpose in the platform, as listed below:

- the `state-service` will be responsible for managing the users and taxis registered in the platform, as well as providing authentication, which will enable users to access internal services with a valid token.
- the `trip-service` will be used to manage trips in the platform. This service will allow taxis to create and update trips as well as allow users to get the trips made in the past.
- the `rating-service` will be used to manage the ratings of trips made in the platform. This service will be in charge of sending emails to users asking them to review their journey, as well as receiving the requested classification results.
- the `authorisation-service` will control each user's permissions, allowing each service to grant or deny access to a certain resource on it, i.e. trips, users, taxis, etc.

In order to simplify the diagram representation, the Figure 3.2 does not illustrate inter-services relationships. However, these services will be able to interact with one another via internal interfaces.

The services will have similar internals components. The Figure 3.3 presents the components diagram example of an individual service.

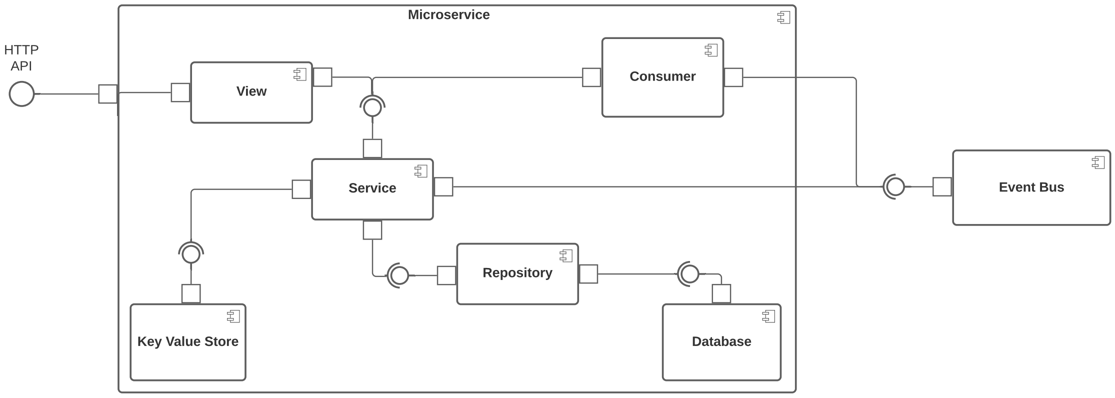

Figure 3.3: Microservice components diagram example.

The service will expose a HTTP interface that can be consumed by external and internal services. Each request will be handled by the view layer to validate the requests content, and then the service layer will handle and process the request, by managing resources from a database or a key-value store. The database is accessed by a repository layer to avoid code duplication. The microservice will also include a consumer capable of polling events from the bus, enabling the service to react to platform changes.

Each microservice will have its own database, a common pattern in MSA, as studied before. It's worth mention that would be possible to use a a shared database server or a database cluster for all services with different schemas, so that each service would have complete control over its data.

A key-value store will be included in all services to temporarily cache the user permissions in the platform. When a new request is made, the service verifies if the user permissions have already been fetched. If it hasn't already, a new request to the `authorisation-service` is made, and the user permissions are then cached temporarily for future access, decreasing the number of requests to the `authorisation-service`.

Some of the decisions that reinforce the proposed architecture are explained in the following sections. Further details on how the architecture was developed are present in Chapter 4.

Development 4

The system architecture that will be used in the development of this system was presented in the previous chapter.

In the following sections it is discussed the decisions and development that allowed the conception of the architecture previously proposed.

4.1 System technologies

Before starting the development of the system, it is important to define the technologies that will hold the platform. As studied before, microservices can be developed using different programming languages and persistent technologies, which means there is a variety of choices.

Since a microservice architecture is a new research area, systematic mapping studies are crucial to summarise the progress so far and to identify the common architectures and technologies used [1]. Some of these studies were taken in consideration to evaluate how this architecture will be developed.

It was studied that microservices generally expose interfaces that can be used to interact with other services or applications. This interfaces called APIs are typically built HTTP and they tend to follow standards like REST for exchange data [31]. The majority of the keywords associated with various studies centered on microservices were API and REST [1]. For that reason each microservices will expose an API that other services can use to interact with it. These APIs will be built using HTTP protocol and will follow the REST standards.

REST APIs can be built using a range of technologies and programming languages [31, 12]. `Python`[1] was the selected language as there is already some familiarity with it. In terms of selecting the choice of a technology to abstract low-level protocols and to help building REST APIs, `Flask`[2] was the web application framework of choice. There were other possibilities like `Django Rest framework`[3], however `Flask` is a lightweight framework, with a small set of dependencies and it was design to be flexible, giving more control on what is used inside the service. This choice also results as an opportunity to learn the framework, however nothing blocks the use of other technologies or programming languages when building a microservice architecture.

[1] https://www.python.org/about/

[2] https://flask.palletsprojects.com/en/1.1.x/

[3] https://www.django-rest-framework.org/

This also applies to the choice of a database, where multiple options are available. In the proposed architecture all services will use Postgresql[4], a relational database. There is some familiarity with relational databases and particularly with Postgresql, and for that reason this was the choice for the database server. An Object Relational Mapper (ORM) called SQLAlchemy[5] will be used to manage database interactions. This is a well known and well used Python library used for efficient and high-performing database access.

The key value store of choice was Redis[6]. There were other alternatives like Memcached[7], however Redis is well known in the market, it has a lot of powerful data types and is very fast. For that reason, this seemed a very good choice.

In a microservice architecture it is common to have many instances of a single service running at the same time. Docker[8] containers have been gaining a lot of attention because of their easiness of making new services available and minimising the downtime. Containers allow the microservices to be packaged and available next to their dependencies in a single image [12]. For that reason, Docker containers will be used during the development of the architecture. A tool called docker-compose[9] was also used to help defining and running multiple containers.

Microservices are particularly suitable for cloud infrastructures, as they greatly benefit from the elasticity and rapid provisioning resources. The infrastructures and technologies provided by cloud services play a fundamental role for building and managing services. Multiple companies are migrating their services to cloud [12], which means that considering cloud computing services seemed a good approach.

For the proposed architecture, the cloud computing services offered by Amazon Web Services (AWS) were identified as well suited in opposition to Google Cloud Platform, Azure and others. The services provided by AWS are known to be used by companies creating MSA, and it is important to note that there was already some prior experience and knowledge with these services.

As presented before, an event bus is required in the proposed architecture. AWS includes the Amazon EventBridge[10], a fully managed and scalable event bus that is able to forward events base on well defined rules. However this services is only able to forward events to AWS services, and not

[4] https://www.postgresql.org/
[5] https://www.sqlalchemy.org/
[6] https://redis.io/
[7] https://memcached.org/
[8] https://www.docker.com/
[9] https://docs.docker.com/compose/
[10] https://aws.amazon.com/eventbridge/

directly to external services (for instance, via webhooks). For that reason, AWS also includes the Amazon Simple Queue Service[11] (SQS), a fully managed message queuing service, that will be used to queue the events from the bus. Each microservice consumer will pull and process the events to ensure the service reacts to each update on the platform. Figure 4.1 illustrates how AWS services interact with a microservice's consumer.

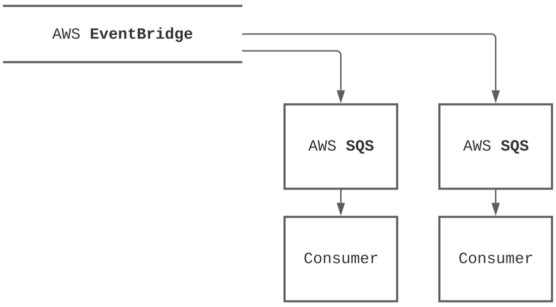

Figure 4.1: AWS services interaction.

Microservices will publish events to Amazon EventBridge, which is then responsible for forwarding them based on a set of configured subscriptions, each of which contain rules for which events to forward.

It's worth mention that each service (and, therefore, each consumer) will have their own SQS queue. However, there may be situations where several queues are necessary, such as in systems where quality of service is expected and certain events must be prioritised in the queue.

For the purpose of this dissertation and to minimise costs, a fully functional local AWS cloud stack known as `Localstack` [12] will be used.

API composition and protocol translations, which API gateways provide, are not needed in the proposed architecture. As a result, using off-the-shelf API gateways appears to be the most reasonable option, as no development and minimal configuration are expected. Several gateways are available: `AWS API Gateway`[13], `Kong`[14], `Traefik`[15], `Tyk`[16], etc.

In the proposed architecture, the API gateway needs to be able to provide authentication mechanisms, which means custom plugins must be supported in order to better integrate the architecture design. For that reason, the `AWS API Gateway` is discarded because custom plugins are not sup-

[11] https://aws.amazon.com/sqs/
[12] https://github.com/localstack/localstack
[13] https://aws.amazon.com/api-gateway/
[14] https://konghq.com/
[15] https://traefik.io/
[16] https://tyk.io/

ported. On the other hand, the other gateways support this feature, making the choice difficult. At the end, Kong was the selected option, as it is based on Nginx, a powerful HTTP server.

4.2 Event bus and message queues

As presented before, the event bus is primarily intended to a allow loosely-coupled integration between stateful microservices. Each service subscribe to a set of events (based on rules) and consume them from a queue.

The use of Localstack was crucial in the implementation of this architecture, as it allowed the simulation of fully functional AWS services running locally on the development machine. The docker-compose command was used to start Localstack with two specific services running: AWS EventBridge and AWS SQS.

Based on the proposed architecture, only one bus is expected and each service will have its own SQS queue.

The event bus and all SQS queues were created programmatically using a Python script. This had to be done ahead of time in order for each microservice to be able to use the AWS services.

A new library called Eventus was developed to abstract the integration of AWS services. This is important because it enables the integration of different services in the future, without requiring significant changes to each service's code.

4.2.1 Eventus library

The Eventus library was created to abstract and facilitate the process of publishing and consuming events. Even if the proposed architecture is restricted to AWS services, it was designed to hide the communication complexity between services and AWS. If at any time it is decided to move out from AWS, only a few changes are required in order to use the new services that are replacing AWS.

The events should all follow the same format. As a result, an Event class was defined to ensure that producers and consumers could effectively handle it. The following Listing 4.1 shows the Event class defined in Eventus library.

```
class eventus.Event:
    event_source: str
        The service/source name that published the event.
    event_type: str
        The event type.
    event_id: UUID
        The event identifier.
    payload: dict
```

```
        The event details.
```

Listing 4.1: Event class defined in Eventus library.

This event takes the format of a structured object. It contains the source, type, unique identifier and a payload with extra information to be consumed.

It is also important to mention that during the implementation of the proposed architecture, Universally Unique Identifiers (UUID) were used, based on Request for Comments (RFC) 4122, which are expected to be globally unique in space and time [26].

To interact easily with AWS services, a Python library called boto3[17] was used. This is a Software Development Kit (SDK) maintained by AWS and it is capable of creating, configuring and managing all AWS services.

The Eventus also provides two distinct classes: Producer and Consumer. Both classes require a boto3 client, that can be created with different AWS accounts in different AWS regions. It provides all the methods to interact with AWS services.

The Producer class is responsible for abstracting how services publish events onto the event bus. The Listing 4.2 presents the Producer interface.

```
class eventus.Producer
    service_name: str
        The name of the service using the library.
    event_bus_name: str
        The AWS EventBridge name.
    events_client: boto3.client
        The boto3 client.

    # public methods
    publish_event(event_type: str, detail: dict, event_id: Optional[UUID]) -> None
        Publish an event onto the bus with a specific type, detail and
        an optional identifier if the services wants to get control on it.
```

Listing 4.2: Producer class defined in Eventus library used to publish events.

To create the producer it is required to specify the service name that is publishing the events, the bus name created on Localstack and the boto3 client. Once the Producer is created, it is possible to publish events using the publish_event function.

The Consumer class is responsible for abstracting how services consume the subscribed events. Keep in mind that EventBridge will forward the events to the service SQS queue only if a rule is matched.

The Consumer interface is presented in Listing 4.3.

[17] https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

```
class eventus.Consumer
    queue_url: str
        The SQS queue url defined by AWS.
    sqs_wait_secs: int
        The number of seconds used on long polling.
    sqs_visibility_timeout: int
        Number of seconds in which an event should be hidden from other consumers.
    wait_secs_between_requests: int
        The number of seconds to wait between AWS calls to receive events.
    handler: function
        The function used to handle each event.
    sqs_client: boto3.client
        The boto3 client.

    # public methods
    run() -> None
        Consume events from SQS and dispatch them to the handler function.
```

Listing 4.3: Consumer class defined in `Eventus` library used to consume events.

To create the `Consumer` the queue URL is required. It was defined during the queue creation. It is also required to specify the handler function that will manage the event.

The AWS SQS queue provides both short and long polling to receive messages from the queue [4]. In the `Eventus` library it was decided the use of long polling as it allows the consumer to wait until a message is available or until the `sqs_wait_secs` is exceeded. This can help reducing the cost of using the SQS by eliminating the number of empty responses when there is no message available.

SQS also sets a visibility timeout if specified. The `sqs_visibility_timeout` value can be used to prevent other consumers to receive and process the same message. If a service has two or more consumers and one of them is slower than the others, a message can be received multiple times. The visibility timeout will not solve the problem, but will avoid it most of the cases.

In addition to the long polling, the `Consumer` also allows for an `wait_secs_between_requests` delay between SQS requests. Based on the number of events published to the bus, this value can be adjusted.

Once the `Consumer` is created, it is possible to start it by executing the `run` function. When a new event is available in the queue, the `handler` function is executed. This function receives a `eventus.Event` as an argument and once the event is consumed successfully (no exception is raised), the event is deleted from the SQS queue.

The `Consumer` is currently single threaded. However, depending on the volume of events, a multi-threaded consumer should be considered to handle multiple messages concurrently.

4.2.2 Subscriptions

According to the proposed architecture, each service has a SQS queue that receives events from `EventBridge`. These events are forwarded to services based on a set of rules.

`EventBridge` allows each rule to have several targets, e.g. multiple SQS queues. However, to get more control about how events are forwarded, it was decided that each subscription represents a rule with one single target.

Let's consider that two services want to subscribe the same subset of events. In this case, two rules are created. Each rule matches the same events subset. `EventBridge` is then responsible for forwarding the events to each service based on these two rules.

One more `Python` script using `boto3` was also developed. It is capable of creating each service subscriptions (rules).

These scripts are very convenient during development. For production, it is recommended the use of tools that manage the infrastructure in a secure and programmatic way, like `Terraform`[18] which is able to manage infrastructures across multiple cloud environments. AWS also provides a similar tool called CloudFormation[19] which manages the AWS services.

The Figure 4.2 shows an example of how events are routed to different services.

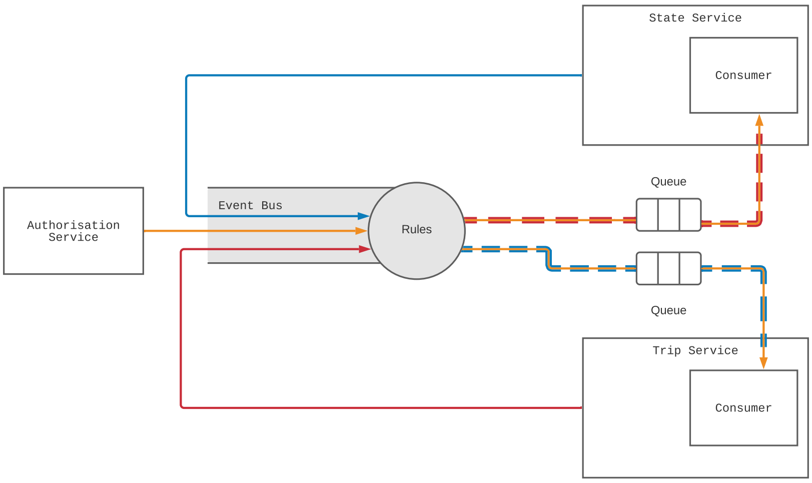

Figure 4.2: An example of how events are routed by the event bus.

If we consider the image above, it is possible to identify that the two scripts mentioned before should be able to: create one event bus; create two different SQS queues, one for `state-service`

[18] https://www.terraform.io/
[19] https://aws.amazon.com/cloudformation/

and another for trip-service and create two rules, one that matches the events marked as orange and red, which allows the bus to forwarded them to state-service, and another that matches orange and blue events, which allows the bus to forwarded them to trip-service.

All services in the platform are able to publish events when something has changed on their own environment. Each service is also able to subscribe its own events, however this is not common and it is not present on the current architecture. This might happen when a service wants to perform something asynchronously. However, in this case it might be better to add an event directly to its own SQS queue.

The EventBridge allows the creation of fine grained rules. As shown on Listing 4.4, only the source and event_type values are used. However, more attributes can be defined to improve the matching.

```
{
    "source": "trip-service",
    "detail": {
        "event_type": ["trips:trip_updated"]
    }
}
```

Listing 4.4: Event rule example.

The example above, allows EventBridge to forward the event to the specific target only if the event is coming from trip-service and the event type is trips:trip_updated.

The table 4.1 shows the events that will be published to the bus. They will be discussed in more detail later. The table below might grow as additional services are added to the platform or new functionalities are added to the platform, so it's useful to share it if several teams are working on different services.

Event	**Description**
trips:trip_updated	Event published when a trip is updated. The event payload includes the trip and taxi identifiers.
trips:trip_finished	Event published when a trip has finished. The event payload includes the trip and taxi identifiers.
auths:policy_updated	Event published when a policy is updated. The event payload includes the user identifier.

Table 4.1: Event types.

4.3 Service template

As described in Section 4.1, all services will share the same technologies. As a result, before getting into depth for each service, it's important to mention that a service template was created to ensure all services follow and share the same practices and patterns.

When services within an platform use different stack of technologies, it might be worth investigate if it is possible to create a template per stack of technologies, so it will be possible to easily bootstrap a new service.

Working on a this type of architecture means new services are often introduced, which means there is a good chance that related technologies (and consequently code) is shared. For that reason, using a template will keep consistency between service, ending up with less bugs and better maintainability.

If a more programmatic approach is required, it is recommended the use of some command-line tools that build projects from marked templates, such as `cookiecutter`[20].

In both approaches, existing services will require updates when a template is changed. This might be problematic if a lot of services are involved. However, future services can benefit of a good and mature template, enabling a faster development and a service that is less vulnerable to errors.

During the implementation of these services a Version Control System (VCS) called `git`[21] was used, and each service was built based on the service template repository using the fork functionality.

The service template is divided in three layers: the view layer, the service layer and the repository layer:

- the view layer contains all logic that validates the request format and content, making sure all incoming information is valid and will structured. It is also responsible for returning an well-formated JSON response;
- the service layer is responsible to validate permissions and for handling data between the view layer and repository layer;
- the repository layer works directly with the database though the use of `SQLAlchemy`. It is responsible for getting data into and out of the database in a secure way.

These layers work together to provide the business logic for a given service.

[20] https://github.com/cookiecutter/cookiecutter

[21] https://git-scm.com/

To handle HTTP requests a web server called `Gunicorn`[22]. This is a Web Server Gateway Interface (WSGI) capable of forwarding requests to frameworks like `Flask`, which is also very compatible and well integrated with it.

The consumer will run on a separate process with access to service and repository layer.

As presented on Figure 3.3, each service consists of an HTTP API handler, a consumer, a database and a key value store. This represents four docker containers.

The following sections describe all services that exist in the platform.

4.4 State service

The `state-service` holds the users and taxis information. It exposes a REST API to manage both users and taxis and provides an authentication mechanism to allow users to get access to the platform.

As mention before, all services use `SQLAlchemy` to translate `Python` classes to database tables. For that reason, we present the various classes that exist in each service to better represent how they were designed.

Various classes in the system include the `created_at` and `updated_at` attributes. Both are `datetimes` and they are very useful to keep track of changes in the system.

The Figure 4.3 shows the `state-service` class diagram.

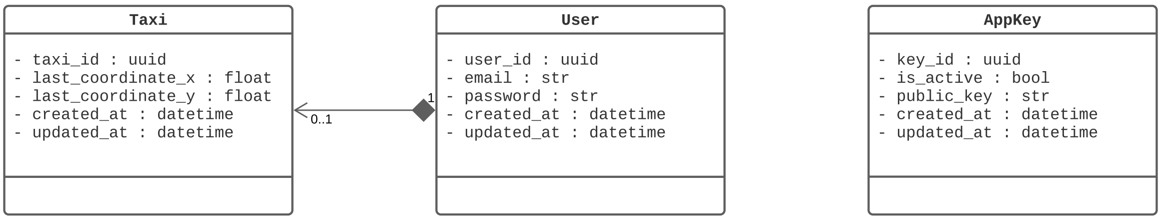

Figure 4.3: State Service class diagram.

The class `Taxi` contains the taxi unique identifier (UUID), the last registered taxi position and when it was created and updated in the system.

The `User` class includes the user unique identifier (UUID), the email, the password hash (for security reasons) and also when the user was created and updated in the system.

The last class in this services is the `AppKey` and it is part of the authentication system, which contains the key information used to produce authentication tokens. This class includes the key unique

[22] https://gunicorn.org/

identifier (UUID), the public and private keys, a boolean that identifies if it is active and also when it was created and updated. It is important to note that storing the private key in the database might bring security problems. As alternative, it is recommended to store it inside a secrets manager, like AWS Secrets Manager[23] or Vault[24].

The Figure 4.3 also shows a relationship between `Taxi` and `User` classes. The taxi will always be managed by a single user, for that reason this relationship was defined.

Keep in mind that additional information related with taxis and users would be stored in real-world applications. However, given the requirements and intent of this dissertation, it is only stored the information required to operate the service.

All services in the platform expose a REST API. For that reason a robust security mechanism protecting and controlling the access to any service within a platform is required. This mechanism was defined based on RFC 7519 and RFC 7517 [20, 19], which use JSON Web Tokens (JWT) to provide a token-based authentication and authorisation mechanisms.

In general, JWT tokens consist on a security mechanism to transfer information between two parties, using tokens that are signed. A signature verification can be performed to identify if the token was issued by the correct service, which means that the information contained in the token is authentic [20].

The `state-service` used an asymmetric signing algorithm called RS256, as recommended in RFC 7518 to sign the tokens. The signing process is well detailed in [18], but briefly an RSA key pair is generated and then an RS256 encryption is applied to the token payload using the private key. This key pair refers to public and private keys [18].

The authentication mechanism works as follows: an user sends its credentials to the `state-service` and after their validation, a JWT token is generated with some attributes, including the token expiration time, the user UUID, and others. After that, anyone with the public key can perform a token verification. That is when the JSON Web Key (JWK) is generally used. It is a JSON data structure that represents the cryptographic key information. This object contains the encryption metadata and the key value itself [19], that can later be used to validate the token signature.

A set of JWK (also know as JSON Web Key Set - JWKS) is the default structure used to share public keys between services, allowing them to validate signatures. In this case, the platform will rely on the API gateway to validate JWT tokens on all incoming request. This described later on 4.8 section.

[23] https://aws.amazon.com/secrets-manager/

[24] https://www.vaultproject.io/

Using JWT provides an additional layer of security, as only the owner of the key pair can generate tokens signed with the private key. With a token-based authentication mechanism, the need for a client credential is replaced with a token that provides efficient client privacy preservation [56]. Access control is enforced by the API gateway, which removes the concerns of validating a token from the microservices, such that they can remain lightweight.

Apart from providing authentication and authorisation mechanisms, this service is also responsible for managing taxis and users. It exposes an REST API with endpoints capable of creating, updating and retrieving information. These are presented on Table 4.2.

Method	Endpoint	Description
GET	/v1/taxis/	List taxis
POST	/v1/taxis/	Create new taxi
GET	/v1/taxis/<taxi_id>	Retrieve taxi
PATCH	/v1/taxis/<taxi_id>	Update taxi
GET	/v1/users/	List users
POST	/v1/users/	Create new user
GET	/v1/users/<user_id>	Retrieve user
POST	/v1/auth/token/	Create new token
GET	/v1/certs/jwks.json/	List JWKs
POST	/v1/certs/jwks/	Create new JWK

Table 4.2: State service API endpoints.

4.5 Trip service

The `trip-service` is responsible for storing and managing the trips information. It also exposes a REST API that enables taxis to create and update trips.

This services holds a single class and it is presented in Figure 4.4.

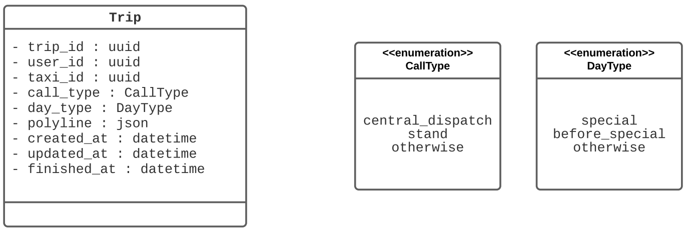

Figure 4.4: Trip class defined in Trip.

The `Trip` class contains the trip unique identifier (UUID), the user and taxi identifiers involved in the trip, the list of coordinates that make up the polyline of the trip and when the trip was created, updated and finished. It also contains information about how it was created (`call_type`) and the type of the day that the trip was performed (`day_type`). These two last values are enumerations and they can have multiple values.

The `call_type` identifies the way used to demand the trip. The possible value are:

- `central_dispatch` when the trip was dispatched from the central;
- `stand` when the trip was demanded directly to a taxi driver at a specific stand;
- `otherwise` when none of these.

The `day_type` represents the trip's start day type. The possible value are:

- `special` when the trip started on a holiday or any other special day;
- `before_special` when the trip started on a day before a special day;
- `otherwise` when the trip occurs on a normal day.

The `trip-service` exposes three different endpoints, used to create, update and retrieve a given trip. They are listed below on Table 4.3.

Method	**Endpoint**	**Description**
POST	/v1/trips/	Create a new trip
PATCH	/v1/trips/<trip_id>	Update trip
GET	/v1/trips/<trip_id>	Retrieve trip

Table 4.3: Trip service API endpoints.

A trip is updated when the taxi change its position or when the trip has finished. The taxi is able to periodically send update requests to notify the system about its new position. Once the taxi reach the final destination, the trip can be marked as finished using again the update endpoint.

As mentioned before on Section 3.1, some events will be published to the event bus and they will work as notifications. Subscriber services should then react to this changes and request extra information from the service that has published the event.

In this service two events are published to the event bus: the `trip_updated` and `trip_finished`. Both events are prefixed with ”`trips:`” to give some context of where they are being published. We may consider that when a trip has finished it is also updated. However, it is important to mention that events can be more or less specific. They just need to always represent a change in the system.

The `trips:trip_updated` event is published when a trip is created or updated, which notifies the services in the platform that something has changed inside the `trip-service`. This event will be specifically consumed by `state-service` to get the most recent taxi's coordinates. Once the event is consumed, the `state-service` makes a HTTP GET request to the `/v1/trips/<trip_id>` endpoint and uses the last coordinates specified in the polyline. The event's payload is presented in the Listing 4.5 below.

```
{
  "trip_id": "c73189c6-8e83-4c95-8622-bf872c06f350",
  "taxi_id": "b098809c-308c-4214-a852-a1aed58b2148"
}
```

Listing 4.5: The `trips:trip_updated` event payload.

The `trips:trip_finished` event is published when a trip has finished. This event will be consumed by the `rating-service`. This event is then used to email the user to rate the trip taken. This interaction is later detailed when describing the rating service. The event's payload is presented in the Listing 4.6 below.

```
{
  "trip_id": "0ce72caf-047d-4880-b8cc-c6fbffc1e589",
  "taxi_id": "29ec40fc-0584-477b-8a19-6d8e503afe69",
  "user_id": "a4282450-7692-44f2-8bc2-fef7c9e66e18"
}
```

Listing 4.6: The `trips:trip_finished` event payload.

4.6 Rating service

As presented before, the rating service is responsible for providing mechanisms for a client to rate a given trip. Once a trip is completed, the rating process begins. The `trip-service` publishes a new event that is then consumed by the `rating-service`, which sends an email requesting the user to rate the trip.

An email can be sent using the Simple Mail Transfer Protocol (SMTP). `Flask` provides the `Flask-Mail`[25] library, which defines an SMTP client that can easily interact with a remote SMTP service. For

[25] https://pythonhosted.org/Flask-Mail/

the purpose of this functionality, MailTrap[26] was used to simulate a service forwarding emails to the users.

This service only has one single class, which stores the rating details for a specific trip, which is linked to a taxi and an user. This class is present in figure 4.5.

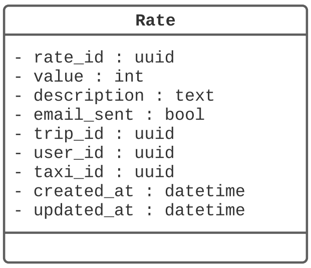

Figure 4.5: Rate class defined in rating-service.

The Rate class contains the rate unique identifier, the trip identifier, the user and taxi identifiers involved in the trip, a flag that dictates if the email was sent, the value assigned by the user, an optional description and when the rate was created and updated. The value assigned varies between 1 and 5.

This service exposes two endpoints: one used to retrieve a given rate and another that is used to rate the trip. The first one can be used to validate if a trip has been rated before. Both endpoints are listed on Table 4.4.

Method	**Endpoint**	**Description**
GET	/v1/ratings/<rate_id>	Retrieve rating
PATCH	/v1/ratings/<rate_id>	Update rating

Table 4.4: Rating service API endpoints.

This service includes a consumer that is subscribed to the trips:trip_finished event that trip-service publishes. When this event is consumed, a new rate entry is created based on the event payload information.

The trips:trip_finished payload contains the trip_id, user_id and taxi_id. Once again, each event should only contain the minimum identifiers to perform the necessary requests.

Only after this event is handled (the email was not sent yet), the service requests the state-service to get the user and taxi information, including their names and the user email. In this case, the service does not request the trip information because there is no useful information required. It

[26] https://mailtrap.io/

would be required to request extra information if the `trip-service` had the names of the starting and stopping trip locations.

4.7 Authorisation service

The authorisation system will be based on traditional Role Based Access Control (RBAC). In RBAC systems, users are assigned to one or more business-facing roles (e.g. Administrator, Manager, Consultant, User, etc). Those roles grant the user a set of permissions, which are typically defined as some mode of access to a resource. This mapping from users to roles, and from roles to permissions is frequently referred to as a policy.

In such a policy the specific permissions are defined by the platform. The different services can define the actions/permissions that are permitted on their resources, but should not know about specifics of roles that are assigned to users, which means each service should fetch the user policy to determine if the user has permission to perform a given action. Each role is defined by the permissions that are assigned to it.

Various permissions were defined in each service. It is likely that more permissions will be defined later if new business logic is added to the platform.

Each permission is added manually to the `authorisation-service`, and the roles are modified or generated according to the platform business logic. The user's final policy is generated by combining the role-permission mapping, which creates a list of permissions that an user has.

The service holds three classes that allow the authorisation management using the RBAC approach. These classes and their relationship are presented in Figure 4.6.

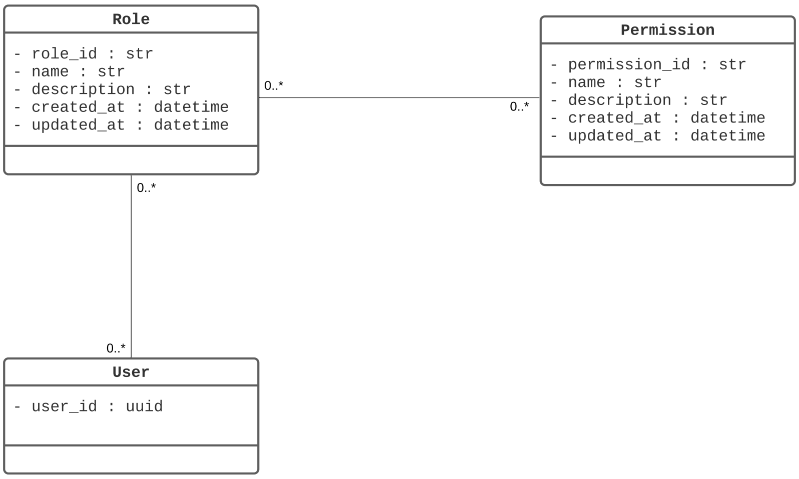

Figure 4.6: Authorisation service class diagram.

The class Role contains the role unique identifier (UUID), the name and description, and when it was created and updated.

The class Permission contains the permission unique identifier (UUID), the name and description, and when it was created and updated.

Finally, the class User which only contains the user unique identifier.

The relationship between the different classes are also presented in Figure 4.6, and both are N-M relationships. Each user can have multiple roles and each role can be associated to multiple users. Each role may also have multiple permissions and each permission may be associated to multiple roles.

To manage these resources, the service exposes some endpoints that allow the management of roles and permissions. These endpoints are listed on Table 4.5.

Method	Endpoint	Description
PUT	/v1/user-roles/<user_id>	Upsert user roles
GET	/v1/policy/<user_id>	Retrieve user policy
GET	/v1/permissions/	List available permissions
PUT	/v1/permissions/<permission_id>	Upsert permission
GET	/v1/permissions/<permission_id>	Retrieve permission
DELETE	/v1/permissions/<permission_id>	Remove permission
GET	/v1/roles/	List roles
PUT	/v1/roles/<role_id>	Upsert role
GET	/v1/roles/<role_id>	Retrieve role
DELETE	/v1/roles/<role_id>	Delete role

Table 4.5: Authorisation service API endpoints.

When a new user is registered in the platform it is important to define the roles the user will be able to assume. When there is some control of users who register in the platform, it becomes easy to register and also give the correct roles. However, in platform such as the one being described, it is important that the user roles are automatically defined during registration.

Thus, during the creation of a new user or taxi, the state-service requests the authorisation-service to upsert the user roles, using the first endpoint presented on Table 4.5. This guarantees that all created users have some set of permissions that enables them to use the platform.

The permissions required in each service are presented in Table 4.6. It was used the same approach defined for events, where a prefix was used to easily identify the service that owns the permission.

Service	Endpoint	Permission
state-service	GET /v1/taxis/	states:list_taxis
	POST /v1/taxis/	states:create_taxi
	GET /v1/taxis/<taxi_id>	states:retrieve_taxi
	PATCH /v1/taxis/<taxi_id>	states:update_taxi
	GET /v1/users/	states:list_users
	POST /v1/users/	states:create_user
	GET /v1/users/<user_id>	states:retrieve_user
	POST /v1/auth/token/	-
	GET /v1/certs/jwks.json/	-
	POST /v1/certs/jwks/	states:create_key
trip-service	POST /v1/trips/	trips:create_trip
	PATCH /v1/trips/<trip_id>	trips:update_trip
	GET /v1/trips/<trip_id>	trips:retrieve_trip
rating-service	GET /v1/ratings/<rate_id>	ratings:retrieve_rating
	PATCH /v1/ratings/<rate_id>	ratings:update_rating
authorisation-service	PUT /v1/user-roles/<user_id>	auths:upsert_user_roles
	GET /v1/policy/<user_id>	auths:retrieve_policy
	GET /v1/permissions/	auths:list_permissions
	PUT /v1/permissions/<permission_id>	auths:upsert_permission
	GET /v1/permissions/<permission_id>	auths:retrieve_permission
	DELETE /v1/permissions/<permission_id>	auths:delete_permission
	GET /v1/roles/	auths:list_roles
	PUT /v1/roles/<role_id>	auths:upser_role
	GET /v1/roles/<role_id>	auths:retrieve_role
	DELETE /v1/roles/<role_id>	auths:delete_role

Table 4.6: Permissions required on each service.

It is possible to identify that some endpoints don't require any permissions. These endpoints are defined in `state-service` and they were designed to be exposed publicly. As discussed before, the first one can be used to generate a new access tokens and the second on can be used to list the available JWK set.

Once these permissions are well defined and established on each service, it is important to define the available roles in the platform. It has been identified 3 actors in the system: the administrators, the normal users and taxi drivers. For that reason, .

Table 4.7 shows the three different roles that were created, as well as the permissions that each role contains.

Role	Permissions
admin	states:list_taxis states:create_taxi states:retrieve_taxi states:update_taxi states:list_users states:create_user states:retrieve_user states:create_key auths:upsert_user_roles auths:list_permissions auths:upsert_permission auths:retrieve_permission auths:delete_permission auths:list_roles auths:upser_role auths:retrieve_role auths:delete_role
taxi	trips:create_trip trips:update_trip trips:retrieve_trip
user	ratings:retrieve_rating ratings:update_rating

Table 4.7: Roles and their permissions.

It is possible to easily add new roles and permissions if new actors or new functionalities are added to the platform. It is also important to mention that fine-grained roles may be created if required. In the context of this platform, the roles defined in Table 4.7 are more than enough to secure each service resources.

As defined in the proposed architecture, each service holds a key value store to temporarily save the user policy, which defines the user permissions.

Considering the Table 4.7, when an user policy is requested, the `authorisation-service` returns the following response.

```
{
  "permissions": [
    "ratings:retrieve_rating",
    "ratings:update_rating"
  ]
}
```

Listing 4.7: User policy example.

The policy is cached in `Redis`, therefore available to all containers and processes running a given service. It could be considered running this cache in memory on each process, however this could raise some input/output issues on the `authorization-service`, as more requests would be required to validate policies.

If there are any limitation on the caching server or the number of users in the platform starts growing, the cache should be capped in terms of the number of entries.

Policies that remain in the cache should be automatically refreshed based on some Time To Live (TTL). If this value is too small unnecessary load will be placed on the `authorization-service` when fetching a policy that hasn't changed.

Even if it is not common, changes in policies should be detected and propagated quickly. To avoid security problems, cached policies must be invalidated or modified as fast as possible.

This is a great example of when to use the event bus. The `auths:policy_updated` event is published hen the user policy has changed. This event is consumed by all services, excluding the `authorisation-service` which is capable of validating permissions directly. The event payload only contains the user identifier and it is present on Listing 4.8.

```
{
  "user_id":"6438fffe-25a5-49f0-80a1-d35d173fa852"
}
```

Listing 4.8: The `auths:policy_updated` event payload.

It could be worth thinking about getting the new policy as soon as the event is received. However, if multiple policies are modified at the same time, multiple events will be published, which means fetching the new user policy can cause an unnecessarily overloading in the `authorization-service`. As a result, it was decided to only retrieve the new policy when the authenticated user does a new request.

Once almost all requests are protected with this authorisation logic, it is important to allow internal requests to pass through this validation.

In Section 4.8 it is discussed how services know who is the user requesting a given resource. Basically, the API gateway will forward the user identifier inside an HTTP header. However, since internal requests don't hit the API gateway, the services will not be able to get this new header. For that reason, it was decided that all internal requests should add this header to each request. The header value will be constant and known by all services. The services will have it some kind of white list and will always authorise requests.

Now that `authorisation-services` is described, it is important to state how services are able to validate if an user has authorisation to perform some actions on the requested resources.

A new library called `Auths` was created to abstract how services perform policies validation. This process is described in the next section.

4.7.1 Auths library

As described before, each service must implement some business logic to verify if an user is authorised to perform a given action. Thus, the `Auths` library was created to abstract how this process should be accomplished.

On Section 3.1 it was mentioned that a key value store would be used by each service to cache the user policies in order to reduce the load on `authorisation-service` through the reduction of requests to get the user policies. The selected key value store was `Redis`.

Listing 4.9 shows a pseudo code on how the validation is performed.

```
def require_authorisation(permission, user_id):
    if policy_in_cache(user_id):
        policy = get_policy_from_cache(user_id)
    else:
        policy = get_policy_from_auhtorisation_service(user_id)

    if permission not in policy:
        raise UnauthorisedException()
```

Listing 4.9: Example of how permissions are validated.

To interact easily with Redis, a Python library called redis[27] was used. This provides a client that abstracts how requests are done to the Redis instance.

The Auths library provides one single class with the same name, and it is defined in Listing 4.10. The Auths class requires the base URL of authorisation-service used to get user policies and the redis client used to interact with Redis.

```
class auths.Auths
    base_url: str
        The `authorization-service` url used to fetch policies.
    redis_client: Redis
        The client used to communicate with the redis instance.

    # public methods
    require_authorisation(permission_id: str, user_id: str) -> None
        If the policy isn't already in the cache, fetch it and check if the user
        has the permission.
```

Listing 4.10: Auths class defined in Auths library to validate permissions.

As described before, all services in the platform expose HTTP API endpoints. The endpoints that require authorisation use this library to validate if an user can perform the request. If the authorisation process fails the service returns a 403 HTTP status error, so the client knows that the its access is forbidden.

4.8 API gateway

Kong[28] was the API gateway of choice to use in the platform. This is an important piece in the architecture, because it is the entry point of the system and it will forward requests to each service.

One of the good features of Kong is the integration with custom Lua plugins. This is important for this infrastructure, since it enables the addiction of business logic, such as authorisation in the system.

As discussed in 4.4, the state-service is capable of generating JWT tokens during the authentication process. In order to validate if an user is authorised to request internal services, this token must be validated. Instead of validating the token on each service, which would cause some overhead on it, the API gateway will be responsible for it, which makes sure no invalid requests reaches the internal services.

The token validation will not be performed on public endpoints. These are the endpoints exposed by state-service: the one to authenticate the user and the one to list the JWK cryptographic keys.

[27] https://pypi.org/project/redis/

[28] https://konghq.com/

A Lua script was developed to handle the security checks inside the API gateway. This script is invoked on all requests and it is responsible to:

- check requests with no or invalid JWT token;
- reject requests that are invalid or not in the allow list. Which means it will block access to any endpoint not specifically listed as an unprotected. As mentioned before, only two endpoints from state-service will be defined in the allow list.
- validate the JWT token against the JWK set returned by state-service. The API gateway is able to get this set from state-service if not already in memory;
- transform a valid JWT token into additional headers to be used on internal requests, such as X-User-ID.

Each JWT token contains the signature and the information about the key used to sign, and each JWK provides all the details about the signature. It would be possible to manually implement the JWT token validation. However, there is no need to reinvent the wheel. An open source plugin called lua-resty-opendic[29] was used to validate the JWT against the JWK set returned by state-service. It is important to mention that by default this plugin caches the key set during 1 minutes, which avoid requesting state-service multiple times.

Kong also provides some standard plugins[30] that provide additional functionalities, which might be important in production environments, such as: rate limiting, canary releases, ACL, etc. In the proposed architecture, it is also used the Correlation ID[31] plugin, which is able to correlate requests and responses using a unique identifier transmitted over an HTTP header. This identifier can later be used to track requests and debug some problems in the system.

As mention before, the Lua script is able to transform the JWT token into useful information that can be used by internal services. More specifically, the X-User-ID header is used to validate business logic authorisation through the use of Auths library.

The two sequence diagrams below show how requests are routed between the different services in the architecture.

[29] https://github.com/zmartzone/lua-resty-openidc

[30] https://docs.konghq.com/hub/

[31] https://docs.konghq.com/hub/kong-inc/correlation-id/

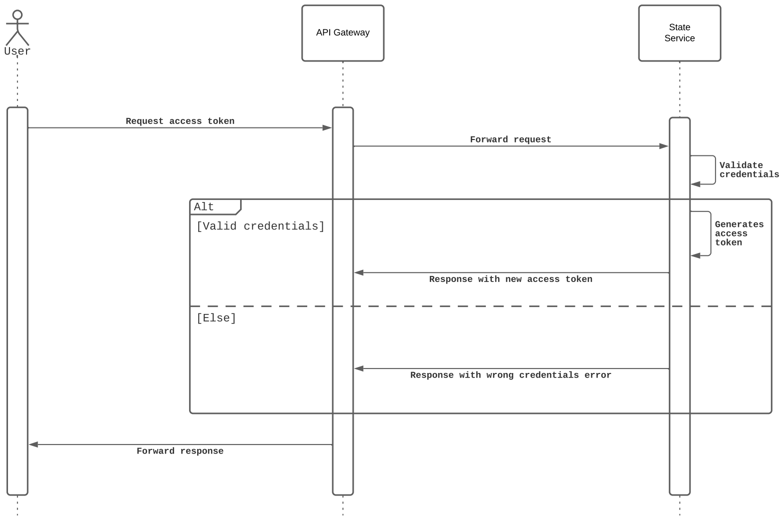

Figure 4.7: Sequence diagram that describes the authentication process.

The Figure 4.7 presents how the authentication request is performed through the API gateway and `state-service`. Is is possible to confirm that the gateway does not do any validation, as the authentication endpoint is in the white list, which means requests reach the internal services.

The Figure 4.8 shows the API gateway providing an additional security to internal services by checking upfront if the JWT token is valid. This process was described in this section. Once the token is verified, the request is forward to `rating-services` which handles the user rating. This involves verifying if the user is authorised to perform this actions, as described in Listing 4.9. If everything worked as planed, the user receives the confirmation response.

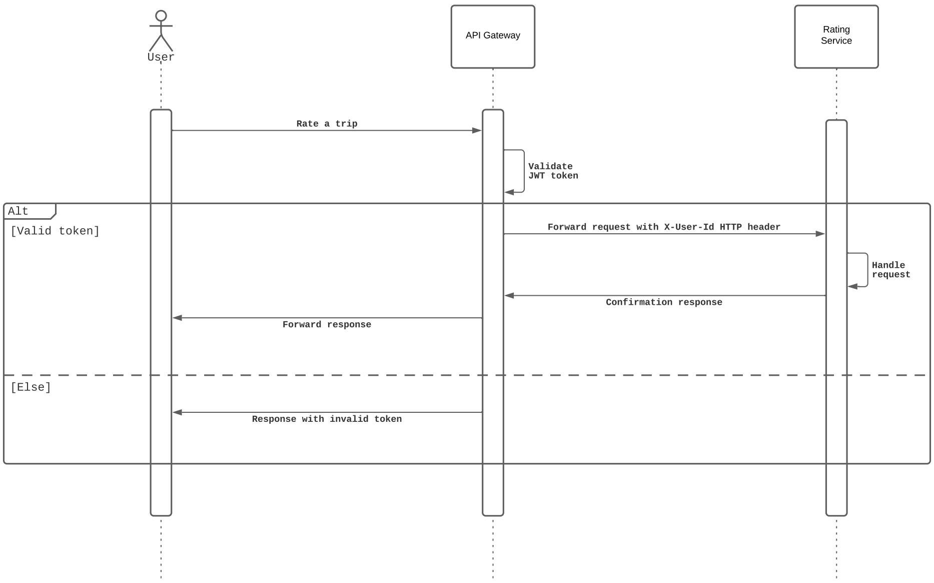

Figure 4.8: Sequence diagram that describes the rating process.

4.9 Evaluation

Throughout this chapter a microservice architecture was developed using the most common patterns. As previously stated, this focused on providing the best strategies for building a MSAs based on the research conducted.

As a result, it's important to summarise when some patterns and strategies should be used: when requests need to be forwarded to different internal services, an API gateway or a BFF should be used; when data needs to be persisted, each service should use one of the data storage patterns previously identified, with the use of one database per service being the most recommended for isolation and performance; when services require external interactions, some sort of interface must be provided, which can be accomplished by providing a REST API over HTTP; and when services need to share information internally, synchronous and asynchronous communication can be considered, and the use of each mechanism will depend on the underlying requirements.

The preceding study was found to be important in understanding which patterns exist, how to apply them, and helping in the conceptualisation of the architecture based on the use case. Although the architecture was designed for this specific use case, all of the decisions taken can be adapted to other microservice architectures. For that reason, it is important to analyse how the designed and developed architecture can help other engineers and companies building a MSA for different use cases. As an example, a music streaming platform will be considered to identify how some decisions can also be applied.

In general, this type of platform includes mechanisms for managing users, songs, recommendations and ratings. These are precisely the different contexts that can be considered as different microservices, for example: `user-service`, `song-service`, `recommendation-service`, and `rating-service`. Other services, such as a payment service, can also be required on these platforms.

The platform might require the use of an API gateway to work as the first entry point in the system, providing security and forwarding requests to each microservice within the platform. This can easily be achieved using well-known services in the market as discussed in Section 4.1, and also following the development approach described in Section 4.8.

In terms of data storage, using a database per services seems the right choice: each service might use a different database type (relational vs non-relational) and each service should only access its data in an isolated way. This was the approach considered for the proposed architecture and based on the previous research this is the most common pattern used on other architectures.

One of the key designs of microservices is to split the different contexts of the application into distinct and well-defined services, and as previously stated, four microservices can be considered. They should offer a way of communication so that it is possible to manage their resources, i.e. users, songs, etc. The most common approach is to create a well-structured API that uses the REST protocol. This offers several advantages, and it is the most straightforward method of integrating other services and clients, such as backend, frontend, and mobile applications. The API endpoints provided by each service will depend on the platform requirements.

Asynchronous communication, such as the publish/subscribe paradigm, may also be useful to this platform. Music streaming services typically recommend playlists based on the user's activities. This is an excellent example of how `song-service` can publish a message to an Event Bus when a user plays music. This message can then be consumed by `recommendation-service`, which can subsequently readjust the user's preferences. Another scenario is that customers might want to be notified when new music is added to the platform. This may be accomplished by publishing a new event from `song-service` and then the `recommendation-service` consumes it and notify each user.

On these platform, several plans are usually offered. As a result, some form of permissions management may be necessary, particularly if each plan offers distinct functionalities. The mechanism provided on the proposed architecture through the use of an `authorisation-service` appears to be a good fit for this requirement. There are also additional users in the platform that are in charge of managing it, each with their own set of roles and permissions. Thus, using RBAC mechanism might bring a lot of advantages.

Through the various observations mentioned above, it is possible to infer the study conducted and the development exposed on this chapter is capable to offer the best practices and patterns used to build an architecture based on microservices. Moreover, following them will enable the development of scalable solutions.

Conclusion 5

Better approaches in designing structures have been forced by technological innovation. IT architectures have been redesigned to make it easier to scale and add additional functionalities to the applications. Furthermore, the rise of cloud infrastructures has prompted businesses to implement flexible and scalable applications.

The goal of this dissertation was to conduct a comprehensive study of microservices-based architectures, including their patterns, a research on how they compare with other software architectures, and their advantages and disadvantages. At the end, the idea was to expose and describe the key elements that should be considered in the development of microservices-based solutions.

Scalability is one of the primary motivations for moving to a microservice architecture. The MSA, once implemented, has the potential to bring long-term benefits. However, MSA is not a one-size-fits-all solution that can be extended to any use case, so moving to this architecture style should be deliberate and well-planned.

When compared with other architectures, it is possible to conclude that MSA provides great advantages, particularly for large enterprise applications. Other architectures are preferred in some situations, such as for smaller applications, because MSA is more complex and harder to implement. Through the study carried out, it is important to also note that the way a company implements a microservice architecture is likely to vary from how large organisations implement their solutions. However, it is noticed that these architectures follow the same design patterns.

During the investigation and development, several patterns and strategies were identified. On microservices-based architectures, the API gateway, the Backend For Frontend, and various data storage patterns were recognized as being extremely popular and extensively used to create scalable architectures. In terms of strategies used in the development, it was possible to identify the synchronous and asynchronous communication mechanisms, the various encoding formats, the usage of well-defined APIs, how to implement security, and how caching might improve services performance.

The chosen use case made it possible to design and implement a MSA. It was extremely beneficial to evaluate and validate the research that has been conducted. While this was a basic use case, the primary goal was to understand how to apply existing patterns to assist those in introducing a MSA. The various patterns were successfully applied and it was possible to conceptualise the architecture.

The previous study was a valuable asset that enabled a careful and well-structured system. Despite the fact that the architecture was created for this particular use case, it is clear that all of the decisions made can be applied to other microservice architectures.

The API gateway brought a lot of advantages to the architecture. Particularly, it has provided security to the internal services. It was mainly used to forward requests to the respective services and to perform authorisation checks. This verification consisted of validating the JWT tokens and reject all requests where the token was invalid. There is still one open question that was not address during the development: whether the API gateway creates a single point of failure or a bottleneck in the system. This can possibly be addressed using the Backend For Frontend pattern, where multiple API gateways are forwarding requests to a group of services within the same context.

All services were able to communicate synchronously and asynchronously. This was very important for two different reasons: using synchronous communication allowed services to request others in the same network and wait for a response. This is important when services want to make sure something has changed on the other part with a confirmation; lastly, using asynchronous communication allowed services to notify others about a change in the system. The synchronous communication was done over HTTP and using REST. This was used internally and externally, and it has enabled two parties to communicate in a well-structured and well-designed manner. Following the RFC designs has also allowed the development of an API that followed the best practices. The asynchronous communication was done using an event bus called AWS EventBridge. It allowed services to react to system changes based on the different events published on the bus.

The API gateway was not sufficient to protect internal services. It was important that each user had only the permissions that its role allowed it to have in the system. Almost all applications have this requirement, where different users with different roles interact with the services in different ways. This was possible to achieve using the `authorisation-service`. It was responsible for managing permissions using a RBAC mechanism, where users have multiple roles and each role has multiple permissions. There are two drawbacks to this approach: the first is that the platform administrator must specify the system's roles and permissions. Even if this is not changed often, and most of the cases only when a new services or functionality is introduced to the platform, it may be a time-consuming task; lastly, this service contains all of the system's permissions, and without them, all other services can fail. This means that a lot of monitoring and logging should be taken in place once the services are running in production, in order to guarantee the availability of the platform.

Last but not least, two libraries were created during the development phase. This is very common in a business development environment. Not only were these libraries able to abstract interactions with other services, but they were also able to minimise code duplication in the different platform services. The `eventus` library was developed to abstract the process of publishing and consuming events. As previously discussed, AWS services were used during production because they were well suited for this sort of communication. Publishing events to AWS EventBridge was not hard to implement and having a library that formats and publishes events directly on a cloud services was very useful and practical. Furthermore, having a library which has a well defined interface, might help in the future migrating to other services. Consuming events from AWS SQS was not as easy as publishing events, because there was the need of understanding the functionalities that SQS had to offered and how these could be used and implemented, for example the long polling. This library helped to abstract how events were handled. Bigger teams would easily benefit with this library, helping them to focus on building services and forgetting the interaction with AWS or other services.

The created platform was not tested in production since it would not be possible to use data provided by real users without requesting permission to companies related with this business, and monitoring would also be essential to keep track of which services were operating properly. Despite the fact that this was only tested locally, microservices foundations can be seen in the development of these services, and all the decisions were taken based on the study carried out.

In complex real-world business environments, effective adoption of microservices requires a deep knowledge of the application domain in order to determine service boundaries. Furthermore, communication overhead is common with these fine-grained services, which can reduce the architectural overall benefits.

Microservices were greatly influenced by large organisations like Netflix and Amazon. They began experimenting them to improve the speed and scalability of their applications. Martin Fowler and Chris Richardson had a great impact on this architectural style as well, and their articles and books provided in part the research material for this dissertation. It is also worth noting that cloud service providers' technology advancements, as well as the use of containers, have contributed to the evolution and progression of microservices.

Based on the study and development approached throughout this dissertation, it might not make sense to start a microservice architecture for small applications. This type of architectures require a deep understanding and expertise on the implications of adopting this style. As a result, it might make sense to start with a single application but always with the idea of having well-defined contexts and boundaries, such that the main application can be migrated or even expanded in different services in the future. However, if the development team has a lot of experience and there is the need of building a scalable application, it might be better to go straight to a microservice architecture.

5.1 Prospect for future work

During research and especially in the development of the proposed system, some ideas that could help and enhance the implementation of a microservice architecture came up.

Companies are increasingly adopting microservices architectures, and others are also attempting to move out from monolithic systems. In the future, it would be useful to explore which patterns are used when migrating these applications to microservices. There are several articles and books discussing this problem, so identifying the most common patterns would be helpful.

The deployment of microservices in a cloud computing environment are not discussed in this dissertation. Future work can be developed to present how microservice architectures are implemented using the cloud, and which technologies are used to programmatically deploy services on these environments.

Old systems are usually validated using unit and functional tests. They are very important to give the confidence that the code does what it is supposed to do and to make maintenance simpler. Each service in a microservices architecture should be tested in the same way. However, the most difficult aspect in microservices is to identify how different services interact with each other. For future work it would be interesting to research how a microservices-based architecture can be tested and validated.

Bibliography

[1] Nuha Alshuqayran, Nour Ali, and Roger Evans. “A systematic mapping study in microservice architecture”. In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2016, pp. 44–51.

[2] Apache. *Apache Avro*. [Accessed 2 Sep. 2020]. url: http://avro.apache.org.

[3] Apache. *Apache Thrift*. [Accessed 2 Sep. 2020]. url: http://thrift.apache.org.

[4] AWS. *Amazon SQS short and long polling*. [Accessed 31 Mar. 2021]. url: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-short-and-long-polling.html#sqs-long-polling.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”. In: *IEEE Software* (May 2016), pp. 42–52.

[6] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. “Hyperbolic Caching: Flexible Caching for Web Applications”. In: 2017.

[7] Eric Braun, Thorsten Schlachter, Clemens Dupmeier, Karl-Uwe Stucky, and Wolfgang Suess. “A generic microservice architecture for environmental data management”. In: *International Symposium on Environmental Software Systems*. Springer. 2017, pp. 383–394.

[8] Miguel A Brito, Jácome Cunha, and João Saraiva. “Identification of microservices from monolithic applications through topic modelling”. In: (2020).

[9] C. Fan and S. Ma. “Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report”. In: *2017 IEEE International Conference on AI Mobile Services (AIMS)*. 2017, pp. 109–112.

[10] Jianhua Feng and Jinhong Li. “Google protocol buffers research and application in online game”. In: *IEEE conference anthology*. IEEE. 2013, pp. 1–4.

[11] Martin Fowler. “Domain Event, 2005”. In: (2005). [Accessed 7 Apr. 2021]. url: https://www.martinfowler.com/eaaDev/DomainEvent.html.

[12] P. D. Francesco, I. Malavolta, and P. Lago. “Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017, pp. 21–30.

[13] GEOlink. *Taxi Service Trajectory (TST) Prediction Challenge 2015*. [Accessed 2 Jan. 2021]. url: http://www.geolink.pt/ecmlpkdd2015-challenge/index.html.

[14] Google. *Protocol Buffers*. [Accessed 2 Sep. 2020]. url: https://developers.google.com/protocol-buffers.

[15] Konrad Grochowski, Michał Breiter, and Robert Nowak. “Serialization in Object-Oriented Programming Languages”. In: *Introduction to Data Science and Machine Learning*. IntechOpen, 2019.

[16] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012.

[17] Kibo Hutchinson. *Distributed data caching using big data*. [Accessed 25 Jan. 2020]. Apr. 2019. url: https://dataconomy.com/2019/04/distributed-data-caching-using-big-data.

[18] Michael Jones. *JSON Web Algorithms (JWA)*. RFC 7518. May 2015.

[19] Michael Jones. *JSON Web Key (JWK)*. RFC 7517. May 2015.

[20] Michael Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015.

[21] Tanya Schlusser Kenneth Reitz. *The Hitchhiker’s Guide to Python: Best Practices for Development*. O’Reilly Media, 2016.

[22] Martin Kleppmann. *Designing Data-Intensive Applications*. O’Reilly, 2017.

[23] Pakorn Kookarinrat and Yaowadee Temtanapat. “Design and implementation of a decentralized message bus for microservices”. In: *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2016, pp. 1–6.

[24] Petar Krivic, Pavle Skocir, Mario Kusek, and G. Jezic. “Microservices as Agents in IoT Systems”. In: Jan. 2018, pp. 22–31. isbn: 978-3-319-59393-7.

[25] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. “Designing a Smart City Internet of Things Platform with Microservice Architecture”. In: *2015 3rd International Conference on Future Internet of Things and Cloud*. 2015, pp. 25–30.

[26] Paul J. Leach, Rich Salz, and Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005.

[27] James Lewis and Martin Fowler. *Microservices*. [Accessed 14 Jan. 2020]. Mar. 2014. url: https://martinfowler.com/articles/microservices.html.

[28] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, and Alfonso Urso. “A simplified database pattern for the microservice architecture”. In: *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*. 2016, pp. 35–40.

[29] Microsoft. *Domain events: design and implementation*. [Accessed 20 Apr. 2021]. url: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation#what-is-a-domain-event.

[30] Kwabena A Mireku. *Serialization and preservation of objects*. Apr. 2007.

[31] Luciano Monteiro, Raphael Hazin, Anderson Lima, Felipe Ferraz, and Washington Almeida. “Survey on Microservice Architecture -Security, Privacy and Standardization on Cloud Computing Environment”. In: Oct. 2017.

[32] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. “Predicting Taxi–Passenger Demand Using Streaming Data”. In: *IEEE Transactions on Intelligent Transportation Systems* 14.3 (2013), pp. 1393–1402.

[33] Irakli Nadareishvili. *Microservice Architecture: Aligning Principles, Practices, and Culture*. 1st ed. O’Reilly Media, 2016.

[34] Dmitry Namiot and Manfred Sneps-Sneppe. “On micro-services architecture”. In: *International Journal of Open InformationTechnologies* (2014), pp. 24–27.

[35] Sam Newman. *Building Microservices*. 1st. O’Reilly Media, Inc., 2015.

[36] Sam Newman. *Pattern: Backends For Frontends*. [Accessed 22 Oct. 2019]. 2015. url: https://samnewman.io/patterns/architectural/bff.

[37] Oracle. *Interface Serializable*. [Accessed 28 Aug. 2020]. url: https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html.

[38] Claus Pahl and Pooyan Jamshidi. “Microservices: A Systematic Mapping Study.” In: 2016, pp. 137–146.

[39] M. Pahl, S. Liebald, and L. Wüstrich. “Machine-Learning based IoT Data Caching”. In: Apr. 2019, pp. 9–12.

[40] Dewayne E. Perry and Alexander L. Wolf. “Foundations for the Study of Software Architecture”. In: 17.4 (Oct. 1992), pp. 40–52.

[41] Lukasz Plotnicki. *BFF SoundCloud*. [Accessed 10 Nov. 2019]. Dec. 2015. url: https://www.thoughtworks.com/insights/blog/bff-soundcloud.

[42] Adam Polak and Adrian Zmenda. *Microservices design patterns for CTOs: API Gateway, Backend for Frontend and more*. [Accessed 20 Jan. 2020]. Sept. 2019. url: https://tsh.io/blog/design-patterns-in-microservices-api-gateway-bff-and-more.

[43] Python. *Python object serialization*. [Accessed 28 Aug. 2020]. url: https://docs.python.org/2/library/pickle.html.

[44] Chris Richardson. *Microservices Patterns: With examples in Java*. 1st ed. Manning Publications, 2018.

[45] Chris Richardson. *Pattern: Shared database*. [Accessed 23 Jan. 2020]. 2019. url: https://microservices.io/patterns/data/shared-database.html.

[46] A. Sill. “The Design and Architecture of Microservices”. In: *IEEE Cloud Computing* 3.5 (2016), pp. 76–80.

[47] *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2008.

[48] Audie Sumaray and S. Kami Makki. “A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform”. In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. 2012, 48:1–48:6.

[49] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. “Architectural Patterns for Microservices: A Systematic Mapping Study.” In: 2018, pp. 221–232.

[50] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Valente, and Eduardo Figueiredo. “Microservices in Practice: A Survey Study”. In: Sept. 2018.

[51] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *2015 10th Computing Colombian Conference (10CCC)*. 2015, pp. 583–590.

[52] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. “GRPC: A Communication Cooperation Mechanism in Distributed Systems”. In: *SIGOPS Oper. Syst. Rev.* 27.3 (July 1993), pp. 75–86. issn: 0163-5980.

[53] Yali Wang, Yang Zhang, and Junliang Chen. *An SDN-based publish/subscribe-enabled communication platform for IoT services*. 2018.

[54] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. "Data mining with big data". In: *IEEE transactions on knowledge and data engineering* (2013), pp. 97–107.

[55] Xinyang Feng, Jianjing Shen, and Ying Fan. "REST: An alternative to RPC for Web services architecture". In: *2009 First International Conference on Future Information Networks*. 2009, pp. 7–10.

[56] Rongxu Xu, Wenquan Jin, and Dohyeun Kim. "Microservice Security Agent Based On API Gateway in Edge Computing". In: *Sensors* 19.22 (2019). issn: 1424-8220.

Appendix A

A.1 Dataset

The Taxi Service Trajectory (TST) dataset is the result of information collected during one year by taxis in the city of Porto. It contains 1710671 entries and a total of 9 attributes [13], described as follows:

- TRIP_ID a string that represents a unique identifier for each trip;
- CALL_TYPE a char that identifies how the service was requested. One of three potential values can be assumed:
  - "A" if this trip was dispatched from the central;
  - "B" if this trip was demanded directly to a taxi driver at a specific stand;
  - "C" otherwise (i.e. a trip demanded on a random street).
- ORIGIN_CALL an integer that represents a unique identifier for each phone number which was used to demand, at least, one service. If CALL_TYPE equals to "A" it identifies the trip's customer. Otherwise, a NULL value is assumed;
- ORIGIN_STAND an integer that represents a unique identifier for the taxi stand. It identifies the starting point of the trip if CALL_TYPE is "B". Otherwise, a NULL value is assumed;
- TAXI_ID an integer that represents a unique identifier for the taxi driver that performed each trip;
- TIMESTAMP an integer that identifies the trip's start using a Unix Timestamp (in seconds);
- DAY_TYPE a char that identifies the day type of the trip. One of three potential values can be assumed:
  - "B" if the trip started on a holiday or any other special day;
  - "C" if this trip started on a day before a type-B day;
  - "A" otherwise (i.e. a normal day, including workdays and weekends).

- POLYLINE a string which contains a list of GPS coordinates in WGS84 format. The first pair of coordinates represents the start of the trip and the last one the final destination. Each pair of coordinates identify the longitude and latitude on each 15 seconds of the trip.

- MISSING_DATA a boolean that identifies if the GPS data stream is complete or if there are values (locations) missing.