P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

MICROSERVICE-BASED INTEGRATION
FRAMEWORK FOR A BACK-OFFICE SOLUTION
LUÍS, PEDRO

P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

MEI MESTRADO
ENGENHARIA INFORMÁTICA

MICROSERVICE-BASED INTEGRATION FRAMEWORK FOR A BACK-OFFICE SOLUTION

LUÍS, PEDRO

10/2020

LUÍS, PEDRO. MICROSERVICE-BASED INTEGRATION FRAMEWORK FOR A BACK-OFFICE SOLUTION

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO
POLITÉCNICO
DO PORTO

P.PORTO

MEI
MESTRADO
ENGENHARIA INFORMÁTICA

MICROSERVICE-BASED INTEGRATION FRAMEWORK FOR A BACK-OFFICE SOLUTION

PEDRO FILIPE MARCOS LUÍS

RICARDO JORGE DA SILVA SANTOS

Acknowledgements

I would like to start this document by thanking the School of Management and Technology for allowing me to pursue my passion for software and providing me the tools that I need to face the challenges in my future in the field. I would also like to thank all the professors that helped me throughout my academic path. And a particularly special “thank you” to Professor Ricardo Santos for the help and guidance, not only in the conclusion of this document, but for the assistance and availability since day one. A special “thank you” is also due to Hugo Conceição, Jumia’s CEO for proposing the current project as well as to all Jumia’s team and colleagues that helped me in this crazy journey. I also want to thank all my family and friends for your support and patience in the hardest of times. And last, but not least, a very warm and special thank you to Monika Merk, for all the unconditional support and love through good and though times.

To all the mentioned and unmentioned, a deep and warm *Thank You*.

Abstract

Not long ago, monolithic applications ruled among production servers – these applications had massive scopes which made them difficult to maintain, with constraints of libraries shared between modules and where every change or update is attached with big downtimes.

To stray from this approach, enterprises chose to divide their big applications into smaller ones with fewer responsibilities, a clearer notion of boundaries and for the better part of it, more maintainable and scalable. The microservice approach allows enterprises to better divide themselves among teams that follow the full stack and spectrum of development in each application, from the persistence layer through the API and to the client, and from planning, through development to later support. The project exposed in this paper enlightens the scenario of an e-commerce platform's back-office - where the implementation of a strangler pattern divided a large monolithic application into smaller microservices – leaving the door open for the integration of the multiple client applications to interconnect.

The proposed solution intends to integrate the various systems of Jumia and take on this exposed opportunity, resorting to a microservice architecture and integration patterns with the objective of easing the flow of operations for processes that involve several management tools.

Keywords: Microservices, Microservice Integration Patterns, Back-Office Integration

Resumo

Recentemente, o desenvolvimento de aplicações mudou à escala mundial, os sistemas distribuídos permitiram a introdução de um novo paradigma. Este paradigma baseia-se na redução de uma grande aplicação (monólito) em pequenos sub-módulos (micro-serviços) que comunicam perfeitamente entre si como se de uma única aplicação se tratasse. Este paradigma veio também refrescar as estruturas internas das empresas, ao distribuir os diversos serviços entre equipas, de forma a que cada uma delas esteja presente em todo o ciclo de vida das aplicações, desde o conceito até ao lançamento, passando pelo desenvolvimento e posterior manutenção e suporte da mesma. As mesmas equipas são também responsáveis por toda a *stack* que cada micro-serviço contém partindo da *user interface* (UI), passando por toda a API que contém a lógica de negócio até à camada de acesso de dados.

Esta nova abordagem oferece algumas vantagens quando comparada com outras soluções disponíveis no mercado, tais como a liberdade de cada um dos serviços em ser desenvolvido nas tecnologias e linguagens que melhor se adequam ao seu propósito, sem que estejam presas a uma decisão tomada numa ocasião anterior para um propósito diferente ou a restrições de dependências incompatíveis entre si.

Sendo que um dos principais problemas da computação distribuída é a possível indisponibilidade de cada um dos seus intervenientes, a arquitetura orientada a micro-serviços (*microservice architecture,* MSA) prevê que cada um dos seus serviços esteja contido no seu contexto (*bounded context*) e que disponha de todos os dados que lhe correspondem, desta forma a indisponibilidade de qualquer serviço não deve impactar o desempenho de nenhum dos seus pares.

A reduzida dimensão de cada um destes serviços permite a existência de processos de *deploy* mais rápidos o que acaba por se refletir em *downtimes* mais reduzidos. Outra das vantagens da redução das dimensões e dos contextos de cada um dos serviços é a sua fácil manutenção, uma vez que o código se torna mais conciso e específico ao propósito que prevê cumprir. A modularidade dos micro-serviços permite-lhes também ajustar o número de réplicas de cada um deles de forma independente de acordo com as necessidades e previsões de volume de tráfego a cada momento. Apesar de todas as vantagens acima expostas, uma MSA traz consigo também alguns desafios tais como os testes de integração, *debugging*, *deploying,* retrocompatibilidade com outros serviços, entre outras abordadas em maior detalhe neste documento.

O projeto exposto neste documento é um projeto proposto pela Jumia, uma empresa que disponibiliza uma plataforma de comércio *online* no continente africano. Esta plataforma está disponível em onze países africanos com mais de cem armazéns espalhados por todo o continente e que conta com mais de cinco mil colaboradores espalhados pelo mundo. Tal como muitas outras empresas no mercado a Jumia idealizou os seus processos de operações numa aplicação única que controlava todos os fluxos de negócio e continha em si toda a informação de armazenamento, produtos, entregas, pagamentos, encomendas entre outras. Rapidamente a aplicação de *back-office* da Jumia tornou-se insustentável e, tal como tinha sido executado noutras empresas do mesmo ramo, foi implementado um *strangler pattern*. Desta forma tornou-se possível fazer uma separação de dependências gradualmente, isolando cada um dos processos de negócio num serviço independente que persiste todos os dados necessários para a execução de cada uma das operações. No entanto, a implementação deste padrão deu origem a uma lacuna nos processos da empresa, uma vez que cada um dos serviços possui o seu *user interface*, algumas das operações requerem que os agentes de operações transitem entre aplicações, e necessitem de se autenticar novamente. Este processo acaba por ter impacto no fluxo de operações, refletindo-se no número de encomendas processadas e por consequência nas receitas da empresa. O presente documento pretende explorar a oportunidade de negócio proposta, assim como os mais essenciais padrões de integração de micro-serviços, de forma a apresentar uma solução que consiga colmatar a lacuna apresentada sem pôr em causa a segurança das aplicações e as normas de conformidade exigidas. Esta proposta foi elaborada através da conceção de uma arquitetura orientada a micro-serviços de forma coreografada tendo como objetivo ser integrada nas diversas aplicações de Back-Office com recurso a uma biblioteca importada através do gestor do Node Package Manager.

Palavras-chave: Micro-serviços, Padrões de Integração de Micro-serviço, Integração de Back-offices

Table of Contents

Table of Figures

Table of Tables

Table of Listings

Acronyms

ACID – Atomicity, Consistency, Isolation, Durability

ACL – Access Control List

API – Application Programming Interface

AWS – Amazon Web Services

BASE – Basically Available, Soft state, Eventually Consistent

BOIA – Back-Office Integration Application

CAP – Consistency, Availability and Partition Tolerance

CDC – Consumer-Driven Contract

CLTV – Customer Lifetime Value

CRUD – Create/Read/Update/Delete

DAO – Data Access Object(s)

DDD – Domain Driven Design

DEVOPS – Development of Operations

HTTP – Hypertext Transfer Protocol

IDE – Integrated Development Environment

IPC – Inter-Process Communication

MS – Microservice

MSA – Microservice Architecture

NPM – Node Package Manager

OMS – Order Management System

ORM – Object Relational Mapper

REST – Representational State Transfer

RSA – Rivest–Shamir–Adleman (Encryption Algorithm)

SOA – Service Oriented Architecture

SOAP – Simple Object Access Protocol

SRP – Single Responsibility Principle

TDD – Test Driven Design

UI – User Interface

URL – Uniform Resource Locator

WMT – Warehouse Management Tool

1. Introduction

E-commerce fights a constant battle against time, enterprises strive when they can consistently deliver their products fast, safely, and reliably. A delivery that fills these requirements, is more likely to help a new customer overcoming the lack of trust that is inherent to online shopping, converting that customer into a potential repeat costumer, and help the repeat costumers cement the relationship they have with the company – thus enlarge the costumer's lifetime value (CLTV) [1]. To attain speed and reliability when working in a microservice environment, enterprises need to possess robust and decoupled processes that ensure a non-blocking flow of packages from the vendor to the end-user, those processes can only be achieved through a seamlessly integration between the different tools used by the back-office that perform them.

The presented project dwells on a solution for integration between back-offices, from Jumia – an enterprise that operates in the African market, through an e-commerce platform, that is available in eleven countries, with more than one hundred warehouses across the continent and more than five thousand workers. Currently, after the implementation of a strangler pattern [2] that split a single management tool into several, the enterprise was left with an integration problem that require agents to switch between applications to complete the flows necessary to process an order. The permute between applications is always followed by a new authentication, which ends up slowing the process ever so slightly, impacting operations and creating entropy that reflects upon the speed at which orders are processed.

The proposed implementation described in this document follows a microservice architecture (MSA), in order to integrate seamlessly the system currently in production at Jumia, the project also referred to as Back-Office Integration Application (BOIA) communicates asynchronously with the application programming interface (API) that manages the users and applications, listening for upserts to update its own storage layer with the relationships between users and applications.

The project also possesses a frontend layer, that is meant to be imported by other back-office applications, this frontend library will only be responsible for the application switch component, which means it will have a very limited scope and it will be highly reusable. This requirement suits the system favourably, given the high number of sharded applications in Jumia's Back-Office.

1.1. Reference Scenario

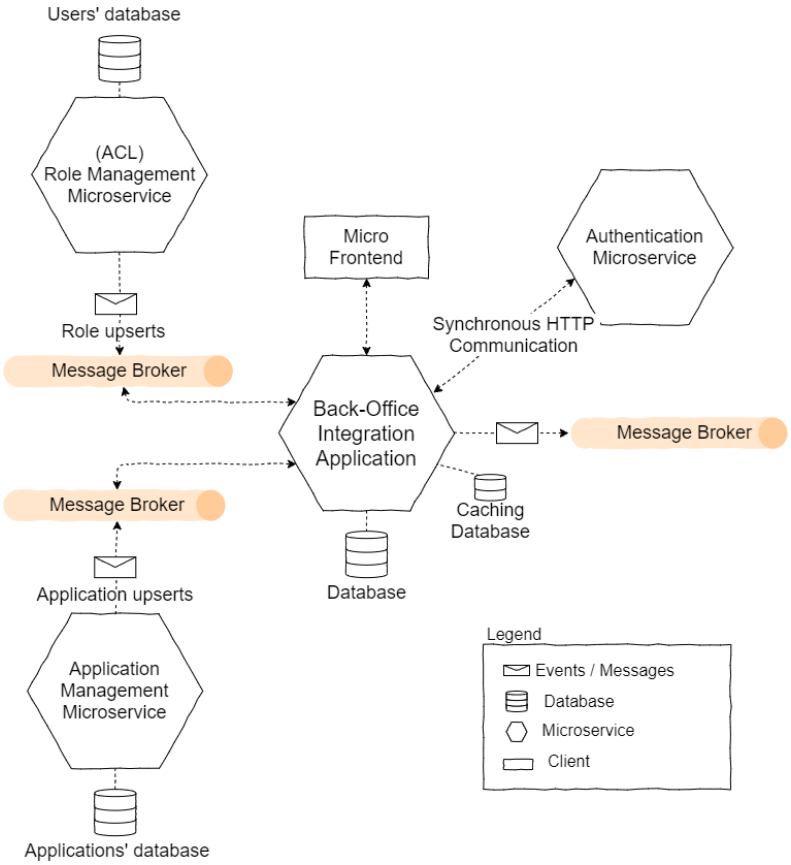

Figure 1- BOIA's architectural integration in Jumia's environment

The chosen Reference Scenario depicted in Figure 1, displays a microservice architecture that integrates BOIA, with other Jumia Service's microservices. In this scenario, BOIA consumes messages asynchronously from the application's and user's message brokers to which the microservices responsible for the applications and users, respectively, publish updates about their data. BOIA also publishes its updates to a message broker, in order to be

used by other applications. It can also be observed that BOIA communicates synchronously with the Authentication Microservice to validate the sessions of all its requests and to generate tokens whenever needed. The client communicates with the API via simple HTTP requests.

1.2. Objectives and Expected Results

The main objective of this document is to propose an integration application that allows agents of operations to switch between applications without the need for them to authenticate, while respecting the current processes at place, as well as the compliance requirements of the systems involved. Additionally, this document intends to provide a body of knowledge that will aid the better understanding of microservices, as well as their communication and integration patterns between them.

The study of this document together with the proposition of the consequent project expect to achieve:

- A proposal of a system able to achieve a reduced number of logins performed by users when transitioning between applications.
- A proposal that assists users visualizing the applications they have access to, and as such reducing the time looking for applications.
- A proposal of an overall faster workflow performed by the users when executing actions across multiple applications.
- A better understanding of microservices and how they can integrate with each other.

1.3. Document Structure

This document is structured in seven main chapter, the first and current is comprised of a brief introduction to the document and the project. The second chapter will consist of a deep analysis over the subject of microservices and their integration and interaction in a real-world scenario. The third chapter overviews the state of the art of microservices in today's era. On the fourth chapter the real-world necessity of Jumia will be showcased and analysed with some detail and a proposal of solution will be formulated. The fifth chapter will focus in a high-level implementation that follows the proposed solution. Finally, the sixth chapter will be

reserved for the conclusion, where assertions will be made about the developed work and the future work will be discussed.

2. Theoretical Framework

This chapter aims to overview the overall concepts used in this document, it intends to lay ground for the knowledge and assertions of the following sections. It also aims to display the most recurring, common, and modern approaches in each of the topics that will be discussed further ahead.

2.1. Monoliths

Over time, enterprises have accumulated applications that follow a monolithic architecture [3], these accumulation is mainly linked with the combination of the fast change in technologies together with the rapid and widespread evolution of applications to large scales. These changes tend to make architectures brittle and error prone [4], rapidly scaling an application out of the proportions where it is viable.

But what really is a monolith? – A monolithic application is an application that provides all of its services within a single code base shared among multiple modules and developers [5]. This usually inherently means that once a developer or a team, want to implement a new service, they have to do so employing the same technologies in which the stack is built, and forcefully disregarding what would be the ideal technology to accomplish the task. In monoliths after each development, a thorough battery of tests must be ran through to make sure that the whole product is running smoothly and as intended, this happens given the complex juxtaposition of the different components of the application in which a single change in a module can inadvertently affect its peers. Despite this, monoliths are rather painless to test and to develop for, as the manner in which Integrated Development Environments (IDE) have evolved, makes it so that they are ideal for this sort of architecture [6]. A monolithic application usually follow a three layer pattern (Figure 2), which is composed by a user interface layer, a business logic layer and a data access layer [3]. This makes the architecture of these applications furtherly fragile, in the sense that whenever the application is unavailable then all of its components are also unreachable.

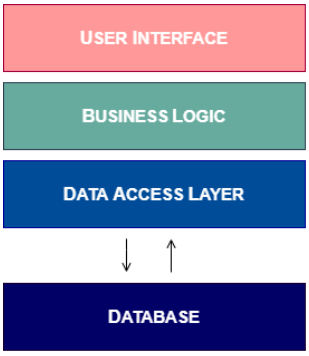

Figure 2 – Diagram of a monolithic architecture

Monolithic applications are effortless to scale, yet they limit scalability. The most common strategy to balance the load of these applications, on occasions where the traffic increases, is to create new instances of the same applications and split the load among them. Although this increase in traffic might only apply to a subset of the modules, this process makes the resources it spends excessive and costly to enterprises. An additional characteristic of monoliths regarding DevOps is the fact that upon deployment of the application, developers must choose the correct deployment environment and provider, since some of the dependencies of such a large application can be performance-intensive, usually requiring teams to compromise in a one-size-fits-all solution [7]. Although, as a study from 2016 from the University of Bogotá [8] came to conclude, applications with this type of architecture tend to be more expensive to keep in cloud-based solution than its counterparts.

Despite the long reign of monoliths, the largest and most influent enterprises in the market, like Amazon, Netflix, LinkedIn, Google and others, seem to be setting tone to stray away from these implementations [5], making room for better-scaling technologies that allow non-blocking development among teams and reducing the costs of enterprises for load balancing processes in traffic intense moments.

2.2. Microservices

The popularity of microservices has been increasing in software-oriented companies in the last few years, the sub-chapter ahead will focus on explaining what is a microservice, how they came to be, and also overview their advantages and disadvantages. Although already being found in similar implementations, gravitating around a Service Oriented Architecture (SOA), the term “microservice” was introduced in May 2011 in a workshop near Venice, by a group of software enthusiast, among them were Martin Fowler and James Lewis. The same group would end up agreeing on the term “microservices” (plural), as the definitive terminology

for the newly uncovered architectural approach, about a year later in May 2012 according to Fowler [9].

In order to get a better grasp on microservices, it is perhaps better to firstly define them – A microservice is a mini-application that has its own architecture consisting of business logic along with its various adapters, at runtime, each instance is often a cloud VM or a Docker container [10]. In this sense a microservice provides a business or platform capability through a well-defined API, and by doing so it provides this purpose and only this purpose – It does one thing and it does it well [11]. The APIs build for microservices tend to follow an approach of "smart endpoints and dumb pipes", that can be broken down to microservices aiming to be as decoupled and cohesive as possible – being the owners of their domain and business logic – as such, upon receiving a request, microservices simply process the received data by applying their subset of rules and producing a response. This can be achieved using a combination of REST and lightweight messaging protocols [12].

When compared with the previously referred monolithic architecture, a microservice architecture tends to come on top with regards to scalability. Given that monolithic applications scale by creating new instances of themselves in order to properly respond to an increase of traffic, disregarding which module is getting an increased number of requests. In a microservice architecture each service is provided by a single small application, as such the scaling can be performed individually according to which services require more notice. Allied with the current technologies of cloud-based solutions, provided by big players such as Amazon Web Service (AWS), Google Cloud Platform, Microsoft Azure, alongside others, that provide integrated automation solutions that make scaling automatic, effortless and cost-efficient [13].

Keeping in mind that microservices are small services that are sorted by their bounded contexts, that follow Robert C. Martin's classification of Single Responsibility Principle (SRP) defining itself by "Gather together those things that change for the same reason, and separate those things that change for different reasons" [14]. With this thought it is understood that microservices aim to have very limited contexts, therefore microservices should focus on doing only one thing and doing it well. Thus, granting an advantage regarding the development process, when compared to the prior monolithic architecture, where applications had the tendency to grow in the advent of new business requirements, making code development and bug fixing a lot easier, since the general scope of each implementation is always the same [15]. Also, regarding the implementation process, microservices are advantageous because their decoupled nature allows each team to pick the best dependencies and technologies to better suit the purpose of their service.

In short, a microservice, can be reduced to a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility [16].

With all of the above mentions, it might appear that microservices are a one-size-fit-all solution, that came to solve all the problems of the software development industry, when in reality it happens not to be a silver-bullet. As such they can be seen as an implementation that despite solving some scalability problems they bring with them a whole new complexity level to it, despite mending some problems concerning the big size of previous implementations they bring with them a complex set of integration tests. Microservices require teams to adapt and change the way they tackle development and to review some of their strategies in order to better approach this distributed system solution.

2.3. Microservice Architecture

Whilst having covered the definition of a microservice as a unit it is also important to define them as a group and to get a better grasp of how they position themselves on an architectural level. Given that a microservice application works under the sway of a load balancer, to define how many instances a certain service possesses at a certain time and moment, it is not realistic to expect any client to know the addresses of these dynamic endpoints. As such, a microservice architecture needs a mechanism that allow clients to freely request the application, and that can be achieved by exposing an API Gateway. In this matter the API Gateway serves the purpose of encapsulating the application and serving as a single-entry point to which each client then makes requests [17]. As displayed in Figure 3, an example taken from Chris Richardson's article [17], there can be seen an architecture where the clients communicate with the application via an API Gateway, that queries each of the service according to their necessity in order to retrieve the data to display to the final user. In Netflix's example, the team initially chose to develop a one-size-fit-all solution for their API Gateway, but soon realized the limitations of it, given the different specifications and requirements of each end-device. As such Netflix chose to enrich their astonishing microservice architecture by implementing a device-based API, that provides different options and strategies of access varying on what device the user is employing [18].

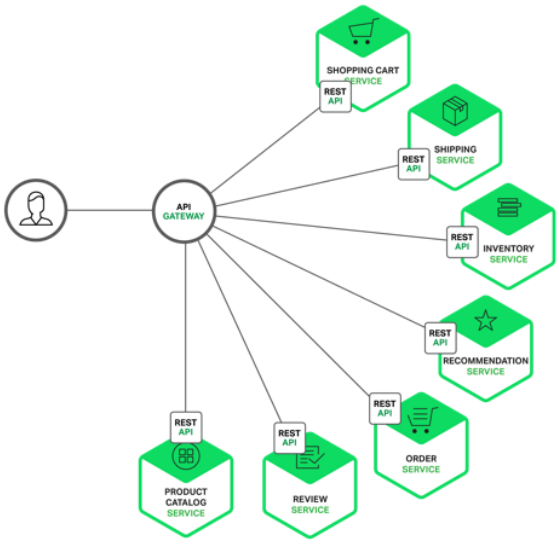

Figure 3 - Diagram of a Microservice Architecture recurring to an API Gateway

As previously mentioned in this document, microservices are the owners of their domain and business logic. Therefore, the data owned by each microservice should be private and only accessible via that microservice's API [19]. Looking at an example of an e-commerce microservice-based solution, whenever the price of a product is updated in the Catalogue's Microservice then the Catalogue's Microservice cannot, in any circumstance, update any of its peers' databases, as that constitutes a violations of their data's integrity. Since each microservice's data is strictly owned by the microservices holding them. To properly confront this issue, the communication between microservices is made via RESTful requests or lightweight messaging protocols [9], in a manner exposed more in depth in the sub-chapter "Microservice Integration Patterns" from this document.

When emerging in the market, microservices not only revolutionized the way application's architectures were designed and implemented, but they also brought revolution to the way organizations coordinated their teams. Whilst when employing a conventional three layer monolithic architecture had enterprises dividing their development teams in three different sectors, one team would develop and maintain the user interface, the second team would develop and maintain the business layer's API and the third team would tackle the database layer. Given that microservices are developed around business capabilities, in which each service is composed by a broad-stack, including user-interface, persistence storage, and any external collaborations [20]. Now enterprises find themselves adopting a strategy where they choose to employ cross-functional teams, to cover the whole stack of each service, these teams need to be able not only to develop emerging features for each service, but also to maintain it. As such these teams end up covering a wide variety of development roles, from

UI developers, to backend, testers, database analysts, product owners and sometimes others. This distribution of elements among teams was predicted by Melvin Conway, in its famous Conway's law [21]. The Conway's law states that systems are designed to mirror the teams that develop them.

2.4. Monoliths to Microservices

After discussing what is the meaning of monoliths and microservices, and how they position themselves in their own architectures, it is important to notice that none of them lasts forever. If an application started out its life cycle with a monolith, then it should not be fated that it will end as one. The reverse might also be true in some scenarios. Some applications start out as microservices and need to be converted to a monolithic application, given the complexity and entropy that seizes them, or they are simply not big enough to justify the additional effort required by such an approach. Although this subchapter will emphasise the former, the migration from monoliths to microservice architectures. It will explain some of the most common strategies used in enterprise-level applications, as well as their dangers, in order to create good cleaving between the different bounded contexts and business necessities.

When looming the subject of a migration from a monolithic architecture to a microservice architecture, there seems to be no unanimity. In some cases, one of the biggest mistakes a company can make is starting to build their architecture using microservices [22]. In an early phase of development, enterprises thrive by implementing speedy processes [23], that allow them to deliver a first product fast, in order to maximize the value returned by the development as early as possible. In this sense, a microservice architecture might not be the ideal solution for the early stages of an application. Since the size of the application in its genesis does not justify the overhead effort, that is inherited by microservices. Being so by the extra effort that needs to be dedicated to the infrastructure, to the testing of to simply define the bounded contexts that will make for a good and stable application. Another concern to have in mind, regarding the early phase of any application and the necessity of the implementation of microservices, is the proper definition of the bounded contexts. As studies suggest, "Wrong Cut" – defined by when microservices are split in the basis of technical layers instead of business capabilities – is one of the leading causes for a failing microservice architecture [24]. A refactor in a microservice integration is much harder than a refactor in a monolith, it requires the developers to interact with more moving parts which also can affect the integrity of the system. In this sense, the migration of a monolithic architecture to a microservice-based architecture is very much feasible, given that the monolithic application is well modularized from the start [25].

One of the approaches for a monolithic break down is called the strangler pattern [2]. It gets its name from the strangler fig, a vine that grows its roots in a host tree and grows upwards to reach the light above the host's canopy, whilst wrapped around the host's trunk. Often the host tree ends up dying [26]. In this metaphor, the monolith is represented by the host tree, whilst the microservice application is the fig vine. In very much the same way, strategically positioned business functions are carved out into their own microservice [22], and in the end the monolith either disappears completely or becomes a microservice. The strangler pattern represents an incremental refactoring pattern. This means that the application (as a whole), keeps being maintained and tested, as well as getting new features, while it transitions to the new architecture. Although the strangler pattern predicts new features being developed, in order to stop the core monolith from growing any further, the newly implemented features should not be developed within that monolith's scope, but within its own bounded scope. Ultimately, the strangler pattern approach is considered to be the most common migration strategy [27]. This happens because, often, enterprises seeking a microservice implementation, already possess a fully functioning and running application in production servers. As such, these enterprises cannot afford the shut down their systems indefinitely until they reemerge with an optimally functioning microservice architecture.

One other approach that can be used to migrate from a monolith to a microservice, in a far less gradual manner is the complete replacement of the monolith for a newly implemented microservice architecture. This approach, forces the initial monolith to be built in a sacrificial architecture fashion [28], meaning that an architecture is built with the projection that it will be replaced in the future. The total replacement of the monolith might be the ideal solution for small applications that do not change much with time and find themselves in a stable lifecycle. These types of applications give developers the time they need to fully focus their resources in creating a new architecture behind the scenes, while the original monolith keeps working undisturbed. In the case of an implementation from scratch, the monolith-first with a subsequent replacement, can benefit from the initial monolith to be used as a probe that is deployed to production in an early stage of development [23], allowing the product to enter the market before all the microservices are implemented. Still, the formerly mentioned approach has some caveats to it, one of them being that a rapid and sudden transition might not give the opportunity for developers to take notice of certain flaws in the application, therefore these shortcomings will only be revealed in the production environment. Another possible problem faced by this replacement is due to the complexity of microservices and their integration [22]. Such complexity should be tacked with the knowledge of which are the modules that should be migrated to a microservice and what should be incorporated with an existing microservice. The balance to find the appropriate implementation is much easier when done incrementally together with meticulous tuning, rather than all at once in a Big Bang-like fashion [29].

Yet another approach worth mentioning is one considered to be the middle ground between the two priors. This strategy allows developers to start off with just two major services, rather larger than those intended for a microservice architecture. With the two distributed monoliths, the teams have the space to understand and improve the integration between a multiservice system, as well as consolidating the boundaries of the application. This knowledge and overall better understanding of the applications boundaries will enhance the decision making when the time comes of starting to carve out the different functionalities into their own services.

The conversion of a monolith to microservices is not a consensual topic, despite the previous mentions some software thinkers believe that the best solution is to start from a microservice architecture right off the bat. Claiming that the creation of a monolith, intended to separate business requirements into perfectly moduled containers, will most likely fail at doing so, creating tight coupling between the different components. This tight coupling reflects itself in high entropy once it is time to migrate the architecture [30]. Another claim supporting the microservice-first approach is the fact that it allows teams to get used to a distributed system right from the start of development, and even if the boundaries of the microservices are not correctly matched right away then the cost of repairing is lower than the implementation of an intermediary monolith.

As it is common in the world of software, there are no silver bullets [31], regarding the way a task should be performed. Even though the market offers a vast array of solutions regarding the transition to a microservice approach, it is important for companies to deeply analyze the task at hand and how they can better benefit from the upsides of each solution against their drawbacks. A deeper analysis will allow for a more successful migration or start of an application.

2.5. Microservice Integration Patterns

When analysing a microservice architecture it is natural to wonder - “How do isolated applications that can be deployed independently, scaled independently, and tested independently [16], communicate and integrate seamlessly?”. To properly answer this question, it is perhaps better to analyse some of the microservice architecture’s limitations and try to comprehend why a conventional flow of data is not a suitable solution for the architecture at hand, and subsequently discuss some of the essential integration patterns commonly used in the field.

2.5.1. Challenges

To start analysing and comprehending the underlying problem in the integration of microservices and their respective solutions, let us start by properly evaluating and exposing the posed challenges and how they defy and limit the current architecture.

On a monolithic architecture, the different services invoke one another via language-level method or function calls [32]. Being that by calling each other through a coupled instantiated object or a looser way like dependency injection, in either case the caller ends up summoning the callee to which it has access to, because they are running within the same process. Although, when moving to a microservice architecture these procedures become impossible, given that services are now running in different processes, different clusters, and ultimately different addresses. In order to properly communicate microservices need to resort to interact using an inter-process communication (IPC) mechanisms [33]. An IPC is characterized by an application interaction that categorize as clients and/or servers. A client being an application or a process that requests a service from some other application or process. A server being an application or a process that responds to a client request. Commonly applications act as both a client and a server, depending on the situation [34]. Still, this does not fully satisfy the question of how microservices intercommunicate, since there are numerous IPC mechanisms that fulfil different necessities, microservices should be able to communicate on a one-on-one basis or a one-to-many basis, as well as establishing synchronous and asynchronous connections, taking in consideration the task at hand. Another challenge posed by a microservice architecture, regarding integration, is data independence in a loosely coupled environment. As previously mentioned, the data owned by a microservice is restricted to that microservice, and can only be accessed through its API, this means that microservices with poor bounded context definition will suffer from lacking data and furtherly require chatty [32] interactions with other microservices.

When discussing data management, it is also important to address the challenge of how the Consistency, Availability and Partition Tolerance (CAP) Theorem applies to a microservice architecture and what were the choices made in order to achieve higher performance and faster response times.

Overviewing the main and most common challenges that a microservice architecture faces, allows developers to design and implement better solutions that fit each problem, in order to mitigate the headaches of an otherwise convoluted approach.

2.5.2. The CAP Theorem

The CAP theorem states that it is impossible to design a distributed data management platform that provides always consistent (C), data accessed through always available (A) and operations with the possibility of a subset of nodes being partitioned (P). In this sense, a node partition is comprehended by a segregation, in which there may be modules which are unable to communicate with each other [35]. The CAP theorem describes a trilemma or an impossible trinity, firstly introduced by Armando Fox and Eric Brewer in 1999 [36], which it asserts that a distributed data management platform can only provide at most two of the three variables of the equation.

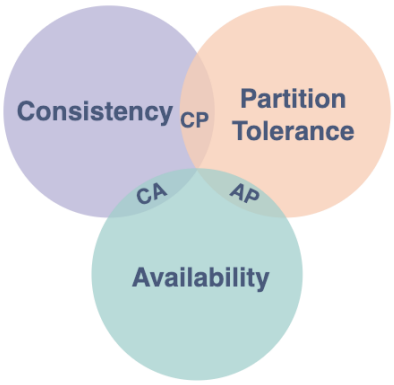

Figure 4 - Visualization of the CAP Theorem

- **Consistency** and **Partition Tolerance** without **Availability** (CP) – A system that requires relentless consistency and allows the different nodes of data to be partitioned along different networks cannot provide availability, given the fact that a transaction that is made across networks needs a downtime to reconcile all its data in order to be consistent.

- **Consistency** and **Availability** without **Partition Tolerance** (CA) – A system that obliges continuous consistency and availability can only do so in the absence of network partitioning and separation of server peers [36].

- **Availability** and **Partition Tolerance** without **Consistency** (AP) – In order for a data management system to provide constant availability and allow its nodes to be distributed among different network services is through the trading of consistency. If a system requests a data update across a network it is likely that a request for the data arises before the data

update is applied, and in a situation of maximum availability the client shouldn't await for the data to be consistent, the data will be provided before the consistency is achieved.

Extrapolating the CAP theorem to microservices and their architectures it perfectly fits the definition of this trilemma by falling under the category of a distributed architecture that manages data, in which each service is responsible for a subset of data and transactions are preformed when microservices interact. In microservices, just like any other distributed system approach, architectural trade-offs must be made when designing microservices, to address the limitations imposed by the CAP Theorem [37]. Since microservices are units of software independently scalable and deployable, that follow a loose coupling approach. It is hard – if not impossible – to imagine a microservice architecture that is not segmented across multiple networks. As such it unfeasible to sacrifice Partition Tolerance (P), a process without P can't run across multiple networks, and so it becomes a process running locally in a single host. Constantly consistent and available (CA) systems do not exist in distributed systems [15]. This leads to question which one is the right approach between CP and AP, and in truth there is no answer for that question, it all comes down to what the situation at hand benefits the most from. In some cases, it might be more beneficial to provide data in an eventually-consistent form than no data at all, although in other scenarios where the information is more sensitive it might only be beneficial to display the data if it is accurate. In each case, a peculiar characteristic of microservice is the ability to allow the slider to be moved in one direction or the other on a service-by-service or even request-by-request basis [38], assuring that in no service or the architecture itself is constrained by a decision that better fits the bulkier part of its providers or clients.

2.5.3. ACID Transactions

As previously discussed, the decoupled nature of a microservice architecture allows each service to choose its values consistency over availability or otherwise. While in reality, the granularity of this decision can be scaled to a business functionality level, in which each service is able to choose availability over consistency based on the business necessities or the criticality of the data being handled [39]. To further understand how to better benefit from availability or consistency it is important to analyse how both strategies prefer to handle their transactions. A transaction is a group of operations that intend to carry data from one point to another, for a long time transactions were seen as operations that needed to have the properties of atomicity, consistency, isolation and durability (ACID) [40].

In an ACID transaction, either all operations are completed successfully, or none is, this property is known as **atomicity (A)** [41]. Traditional transaction systems use a two-phase protocol to achieve atomicity between participants [42]. With two-phase commitment protocols, services preform a temporary transaction operation to each of the participants and wait for a successful return, if this success occurs then the transaction is considered successful and the changes are committed, as depicted in Figure 5. Otherwise, if in any of the nodes where the transaction was preformed does not allow that transaction (abortion) then all the operations in the other nodes are reverted to their previous state (rollback), leaving the nodes in the state that they found themselves in, before the operation was started [42]. This process requires the transaction to be blocked while waiting for the response from all of its components. Falling perfectly in the category CP from the CAP Theorem, as all nodes in the transaction – partitioned as they are – end up consistent at the end of each transaction at the expense of complete availability.

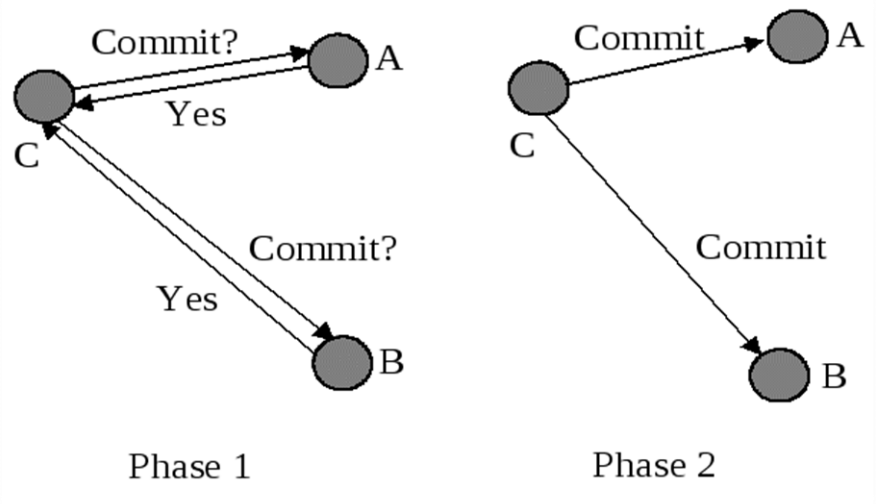

Figure 5 - Diagram of a two-phase protocol

Consistency (C) is another of the properties presented in ACID Transactions, and it states that at the end of every transaction, all of the participant nodes should hold consistent information and the integrity of the data should be assured, in short the nodes should keep semantic invariance [35]. For instance, in a scenario of an ecommerce platform, very much like the example displayed in this document, if the service that holds the stock has one single item of the product A, and two costumers are trying to acquire that item at sensibly the same time, the item should only be available to one of them. This process can only be completed through a temporary unavailability of the service, in order to recalculate the number of items currently obtainable. In this sense, it also provides basic consistency at the cost of availability, when analysed against the limitations offered by the CAP Theorem.

Transactions between microservices in an ACID environment, should be **isolated (I)** and which means that at any point the data passed along in a transaction should be only known within its scope, this increased the consistency of the operations by preventing data to be shared in an unsafe way. A violation of the isolation property is a consequence of semantic atomicity, since the partial effects of sub-transactions that unilaterally commit are then exposed to other transactions [43]. This means that transactions that are concurrent are not executed concurrently. In order to keep segregation, each node of a transaction must time its operation using a blocker protocol to inhibit overrides of an intermediate process, thus promoting consistency.

Durability (D) is the last property of ACID, and it states that any change performed by a transaction should be persistent, even in the event of a system collapse. This process is usually assured during the atomic two-phase commit in which if a transaction node is not able to durably persist the incoming data, then all of the transaction is rolled-back.

When comparing the properties of ACID transactions, described above, against the limitations provided and proven by the CAP Theorem it is evident that ACID transactions promote consistency over availability. Either by implementing the two-phase commitment protocol or by timing concurrent requests in order to block changes before each transaction process is totally finished. All the implementations aimed at reinforcing the ACIDity of a system, also result in the reduction of the availability of resources, since they promote an environment where the services are blocked in processes to ensure consistency and cannot provide their held requested data. The implementation of ACIDity in microservices, despite being critical in some cases, fosters the high coupling of components turning the architecture of the application more brittle, in the sense that if at any point one service is unavailable during a transaction then the whole process might fall apart.

2.5.4. BASE Transactions

Despite the popularity of ACID for many years in a world of data management, with the emergence of distributed systems the paradigm shifted giving place to an alternative of eventual consistency. While ACID is based on pessimistic assumptions and forces consistency at the end of every operation, BASE is based on optimistic assumptions and accepts that the database consistency will be in a state of flux to a level acceptable to each business transaction [41]. The base strategy stands for Basically Available, Soft state, Eventually consistent, and it fits an event-based approach to distributed data management systems. The BASE protocol does not guarantee that any of the data draws will be updated

with the latest information, and it represents a form of weak consistency, in which it states that if no updates are made to the accessed object then all the accesses to it will eventually be consistent [44]. The timeframe from when the data becomes inconsistent until it eventually updates and becomes consistent again is known as the inconsistency window. And for situations without failures, the maximum size of the inconsistency window can be bounded based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme [45]. BASE-like transactions provide an alternative to ACID by endorsing high availability at all times, disregarding the state in which the data is at the time of the request. A soft state (S) of data is common in this approach since it is never clear when the data is at a state in which it is considered updated. When contrasted with Brewer's CAP Theorem, BASE-like approaches tend to relax consistency in order to achieve high availability, placing it in the classification of AP. In a microservice architecture, fostering availability over consistency tends to be the best option [46], given the architectures scalability potential as the number of instances of each microservice increases, so does the amount of transactions they perform and so would the unavailability of data in an ACID-like environment. Another advantage of using BASE-like approaches for microservice architectures is the fact that eventual consistency promotes the decoupling of components by resorting to event-like communication protocols (tackled further ahead in this document).

2.5.5. Orchestration and Choreography

A microservice, as an independent unit of software, can decide how it integrates with other services based on the business requirements and the sensitivity of the data being passed along. After understanding the ACID – BASE relationships between transactions it is important to analyse how microservices interchange data with respect to each of these principles. A microservice cluster can intercommunicate either through an orchestration strategy or a choreographed strategy.

Orchestration - Orchestration refers to a centralized business process that coordinates a series of service invocations [47]. Very much like an actual orchestra is reliant on its maestro to dictate the rhythm of the composition, a microservice architecture based on an orchestration approach is dependent on a microservice controller that directs each service to perform the intended function [48]. This tactic is reliant on a synchronicity principle in which every operation only happens after the success of the previous one, and it promotes a centralized service that enforces tight coupling of components, because the orchestrator needs to know the entire "orchestra" in order to command them to execute. Although this interaction is not

recommended in most scenarios, regarding microservices, just like in ACID Transactions, sometimes the consistency required in some operations force integrations to be synchronous in order to get the most updated information possible from each service.

Choreography – On a choreographed approach each microservice has a grasp on the business logic and is expected to know what to do at each step of the microservice interaction. “In a choreographed dance team, everyone knows what they’re supposed to be doing, and is able and required to take the right step as each beat hits” [48]. The choreography between microservices happens in an asynchronous manner, resorting to lightweight messaging protocols and/or an event broker. Every microservice involved preforms an operation and fires an event for the other microservices to take over, at that moment the publisher becomes unaware of the following steps, trusting its peers to carry on with the operation. This integration approach encourages the loose coupling between the services, because at each moment the service publishing the event does not need to know the physical address of its consumers. The publisher simply needs to know the address of the broker to which it publishes the event unto, and then each of its consumers will then resort to that broker to hydrate their data. Another property enhanced by the loose coupling of choreography is the fact that microservices still work fine in case of a failure from any of its peers.

An architecture that is choreography-based lays on BASE-like transactional principles, by relying on eventual consistency to manage their databases and benefiting from the high availability inherent to the loose coupling of the different parts.

Although some cases require the transactions of data to be pin-point accurate, and cannot risk inconsistencies in the system, for this scenarios orchestration might be a better option. Despite this, when taking a look at a microservice architecture and comparing their requirements and necessities with the overtures provided by choreography and/or orchestration, it is rather easy to realize that in most scenarios the choreography strategy seems more beneficial. As microservices tend to favour a style of smart endpoints and dumb pipes. Applications built from microservices aim to be as decoupled and as cohesive as possible – they own their own domain logic – receiving a request, applying logic as appropriate and producing a response [12].

2.5.6. Communication Strategies

According to Chris Richardson’s “Building Microservices: Inter-Process Communication in a Microservices Architecture“ [49], microservice interaction protocols categorize themselves along two dimensions, displayed on Table 1. The first dimension is whether the interaction

happens on a one-to-one or on a one-to-many basis. The second dimension is whether the interaction happens synchronously or asynchronously. Further along in this sub-chapter, the different communications based on their synchronicity type and how they adapt to the principles previously referenced.

	One-to-One	**One-to-Many**
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async response

Table 1- Microservice interaction strategies based on synchronicity and amount of services

2.4.6.1. Synchronous Communication

As previously mentioned, in some cases microservices require a communication that is synchronous in order to achieve higher consistency in the data being passed along. Utilizing synchronous communication is an easy and effective way to get two microservices interacting on a one-to-one fashion. An asynchronous call refers to a process in which the client – being that the microservice querying – invokes the server – the application being queried – and awaits the response from the server to arrive or to eventually time out. This means that the process of the client is blocked, while the request is traveling to the server, while the server processes the request and turns it into a response and while the response is making its way to the client. Any time there is a number of synchronous calls between services, the multiplicative effect of downtime will be present. Simply put, this is when the downtime of a system becomes the product of the downtimes of the individual components [9].

The most common protocol for synchronous interactions of microservices is the Hypertext Transfer Protocol (HTTP), this happens due to the simplicity and familiarity of HTTP, combined with its comfort to test and the fact that it does not require and intermediary broker that turns the architecture ever more complex [49]. Despite the benefits presented by a simple HTTP integration, there are also some downsizes to this approach, some being linked to the requirement that both services need to be running at the same time, in order for an interaction to be considered successful. Another bump in the road is linked to the requirement that for any interaction an HTTP request needs to be performed while knowing the address of the interface it is trying to query. This poses a problem since, as discussed in the previous sub-chapter Microservice Architecture, the addresses are dynamic due to load balance strategies applied over cloud-base infrastructures. This obliges the client application to resort to a service

discovery strategy in order to submit its request, obscuring the simplicity of the microservice architecture ever so slightly.

In reality, synchronous microservice integrations can resort to other strategies such as Simple Object Access Protocol (SOAP) or Thrift, in the end both approaches pose very similar pros and cons to those of HTTP, and the final decision comes down to preference of the development team or a clear benefit on a case to case basis.

2.4.6.2. Asynchronous Communication

For the integration of microservices using asynchronous strategies, it is possible to find a vaster variety of solutions. An asynchronous flow of data is a flow where the process is not blocked while waiting for the response to return, in most scenarios it is not even relevant that a response should arrive. The most common form of asynchronous communications are lightweight message-based protocols, these protocols allow for application resilience, failure tolerance and better scalability. Since, if not for the use of messaging, applications would need to be available one hundred percent of the time without room for downtime or failure [50].

Analysing the Table 1, published by Chris Richardson [49], it is easy to realize that microservices communicate asynchronously using three main strategies. Either by a simple notification process, where the client sends a message to the server and awaits no response. Either by request/asynchronous response which is a process where the client sends a message to the server, and in a non-blocking way follows its process knowing that a response will eventually arrive in the form of a message. Or, by the most popular type of asynchronous integration approach the publish/subscribe – displayed in Figure 6, in this scenario a message is sent by the producer/publisher to a queue/message broker, that acts as an intermediary between services by storing messages, this messages can then be consumed by zero or more services (subscribers/consumers). Typically, message brokers divide themselves among channels, making the separation of responsibilities easier to handle and allowing microservices to use the same broker to share different kinds of data. Message brokers can be found in all the strategies mentioned above. A publisher does not need to know about the existence of subscribers, it just needs to know the repository to which it is publishing the messages to, as such there is no need to refer to service discovering mechanisms. Just requiring the address of the message broker allows services to scale freely as they can create more instances of themselves to publish or consume messages from a broker according to the traffic necessities at any given point.

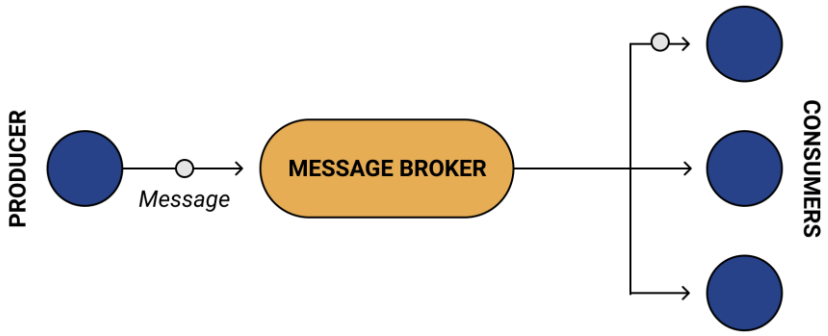

Figure 6 - Message broker

Despite the previous there are still some considerations that need to be considered when implementing a messaging-based solution. Even though the system impact is lower in case of availability, the queue/message broker is another moving part in the architecture and as such it needs to be maintained and configured in a way to assure the highest rate of availability possible. Another concern to have in mind, is the complexity of developing publish/asynchronous response strategies, since the client needs to know which request it should link the response to. In these scenarios it is frequent to implement unique identifiers for requests to which then a response should be generated using a matching unique identifier. Although this strategy does not pose a big threat to the system it simply represents a higher level of complexity that can lead to a breaking point in the application when left untamed.

Transactions based on asynchronous communications, are typically BASE-like transactions. They benefit from basic availability, given that the downtime of any microservice does not affect the performance of any of its peers because they benefit from non-blocking processes. Not having transactions across services is not bad per se. The big risk here is that it requires a change in the way function requirements are designed [51] and a rethinking of what the interruption of a sub-process at a certain step represents to the process as a whole. Another property of transactions that are based on asynchronous communications is the soft state of the data involved, the data is always at a state in which is potentially up-to-date or potentially outdated. The final property that makes the connection between BASE and the microservice asynchronous strategies is the fact that at any point, if there are no more updates to a certain node of data, after a certain amount of time it will eventually be updated. These updates happen when consumers process the messages from the queue and renew the data in their databases.

2.5.7. Error Handling

Just like about everything else in the world, software breaks and microservices are no exception to the rule. Although there are some things that can be done to prevent that from happening, developers have accepted this scenario as another expected variable in the equation of software development. By doing so, developers recognized the inevitable risks of errors and implemented strategies that enhance the application's resilience and failure tolerance as a whole. By nature, the architecture of a microservice application is – for the most part – failure tolerant. Microservice architectures excel by performing communication through a mainly BASE-like pattern, that in turn revolve around a message-based type of communication. In this architecture, if a message falls in a broker and the consumer is down or unavailable to read it, the message will simply be persisted in the broker until it is read and processed, not stopping the application as a whole. On the other side, if a producer is unavailable and cannot produce a message for its consumers then they will keep working in the same manner, unaffected. In this regard – also considering the broker as another moving piece in the architecture – if the message broker is unavailable or unreachable, the producer should have mechanisms that allow it to retry the publishing of the message until there is a successful publish, keeping the message persisted while it is still unpublished [41]. Although there is a caveat to the last presented scenario, in the moment when the message broker fails all the messages that is carried are lost, as such microservices need to develop mechanisms that allow them to republish messages in case of a communication rupture.

In the examples mentioned previously, the architecture itself and the way services are distributed shield the different microservices from a cascading failure, that ultimately would result in an overall unavailability of the entire system. But unfortunately, not every microservice communication benefit from this protection. Some microservices that communicate via synchronous strategies need to implement some further approaches that allow them to keep functioning properly in case of failure. Currently, developers use design patterns like circuit breaker and retry with exponential back off to minimize the negative impact of failures on their application [52].

- **Circuit Breaker –** When a service tries to reach for another service, sometimes the latter might be unavailable, and recurring requests might reach a point where it overwhelms the system to a point of further failure. To prevent this event from occurring developers should implement systems called circuit breakers. These circuit breakers, displayed in Figure 7, act as a wrapper around the error response received from the unavailable system. When a threshold of number of failed requests has been reached, between per se microservice A and microservice B, service A stops calling the service B and instead immediately responds with the error wrapped by the circuit breaker, without resorting to another request [53]. After a

particular amount of time, the microservice A requests a reset of the circuit breaker, that in turn will try to request the microservice B in its next request, if it fails again and the threshold is reached yet again, then the circuit breaker gets a reset and the process repeats itself until the service B is available. This strategy allows the application to fail fast and restrict an exceeding number of connections, which sequentially allows the application to save its resources.

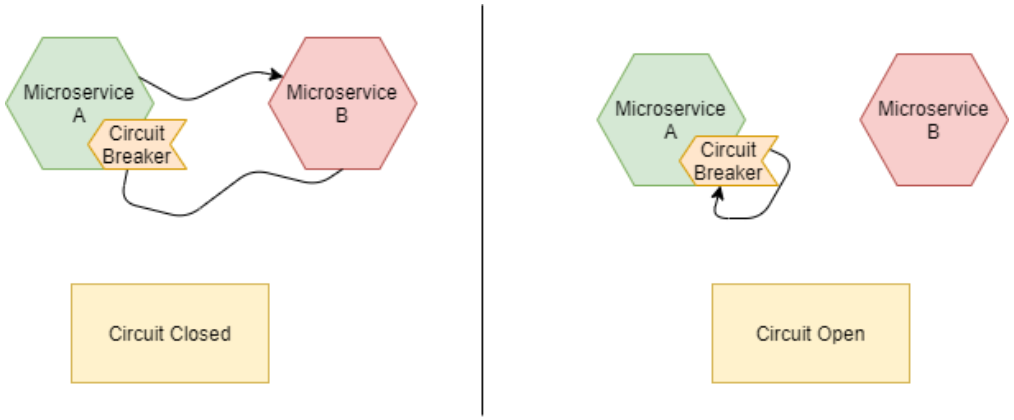

Figure 7 - Diagram of a circuit breaker as a middleware between two microservices

- **Retry with exponential back off –** In some scenarios, services simply fail due to a temporary unavailability created by a higher traffic period, specially under the lack of load balance solutions [54]. In cases like these, the availability of the service should be resumed when the traffic volume of the service diminishes, and a simple retrial of the request will solve the problem. Despite this being true in some cases, it is not absolute and for those scenarios the “exponential back off” component needs to be applied. A service cannot expect to request another service in a persistent manner and expect different results, at the risk of crashing the system. As such, the requests will be spaced in time in an exponential increase between requests until a request is successful.

- **Time Out Pattern –** During times of increased traffic volume services might take longer to process their requests, in which clients must await their responses. Although when that happens, your services cannot just wait forever for a response that might never come – sooner or later, it needs to give up. “Hope” is not a design method. [55] In this regard a very common (and recommended) solution is the implementation of time outs, as they allow the connection to be disrupted, once a set amount of time has expired. This mechanism allows applications to save resources by closing ongoing connections that might have a high chance of having an unwanted result.

- **Bulkheads –** In a ship, bulkheads are metal partitions that can be sealed to divide the ship into separate, watertight compartments. Once hatches are closed, the bulkhead prevents water from moving from one section to another. In this way, a single penetration of the hull does not irrevocably sink the ship [55]. In the same manner, in a microservice architecture, bulkheads are patterns that require a reevaluation or a preemptive evaluation of the failure scenarios in the application in order to restructure the applications to contain damage in a non-spreading way. A common way to implement bulkhead solutions is the usage of dedicated resources to handle different partitions of a service that might be business-critical [53].

- **Cache Fallback Mechanisms –** Following the BASE concept of eventual consistency, some interactions do not suffer from resulting in eventually consistent transactions. As such, some frequent request/response interactions can be cached, in order to be used as templates in moments where the requested service is unresponsive. Although the result of the request might not correspond to the result of an eventual response from the server, in some cases of unavailability it might be better than an error.

The above mentions are only some of the main solutions utilized by developers to enhance the resilience of their microservice architecture, and although they do not solve every problem of such an approach they certainly help to contain the spread of errors through the entire application. Each solution is not unique and can be implemented in cooperation with the others.

2.5.8. Tackling Evolving Contracts

In the world of business-level software development, requirements arise with relatively high frequency, which means that inevitably APIs are bound to change with the same frequency. Whilst in a monolith these API upgrades are usually straight forward to implement and impose, when talking about microservices these changes can pose a challenge that threatens the stability and reliability of the entire application. On a microservice architecture, client integrations need to work regardless of any upgrades to the API, and clients cannot be expected to change their implementations over-night to incorporate the ever-evolving changes of the servers [49]. To do this, services should implement a robustness principle – which states that an API should be conservative in what it implements, and liberal in what it accepts from others [56]. By this metric, an API needs to evolve in a versioned and partitioned way. Some changes might be retro compatible, like the addition of a new field in which case it can simply be ignored.

Nevertheless, some changes are labelled as major and cannot provide retro compatibility, in such cases new network connection points need to be provided. In the case of message-based communications a new channel might have to be created, in order to exchange events that are compliant with the new contract. When referring to http-based communications the most common solution is to embed the version of the endpoint in its Uniform Resource Locator (URL), allowing old clients to maintain the previous contract, while allowing – at any point – the migration to the newer and upgraded contract.

2.6. Scale Cube on Microservices

The scale cube was introduced by Martin Abbott and Michael Fisher, in the book “The Art of Scalability” [57]. It describes a model of a cube represented along three axes, these being X, Y and Z – depicted in Figure 8.

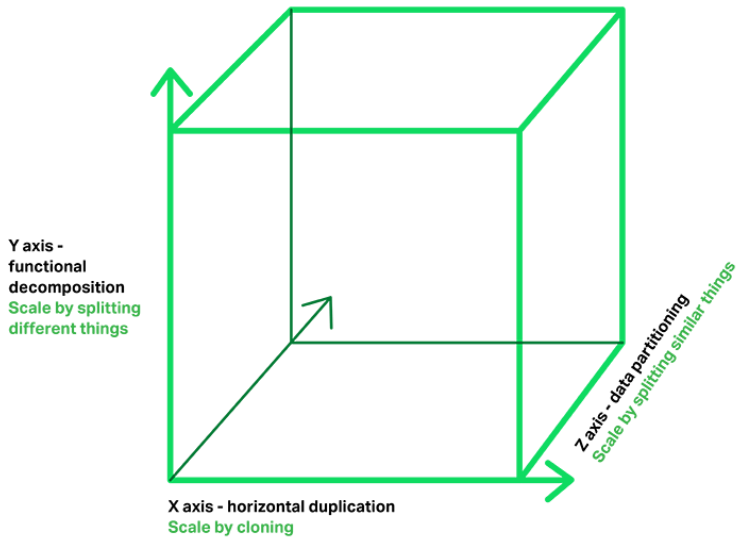

Figure 8 - The Scale Cube

The **X-axis** represents scaling by redundancy [58], this happens when an application is scaled by creating multiple identic instances of itself, and positioning them behind a load balancer in order to distribute the traffic load among them. Scaling with resort to the X-axis is the most common approach for monolithic applications, and it does not solve the deployment problems attached to monoliths that were mentioned before.

The **Y-axis** depicts scaling by functional decomposition. In this sense, functional decomposition means the separation of concerns within an application into new services. Each service is responsible for one or more closely correlated tasks.

The **Z-axis** describes the scaling through data partitioning [59]. In its essence the Z-axis scaling does not diverge much from the X-axis scaling, as it also consists of deploying multiple instances of the same codebase behind a load balancer. However, it distinguishes itself by the manner in which it balances the traffic arriving to each node. In this strategy each service is purely responsible by a subset of all the data that the application contains.

After a closer look to the scale cube, an argument can be made that microservices can fully take advantage of all the dimensions of the scale cube. Contrary to monolithic applications, that failed to do so, due to the lack of the verticality provided in functional decomposition. Microservices, can take advantage of the width of the scale cube (X-axis) by creating multiple instances of each service, depending on the actual or predicted traffic at each point in time. Microservices also seem to take advantage of the height of the cube (Y-axis), as when a service simply gets too large, it can be broken down into a smaller level of granularity and so reducing the overall load of the service. The last aspect of the cube – depth (Z-axis) – also appears to be advantageous to a microservice architecture, as services that are deemed indivisible can benefit from data partitioning in order to better regulate its request load management, instead of an identical replication in which data accesses would still be congested.

2.7. Testing

Testing software is one of the single best ways to reduce the likelihood of undesired errors in a business-critical scenario. This being said, functional testing of distributed systems presents one of the greatest challenges to any test automation tool [60]. To achieve high levels of confidence, enterprises' Test-Driven Design (TDD) needs to be refined and broken down into the different test strategies, so that teams can test their applications individually as well as its integration with the surrounding environment. In the sense of breaking down the different test phases and scenarios, this dissertation will invoke the Mike Cohn's approach to agile testing (depicted Figure 9) [61].

In the Test Automation Pyramid there are three main layers:

- **Unit Tests** – They are positioned in the bottom of the pyramid, and they represent the smallest units of tests. Unit tests have the purpose to test each unit of code, in an isolated manner. Despite being around for a long time, it is still not clear what qualifies as a unit, some might consider a class to be a unit whereas others might reach for individual methods as their units. In either of these cases the scope of the unit tests does not have a strong impact in the test automation process as a whole. Due to their reduced scope, these tests embody a rather

low cost to implement and if correctly executed they can be performed rapidly. The strong isolation property of unit tests allows them to provide meticulous feedback, as when any test or suit of tests fail it is rather apparent where the error is.

- **Service Tests** – Service tests are designed to bypass the user interface and test services directly [15]. These tests intend to test the application's API by the functions it is designed to provide. In this type of tests, it is common to query the API expecting certain results and comparing the response of the API with the expected outcomes. The granularity and isolation of service tests are lesser than those of the unit tests and as such these usually take more time, time that can be dependent on the response time of database connections and other remote dependencies. With the decrease in granularity also comes the decrease in feedback capacity, as a service test can cover a larger scope than unit tests, it becomes harder to determine the breaking point of the application.

- **UI Tests** – User Interface (UI) Tests, also known as End-to-End Tests, cover the largest scope of all tests. These tests tend to be performed as a simulating of the actions executed by the user in a real case scenario. UI Tests have a very small granularity level which reflects itself in very low feedback upon failure. It is easy to see that the integration fails, but it is hard to pinpoint the location of the error.

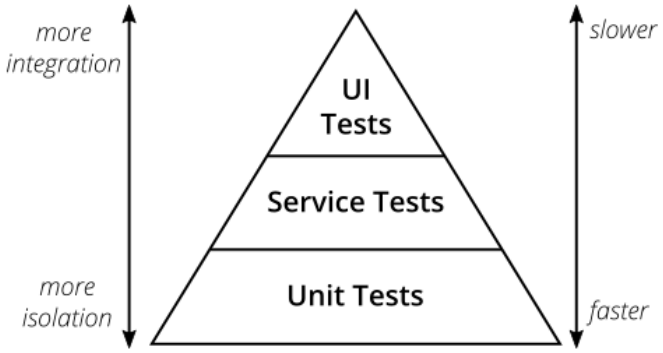

Figure 9 - Test Automation Pyramid by Mike Cohn

The diagram of the Test Automation Pyramid allows for a good analysis of the test suit of regular applications. The fast and rather cheap nature of unit tests allow them to compose the bigger majority of the test suite. Followed by the Service Tests that are a slightly slower and more costly than the former, these should represent the second bulkiest set of tests. And lastly, the UI Tests that represent the heaviest overheads should represent the smallest

fraction. When looking at the Test Automation Pyramid it is also important to realize that the confidence level has an inverse proportion to the isolation of the test [61]. The bigger the isolation, the smaller is the confidence and contrariwise. As such, when a unit test is successfully completed that does not provide any confidence that the system as a whole works, in the other hand, when an UI Test fails it is pretty much clear that the integration between components is broken.

2.7.1. Challenges of Distributed Testing

Despite providing a reliable diagram for single applications, the Test Automation Pyramid comes short when contrasted with a distribution system approach. For instance, significant execution problems exist, such as: system latency for individual test cases, global state reset, non-determinism leading to unrepeatable errors, and the existence of faults in communication infrastructure, to name but a few [60]. The integration between the different microservices needs to be tested in order to have a reliable system, and yet the expensive UI Tests seem to be the only available mean to perform the task. Not quite, as it will be clarified further.

In a microservice architecture, the individual microservices need have to the ability to be independently deployed, and as such also independently tested. If a company is expected to release a new version for all of its services, due to a modification in a single one of them, then one of the most basic principles of a microservice architecture is violated, which results in tangled operations. Consumer-Driven Contract (CDC) tests are a form of service-level testing that allow to test the integration between application with the mean of a contract [62]. In this sense a contract represents an agreement between a consumer (the receiving application) and the provider (the delivering application), on what is going to be the payload shared between their interactions. In short, the contract contains information about how the consumer calls the provider and what is being used from the responses [63]. If the contract is respected, then the integration between both ends can be assured. These tests are rather cheap to develop and execute, as they simply consist of querying a provider's API - in this particular scenario from a consumer's perspective - and comparing the response with the actual necessities of the application. Typically, the team working on the consumer application is responsible for developing these tests and exposing them to the provider. In this manner the team working on the provider application can execute the tests for all its consumers, upon changing an implementation in their application to make sure that the integration between services is still working properly [64].

In a way, the implementation of Consumer-Driven Contract Testing tools can allow applications to enhance their service-level test suites in other regards. The consumer can use

it to mock the provider in its tests. The provider, on the other hand, can use it to replay the consumer requests against its API [63]. They might also serve as user stories guidelines for the provider, in order to enhance their contracts in accordance with the demands of its consumers. Moreover, CDC Testing can be used to anticipate scheduled changes in the course of service evolution against current expectations and obligations [65].

So, given their smaller scoped nature and apparently cheaper implementation, should Consumer-Driven Contract Tests replace UI Tests? Some experts say that with time companies lose the need for End-to-End Testing due to the reliability of CDC Testing together with strong monitoring [15]. While others insist that the simple restriction of these tests to the barely essential flows provide a good enough coverage to assure the confidence levels of the application [64].

3. State of the Art

The ensuing chapter presents and examines some of the solutions and strategies implemented by the state-of-the-art enterprises in the world of technology, regarding microservices. By doing so, it will be possible to see the evolution of microservices and how they can be used in high-demanding scenarios where millions of transactions are exchanged in the production servers. This chapter will cover the implementations of Netflix, Spotify, Uber and – for the purpose of the proposed solution – Jumia. The chapter intends to make a deeper analysis of some of the most beneficial outcomes of microservice in high-end companies.

3.1. Microservices

3.1.1. Netflix

In the current world of technology, it seems impossible to talk about microservices without mentioning the immense success of Netflix. Not only because Netflix is one of the pioneers in the development and improvement of MSA, but also because they are kind enough to leave the door open so that developers around the world can take a peek and learn more from what they are doing.

Netflix entered the market as a DVD rental business API [66], with an initial team of around 100 engineers working around a microservice architecture [67]. As of today, Netflix is comprised of many small teams working on the full stack of hundreds of small independently deployable microservices as it transitioned to the online streaming of digital content business. In order to proceed to this migration Netflix chose to implement a strangler pattern by migrating many of its individual services into Amazons Web Service (AWS) in 2009 [68] [16]. In doing so, Netflix started segregating its monolith into a distributed system, composed of small independent services, using cloud solutions before the term microservices was even in the table. In terms of microservice integration, Netflix is one of the pioneers in some of the strategies mentioned in the previous section, such as failure tolerance strategies. To handle failure, Netflix developed Hystrix [69] a mechanism that allows services to provide static content from other services in case of eventual unavailability. Another advantage of Hystrix is in the providence of a circuit break strategy that prevents call to recurrently failing API's until a retry time has passed. In Netflix's microservice architecture, the most important services were identified in order to create strong bulkhead strategies to allow the most basic features to be available at any times, even in case of an unpredictable error. Regarding testing Netflix

has decided in some bold choices, such as Fault Injection Testing [70] and the Chaos Monkey [71]. These strategies try to create flaws in the system in the live environment, to assess how the system reacts. In doing so, together with fine tuning of their processes, Netflix improved the robustness of their application, in the sense that a node being destroyed represents almost a non-event.

3.1.2. Spotify

Contrary to other mentions in this chapter, Spotify did not start its applications with a monolithic architecture. Rather, the company chose to implement a set of small independently deployable services from the start. The choice of this strategy allowed the company to better understand the benefits and limitations of the microservice architecture since their genesis. This understanding inherited by the experience of seeing microservice growing in a highly demanding environment, allowed the company to focus their efforts in creating strong teams. The team structure of Spotify represents a complex adaptation of the Conway's law [72].

In Spotify, developers distribute themselves among Squads, each squad is composed of a cross-functional that manages one or more services. When coordinating feature areas in between squads, they form tribes. To better fulfill their roles, the different layers of squads cooperate with each other forming chapters. Individual member of the company can share common interests by creating a guild, which despite being voluntary they allow different members of the enterprise to share experiences and knowledge about the different scopes of the application [73].

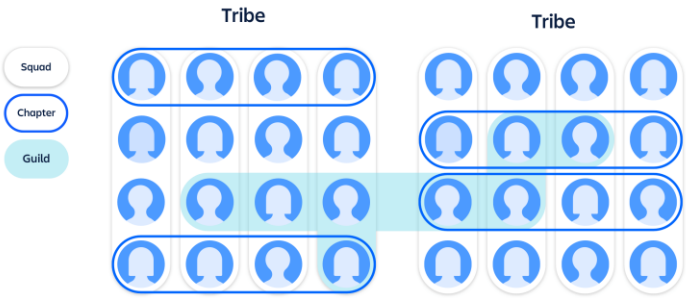

Figure 10 - Depiction of Spotify's team composition

The way Spotify organizes their teams allows them to have a better grip on the context boundary of each service. And by providing each squad and tribe with the exact definition of their scope they can guarantee that there are no overlapping services being built in the full span of the company.

3.1.3. Uber

Around the years of 2012-2013, Uber was composed of two large monolithic applications [74]. After careful analysis, Uber realized that some of the main problems they were routinely facing could be mitigated with the implementation of a service-oriented architecture, namely a microservice architecture. Resorting to microservices, Uber managed to reduce the rollout time of new feature by 25-50%, by substantially reducing the scope of each service and the overhead calls it would need to perform in the older version of the application [74]. Contrary to Netflix and Spotify, that chose to have different API Gateways for different devices, Uber chose to implement a single Gateway that serve as a single entry-point to its systems. With the implementation of an API Gateway Uber managed to benefit from discoverability and load balancing with the assistance of service registry mechanisms.

3.1.4. Jumia

Jumia started its life with the development of four distributed monoliths, they comprised of Bob – the application that managed the catalogue. The Delivery Manager – the application that managed driver and deliveries from sellers all the way until the costumer. There was also the Order Management System – arguably the biggest monolith of all, which managed all the operations inside hubs and warehouses. And finally, there was the Shop - the application responsible for the front-office that the user sees when accessing Jumia.

With time, and with the business of Jumia growing to an unpredictable scale, soon each of these applications needed to deploy new versions multiple times a week. This happened due to the fact that everyday different teams were developing around different modules of the mentioned monoliths. To solve the troubles of Jumia, many solutions were proposed, when the performance of the application was not good enough more instances of it were deployed (X axis in the scale cube). When the database performance was lacking, the solution was to shard it in different countries (Z axis in the scale cube). Still, these solutions were not ideal, as they were starting to represent a very high cost of infrastructure. Therefore, Jumia decided to start scaling in the Y axis of the scale cube, by chipping away functionalities into their own separate domains. And thus, microservices became the norm in the company. Since the beginning of its life, Jumia has managed to migrate all its architecture to microservices and definitively has seen some success. Internal numbers point to a speed up in average development time per story point, as teams moved from averaging 30 story points per week

to averaging 50 story points per week. Even Though, the analysis of these result can be extrapolated to different reasons, like better understanding of the system as a whole or a story point being an imprecise value, it clearly shows some improving.

In the matter of microservice integration, Jumia tries to handle their processes in the most BASE-like ways possible, implementing messaging wherever feasible in between inter-process communications, to prevent blocking APIs and create chained dependencies. The most common technologies used by Jumia to achieve this feat are Kafka and RabbitMQ.

4. Requirement Analyses and Solution Proposition

The task at hand focuses on Jumia's Back-Office environment, and it represents a necessity that emerged in the recent years after Jumia migrated its systems to a microservice architecture.

4.1. Problem Analysis

After implementing a Strangler Pattern [2] over its former application, Jumia ended up with a sharded set of applications for its back-office environment. This transformation allowed for a more structured and organized flow of operations. With each new service specializing in a different process. However, it is frequent that an agent is required to switch in between applications to access different information about either the order, the package or even the transport that is designated to take. The systematic switch between application can represent a slowdown in every process, especially given the fact that each access token has an expiration date of thirty minutes.

To better understand the problem at hand, this chapter will take a deeper look at an example of an agent accessing the Order Management System (OMS) in order to process a return of an item to the warehouse. For the item to be successfully inbounded in the warehouse it needs to go through a quality check inquiry that is process overt at the Warehouse Management Tool (WMT).

Figure 11, below, depicts the sequence diagram of the whole authentication process that occurs since the user starts accessing the application, until the login in considered successful and the user can continue with the workflow. Firstly, the user tries to access the application via the browser (1.1), after the client is started in the user's browser it immediately validates if it has a token and if so validates that the token is not expired (1.1.1). After checking that the session is not valid, OMS redirects the user to the Authentication Control List's (ACL) login page (1.2), with a query parameter that allows the ACL to know which application the user is trying to access. ACL then presents a login form (1.3) that a user needs to fill in order to gain access to OMS (2.1). After filling the form ACL validates the provided credentials as well as if the user has access to OMS (2.1.1), if so, ACL then generates an exchangeable code and encrypts it through an Rivest–Shamir–Adleman (RSA) encryption strategy, resorting to a public key previously generated by OMS (2.1.2). The web client of ACL proceeds to redirect the user again to OMS with the encrypted code in the URL under the query parameter "code". The exchangeable code has an expiration time of 30 seconds and can only be exchanged

once. The OMS client decrypts the provided code (2.3) and requests the API of ACL (3.1) to exchange the code for an authentication token (3.2). Before responding with the authentication token, the ACL API validates the exchangeable token (3.1.1) to infer if it has been redeemed or if it corresponds to the one that was generated. After having the authentication on its side, OMS stores it in order to inject the token in every request. By doing so, the API can identify the user in every request and confirm the access of the user to the resources it is trying to access.

After the authentication process the user is able to enter the OMS application and push the package to be subject to a quality check in WMT.

To access the WMT application the user has to go through all the authentication process again. Open the application (4.1), fill the form (5.1), wait for the exchangeable code generation, encryption and subsequent decryption (5.1.2, 5.2 and 5.3), wait for the exchange of the code for a token (6.1 through 6.2) so that the user can finally get a successful login in WMT (6.3) and access the application.

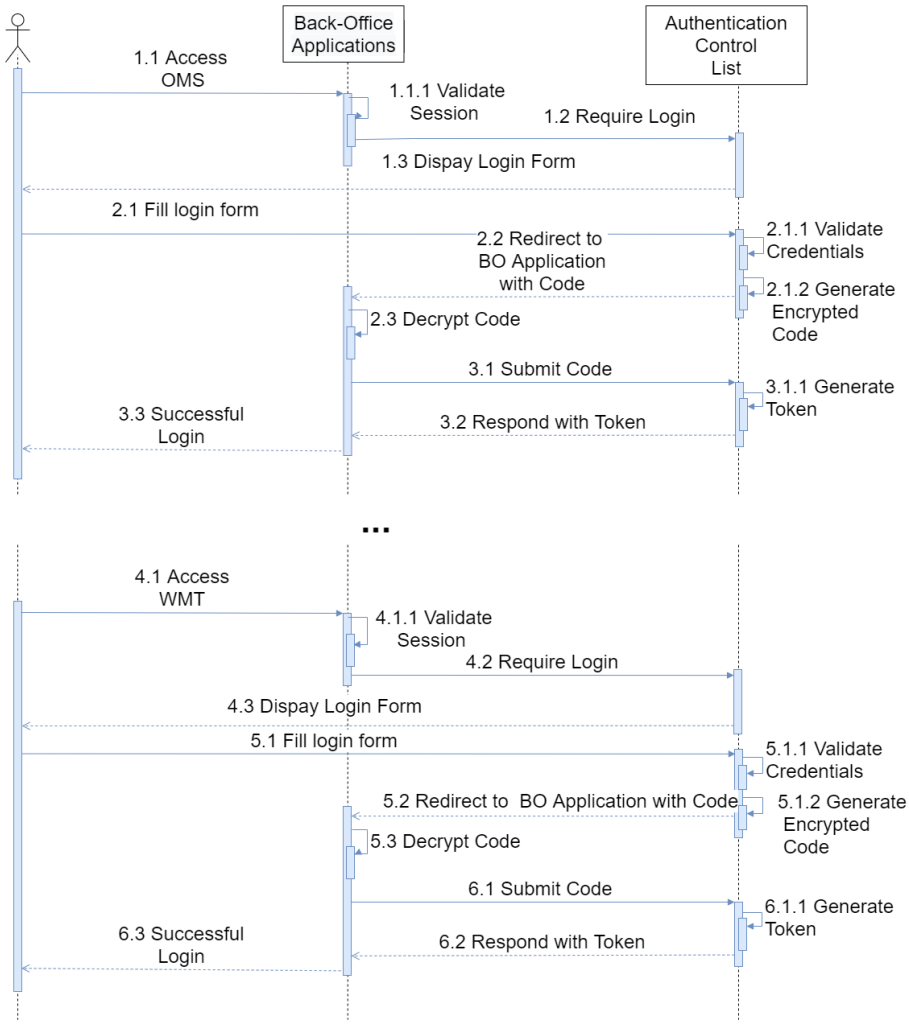

Figure 11 - Current Sequence Diagram for Application Switch

The process exposed above portrays the process a user needs to go through in order to switch between application A and application B. After taking a closer look at this procedure Jumia's engineers decided to act and take a step forward towards optimization. Although many of these sub-processes cannot and should not be averted, there is space for improvement. This improvement can speed up the user's navigation between applications maintaining the safety mechanisms previously implemented.

In The following chapter will focus on the proposal of a solution, with a practical example, that fits the architecture of the organization and has into consideration all the knowledge gathered in this document.

4.2. Requirement Analysis

Software engineering is done at its best when the requirements of the system are properly categorized and well-known to all the parties involved. Subsequent to the showcasing of the problem, done previously, it seems of great importance to understand what a possible new solution can bring to the table. The current sub-chapter will emphasize the requirements necessary to implement a solution that best fits the company's interest.

1) **Seamless Service Integration** – The proposed development must be able to integrate with the existing ecosystem of Jumia, and its development should cause as minimum impact as possible in the systems it integrates with.

2) **Eventual Consistency** – As there exists a way of accessing all applications currently, there is no need for blocking protocols that create entropy throughout the current architecture. As such, an eventually consistent environment should be promoted, where if unchanged data will sooner or later be coherent.

3) **Request Identification** – All the requests carried throughout the proposed model must be identifiable through an authentication token generated by Jumia's ACL. By doing so, the API becomes more resilient and compliant with the current protocols in place.

4) Critical Data Protection – All the critical data passed around in a HTTP-like environment should be encrypted. This information can consist of authentication tokens, exchangeable codes, or user emails.

5) **Available to all Projects** – The solution proposed to transition between environments should be available in every back-office application.

6) **Minimal Client Integration Effort** – The proposed solution must not pose a substantial overhead, either of costs or time, for the different teams to implement in their applications' clients. The integration with the different systems should be as simple as running a command and make small UI adjustments.

7) **Independently deployable** – The application should be capable of deploying independently of all the other systems at Jumia. Likewise, it should also not be impacted by any other project's deploy.

8) **Continuously Integratable and Continuously Deployable** – Any adjustments or new requirements should be able to be planned, developed, tested, and subsequently deployed effortlessly.

The first requirement (Seamless Service Integration) can be achieved by implementing an independent small service. The existing nature of Jumia's service architecture allows for the generation of a small service in an enclosed environment that can subscribe to the messages already being published by the different services. This property is advantageous to the second requirement (Eventual Consistency) since it provides an eventually consistent approach that does not hinder the data transaction between systems.

The third requirement (Request Identification) can easily be tackled with the injection of a bearer authentication token in the head of each request. This mechanism is already in place for every other Jumia's Services, so its integration would be rather effortless.

For the fourth requirement (Critical Data Protection), one of the most straight forward solutions here would be the generation of two RSA key-pairs. One of the key-pairs would be generated by the ACL and the other by the proposed solution. Subsequently the public keys would be exchanged and thus both applications can encrypt the outgoing data using an RSA protocol with the homologous' public key. This process is ideal because it is compliant with the low overhead and minimum impact requirement. Since the current system already supports RSA encryption.

The fifth (Available to all Projects) and the sixth requirement (Minimal Client Integration Effort) can be achieved through the implantation of a JavaScript library that can be imported by all projects though tools like Node Package Manager (NPM) or YARN.

Just like the first requirement suggests the implementation of an independent service can compel it to be independently deployable (Independently deployable), and by nature such a service can make use of tools like Jenkins and Docker to allow the creation of a pipeline that

allows for continuous integration and deployability (Continuously Integratable and Continuously Deployable).

4.3. Solution Proposition

This sub-chapter intends to lay the foundation in which the proposed solution is based on. In here the decision-making process will be broken down, in order to better analyze why this approach was chosen over others. The expertise and techniques applied and suggested in this chapter are supported by the body of knowledge referred in the previous chapters.

The current chapter introduced by an analysis over the requirements and the problem at hand, followed by a proposed architecture. The assessment of the proposed working model will be supplemented with a deeper glance at the technologies and tools utilized, as well as an explanation as wherefore they were chosen over their respective alternatives.

It is also important to mention that the approach hereby mentioned and exposed never got to be implemented in live servers, and as such the results inferred from it might be considered inconclusive.

4.3.1. Proposed Interaction Sequence

Upon assessing the problem at hand and evaluating the requirements, a proposal was drawn over a potential flow of authentication. The new flow had as a main goal the reduction of times the users were required to dial their credentials in order to gain access to the back-office applications. Consequently, speeding the flow of operations in the hubs and warehouses ever so slightly.

In Figure 12, the proposed model is illustrated in a sequence diagram that aids the visualization of the chain of events that lead to the authentication between applications. In this model, the first steps of the authentication remain untouched (1.1 – 3.3). The reason being that to be allowed access into the application, the users still need to introduce their credentials into the system. This first step of the authentication allows the web-client, to have in its possession the authentication token that will enable the future interactions between systems. However, the second phase of the application traversal is where the suggested solution innovates.

The following overview, assumes that the back-office application at hand, has imported and applied a library component for the Back-Office Integration Application (BOIA) into their frontend client.

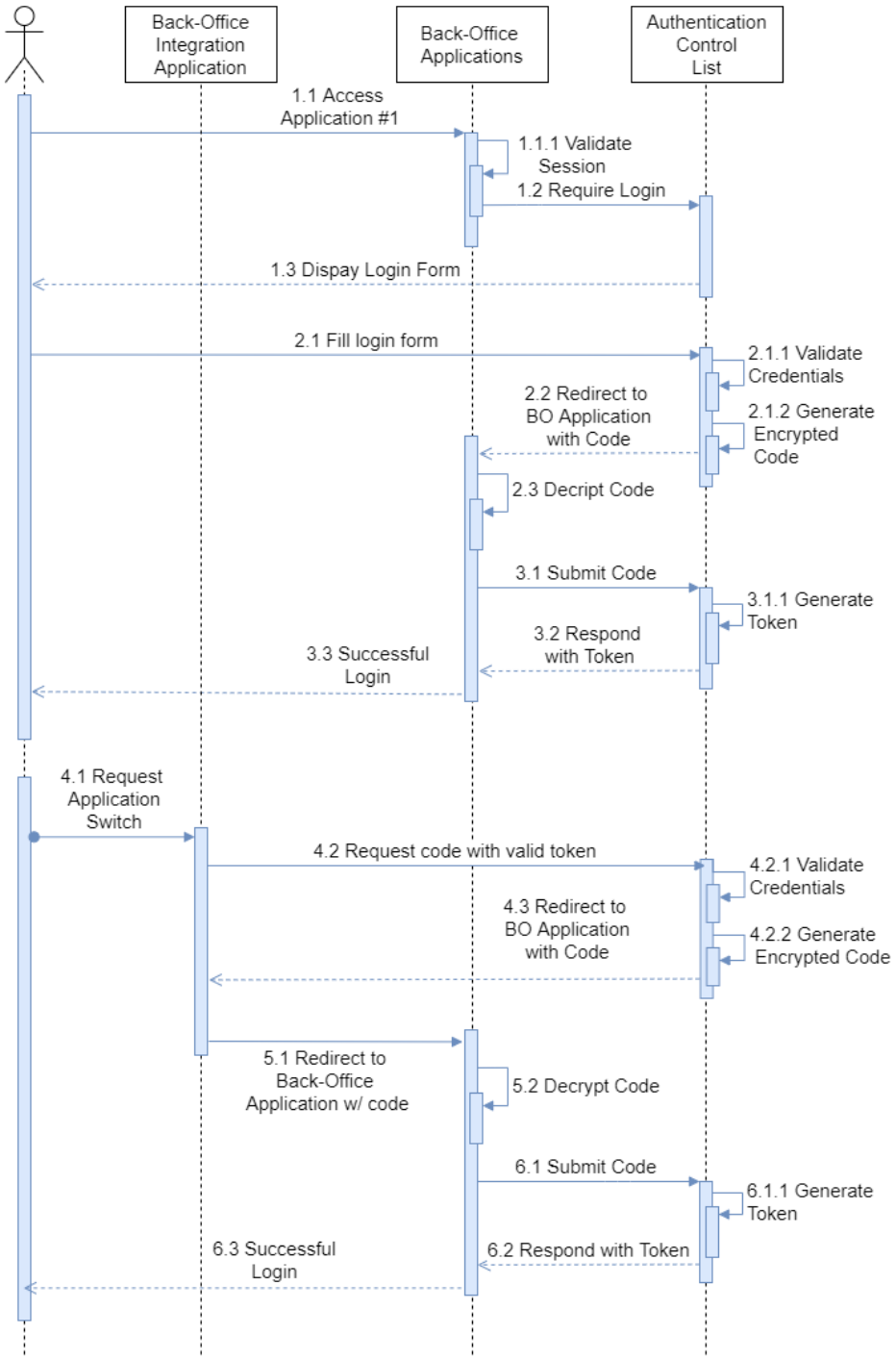

Figure 12 - Proposed Sequence Diagram for Application Traversal

The process is started when the users open the modal provided by BOIA, displaying the applications to which they have access to. The users can then select what is the application they want to navigate to and clicks the icon of said application. After that, a request is sent to the API of ACL for an exchangeable code. Although this time, as the request is being made directly to the API and without the insertion of user credentials, the previously generated token is used as a form of authentication. The ACL validates the given token and if it is legitimate, it

then generates a code and encrypts it, responding to the application with said encrypted exchangeable code. BOIA's frontend component then reroutes the browser to the URL of the selected application with the query parameter "code" holding the encrypted exchangeable code provided by ACL. From here, the processes already at place take over as each application already possess the mechanisms that allow them to decrypt and exchange the code that arrives from the query parameter "code" into a valid token that they can use to request their respective APIs.

4.3.2. Proposed Model

Taking into account the knowledge gathered earlier in the present document, together with the analyses of the system requirements, acquired in the previous chapter. The solution that seemed to be able to meet most, if not all, the requirements was the implementation of a microservice. The implementation of a microservice would allow for the seamless integration with other services through an eventually consistent system. As visually supported by Figure 13, this microservice exposes an API that is consumed by a small node package module that provides a component that Jumia's back-office applications can import and integrate into their clients.

As Jumia's back-office application clients are developed in Angular, the library provided by BOIA through a repository in Node Package Module (NPM) will consist of an Angular Module that can be included in the root module of any application. This module exports a component that allows the display of a modal where the user can visualize and choose between the applications to which they have access.

On the server side of BOIA, the goal is to achieve eventual consistency, through BASE-like transactions, over the data managed by the application. Since the relationship between users and applications does not represent critical information, it can be stored in a soft state. Meaning that it can be inaccurate at each point in time. By doing so, non-blocking processes can be implemented with the certainty that sooner or later, if the data is not modified, it will become up to date. The approach to populate and update the data of users and applications will consist of the consumption of messages that are published to a RabbitMQ's queue. The messages will be published by the User and Application Management Systems. By resorting to a message broker like RabbitMQ, it is possible to achieve loose coupling between the publisher and the consumer. This happens because at no point of the interaction they require to know about the existence or availability of one another. Each management system can simply populate its queue with messages, and on the other side the consumers will eventually process them according to their availability.

On the other hand, however, the interactions between the authentication service and BOIA need to follow an ACID-like transaction protocol. Since all of their interaction require the exchange of critical information, for instance the exchange of authentication codes and validation of tokens. At all points of interaction, the information carried between this to services needs to be accurate and consistent.

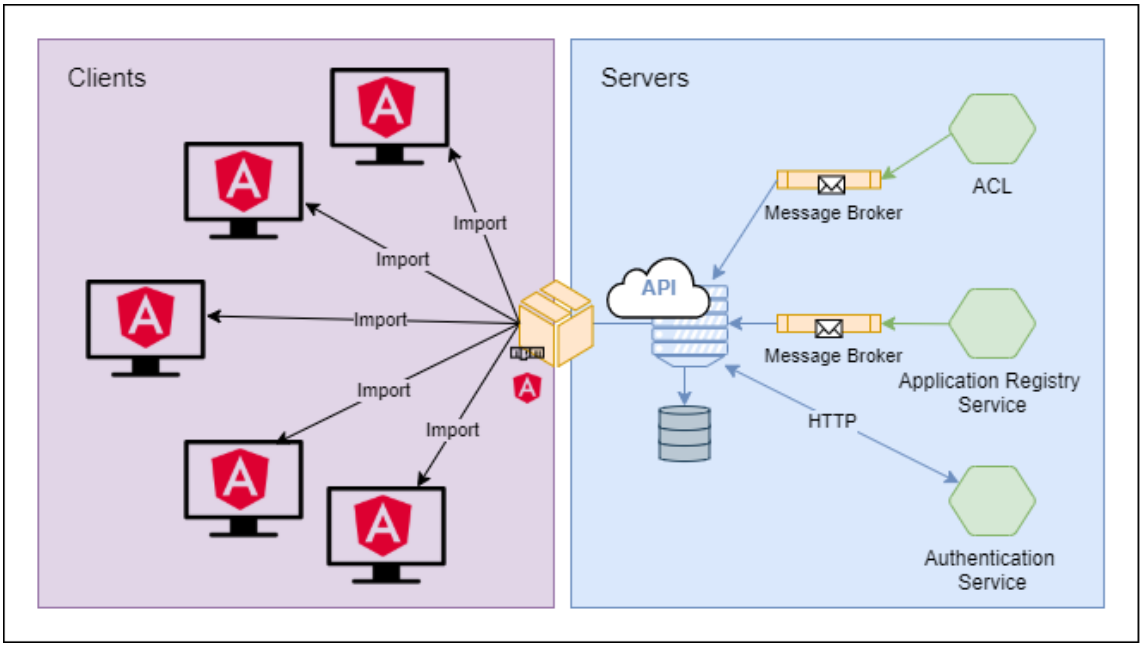

Figure 13 - Proposed Model

4.3.3. Data Management

After a quick analysis of the requirements and the proposed solution, it is easy to speculate that the systems of Jumia's back-office already possess all the data required for the implementation of BOIA. This is evident, given that there are APIs that populates the information about the applications and about the users with their own, via asynchronous messaging. If this is the case, why is there the need for a new system to aggregate the information between users and applications?

ACL, the application that owns the information of all users as well as their roles, in the back-office environment, is ultimately the service where the accesses of a certain role to each application is managed. As such, it needs to have a basic representation of the data of an application. In the case of Jumia's systems, this data corresponds to the unique key of the application, this key (usually comprised of an acronym for the application's name) is populated by the application registry each time a new application is created.

On the other side, the Application Registry Service owns the data of all the applications of the environment of Jumia (such as URLs, names, logos, etc.). The Application's Registry Service does not contain any information about users or their roles. In fact, this service has no idea about what a user is, it only knows applications, because that is its context boundary.

This being said, and with the aid of Figure 14, it is possible to assess what would need to be the data flow without the implementation of BOIA. To be able to visualize which applications they could transition to, given the role they possessed, a web client would need to query the API of ACL to see what are the applications that the authenticated user's role has access to. After having this information, the application would need to query the Application Registry's API to match each application key to their own application object and only then return to the client all the information that the user has access to. This process proves itself inefficient because it follows a synchronous interaction pattern that is prone to fail, gambling on the availability of all systems involved to return data in useful time.

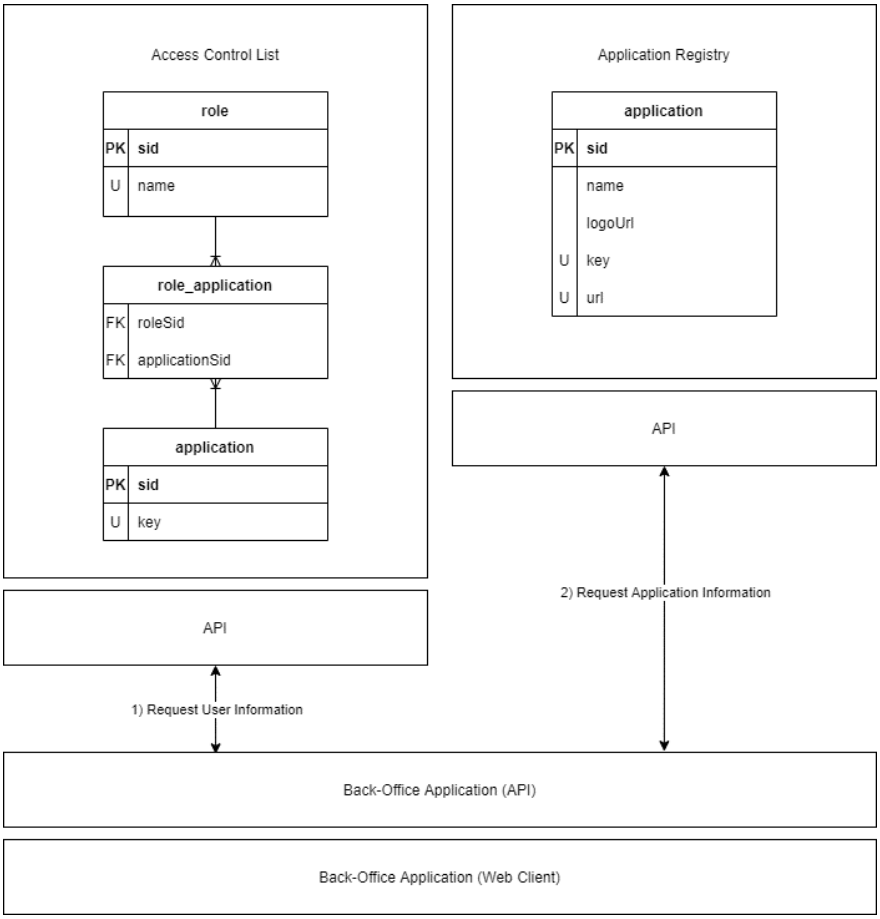

Figure 14 - Data flow according to previous data ownership

Instead of the model displayed and exposed above, BOIA proposes the synchronization of data as it arrives to the system. As such it is never dependent on the availability of others. For

instance, if both ACL and the Application Registry happen to be down, then the updates just stop being published and BOIA keeps serving its clients with the data it holds at the moment. It is not crucial that the data is perfectly accurate, since at the time of authentication, there are still bulkheads in place that protect the system from faulty scenarios. These bulkheads are often situated in the Authentication Service. For example, if a user tries to access an application that he does not have access to because the information of BOIA is not up to date, then the access will simply be denied, and a message will be shown. This way BOIA can simply concern itself in showing the data it owns, even if in a soft state, with the assurance that no inconsistencies will have a negative impact in the big picture of the system.

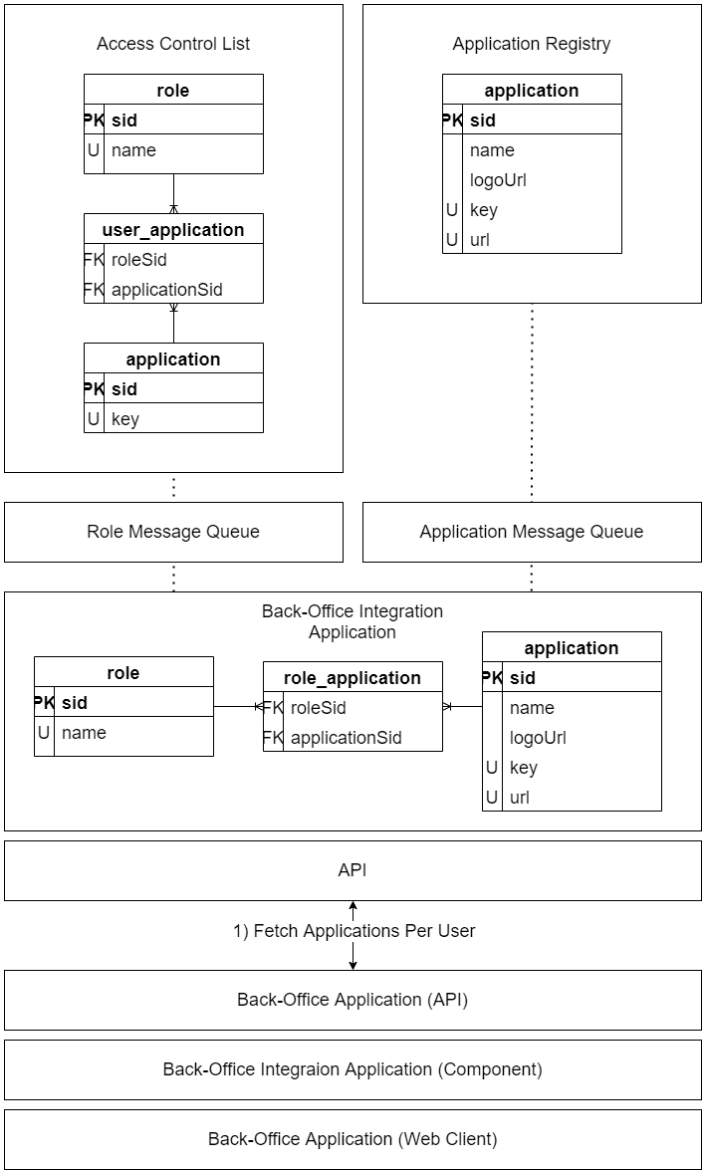

Figure 15 - Data flow with the implementation of BOIA

Figure 15 displays the way that the implementation of BOIA changes the data flow showcased previously. By subscribing and storing the essential information published by both the ACL and the Application Registry system, BOIA can relate the roles with the applications they have access to. In doing so, the back-office client can use BOIA's web component to query its API to fetch the data to be displayed to the end-user. This approach differs from the previous because it does not rely on a chain of events in order to provide the information to the user. It simply needs to query the service's database to produce a valuable result.

5. High-Level Implementation

5.1. Development Environment

After carefully clarifying the task at hand and subsequently evaluating and analyzing the requirements of the system, a high-level implementation was made. This implementation has the vision to serve as a solution to the current mismanagement of Jumia's back-office applications' transitions.

The development environment of this project was isolated from Jumia's, this posed a limitation for the project. As such, a decision was made to mock the services of Jumia, that would be required to create a fully functioning environment that would bring the implementation of BOIA to life.

5.1.1. Application Registry

To simulate the application registry service, a simple CRUD application was developed. The application consists of a simple Node.js API, with a non-relational database using MongoDB. The application registry mock was designed to receive an object containing the data of a newly created service/application that is introduced in Jumia's environment and subsequently store it in the database. Upon getting a request to create said service/application, the API constructs a message with the data relevant to BOIA and publishes it onto a queue. This queue will then be subscribed by BOIA.

5.1.2. Access Control List (ACL)

To this day, ACL is still one of the biggest and most complex services in Jumia's environment. It stores information about users, roles, role permissions, application accesses, authentication, and so on. Being so, it also becomes complex to integrate with, given that systems that manage such critical data require a high standard of security and compliance. For the high-level implementation presented in this chapter, ACL was mocked and only the role management and authentication capabilities were replicated to an extent where the integration between systems was possible.

5.2. Server API

5.2.1. Implementation

The business logic of BOIA is abstracted away within a RESTful API. This server was developed with the resort of NodeJS with Express. The choice of NodeJS as the main tool of development is due to its non-blocking properties provided by NodeJS' Event Loop [75]. These properties are then enhanced using Express, a framework that applies a thin layer, providing all the tools required to create a REST API, without obscuring away the simplicity and versatility of NodeJS' features [76]. Another implementation choice worth noticing was the adoption of Typescript [77]. By offering statically typed options and OOP tools Typescript allowed for a more solid object-oriented paradigm, which is optimal for large enterprises where code consistency is a must for a seamless integration between systems.

In an initial phase of the API development the models of a role and of an application were designed. In the current implementation a Role consists of an **id (**generated by the database, as a primary key of the object), a **name**, a **description (**brief depiction of what the role consists of), a **key** (serving as a unique identifier of the role object, within other objects' scopes), an **expiration date** (representing a timestamp at which the role becomes deprecated/invalid), a **date of creation**, a **date of last update** and a **date of deletion** if applied. The Role model contract is displayed in Listing 1, and it represents the baseline structure that each role implementation should follow.

```
export interface IRole {

  id: number;
  name: string;
  description: string;
  key: string;
  expirationDate: Date;
  createdAt: Date;
  updatedAt: Date;
  deletedAt: Date | null;
}
```

Listing 1 - Role model interface

Depicted in Listing 2, is the data schema of what represents an Application in BOIA is embodied by an **id** (generated by the database, serving as a primary key for the object), a **name,** a **description** of what the application does in the environment of Jumia, a **code** that represents a unique value of how the application is called (typically an acronym for the

application, e.g. Order Management System becomes OMS), the **URL** in which the application is provided (this is crucial to allow the redirect of the user), the **URL of the application's logo**, as well as a **creation date**, **last update date** and a **deletion date** (if applied).

```
export interface IApplication {

    id: number;
    name: string;
    description: string;
    code: string;
    url: string;
    logoUrl: string;
    createdAt: Date;
    updatedAt: Date;
    deletedAt: Date | null;
}
```

Listing 2 - Application model interface

The creation of these interfaces allows for the further implementation of models, these serve as a data access object (DAO) layer, provided under an object relational mapper (ORM) available through the library Sequelize [78]. This model can be then exported and injected into the services in order to be used to query the database either to fetch, create or update data. Listing 3 displays the example of the implementation of a Sequelize ORM model.

```
export class RoleModel extends Model<Role> implements Role {

    @PrimaryKey
    @AutoIncrement
    @Column
    id: number;

    @Column
    name: string;

    @Column
    description: string;

    @Column
    @Unique
    key: string;

    @Column(DataType.DATE)
    expirationDate: Date;

    @BelongsToMany(() => ApplicationModel, () => ApplicationRoleModel)
```

```
    applications: Array<ApplicationModel & ApplicationRoleModel>;

    @CreatedAt
    createdAt: Date;

    @UpdatedAt
    updatedAt: Date;

    @DeletedAt
    @AllowNull
    deletedAt: Date | null;
}
```

Listing 3 - ORM Model example of a Role

The business logic of the application resides in the service-layer. Services were implemented in the shape of classes that have methods (behaviors) that mirror the requirements of the system. Services allow for a better granularity of the code by segregating the logic from the controllers and by doing so, preserving the sanity of the code. For instance, as exposed in Listings 4-5, when a request arrives in the endpoint '`roles/{id of role}/applications`', the role service is invoked with the method to get the applications by role and the role id - retrieved from the request's query parameters - is provided. The service then proceeds to call the DAO to fetch all the relationships between applications and roles, where the role id matches the provided id. The result is then returned, or an error is thrown to the http client.

```
App.get(
    '/roles/:id/applications',
    async (request: Request, response: Response) => {
    try {
        const id: number = +request.params.id;

        response
            .status(200)
          .json(await roleService.getApplicationsByRole(id));
    } catch (error) {
        handleHttpError(error, response);
    }
});
```

Listing 4 - Example of the Role controller (GET application by role endpoint)

```
    export class RoleService {
```

```
…
        public async getApplicationsByRole(
            roleKey: string):
            Promise<Application[]> {

         const applications: Application[] | null =
               await this.roleDAO.getApplicationsByRole(roleKey);

         if (applications === null) throw new NotFoundException();

         return applications;
    }
…
}
```

Listing 5 - Example of Role Service, with the method to fetch applications by Role

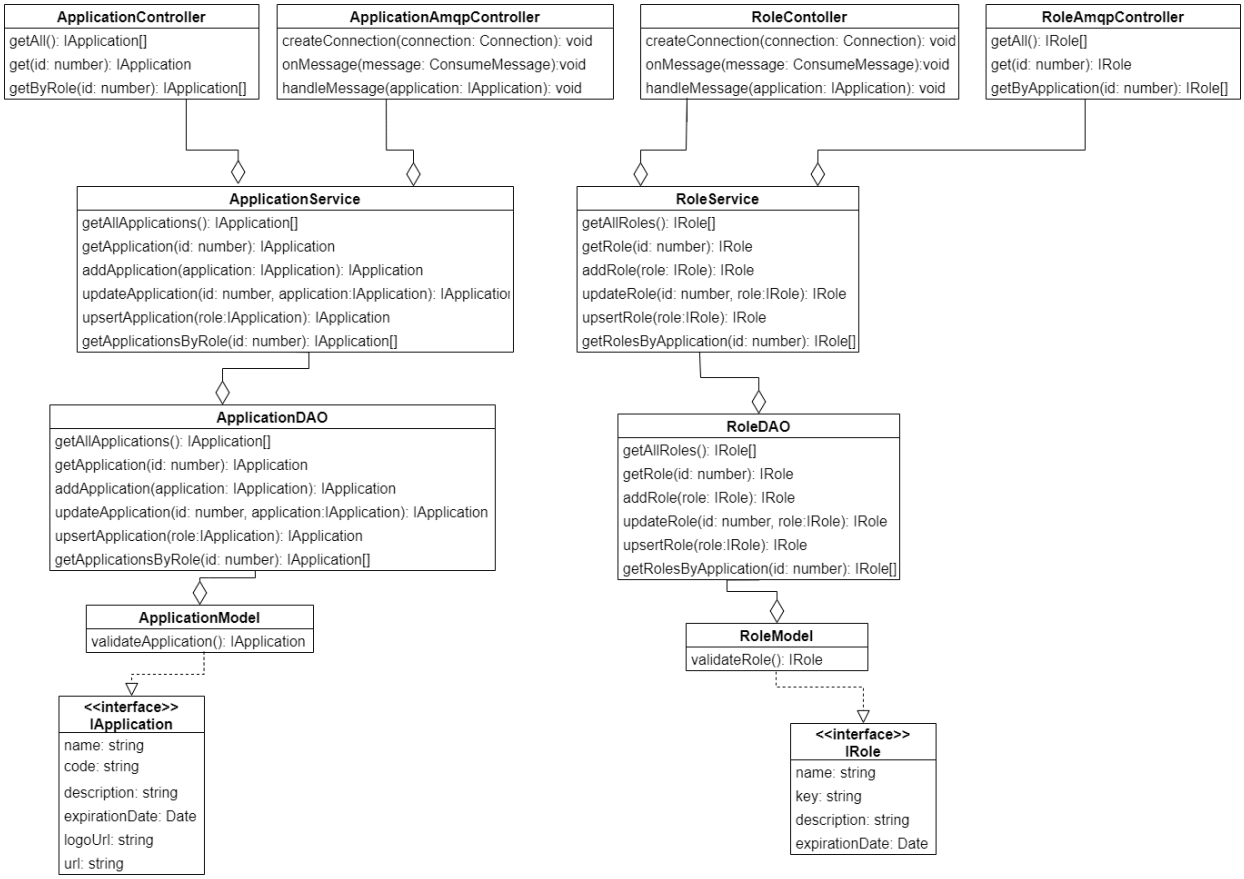

Figure 16 - Class Diagram of BOIA

As previously explained above and as depicted in the class diagram found in Figure 16, the API application follows a pattern in which interfaces define the data structure and serve as a guideline to implement the ORM's models. These models are then used by DAO to access the database and execute queries with the purpose of modifying or retrieving content. The

business logic is then implemented by the services that uses the DAO to mold and modify the requests' bodies and convert them into valuable and usable responses. While using a RESTful protocol through express, the controllers (`ApplicationController` and `RoleController`) are used to serve the data fetched and processed by the services. Whilst in the AMQP controller, the services are used to validate the received messages and handle them accordingly.

5.2.2. Middlewares

To ensure the API works correctly, there was a need to ensure that all the requests had the adequate form to be able to be consumed. A middleware can also be used to enrich or hydrate the response before it is sent out to the client. In the development of BOIA the middleware that seems more interesting to discuss is the one responsible for the authentication, the token validation middleware.

In the token validation middleware, demonstrated below in Listing 6, the request is intercepted, and the 'Authorization' property of the request's header is scrutinized. This property should correspond to a valid authentication token. The token is then sent to the authentication service, where some validations are going to be asserted and a response is going to be produced, if the authentication is correct a boolean value of true is received, false otherwise. In the case of the validation succeeding the process is continued and the request is then forwarded in the direction it was intended to. However, if the authentication fails, an immediate response is generated with the http status of 401 (meaning unauthorized), preventing the unauthenticated users to query the API.

Inevitably, the authentication validation is made in a synchronous manner, due to the high consistency that such a process requires. If for instance the authentication token is not valid, then the client cannot access the data owned by the API and as such is not able to proceed. Therefore, it needs to follow an ACID-like transaction enlightened in chapter 2.5.3 of this document.

```
async function validateUserToken(
   request: Request,
   response: Response,
   next: NextFunction): Promise<any> {

   try {
      const authorized: boolean = await authenticationService
          .validateUserToken(request.headers.authorization);

      if (authorized) return next();

      response.status(401).send('Unauthorized');
```

```
    } catch (error) {
      if (error instanceof HttpException) {
        response.status(error.getStatus()).json(error.toJson());

        return;
      }

      response.status(401).send('Unauthorized');
    }
  }
```

Listing 6 - Token Validation Middleware

5.2.3. Fault Tolerance

As assessed before, in a distributed environment, nodes break easily. This means that developers need to take measures that assure that each service keeps functioning in the best way possible without being affected by said failure. In some cases, the unaffected functioning of the application is not possible, as such developers should move to the next best solution possible – failure tolerance.

In the case of BOIA, the main point of failure that requires attention is the authentication process, in here another service (ACL) is invoked. And its recurring unavailability might mean a constraint for the application. As such some measurements were taken, for instance, every request fired has configured a timeout of 15 seconds. This means that, provided ACL does not return a response within 15 seconds, then the request is aborted, and a failure response is returned.

Yet another fault tolerance measure approached in this document and implemented by BOIA are the circuit breakers (chapter 2.5.7). In order to implement a circuit breaker in BOIA the library “Opossum” was used. As the developers’ description cites: “Opossum is a Node.js circuit breaker that executes asynchronous functions and monitors their execution status. When things start failing, opossum plays dead and fails fast” [79]. This required some changes to the code displayed in Listing 6, originating the code displayed in Listing 7 and 8. With the new implementation, the API call is now wrapped by a circuit breaker, this means that the circuit braker will accumulate failures to a total of 3 (as indicated in the configurations under the property ‘`maxFailures`’) and stop calling the service for 5 minutes (‘`resetTimeout`’ property in listing 8), providing a fast failing environment where a response is immediately generated. After the timeout expires the process repeats itself until availability is asserted.

```
const circuit = new CircuitBreaker(
    () => authenticationService
       .validateUserToken(request.headers.authorization);
    , circuitBreakerOptions);


async function validateUserToken(
   request: Request,
   response: Response,
   next: NextFunction): Promise<any> {

   try {
       const authorized = await circuit.fire();

       const authorized: boolean = await authenticationService
          .validateUserToken(request.headers.authorization);


      if (authorized) return next();

      response.status(401).send('Unauthorized');
  } catch (error) {
    if (error instanceof HttpException) {
      response.status(error.getStatus()).json(error.toJson());

      return;
    }

    response.status(401).send('Unauthorized');
  }
}
```

Listing 7 - Token Authentication Middleware with a Circuit Breaker

```
export const circuitBreakerOptions = {
    timeout: 15000,
    maxFailures: 3,
    resetTimeout: 300000,
};
```

Listing 8 - Circuit breaker configurations

5.3. Messaging Queues

In order to achieve loose coupling between services, asynchronous communication was implemented wherever possible. By doing so, BOIA's API could attain resilience regardless of the availability of the applications it intended to integrate with. To achieve true decoupling messaging queues were used, resorting to RabbitMQ, this would allow the producers to only know the address of the exchange.

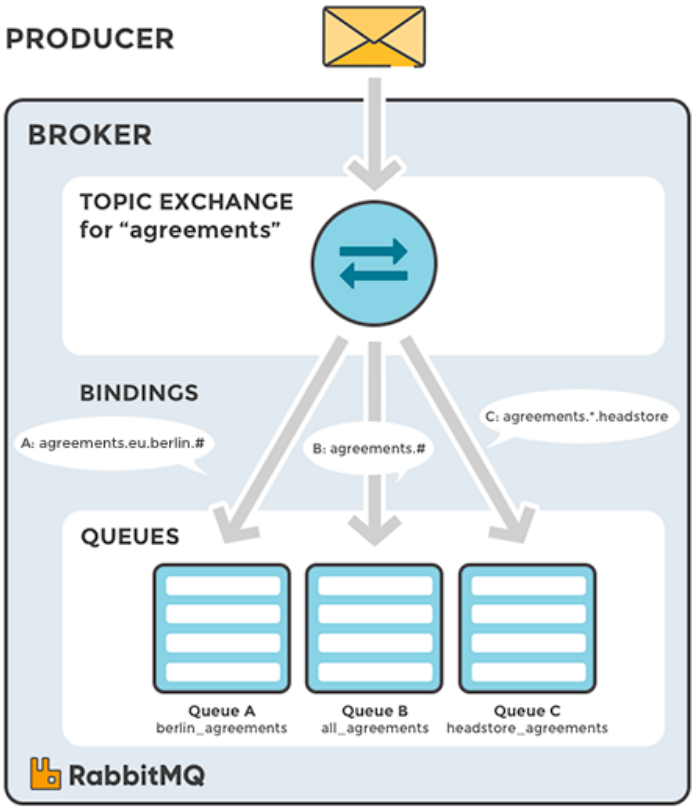

Figure 17 – RabbitMQ's Topic Exchange *[80]*

The configuration of the RabbitMQ's ecosystem started with the creation of two exchanges, one for applications and another one for roles. These exchanges were both configured with a type 'Topic' allowing each message to be routed to a specific queue based on a routing key. Like so, messages can fall into distinct queues and be treated differently. For instance, both roles and applications, have three queues each that are then subscribed to by three different handlers that treat each message as a creation, edition or deletions.

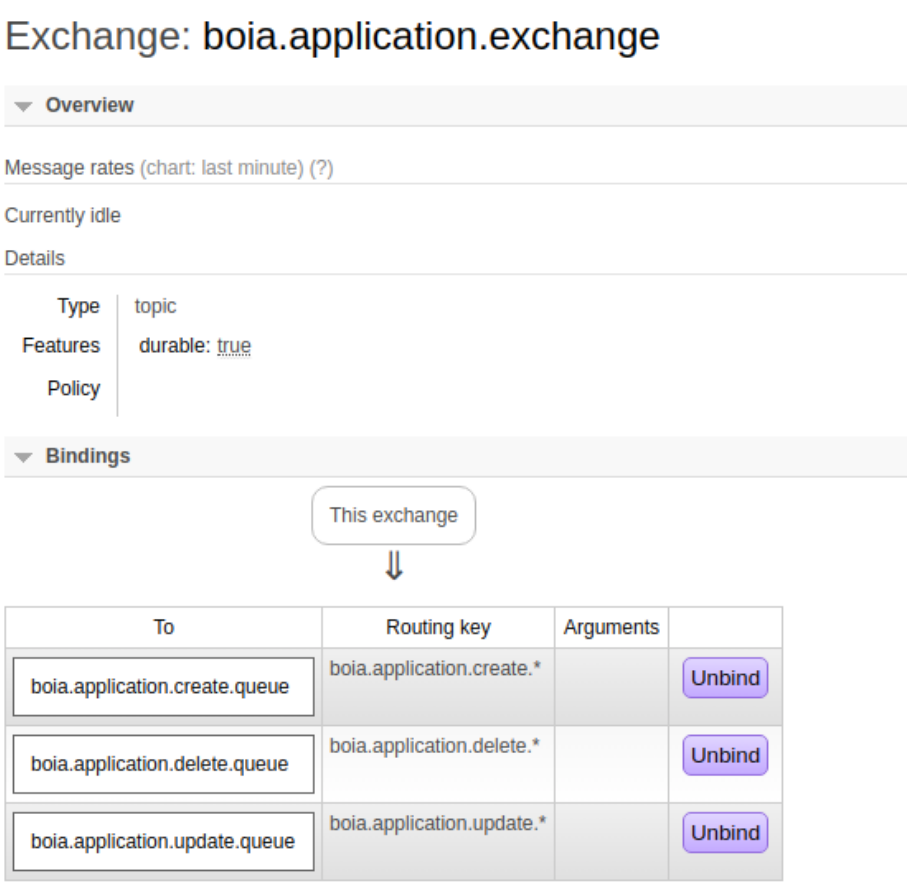

Figure 18 - Example of Application Exchange's Routing Configuration

5.4. Client

To develop the client application to of BOIA, the ideal choice was Angular. This choice became obvious, given that all the back-office applications of Jumia are developed in Angular, and this means that the integration between systems would become easier and would not require further framework adjustments. The main goal of the development of the client application is to develop a component, that can be imported through NPM into any of the back-office applications. This component allows the users of the application it was implemented at to open a modal and see all the back-office applications they have access to, with the role that they are currently authenticated with.

Upon being initialized, the component checks the health of the API, and if the latter is up and running, then the component is displayed. However, if otherwise the API is unavailable, then the component is hidden. This health check allows each client to better manage its resources, preventing them from generate HTTP requests that are expected to fail even before they are fired. If the users do not have access to any other application or the application fetch

failed, then an error message is displayed, and the user can retry the request at any time (depicted in Figure 19).

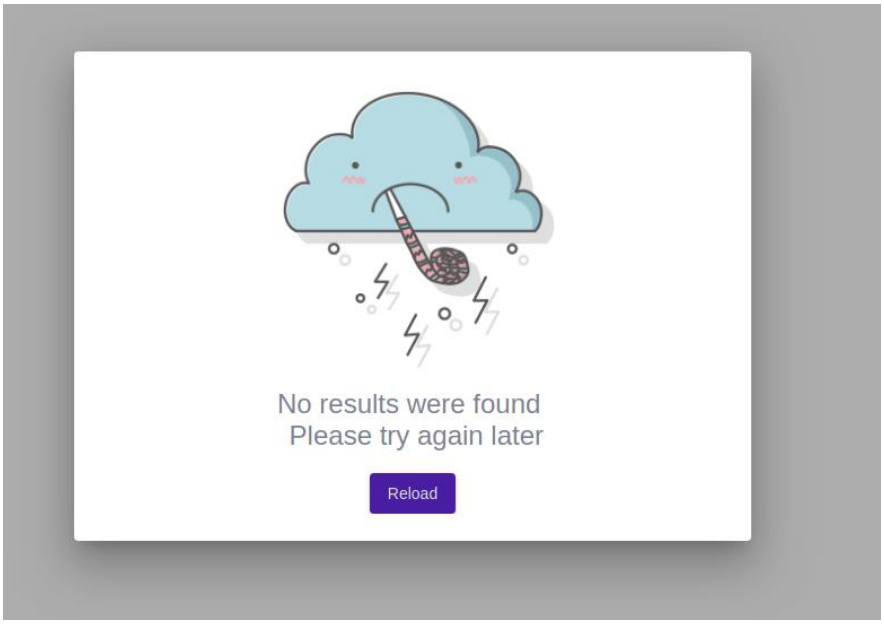

Figure 19 - Example of an application modal with no applications to display

On the other hand, however, when the request for applications is successful the modal is filled in a grid-like pattern with all the applications that the user's role gives access to (depicted in Figure 20).

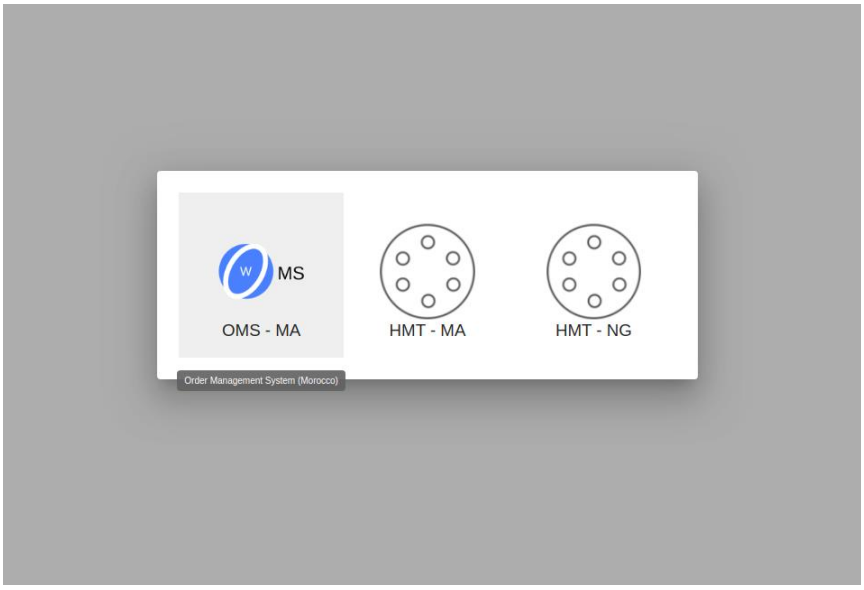

Figure 20 - Modal with applications

Upon selecting one of the applications in the modal, a request is sent to the API in order to retrieve an authentication code for the selected application. When the authentication code arrives in the frontend application, it mounts the URL by joining the base URL from the chosen application with the query parameter code holding the received value. The back-office application will then exchange the code for a valid token, in very much the way it did and regardless of having BOIA integrated already.

```
public redirectTo(application: Application): void {
  const codeBearerParam: string = this._config.redirectCodeBearer || 'code';
  this.getAuthorizationCode(application: Application).subscribe(res => {
    const authenticationCode: string = res;
    const redirectUrl: string =
        `${application.url}?${codeBearerParam}=${authenticationCode}`;

    if (this._config.openApplicationInNewTab) {
      window.open(redirectUrl, "_blank")
    } else {
      window.location.href = redirectUrl;
    }
  })
}
```

Listing 9 - Redirect to the chosen application

As the library is intended to be imported by different applications, it needs to possess appropriate configurations that allow each team to mold it around their business requirements. At its current state the client allows for the configuration of the number of applications per row, the width of the dialog, the URL to where the code can be retrieved and the query parameter in which the authentication token will be carried upon redirect, as well as some Material Library's configurations for the modal and the opening button component.

```
export class Config {
    closeDropDownMessage?: string;
    columns?: number;
    customDialogOptions?: MatDialogConfig;
    dialogWidth?: string;
    getApplicationsUrl?: string;
    getApplicationsHeaders?: Params;
    materialButtonIcon?: string;
    openApplicationInNewTab?: boolean;
    redirectCodeBearer?: string;
    retrieveCodeUrl: string;
    retrieveCodeHeaders?: Params;
}
```

Listing 10 - Configurations of BOIA's client

6. Conclusion and Future Work

6.1. Conclusion

Microservices are getting more and more popular by the day. They do not provide a silver bullet for the problems of every project, instead they provide an architecture that allows applications to scale their services in a more granular manner. When applied correctly, microservices can prolong and promote code quality, given that each scope is encircled within its boundaries, therefore relying on fewer dependencies.

In the case of study demonstrated and explored in this document, the implementation of a microservice that associates roles to applications, whilst holding the detailed information of each application seemed like the best option. This is due to the very nature of Jumia's architecture where microservices were already at work. Furthermore, the conceptualization of the proposed service means that the code necessary to associate roles to applications is encapsulated away in a single service, and as such does not need to be replicated through innumerous services, which would greatly increase the scope of the feature as well as the likelihood of bugs.

Regarding the expected results for the current dissertation:

- An application was conceptualized that prevents users from having to login every time they want to switch between applications. Provided they have a valid session ongoing;
- A modal was developed that displays to the user what are the applications it has access to, mitigating the necessity to actively look between applications to assess which ones are accessible to him;
- The overall flow pace increase was inconclusive since it was not possible to effectively integrate the proposed solution with Jumia's systems. Despite this, an argument can be made that for most of the cases, when the system works as intended, the proposed solution has the potential to be swifter given that an exchange between systems tends to be faster than a user authentication interaction;
- A deep analysis was conducted, with the discern of creating stronger bases of what microservices represented and how they interconnect, as well as their benefits and

drawbacks. This combined with the research of several documented and distinguished articles, papers and books allowed for a deepening and strengthening of the knowledge about the subject.

6.2. Future Work

The work showcased in this document serves as a departure point of what can be a real-world implementation that intends to mitigate a real requirement of Jumia. As of today, it might not be entirely possible to implement such a solution, due to the constraints and roadmaps of the teams, but perhaps this work will serve as a study case that leads to the implementation of a similar solution.

Following to that, being implemented into a real case scenario a solution needs to be crafted to allow for the encryption of the authentication data that is carried around in http requests. The same goes for the request themselves, in a real-world application maybe the HTTP protocol should be replaced by a protocol with stronger security, such as HTTPS.

In the client, the future work would pass through the possibility for developers to style the components so that the style guide of the modals would follow the patterns of each application. As with this implementation, that styling can only be achieved resorting to the Angular Material Lib.

Regarding microservices and their integration, the future work would lie in the better exploration of deployment strategies following patterns of continuous delivery/continuous integration. In a way that would assure that the integration between services would be unbroken upon deployment resorting to a reliable test battery.

7. References

[1] R. Venkatesan and V. Kumar, *A Customer Lifetime Value Framework for Customer Selection and Resource Allocation Strategy,* 1 October 2004.

[2] M. Fowler, "StranglerFigApplication," 29 June 2004.

[3] V. Velpucha, P. Flores and J. Torres, "Migration of Monolithic Applications Towards Microservices Under the Vision of the Information Hiding Principle: A Systematic Mapping Study," *Advances in Emerging Trends and Technologies,* 2020.

[4] D. Escobar, D. Cádenas, R. Amarillo, E. Castro, K. Garcés, C. Parra and R. Casallas, "Towards the Understanding and Evolution of Monolithic Applications as Microservices," 2016.

[5] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca and C. R. , "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud," 2015.

[6] C. Richardson, "Pattern: Monolithic Architecture," [Online]. Available: https://microservices.io/patterns/monolithic.html. [Accessed 2020 June 24].

[7] N. Dragoni, S. Giallorenzo, A. L. Lafuente and M. Mazzara, "Microservices: yesterday, today, and tomorrow," 20 April 2017.

[8] M. Villamizar, O. Gárces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zabrano and M. Lang, "Infrastructure Cost Comparison of Running Web Applications in the Cloud using AWS Lambda and Monolithic and Microservice Architectures," 2016.

[9] M. Fowler and J. Lewis, "Microservices," 25 March 2014.

[10] C. Richardson, "Introduction to Microservices," 19 May 2015.

[11] B. Familia, "What Is A Microservice," *Microservices, IoT, and Azure,* 2016.

[12] J. Lewis and M. Fowler, "Smart endpoints and dumb pipes," *Characteristics of a Microservice Architecture,* 24 March 2014.

[13] M. Kalske, N. Mäkitalo and T. Mikkonen, "Challenges When Moving from Monolith," 22 February 2018.

[14] R. C. Martin, Clean Code, Prentice Hall, 2009.

[15] S. Newman, Building Microservices - Designing Fine-Grained Systems, 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc, 2015.

[16] J. Thönes, "Microservices," January 2015.

[17] C. Richardson, "Building Microservices: Using an API Gateway," 25 June 2015.

[18] N. T. Blog, "Embracing the Differences : Inside the Netflix API Redesign," 9 July 2012.

[19] M. Wenzel, Y. Victor, J. Parente and D. Pine, *Challenges and solutions for distributed data management,* 20 September 2018.

[20] J. Lewis and M. Fowler, "Characteristics of a Microservice Architecture," 25 March 2014.

[21] M. Conway, "Conway's Law," 1967.

[22] A. Hochrein, Designing Microservices with Django, 2019.

[23] M. Fowler, "MonolithFirst," 3 July 2015.

[24] T. Bures and L. Duchien, Software Architecture, Paris: Springer, 2019.

[25] S. Sarkar, G. Vashi and A. PP, "Towards Transforming an Industrial Automation System from Monolithic to Microservices," in *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2018.

[26] "Wikipedia on Strangler fig," 22 July 2020. [Online].

[27] C. Richardson, "Refactoring a Monolith into Microservices," 8 March 2016.

[28] F. Martin, "SacrificialArchitecture," 3 June 2015.

[29] R. Shoup, "Evolutionary Architecture," 16 October 2014.

[30] S. Tilkov, "Don’t start with a monolith," 09 June 2015.

[31] F. P. B. Jr., "No Silver Bullet - Essence and Accident in Software Engineering," 1987.

[32] S. Shariq, M. Wenzel, J. Parente, N. Anil and M. Veloso, "Communication in a microservice architecture," 2020 30 January.

[33] B. M. I.-P. C. i. a. M. Architecture, "Richardson, Chris;," 24 July 2015.

[34] M. Satran and M. Jacobs, "Interprocess Communications," 31 May 2018.

[35] L. Liu and M. Tamer Özsu, Encyclopedia of Database Systems, New York: Springer, 2009.

[36] A. Fox and E. Brewer, "Harvest, Yield, and Scalable Tolerant Systems," 1999.

[37] B. Christudas, Practical Microservices Architectural Patterns, Springer, 2019.

[38] M. José Escalona, F. Domínguez Mayo, T. A. Majchrzak and V. Monfort, Web Information Systems and Technologies, Cham, 2019.

[39] A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera and A. Sadovykh, Microservices - Science and Engineering, Cham, 2020.

[40] M. Satran and M. Jacobs, "What is a Transaction?," 31 May 2018.

[41] B. Christudas, Practical Microservice Architecural Patterns -, Trivandrum, 2019.

[42] J. McGovern, O. Sims, A. Jain and M. Little, Enterprise Service Oriented Architecture - Concepts, Challenges, Recommendations, Netherlands: Springer, 2006.

[43] M. Younas, B. Eagelstone and R. Holton, A Review of Multidatabase Transactions on the Web: From the ACID to the SACReD., 2000.

[44] W. Vogels, "Building reliable distributed systemsat a worldwide scale demands trade-offs between consistency and availability.," *Communications of the ACM,* vol. 52, no. 1, p. 42, 2009.

[45] D. Bermbach, L. Zhao and S. Sakr, "Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services," August 2013.

[46] J. D. Cook, "ACID versus BASE for database transactions," 6 July 2009.

[47] D. Monteiro, R. Gadelha, P. H. Maia, L. Rocha and N. Mendonça, "Beethoven: An Event-Driven Lightweight Platform for Microservice Orchestration," 2018.

[48] J. Schabowsky, "Microservices Choreography vs Orchestration: The Benefits of Choreography," 26 November 2019.

[49] C. Ricardson, "Building Microservices: Inter-Process Communication in a Microservices Architecture," 24 July 2015.

[50] B. Christudas, "Distributed Messaging," in *Practical Microservices Architectural Patterns*, c, 2019, pp. 105-145.

[51] M. (. .. d.-1.-4.-3.-4. Macero, "Starting with Microservices," in *Learn Microservices with Spring Boot*, 2017, p. 99–177.

[52] A. R. Sampaio Jr., J. Rubin, I. Beschastnikh and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," 2019.

[53] K. Indrasiri and P. Siriwardena, "Integrating Microservices," in *Integrating Microservices. Microservices for the Enterprise*, 2018, pp. 167-217.

[54] R. Sharma and A. Singh, "Getting Started with Istio Service Mesh.," in *Retry Requests*, 2020, p. 205.

[55] M. Nygard, Release It! Design and Deploy Production-Ready Software, 2nd edn., 2018.

[56] J. Postel, "Transmission Control Protocol," January 1980.

[57] M. L. A. &. M. T. Fisher, The Art of Scalability, 2009.

[58] K. Apte, "SCALING APPLICATIONS : THE SCALE CUBE," 6 March 2018. [Online]. Available: https://geeknarrator.com/2018/03/06/scaling-applications-the-scale-cube/. [Accessed 1 August 2020].

[59] C. Richardson, "The Scale Cube," [Online]. Available: https://microservices.io/articles/scalecube.html. [Accessed 1 August 2020].

[60] K. Meinke and P. Nycander, "Learning-Based Testing of Distributed Microservice Architectures: Correctness and Fault Injection," *Lecture Notes in Computer Science,* pp. 3-10, 2015.

[61] L. Crispin and J. Gregory, Agile Testing - A Pratical Guide for Testers and Agile Teams, United States of America: Pearson Education, Inc., 2009.

[62] I. Robinson, "Consumer-Driven Contracts: A Service Evolution Pattern," 12 June 2006.

[63] J. Lehvä, N. Mäkitalo and T. Mikkonen, "Consumer-Driven Contract Tests for Microservices: A Case Study," *Product-Focused Software Process Improvement. Lecture Notes in Computer Science.,* 2019.

[64] H. Vocke, "The Practical Test Pyramid," 26 February 2018.

[65] J. Stählin, S. Lang, F. Kajzar and C. Zirpins, "Consumer-Driven API Testing with Performance Contracts," *Advances in Service-Oriented and Cloud Computing,* p. 135–143, 2018.

[66] P. Siriwardena, "APIs Rule!," in *Advanced API Security*, 2020, pp. 24-25.

[67] T. Mauro, "Adopting Microservices at Netflix: Lessons for Architectural Design," 19 February 2015.

[68] C. &. S. T. Carneiro, "Microservices From Day One," in *Microservices: The What and the Why*, 2016, pp. 3-18.

[69] B. Christensen, "Netflix Technology Blog - Introducing Hystrix for Resilience Engineering," 26 November 2012. [Online]. [Accessed 28 July 2020].

[70] K. Andrus, N. Gopalani and B. Schmaus, "Netflix Technology Blog - FIT: Failure Injection Testing," 23 October 2014. [Online]. Available: https://netflixtechblog.com/fit-failure-injection-testing-35d8e2a9bb2. [Accessed 30 July 2020].

[71] C. Bennett and A. Tseitlin, "Netflix: Chaos monkey," 2012.

[72] M. E. Conway, "How Do Committees Invent?," April 1968.

[73] M. Cruth, "Discover the Spotify model at Atlassian," [Online]. Available: https://www.atlassian.com/agile/agile-at-scale/spotify.

[74] A. Gluck, "Introducing Domain-Oriented Microservice Architecture," Uber Engineering, 23 July 2020. [Online]. Available: https://eng.uber.com/microservice-architecture/. [Accessed 30 July 2020].

[75] "The Node.js Event Loop, Timers, and process.nextTick()," OpenJS Foundation, [Online]. Available: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/. [Accessed 31 August 2020].

[76] "Express," [Online]. Available: https://expressjs.com/. [Accessed 31 August 2020].

[77] "Typescript," [Online]. Available: https://www.typescriptlang.org/. [Accessed 31 August 2020].

[78] "Sequelize," [Online]. Available: https://sequelize.org/. [Accessed 31 August 2020].

[79] "NPM - Opossum," [Online]. Available: https://www.npmjs.com/package/opossum. [Accessed 2020 September 1].

[80] L. JOHANSSON, "CloudAMQP - Part 4: RabbitMQ Exchanges, routing keys and bindings," [Online]. Available: https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html. [Accessed 2 September 2020].

[81] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano and A. Urso, "The Database-is-the-Service Pattern for Microservice Architectures," 2016.

[82] Y. Aradhye, "Principles for Microservices Integration," 24 August 2018.

[83] A. Rotem-Gal-Oz, SOA Patterns, New York: Manning Publications Co., 2012.

[84] S. Tilkov, "Don’t start with a monolith," 9 June 2015.

[85] J. Fritzsch, J. Bogner, A. Zimmermann and S. Wagner, From Monolith to Microservices: A Classification of Refactoring Approaches, 2019.

[86] S. Brown, "Distributed big balls of mud," 6 July 2014. [Online].