INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Proposta de uma arquitetura de microserviços para uma aplicação de abertura de conta bancária

JOÃO BRUNO MACEDO DA ROCHA
Setembro de 2024

POLITÉCNICO
DO PORTO

Proposal of a Microservices Architecture for a Bank Account Opening Application

João Bruno Macedo da Rocha

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Cybersecurity and Systems Administration

Supervisor: Dr. Paulo Gandra de Sousa

Porto, September 14, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 14, 2024

Dedicatory

"I would like to be alive to see you with as an engineer". These words from my grandfather stayed with me throughout my journey. Unfortunately, he passed away three years ago, but I know he is watching over me, proud of this accomplishment. So, this dissertation is dedicated to him with the deepest love and logging.

I would like to express my heartfelt gratitude to my parents and brother for giving me the opportunity to pursue this work, for believing in me, encouraging me, and supporting me during every step until now. To my girlfriend, thank you for your support, for pushing me forward, and for understanding the weekends spent at home, so I could work on this dissertation.

I am also grateful to my colleagues and professors, especially Dr.Paulo Gandra de Sousa, for their continuous support and guidance. Lastly, to my friends who believed in me and encouraged me throughout this journey.

Abstract

This dissertation aims to propose a solution to increase the performance of an existing bank account opening application developed in the low-code platform Appian. This application faces challenges related to the growing complexity of technologies and the limitations of monolithic architectures. The investigation suggests that it might be the right time to reengineer the application, implementing a microservices architecture to increase the performance, scalability, maintainability, and ease of deployment of the application.

The first phases of the investigation involved the understanding of the challenges associated with low-code applications and exploring migration challenges and strategies to improve application performance and facilitate proactive planning of the migration process. Afterward, it was done a study of the functional and non-functional requirements, as well as defining the use cases needed to replicate the functionalities of the original application. Based on this, a NoSQL data model was designed using MongoDB, and a microservices architecture was developed using the Spring Boot framework. When the development finished, evaluation tests were conducted to validate if the proposed solution actually can increase the performance and scalability of the real application.

The evaluation tests included unit and integration tests, confirming that the final solution was well-developed according to the business use cases. Additionally, load tests using Apache JMeter, made on the student's machine, assessed the application's performance and scalability, showing a 10% improvement when microservices were scaled with two instances each. This demonstrates a robust alternative to the limitations of monolithic architectures.

Keywords: Low-code, Microservices Architecture, Reengineering, Monolith to Microservices, Performance Optimization

Resumo

Esta dissertação visa propor uma solução para a melhoria do desempenho de uma aplicação de abertura de conta bancária existente, desenvolvida na plataforma low-code, Appian. Esta aplicação enfrenta desafios relacionados à complexidade crescente das tecnologias e às limitações das arquiteturas monolíticas. A investigação realizada sugere que poderá ser o momento ideal para fazer a reengenharia da aplicação e adotar uma arquitetura de microserviços como estratégia para aumentar o desempenho e a escalabilidade da aplicação.

As fases iniciais da investigação envolveram a compreensão dos desafios associados às aplicações de low-code e a exploração dos desafios e das estratégias de migração, visando melhorar o desempenho da aplicação e facilitar o planeamento proativo do processo de migração. Seguidamente, foi realizado um levantamento dos requisitos funcionais e não funcionais, bem como a definição dos casos de uso necessários para replicar as funcionalidades da aplicação original. Com base nisso, foi desenhado um modelo de dados NoSQL utilizando MongoDB e uma arquitetura de microserviços desenvolvida com a framework Spring Boot. Após a conclusão do desenvolvimento, foram conduzidos testes de avaliação para verificar se a solução proposta realmente pode melhorar o desempenho e a escalabilidade da aplicação real.

Os testes de avaliação incluíram testes unitários e de integração, confirmando que a solução final foi corretamente desenvolvida conforme os casos de uso do negócio. Além disso, os testes de carga, utilizado o Apache JMeter, foram efetuados na máquina do estudante e avaliaram o desempenho e a escalabilidade da aplicação, mostrando uma melhoria de 10% quando os microserviços foram escalados com duas instâncias cada, demonstrando uma alternativa robusta às limitações das arquiteturas monolíticas.

Palavras-chave: Low-code, Microserviços, Re-engenharia , Monolitos para Microserviços, Otimização da aplicação

Contents

List of Figures xiii

List of Tables xv

List of Acronyms xix

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem . . . 2
- 1.3 Objective . . . 2
- 1.4 Work Methodology . . . 2
- 1.5 Ethical Considerations . . . 3
- 1.6 Document Structure . . . 3

2 State of the Art 5
- 2.1 Literature Review Methodology . . . 5
- 2.2 Low-Code Applications . . . 7
  - 2.2.1 Performance Issues . . . 7
- 2.3 Microservices . . . 7
  - 2.3.1 Architecture . . . 8
    - Communication Styles . . . 8
    - Data Management . . . 10
    - Deployment Patterns . . . 10
  - 2.3.2 Migration Challenges . . . 11
  - 2.3.3 Migration Strategies . . . 11
- 2.4 Discussion . . . 13

3 Analysis and Design 15
- 3.1 Introduction . . . 15
- 3.2 Requirements . . . 15
- 3.3 Use Cases . . . 17
- 3.4 Domain Model . . . 19
- 3.5 Architecture . . . 20
  - 3.5.1 Components . . . 22
  - 3.5.2 Microservices Technical Description . . . 23
    - Account Service . . . 23
    - Customer Service . . . 23
    - Intervention Service . . . 24
    - Relation Service . . . 25
    - Document Service . . . 25
- 3.6 Data Model . . . 26

4 Implementation **31**
4.1 Microservices 31
4.2 API Gateway 34
4.3 Authentication 37
4.4 Message Broker 39

5 Experiment and Evaluation **43**
5.1 Application Tests 43
5.1.1 Unit Tests 44
Repository Layer 44
Service Layer 46
Controller Layer 48
5.1.2 Integration Tests 51
5.2 Performance and Load Testing 54

6 Conclusion **57**
6.1 Difficulties 57
6.2 Achievements 57
6.3 Future Work 58

Bibliography **59**

List of Figures

1.1 Example of the existing system's architecture 1

2.1 Monolith vs Microservices Architecture 8
2.2 Different styles of inter-microservice communication along with example implementing technologies 9

3.1 Use Cases Diagram 18
3.2 Domain Model Diagram 19
3.3 System's Architecture 21
3.4 Component Diagram 22
3.5 Account Service Data Model 26
3.6 Customer Service Data Model 27
3.7 Intervention Service Data Model 28
3.8 Relation Service Data Model 28
3.9 Document Service Data Model 29

4.1 Bitbucket Repositories 31
4.2 Example of Spring Initializr 32
4.3 Microservices organised using Multi-module project 33
4.4 Library organised using Multi-module project 33
4.5 Docker Container with ZooKeeper and Kafka 40

5.1 Test Pyramid [47] 44
5.2 Repositories Unit Tests 44
5.3 Account Repository Tests Passed 46
5.4 Account Service Tests Passed 48
5.5 Account Controller Tests Passed 51
5.6 Account Service Integration Tests Passed 54
5.7 Microservices running in Docker Containers 54

List of Tables

2.1 Source Evaluation Matrix 6

3.1 Functional Requirements 16
3.2 Non-Functional Requirements 16
3.3 System's use cases 17
3.4 Microservices Description 20

5.1 Performance and Load Experiments 55

Listings

4.1 Account Service Configurations (Properties). . . . 32
4.2 Docker Compose File with network specification (YML). . . . 34
4.3 Configurations file for API Gateway (1). . . . 34
4.4 Configurations file for API Gateway (2). . . . 35
4.5 Docker Compose File for API Gateway container (YML). . . . 36
4.6 SecurityConfig class (Java). . . . 37
4.7 JwtFilter class (Java). . . . 38
4.8 Docker Compose File for ZooKeeper and Kafka (YML). . . . 39
4.9 Kafka Configs (Properties). . . . 40
4.10 Kafka Producer (Java). . . . 40
4.11 Kafka Consumer (Java). . . . 41
5.1 AccountRepository Unit Tests (Java). . . . 45
5.2 AccountService Unit Tests (Java). . . . 47
5.3 AccountController Unit Tests (1) (Java). . . . 49
5.4 AccountController Unit Tests (2) (Java). . . . 50
5.5 Account Service Integration Tests (1) (Java). . . . 52
5.6 Account Service Integration Tests (2) (Java). . . . 53

List of Acronyms

ACID Atomicity, Consistency, Isolation, and Durability.
API Application Programming Interface.

CD Continuous Delivery.
CI Continuous Integration.

DDD Domain-Driven Design.
DTO Data Transfer Objects.

Faas Function as Service.

IT Information Technology.

LCPS Low-Code Platforms.

MSA Microservices Architecture.

Paas Platform as Service.

UML Unified Modelling Language.

Chapter 1

Introduction

This chapter is split into five sections, clarifying the project's context, the identified problem, the set objectives, the methods used for achieving them, and an overview of how this document is organised. The initial part aims to explain the ongoing project in order to guide the reader. Following that, it delves into the specific problem this dissertation addresses by highlighting its seriousness. The objectives section outlines what this project is trying to achieve, with clear and straightforward objectives. Next, the fourth section, describes the approach to pursue the goals set in the previous section. The last section, shows a preview of the upcoming content dissertation.

1.1 Context

This project centers around a bank account opening application that was developed as part of my current job for a client. Due to confidentiality considerations, more specific details are limited. However, it is possible to explain that the application is built using Appian, a low-code platform, with business rules and the front-end developed in Angular. The purpose of the application is to open bank accounts. Although, the application can be only accessed by the account manager, responsible for initiating the account-opening process and collecting client information. During this process, requests are being sent to the back-end, to Appian. Figure 1.1 shows an example of the existing architecture.

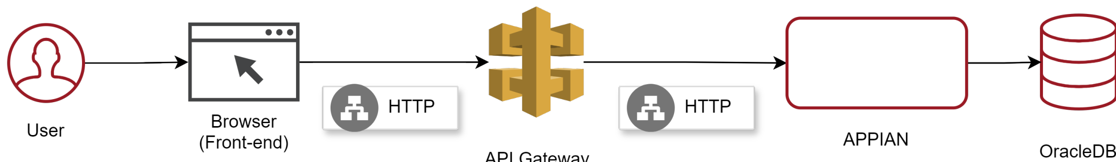

Figure 1.1: Example of the existing system's architecture

Since this application is developed using a low-code platforms and running in one instance, it can be considered as a monolithic application. These types of applications allow a much simpler deployment, simpler developer workflows, and monitoring, troubleshooting, and activities like end-to-end testing, as well as code reuse [1]. Additionally, Sam Newman and Martin Flower [1, 2], defend that should not start a new project with microservices, even if there are certainties about the complexity and the size of the application.

With more features or even with more complex business rules inside this monolith, maybe it is time to reconsider to move forward to microservices architecture (MSA). Microservices are a good choice when combining the concepts of information hiding and Domain-Driven Design (DDD)[1] [1]. Moreover, this architectural style provides benefits such as technology diversity, easy of deployment [1, 3] and scalability [1].

1.2 Problem

In the context of banking industry, this long-standing project faces a big problem, showing a decrease of performance in the application. Challenges related with complex business rules, underused code implemented in Appian, and bad programming practices are the reasons for the decrease. One concrete example of this problem is the prolonged duration it currently takes for the entire account opening process, hiding operational efficiency and impacting the overall user experience.

In summary, the decrease of the application performance comes from the complexity of the business rules and maintenance of the code. The application deals with complicated rule structures that might not match the straightforward goals of the opening account process. Therefore, there is a need to analyse the best option for solving these issues. MSA seems to be a suitable approach for such problems, because of its scalability, which is expected to improve application performance and impact the user experience positively.

1.3 Objective

The objective of this dissertation, is to propose a solution that splitting the application into partitions is a viable approach in order to improve the application in many ways like performance, scalability, maintainability and easier deployment.

So, to accomplish this goal, based on existing studies [1, 3], the adoption of MSA might be the most suitable option for this project. It is crucial to know the best strategies to split the monolith into microservices based on the business rules, and identify and propose the best approach how the project could have the refactoring process. However, comparing the results between the real application and the proposed solution are not possible due to confidentiality constraints, explained further in section 1.5.

1.4 Work Methodology

The development of this dissertation begins with understanding the existing issue in the bank account opening application that will further be validated with a literature review getting insights from similar approaches. With a clear understanding of the problem, the final objective is defined.

Anticipating challenges and getting knowledge about migration strategies is crucial, informed by insights from the literature. This helps to expect those challenges and selecting the suitable strategy. The next step involves analysing and designing the system, which is

[1] A concept whereby the fundamental problem/business domain is explicitly modelled in the software.

essential before developing the final solution. Defining the requirements and the use cases to replicate the real project helps to design the solution's architecture and data model. Once this is done, the application development and testing can be done. After that, experiments and evaluations are conducted to ensure the final solution effectively meets the objectives.

This methodological approach ensures a structured process to address the identified problem and enhance the application's performance, scalability, maintainability and easier deployment.

1.5 Ethical Considerations

This dissertation follows ethical considerations concerning data management and protection. Due to confidentiality and data privacy regulations, it is not possible to extract the real application or fully replicate all business rules in this project. Such actions could expose sensitive data and have significant implications for all stakeholders, including customers and the financial institution itself. Therefore, the final proposed solution will include known banking business rules and use fake data to represent the real project.

1.6 Document Structure

This section outlines the organization of the dissertation, providing details about the current chapters included in the document:

- **Chapter 1 - Introduction**: introduces the context of the project, discusses the problem at hand, and outlines the essential steps required to achieve the final goal.
- **Chapter 2 - State of the Art**: gives an explanation of what is exactly MSA, outlines the methodology used for the literature review, presents different concepts in order to answer to the research questions outlined in the literature review methodology.
- **Chapter 3 - Analysis and Design**: analyses the functional and non-functional requirements, along with the uses cases, to replicate the real project. It also presents the solution architecture and database design.
- **Chapter 4 - Implementation**: presents the implementation of the final solution, in order to meet the requirements, use cases, architecture, and data model described in Chapter 3.
- **Chapter 5 - Experiment and Evaluation**: presents the experiment conducted with the developed solution and evaluates key characteristics to demonstrate that the initial objectives were achieved.
- **Chapter 6 - Conclusion**: discusses the achievements made during the project, the challenges encountered, and potential future improvements.

Chapter 2

State of the Art

This state of the art, aims to explore various challenges found in the realm of banking sector using low-code platforms. The goal is to meticulously analyze existing literature, peer-reviewed publications, and industry practices, investigating the layers of complexity intrinsic to these systems, as performance considerations and the persistent challenges associated with maintaining and evolving applications.

2.1 Literature Review Methodology

This section outlines the methodology employed to conduct a literature review in order to solve the existing problems found on an account opening bank application. The first step involved formulating my research questions to address the identified challenges:

- **RQ1** : What key challenges have a significant impact on the performance of a low-code financial application?
- **RQ2** : How can the transformation from a monolithic architecture to microservices be effectively executed, and what are the existing strategies?

To answer these questions, researching in various platforms such as the Internet, Conference Papers, Scientific Articles, Books, or in repositories such as IEEE XPlore [4], Google Scholar [5], and ACM Digital Library [6] was an important step to find relevant materials. A source evaluation matrix, based on the presented in PREPD class [7], was employed to assess the selected literature, using four important criteria for evaluation:

- **Who** is the author? If it is a known author or if it has some studies on that field.
- **How** many times was cited?
- **When** was published?
- **What** is the relevance for the purpose challenge?

Table 2.1, illustrates the evaluation process for assigning a score from 1 to 5.

Source Evaluation Matrix					
Criteria	**1 (Low)**	**2**	**3**	**4**	**5 (High)**
Who is the author?	Author background is unknown	Some evidence of author works in the area, but very few	Evidence of some publications in the area by the author	Author has several published works in the area	Author is a known authority in the area
How many times was cited?	Very limited citations, indicating low recognition	Some citations, but it falls below the average for similar types of literature	Moderate number of citations, suggesting a reasonable level of recognition	Cited many times, indicating a high level of recognition and influence	Cited numerous times, signifying a very high level of recognition and influence
When was published?	Date is unknown or older than 20 years old	Old reference – between 10 and 20 years old	References if between 5 and 10 years old	Recent reference – 2 and 5 years old	Up-to-date source – published in last two years
What is the relevance for the purpose challenge?	Contents and arguments of little or no relevance for the task	Only of peripheral/little relevance for the task	Some content is relevant to task requirements	Several points made are of relevance to task	Contents and arguments closely match the needs

Table 2.1: Source Evaluation Matrix

It is important to acknowledge that, in some instances, one or more criteria may be ignored if there is a limited pool of studies to respond to a particular research question.

2.2 Low-Code Applications

The impact of low-code platforms on banking applications has outgrown significantly since the banks understood that the customers have been using more online services. According to a McKinsey report [7] says 89% of customers engage with at least one payment service daily and 54% of the respondents expressing preference for a bank-provided wallet, which means the banks need to modernize themselves. Low-code platforms offer a distinct advantage by facilitating the development and deployment of financial applications or services, allowing banks to relocate software developers to more challenging projects or even reduce their IT departments. Although, it is crucial to acknowledge that low-code platforms showed common bottlenecks in performance, which are critical considerations for the banking sector [8].

2.2.1 Performance Issues

Before explaining about what are the main issues in terms of performance in Low-Code applications, it is important to understand what is performance exactly. Performance is nothing less than everything that relates with efficiency, effectiveness, or success of an activity or an entire process. Performance is one of the most important metrics in software systems. A bad performance can negatively affect the software efficiency and the user experience. Appian company defends that a high number of process variants, and thus low standardization, often has a negative impact on process performance [9].

For a lower application sophistication with non-complex business logic, positively impact the adoption for Low-Code Platforms (LCPS) in terms of efficiency improvements, especially in cost and speed [10]. However, when leading with too-high application complexity, will exist challenges that have to be addressed, such as interoperability in terms of proprietary and closed-sources, lack of standards makes extensibility difficult, learning curves issues, and scalability, pose considerations for organizations with limited IT resources [11]. Understanding resource usage and configuration choices is essential for maintaining software efficiency in the banking domain, despite the fact that, the project is not using different platforms to handle the business logic, variations in bug patterns between commercial and open-source LCPS highlight potential differences in system-level issues and management practice, while the common performance issues often result from misconfigurations, with 49% being addressable through proper configurations [12]. Another existing study has shown the bugs are notably around 60% during design specification stages, showing performance degradation [8].

2.3 Microservices

Nowadays, in software development, MSA are more present than never, playing a crucial role offering agility, scalability, and flexibility. Based on a survey conducted by the International Data Corporation, it has been found that 89% of approximately three hundred respondents from North American enterprises are employing microservices, also it predicts that 90% of upcoming applications will be constructed using the microservice architecture [13, 14]. According to another online surveys, targeting Information Technology (IT) Professionals, identified performance/scaling issues and maintainability as the main drivers for adopting MSA [15, 16], that's why a lot of organizations are currently navigating or considering the sift to microservices. However, this transition presents migration challenges and strategies that will be discussed further.

Microservices are small, loosely coupled [17], and independently deployable services [1], which of them consisting in one or more subdomains, and owned by the team responsible for it [18]. Additionally, these services manage their own data, using the Database per Service pattern [17], and are technology-agnostic which means they do not need to share the same programming language [1]. Another characteristic of microservices is that they communicate with each other by using well-defined APIs or even a message-broker [17], that will be discussed in more detail later.

As Sam Newman said in [1], "*Microservices are an architecture choice that is focused on giving you many options for solving the problems you might face*".

2.3.1 Architecture

Microservices architecture is an architectural style that structs an entire application as a collection of the services explored previously. Martin Fowler defends that, should start a new project with a monolith architecture even if the application is expected to grow significantly [2], this means that, in the beginning of a project, the monolith it will be easier to develop and manage instead of an MSA. At some point, it is necessary to re-think and outgrow the monolithic architecture to an MSA in order to meet the needs of the business. The figure 2.1 illustrates the difference between these two architectures.

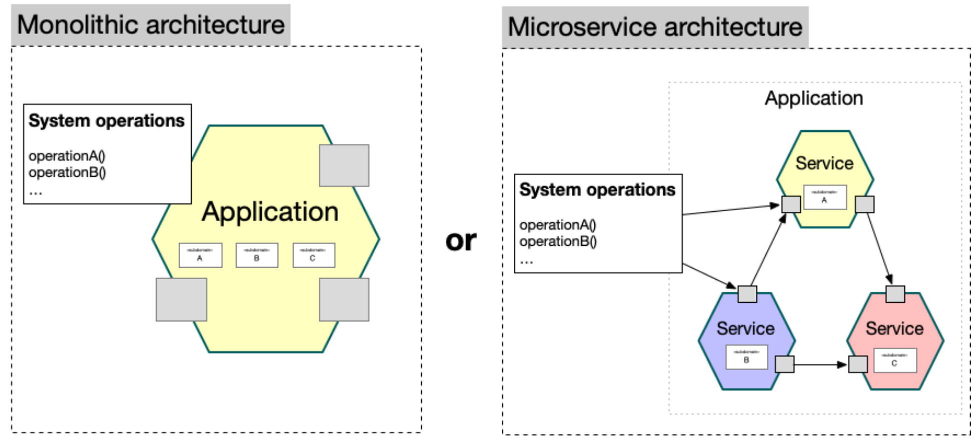

Figure 2.1: Monolith vs Microservices Architecture
[19]

This architecture involves considerations in terms of communication styles, data management and deployment patterns.

Communication Styles

Achieving effective communication between microservices is a challenge for many due to the fact the people choose technological approaches instead of considering the different types of communication. Figure 2.2 highlights synchronous and asynchronous communication mechanisms that allow microservices to communicate to each other. Synchronous communication involves a microservice making a call to another and waiting for a response, while asynchronous communication allows the microservice to continue processing without waiting for the call to be received [1].

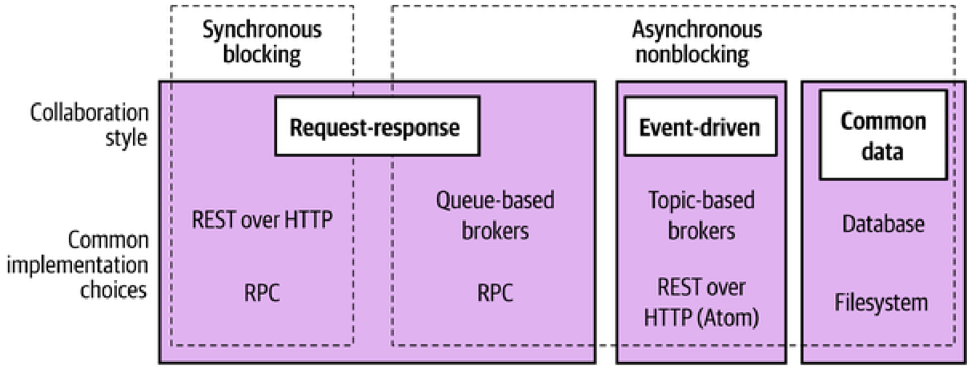

Figure 2.2: Different styles of inter-microservice communication along with example implementing technologies
[1]

Event-driven and Common data patterns are asynchronous styles, whereas the Request-response can be either synchronous and asynchronous depending on the technology used to effectively do the communication.

In this communication, the microservice sends a request to a downstream service asking it to do something and expects to receive a response with the result of the request. The difference between being a synchronous style or asynchronous one is when this type of pattern uses HTTP or a message-broker. On one hand, using HTPP means that when sending the request, a connection is open, and it is kept open until the downstream microservice responds, the problem with this type of communication is when the downstream microservice is not available at that moment which will block the user's work. On the other hand, using a message-broker, for instance RabbitMQ [20], means using the asynchronous style, instead of the message going directly to the downstream microservice sits in the message broker queue, then when the downstream microservice is free or available can handle that message and answers to the upstream microservice [1].

The Event-Driven communication consists of a microservice emitting an event[1] that may not be received by other microservices. The microservice which is emitting the event doesn't know the existence of the microservices that will use the event, neither use of it. [1].

The Common data, although is a collaboration asynchronous style, is the less seen as a communication style. In this pattern, the microservices collaborate via some shared data source.It is usually used when one microservice puts data into a defined location and the others can make use of that data [1].

Despite all of these patterns in terms of communication, the microservice architecture as a whole may have a mix of styles of collaboration, and this is the typically normal approach, depending on the needs of the project.

[1] Is a statement about something that has occurred, nearly always something that has happened inside the world of the microservices that is emitting the event. [1]

Data Management

Managing the data in a microservices architecture is very important to mitigate data consistency problems. For managing the data in a good way, Chris Richardson shows known patterns [21]:

- **Database per Service**: each service has its own private database. This type of pattern forces the services being loosely coupled, meaning they are developed, deployed and scaled independently. Additionally, this is a good approach when different services have different data storage requirements because for some services a relation database is the best choice [22].
- **Shared database**: services share a database. This type of pattern is a good way for developers reducing their operation time because it is much simpler than the previous one, also they can use straightforward Atomicity, Consistency, Isolation, and Durability (ACID) transactions to enforce data consistency [23].
- **Saga**: use sagas, which is a sequence of local transactions, to maintain data consistency across services. Once applied, the Database per Service pattern, each service will have its own database. In order to span multiple services, the Saga pattern is a good addition [24].

Deployment Patterns

Microservices should run in an isolated environment and ideally be deployed in a way that avoids downtime. To conquer this, the DevOps team should pick a deployment option to allow culture automation, and define infrastructure for microservices. Sam Newman, in their book [1], briefly summarized various deployment options such as:

- **Physical machine**: A microservice instance is deployed directly onto a physical machine with no virtualization.
- **Virtual Machine**: A microservice instance is deployed on to a virtual machine.
- **Container**: A microservice instance runs as a separate container on a virtual or physical machine. That container runtime may be managed by a container orchestration tool like Kubernetes [25].
- **Application container**: A microservice instance is run inside an application container that manages other application instances, typically on the same runtime.
- **Platform as Service(Paas)**: A more highly abstracted platform used to deploy instances, often abstracting away all concepts of the underlying servers used to run microservices.
- **Function as Service(Faas)**: an instance is deployed as one or more functions, which are run and managed by an underlying platform like AWS Lambda [26] or Azure Functions [27].

2.3.2 Migration Challenges

Migrating from a monolithic architecture to microservices is a complex process that will have a lot of challenges. According to the existing literature, the migration process should be divided into three phases: Analysis, Design and Execution. An online survey [15] revealed that IT Professionals found challenges in the Analysis phase, including undocumented functionalities, missing tests, insufficient database documentation, and problems with the old codebase and technologies. Similar challenges persisted in the Design phase, such as lack of documentation and efforts to reduce service coupling [15]. However, new challenges were found on this phase as well, including effort estimation overhead, DevOps infrastructure [16], encompassing architectural concerns, API versioning, and service contracts [28]. During the final phase, Execution, technical challenges like debugging microservices [29], breaking down the monolithic system, and difficulties in managing data consistency, microservices integration, programming language differences and the testing process [15], were more present than in the previous phases. Another case study [28] also highlighted challenges related to testing complexities and issues associated with distributed data stores.

Additionally, non-technical challenges like the division of business capabilities, the need for expert judgment in developing microservices, resource management, complex environment settings, difficulties in implementing transactions, and organizational adaptation [29, 30] are present in organizations leading to bad migration process or, worse, a bad developed application.

2.3.3 Migration Strategies

As mentioned in the previous subsection, the migration from monolithic architecture to MSA is a process that requires careful planning and execution. Sam Newman emphasizes the main decision of selecting the initial microservice for extraction, highlighting the importance of choosing one that aligns with the end-to-end goal and has significant impact on the final outcome [1]. Additionally, the decomposition process can start, by either splitting the code or the data in first [1]. If the code is divided in first place, which is the most common first step, delivers more short-term benefits. On the other hand, initiating the decomposition process by splitting the data is useful in situations in which the uncertainty remains whether the data can be separated cleanly. Furthermore, Newman introduced three key patterns: the Strangler Fig, Parallel Run and Feature Toggle patterns, each serving a distinct purpose in the decomposition process, already explained previously. [1].

The Strangler Fig pattern is the most used pattern for rewriting a system. This pattern describes the process of wrapping an old system with the new system over time, allowing the new one to take over more and more features of the old system incrementally. When a request arrives, if the functionality is implemented in MSA then the request is redirected to the microservices architecture logic, otherwise should go to monolith itself [1]. On the other side, Parallel Run pattern, it is a pattern used to make the that a new functionality is working well. For instance, both the monolithic implementation and the new microservice implementation of that of the functionality are running side by side, serving the same requests, and comparing the results between both architectural styles [1].The last one, the Feature Toggle pattern, is described as a mechanism that allows a feature to be switched off or on, or to switch between two different implementations of some functionality. This pattern is useful as part of a microservice migration. [1].

In addition, a case study [31] complements Newman's work, proposing a five-step migration approach:

- **Analysis of existing monolith application**: this step collects three important requirements, for monolith code decomposition, for communication establishment between microservices and for database adaption to microservice architecture.
- **Monolith code decomposition into microservices**: this step involves breaking down the code using methods like code-based, meaning that de application decomposition should be implemented based on code items. There's also the business domain-based approach, where the application is divided according to business domains, with each one corresponding to a microservice. Another method is storage-based, which suggests that all code related to specific storage should place in one microservice. To add to this, another case study [32] introduced a migration strategy based on business functionalities. This involves identifying business functionalities, analysing them to extract data like usage rates and other statistical information, and assigning these functionalities to microservices theoretically. If a business functionality is used a lot comparing to the others, it is assigned to a single microservice; if instead, two or more functionalities are used more or less the same number of times, and their scope is similar, they are all inserted into a single microservice.
- **Communication establishment between microservices**: the main goal here is to choose a communication technology and establish communication between microservices. There are multiple communication technologies that can do that including HTTP Rest, RabbitMQ [20], Kafka [33], gRPC [34], and GraphQL [35]. On one hand, if horizontal scalability is an important aspect, Kafka and RabbitMQ are the best candidates as they have built in cluster functionality. On the other hand, if latency and throughput are main criteria, then RabbitMQ and gRPC are the most suitable technologies. However, Kafka showed the best throughput results in the most loaded conditions.
- **Database adaptation to microservice architecture**: during the fourth step, the existing legacy monolith application database must be adapted to MSA. This approach can extract a database from a monolith application and transform it into a multimodel polyglot persistence, which is encapsulated as a microservice itself and exposes data access through application programming interface (API) allowing the use of the benefits of microservices such as agility and scalability. The encapsulation of a database into a microservice reduces the complexity and increases the performance.
- **Release and Deployment**: the final step aims to release and deploy extracted microservices and adapted database. It involves choosing and preparing an execution environment, setting up CI/CD pipelines for each microservice, and addressing aspects such as monitoring and logging. Once infrastructure is established, all microservices can be deployed into production environment.

2.4 Discussion

After a review of the existing literature, this section delves into the answers to the research questions introduced previously.

Firstly, answering to **RQ1**, low-code applications can significantly impact the performance of financial applications. LCPS offer accelerated development and cost benefits, but their impact on performance in banking applications requires a nuance approach. Considerations in terms of interoperability, extensibility, and scalability should be taken when thinking to use LCPS for a high sophistication and high complexity applications, which is the case of this banking application. However, migrating to microservices it is not a perfect world, during that migration, challenges can occur, leading to choose the best approach for its purpose.

That's why addressing **RQ2** is an important step for that. The migration process involves dealing with a range of challenges across system analysis, design and execution, related with technical and organizational complexities. Proper and continuous system maintenance can mitigate these challenges, enhancing the migration process and reducing its duration. Despite these challenges, adopting microservices offers advantages like improved scalability, flexibility, fault isolation, and quicker development and deployment cycles. Additionally, during this process, factors such as microservice selection, decomposition, communication, database adaptation and phased migration strategies are intended to be a part of a migration strategy. Real-world insights [36] highlight the diversity of approaches, emphasizing the need for tailored solutions based on the context, goals, and environment. There isn't the best strategy to migrate from monolithic architecture to MSA, rather, strategies exist that can be more applicable depending on specific instances.

Chapter 3

Analysis and Design

The goal of this chapter is to introduce the experiment solution by describing the main characteristics of this work. To avoid revealing all the business rules of the real project, the functional and non-functional requirements, and the use cases provided are similar examples to those in the real project. Additionally, this section will discuss the developed architecture, explain how and why it was chosen, and list the necessary technology stack to implement it.

3.1 Introduction

The proposed work is a banking application for account managers to create bank accounts for customers based on the real system developed using a low code platform, as described in the Introduction Chapter.

To develop the system to meet the requirements, use cases, and the architecture, which are going to be detailed further, the system shall be built with a microservices architecture style, with each service developed in Spring Boot, this is a choice based on the student's knowledge. To interact with microservices an API Gateway shall be used to enable communication between microservices via HTTP Protocol, and the communication between them is handled through JSON events using Apache Kafka. The database of each microservice should be in MongoDB for its scalability and performance capabilities, as well as it is an open-source database system. All of these technologies were selected because they are efficient and familiar to the student.

To interact with the system, Postman shall be used, as it is very useful and effective for testing HTTP connections and communication between services.

3.2 Requirements

The main objective of this section is to present the functional and non-functional requirements for the system to be developed. Both types of requirements were carefully defined to ensure that all necessary steps and features are included, facilitating a seamless with the actual project, and an efficient account creation experience for account managers. Tables 3.1 and 3.2 include the requirement name, its identifier, and its description to provide a comprehensive understanding of each one.

Firstly, the functional requirements aim to describe the principal functionalities of a system regarding the business and domain. These requirements are present in Table 3.1, detailing all the functionalities needed to support the entire bank account opening process.

Identifier	Functional Requirement	Description
FR1	Initiate bank account opening process	The system shall allow account managers to be able to initiate a new account opening process and get essential information about the primary account holder, such as personal details and identification documents.
FR2	Customer management	The system shall provide account managers to add or delete interventions and relationships, and update customer and account information. When there is a new customer, this one should be added to the account as an intervenient containing the relevant information about him/her, including his/her account intervention type.
FR3	Account management	The system shall provide options for account managers to select the type of account being opened, to select the appropriate card, to enable or disable online banking services.
FR4	Documents upload	The system shall provide account managers to upload and attach relevant documents to the account case.

Table 3.1: Functional Requirements

Finally, the non-functional requirements focus on how the system shall be developed [37], and they are described in Table 3.2, with all the technical functionalities needed to develop the system, and to meet the functional requirements detailed in Table 3.1.

Identifier	Non-Functional Requirement	Description
NFR1	Use microservice architecture style	The system shall follow the MSA style, and its principles.
NFR2	Use Data Base Per Service pattern	The system shall have every microservice having its own database.
NFR3	Event Streaming	The system shall have the microservices communicating to each other using Kafka events.
NFR4	Technology choice	The system shall have the microservices developed in Spring Boot and database must be in MongoDB.
NFR6	API Gateway use	The system shall call microservices' endpoints through an API gateway.
NFR7	Authorization Token	The system shall provide an authorization mechanism when calling the API gateway.

Table 3.2: Non-Functional Requirements

3.3 Use Cases

The use cases are important to capture and explain the requirements of a system, making sure all functional needs are captured and understood [38]. They also identify actors and their interactions with the system, although for this one, it is just present one actor. Based on the real project, the uses cases for this system describe all the operations that the account manager can perform in order to create the bank account for the customer.

The Table 3.3 shows all the use cases for the system, providing what the account manager can perform.

Identifier	**Related Requirement**	**Use Case**	**Description**
US1	**FR1**	Create case	Initiate the bank account opening process. This use case, shall also add intervention for the holder customer of the account.
US2	**FR3**	Add intervention	Add interventions in the account.
US3	**FR3**	Delete intervention	Delete interventions in the account.
US4	**FR2**	Add relation	Add customers relationships in the account.
US5	**FR2**	Delete relation	Delete customers relationships in the account.
US6	**FR2**	Update customers	Update customers information.
US7	**FR3**	Select type account	Selection of the account type.
US8	**FR3**	Select account card	Selection of the account card.
US9	**FR3**	Delete account card	It shall be possible to delete the card from the account.
US10	**FR3**	Select online banking	Selection of online banking.
US11	**FR4**	Upload document	Upload customers and accounts documents.
US12	**FR4**	Delete document	Delete customers and accounts documents.
US13	**FR3**	Move to next phase	Moves the account process into the next phase if all requirements are met.

Table 3.3: System's use cases

For a different point of view, the image 3.1, using the Unified Modelling Language (UML), shows a dynamic behaviour of the system, providing a better communication between all stakeholders.

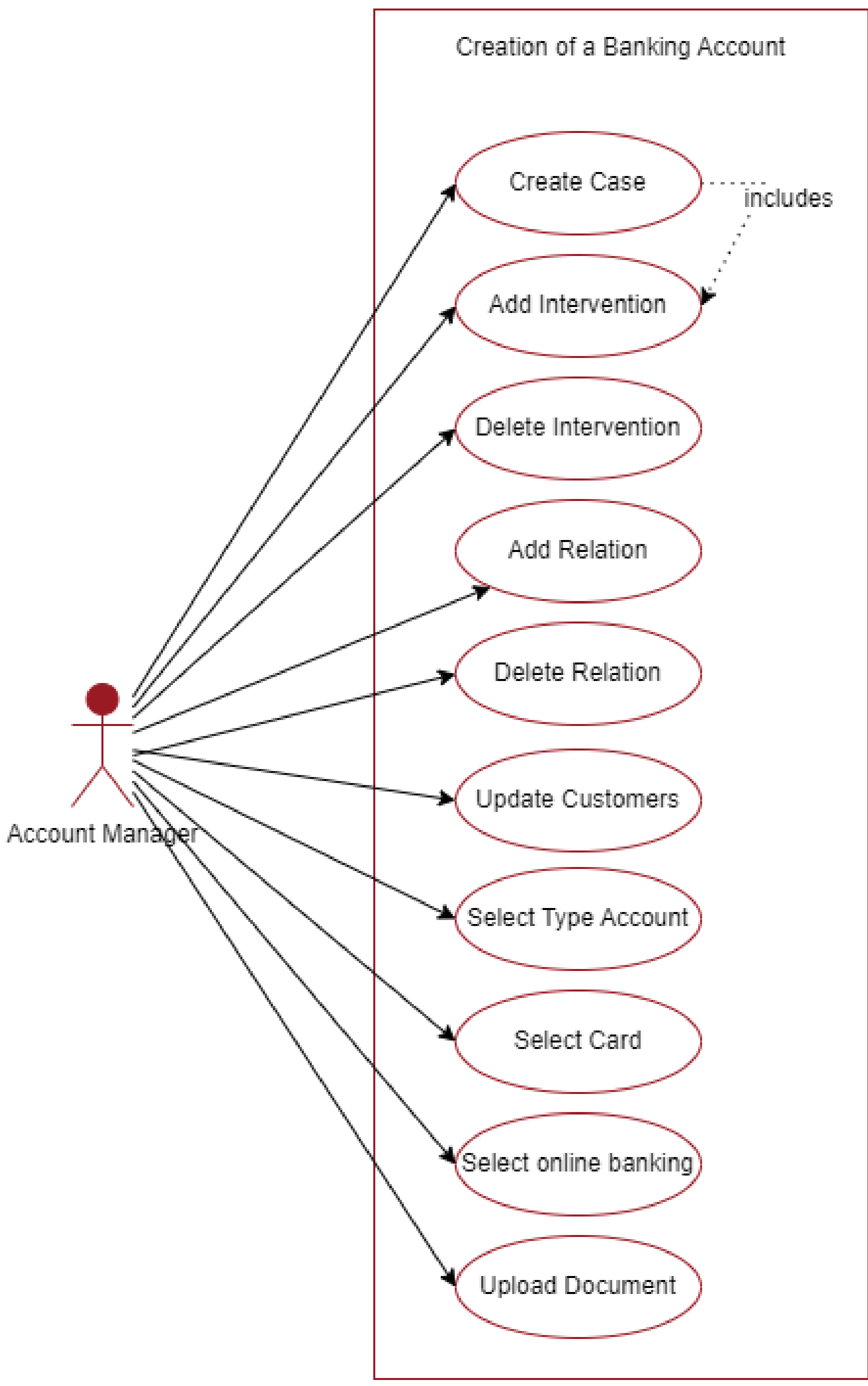

Figure 3.1: Use Cases Diagram

3.4 Domain Model

The Domain model is a structured visual representation of interconnected concepts or real-world objects that incorporates vocabulary, key concepts, behaviour, and relationships of all of its entities [39]. For that reason, five major business entities were identified as key components of the system:

- **Account**: Represents all the accounts.
- **Customer**: Represents all the customers.
- **Intervention**: Represents all the interventions that customers have within their accounts.
- **Relation**: Represents the relationships between customers.
- **Document**: Represents all the documents associated with accounts and customers.

The Figure 3.2 represents the domain model diagram with the respective relations between entities.

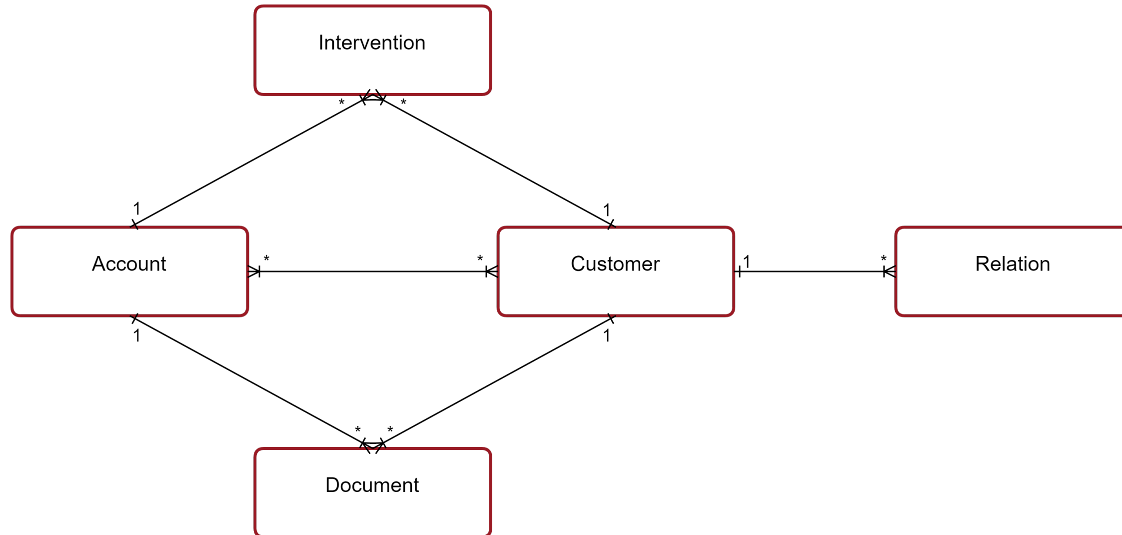

Figure 3.2: Domain Model Diagram

3.5 Architecture

This section aims to present the architecture that the system shall have, illustrating it with an overview of how the system behaves, completing, in more detail, using a component diagram following UML, and describing the microservices. Defining a software system's architecture plays a vital role in representing the system through an illustration or diagram that defines its components, its interactions, and its behaviour. Additionally, the software architecture is crucial for maintaining software quality, security, performance, scalability, and flexibility, all of which impact the system.

The choice of microservices shall follow the Domain-Driven Design (DDD) philosophy according to the business entities already explored in 3.4. Therefore, the following microservices were chosen:

Service	Description
AccountService	Shall handle all the business rules related to accounts. Additionally, this service will also be related with the customers, as it maintain a collection that stores information such as customer numbers and their validity status. Therefore, the service needs to update the customers' collection whenever a new customer is added or an existing customer is updated.
CustomerService	Shall hold the logic related to customers, such as when a new customer is created, when a customer is updated, or if there is an operation related to interventions or relations.
InterventionService	The service shall add, delete, and store customer interventions related to accounts.
RelationService	This service shall be responsible for adding, deleting, and storing relations between customers.
DocumentService	It shall contain the business rules for storing account or customer documents.

Table 3.4: Microservices Description

The Figure 3.3 provides an overview of how the system shall behave. The account manager ("User" in the figure) shall use a front-end application which communicates via HTTP with an API Gateway, which then redirects the HTTP request to the appropriate service. Each service shall use its own database in MongoDB using the Database per Service pattern described in Section 2.3.1, and publishes events to others services' topics and consumes events from its own topic in a Kafka Cluster. This will be explained in detail in the following subsections.

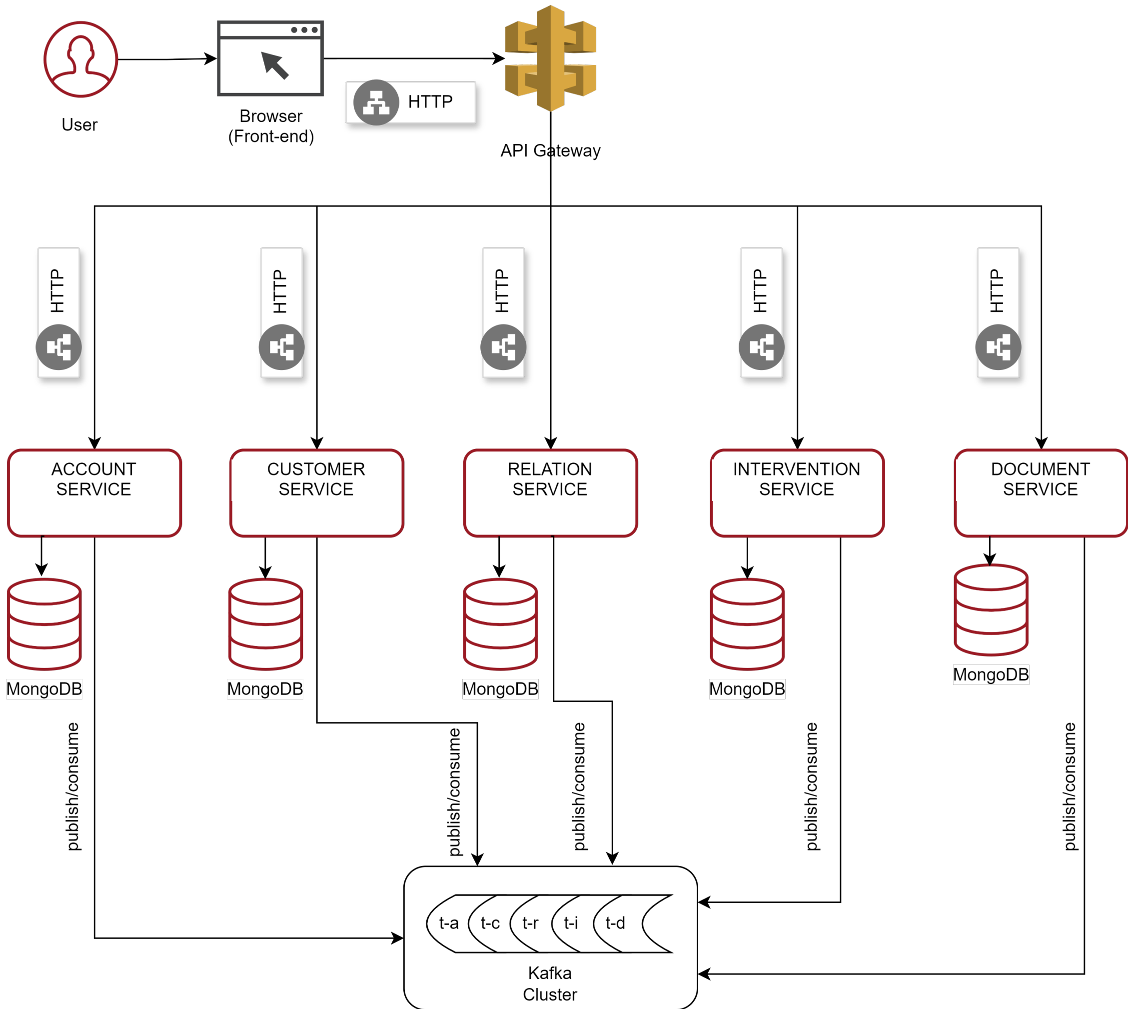

Figure 3.3: System's Architecture

3.5.1 Components

The architecture can also be described with component diagram using UML. The system's architecture shall consist in a group of five microservices in order to provide all the features necessary to represent the process of creating a banking account as in the real project. The diagram, shown in Figure 3.4 includes the API Gateway which shall serve as the entry point for communication with the microservices. Each microservice must have its own database and deployed on different servers.

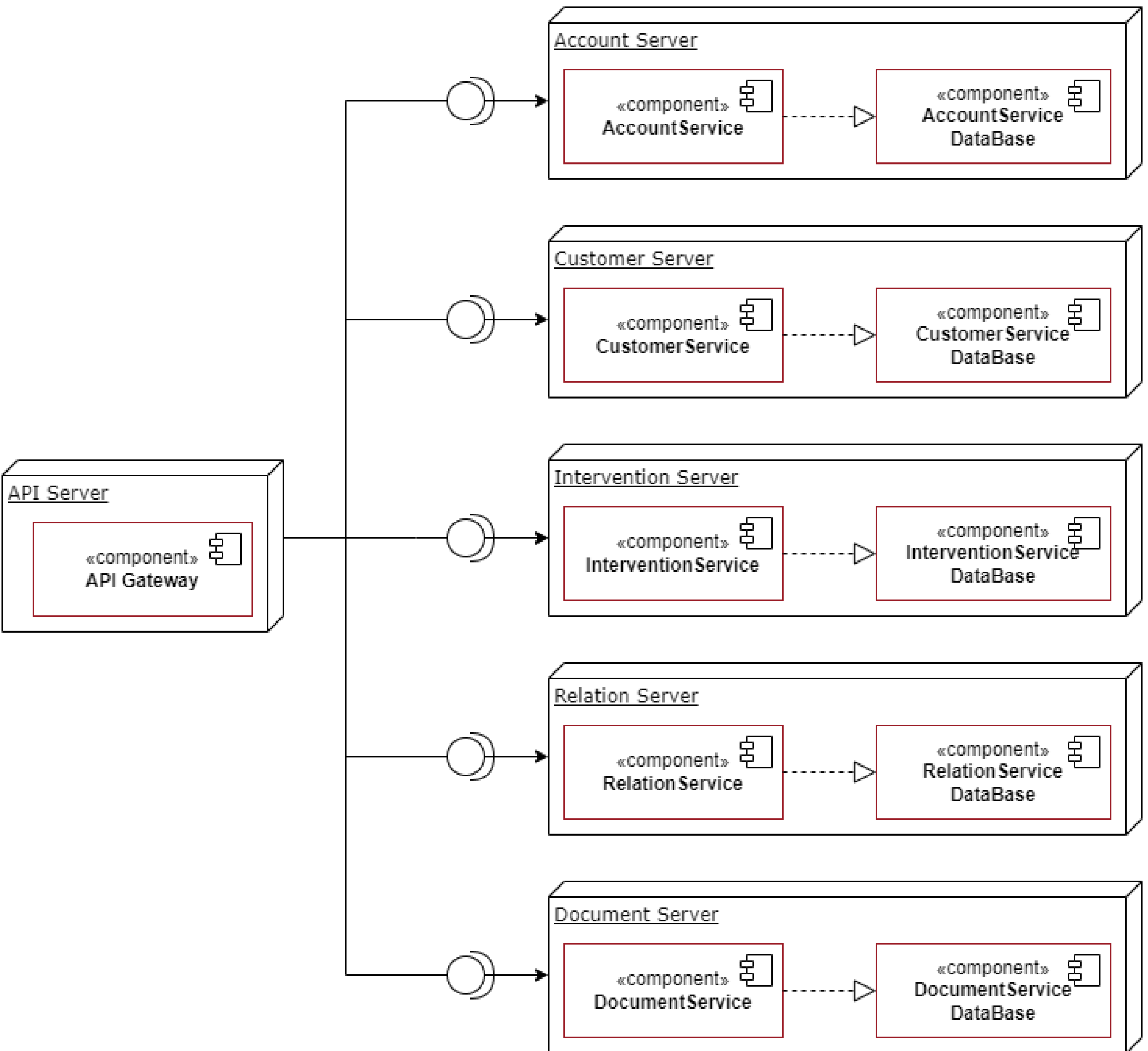

Figure 3.4: Component Diagram

3.5.2 Microservices Technical Description

This section describes the design of the microservices in detail, mentioning the events that each microservice shall send and consume, and the exposed HTTP endpoints.

Account Service

This microservice shall handle requests for almost all the use cases described in section 3.3, and shall produce and consume Kafka events. This microservice shall produce one type of event for the Customer Service topic called "customer-service-topic" in order to proceed with the **US1**. On the other hand, The Kafka Consumer of this microservice shall consume from the Account Service topic called "account-service-topic" and recognize the following three types of events based on their key:

- **UPDATE_CUSTOMER_REF**: This event shall be used to create or update the customer in the Customer Collection inside of Account Service database.
- **DOCS_UPLOAD**: This event shall store the information that the account's documents are valid. This type of event shall be sent by the Document Service.
- **ERROR_EVENT**: When there is an error in other microservices that this service needs to know, then it shall receive an event of this type.

Apart from receiving Kafka events, this microservice shall also have the following exposed endpoints to accomplish the use cases described in Table 3.3.

- **POST /accounts**: Shall start a new account opening process with the essential information about the primary account holder, such as personal details and identification documents. This corresponds to **US1**.
- **PUT /accounts/{accountNumber}**: Shall specify the type of account being opened, with details sent by the account manager. This corresponds to **US7**.
- **PUT /accounts/{accountNumber}/card**: Shall add a debit or credit card to the account. This corresponds to **US8**.
- **DELETE /accounts/{accountNumber}/card/{cardNumber}**: Shall delete a specific card from an account if it exists. This corresponds to **US9**.
- **PUT /accounts/{accountNumber}/netbanco**: Shall enable or disable online banking services for the account. This corresponds to **US10**.
- **PUT /accounts/{accountNumber}/moveNextPhase**: Shall move the account to a new phase if all requirements are met. This corresponds to **US13**.

Customer Service

The second microservice is the Customer Service, which shall handle all the operations directly related to customers. It shall produce events for all the other topics in the system's architecture: "intervention-service-topic", "document-service-topic", "account-service-topic" and "relation-service-topic". Its Kafka Consumer must consume the specific following events based on their key from its own topic called "customer-service-topic".

- **CREATE_ACCOUNT**: This event shall validate and save the first holder data in its own database when creating an account. This event is sent from the Account Service.

- **CARD_ACCOUNT**: When this event is consumed, the service shall store information that the customer requested a card.
- **NETBANCO_ACCOUNT**: This event must tell the service to store information that the customer wants online banking
- **DOCS_UPLOAD**: This event shall be responsible for storing information that the customer's documents are valid. This type of event mug be sent by the Document Service.
- **ERROR_EVENT**: When there is an error in the other microservices and this one needs to know, then it shall receive an event of this type.

Finally, this service also must handle requests from external sources using exposed endpoints. Therefore, it must contain the following endpoints, according to the use cases related to the customer in section 3.3.

- **PUT /customers/{customerNumber}**: This endpoint shall update customer information with more detailed data. This corresponds to **US6**.
- **PUT /intervention**: Must create a new intervention to an existing customer or add a new customer who must be an intervenient in the account. This endpoint shall be part of this microservice because it must handle basic customer data sent on the request body. This corresponds to **US2**.
- **PUT /relation**: This endpoint shall be created in this microservice as the same reason as the previous one, but, in this case, must create a new relation to an existing customer or add a new customer who must have a relation with another customer. This corresponds to **US4**.

Intervention Service

The next microservice is the Intervention Service, responsible for handling operations related to customer interventions within accounts. This service must produce Kafka events for the Account Service, Customer Service, and Document Service, which means producing to the topics "account-service-topic", "customer-service-topic", "document-service-topic" respectively. The microservice must also have a Kafka Consumer for consuming the following events from the "intervention-service-topic" where are the events published by other microservices in the system's architecture:

- **CREATE_ACCOUNT**: This event shall be sent by the Customer Service and be responsible for validating and storing the intervention related to the Account Opening operation.
- **ADD_INTERVENIENT**: Similar to the previous event, this event must validate and store the intervention for the Add intervention or Add intervenient operation. It must be sent by Customer Service.
- **ERROR_EVENT**: It is an error event that occurs when there is an issue in other microservices, which this service must need to be aware of.

The Intervention Service must handle only one HTTP request to delete an intervention, as it follows:

- **DELETE /interventions/{interventionId}**: This endpoint must delete an intervention that a customer has within an account. If the customer only has this intervention, which is to be deleted, an event must be sent to the Customer Service to remove the customer from the account. This corresponds to **US3** in table 3.3.

Relation Service

This microservice manages operations for adding and deleting relations between customers. It must have a Kafka Producer for producing events to other topics available in the system's architecture, in this case, this microservice produces events for the "account-service-topic", "customer-service-topic", "document-service-topic". On the other hand, the microservice must also have a Kafka Consumer to consume events from the topic "relation-service-topic" and perform certain logic based on the event key:

- **ADD_REL**: Sent by the Customer Service, this event must validate and store the relation between customers for the Add Relation operation.

This service must also handle HTTP requests however it only has one exposed endpoint, similar to the Intervention Service, but in this case, it is to delete a relation:

- **DELETE /relations/{relationId}**: It shall delete a relation between customers. If it is the only relation the customer has, and it is deleted, an event must be sent to the Customer Service in order to remove the customer from the account. This corresponds to **US5** in table 3.3.

Document Service

The last microservice in the system's architecture is the Document Service, which handles the logic for uploading and deleting account and customer documents. This service shall produce events for three different topics: "account-service-topic", "customer-service-topic", "intervention-service-topic". Apart from producing events, it must consume only the following type of event, published to its own topic "document-service-topic" by the Customer Service, in order to accomplish the create account operation.

- **CREATE_ACCOUNT**: After the Customer Service validates the primary account holder's details, it sends this event to the Document Service, which then must validate and store the customer's document.

In addition to consuming and producing events, this microservice must also receive HTTP requests related to uploading and deleting documents, either for the account or for customers, depending on the request body:

- **PUT /documents**: It must validate and store the document for the account or the customer. If a document type already exists, it must replace it with the new document. This corresponds to **US11** in table 3.3.
- **DELETE /documents**: Shall delete a document from the account or from the customer. If the document is deleted, an event must be sent to the Account Service or the Customer Service to inform them that the account or customer no longer has the required documents. This corresponds to **US12** in table 3.3.

3.6 Data Model

The Data models usually are built around business needs and illustrate the types of data used and stored within the system, the relationships between them, and its formats and attributes [40]. Therefore, since the business has five different entities and the microservices shall use the Database per Service pattern, the system will contain five different data models.

The Figure 3.5 represents the data model for the account service and its entities and relationships. The identified entities and relationships are also described as follows:

- **accounts**: Represents the main entity of the service. It stores the information of the account.
- **cards**: Represents the entity of the cards associated within the account.
- **customers**: Represents the relationship between the account and the customers.

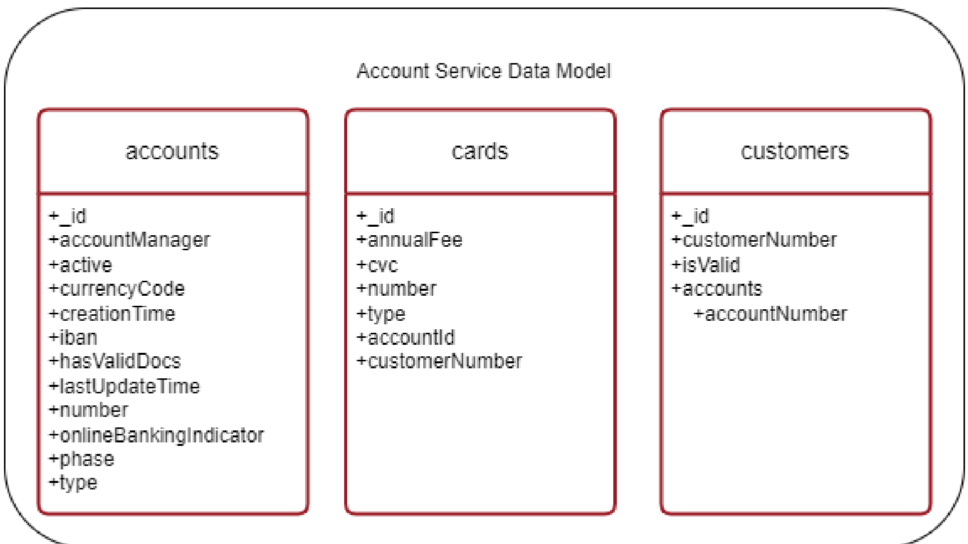

Figure 3.5: Account Service Data Model

The Figure 3.6 shows the data model for the customer service, including its entities and their relationships, which are described below:

- **customers**: Represents the information about the customers, also it has the relationship between the accounts, addresses, and contacts.
- **addresses**: Represents the entity of the addresses associated within the customer.
- **contacts**: Represents contacts of the customer.

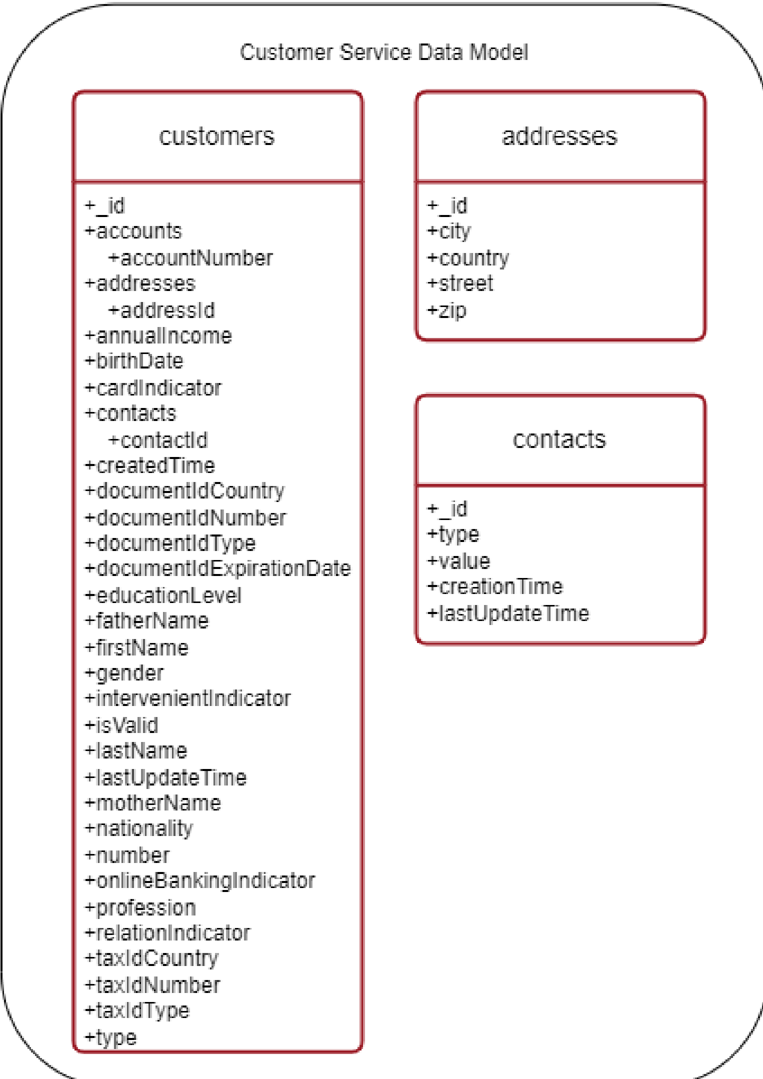

Figure 3.6: Customer Service Data Model

The Figure 3.7 represents the data model for the intervention service, with all the entities and their relationships, and which details explained below:

- **interventions**: Represents the interventions of a customer in an account.

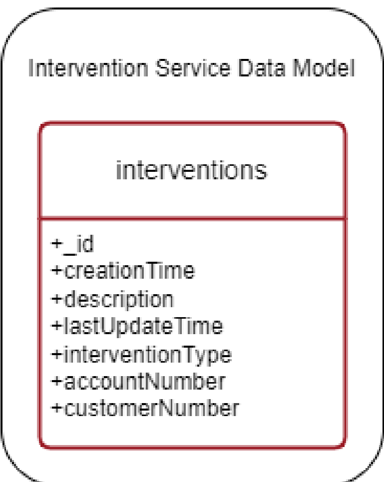

Figure 3.7: Intervention Service Data Model

Figure 3.8 displays the data model for the relation service, containing the entities and relationships. These are explained below:

- **relations**: Represents the main entity of the relation service with the relations between customers.

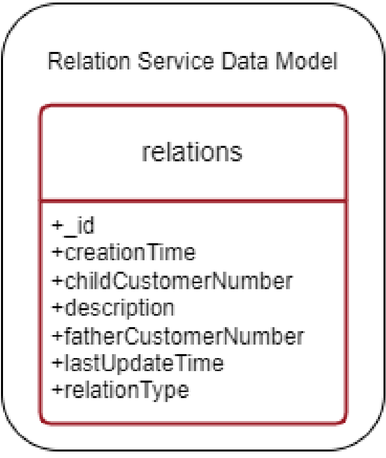

Figure 3.8: Relation Service Data Model

The data model for the document service is illustrated in Figure 3.9, showing the entities and their relationships, described as follows:

- **documents**: Represents the documents within the account or customer.

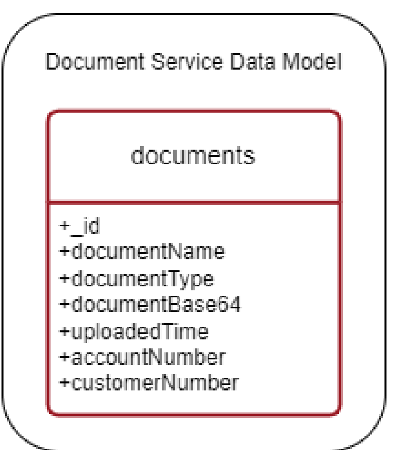

Figure 3.9: Document Service Data Model

Chapter 4

Implementation

This chapter provides an overview of how the system operates and explains the implementation of the microservices, the API Gateway, and the Message Broker.

4.1 Microservices

The solution is based on an MCS architecture style as described in the previous chapters of this document. To meet the requirements, use cases, architecture, and data model, it is necessary to prepare the system before starting to write some code.

Firstly, Bitbucket repositories are created for each microservice and the library. The library, is a Spring Boot application that contains common code to avoid duplication, and it can be used by the microservices. The Figure 4.1 shows the repositories in the student's Bitbucket account.

João Rocha / Projects / TESE

Repositories

Add repositories

Name	Size	Last updated	Builds
Onboarding_Account_Service	1.2 MB	2024-07-15	
Onboarding_Commons	1.4 MB	2024-07-15	
Onboarding_Customer_Service	1.2 MB	2024-07-15	
Onboarding_Document_Service	948.3 KB	2024-07-14	
Onboarding_Intervention_Service	948.6 KB	2024-07-14	
Onboarding_Relation_Service	898.6 KB	2024-07-14	

Figure 4.1: Bitbucket Repositories

Since each Spring Boot application has its own repository, it is the time to create them, using the Spring Initializr [41], a very useful tool that helps to create Spring applications. As shown in Figure 4.2 this tool allows the selection of the Language, Spring Boot version, project metadata, and Java version. Additionally, it provides the option to add Spring

dependencies. In this case, four dependencies are imported: Spring Data MongoDB, Spring for Apache Kafka, Lombok, and Spring Web.

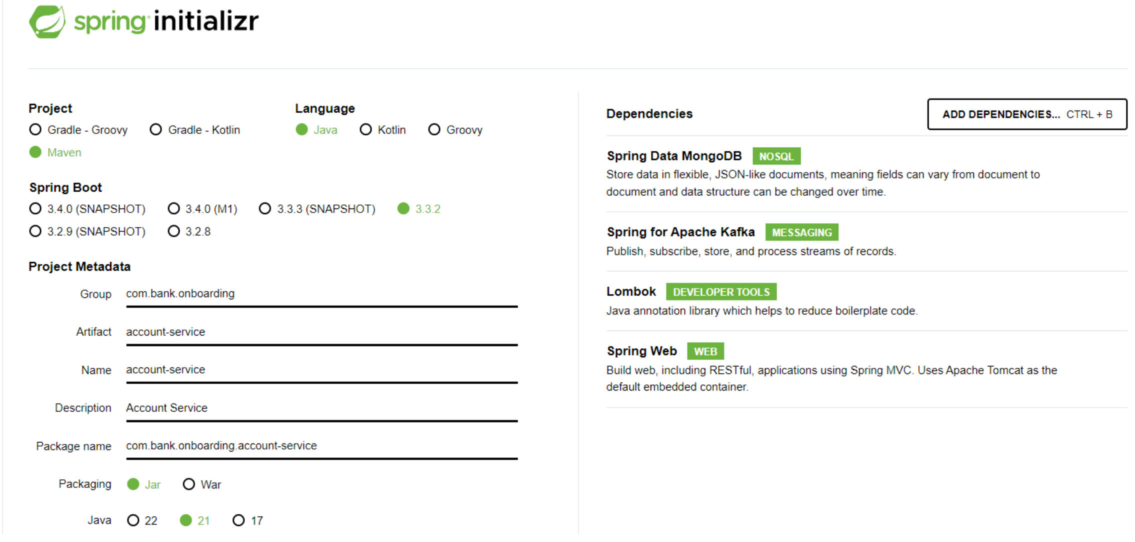

Figure 4.2: Example of Spring Initializr

To organize and improve each application, their configurations are placed in a specific properties file, as shown in Listing 4.1, using some environment variables to prevent exposure.

```
1  server.port=${SERVER_PORT}
2  spring.application.name=Account Service
3  spring.data.mongodb.database=${DB_NAME}
4  spring.data.mongodb.uri=mongodb+srv://${DB_USER}:${DB_PASSWORD}@cluster0
       .cr37uol.mongodb.net/${DB_NAME}
5  spring.kafka.bootstrap-servers=localhost:29092
6  spring.kafka.producer.value-serializer=org.apache.kafka.common.
       serialization.StringSerializer
7  spring.kafka.producer.key-serializer=org.apache.kafka.common.
       serialization.StringSerializer
8  spring.kafka.producer.customer.topic-name=customer-service-topic
9  spring.kafka.consumer.topic-name=account-service-topic
10 spring.kafka.consumer.group-id=account-consumer-group
11 jwt.secret.key=${JWT_SECRET_KEY}
12 bank.onboarding.client.id=${BANK_CLIENT_ID}
```

Listing 4.1: Account Service Configurations (Properties).

The applications are organised following a Multi-Module Project structure [42], which involves having a project with multiple modules aggregated under a main POM file, typically located in the project's root directory.

The microservices are structured into three modules: boot, services, and web, as shown in the following figure 4.3. The boot module has the Main method and essential Spring Boot annotations to initialize the service with all necessary resources. This module also includes the "application.properties" file with configurations and the Unit and Integrations Tests, which will be discussed later in the Chapter 5. The services module contains the "Kafka Consumer.java" class with the configuration to consume from a specific topic, and

it also contains all the logic to handle various use cases, whether it comes from an HTTP request or a Kafka event. Lastly, the web module exposes the endpoints which require specific headers and response status, described in 4.2. The request body of each endpoint is always validated using the Spring Validator annotations.

boot [boot (1) (com.bank.onboarding.accountservice)]
services [services (1) (com.bank.onboarding.accountservice)]
web [web (1) (com.bank.onboarding.accountservice)]

Figure 4.3: Microservices organised using Multi-module project

Similarly, the library also follows a Multi-module Project structure, however it is organised into four modules: boot, persistence, utils, web, as illustrated in 4.4. The boot module contains, components similar to those in the microservices. The persistence module includes services with public methods to use the repository interfaces, which extends the "MongoRepository.java" interface for performing database operations. This module also contains the database models for each repository and enums used to validate database operations. The utils module provides utility classes to be used by all microservices like "Kafka Producer.java" class, "EventSerializer.java" class, and other helper methods. Finally, the web module contains the Data Transfer Objects (DTO) used for the endpoint requests and handles the implementation of authentication configurations, which will be discussed later in this chapter.

boot
persistence
utils
web

Figure 4.4: Library organised using Multi-module project

Moreover, all the microservices in the system's architecture are deployed into a Docker Container using a "docker-compose.yml" file. Since the Docker Containers are designed to be isolated, a custom network was created using the command "docker network create -d onboardingbank". This network configuration was specified in the "docker-compose.yml" file to enable communication between the containers. Listing 4.2 provides an example of the "docker-compose.yml" file used for the Account Service.

```yaml
services:
  account-service:
    environment:
      BANK_CLIENT_ID: ${BANK_CLIENT_ID}
      DB_USER: ${DB_USER}
      DB_NAME: ${DB_NAME}
      DB_PASSWORD: ${DB_PASSWORD}
      JWT_SECRET_KEY: ${JWT_SECRET_KEY}
      KAFKA_SERVER: ${KAFKA_SERVER}
    build: .
    ports:
      - "9001-9050:8080"
    deploy:
      replicas: 2
    networks:
      - onboardingbank
networks:
  onboardingbank:
    external: true
```

Listing 4.2: Docker Compose File with network specification (YML).

4.2 API Gateway

An API Gateway is a fully managed service which sits between the frontend and the backend, acting as a reverse proxy. It accepts HTTP calls, aggregates all the back-end services needed to fulfil them, and returns the right response. Typically, API gateways handle user authentication, rate limiting, and provide statistics for a system of API services [43]. However, for this academic project, the API gateway is only used for user authorization. To implement an API Gateway for handling requests in this project, Nginx was chosen due to its robust reverse proxy capabilities. The first step in this implementation was creating the “default.conf” file, as shown in listings 4.3 and 4.4, which contains the necessary configurations to perform reverse proxy operations and route requests to the appropriate service.

```
upstream accountService{
    server account-service:8080;
}

upstream customerService{
    server customer-service:8080;
}

upstream interventionService{
    server intervention-service:8080;
}

upstream relationService{
    server relation-service:8080;
}

upstream documentService{
    server document-service:8080;
}
```

Listing 4.3: Configurations file for API Gateway (1).

```
server {
    listen       80;
    listen  [::]:80;
    server_name  localhost;

    #access_log  /var/log/nginx/host.access.log  main;

    location ~ ^/account/([^/]+)/moveNextPhase {
            proxy_pass http://accountService/accounts/$1/moveNextPhase;
    }

    location /account {
       proxy_pass http://accountService/accounts;
    }

    location /customer/intervention {
            proxy_pass http://customerService/customers/intervention;
    }

    location /customer/relation {
            proxy_pass http://customerService/customers/relation;
    }

    location ~ ^/customer/([^/]+) {
           proxy_pass http://customerService/customers/$1;
    }

    location /customer {
           proxy_pass http://customerService/customers;
    }

    location /intervention {
           proxy_pass http://interventionService/interventions;
    }

    location /relation {
           proxy_pass http://relationService/relations;
    }

    location /document {
           proxy_pass http://documentService/documents;
    }

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root   /usr/share/nginx/html;
    }
```

Listing 4.4: Configurations file for API Gateway (2).

To make use of this configuration file, a "docker-compose.yml" file was created within the commons library repository. This file defines a container using the Nginx image from Docker Hub, importing the configuration file into the container, as shown in listing 4.8. With these configurations, Nginx routes the HTTP requests to the appropriate service, sending the request headers and body.

```
apigateway:
  image: nginx
  ports:
    - 80:80
  volumes:
    - ./default-api-gateway.conf:/etc/nginx/conf.d/default.conf
  networks:
    - onboardingbank
```

Listing 4.5: Docker Compose File for API Gateway container (YML).

When the service receives a request, then verifies if it contains a valid JWT Token and a custom header called "X-Onboarding-Client-Id", which is a UUID key, designed to enhance the security of each request. The implementation of this is described on the following section.

4.3 Authentication

To implement the authentication, a configuration class was created to manipulate the "SecurityFilterChain" from Spring Security by adding a filter before authorizing any request. This class, detailed in listing 4.6, includes methods for generating a valid JWT token for testing purposes and public methods used by the "JWTFilter.java" class filter, which is described further. These JWT tokens are essential for ensuring user authentication, like account managers, can access the exposed endpoints in each microservice.

```java
@Configuration
@EnableWebSecurity
@Slf4j
@RequiredArgsConstructor
public class SecurityConfig {

    @Value("${jwt.secret.key}")
    private String secretKey;

    private static final String ONBOARDING_USER_NAME = "onboarding-user-admin";

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
        JwtFilter jwtFilter = new JwtFilter(this);
        return httpSecurity.csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(auth-> auth
                        .anyRequest().authenticated())
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)
                .build();
    }

    @PostConstruct
    public String generateJWToken(){

        Map<String, Object> claims = new HashMap<>();

        String jwts =  Jwts.builder()
                .claims(claims)
                .subject(ONBOARDING_USER_NAME)
                .issuedAt(new Date(System.currentTimeMillis()))
                .expiration(new Date(System.currentTimeMillis() + 1000*60*3))
                .signWith(getKey()).compact();

        log.info("Token generated: {}", jwts);

        return jwts;
    }

    public String getUsername(String token) {return extractClaim(token, Claims::getSubject);}
```

Listing 4.6: SecurityConfig class (Java).

The “JWTFilter.java” class, shown in listings 4.7, extracts the token from the Authorization header, removing the “Bearer " prefix. It then checks if the username is valid and if the token has not expired. If these conditions are met, the Authentication is set in Spring’s “SecurityContext”, allowing the request to proceed. Otherwise, a 403 Forbidden HTTP status is returned.

```java
@Component
@RequiredArgsConstructor
public class JwtFilter extends OncePerRequestFilter {

    private final SecurityConfig securityConfig;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
   HttpServletResponse response, FilterChain filterChain) throws
   ServletException, IOException {
        String authHeader = request.getHeader(HttpHeaders.AUTHORIZATION)
   ;
        String token = null;
        String userName =  null;

        if(authHeader != null && authHeader.startsWith("Bearer ")){
            token = authHeader.substring(7);
            userName = securityConfig.getUsername(token);
        }

        if(userName !=  null && SecurityContextHolder.getContext().
   getAuthentication() == null && securityConfig.validateToken(token,
   userName)){

            List<GrantedAuthority> authorities = new ArrayList<>();
            authorities.add(new SimpleGrantedAuthority("ROLE_USER_ADMIN"
   ));

            UsernamePasswordAuthenticationToken authenticationToken =
   new UsernamePasswordAuthenticationToken(userName, null, authorities);
            authenticationToken.setDetails(new
   WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(
   authenticationToken);
        }

        filterChain.doFilter(request, response);
    }
}
```

Listing 4.7: JwtFilter class (Java).

This task was particularly challenging as the student had no prior experience on developing such security configurations using the Spring Boot framework, which required significant research and learning to ensure that the security of HTTP requests was robustly implemented. However, the student have passes successfully the implementation of this mechanism.

4.4 Message Broker

To enable communication between microservices, a message broker is required. In this case, Apache Kafka [33] was selected. This choice was made because the student is familiar with Kafka, and it offers several advantages for this project. Kafka is capable of processing data multiple times based on a retention period, and it supports real-time event handling, which is crucial for microservices to update their own databases. This makes Kafka a more suitable option compared to RabbitMQ [44], another message broker that the student is familiar with.

To implement this setup, a Docker Compose file was used, as shown in Listing 4.8. This file, provided by Tapan Avasthi in [45], contains the configuration needed to start Docker containers for ZooKeeper and Kafka in a local environment. In this setup, Kafka uses ZooKeeper to store and manage cluster metadata, which includes topic information, partition details, and consumer group information.

```yaml
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.4
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181
    networks:
      - onboardingbank
  kafka:
    image: confluentinc/cp-kafka:7.4.4
    depends_on:
      - zookeeper
    ports:
      - 29092:29092
    networks:
      - onboardingbank
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

Listing 4.8: Docker Compose File for ZooKeeper and Kafka (YML).

To start these containers, the command "docker-compose up -d" is used in the command line. The Docker Desktop application then shows running containers for both Kafka and ZooKeeper, as seen in Figure 4.5.

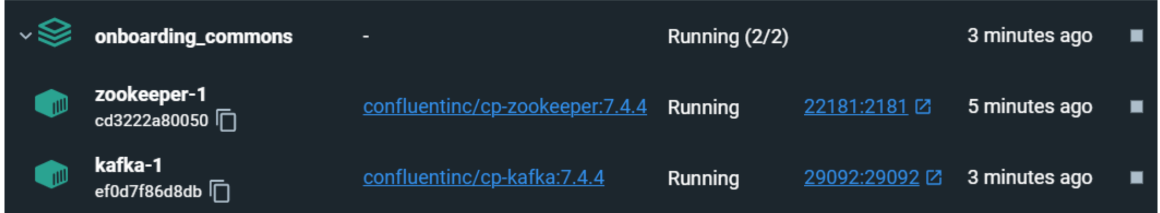

Figure 4.5: Docker Container with ZooKeeper and Kafka

With ZooKeeper and Kafka running, the next step is to add configurations to the application.properties file in each service and create the Kafka Consumers and Kafka Producers for the services that will use Kafka. The necessary configurations, shown in Listing 4.9, enable the services to send events for specific topics and consume events from its own topic in a Kafka Cluster running on a specific port.

```
spring.kafka.bootstrap-servers=localhost:29092
spring.kafka.producer.value-serializer=org.apache.kafka.common.
    serialization.StringSerializer
spring.kafka.producer.key-serializer=org.apache.kafka.common.
    serialization.StringSerializer
spring.kafka.producer.customer.topic-name=customer-service-topic
spring.kafka.consumer.topic-name=account-service-topic
spring.kafka.consumer.group-id=account-consumer-group
```

Listing 4.9: Kafka Configs (Properties).

Additionally, the Kafka Producer was developed inside the commons library, as it will be used by all services. This Java class, shown in Figure 4.10, includes a public method responsible for serializing an event as a String and sending it to a specific topic, as specified by a parameter. This is done using the Kafka Template provided by the "Spring for Apache Kafka" dependency, as mentioned in Figure 4.2.

```java
public class KafkaProducer {

    private final EventSeDeserializer eventSeDeserializer;

    private final KafkaTemplate<String, String> kafkaTemplate;

    public void sendEvent(String topicName, OperationType operationType,
     Object sendingEvent) {
        String operationTypeString = "";
        if(operationType != null) operationTypeString = operationType.
    name();
        String eventSerialized = eventSeDeserializer.serialize(
    sendingEvent);
        log.info("Sent message to {} for operation type {} with event {}
    ", topicName, operationTypeString, eventSerialized);

        kafkaTemplate.send(topicName, operationTypeString,
    eventSerialized);
    }
}
```

Listing 4.10: Kafka Producer (Java).

Finally, a Kafka Consumer must be developed for each service. Figure 4.11 shows the Kafka Consumer Java class for the Account Service as an example. This class consumes events from a specific topic, "account-service-topic", using a consumer group, "account-consumer-group". It then performs specific logic based on the key provided by the producer in the event.

```java
@Service
@RequiredArgsConstructor
@Slf4j
public class KafkaConsumer {

    private final EventSeDeserializer eventSeDeserializer;
    private final AccountService accountService;
    private final CustomerRefRepoService customerRefRepoService;

    @KafkaListener(topics = "${spring.kafka.consumer.topic-name}", groupId = "${spring.kafka.consumer.group-id}")
    public void consumeEvent(ConsumerRecord event){
        String eventValue = event.value().toString();
        switch (event.key().toString()) {
            case "UPDATE_CUSTOMER_REF" -> {
                CustomerRefDTO customerRefDTO = (CustomerRefDTO) eventSeDeserializer.deserialize(eventValue, CustomerRefDTO.class);
                log.info("Event received to update Customer Ref with number {}", customerRefDTO.getCustomerNumber());
                CustomerRef customerRef = customerRefRepoService.findCustomerRefByCustomerNumber(customerRefDTO.getCustomerNumber());
                customerRef.setValid(customerRefDTO.getIsValid());
                customerRef.setAccounts(customerRefDTO.getAccounts());
                customerRefRepoService.saveCustomerRefDB(customerRef);
            }
            case "DOCS_UPLOAD" -> {
                DocUploadEvent docUploadEvent = (DocUploadEvent) eventSeDeserializer.deserialize(eventValue, DocUploadEvent.class);
                log.info("Event received to validate account docs with number {}", docUploadEvent.getAccountNumber());
                accountService.updateDocsValidOrNotValid(docUploadEvent);
            }
            default -> {
                ErrorEvent errorEvent = (ErrorEvent) eventSeDeserializer.deserialize(eventValue, ErrorEvent.class);
                log.info("Error event {} received for account number {}", errorEvent, Optional.ofNullable(errorEvent.getAccountRefDTO()).map(AccountRefDTO::getAccountNumber).orElse(""));
                accountService.handleErrorEvent(errorEvent);
            }
        }
    }
```

Listing 4.11: Kafka Consumer (Java).

Chapter 5

Experiment and Evaluation

Unfortunately, due to confidentiality constraints, it is not possible to compare the results of this solution with those of the real application, however to demonstrate that the objectives outlined in Section 1.3 can be achieved with the proposed solution, a series of experiments will be conducted. Therefore, this chapter provides a detailed description of that experiments conducted and their subsequent evaluation. It is divided into two sections: the Application Tests which describes the unit and integration tests of the application in order to validate what was implemented and the results of the tests; the Performance and Load Testing section which outlines the approach used for assess performance, loading and scalability, and provides the results of each test.

However, experiments for maintainability and ease of deployment will not be performed. Maintainability can be supported through the use of the commons library, as described in Chapter 4, which contains reusable code accessible to all microservices if needed. The ease of deployment can be supported by the state of the art, present in Chapter 2, which highlights that one of the benefits of MSA is that the microservices are isolated and independent, allowing for independent deployment.

5.1 Application Tests

The final section of this chapter focuses on the implementation of unit and integration tests, which are crucial in the software development lifecycle. Tests help to evaluate and improve the quality by mitigating the risks, providing confidence, increasing user satisfaction, and enhancing overall software quality. [46].

The testing strategy for the microservices in the system's architecture is based on the Test Pyramid, a model that visually represents the recommended number of tests at different levels of granularity [47], as shown in Figure 5.1. However, it is important to note that UI tests are not implemented in this thesis.

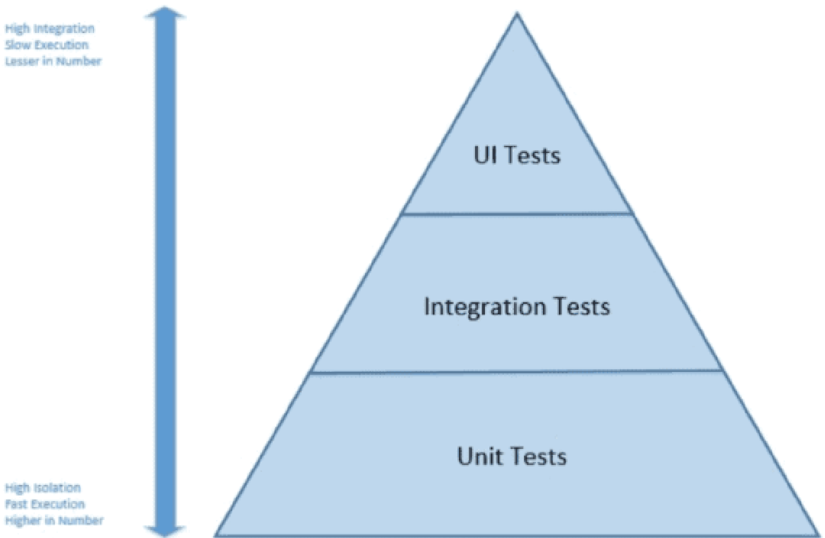

Figure 5.1: Test Pyramid [47]

5.1.1 Unit Tests

Unit tests are vital for testing small parts of an application individually and independently. In this architecture, each microservice includes Unit tests at various layers such as the Repository Layer, Service Layer and Controller Layer. Additionally, to implement these unit tests, the approach involves mocking operations using Mockito [48] as the mocking tool, and JUnit [49] for creating and validating the tests.

Repository Layer

Since the repositories and the entity classes are located in the commons library within the persistence module, the unit tests for the repository layer are also housed in the commons library. The main objective of these tests is to validate the repositories used by the application. As illustrated in Figure 5.2, each repository has its dedicated Java class for unit tests.

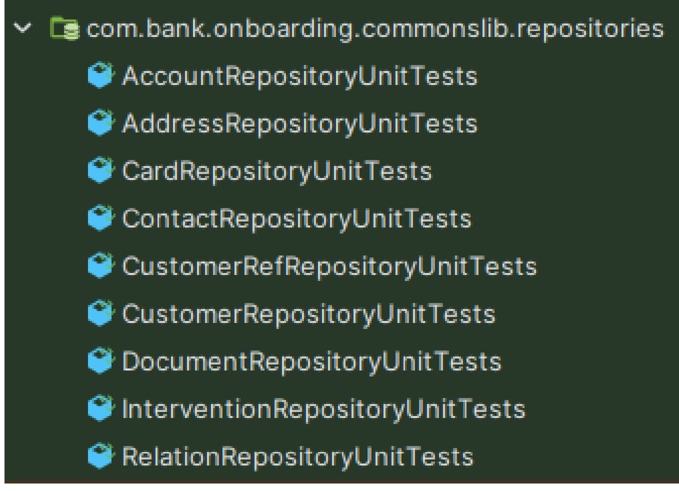

Figure 5.2: Repositories Unit Tests

The Listing 5.1 provides an example of the Java class for the Account Repository Unit Tests Java class, illustrating how the repository unit tests are implemented. This class includes several important Spring Boot annotations:

- **@SpringBootTest**: Used to create the Application Context necessary for running the tests.
- **@Autowired**: Allows the injection of beans into Spring's Application Context, making use of the dependency injection.
- **@BeforeEach**: Executes the code inside the method before each test runs. In this case, it deletes all entries in the Account Repository to ensure each test operates independently.
- **@Test**: This annotation refers that the method corresponds to a new test.

```java
@SpringBootTest(classes = Application.class)
class AccountRepositoryUnitTests {

  @Autowired
  private AccountRepository accountRepository;

  @BeforeEach
  public void cleanDatabase() {
    accountRepository.deleteAll();
  }

  @Test
  void saveAccountTest() {

    //ARRANGE
    Account account = buildAccount();

    //ACT
    Account accountSaved = accountRepository.save(account);

    //ASSERT
    Assertions.assertThat(accountSaved).isNotNull();
    Assertions.assertThat(accountSaved.getId()).isNotEmpty();
    Assertions.assertThat(accountSaved).isEqualTo(account);
  }

  @Test
  void findAccountByNumberTest() {
    Account account = buildAccount();
    accountRepository.save(account);
    Account accountSearched = accountRepository.findByNumber(account.
   getNumber());

    Assertions.assertThat(accountSearched).isNotNull();
    Assertions.assertThat(accountSearched.getId()).isNotEmpty();
    Assertions.assertThat(accountSearched.getNumber()).isEqualTo(account
   .getNumber());
  }
```

Listing 5.1: AccountRepository Unit Tests (Java).

To validate the tests, the Assertions class is employed. If all assertions match the expected result, the test is marked as passed, as demonstrated in Figure 5.3. This process ensures that the repositories function as intended and that the unit tests are reliable indicators of the system's stability.

AccountRepositoryUnitTests (com.bank.onboarding. 1 sec 398 ms
saveAccountTest() 1 sec 245 ms
deleteAccountByNumberTest() 94 ms
findAccountByNumberTest() 34 ms
findAllAccountsByIbanTest() 25 ms

Figure 5.3: Account Repository Tests Passed

Service Layer

Unit Tests for the Service Layer involve testing all the methods within components that are annotated with the @Service annotation in Spring. These tests are implemented in the boot module of each microservice since this module has access to all other modules inside within the project. In addition to the Spring annotations mentioned earlier, the following annotations are also used:

- **@ExtendWith**: According to JUnit documentation [50], this repeatable annotation is used to register extensions for the annotated test class, test interface, test method, parameter, or field.
- **@Mock**: This annotation is used to create and inject mocked instances, such as the "AccountService" interface in this case.

In the Service Layer unit tests, the "AccountService" interface, which contains the business logic methods to be tested, is mocked using the Mockito tool. The "when" method from Mockito is used to simulate the actual behaviour of the "AccountService" interface, while the "assertEquals" from JUnit is used to verify that the expected result matches the actual result produced by the business logic method.

Listing 5.2 provide examples of the unit tests implemented for the "AccountService" interface, demonstrating how each method is validated.

```java
@SpringBootTest(classes = Application.class)
@ExtendWith({SpringExtension.class})
class AccountServiceUnitTests {

    @Mock
    private AccountService accountService;

    private AccountDTO accountDTO;
    private String accountNumber;

    @BeforeEach
    public void setUp() {
        Account account = buildAccount();
        accountDTO = AccountMapper.INSTANCE.toAccountDTO(account);
        accountNumber = account.getNumber();
    }

    @Test
    void createAccountTest() throws Exception{
        CreateAccountRequestDTO createAccountRequestDTO =
    buildCreateAccountRequestDTO();

        when(accountService.createAccount(createAccountRequestDTO)).
    thenReturn(accountDTO);

        assertEquals("M rio Ferreira Neves", accountService.
    createAccount(createAccountRequestDTO).getAccountManager());
        assertEquals("EUR", accountService.createAccount(
    createAccountRequestDTO).getCurrencyCode());
        assertEquals("PT50 0000 2927 8040 8012 4082 5", accountService.
    createAccount(createAccountRequestDTO).getIban());
        assertEquals("80408012408825", accountService.createAccount(
    createAccountRequestDTO).getNumber());
        assertEquals(AccountPhase.INTYPE.getValue(), accountService.
    createAccount(createAccountRequestDTO).getPhase());
        assertEquals(AccountType.ORDEM.name(), accountService.
    createAccount(createAccountRequestDTO).getType());
    }

    @Test
    void patchAccountTypeTest() throws Exception{
        AccountTypeRequestDTO accountTypeRequestDTO =
    buildAccountTypeRequestDTO();

        when(accountService.patchAccountType(accountNumber,
    accountTypeRequestDTO)).thenReturn(accountDTO);

        assertEquals("M rio Ferreira Neves", accountService.
    patchAccountType(accountNumber, accountTypeRequestDTO).
    getAccountManager());
        assertEquals("EUR", accountService.patchAccountType(
    accountNumber, accountTypeRequestDTO).getCurrencyCode());
        assertEquals("PT50 0000 2927 8040 8012 4082 5", accountService.
    patchAccountType(accountNumber, accountTypeRequestDTO).getIban());
        assertEquals("80408012408825", accountService.patchAccountType(
    accountNumber, accountTypeRequestDTO).getNumber());
        assertEquals(AccountPhase.INTYPE.getValue(), accountService.
    patchAccountType(accountNumber, accountTypeRequestDTO).getPhase());
        assertEquals(AccountType.ORDEM.name(), accountService.
    patchAccountType(accountNumber, accountTypeRequestDTO).getType());
    }
```

Listing 5.2: AccountService Unit Tests (Java).

After running the Unit Tests for “AccountService”, all the tests passed successfully, as shown in Figure 5.4.

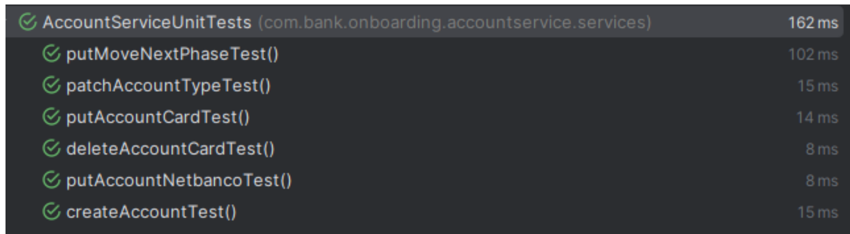

Figure 5.4: Account Service Tests Passed

Controller Layer

The Controller Layer unit tests focus on testing the controllers, which typically expose endpoints and are annotated with the @RestController. This layer, introduces a few new annotations:

- **@WebMvcTest**: This annotation disables full auto-configuration and applies only relevant configurations for MVC tests.
- **@Import**: This annotation is used to import a configuration class, such as “SecurityConfig.class”, which handles JWT Tokens and the “X-Onboarding-Client-Id” HTTP Header required for each endpoint.
- **@MockBeans**: This annotation can be used to mock multiple beans by providing a list of @MockBean.
- **@MockBean**: This annotation adds mock objects to the Spring application context to perform their operations.
- **@Value**: This annotation is typically used to inject external properties, such as those from an “application.properties” file.

In the Controller Layer unit tests, the Mockito when clause is again used, similar to the Service Layer. Additionally, the MockMvc framework is employed to perform actual HTTP calls. The “perform()” method from MockMvc is used to simulate an HTTP request, and the “andExpect()” method verifies that the result matches the expected outcome. Listings 5.3 and 5.4 shows examples of the tests developed for the AccountController.

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(AccountController.class)
@Import(SecurityConfig.class)
@MockBeans({
        @MockBean(OnboardingUtils.class),
        @MockBean(AccountRepoService.class),
        @MockBean(CardRepoService.class),
        @MockBean(CustomerRefRepoService.class),
        @MockBean(CustomerRefRepository.class),
        @MockBean(CustomerRepository.class),
        @MockBean(AccountRepository.class),
        @MockBean(CardRepository.class),
        @MockBean(DocumentRepository.class),
        @MockBean(InterventionRepository.class),
        @MockBean(RelationRepository.class),
        @MockBean(ContactRepository.class),
        @MockBean(AddressRepository.class)
})
class AccountControllerUnitTests {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private SecurityConfig securityConfig;

    @MockBean
    private AccountService accountService;

    @Value("${bank.onboarding.client.id}")
    private String clientId;

    private AccountDTO accountDTO;
    private String accountNumber;
    private String token;

    private final ObjectMapper objectMapper = new ObjectMapper();

    @BeforeEach
    public void setUp() {
        token = securityConfig.generateJWToken();
        objectMapper.registerModule(new JavaTimeModule());

        Account account = buildAccount();
        accountDTO = AccountMapper.INSTANCE.toAccountDTO(account);
        accountNumber = account.getNumber();
    }
```

Listing 5.3: AccountController Unit Tests (1) (Java).

```
@Test
void createAccountTest() throws Exception{
    CreateAccountRequestDTO createAccountRequestDTO =
buildCreateAccountRequestDTO();

    when(accountService.createAccount(createAccountRequestDTO)).
thenReturn(accountDTO);
    mockMvc.perform(post("/accounts")
                    .header("Authorization", "Bearer " + token)
                    .header("X-Onboarding-Client-Id", clientId)
                    .content(objectMapper.writeValueAsString(
createAccountRequestDTO))
                    .contentType(MediaType.APPLICATION_JSON_VALUE))
            .andDo(print())
            .andExpect(status().isCreated())
            .andExpect(content().contentType(MediaType.
APPLICATION_JSON_VALUE))
            .andExpect(jsonPath("$.accountManager").value(accountDTO
.getAccountManager()))
            .andExpect(jsonPath("$.currencyCode").value(accountDTO.
getCurrencyCode()))
            .andExpect(jsonPath("$.iban").value(accountDTO.getIban()
))
            .andExpect(jsonPath("$.number").value(accountNumber))
            .andExpect(jsonPath("$.onlineBankingIndicator").value(
accountDTO.getOnlineBankingIndicator()))
            .andExpect(jsonPath("$.phase").value(accountDTO.getPhase
()))
            .andExpect(jsonPath("$.type").value(accountDTO.getType()
));
}

@Test
void patchAccountTypeTest() throws Exception{
    AccountTypeRequestDTO accountTypeRequestDTO =
buildAccountTypeRequestDTO();

    when(accountService.patchAccountType(accountNumber,
accountTypeRequestDTO)).thenReturn(accountDTO);
    mockMvc.perform(put("/accounts/"+ accountNumber)
                    .header("Authorization", "Bearer " + token)
                    .header("X-Onboarding-Client-Id", clientId)
                    .content(objectMapper.writeValueAsString(
accountTypeRequestDTO))
                    .contentType(MediaType.APPLICATION_JSON_VALUE))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.
APPLICATION_JSON_VALUE))
            .andExpect(jsonPath("$.accountManager").value(accountDTO
.getAccountManager()))
            .andExpect(jsonPath("$.currencyCode").value(accountDTO.
getCurrencyCode()))
            .andExpect(jsonPath("$.iban").value(accountDTO.getIban()
))
            .andExpect(jsonPath("$.number").value(accountNumber))
            .andExpect(jsonPath("$.onlineBankingIndicator").value(
accountDTO.getOnlineBankingIndicator()))
            .andExpect(jsonPath("$.phase").value(accountDTO.getPhase
()))
            .andExpect(jsonPath("$.type").value(accountDTO.getType()
));
}
```

Listing 5.4: AccountController Unit Tests (2) (Java).

As illustrated in Figure 5.5, the unit tests for the "AccountController" have passed successfully, ensuring that the controller are functioning correctly and handling requests as expected.

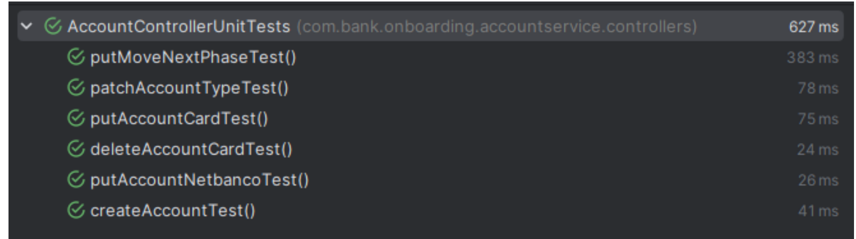

Figure 5.5: Account Controller Tests Passed

5.1.2 Integration Tests

Integration tests combine units, modules, or components to ensure they work together as expected. These tests involve performing real operations to verify that all the layers (Repository, Service, and Controller) are functioning correctly in conjunction. Several new annotations are introduced to facilitate these tests:

- **@SpringBootTest**: This annotation, previously used in unit tests, is now configured to run the web environment on a random port of integration testing.
- **@LocalServerPort**: This annotation binds the port to the API URL, allowing the tests to interact with the application running on this port.
- **@AfterEach**: This annotation ensures that the method annotated with it is executed after each test.

Since integration tests involve real operations, components like "AccountRepository", "AccountService", and "AccountController" must be injected using "@Autowired" annotation instead of "@Mock" or "@MockBean". This approach ensures that the tests interact with the actual components rather than mocked instances.

To test the "AccountController", the "TestRestTemplate" class is used to perform HTTP requests against the exposed endpoints. The results of these requests are then compared with the expected behaviour of the "AccountService" methods. Additionally, the "AccountRepository" is validated by checking if the expected data is present in the database after the operation. The workflow of these integration tests, as shown in Listings 5.5 and 5.6 follow these steps:

- Perform an HTTP request using "TestRestTemplate" to trigger the controller.
- Compare the result with the expected behaviour in "AccountService".
- Validate the data in the "AccountRepository" to ensure it was correctly stored or retrieved from the database.

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.
    RANDOM_PORT)
class AccountApiIntegrationTests {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private AccountRepository accountRepository;

    @Autowired
    private CardRepository cardRepository;

    @Autowired
    private AccountService accountService;

    @Autowired
    private SecurityConfig securityConfig;

    @Value("${bank.onboarding.client.id}")
    private String clientId;

    private String token;
    private Account account;

    private HttpHeaders httpHeaders;
    private final ObjectMapper objectMapper = new ObjectMapper();

    @BeforeEach
    public void setUp() {
        token = securityConfig.generateJWToken();
        httpHeaders = new HttpHeaders();
        httpHeaders.setContentType(MediaType.APPLICATION_JSON);
        httpHeaders.setBearerAuth(token);
        httpHeaders.set("X-Onboarding-Client-Id", clientId);
        objectMapper.registerModule(new JavaTimeModule());

        account = buildAccount();
        accountRepository.save(account);
    }

    @AfterEach
    public void setDown(){
        accountRepository.deleteByNumber(account.getNumber());
        cardRepository.findAllByAccountId(account.getId()).forEach(card
    -> cardRepository.deleteById(card.getId()));
    }

    private String createURLWithPort() {
        return "http://localhost:" + port + "/accounts";
    }
```

Listing 5.5: Account Service Integration Tests (1) (Java).

```java
@Test
void createAccountTest() throws JsonProcessingException {
    HttpEntity<String> entity = new HttpEntity<>(objectMapper.writeValueAsString(buildCreateAccountRequestDTO()), httpHeaders);
    ResponseEntity<?> response = restTemplate.exchange(
            createURLWithPort(), HttpMethod.POST, entity, AccountDTO.class);

    AccountDTO accountDTO = (AccountDTO) response.getBody();
    assertEquals(response.getStatusCode(), HttpStatusCode.valueOf(201));
    assert accountDTO != null;
    assertEquals(accountDTO.getType(), accountService.createAccount(buildCreateAccountRequestDTO()).getType());
    assertEquals(accountDTO.getType(), accountRepository.findByNumber(accountDTO.getNumber()).getType());
}

@Test
void patchAccountTypeTest() throws JsonProcessingException {
    HttpEntity<String> entity = new HttpEntity<>(objectMapper.writeValueAsString(buildAccountTypeRequestDTO()), httpHeaders);
    ResponseEntity<?> response = restTemplate.exchange(
            createURLWithPort() + "/" + account.getNumber(), HttpMethod.PUT, entity, AccountDTO.class);

    AccountDTO accountDTO = (AccountDTO) response.getBody();
    assertEquals(response.getStatusCode(), HttpStatusCode.valueOf(200));
    assert accountDTO != null;
    assertEquals(accountDTO.getType(), accountService.patchAccountType(account.getNumber(), buildAccountTypeRequestDTO()).getType());
    assertEquals(accountDTO.getType(), accountRepository.findByNumber(account.getNumber()).getType());
}

@Test
void putAccountCardTest() throws JsonProcessingException {
    HttpEntity<String> entity = new HttpEntity<>(objectMapper.writeValueAsString(buildAccountCardDTO()), httpHeaders);
    ResponseEntity<?> response = restTemplate.exchange(
            createURLWithPort() + "/" + account.getNumber() + "/card", HttpMethod.PUT, entity, CardDTO.class);

    CardDTO cardDTO = (CardDTO) response.getBody();
    assertEquals(response.getStatusCode(), HttpStatusCode.valueOf(200));
    assert cardDTO != null;
    assertEquals(cardDTO.getType(), accountService.putAccountCard(account.getNumber(), buildAccountCardDTO()).getType());
}
```

Listing 5.6: Account Service Integration Tests (2) (Java).

If all conditions are met, the test is marked as passed, indicating all the layers (Repository, Service, and Controller) are working together as expected. Figure 5.6 illustrates that all integration tests, which involved performing real actions, ran successfully without errors, confirming that the entire application stack is functioning correctly.

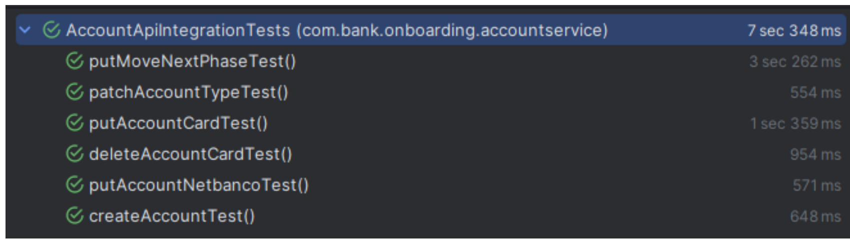

Figure 5.6: Account Service Integration Tests Passed

5.2 Performance and Load Testing

To evaluate the performance of the application, Apache JMeter [51] was used to automate performance and load tests. The scenario of the tests followed the most important use cases of the application, and it took into consideration the student's computer resources:

- **Processor**: INTEL Core i5-4590 3.30GHz
- **Video Card**: NVIDIA GeForce GTX 960
- **RAM**: 16GB
- **Operating System**: Windows 10

Additionally, the tests were done with the database being in a cloud environment provided by MongoDB free plan, and Kafka, Nginx API Gateway and all the microservices running in Docker Containers on the student's computer as illustrated in the following figure.

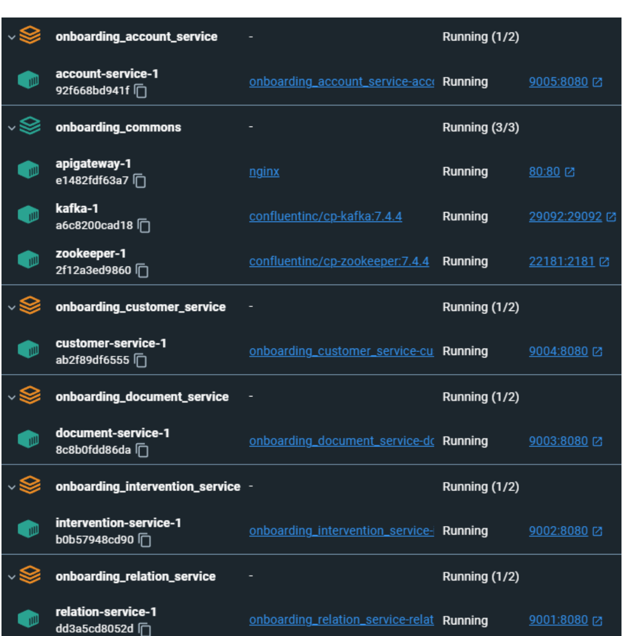

Figure 5.7: Microservices running in Docker Containers

Therefore, the testing plan was testing each use case with three different scenarios: 5 virtual users, 50 virtual users and 500 virtual users. Table 5.1 shows the results of performance and load experiments also for each use case, the time elapsed for processing was measured, and it is also compared with the time elapsed when horizontal scalability is applied (using 2 instances per microservice).

Use Case	Number of Users	Time elapsed (s)	Time elapsed with horizontal scalability (s)
Create case	5	0.14	0.13
	50	1.14	0.57
	500	7.55	5.6
Add intervention	5	0.48	0.45
	50	1.12	0.99
	500	12.34	10.92
Add relation	5	0.25	0.28
	50	1.10	0.85
	500	13.85	11.96
Update customers	5	0.41	0.52
	50	2.20	1.35
	500	18.34	16.36
Move to next phase	5	0.12	0.12
	50	1.73	0.2
	500	7.79	5.78

Table 5.1: Performance and Load Experiments

The same pattern is seen across all use cases. For example, in the "Create case" use case, when 5 users were involved, the process took 0.14 seconds without scalability, and 0.13 seconds with it. As the number of users increased to 500, the time without scalability growth to 7.55 seconds, but with scalability, it reduced to 5.60 seconds. Similarly, in the "Add intervention" use case, with 5 users, the time was 0.48 seconds without scalability and 0.45 seconds with it. For 500 users, the process took 12.34 seconds without scalability, and with horizontal scalability, it improved to 10.92 seconds.

In summary, using 2 instances per microservice for horizontal scalability showed an average performance improvement of 10% when processing each use case compared to not using horizontal scalability, especially when the system handled a larger number of users, such as 500 virtual users. Although the improvement percentage is relatively low, it is important to consider that the tests do not reflect real-world conditions, as they were conducted on the student's computer, with only one CPU managing all the Docker Containers.

Chapter 6

Conclusion

This section aims to provide a critical overview of the difficulties experienced, accomplished work, and outlines future directions. It's evident that adopting MSA can improve the application's scalability and maintainability, based on the literature review focused on low-code applications and microservices, and based on the evaluation of this thesis.

6.1 Difficulties

The development of this thesis presented several challenges. The first significant challenge was the scarcity of studies on performance, scalability, and maintainability issues in low-code based applications. While these topics are critical for understanding the limitations and potential of low-code-based applications, especially in complex sectors like banking, there is a notable lack of scientific and technical studies addressing them. The difficulty was even more pronounced when attempting to find studies focused specifically on the banking sector, where the stakes for performance, scalability, and maintainability are particularly high.

Another difficulty, was replicating the tests using real-world scenarios, because the tests were conducted on the student's computer. The performance improvement was only around 10%, but this could have been higher if virtual machines in cloud environments had been used.

Despite these challenges, the student was able to overcome them successfully, demonstrating the application and supporting the work with relevant studies and a fully developed microservices-based solution. This process provided valuable insights into the complexities of software development, particularly in the context of distributed systems, to enhance system performance, scalability, maintainability, and ease of deployment.

6.2 Achievements

In Introduction Chapter, the objective of this thesis was defined: partitioning an application into microservices can improve performance, scalability, maintainability, and ease of deployment. Based on existing studies [1, 3], the solution was implemented using a microservices-based architecture developed with the Spring Boot framework in the Java programming language. The implementation details, including some code analysis, were discussed in Chapter 4.

Throughout this document, it has been demonstrated how each microservice operates with its own database, following the Database per Service Pattern described in Chapter 2. The microservices communicate to each other via Apache Kafka, using events with specific keys. This thesis also covered the experiments of the application using unit and integration tests to ensure the functionality of each microservice, and load tests that confirmed improvements in performance and scalability. Moreover, containerization was employed to independently deploy each microservice using a "docker-compose.yml" file, improving the deployment of the services, one of the key characteristics referred in the initial objectives. Lastly, maintainability improvements were achieved through the commons library described in Chapter 4, promoting code reuse over duplication.

In conclusion, despite confidentiality limitations preventing a comparison with the existing low-code application, the project successfully met its key objectives.

6.3 Future Work

For the future, it will be good to make performance and load tests based on real-world scenarios, with each microservice running independently on virtual machines in a cloud environment. This would allow for more accurate comparison with the real application developed in Appian. Such tests are expected to demonstrate better results for the microservices architecture than those achieved in this project, which was limited to the student's computer.

Additionally, would be very interesting to implement the following two enhancements that could significantly strengthen the architecture developed in this thesis:

1. **Implement the Saga Pattern:** Implementing this pattern, as described in Chapter 2, would be a valuable addition to the existing architecture, especially when combined with the Database per Service pattern already implemented. It also enhances the system's robustness by providing mechanisms for failure recovery, such as reverting the failure and cleaning up afterwards, like a rollback, or recover from the point when the failure occurred and keep processing. The Saga pattern would help maintain data consistency across multiple services in a distributed system by managing and coordinating database transactions.
2. **Create CI/CD pipeline:** This improvement would reduce manual errors and ensuring that the new code changes are quickly and reliably deployed by automating the build, testing, and deployment processes. A good approach for the pipeline could involve monitoring a specific branch, like the develop branch, triggering a build process using Maven, executing unit and integration tests, and finally deploying the application into containers.

By implementing these two features, the system's architecture would be further strengthened, offering enhanced reliability, scalability, and ease of deployment. These improvements would make the system more resilient and adaptable to future changes and challenges.

Bibliography

[1] Sam Newman. *Building Microservices*. 2nd. O'Reilly Media, Inc., 2021. isbn: 1492034029.
[2] Martin Flower. *MonolithFirst*. url: https://martinfowler.com/bliki/MonolithFirst.html.
[3] Martin Flower. *Microservices*. url: https://martinfowler.com/microservices/.
[4] IEEE Xplore. url: https://ieeexplore.ieee.org/.
[5] Google Scholar. url: https://scholar.google.com/.
[6] ACM Digital Library. url: https://dl.acm.org/.
[7] Kaye Towlson. "The Information Source Evaluation Matrix: a quick, easy and transferable content evaluation tool". In: (2010).
[8] Shanshan Li et al. "Detecting Performance Bottlenecks Guided by Resource Usage". In: *IEEE Access* 7 (2019), pp. 117839–117849. doi: 10.1109/ACCESS.2019.2936599.
[9] Mckinsey. "Consumer trends in digital payments". In: (2022).
[10] Appian. *Performance*. url: https://appian.com/process-mining/performance.html.
[11] Sebastian Käss, Susanne Strahringer, and Markus Westner. "A Multiple Mini Case Study on the Adoption of Low Code Development Platforms in Work Systems". In: *IEEE Access* 11 (2023), pp. 118762–118786. doi: 10.1109/ACCESS.2023.3325092.
[12] Apurvanand Sahay et al. "Supporting the understanding and comparison of low-code development platforms". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020, pp. 171–178. doi: 10.1109/SEAA51224.2020.00036.
[13] Mike Anand. *Microservices and the Data Layer—a New IDC InfoBrief*. url: https://redis.com/blog/microservices-and-the-data-layer-new-idc-infobrief/.
[14] C. Olofson and G. Chen. *The Impact of Application Modernization on the Data Layer*. url: https://redis.com/docs/application-modernizaton-impact-on-data-layer/.
[15] Sh. Salii, J. Ajdari, and Xh. Zenuni. "Migrating to a microservice architecture: benefits and challenges". In: *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*. 2023, pp. 1670–1677. doi: 10.23919/MIPRO57284.2023.10159894.
[16] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation". In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32. doi: 10.1109/MCC.2017.4250931.
[17] Microsoft. *Microservice architecture style*. url: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.
[18] Chris Richardson. *Microservice Architecture*. url: https://microservices.io/patterns/microservices.html.
[19] Chris Richardson. *When you outgrow your monolithic architecture*. url: https://microservices.io/.
[20] Broadcom Inc. *RabbitMQ*. url: https://www.rabbitmq.com/.
[21] Chris Richardson. *A pattern language for microservices*. url: https://microservices.io/patterns/index.html.

[22] Chris Richardson. *Database per service*. url: https://microservices.io/patterns/data/database-per-service.html.

[23] Chris Richardson. *Shared database*. url: https://microservices.io/patterns/data/shared-database.html.

[24] Chris Richardson. *Saga*. url: https://microservices.io/patterns/data/saga.html.

[25] The Kubernetes Authors. *Kubernetes*. url: https://kubernetes.io/.

[26] Amazon Web Services. *AWS Lambda*. url: https://aws.amazon.com/pt/lambda/.

[27] Microsoft. *Azure Functions Overview*. url: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp.

[28] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232. issn: 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.09.082. url: https://www.sciencedirect.com/science/article/pii/S0164121218302139.

[29] Javad Ghofrani and Daniel Lübke. "Challenges of Microservices Architecture: A Survey on the State of the Practice". In: (May 2018).

[30] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. "Migrating from monolithic architecture to microservices: A Rapid Review". In: *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. 2019, pp. 1–7. doi: 10.1109/SCCC49216.2019.8966423.

[31] Justas Kazanavičius and Dalius Mažeika. "An Approach to Migrate from Legacy Monolithic Application into Microservice Architecture". In: *2023 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*. 2023, pp. 1–6. doi: 10.1109/eStream59056.2023.10135021.

[32] Lorenzo De Lauretis. "From Monolithic Architecture to Microservices Architecture". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 93–96. doi: 10.1109/ISSREW.2019.00050.

[33] Apache Software Foundation. *Kafka*. url: https://kafka.apache.org/.

[34] Google. *gRPC*. url: https://grpc.io/.

[35] The GraphQL Foundation. *GraphQL*. url: https://graphql.org/.

[36] Jonas Fritzsch et al. "Microservices Migration in Industry: Intentions, Strategies, and Challenges". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 481–490. doi: 10.1109/ICSME.2019.00081.

[37] Lawrence Chung and Julio Cesar Prado Leite. "On Non-Functional Requirements in Software Engineering". In: *Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos*. 2009, pp. 363–379. isbn: 9783642024627.

[38] Alistair Jacobson Ivar & Cockburn. *Use Cases are Essential*. url: https://dl.acm.org/doi/fullHtml/10.1145/3631182.

[39] Oleg Chursin. *A Brief Introduction to Domain Modeling*. url: https://olegchursin.medium.com/a-brief-introduction-to-domain-modeling-862a30b38353.

[40] IBM. *What is data modeling?* url: https://www.ibm.com/topics/data-modeling.

[41] Spring. *Spring Initializr*. url: https://start.spring.io/.

[42] Spring. *Creating a Multi Module Project*. url: https://spring.io/guides/gs/multi-module.

[43] Dr Milan Milanović. *https://medium.com/@techworldwithmilan/what-is-api-gateway-91387e19dbd9*. url: https://medium.com/@techworldwithmilan/what-is-api-gateway-91387e19dbd9.

[44] Amazon Web Services. *What's the Difference Between Kafka and RabbitMQ?* url: https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka.

[45] Tapan Avasthi. *Guide to Setting Up Apache Kafka Using Docker*. url: https://www.baeldung.com/ops/kafka-docker-setup.

[46] IEEE Computer Society. *The Importance of Software Testing*. url: https://www.computer.org/resources/importance-of-software-testing.

[47] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. 2009. isbn: 9780321579362.

[48] Mockito. *Mockito Framework*. url: https://site.mockito.org/.

[49] JUnit. *JUnit 5*. url: https://junit.org/junit5/.

[50] JUnit. *Annotation Interface ExtendWith*. url: https://junit.org/junit5/docs/5.8.0/api/org.junit.jupiter.api/org/junit/jupiter/api/extension/ExtendWith.html.

[51] The Apache Software Foundation. *Apache JMeter*. url: https://jmeter.apache.org/.