



## Assignment of master's thesis

<b>Title:</b>	Data consistency in microservices architecture
<b>Student:</b>	Bc. Iaroslav Kolodka
<b>Supervisor:</b>	Ing. Nikolay Barbariyskiy
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Software Engineering
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	until the end of summer semester 2022/2023

### Instructions

The aim of this work is to conduct research on a common problem in a microservice architecture, which is data consistency.

There are few architectural patterns and techniques, which can help developers solve this problem. The student will analyze given patterns and techniques and propose use-cases in which given approaches are more suitable and beneficial.

Based on this analysis and gained knowledge, the student will propose a solution for a concrete problem (Applifting s.r.o will provide a project with technical requirements), which is dealing with the problem of consistent data migration from RDBMS to Elasticsearch to enable efficient full-text search over given set of data. This project will be based on a commercial project, but it will be used only for research purposes in this work.

#### Tasks:

- Gain an understanding of the problem of data consistency in microservice architectures;
- Analyze common approaches for handling a given problem;
- Propose which patterns and approaches are suitable for particular use-cases;
- Based on research, apply a suitable pattern for a concrete project for consistent data migration from RDBMS storage to Elasticsearch;
- Test and evaluate whether the proposed solution is efficient and applicable in real-world conditions.



Master's thesis

# **DATA CONSISTENCY IN MICROSERVICES ARCHITECTURE**

**Bc. Iaroslav Kolodka**

Faculty of Information Technology  
Department of software engineering  
Supervisor: Ing. Nikolay Barbariyskiy  
January 1, 2023

Czech Technical University in Prague  
Faculty of Information Technology

© 2022 Bc. Iaroslav Kolodka. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

Citation of this thesis: Kolodka Iaroslav. *Data consistency in microservices architecture*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

# Contents

<b>Acknowledgments</b>	<b>viii</b>
<b>Declaration</b>	<b>ix</b>
<b>Abstract</b>	<b>x</b>
<b>Summary</b>	<b>xi</b>
<b>List of acronyms</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Purpose of the work . . . . .	1
1.2 Structure of the work . . . . .	1
<b>2 Research</b>	<b>1</b>
2.1 What is the Microservices Architecture approach? . . . . .	1
2.2 Problems Microservices Architecture brings . . . . .	1
2.2.1 Splitting up the project . . . . .	2
2.2.2 Management . . . . .	3
2.2.3 Availability . . . . .	3
2.2.4 Monitoring . . . . .	3
2.2.5 Testing . . . . .	4
2.2.6 Security . . . . .	4
2.2.7 CI/CD . . . . .	4
2.2.8 Data consistency . . . . .	4
2.3 The problem of data consistency . . . . .	4
2.4 Terminology . . . . .	7
2.4.1 Data consistency . . . . .	7
2.4.2 Distributed transaction . . . . .	7
2.5 Evaluation approaches . . . . .	8
2.5.1 ACID . . . . .	8
2.5.2 Required effort . . . . .	8
2.6 Provided project . . . . .	9
2.7 Keycloak . . . . .	10
2.8 Apache Kafka . . . . .	10
2.9 Elasticsearch . . . . .	10
2.10 PgSync . . . . .	10
2.11 Logstash . . . . .	11
2.12 Debezium . . . . .	11
2.13 Spring Cloud Stream . . . . .	11
2.14 Locust . . . . .	13
2.15 Programming language . . . . .	13
2.16 Git approach . . . . .	13
2.17 Build automation tool . . . . .	15

2.18	Research summary . . . . .	15
<b>3</b>	<b>Analysis</b>	<b>17</b>
3.1	Two-phase commit pattern . . . . .	17
3.1.1	Description . . . . .	17
3.1.2	Use-case . . . . .	17
3.1.3	Evaluation . . . . .	19
3.1.4	Implementations . . . . .	20
3.2	Saga pattern . . . . .	21
3.2.1	Description . . . . .	21
3.2.2	Use-case . . . . .	23
3.2.3	Evaluation . . . . .	23
3.2.4	Implementations . . . . .	23
3.3	Transactional outbox pattern . . . . .	23
3.3.1	Description . . . . .	23
3.3.2	Use-case . . . . .	25
3.3.3	Evaluation . . . . .	25
3.3.4	Implementations . . . . .	25
3.4	Event sourcing . . . . .	25
3.4.1	Description . . . . .	25
3.4.2	Use-case . . . . .	27
3.4.3	Evaluation . . . . .	27
3.4.4	Implementations . . . . .	27
3.5	CQRS pattern . . . . .	27
3.5.1	Description . . . . .	27
3.5.2	Use-case . . . . .	27
3.5.3	Evaluation . . . . .	27
3.6	Back to the monolith . . . . .	28
3.6.1	Description . . . . .	28
3.6.2	Use-case . . . . .	28
3.7	Analysis summary . . . . .	28
<b>4</b>	<b>Design</b>	<b>29</b>
4.1	Application architecture . . . . .	29
4.1.1	Chosen scope of "Project Alpha" . . . . .	29
4.1.2	Application use-cases and requirements . . . . .	30
4.2	Consistent data migration . . . . .	33
4.2.1	Transactional outbox as chosen pattern . . . . .	33
4.2.2	Transaction log tailing as Message relay . . . . .	33
4.2.3	Message ordering . . . . .	33
4.2.4	Message deduplication . . . . .	33
4.3	Components . . . . .	33
4.3.1	Domain service . . . . .	34
4.3.2	Identity management service . . . . .	35
4.3.3	Identity provider service . . . . .	36
4.3.4	Binder . . . . .	36
4.3.5	Elasticsearch consumer service . . . . .	36
4.3.6	Elasticsearch instance . . . . .	36
4.4	Design summary . . . . .	39

<b>5</b>	<b>Implementation</b>	<b>41</b>
5.1	Repositories . . . . .	41
5.2	Architecture . . . . .	41
5.3	Consistent data migration . . . . .	42
5.3.1	Message ordering . . . . .	42
5.3.2	Message deduplication . . . . .	43
5.4	Domain service . . . . .	43
5.5	Apache Kafka binder . . . . .	43
5.6	Elasticsearch instance . . . . .	43
5.7	Identity management service . . . . .	43
5.7.1	REST API adapter . . . . .	44
5.7.2	Elasticsearch adapter . . . . .	45
5.7.3	Identity provider adapter client . . . . .	46
5.7.4	Domain logic . . . . .	46
5.8	Identity provider service . . . . .	47
5.8.1	REST API adapter . . . . .	47
5.8.2	Domain logic . . . . .	47
5.8.3	Message relay . . . . .	48
5.9	Elasticsearch consumer service . . . . .	48
5.9.1	Binder . . . . .	49
5.9.2	Consumer . . . . .	49
5.10	Documentation . . . . .	49
5.10.1	API documentation . . . . .	49
5.10.2	Developer guide . . . . .	50
5.11	Potentially useful upgrades . . . . .	50
5.11.1	Schema registry for messages . . . . .	50
5.11.2	Use prepared Spring Cloud application . . . . .	50
5.11.3	Monitoring . . . . .	50
5.12	Implementation summary . . . . .	51
<b>6</b>	<b>Testing</b>	<b>53</b>
6.1	End-to-end tests . . . . .	53
6.2	Load tests . . . . .	54
6.3	Future steps . . . . .	55
6.4	Testing summary . . . . .	55
<b>7</b>	<b>Conclusion</b>	<b>57</b>
<b>A</b>	<b>Testing results</b>	<b>59</b>
	<b>Contents of the attached media</b>	<b>65</b>

## List of Figures

2.1	Component diagram for Book store . . . . .	2
2.2	Two generals paradox illustration from [1] . . . . .	5
2.3	Byzantine generals attack from [3] . . . . .	5
2.4	Generals approaches in "Two Generals Paradox" . . . . .	6
2.5	Reservation payment sequence diagram . . . . .	7
2.6	"Project alpha" component diagram . . . . .	9
2.7	Example of the pipeline built with the Debezium from official documentation [7] . . . . .	11
2.8	Spring Cloud Stream Application architecture from official documentation [8] . . . . .	12
2.9	Spring Cloud Stream relationship between Producer and Consumer from official documentation [8] . . . . .	12
2.10	Example of GitHub commit tree with applied rules for the format . . . . .	14
3.1	Distributed transaction sequential diagram . . . . .	18
3.2	Two-phase commit sequential diagram . . . . .	18
3.3	Distributed transactions interference sequential diagram . . . . .	20
3.4	TM and RMs exchange transaction information from [15] . . . . .	21
3.5	Orchestration Saga flow . . . . .	22
3.6	Choreography Saga flow . . . . .	22
3.7	The process of publishing events . . . . .	24
3.8	Transactional outbox pattern sequential diagram . . . . .	24
3.9	The Event sourcing diagram . . . . .	26
4.1	User creation use-case . . . . .	30
4.2	Get user by id use-case . . . . .	31
4.3	User delete use-case . . . . .	31
4.4	User update use-case . . . . .	32
4.5	User search use-case . . . . .	32
4.6	IMS component architecture . . . . .	35
4.7	IDP component architecture . . . . .	37
4.8	Elasticsearch consumer component architecture . . . . .	38
5.1	Component diagram of the resulting architecture . . . . .	42
5.2	IMS Swagger UI . . . . .	45
5.3	IDP Swagger UI . . . . .	47
5.4	IDP Entity diagram . . . . .	48
6.1	End-to-end tests summary . . . . .	54
6.2	Performed requests during the load testing . . . . .	55
A.1	Response times during the load testing . . . . .	60
A.2	Number of requests per second during the load testing . . . . .	61



## List of Tables

3.1	ACID evaluation for 2-PC pattern . . . . .	19
3.2	ACID evaluation for Saga pattern . . . . .	23

## List of code listings

2.1	Template for the Git commit . . . . .	14
5.1	Spring Cloud Stream configuration for the Confluent Apache Kafka . . . . .	44
5.2	Custom application properties for the communication with Elasticsearch . . . . .	44
5.3	Elasticsearch User index . . . . .	45
5.4	Elasticsearch request builder DSL . . . . .	46
5.5	Message key expression configuration . . . . .	48
5.6	Producer distribution configuration . . . . .	49

*I am incredibly thankful to my supervisor, Ing. Nikolay Barbariskiy, and the Applifting s.r.o. team, who gave me the opportunity to work on this topic and supported me throughout the time I was working on this thesis work. They allowed me to work on this thesis alongside my main employment and do not limit me on my ideas.*

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 1, 2023

.....

## Abstract

This thesis work aims to analyze the architectural patterns which can be used to provide consistent data migration from RDBMS storage to Elasticsearch in a microservice architecture. This thesis work covers the analysis of the most popular patterns that can be found on the internet with a short description, comparisons, and possible implementations. Based on the performed analysis, a design of the solution for the problem is proposed for the provided project. Afterward, the proposed architecture was implemented and tested as greenfield microservices. The conclusion summarizes all aspects of what was done and the results of the work.

**Keywords** microservices architecture, data consistency, Spring Cloud Stream

## Abstrakt

Cílem této práce je analyzovat architektonické vzory, které lze použít k zajištění konzistentní migrace dat z úložiště RDBMS do Elasticsearch v architektuře mikroslužeb. Tato diplomová práce zahrnuje analýzu nejpopulárnějších vzorů, které lze nalézt na internetu, s krátkým popisem, porovnáním a možnými implementacemi. Na základě provedené analýzy je udělán návrh řešení dané problematiky pro poskytnutý projekt. Následně byla navržená architektura implementována a testována jako mikroslužby na zelené louce. Závěr shrnuje všechny aspekty provedené práce a její výsledky.

**Klíčová slova** architektura mikroslužby, konzistence dat, Spring Cloud Stream

## Summary

### Research section

This chapter is dedicated to covering several important topics that will immerse in the theme of Microservice Architecture and the problem of data consistency. Besides the theoretical part, the chapter also covers more practical topics, such as the introduction to the technologies that will be used in the follow-up chapters.

### Analysis section

This chapter describes, evaluates, and compares different patterns for providing data consistency in Microservices Architecture. These patterns can be found on the internet and are commonly used across the industry for different specific use cases. The problem is determining what these use cases are and what particular brings to the project. This chapter will clarify some aspects of these use cases for each observed pattern.

### Design section

This chapter will describe the design of the implementation that will solve the problem of con-

sistent data migration from the RDBMS to Elasticsearch. The solution consists of choosing the correct pattern based on the previous chapter and designing the architecture for all participating microservices.

### Implementation section

This chapter is dedicated to describing the applied pattern in practice and describing the implemented microservices. Each microservice will be described both individually and as a part of the whole infrastructure.

### Testing section

This chapter is dedicated to testing the implemented infrastructure, including load testing, to verify whether the implementation meets expectations.

### Conclusion section

In this chapter will be described the result of the work done in this thesis work.

## List of acronyms

2-PC	Two-phase commit
API	Application Programming Interface
CD	Continuous deployment
CDC	Change Data Capture
CI	Continuous integration
CQRS	Command and Query Responsibility Segregation
CRUD	Create Read Update Delete
CT	Compensation transaction
IDP	Identity Provider
IMS	Identity Management Service
JTA	Java Transaction API
MR	Message Relay
RDBMS	Relational Database Management System
SCS	Spring Cloud Stream
SEC	Saga Execution Coordinator
SpEL	Spring Expression Language
UUID	Universally unique identifier

# Introduction

## 1.1 Purpose of the work

Nowadays, Microservices Infrastructure is a widely used pattern across the whole industry. Quite often, projects are getting to the point where maintenance is already a significant problem. Each time when there is a new issue comes up that needs to be done, there is a new development process life-cycle that needs to be conducted.

The project provided for this thesis work has the same problems as was mentioned. Through this project flows a huge amount of data, and at some point, a decision was made to add an additional element to the infrastructure, which is Elasticsearch. It should significantly level up the user experience, but at the same time, it is a complex task, as the source of truth remains the same, RDBMS storage. Therefore, it adds the problem of data consistency to this project change.

Subsequently, the purpose of this work is to provide a possible solution that will solve the problem of consistent data migration between PostgreSQL and Elasticsearch.

## 1.2 Structure of the work

### Research

The problematics of Microservices Architecture are highly complex. Therefore, firstly, the reader will be introduced to Microservices Architecture and its benefits and drawbacks.

Then, the reader will be immersed in the problem of data consistency with an introduction to the theory and terminology surrounding this problem.

And as the last part of the chapter, the reader will be introduced to the main complex technologies that will be used across this thesis work.

### Analysis

Nowadays, the internet is full of different patterns for data consistency, and most of them claim that they solve all issues, which is dangerous. Therefore, this chapter will be dedicated to an analysis of the popular patterns developer can find on the internet on this topic. The analysis will include a description, evaluation, potential use cases, and possible implementation.

## Design

With the knowledge from the Analysis, the chapter will be created a design of the implementation solving giving problem within the provided project.

## Implementation

Provided project is running a solution that is too complex to change right away. Therefore, in this thesis work will be created a separate version of this infrastructure that includes only parts that are involved in data migration, and each of these parts will be simplified. In this manner, new infrastructure will be real-life tested with the least amount of spend time and effort. Otherwise, it will be much more than one thesis work by itself due to the total size of the project and its complexity.

## Testing

An implementation needs to be tested to determine whether it meets the expectations of the provided project. It will be:

- end-2-end tests;
- Load tests.

## Conclusion

The last chapter will be a Conclusion where the results of the work will be summarized, and possible next steps for the project will be described.



*This chapter is dedicated to covering several important topics that will immerse in the theme of Microservice Architecture and the problem of data consistency. Besides the theoretical part, the chapter also covers more practical topics, such as the introduction to the technologies that will be used in the follow-up chapters.*

## 2.1 What is the Microservices Architecture approach?

Suppose some developers think that creating an application with Microservice Architecture means that there is a need to split the application into a few independent parts. In that case, it is true, but this is only the tip of the iceberg. Microservices Architecture is a complicated approach that brings many benefits to the project, but at the same time, it brings many challenges to solve.

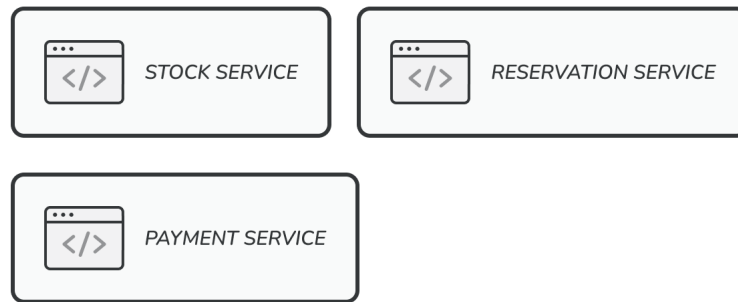
Many companies with experience developing monolithic applications want to keep up with the progress and change their approaches. One of the paths they can choose is, of course, to go the way of Microservice Architecture. This path is incredibly straightforward when the project already requires splitting the logic into independent parts.

However, unfortunately, many of these companies or teams (even successful ones) do not understand that this approach is more complex than it seems when you want to enrich its full potential and break through all of its challenges. The company also needs to change its approaches on a testing level, management level, and leading level. Moreover, keep this approach through all the technologies they use to eliminate potential bottlenecks.

## 2.2 Problems Microservices Architecture brings

We can start talking about all the benefits this architecture brings to the project and the whole company. However, there is much information everywhere on the internet, and the vast majority of topics describe how this approach can be applied or, in general, that this approach is perfect. Nevertheless, Microservices Architecture is not a perfect approach, and it has many drawbacks, and there is not much information on the internet that developers can find. Therefore, this section will describe potential problems developers must be aware of.

Starting this part of the work, we will slowly go deep down to the technical and more complicated topics. It is much easier to describe these aspects with some examples in hand. So, to give some context, let us define an example project that will be used across all the sections. This project does not need to be complex or close to real life, as we need it only as an example. Imagine a bookstore allowing customers to purchase a book from the stock.



■ **Figure 2.1** Component diagram for Book store

The main purpose of the application will be the process of purchasing the book:

1. The client chooses a book from a limited stock;
2. The client making a reservation for a particular book;
3. The client is making payment for his reservation;
4. As soon as payment is successful, the store sends the book to the client and decreases the number of the same books in stock.

### 2.2.1 Splitting up the project

The most apparent problem developer can face, of course, is splitting the project itself. It could sound easy on paper, but the problem is that it could be extremely challenging if the project already exists or if there is already existing architecture. The project could be tightly coupled, as it was or still is a monolith application.

As for our example application, it is obvious where we can draw lines for future independent parts (see 2.1). Each part will have its own responsibility:

- Stock service;
- Reservation service;
- Payment service.

It is straightforward how we split up the application, but it can be different for some projects. There can be many possibilities for how to do it. We can draw the lines between parts based on their responsibilities from the domain perspective, but we can also look at it from another perspective. For example, some of our functionalities are used much more often than others. Therefore we want to separate these functionalities into separate instances and multiply instances to eliminate the system's bottlenecks.

As a result, splitting up the example project will allow the creation of multiple instances of the Reservation service without the need to multiply the compute power for the Stock service, as the reservation process will be much more extensive. Drawing these lines, we must keep in mind that we are creating separate applications that need to be independent of each other. As for the other perspective, each instance can be managed, maintained, or even replaced entirely independently from others.

### 2.2.2 Management

From the management perspective, Microservices Architecture lets companies have a dedicated team per microservice. Companies are getting to this approach at some point, as this architecture requires individual microservices to be independently deployable.

You can ask: "Where is the problem?". However, not every company considers that they need to make these changes from the beginning. Otherwise, they spend much additional effort during the ongoing development.

Moreover, the approach to development should be different as well. There can be many options that depend on team configuration or the domain of the project. However, as an example, we can take an agile methodology that assumes that teams are working on smaller parts of the whole project in collaboration with the client.

As for our example project, we can dedicate separate teams for each microservice, and each of these teams will specialize in a specific domain. This approach will certainly increase the quality of code, but on the other hand, it is not cheap. Nevertheless, regardless of the team allocations, we need to maintain microservices life cycles separately.

### 2.2.3 Availability

Depending on the domain of the application and technologies used, latency can be a big problem in the microservices architecture. Each service can be deployed independently, and communication between them would take some time. However, what if some message was lost or some microservices were down? In this case, we are not talking about latency but about availability, the term we are not used to in monolithic architecture.

Let us look at our Book store. For example, the user wants to pay for the reserved book, but the Payment service is not responding or, in other words, unavailable. We need to cover this situation on the application level and respond to the user with a human-readable result, as this situation can happen.

### 2.2.4 Monitoring

Wide variety of projects going in the direction of microservices for a certain reason. Mostly, this reason is the scope of the project. Therefore the project has a lot of independent microservices or many instances of these microservices, and you need to be aware of the status of these instances:

- Overall health;
- The workload of each instance;
- Whether each API endpoint is reachable;
- Whether transactions interfering with each other;
- Latency for the end-user experience.

Let us look at our example application. There are many things that we could assume as subjects of monitoring. Moreover, some things can even be required to be monitored. For example, we can use an external payment service that requires us to track payment request unique ids to be traced in case of failures. For our Book store, it is not a big problem, but imagine some vast projects with similar requirements. For example, Netflix has a colossal microservice network all over the world.

### 2.2.5 Testing

Testing is an essential part of the development of any project, and we need to test both small independent parts of the application and the entire application. We, of course, can implement unit tests the same as for monolithic applications, but as for the end-to-end test, it is not as easy as before. For example, our Book store now has completely independent microservices, but we still want to test whether we correctly cover all use-case scenarios, such as the book reservation or the process of payment for the reservation.

### 2.2.6 Security

Along with other problems, developers face one of the most unpleasant problems, which is security. In the case of the monolithic, all modules or layers of the application communicate with each other within one application, which is a secure way from the communication perspective.

But when we are building the Microservices architecture, our services communicate with each other within some network, which can be an intranet or even an internet. Therefore there can be cyberattacks, such as man-in-the-middle, that we need to be aware of.

### 2.2.7 CI/CD

Continued integration and continuous development are essential parts of microservices development. Just imagine that the project has a dozen of services that communicate with each other and are deployed in AWS. In that case, even if you want to run the project on a local machine, the project will most probably depend on some service running in the cloud.

Our example project would already have three independent services we need to set up and run to run the whole application.

### 2.2.8 Data consistency

All of these problems that were mentioned are solvable. Some of them are complex, and some are time-costing, but now, we need to talk about problems that are not simply solvable. One of these problems is data consistency.

In case our system is a set of independent services, we need to provide communication between them. Most importantly, this communication needs to be consistent and robust. However, the network is not reliable by its design. All of the possible problems are consequences of the two core problems:

- exactly once delivery;
- correct message order.

Depending on the project and domain we are talking about, there can be a lot of different problems that can come up. Nevertheless, we always have independent services that communicate with each other. Therefore, there will always be a message, and there will always be problems that we need to solve. Otherwise, we can lose consistency across our microservices.

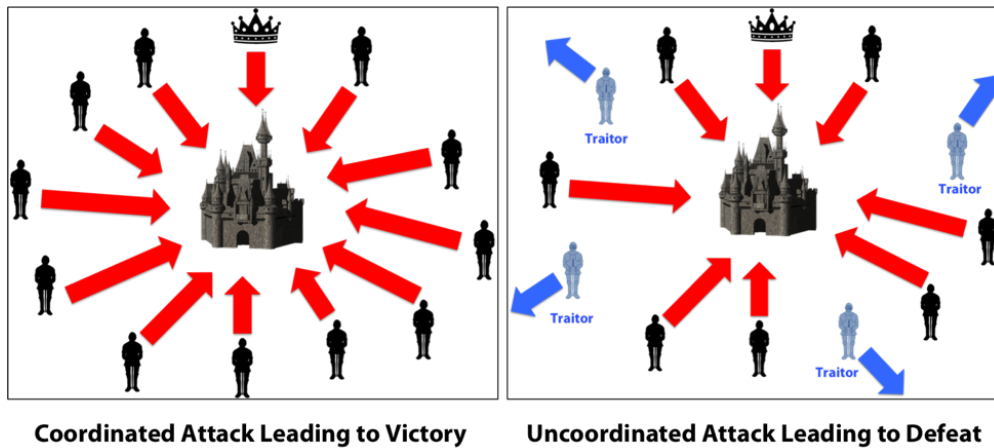
The problem of solving data consistency in Microservice Architecture will be the main topic of this thesis work. This thesis work will examine all possible solutions we can use and determine the more suitable situations for each of them.

## 2.3 The problem of data consistency

In this topic, we will delve into the problem of data consistency, which will be the main focus for the rest of this thesis work. We will not talk about the possible solution yet, but we will expand on the problem more to fully cover it.



■ Figure 2.2 Two generals paradox illustration from [1]



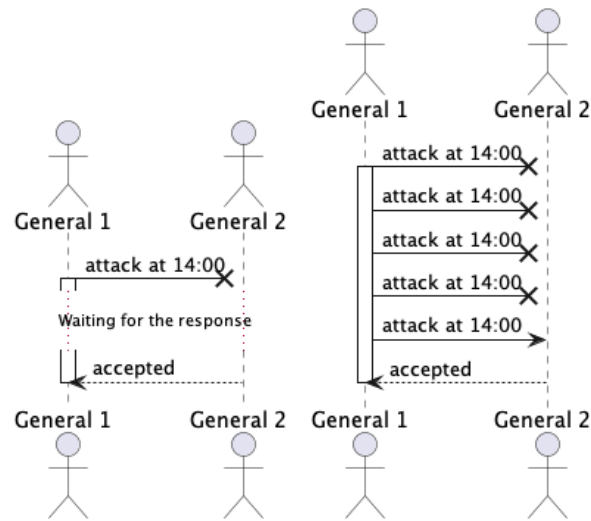
■ Figure 2.3 Byzantine generals attack from [3]

Firstly, let's look at a high-level simple example: "Two Generals Paradox" (see 2.2). There are two generals that want to capture the castle, but the only way to do it is to attack simultaneously. Otherwise — they will lose. They can communicate with each other via the communication channel, which can be interrupted by the enemy. Therefore there is no consistency or robustness. This problem was first mentioned with this name by Jim Gray in 1978 in the book "Notes on Data Base Operating Systems." [2]

Possible situations that can happen:

- The first general sends a message, but it does not arrive;
- The first general sends a message, and it arrives successfully, but there is no response;
- The first general sends a message, but the enemy changed it.

The two generals paradox does not have a simple solution. In fact, it does not have a solution at all. Let us look at this problem from the perspective of the first general. (see 2.4)



■ **Figure 2.4** Generals approaches in "Two Generals Paradox"

The first approach he can use is to send the message to the second one and wait for the response. In this message, he will choose the moment they both need to attack. In this case, he can wait forever, as the message can be lost.

The second option he can choose is to send messages until the second one responds. From the perspective of the second one, he has the same two options: try to send once and send many messages until a successful response. This perspective goes around forever, and they are not sure when to attack in both scenarios.

This paradox is a common problem in Microservices Architecture. Generals are our independent microservices that are trying to communicate with each other and cannot rely on a communication channel.

Let us look at our example Book store project from the business perspective and simulate the same situations that were mentioned:

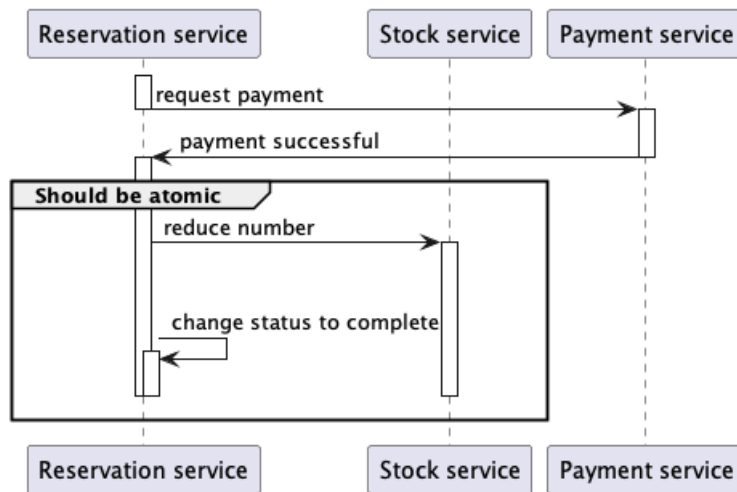
- The client approved the payment, but the approval was lost (money was not withdrawn from the account);
- The client successfully paid for the book, but there is no confirmation from the external payment service (money was withdrawn from the account);
- The client tried to pay for the book, but the hacker tried to change the reserved item. Therefore the message was corrupted by the hacker.

The roots of these problems that we can face fall down to the communication channel, which is unreliable by its design. There are two main problems that make the basis for all mentioned problems:

- Exactly once delivery;
- Messages order.

In other words, we never knew the current state of the other service we are trying to reach. Same as this other service does not know with certainty our state.

Moreover, what if there are multiple generals, and some of them can be a traitor? It brings us to a more common problem: "The Byzantine Generals' Problem." This formulation of this problem was presented in the 1982 paper, "The Byzantine Generals' Problem" [4]. (see 2.3)



■ **Figure 2.5** Reservation payment sequence diagram

This problem is a common problem in the blockchain domain. However, it is already outside of the concern of this thesis work. Therefore we will not be going to expand more than this.

## 2.4 Terminology

We are getting to more technical discussions, and at this point, we need to get along with important terminology.

### 2.4.1 Data consistency

In Microservices Architecture, we have a lot of independent services that can have their own data storage. Nevertheless, their data are still a part of the data state of the whole infrastructure. Therefore the data can depend on each other or be distributed across several independent instances. In this case, when we are talking about data consistency, we are talking about the moment when the data across all microservices are in a consistent state.

### 2.4.2 Distributed transaction

Data consistency in Microservice architecture is quite an abstract term. The causes of inconsistency could be different, but the problem itself happens on a communication level. In the case of microservices, this communication is performed via distributed transactions ( see2.5).

In the case of a monolithic application, transactions happen within one application. Therefore we can provide ACID characteristics:

- Atomicity;
- Consistency;
- Isolation;
- Durability.

### Atomicity

Within one transaction, we can access multiple sources and perform many operations with these sources. Atomicity ensures that either all of these operations succeeded or neither of them. For example, within our Book store, we want to mark user orders as completed and, at the same time, decrease the number of available books. Atomicity, in this case, will ensure that these two changes are either performed or neither.

### Consistency

The ACID transaction should keep resources in a consistent state after the transaction, which is ensured by consistency. In the case of our example applications, it means that when we try to decrease the stock amount after the successful payment, we should reach a consistent state without violating some constraints, such as falling below 0.

### Isolation

Microservices infrastructure is a complex architecture, and there are many simultaneous transactions. These transactions can require, in some cases same resources. Isolation ensures that other transactions will not interfere with one transaction. For example, when the first client buys a book and we are trying to decrease the number of books in the stock, some other transaction does not change this value between changing the order status and decreasing the stock amount.

### Durability

The last characteristic of the ACID is durability, which is an important characteristic that ensures that changes performed during the transaction will stay permanently. In the case of our example projects, it means that even if the database gets into the failure state, the data remains.

Ideally, each distributed transaction should transition from one consistent state to another. In Microservices Architecture, we aim to reach all ACID characteristics, but it is not as simple as in monolithic applications.

## 2.5 Evaluation approaches

Analysis of this thesis work will be dedicated to finding, analyzing, and comparing modern solutions to the problem of data consistency in Microservices Architecture. Therefore we need some measurement tactics to compare these solutions.

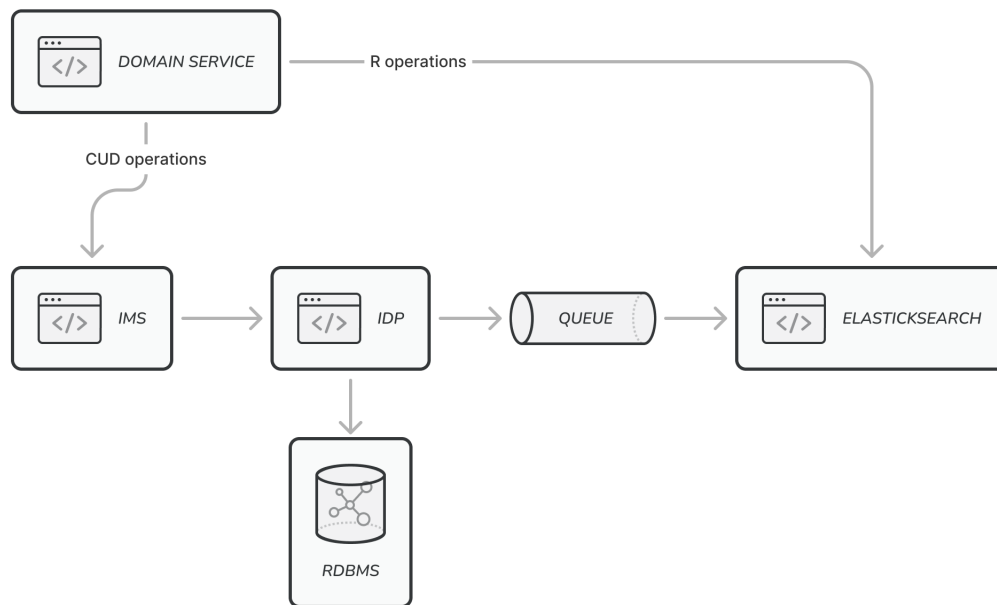
### 2.5.1 ACID

The main goal of all these patterns is to provide some ACID characteristics, but it is not possible to provide them all in Microservices architecture simultaneously. Therefore it is important to determine which ones were provided and which were not. For example, the main goal of a pattern or approach could be providing atomicity for distributed transactions, and therefore other ACID characteristics could not be provided at all. Thus, the primary evaluation of analyzed approaches will be conducted based on ACID characteristics with the applied studied approach.

### 2.5.2 Required effort

Every solution has its own price companies pay to make it work. It could be an additional microservice, changes to the data contract, unique ids, and so on. Nevertheless, the price is always there. Depending on the type of changes the team needs to apply, it can impose restrictions on the project, or it could be hard to maintain. It is crucial to mention as there is always a budget, deadlines, or developers' skills, that can limit us.





■ **Figure 2.6** "Project alpha" component diagram

## 2.6 Provided project

The next chapter will analyze possible solutions for the project that is provided for this thesis work. This section will be dedicated to the description of this project. Based on this information, the analysis of patterns can be slightly limited and narrowed to be helpful in the following chapters.

The project will be described from a high-level perspective with more detailed information about important parts according to the concern of this thesis work. This provided project will be called "Project Alpha" for the rest of this thesis work.

The application has several components, but not all of them will be subjects of concern for this thesis work (see 2.6). The application has its main microservice, which is responsible for the main domain logic. Let us call it a Domain microservice (DM). In case this microservice needs some info regarding the **User** entity, it needs to communicate with the **IMS**<sup>1</sup>, which is responsible for the **User** domain business logic. However, it does not store these data and does not manage access rights and roles. It is the responsibility of the separate microservice, **IDP**<sup>2</sup>. This microservice is based on Keycloak and works directly with PostgreSQL **RDBMS**<sup>3</sup> to store user data.

Nevertheless, the application's domain logic requires a quick full-text search of the user data. Therefore, an additional microservice can handle these requests better, a separate Elasticsearch instance. The source of truth for user data is the **RDBMS** instance.

<sup>1</sup>Identity Management Service

<sup>2</sup>Identity Provider

<sup>3</sup>Relational Database Management System

## 2.7 Keycloak

Keycloak is an open-source identity and access management tool. This tool can be simply integrated into existing projects to delegate Identity provider functionality. Being quite a young project initially released in 2014, it already has a community behind it and is relatively popular nowadays.

This tool can be deployed as a fully standalone project, for example, via the Docker image. By taking this approach, there are no noticeable features that are sacrificed. Nevertheless, if the project requires some custom logic, Keycloak can be integrated as a dependency, which gives the project all functionalities that have a standalone one. This tool has good documentation and a strong community that provides a lot of helpful tutorials and descriptions, so there is no point in going into small details in this thesis work.

## 2.8 Apache Kafka

Apache Kafka is a distributed event streaming platform. Nowadays, it is a streaming platform, not by chance. It is more than simply a message broker. It is a highly horizontally scalable, fault-tolerant distributed system with many possibilities to customize it and many additional functionalities developers can benefit from. One of the best short descriptions of the Apache Kafka platform is in the book by the Confluent:

*We've come to think of Kafka as a streaming platform: a system that lets you publish and subscribe to streams of data, store them and process them, and that is exactly what Apache Kafka is built to be. Getting used to this way of thinking about data might be a little different than what you're used to, but it turns out to be an incredibly powerful abstraction for building applications and architectures. Kafka is often compared to a couple of existing technology categories: enterprise messaging systems, big data systems like Hadoop, and data integration or ETL tools. Each of these comparisons has some validity but also falls a little short.*

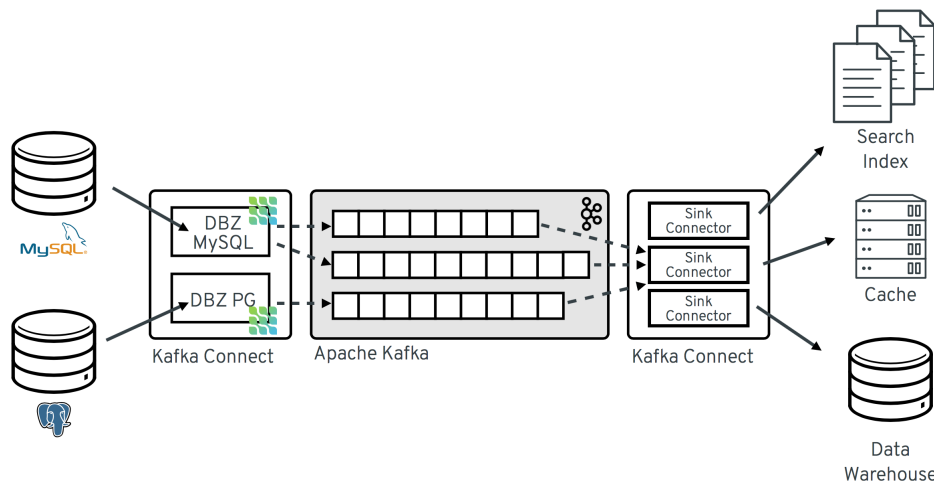
[5]

## 2.9 Elasticsearch

Elasticsearch is a distributed search and analytics engine. It is a part of the Elastic Stack. The data itself is stored in a document-oriented database, which means that data is stored as flexible documents under specific indexes. For example, the index could be a user, and the document would be, in that case, a concrete user. However, this also means that Elasticsearch does not have the same transactions as, for example, PostgreSQL. Nevertheless, Elasticsearch has its mechanisms to ensure some level of consistency for read/write operations.

## 2.10 PgSync

PgSync is a lightweight tool for syncing data from PostgreSQL to Elasticsearch[6]. It works based on a change data capture approach, where the source of truth is the RDBMS. The tool works as a standalone application and provides many possibilities for the configurations, such as the ability to specify tables the tool should work with or the ability to specify the columns that should be transferred to Elasticsearch.



■ **Figure 2.7** Example of the pipeline built with the Debezium from official documentation [7]

## 2.11 Logstash

Logstash is a data processing pipeline that is part of Elastic Stack. The pipeline supports three types of adapters:

- inputs - ingest data to the pipeline;
- filters - parse or transform data inside the pipeline;
- outputs - route data to different destinations, such as Elasticsearch.

## 2.12 Debezium

Debezium is an open-source distributed platform for change data capture. It allows streaming all changes made to the given tables in the RDBMS to the message broker through the application (see 2.7). This tool implements the observer pattern on the level of interaction with the database, as it does not interfere with the transaction. It looks at the database Transaction logs and emits data only detecting new records.

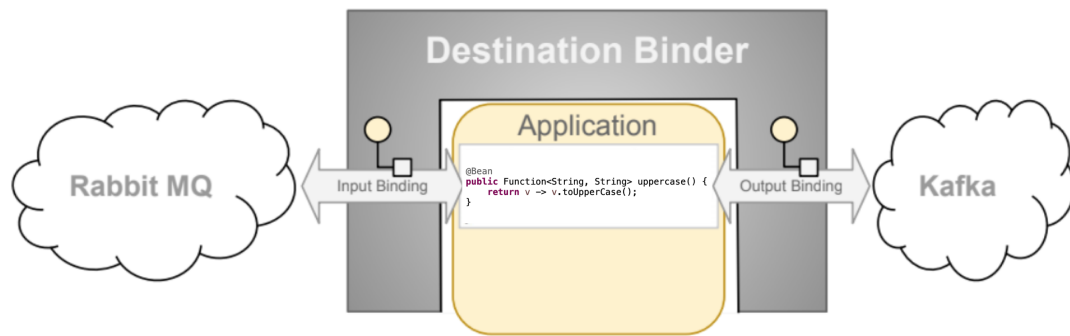
## 2.13 Spring Cloud Stream

Spring Cloud Stream (SCS) is a framework for building highly scalable event-driven microservices built around some message broker, such as Apache Kafka or RabbitMQ. (see 2.8) This message broker serves as a binder within an SCS infrastructure.

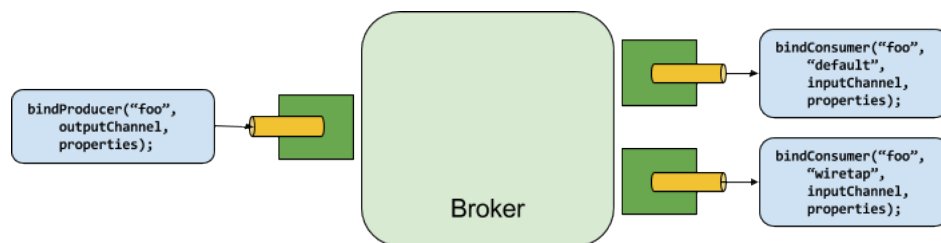
There can be many scenarios for using this framework. One example of such a use case might be a complex migration of data from a source of truth to a secondary resource. The final infrastructure is very scalable and flexible. Therefore, it is relatively simple to add some processing logic to change the data that flows through the binder.

This framework can be something new to some developers, but it is built with the same best practices in mind as other Spring frameworks, so the adaptation period is relatively short. SCS is also available through the Spring Initializr, the same as Spring Boot.

SCS uses Spring Cloud Functions to implement logic that handles binder data flow. It is built with the functional programming model to define each message handler, such as a



■ **Figure 2.8** Spring Cloud Stream Application architecture from official documentation [8]



■ **Figure 2.9** Spring Cloud Stream relationship between Producer and Consumer from official documentation [8]

Producer or Consumer. These functions' implementation can be expressed as beans of type `java.util.function.Function` (same for the `Supplier` and `Consumer`) (see 2.9):

- `Supplier` - serves as a message source;
- `Consumer` - serves a message receiver;
- `Function` - a more generic function that can serve as a message processor for message transformation.

## 2.14 Locust

Locust is a scriptable and scalable performance testing tool [9]. The individual tasks can be defined as regular Python code. It allows developers to configure tests as they please, almost without limitation. The process of running tests is provided via the Web-based UI, which allows adding some final configurations and starting or stopping the testing. Additionally, it provides a real-time demonstration of how the tests are running.

## 2.15 Programming language

Provided project is written and maintained mainly in Java. However, there is an intention to slowly migrate the whole code base to Kotlin. This language brings some risks to the project, but it is not much different from Java, and, most importantly, it is the desired language, and it is welcomed by the tea.

Therefore, in this thesis work, Kotlin will be used for main implementation purposes, and all technologies used or described will be mainly for the JVM or Kotlin projects.

Kotlin was created as an alternative to Java and solves some of its problems. For example, Kotlin solves the problem of the widespread use of `null`, also known as **The Billion Dollar Mistake**, and the problems associated with it. Java itself has no support for `not-null` variables, but Kotlin has such support natively, in the form of separating the `nullable` type with `?` operator. For a complete introduction to the Kotlin language, it is recommended to visit the official website, where you can find a detailed tutorial on how to use the language.[10]

## 2.16 Git approach

The reason for introducing the conventions is the need to make the branch graph more transparent. The implementation part of this thesis work will be an example application, but it will be used to determine how to change the original provided project. Therefore, the code base needs to be documented and readable for the developer, including the commit tree. Understanding the various changes made during the software development process is aided by clear descriptions of each change. The introduction of a uniform format for each `commit` within the project helps to achieve such a result.

To provide consistency and clarity for each `commit` was defined a set of rules, inspired by Angular Git rules [11].

The convention was introduced at the beginning of the author's work on the implementation of this thesis and did not change during the development process. Figure 2.10 shows a piece of the GitHub branch graph after the rules were introduced. The structure for `commit` consists of five parts:









- `type`;
- `scope`;

```

1 <type>(<scope>): <subject>
2 <BLANK LINE>
3 <body>
4 <BLANK LINE>
5 <footer>

```

■ **Listing 2.1** Template for the Git commit

<b>feat(idp adapter): add exceptions handler</b> Jaroslav Kolodka committed on Sep 29	 <a href="#">2a12deb</a> 
<b>feat(idp adapter): add service implementation</b> Jaroslav Kolodka committed on Sep 29	 <a href="#">438989f</a> 
<b>feat(idp adapter): add adapter configuration</b> Jaroslav Kolodka committed on Sep 29	 <a href="#">6fd4968</a> 
<b>compile(gradle): add dependencies for the logging</b> Jaroslav Kolodka committed on Sep 29	 <a href="#">3a2755c</a> 

■ **Figure 2.10** Example of GitHub commit tree with applied rules for the format

- subject;
- body;
- footer.

## Type

The type defines the type of changes that was applied to the project with the given `commit`. There is a set of different values that can be set:

- build** – changes in build process or external dependencies;
- ci** – changes to the CI;
- docs** – changes in documentation;
- feat** – new feature;
- fix** – feature fix;
- perf** – changes in performance;
- refactor** – changes, that does not bring new feature;
- style** – changes to the code style;
- test** – new tests.

## Scope

The scope defines the specific files or packages that have been changed. The scope is not mandatory, but within this thesis, the scope will be given for each `commit`. If changes have been made to several different files or packages at the same time, they should be listed as a comma-separated list with items in round brackets.

### Subject

The subject is a brief description of the changes made. Changes are to be separated from the type by a colon. If `commit` contains several changes, they should be separated by a colon. The first letter of each change must be lowercase. The last change does not end with a period.

### Body

The body is for a more detailed description of the changes made. This section is not mandatory but is recommended if the changes made are not understood after reading the subject or the reason for the change is not clear.

### Footer

The footer is for defining breaking changes and defining dependencies on specific requirements within GitHub or GitLab. The implementation part of this thesis work will not use this part, as the process of implementation will not be divided into separate issues, and there will be no links to the specific issue.

## 2.17 Build automation tool

As a build automation tool will be used Gradle in this thesis work. There is a popular alternative – Maven, but Gradle is more modern and convenient. Configuration is done in Groove or directly in Kotlin. Also, Gradle provides the ability to define custom tasks executable via the command line, which will then be used for analysis and running tests. More detailed information on comparing Gradle and Maven tools can be found at [12] and [13].

## 2.18 Research summary

Firstly, in this chapter reader was introduced to the problematics of Microservices Architecture, including the main topic for this thesis work, which is data consistency.

Then, were introduced technologies that will be used across subsequent chapters for design, implementation, or test purposes.





## Chapter 3

# Analysis

*This chapter describes, evaluates, and compares different patterns for providing data consistency in Microservices Architecture. These patterns can be found on the internet and are commonly used across the industry for different specific use cases. The problem is determining what these use cases are and what particular brings to the project. This chapter will clarify some aspects of these use cases for each observed pattern.*

### 3.1 Two-phase commit pattern

#### 3.1.1 Description

The main idea behind the two-phase commit (2-PC) pattern is to make distributed transactions atomic. We cannot provide atomicity by default as we send multiple transactions to multiple microservices across our infrastructure. This pattern uses an independent coordinator instance to reach atomicity, which is the decision-maker for these transactions.

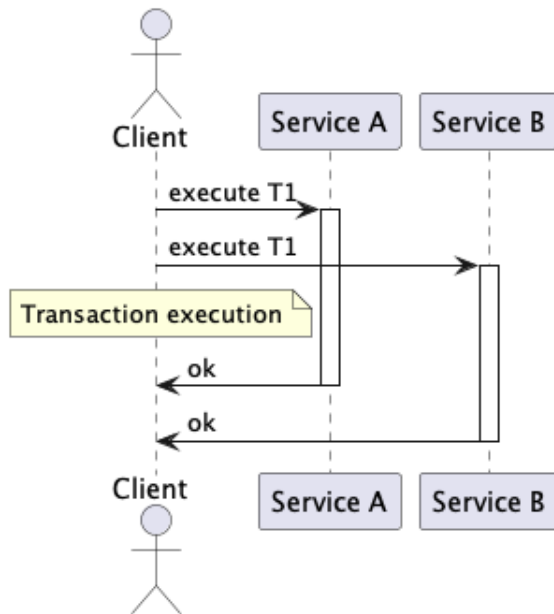
The transaction flow does not differ much from the flow we used to in distributed transactions (see 3.1), but each microservice does not commit transactions immediately after processing them. Each receiver instance only prepares the received transaction for commit. Then this microservice promises the coordinator it can do it and asks permission to commit these changes. At the same time, the coordinator waits for all services that are part of this distributed transaction to ask him for permission. If all instances are ready to commit, the coordinator gives permission to do it, but if something is wrong, the coordinator gives the order to roll back everything, as there will not be a consistent state after all. (see 3.2)

#### 3.1.2 Use-case

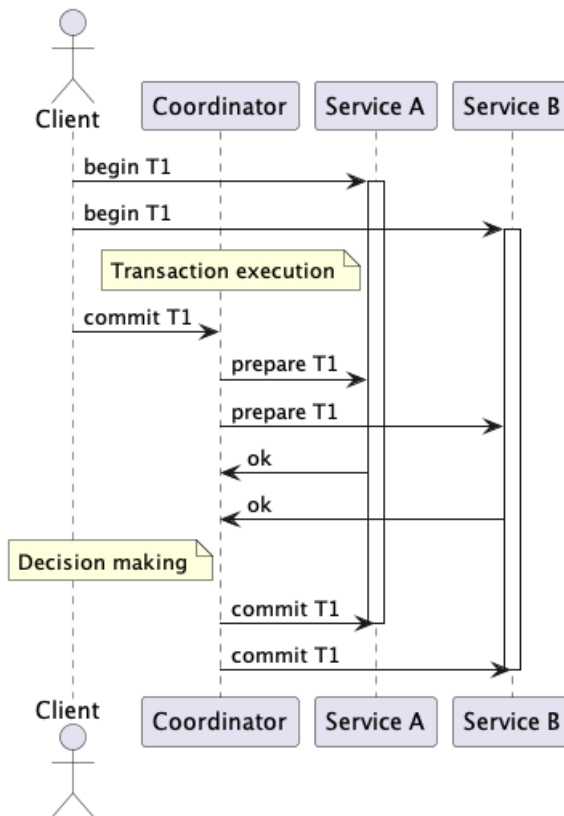
It can seem that a two-phase commit is a way to go for solving all of the challenges developers can face in data consistency, but, as we discussed earlier in the previous section, this approach has many flaws and rely on a lot of existing solutions in the system, that already solves different issues.

The 2-PC is a pattern that is not bound to a specific use case or domain. Let us define some boundaries that developers need to be aware of:

- The fact that distributed transactions should consist of a small number of subtransactions;
- The fact that distributed transactions should not contain subtransaction that is critically slow according to other sub-transactions;



■ Figure 3.1 Distributed transaction sequential diagram



■ Figure 3.2 Two-phase commit sequential diagram

- The need to be able to cover the failure of the coordinator instance without losing data consistency;
- The fact that participating services in distributed transactions cannot undo their "promise" to the coordinator;
- The fact that services should agree on protocols and processes;
- The fact that the final decision of the coordinator still can be lost.

If a developer wants to implement this pattern himself, it could take much effort and, in the end, will still not solve all of the issues. However, this pattern is widely implemented in many projects that are used inside the project. For example, distributed databases like MongoDB use this pattern to store data atomically across partitions, or Kafka uses this pattern to produce messages atomically across multiple partitions.

### 3.1.3 Evaluation

The main goal of the two-phase pattern is to provide **atomicity** of a distributed transaction. Therefore there is an atomicity characteristic, but with a few preconditions:

- Microservices participating in the distributed transactions need to have ACID characteristics;
- Failure of the coordinator instance needs to be managed as the protocol works in a blocking way.

The pattern is not designed to solve **consistency** problem. The precondition for using this protocol is to use ACID databases or microservices with the same characteristic. Therefore this protocol entirely relies on the fact that consistency will already be provided.

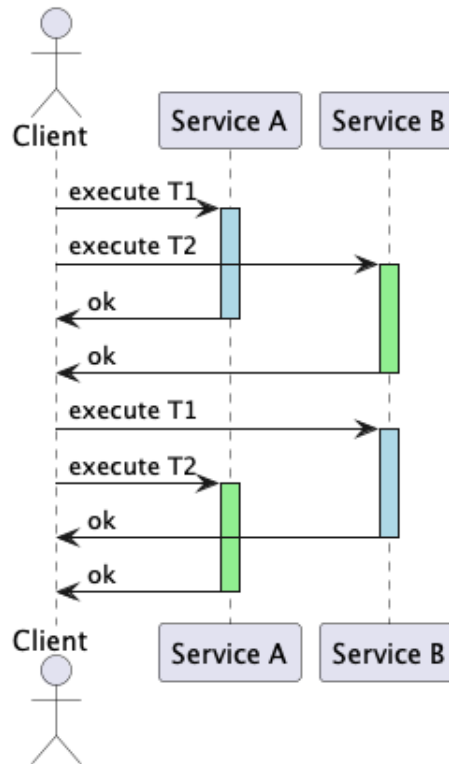
The two-phase commit pattern does not solve **isolation** problem by itself, and, same as consistency, it relies on the fact that all participating microservices have ACID characteristics. Therefore each subtransaction will block the resource until the approval from the coordinator arrives and the transaction is committed or rolled back. Nevertheless, we are still dealing with the race condition between different resources, as there is no provided isolation between them. For example, there are two transactions: T1 and T2. They require the same two resources: A and B. The T1 change the state of resource A, and T2 changes the state of resource B. Then they go to the leftover resources. As for each resource, there are provided isolations by preconditions, but from a higher perspective, there is still interference between these transactions. (see 3.3)

Same as consistency, **durability** is not the primary concern of the two-phase commit pattern. It should be provided by each microservice, as it is giving its "promise" to the coordinator that the received subtransaction can be successfully committed.

As for the complexity, this pattern is not very pleasant, as it does require adding additional logic to the services. Moreover, a company needs to maintain and probably implement coordinator instances, and it seems like much effort only to provide atomicity for transactions.

ACID			
Atomicity	Consistency	Isolation	Durability
+	+	-	+

■ **Table 3.1** ACID evaluation for 2-PC pattern



■ **Figure 3.3** Distributed transactions interference sequential diagram

### 3.1.4 Implementations

In this section, we will discuss the different implementations of the two-phase commit pattern. This pattern can be found in databases too, but we will mainly talk about specifications and implementation for the microservices architecture. This pattern is a well-known pattern. Therefore we will only cover some existing implementations or specifications and also will cover implementations that can be used in the next chapter of this work (see 4).

#### 3.1.4.1 XA transactions

XA transactions is a specification of the two-phase commit pattern published in 1991 by the X/Open group (Open Group for Unix Systems). This specification describes how to manage distributed transactions by providing atomicity within one transaction despite the number of independent subtransactions. The specification is fully based on the pattern of a "two-phase commit."

For the most part, this specification can be found in different databases, such as Oracle or PostgreSQL, but it also applies to any distributed system.

According to the original documentation, the main structure can be described as follows:

*Processing (DTP) model envisages three software components:*

- *An application program (AP) defines transaction boundaries and specifies actions that constitute a transaction.*
- *Resource managers (RMs, such as databases or file access systems) provide access to shared resources.*

- A separate component called a transaction manager (TM) assigns identifiers to transactions, monitors their progress and takes responsibility for transaction completion and failure recovery.

[14]

This specification does not give a working solution that can be used, and it only describes how developers can implement it by themselves for their architecture correctly. In this topic, we only describe this specification at a high level. If you want to delve into the specification more, you can follow the link to the original specification file ([14]).

### 3.1.4.2 Jakarta Transactions

Our "Project Alpha" is written in Kotlin and Java languages, which means that we are talking about a Java environment. Therefore, it leads to choosing Jakarta Transactions as an example of 2-PC pattern implementation. This implementation could also be found by another, more common name, JTA (Java Transaction API). This former name was changed along with renaming Java EE to Jakarta.

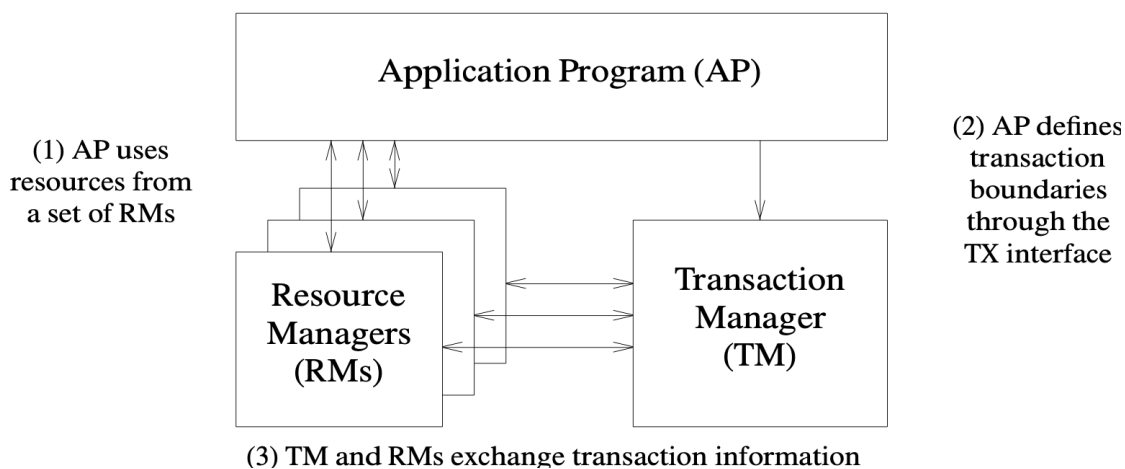
There are many JTA releases, but in this section, we will talk about its latest release as of today, which is Jakarta Transactions 2.0 (see [16]). According to this document, the JTA package consists of two parts, which are:

- A high-level application interface that allows a transactional application to demarcate transaction boundaries;
- A high-level transaction manager interface allows an application server to control transaction boundary demarcation for an application being managed by the application server.

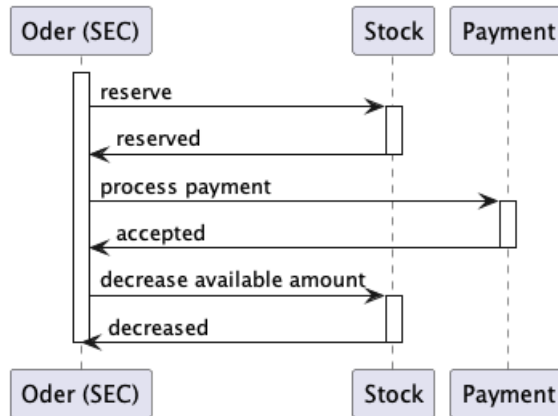
## 3.2 Saga pattern

### 3.2.1 Description

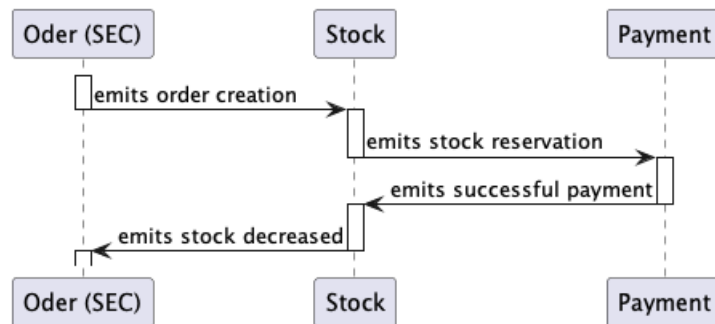
In Microservices architecture, distributed transactions seem like the obvious choice. The main problem behind it is to make it ACID. However, what if we looked at this situation from the other angle? What if we avoided distributed transactions, as it is tough to make ACID? Therefore, we can divide these distributed transactions into a sequence of local transactions that we can



■ **Figure 3.4** TM and RMs exchange transaction information from [15]



■ **Figure 3.5** Orchestration Saga flow



■ **Figure 3.6** Choreography Saga flow

manage more easily. This is the main thought behind the Saga pattern. However, it is not as easy. This pattern has several requirements that we need to provide.

An essential requirement of the Saga pattern is to implement compensating transactions (CT) for every transaction. This CT should roll back performed local transactions, given that changes are already committed. For example, for our example Book store project, in case the client tried to pay for the book, but the payment for rejected, our compensating transaction would be marking the reservation with info that the payment was rejected.

The saga pattern can be implemented in two ways: choreography and orchestration. In the choreography Saga, each microservice is responsible for publishing the event for the next microservice. In an orchestration Saga, a single component is responsible for the status of all transactions. This orchestration component will be responsible for the rollback process if any transaction fails:

**Orchestration Saga** The core of the Orchestration Saga transaction flow is the Saga Execution Coordinator (SEC). It contains the persistent sequence of events of a particular flow. The eventually consistent state with this pattern is reachable with the help of this component. Based on these logs, rollback operations can be performed to the overall consistent state in case of failure (see 3.5);

**Choreography Saga** There is no SEC instance, and therefore there is no potential flow bottleneck. However, the compensation flow is still necessary, and each microservice that consumes, processes, and publishes events should be responsible for the potential compensation operations (see 3.6).

### 3.2.2 Use-case

The saga pattern is suitable for projects where distributed transactions become a problem that is already hard to maintain. However, the application should support the compensating transaction logic.

### 3.2.3 Evaluation

The Saga pattern provides **atomicity** for the transaction, as either all local transactions will be performed or neither of them with the help of compensating transactions and transaction logs.

As for the **consistency**, Saga is designed to solve, but it does not give you immediate consistency. As our distributed transactions are deconstructed into independent local transactions, there will be only eventual consistency for the overall system state.

The main problem of the Saga pattern is the lack of **isolation**. Therefore, to reach the full ACID, there is a need to apply some additional measures alongside this pattern.

This pattern does not ensure durability by itself, but it provides durability transitively from the underlying writes during the transaction. Therefore all writes should be durable.

ACID			
Atomicity	Consistency	Isolation	Durability
+	+	-	+

■ **Table 3.2** ACID evaluation for Saga pattern

### 3.2.4 Implementations

The Saga pattern is extremely popular in Microservices architecture, and consequentially there are many solutions we can use. As our "Project Alpha" uses JVM and Spring Framework, let us look at some examples we can use with these technologies.

As for the Choreography Saga pattern, we can use Axon, which is a lightweight framework for helping build scalable microservice applications around three core concepts:

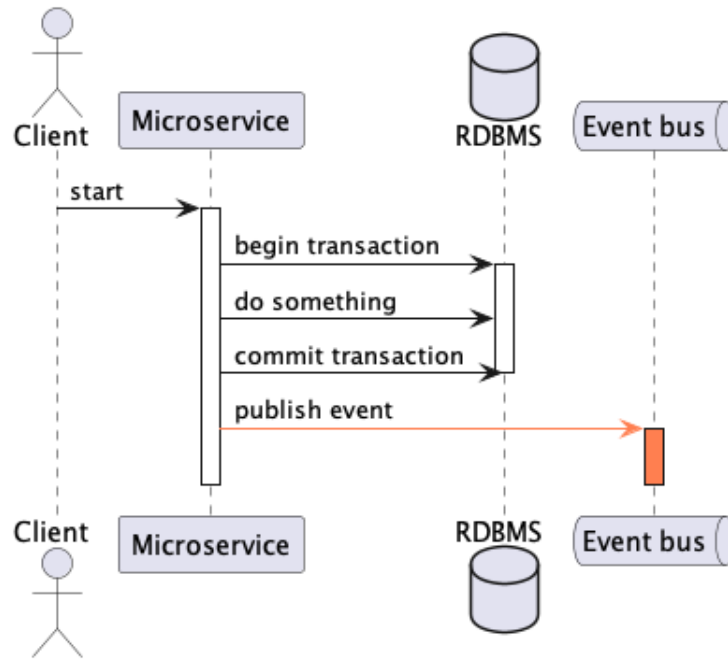
- CQRS;
- Event Sourcing;
- Domain Driven Design.

As for the Orchestration Saga pattern, we can use Apache Camel, an open-source integration framework based on Enterprise Integration Patterns. This framework has a Saga EIP, with which developers can define a series of related actions in the Camel route that should be either all successfully executed or compensated.

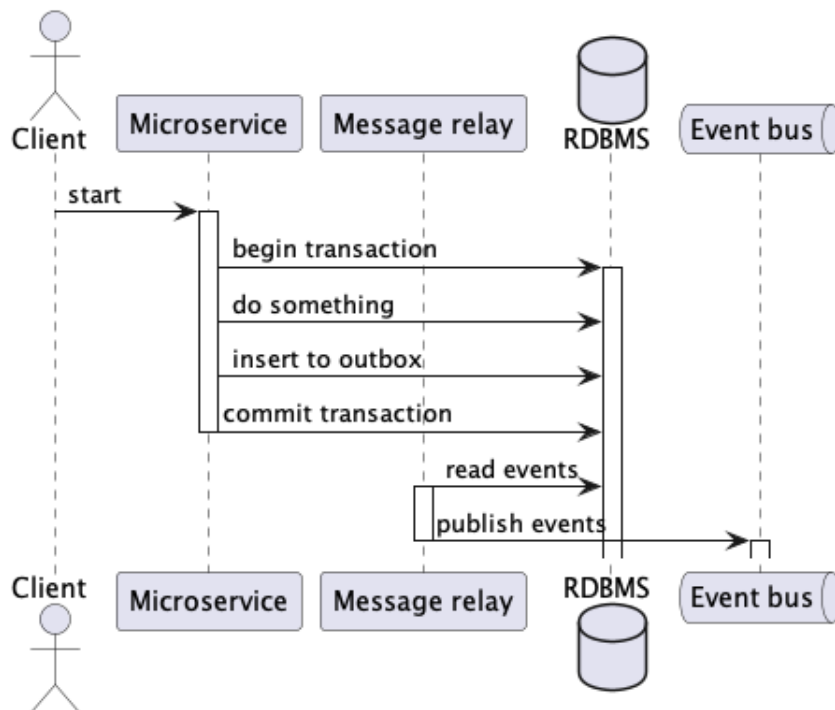
## 3.3 Transactional outbox pattern

### 3.3.1 Description

The main idea behind this pattern is the solution for its use case: sending the message during the local transaction (see 3.7). To eliminate the process of sending as a point of failure. The service, instead of sending the message, saves the message to separate local storage (Outbox table). This operation can be reliable, as it is a part of the local transactions. Then the separate service, Message Relay, reads this message and publishes it.



■ Figure 3.7 The process of publishing events



■ Figure 3.8 Transactional outbox pattern sequential diagram



### 3.3.2 Use-case

This pattern, unlike the previous ones, has a concrete use case, which is the situation when there is a need to send a message in the middle of a transaction.

### 3.3.3 Evaluation

This pattern does not provide ACID characteristics to the distributed transaction for every use case. However, it solves a specific scenario that allows keeping local transaction ACID and sending messages to external services simultaneously.

### 3.3.4 Implementations

To implement this pattern, developers do not need specific tools. However, there is a need to implement additional event storage locally within a service. It could be anything according to project needs. For example, the most obvious choice could be an additional table in the RDMS alongside the primary tables. In this case, adding events to this "buffer" table can still be ACID and work with your regular tables. Afterward, an event streaming platform will constantly pull data from this outbox table.

There are two popular patterns for implementing the Message relay: the Transaction log tailing pattern and the Polling publisher pattern. The rest of the path of these messages is not under the concern of this pattern, but as long as messages reliably reach the streaming platform, the this pattern is completed.

The Polling publisher pattern is a relatively simple to implement Message relay (MR) for the Transactional outbox patterns. Developers might face issues with this approach if they use NoSQL databases for the outbox table, as it might not be efficient to pull whole documents from the databases, but as long as they use RDMS, this approach is helpful.

As for the complexity, this pattern is an additional benefit, as developers do not have to spend much time learning new information. The main idea of this pattern is that MR polls unpublished messages at a fixed rate, and as soon as messages reach their destination topic or channels (for example, in Apache Kafka), MR deletes these messages from the table. In this scenario, the important thing to know is that we are achieving "at least once" delivery, and we need to manage duplicities.

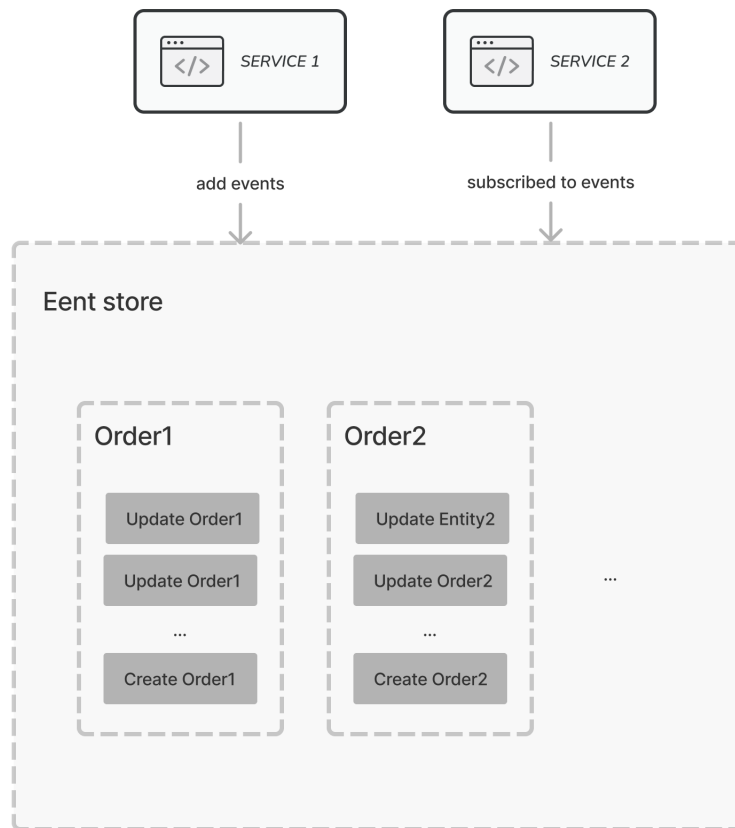
Another pattern to implement the Message relay is Transactional log tailing (see 3.8). We are not working with the table directly but with its logs. As soon as some changes are committed, a new record adds to the logs. Afterward, based on this record, we can create a new message for our message broker. One concrete implementation for this pattern could be Change Data Capture (CDC) with Debezium or Kafka-Connect.

## 3.4 Event sourcing

### 3.4.1 Description

The communication between microservices consists of events instead of common messages. The same approach applies to data storage as well. Instead of storing the records in the Database, data are stored as an event sequence for each record. Adding a new event is one operation, and therefore it is atomic.

The Event sourcing pattern is mostly used alongside the CQRS pattern, which will be described in the next section. This combination means that retrieving the entity's current state is used differently. If some microservice wants to retrieve this state, it reconstructs the entity based on the given sequence of events. The event store serves as data storage but, at the same time, serves as a Message broker. Therefore microservice that retrieves data can be subscribed to the event store (see 3.9)



■ **Figure 3.9** The Event sourcing diagram

### 3.4.2 Use-case

The Event sourcing is only applicable for some use-case, as the business logic should be built around the events. For example, it could be an online store with the main domain as `Order`. In that case, there will be a lot of different orders with a short lifetime.

Using the CQRS pattern, writing and reading logic can be split into separate microservices. For example, the `Order` service will add events to the event store, and the `User` service will be subscribed to it to show the user's orders on the user profile page.

### 3.4.3 Evaluation

This pattern requires a specific use case and a different approach to the development. Nevertheless, it simplifies the application flow and brings additional benefits. For example, the event store allows access to audit records and reconstruct historical data, as there is no data loss while adding new events.

### 3.4.4 Implementations

To implement this framework, developers can use `Eventuate`. It is not the only solution that can be used, but it is a popular and proven solution, so it would be a good example of this kind of framework. This framework consists of two main parts:

**Eventuate Tram** an adapter for the microservice that uses an Event sourcing pattern, allowing developers to write their code as close to the regular `Repositories` as possible.

**Eventuate Local** the implementation of Event sourcing pattern.

## 3.5 CQRS pattern

### 3.5.1 Description

CQRS is a pattern for splitting the read-and-write logic and model, which stands for Command Query Responsibility Segregation. Commands stand for the requests changing data, and Queries stand for the requests that retrieve data.

### 3.5.2 Use-case

This pattern is due to various reasons. For one, there could be different logic behind these operations. For example, the write operations need to be close to `ACID` and fully `ACID`, but the read operations should have full-text search capabilities that the same data storage or service would not provide.

Alternatively, it could be a difference in the load that requires separating this logic for different scalability. This pattern is often used alongside the Event sourcing pattern.

### 3.5.3 Evaluation

This pattern could bring many benefits if it is used when it is needed. However, it can bring a lot of unnecessary complications to the code when it is used without a need for it.

## **3.6** Back to the monolith

### **3.6.1** Description

In some rare cases, teams are getting to the Microservices architecture immediately without even understanding what issues they will face. It is essential to understand that Microservices architecture is not an all-purpose solution that fits any project and business. Sometimes, it is better to keep macroservices instead of several microservices and deal with the distributed transaction. Good architecture can help solve the issue by itself without returning to monolithic architecture.

### **3.6.2** Use-case

Let us look at our example Book store application. We have three independent microservices. Payment service could be the first service to be changed or multiplied, as we can pay by card, bank transaction, or cryptocurrency. But what about stock service and reservation service? If we do not expect these parts to change significantly and independently, we can keep them as one macroservice and not deal with distributed transactions at all. Sometimes, this decision can keep the project alive according to its budget or real-life deadlines.

## **3.7** Analysis summary

In this section, reader was introduced to the variety of patterns that can be applied to the project to enrich data consistency or provide some ACID characteristics to transactions.

For each pattern was provided an introduction, use case, evaluation and implementation description.



## Chapter 4

# Design

*This chapter will describe the design of the implementation that will solve the problem of consistent data migration from the RDBMS to Elasticsearch. The solution consists of choosing the correct pattern based on the previous chapter and designing the architecture for all participating microservices.*

### 4.1 Application architecture

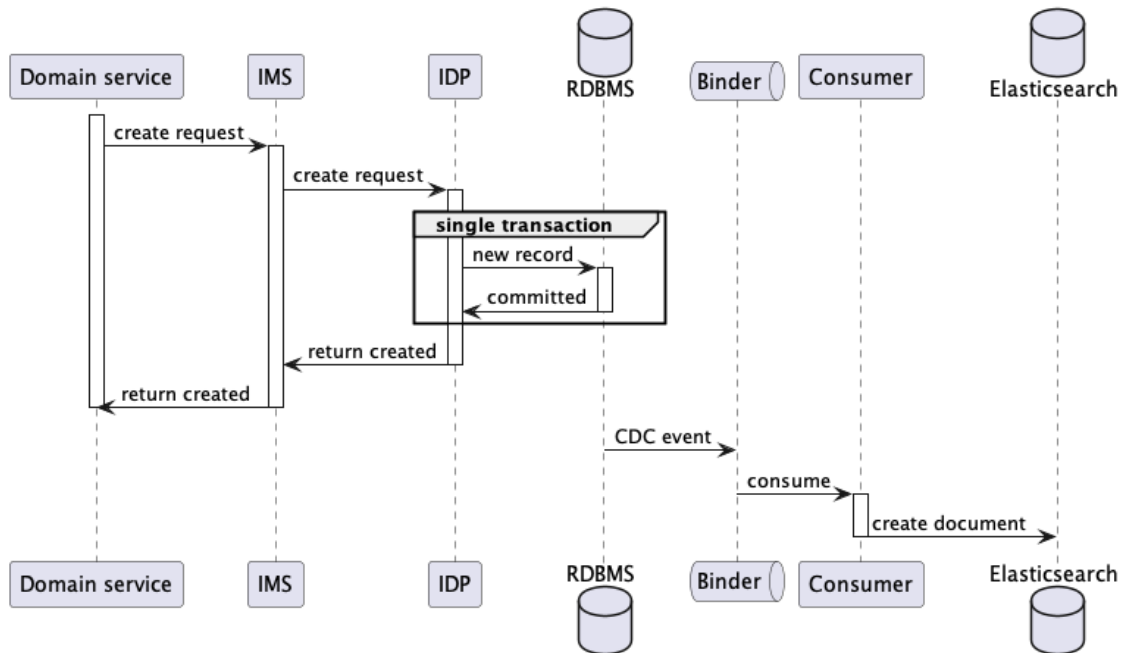
#### 4.1.1 Chosen scope of "Project Alpha"

The original "Project Alpha" is a complex set of microservices that covers a lot of different use cases, including ones that are not in concern of this thesis work. Therefore, only the specific subset of microservices and use cases that are, in fact, relevant for the given scenario of data migration from the RDMBS to Elasticsearch will be used. The components we need are:

- Domain component;
- Identity management service;
- Identity provider service;
- RDMS for the identity provider service;
- Elasticsearch instance;
- Message broker.

For this thesis, work implementation will cover use cases that cover operations with the User instance without business-specific nuances:

- Create user;
- Update user;
- Delete user;
- Find user by id;
- Find user with query.



■ **Figure 4.1** User creation use-case

## 4.1.2 Application use-cases and requirements

### User creation use-case

The first use case is the user creation (see 4.1). Domain service sends the request to the IMS, which covers the business logic of the user instance management, but it does not cover the technical part of how the `User` is stored and what authentication configurations it has. This logic covers IDP, which is a separate service.

IDP in "Project Alpha" is implemented with the use of Keycloak (see 2.7). It has a database to store all users' data that must be migrated to the Elasticsearch instance independently from the RDBMS transaction;

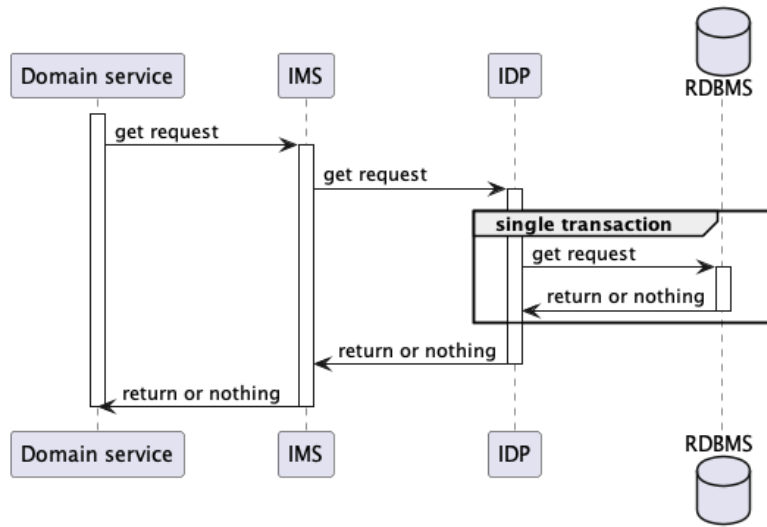
### Get user by id use-case

The second use-case is getting the user by its id. It is the simplest one. IMS sends the request to the IDP with the user's ID that needs to be found. IDP makes a call to the RDBMS and returns the result back to the IMS. (see 4.2);

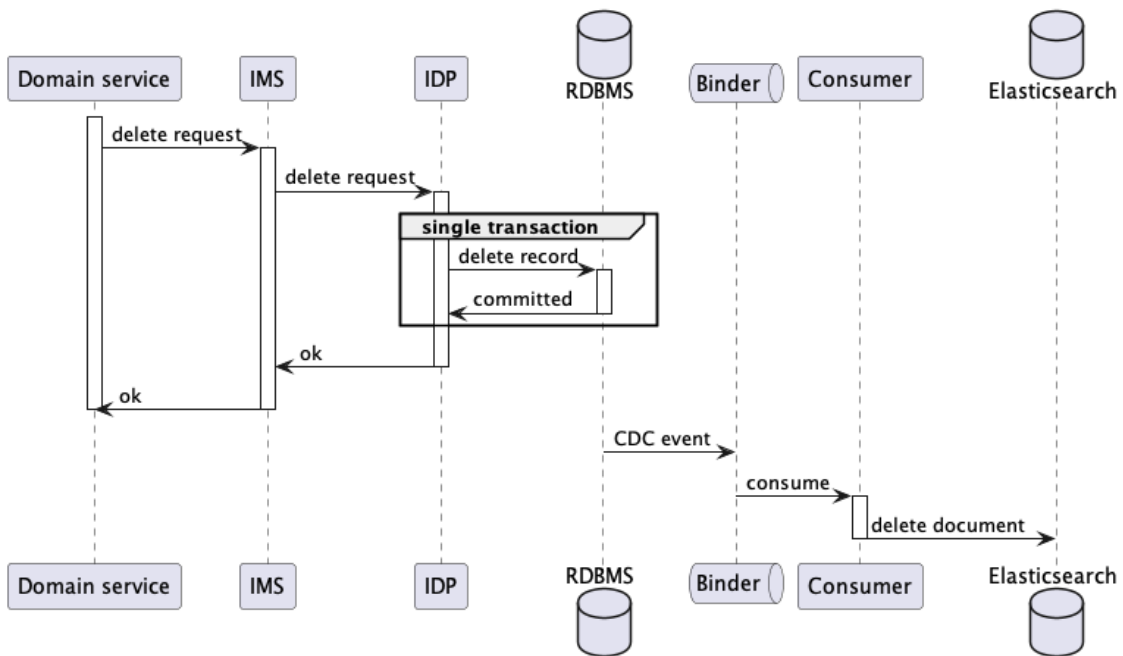
### User delete use-case

The next use-case would be deleting the user by its id. This use-case is similar to create, as we need to perform the deletion of the record in the database within a single transaction and then publish an event to Elasticsearch. The record in the database would be completely removed, and a new record would be added to another table as the log of the performed action.

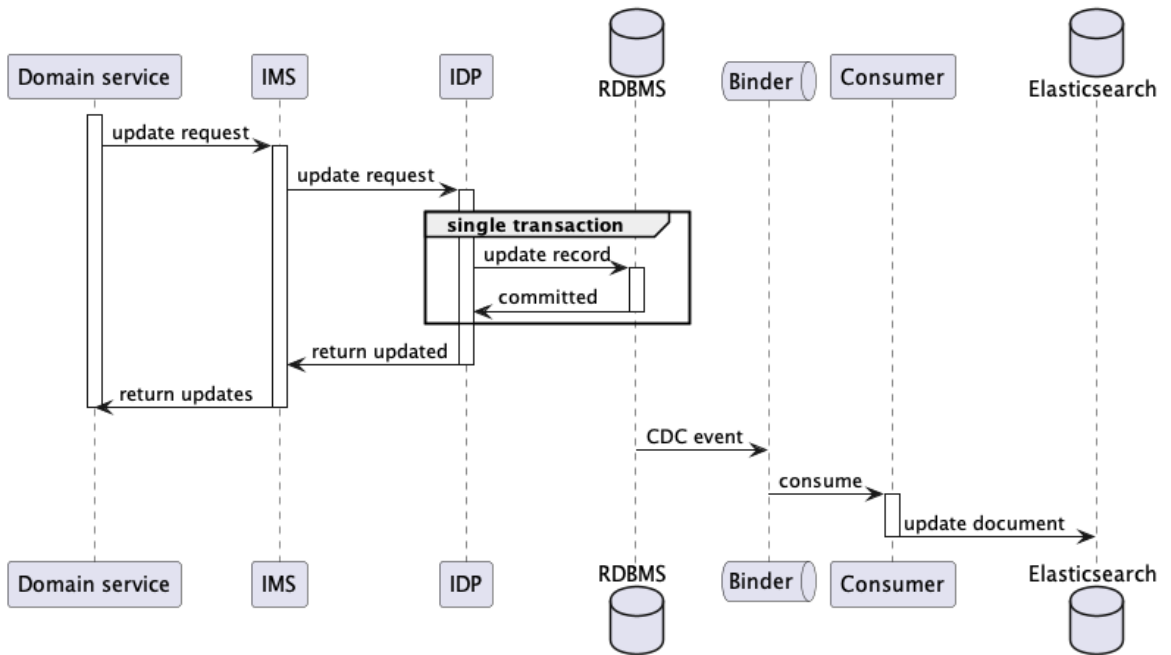
As for Elasticsearch, to preserve the performance and simplicity of the flow, the process of deletion would be implemented as removing all data from the document but keeping the document itself (see 4.3);



■ Figure 4.2 Get user by id use-case



■ Figure 4.3 User delete use-case



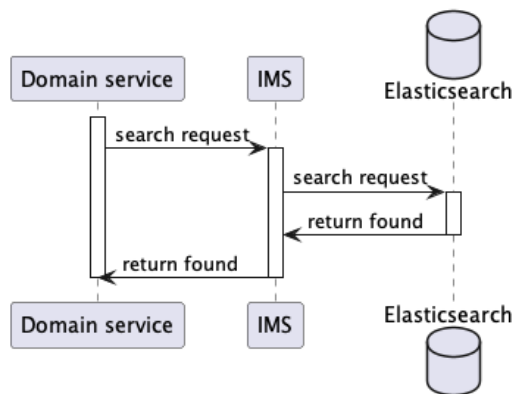
■ Figure 4.4 User update use-case

User update use-case

The fourth use-case is the updating of the user by its id. "Project Alpha" has various options for updating the user index. There is no need to cover all business-specific parameters and possibilities, but the main point is to cover the possibility of updating existing users in a correct way (see 4.4);

User search use-case

The last use-case is the reason for the whole data migration to Elasticsearch. Due to the "Project Alpha" business requirements, there is often a need to search for a user not by id but by the other properties the user might have. For example, one of the popular options is to search for



■ Figure 4.5 User search use-case



users with a full-text search by user's email address. As the full-text search might be done as soon as the new user is created or updated, the data should appear in Elasticsearch in real-time and as soon as possible to keep an acceptable user experience (see 4.5).

## 4.2 Consistent data migration

This section is dedicated to design of the the main part of the solution, which the consistent data migration.

### 4.2.1 Transactional outbox as chosen pattern

A Transactional outbox pattern was chosen to provide consistent data migration between RDBMS and Elasticsearch, as it meets all expectations and does not bring unnecessary complexity to the application. As soon as data in RDBMS (or IDP) are changed, an independent message event is created, which is not part of the initial ACID transaction. That meets the Transactional outbox pattern scenario and gives the project eventual consistency, which is satisfying for the given scenario.

### 4.2.2 Transaction log tailing as Message relay

Message relay is an important part of the Transactional outbox pattern that can be implemented differently. "Project Alpha" requires the data migration to be as fast as possible and in real-time. Therefore, the best option to implement a Message Relay is Transaction log tailing.

"Project Alpha" uses Keycloak for the implementation of the IDP. Consequently, this approach gives us more flexibility, as the Message relay does not directly interfere with transactions.

However, an essential part of the Message Relay design is that it needs to be implemented with "at least once delivery" in mind. Therefore, it needs to be managed separately.

### 4.2.3 Message ordering

As the application will use some event store to transfer messages between the IDP and Elasticsearch, message order could be a problem that needs some explicit solving. Some event stores can ensure correct message ordering out of the box, but not for all possible configurations. For example, Apache Kafka can provide message ordering only within a single partition, but a single partition could not be enough. This problem will be solved on the implementation level, and there will be more on this topic later in the implementation chapter.

### 4.2.4 Message deduplication

By applying "at least once delivery" and maintaining message ordering, the applied concept is getting closer to the desired "exactly once delivery" approach, but there is one problem that needs to be solved to bring it as close as to the perfect conditions as possible. This problem is message duplication, which can cause severe inconsistency between RDBMS and Elasticsearch. For example, producing instances can send several duplicates for the "delete" event due to problems with acknowledgments from the binder, and this can cause a wide variety of errors on the consuming side. Similar situations can happen to any non-idempotent operations.

## 4.3 Components

This section will describe each service's designs independently, their purposes, and how they should communicate with the other services. Each implemented component will be a microservice

application close to the reference applications from "Project Alpha" but will not contain all business-specific aspects, such as specific fields or integrations with the monitoring tools.

There are many industry-standard solutions for data migration that can help developers with data migration problem. Nevertheless, all of these solutions or frameworks have their pros and cons. The most popular ones that are fairly reliable and trustworthy are:

- Logstash;
- PGSync.

### PGSync

The PGSync is a lightweight solution that covers the problem of data migration from the PostgreSQL RDMBS to a NoSQL data storage, such as Elasticsearch storage (see 2.10). It works with the CDC <sup>1</sup>, which is a reliable way to build the migration pipeline without building SQL queries. It has many options for the pipeline's customizability, such as scaling or joining multiple tables. However, this tool is not perfect either. It does not provide zero-downtime migrations as of today, and it is not an option if there is a need to implement a complex pipeline, as it is a standalone application.

### Logstash

Logstash is another option, which is contributed by Elastic and is a part of the Elastic Stack. It is a server-side data processing pipeline that ingests data from a given source, such as Kafka. The pipeline consists of inputs, filters, and outputs. Therefore data can be pulled from the given source, transformed, and, in the case of "Project Alpha," routed to Elasticsearch.

Logstash guarantees at-least-once delivery with the queue for the dead letters. Possibilities to fully secure pipelines and add monitoring tools are available as well. However, it is an additional pipeline we need to maintain and build, as these event needs to be already published to the message queue.

### Spring Cloud Stream

An alternative way to build the pipeline for consistent data migration would be using Spring Cloud Stream (see 2.13). In such a way, this pipeline is highly customizable, but at the same time, it does not require much code to be written and is built around the functions and applications of the same vendor.

This framework is not a widely used solution for data migration from the RDMS to Elasticsearch. Still, it is a popular solution for building robust and highly scalable data pipelines with functions and applications for many types of producers, processors, and consumers, including CDC PostgreSQL Supplier and Elasticsearch sink. Therefore, it will be the way to go for the implementation part of this thesis work.

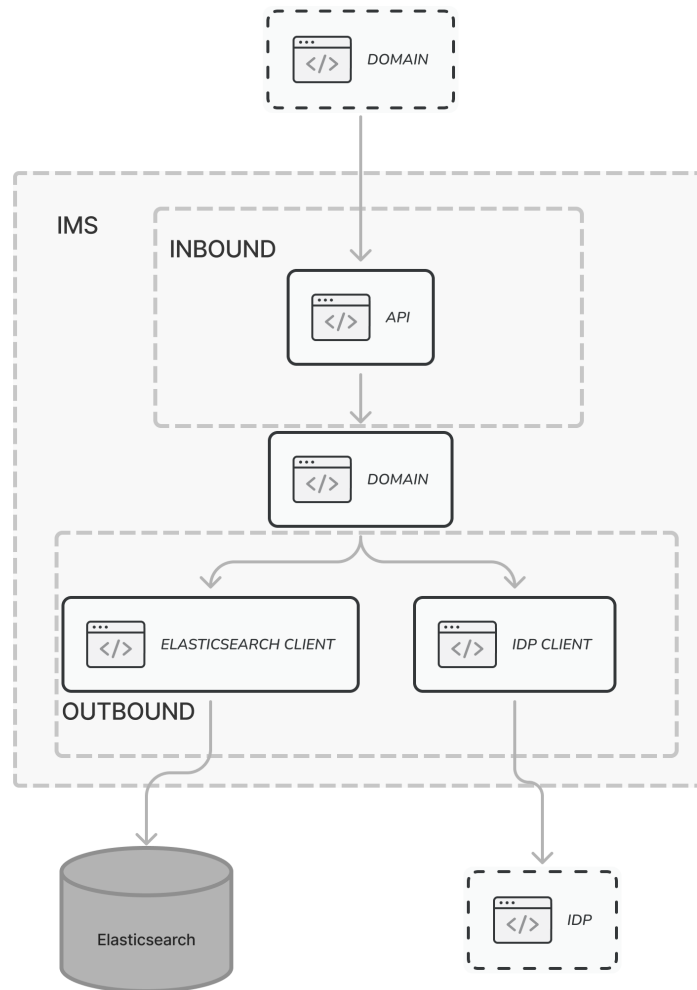
#### 4.3.1 Domain service

The Domain service does not need to be a working and deployable application, as it represents all possible services that can use IMS. Therefore it can be defined as a set of preconfigured requests representing use cases covered by the implementation of other services.

All microservices will be based on a Spring Framework using all needed additional frameworks and libraries to eliminate inventing the already existing industry standard solution as much as possible. This section will mainly describe the design perspective but with the technical aspect of the implementation in mind.

---

<sup>1</sup>Change Data Capture



■ **Figure 4.6** IMS component architecture

### 4.3.2 Identity management service

The Identity management service is one of the vital services that must be implemented as a complete deployable application, as it participates in the use cases involving the data migration problem between RDBMS and Elasticsearch.

The application will contain a REST API that will use Domain service with CRUD operations for Users and additional search requests. This API will represent the input to the application. As for the output, the application will communicate with the IDP service for the base CRUD operation on the user domain, and, on top of it, all search requests will be sent directly to the Elasticsearch instance and will be in the form of full-text search. As for the domain logic of the application, it will mainly serve as a bridge between inbound and outbound adapters.

This microservice already has separate modules and domain logic that connects them. Therefore it is a great opportunity to apply hexagonal architecture, which allows distinguishing these independent adapters and gives clarity to the application architecture. All of our connections to the external services will be our adapters:

- REST API;
- Elasticsearch client;
- IDP client.

REST API will be an inbound adapter, clients for the IDP, and Elasticsearch - outbound adapters (see 4.6).

### 4.3.3 Identity provider service

The identity provider service is another essential service that will be a part of the data migration flow. It contains the business logic for maintaining and storing user data. "Project Alpha" uses the help of the Keycloak infrastructure, which is highly complex. Such a complex infrastructure is not needed to simulate the data migration flow. The application will receive User CRUD requests and store these data in RDBMS within transactions. Then, as a part of the Transactional outbox pattern, this microservice will have a connection to the Spring Cloud Stream binder. Message Relay will be implemented using the CDC due to the real-time data flow approach requirement.

A Message Relay is a vital part of the Transactional outbox pattern in this microservice. From the perspective of the Spring Cloud Stream platform, it will be a supplier instance. In this case, a preconfigured function that works with Debezium: Spring Cloud Stream CDC Supplier will serve as a Message Relay.

This microservice will have hexagonal architecture, the same as IMS, where adapters will be:

- REST API;
- Binder client;
- Message relay.

REST API will be an inbound adapter, client for the binder, and Message Relay – outbound adapters (see 4.7).

### 4.3.4 Binder

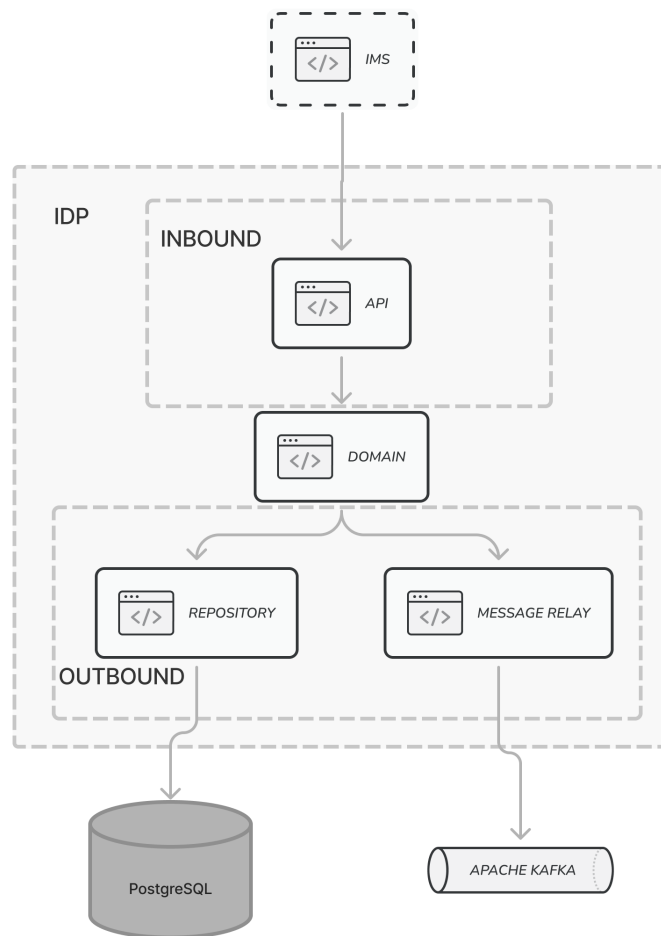
The result microservices infrastructure binder will be a connecting point between the Message Relay in the IDP and Elasticsearch instance. The name itself comes from the Spring Cloud Stream framework. The binder is a message broker that provides the communication flow between producers, processors, and consumers. The binder instance will be the same Apache Kafka queue to keep the architecture as close to the "Project Alpha" as possible. Therefore will be used the Apache Kafka instance running in the cloud.

### 4.3.5 Elasticsearch consumer service

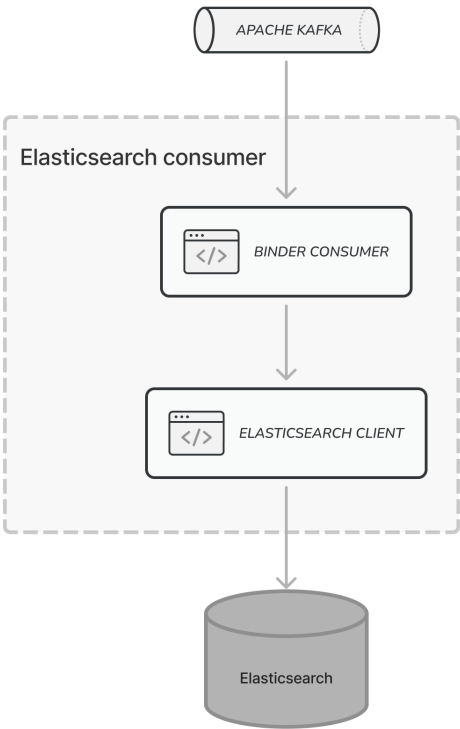
There is a need for an additional single-purpose application to maintain the data flow from the binder to the Elasticsearch instance (see 4.8). From the Spring Cloud Stream perspective, it will serve as a consumer instance.

### 4.3.6 Elasticsearch instance

Like the Binder instance, this project will use a running Elasticsearch instance in the cloud to keep the architecture as close to "Project Alpha" as possible. It will contain the user index, allowing the IMS instance to provide a full-text search across all user fields. Each record of the Elasticsearch index will have the same fields as the user in RDBMS.



■ **Figure 4.7** IDP component architecture



■ **Figure 4.8** Elasticsearch consumer component architecture

## 4.4 Design summary

This section was dedicated to creating a design of the application that will provide consistent data migration from the RDBMS to Elasticsearch.

The design contains a Transactional outbox pattern with Transactional log tailing as a Message relay. With additional changes, this solution will provide "at least once delivery" for messages in the binder.

Each component of the future application was described, and was created a design of the architecture.





# Implementation

*This chapter is dedicated to describing the applied pattern in practice and describing the implemented microservices. Each microservice will be described both individually and as a part of the whole infrastructure.*

## 5.1 Repositories

As was described in the Design chapter, the implementation consists of several independent microservices that should be independent of each other and independently deployable. To split the implementation and keep it close together simultaneously, the way of a multi-repository project was chosen for this thesis work.

The root repository contains the common configuration for the entire infrastructure, such as the Docker Compose configuration file. Each service is a separate repository that can be added to the project as an independent project, but if there is no need to run the whole infrastructure, each project can be developed independently as well:

**master-thesis-src** - root project;

**master-thesis-ims** - IMS;

**master-thesis-domain-service** - Domain service;

**master-thesis-idp** - IDP

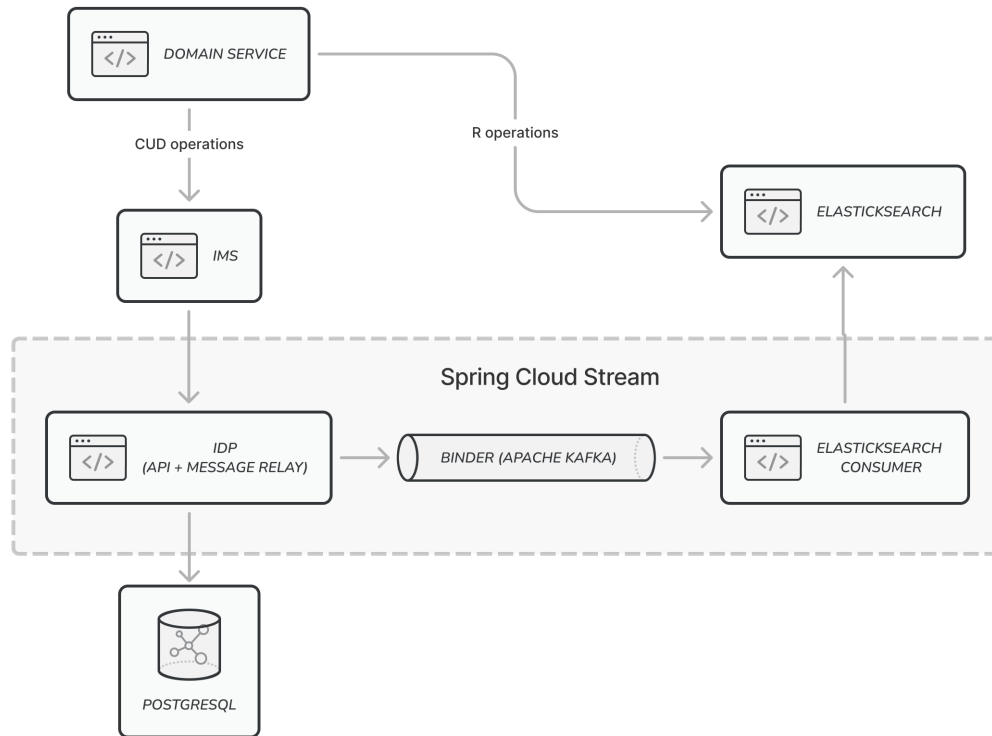
**master-thesis-elasticsearch-consumer** - Elasticsearch consumer service.

## 5.2 Architecture

The infrastructure uses the Spring Cloud Stream framework to implement the Transaction Out-box pattern (see 5.1). Identity provider service serves as a producer of the data. The data is the change data capture events transferred within a binder. The cause of the events are changes in the PostgreSQL `USER_TABLE` table.

On the other side of the related topic is the consumer instance, which is Elasticsearch consumer services. This service consumes data and carries this data to the destination, which is Elasticsearch instances. Afterward, these data can be accessed by the Identity management service with the least amount of delay.

Spring Cloud Stream architecture also supports the third type of service, Processor, that processes or transforms data, but no such type of service will be applied.



■ **Figure 5.1** Component diagram of the resulting architecture

## 5.3 Consistent data migration

### 5.3.1 Message ordering

One of the critical aspects of providing consistency for data migration is to ensure the correct message order on top of at "least one delivery" approach. As the Message broker was chosen, Apache Kafka, there should be the correct order of the messages within Kafka. Just imagine a situation when the consumer tries to delete the user before its creation, which could cause various errors. As long as there is only one partition for one CDC Supplier instance, there is no problem, as Apache Kafka can guarantee correct message ordering within one partition.

However, what if the single partition is insufficient for the given load? With this limitation, the application is not scalable. Therefore, the application should keep messages in the correct order even with the topic having more than one partition. There is an option to add a custom partition key expression for the Spring Cloud CDC Supplier function through the application properties to solve the given problem:

$$partition\_key = payload.id \% partitions\_amount$$

The supplier uses this expression to distribute messages across the topic partition. Changes of the same user will always be transmitted to the same partition. Therefore, as long as auto rebalancing is disabled and each consumer always works with the same partition, we will get the correct message order for all user data to change messages.

The Spring Expression Language (SpEL) was used to get the user id of the given CDC event for the function, allowing complex code to be passed into a partition key expression application property. Therefore, there is a possibility to generate a hashcode from the user UUID for each message and pass it to the hash function,

### 5.3.2 Message deduplication

At this point, messages are sent with an "at least once delivery" approach and have the correct order, but what about message duplications? For example, our CDC Supplier sent a message event to the binder but did not receive an acknowledgment, so it tries to send it again, but there is no certainty that the previous message was received or not. Therefore there can be several events messages for the same CDC event. To solve this problem, each message event in the binder has an id as an id of the CDC event that generates Debezium.

## 5.4 Domain service

The domain service is represented by the Postman collection that contains all requests and needed configurations to perform all calls currently supported by the Identity management service:

- POST for creating the new user with the given request body;
- GET for getting the user by its id;
- PUT for updating the user by its id;
- DELETE for removing the user by its id;
- POST for getting the user with the search query.

## 5.5 Apache Kafka binder

Spring Cloud Stream flow uses Apache Kafka distributed event store to implement the framework's binder. The instance running on the Confluent platform was used to keep the architecture as close to "Project Alpha" as possible. Consequently, each Spring Cloud Stream application covers the configurations required by this vendor to communicate with the given remote instance:

## 5.6 Elasticsearch instance

Same as with the Binder instance, the chosen version of Elasticsearch is the instance provided by the Elasticsearch Cloud platform to keep the architecture as close to "Project Alpha" as possible. Spring Data framework is the obvious choice for communication with Elasticsearch, but it is not always the best. More about the chosen clients will be in each service section that communicates with Elasticsearch.

By default, this framework supports authorization with the username and password, but communication with the API key was required. Therefore custom configuration for the communication with custom properties needs to be applied:

## 5.7 Identity management service

This service is a microservice application written in Kotlin with JDK 17 to provide management for the user domain. It accepts the request through the REST API from the Domain service and communicates directly with the Identity provider service and Elasticsearch instance running in the cloud, depending on the type of the received request.

The architecture of the application is inspired by hexagonal architecture. Therefore there is a set of different adapters and domain logic, which will be described in the following subsections.

```

1  spring:
2    cloud:
3      stream:
4        kafka:
5          binder:
6            brokers:
7              - ${KAFKA_BROKER_URI}
8            configuration:
9              sasl:
10               mechanism: PLAIN
11               jaas:
12                 config:"org.apache.kafka.common.security.plain.PlainLoginModule"
13                 + " required username='${KAFKA_USERNAME}'"
14                 + " password='${KAFKA_PASSWORD}';"
15             security:
16               protocol: SASL_SSL
17             client:
18               dns:
19                 lookup: use_all_dns_ips
20             session:
21               timeout:
22                 ms: 45000
23             acks: all

```

■ **Listing 5.1** Spring Cloud Stream configuration for the Confluent Apache Kafka

```

1  app:
2    consumer:
3      elasticsearch:
4        host: ${ELASTICSEARCH_HOST}
5        port: ${ELASTICSEARCH_PORT}
6        api-key: ${ELASTICSEARCH_API_KEY}

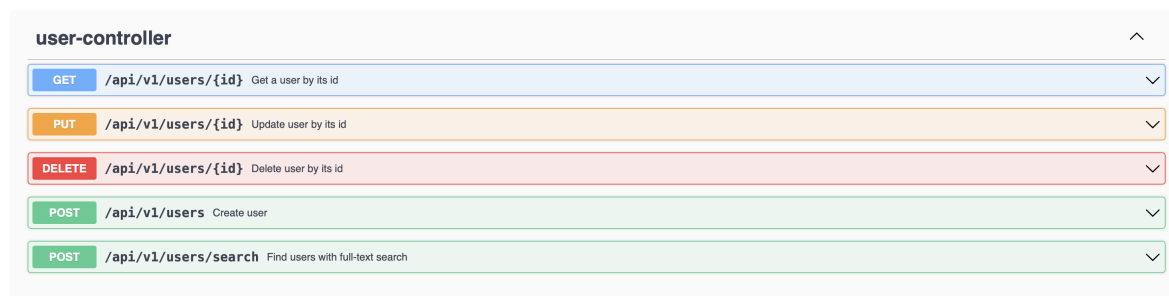
```

■ **Listing 5.2** Custom application properties for the communication with Elasticsearch

### 5.7.1 REST API adapter

The main entry point of this application is the REST API, which supports CRUD operations for the user domain with the additional ability to search for a user with the full-text search. All these requests are corresponding endpoints for the set of requests in the Postman collection for the Domain service:

- POST for creating the new user with the given request body;
- GET for getting the user by its id;
- PUT for updating the user by its id;
- DELETE for removing the user by its id;
- POST for getting the user with the search query.



■ **Figure 5.2** IMS Swagger UI

As for the documentation of the REST API, the Swagger framework was used. It is a very powerful tool tightly integrated with the application's code base. One of the options this tool has is to generate documentation based on the code itself by marking the needed methods and classes with the corresponding annotations, such as `@Operation` or `@ApiResponse`s. Afterward, every time the application is started up, the up-to-date version of the documentation is accessible from the Swagger UI (see 5.2), which runs on the same port as the application itself.

### 5.7.2 Elasticsearch adapter

This adapter retrieves the user data from the Elasticsearch instance with the full-text search. The full text itself is a query provided alongside the search request. For example, it could be as simple as one word with the data to be searched across the whole index.

The first attempt to implement this adapter was with the help of Spring Data Elasticsearch. This approach supports authentication with username and password, but the Elasticsearch Cloud instance reacquires the API key. Therefore the Spring auto-configuration bean is overridden to perform custom authentication, as was described earlier (see 5.6).

```

1  @Document(indexName = "user_index")
2  data class UserDocument(
3      @Id
4      val id: String,
5      @Field
6      val name: String,
7      @Field
8      val surname: String,
9      @Field
10     val email: String,
11     @Field
12     val address: String,
13 )

```

■ **Listing 5.3** Elasticsearch User index

This attempt, as a result, gave a working solution, but the current Spring Data Elasticsearch (4.4.3) supports only the `RestHighLevelClient`, which is already a deprecated solution at the moment of working on this thesis work. Moreover, this client is not recommended by Elasticsearch.

The alternative version to the `RestHighLevelClient` is `ElasticsearchClient`, which is

recommended client by Elasticsearch. The Spring Data Elasticsearch that was used in the application at that moment does not fully support this new client. Fortunately, there was a hint that the next major version of the Spring Data Elasticsearch would support this client naively. This newer major version was already out as a milestone alongside milestone of the new major version of the Spring Boot and Spring Cloud. Therefore, the second attempt at the implementation was to use these two new versions of dependencies and configure the new `ElasticsearchClient`.

Unfortunately, these milestones were in the early stage of development. The documentation did not cover these dependencies, and after a few attempts to run the application was discovered that the Spring Cloud Stream dependencies, which are part of the new Spring Cloud, had bugs that prevented the application from proper running. This attempt was unsuccessful as the Spring needs time to migrate to a newer client version and will release new support only approximately at the end of fall 2022.

The third and last attempt to implement this adapter was to abandon Spring Data Elasticsearch and use the framework directly from Elasticsearch, which uses a new Java API Client and has the same authentication configuration as Spring Data Elasticsearch. This framework does not provide the same functionality as Spring Data Elasticsearch for Repositories, but this adapter does not require complicated logic that these repositories would significantly simplify. On the other hand, this framework allows to build of any request that supports Elasticsearch with the DSL:

```

1  rride fun search(query: String): List<UserDocument> {
2  val response: SearchResponse<UserDocument> = client.search(
3      { s ->
4          s.index(USER_INDEX)
5              .query { q ->
6                  q.queryString { qsb ->
7                      qsb.query(query)
8                  }
9              }
10     },
11     UserDocument::class.java
12 )
13 return response.hits().hits().mapNotNull { it.source() }
14

```

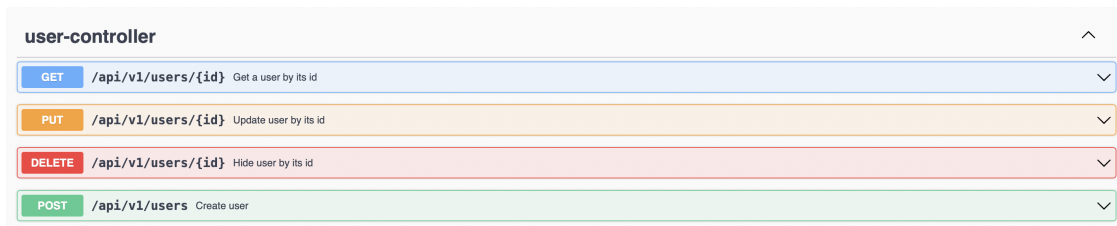
■ Listing 5.4 Elasticsearch request builder DSL

### 5.7.3 Identity provider adapter client

This adapter serves as client for the rest calls to the Identity Provider service. To implement this adapter was used the `RestTemplate`, a synchronous client for performing HTTP requests. This client is a part of the Spring Web framework. As communication with the external service is performed via HTTP, there can easily be many issues. Therefore adapter has the additional custom implementation of the `DefaultResponseErrorHandler` for error handling.

#### 5.7.4 Domain logic

The domain logic in this application serves as a bridge between the inbound and outbound adapter. It has its services and model implementation for each use case but does not alter



■ Figure 5.3 IDP Swagger UI

the request flow or data. It serves as a core logic of the application and defines interfaces for surrounding adapters.

The model of the domain logic defines classes that the business logic works with. All adapters should define their own representation of these classes if they want to work with it. As we work with the Kotlin programming language, the mapping between these classes can be easily implemented via extension functions, which is one of the benefits of this programming language over Java.

## 5.8 Identity provider service

The Identity Provider service represents the Keycloak instance in "Project Alpha." The application receives CRUD requests for the user data through the REST API adapter and saves data to the RDBMS storage through the domain logic. This process happens within a single transaction, and only after the transaction completion Message Relay reads the database transaction logs and sends the event message to the binder.

### 5.8.1 REST API adapter

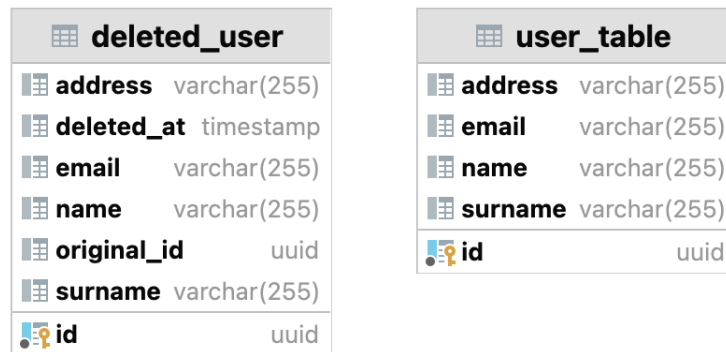
The entry point to the application is the REST API, which is similar to the IMS REST API both in structure and implementation. The main difference is that there is no endpoint for the search requests, as, from an IMS point of view, these requests are handled by the Elasticsearch instance directly. An additional slight difference refers to the process of user deletion, as from the IMS API point of view, the client sends requests for user deletion, but the IDP API only allows to hide given user.

### 5.8.2 Domain logic

The processing of the received requests is happening in the Domain logic adapter. To implement the user instance management was used Spring Data JPA framework with the use of PostgreSQL RDBMS. This combination allows the processing of each request within a single transaction.

The process of hiding the user instead of completely deleting it is represented by the auxiliary table `DeletedUser`, which contains all deleted user data, including the original `User` id indefinitely. The alternative way to implement the same logic would be to add auxiliary `deleted` attribute to the original `User` table and afterward filter record for all operations. The second approach can be efficient in case of complicated table structures with a lot of foreign keys.

The process of creating tables and possible updates of the tables is handled by the Spring Data JPA as well. The application validates the schema on the startup and changes it according to the entity classes if there is a need to do it. Each entity class represents the table in the database (see 5.4).



■ **Figure 5.4** IDP Entity diagram

### 5.8.3 Message relay

The implementation of the Message Relay utilizes the CDC Supplier function from the spring Cloud Stream framework. This function is mainly a wrapper for the Debezium (see 2.12), an open-source distributed platform for change data capture. In the application, Message Relay is the `cdcSupplierFunction` bean, through which each change of the `User` entity in the database flows. As the Debezium platform works with the approach of the Transaction log Tailing, the process of reading changes does not interrupt or slow down the transactions in any way. Therefore, the process of publishing the event to the binder is a subsequent event after the successful commit of the transaction.

The `cdcSupplierFunction` defines the `Supplier<Flux<Message<*>>>` bean, which works reactively. Therefore, as soon as the change data event is captured, it is processed and supplied to the binder. This particular implementation of the supplier bean does not perform any specific operation on for message, but it could be done if there is any need for it. Current supplier implementation only invokes the supplier bean and creates a log record for each supplied message to the console.

Each supplied message is assigned the id of the change data capture event. By taking this approach, we eliminate the duplicity of the same messages in the single partition of the binder:

```
1 message-key-expression: "#jsonPath(headers['cdc_key'], '$.id').getBytes()"
```

■ **Listing 5.5** Message key expression configuration

The producer distribution mechanism was changed so that the partition key is computed with the custom hash function to provide the correct distribution of messages across all partitions of the binder topic, as described in the Consistent data migration section.

## 5.9 Elasticsearch consumer service

This is a single-purpose application that serves to pull data from the topic and update the Elasticsearch index with received data. In terms of Spring Cloud Stream architecture, this application is a consumer. As for the structure, the application consists of a binder, the source of data to be processed, and the consumer part, which processes the received message and updates the corresponding Elasticsearch instance.



```
1 producer:
2   partition-count: ${USER_CHANGE_CDC_SUPPLIER_PARTITION_COUNT}
3   partition-key-expression: "UUID(#jsonPath(payload, '$.id')).hashCode() "+
4                             "% ${USER_CHANGE_CDC_SUPPLIER_PARTITION_COUNT}"
```

■ Listing 5.6 Producer distribution configuration

### 5.9.1 Binder

The binder part of the application utilizes the `spring-cloud-stream-binder-kafka`. Custom `elasticsearchConsumer` function was mapped to the same topic that uses the Identity Provider service.

The consumer instance can be replicated for the purposes of scalability. Therefore each consumer instance is mapped to a specific partition of the given topic. Automatic rebalancing that provides this binder out of the box was intentionally disabled. Therefore, each consumer is permanently mapped to the partition with the same index as the consumer itself. This approach ensures that each partition is processed by the same consumer all the time, and there is no race condition within the same user record.

On the application's startup, the binder checks the state of the topic offset. As the application aims to process each message consistently without the loss, the application will not reset offset but will try to resolve the difference by processing all skipped messages. For example, if the application restarted, but the producer was running all the time, it would consume all messages that were produced during the downtime.

The application changes the partition offset after successfully processing the message to provide consistency in the consuming process. Therefore, if there is an error during the processing of the message, the consumer will try to process the same message again. By taking this approach, the consumer will deliver the data to Elasticsearch with an "at least once delivery" approach.

### 5.9.2 Consumer

The consumer part of the application is an executable logic that is called by the binder function. It receives the message as a deserialized `UserMessage` instance and saves it as `UserDocument` instance to the Elasticsearch.

Implementing the connection to the Elasticsearch cloud instance uses the same dependency used for the implementation of the IMS Elasticsearch adapter, a native Elasticsearch client for the Java-based application.

## 5.10 Documentation

### 5.10.1 API documentation

Each application that has an API uses the Swagger framework. This framework was configured in a way that it generates the documentation based on the API code and annotations, which provides additional information. This documentation can be accessed through the Swagger UI, which is available as soon as the application is started up. To access the UI, go to the `/swagger-ui/index.html` with the same host and port as the application.

### 5.10.2 Developer guide

Each service has the attached `README.txt` file that provides all needed information for the developer for:

- successful setting up;
- successful starting up of the application/requests;
- maintenance.

## 5.11 Potentially useful upgrades

This section will cover some aspects of the implementation that can be upgraded. These aspects are not necessary for this thesis work, but for the production deployment purposes of the potential project, that this implementation can be integrated, it could be quite valuable and important changes.

### 5.11.1 Schema registry for messages

The current implementation relies on the fact that each microservice working with the binder knows what to expect from the binder or keeps a consistent data contract for the sent messages. This is a simple solution, but it can bring challenges for future improvements and maintenance. The alternative solution to improve this situation is to define the data contract outside these microservices.

If the application uses Apache Kafka as a binder implementation, it could be a good choice to use Apache Avro, an open-source data serialization system that manages the data exchange between applications. We can define the data format for the messages in a single place for both producers and consumers, and both of these applications support it naively, as they are using the Spring Cloud Stream framework.

### 5.11.2 Use prepared Spring Cloud application

Both microservices that are part of the Spring Cloud Stream flow (IDP, Elasticsearch consumer) use the relevant function framework function. But this framework has the same function in the form of a prepared application.

These dependencies can significantly speed up the development process, as they have, in addition to the primary function functionalities, prepared autoconfiguration for the cloud infrastructure, such as Datadog or Kubernetes autoconfiguration.

### 5.11.3 Monitoring


As the final architecture is highly scalable and consists of many independent microservices, it is not easy to monitor the data flow and the current state or load of each microservice. Therefore, it could be a good idea to add a monitoring system to the whole infrastructure. It is not crucial for the current example implementation, but for future updates to "Project Alpha," it will be highly important to monitor and keep statistics.

Ideally, keep track of the Apache Kafka instance because the stability of the infrastructure depends on the state of partitions and topics. For example, if the configuration of topics is lost after the restart, it could brake the eventual consistency of the system.

## **5.12** Implementation summary

In this chapter the reader was introduced on how the applications were created and how applications internally work, including the overall application architecture (see 5.1).





## Chapter 6

# Testing

*This chapter is dedicated to testing the implemented infrastructure, including load testing, to verify whether the implementation meets expectations.*

The testing process is divided into two steps. The first step is to verify whether the implemented infrastructure satisfies the expectations of the domain microservice with end-to-end testing (see 6.1). In this particular case, domain microservice is an abstract collective term representing a subset of "Project Alpha" functionalities and defined by the set of HTTP requests. The infrastructure was tested with end-to-end testing to test whether the implementation works as expected from the user's point of view.

As for the requirement of consistency, end-to-end tests are not enough. Firstly, these tests do not provide enough load to the system to verify the bottlenecks, and secondly, at the end of the testing, we are not sure whether there was some problem during the communication. For example, whether there are no duplicated documents in Elasticsearch or whether the messages were, in fact, in the correct order in the binder. Therefore, the infrastructure was additionally tested with the load test to test the stability and consistency (see 6.2), and it will be the second step of the testing.

### 6.1 End-to-end tests

To test whether implemented microservices infrastructure meets the expectations of the domain services was implemented end-to-end tests. These tests were implemented with the use of the API platform, Postman, and you can find them in `master-thesis-domain-service` repository under the `e2e-tests` folder.

The process of running these tests requires running all microservices, including Kafka binder and Elasticsearch instances. Tests cover use cases:

- creating new user;
- updating the user by its id;
- finding the user by its id;
- deleting the user by its id;
- searching for a user with full-text search.

RUN SUMMARY		1
▶ <b>POST</b> add user	1   0	
▶ <b>GET</b> get user by id	1   0	
▶ <b>POST</b> search	1   0	
▶ <b>PUT</b> update user by id	1   0	
▶ <b>DELETE</b> delete user by id	1   0	
▶ <b>POST</b> search (not found)	1   0	
▶ <b>GET</b> get user by id (not found)	1   0	

■ **Figure 6.1** End-to-end tests summary

## 6.2 Load tests

To verify whether the data migration between the PostgreSQL RDMS and Elasticsearch instance is, in fact, consistent, we need to put implemented microservices infrastructure under a heavy load and check whether:

- all messages were successfully sent;
- all messages were successfully received;
- all messages were received in the same order as they were sent;
- there are no obvious bottlenecks.

To perform load tests was used the open-source load testing tool Locust (see 2.14). This tool allows users to manually implement all tasks that need to be performed, giving users a wide range of scenarios. In this particular case, we need to simulate a situation when messages can potentially arrive in the wrong order. Therefore we need to perform many simultaneous operations with different users that can put on load all running consumer instances.

To verify whether the messages were processed in the correct order, load tests were performed by update operation on several users, and each update operation changes address value to value with postfix, where the postfix is an index of update iteration. Therefore, by limiting the iterations amount to a specific value, we can check whether each user has the final value with the index of the last iteration at the end of load tests, then the last message was last, and the order of the messages was correct.

The configuration used for load tests in the `master-thesis-domain-service` repository under the `load-tests` folder. The Locust tool allows us to specify the number of running instances of the client and spawn rate, but the scenario itself needs to be implemented manually. To spawn an update request, firstly application needs to send a POST request and retrieve the user's id. Therefore, an extra `on.start` method is performed only once at the start of the load tests. The second important method is the task itself: update the request according to the chosen scenario.

■ **Figure 6.2** Performed requests during the load testing

Type	Name	Number	RPS	Failures
POST	/api/v1/users	10	0.17	0.0
PUT	/api/v1/users/0b68caf5-b620-4e6d-9e35-a869380eb8eb	100	1.71	0.0
PUT	/api/v1/users/18e419de-dfa9-4d23-8591-d7a6d0161bfa	100	1.71	0.0
PUT	/api/v1/users/3e863a93-cebb-451b-9b60-d1210845547d	100	1.71	0.0
PUT	/api/v1/users/4734b170-e0d1-4074-b5eb-b51ee743fc36	100	1.71	0.0
PUT	/api/v1/users/66fbee94-7c32-40e6-a0a3-0026ea53e3ff	100	1.71	0.0
PUT	/api/v1/users/734e47c1-858d-4db8-aa1f-ed932c2eeff0	100	1.71	0.0
PUT	/api/v1/users/a0177be5-f7dd-4d19-88c0-1cc7bd0d5b06	100	1.71	0.0
PUT	/api/v1/users/b62882f6-ea22-4537-92a0-7bf724b7b23c	100	1.71	0.0
PUT	/api/v1/users/e70f57f1-c7c0-4679-bcca-d6713d21310a	100	1.71	0.0
PUT	/api/v1/users/fd5988ea-a468-4107-8fe6-9415b122e9a9	100	1.71	0.0
	Aggregated	1010	17.27	0.0

All microservices instance was running in local docker-compose as well as Locust instance with configuration (see 6.2):

- number of users: 10;
- spawn rate 100;
- wait time: random value between 0.05 and 1 second;
- amount of iteration: 100.

During the load testing, was used current docker-compose configuration, which can be found in the root repository with one IDP instance and two consumer instances. As shown in the diagram (see A.1), the first responses were slower than the rest, as these first requests initialized Spring DispatcherServlet. The rest of the requests are much faster, as the IDP does not work with the binder within a transaction, and the Message Relay does not slow down the round trip time.

The number of requests per second was reasonably consistent, which points to consistency of request processing, as each task waits for the response before sending the subsequent request (see A.2).

For the local testing, was used author's laptop with the configuration:

**OS** macOS Monterrey 12.3.1

**CPU** Apple M1 Pro

**RAM** 16 GB

### 6.3 Future steps

The load tests were done on a local machine. In order to test the system under load, which will be closer to real life, there is a need to run all microservices in the cloud and test under the load that the cloud services can provide.

### 6.4 Testing summary

In this section, the reader was introduced to the process of testing that was applied to the implemented architecture. The infrastructure was tested on the subject of functional requirements

and, afterward, was tested with load tests to verify whether the infrastructure could successfully work on load on the local machine. Each testing method was described and was described the results of the testing.



 Chapter 7

## Conclusion

*In this chapter will be described the result of the work done in this thesis work.*

This master's thesis aimed to analyze the architectural patterns, which can be applied to microservices to provide data consistency, and, afterward, apply this knowledge to the design, implementation, and testing of the possible architecture for the provided "Project Alpha."

In the first chapter of this thesis, the reader was introduced to the problematic of Microservices Architecture and particularly to the problem of Data consistency in Microservices Architecture. This knowledge is very important to understand how possible solutions can solve the given problem. Additionally, technologies that were used across the work were introduced and described.

Widely known patterns were analyzed, including their pros and cons, which ACID characteristics they bring, and how we can apply them to the projects.

Then, "Project Alpha" was analyzed and proposed a possible solution for providing data consistency for the given use case: consistent data migration from the RDBMS to Elasticsearch.

The proposed design was also implemented. The implementation of the whole infrastructure and each microservice were described in the Implementation chapter.

The "Project Alpha" was given for this thesis work as a reference for the requirements and scope. This is a big project that is complex and demanding enough to benefit from this approach to the problem of consistent data migration. However, there are other working options developers can choose. As described in the Implementation chapter, there are many options depending on the chosen approach by the team and the required functionality.

For example, if there is a need to migrate some analytical data to Elasticsearch and there is no need to process this data or transmit these data to several consumers at once, PGsync as a standalone application could be a good enough option. Alternatively, if the project infrastructure is highly dependent on Elasticsearch and there is a need to use other additional tools from Elastic Stack, Logstash could be the main pipeline of the project infrastructure.

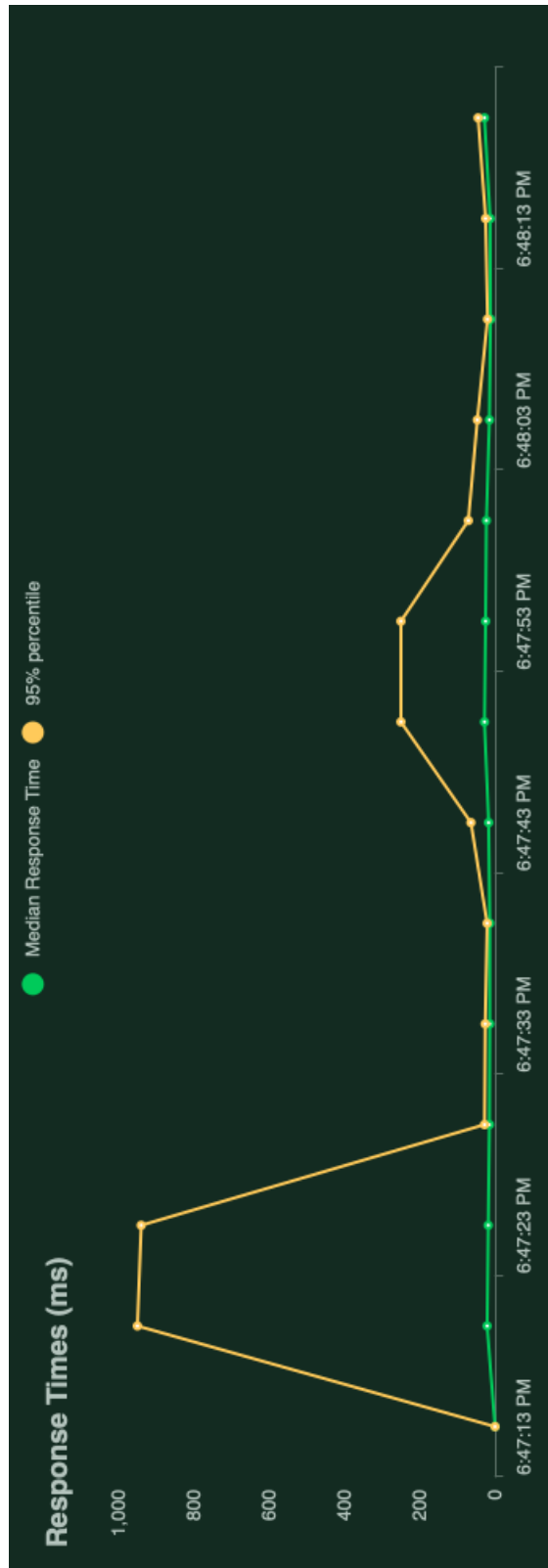
As for the implemented solution, it is a custom pipeline built around the Spring Cloud Stream framework with native consumers and suppliers that can handle the load with "at least once" delivery and correct message delivery without losing the ability to scale horizontally. Implemented infrastructure does not require much code to be written and can be easily maintained.

By the end of this master's thesis, the reader was introduced to the problem, possible patterns were analyzed, and one was chosen that was used for the implementation and afterward tested. It can be claimed that all goals of the work were met. The implemented project, including all documentation, was forwarded to the developer teams of "Project alpha," I hope it will help them develop new changes in the best possible way and upgrade the project without any problems.

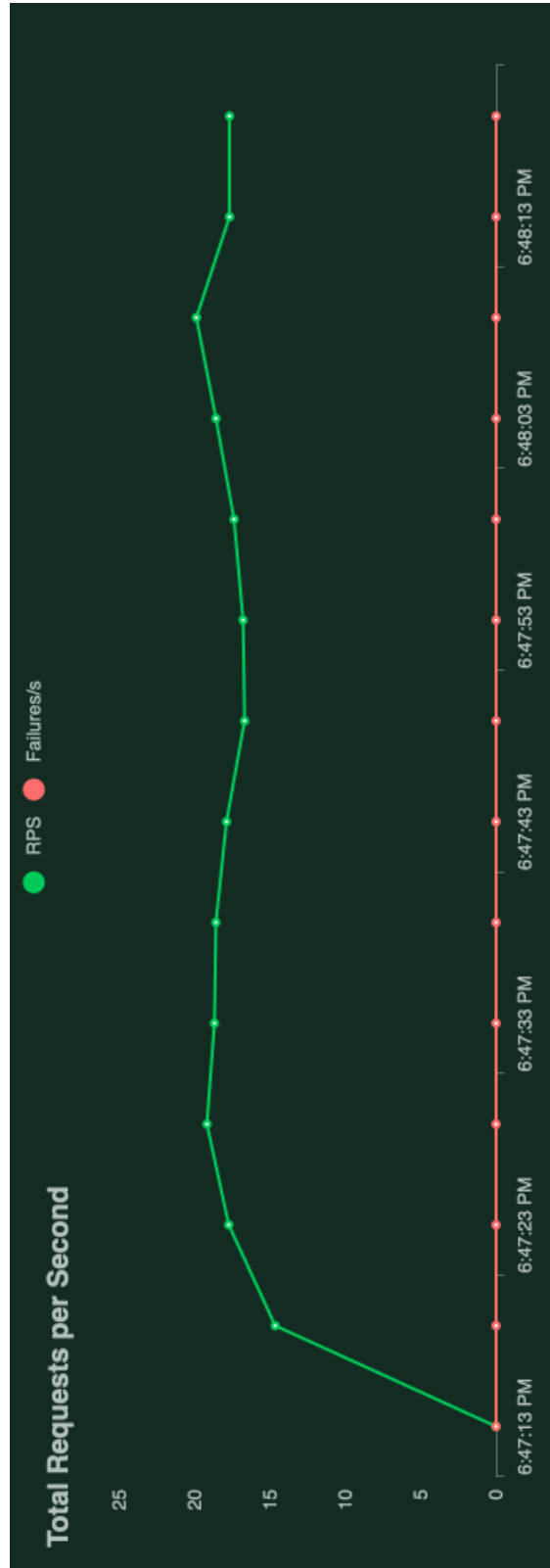


..... Appendix A

# Testing results



■ **Figure A.1** Response times during the load testing



■ **Figure A.2** Number of requests per second during the load testing



# Bibliography

1. JAKUB. *Two Generals' Problem*. 2018. Available also from: <https://finematics.com/two-generals-problem>. [Online; accessed Mon 12, 12, 2022].
2. GRAY, Jim. *Notes on Data Base Operating Systems*. Berlin, Heidelberg: Springer-Verlag, 1978. ISBN 3540087559.
3. IVANNONTECH. *Byzantine Generals' Problem – An Introduction*. 2021. Available also from: <https://academy.moralis.io/blog/byzantine-generals-problem-an-introduction>. [Online; accessed Wed 14, 12, 2022].
4. LESLIE LAMPORT Robert Shostak, Marshall Pease. *The Byzantine Generals Problem*. Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025: SRI International, 1982. ISSN 0164-0925.
5. NEHA NARKHEDE, Gwen Shapira; PALINO, Todd. *Kafka: The Definitive Guide*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2017. ISBN 9781491936160.
6. AINA, Tolu. *PostgreSQL to Elasticsearch sync* [online]. GitHub. Available also from: <https://github.com/toluaina/pgsync>.
7. MORLING, Gunaar. *Streaming Now - Debezium 1.0 Final Is Out* [online]. Available also from: <https://debezium.io/blog/2019/12/18/debezium-1-0-0-final-released/>.
8. TANZU, VMware. *Spring Cloud Stream Reference Documentation Version 3.2.5* [online]. Available also from: <https://docs.spring.io/spring-cloud-stream/docs/current/reference/html/spring-cloud-stream.html#spring-cloud-stream-reference>.
9. HEYMAN, Jonatan; HOLMBERG, Lars; BYSTRÖM, Carl; HAMRÉN, Joakim; HEYMAN, Hugo. *Locust Documentation* [online]. Read the Docs. Available also from: <https://docs.locust.io/en/stable/#locust-documentation>.
10. KOTLIN FOUNDATION. Learn Kotlin. [N.d.]. Available also from: <https://kotlinlang.org/docs/reference/>.
11. Contributing to Angular. In: *GitHub*. 2018. Available also from: <https://github.com/angular/angular/blob/master/CONTRIBUTING.md>.
12. GRADLE INC. Gradle vs Maven Comparison. © 2020. Available also from: <https://gradle.org/maven-vs-gradle/>.
13. BEALDUNG. *Ant vs Maven vs Gradle*. Available also from: <https://www.baeldung.com/ant-maven-gradle>.
14. NEHA NARKHEDE, Gwen Shapira; PALINO, Todd. *Distributed Transaction Processing: The XA Specification*. X/Open Company Ltd., U.K: X/Open Company Ltd., U.K., 1991. ISBN 1872630243.

15. DTM. *Understanding XA Transactions With Practical Examples in Go*. 2022. Available also from: <https://betterprogramming.pub/understanding-xa-transactions-with-practical-examples-in-go-67e99fd333db>. [Online; accessed Mon 21, 12, 2022].
16. JAKARTA TRANSACTIONS TEAM. *Jakarta Transactions 2.0*. 2020. Available also from: <https://jakarta.ee/specifications/transactions/2.0/jakarta-transactions-spec-2.0.pdf>. [Online; accessed Mon 21, 12, 2022].



# Contents of the attached media

- | `readme.txt` ..... the file with CD contents description
- | `src` ..... the directory of source codes
  - | `impl` ..... implementation source codes
  - | `thesis` ..... the directory of  $\text{\LaTeX}$  source codes of the thesis
- | `text` ..... the thesis text directory
  - | `thesis.pdf` ..... the thesis text in PDF format