MMINISTRY OF HIGHER EDUCATION
SCIENTIFIC RESEARCH AND INNOVATION

SULTAN MOULAY SLIMANE UNIVERSITY

NATIONAL SCHOOL OF APPLIED SCIENCES
KHOURIBGA

End of studies dissertation

for obtaining the

State Engineer Diploma

Sector : Computer Science and Data Engineering

Presented by :
Ziyad **RAKIB**

Participation in the Migration from a Monolithic Architecture to a Microservices Architecture

Supervised by :
Abdelghani **Ghazdali** (University)
Nidal **Lamghari** (University)
Bilal **Slayki** (Organization)

Jury members :

OURDOU Amal	Entity	Chairman
OUMMI Hssaine	Entity	Examiner

Academic Year : **2023**

Summary

Dedication ii

Acknowledgements iii

Résumé iv

Abstract v

Contents vii

List of Figures ix

List of Tables x

Introduction 1

1 General project context 2

2 Decision Support and Solution Choices 12

3 Technologies Used 21

4 Solution Implementation 28

Conclusion 45

Bibliography 46

Dedication

"

TO my mother, who showered me with her support and devoted me with unconditional love. You are for me an example of courage and continuous sacrifice. May this humble work bear witness to my affection, my eternal attachment and may it call upon me your continual blessing,

TO my father, no dedication can express the love, esteem, devotion, and respect I have always had for you. Nothing in the world is worth the efforts made day and night for my education and my well-being. This work is the fruit of your sacrifices that you made for my education and training,

TO my dear brothers, thank you for your love, support, and encouragement,

TO all my dear friends, for the support you have given me, I say

Thank you.

"

- Ziyad

Acknowledgements

First and foremost, I would like to thank the Almighty Allah for granting us the strength to persevere and overcome all difficulties.

I would like to express my gratitude to **Mr.Abdelghani Ghazdali**, the head of the Computer Science and Data Engineering department, who ensured the smooth progress of the supervision sessions from start to finish.

I would also like to extend my sincere thanks to **Professor Nidal Lamghari**, for his valuable advice, expertise, and availability throughout this experience. Your attentive guidance allowed me to deepen my knowledge and improve my technical skills. I am grateful for the opportunity to benefit from your expertise and constant support.

At the conclusion of this work, I would also like to express my sincere thanks and gratitude to all those who, through their teaching, support, and advice, contributed to the progress of this project.

Furthermore, I would like to express my gratitude to **IZICAP**, where I had the opportunity to carry out my final year project. I am grateful to the entire **IZICAP** team for welcoming me warmly and allowing me to apply my academic knowledge in a professional environment. Your expertise, mentorship, and resources have been invaluable in the completion of this project.

I would like to give special thanks to **Mr.Bilal SLAYKI**, my supervisor in the company, for his constant support, wise guidance, and invaluable assistance throughout this project. Your experience, insightful advice, and kindness have inspired me and greatly contributed to my professional growth. I am grateful for sharing your expertise with me and for being an exceptional mentor.

Finally, my thanks also go to the entire faculty and administrative staff of ENSA Khouribga for their efforts in providing us with a quality education, as well as the administrative and technical team for all the services provided.

Résumé

Le présent rapport de projet de fin d'études (PFE) met en évidence les différentes étapes de notre projet de transformation de l'architecture monolithique en Groovy vers une architecture basée sur des microservices en Java, tout en effectuant une transition de AngularJS vers React et en adoptant une approche de microfrontend. Nous avons entrepris cette démarche afin d'améliorer la flexibilité, la maintenabilité, les performances, la cohérence des données et la modularité de notre application.

Initialement, notre code monolithique était développé en Groovy. La première étape de notre projet a consisté à analyser cette architecture monolithique et à identifier les composants qui pourraient être isolés en microservices indépendants. Parallèlement, nous avons évalué les avantages de migrer de AngularJS, un framework JavaScript obsolète, vers React, un framework plus moderne et performant.

Nous avons procédé à la conception et à la mise en œuvre de ces microservices en utilisant Spring-Boot, en veillant à découpler les fonctionnalités et à assurer une communication efficace entre les services. Dans le même temps, nous avons migré progressivement notre code AngularJS vers React, en réécrivant les fonctionnalités existantes avec les meilleures pratiques de développement React.

Pour assurer la cohérence des données entre les microservices, nous avons mis en place Delta Lake, une technologie de gestion de données incrémentielle. Delta Lake a permis de garantir la fiabilité des données et de simplifier les opérations de lecture et d'écriture sécurisées entre les services.

Les connecteurs en Java ont facilité la communication entre le frontend basé sur React et Delta Lake, assurant ainsi une connexion robuste et sécurisée. Les connecteurs ont également permis des opérations de lecture et d'écriture efficaces, en garantissant l'intégrité et la cohérence des données.

En parallèle, nous avons également adopté une approche de microfrontend pour notre architecture frontend. Cela nous a permis de découpler les fonctionnalités du frontend en modules autonomes, offrant ainsi une plus grande flexibilité et la possibilité de les développer et de les déployer indépendamment.

Enfin, pour faciliter le déploiement et la gestion de notre architecture basée sur des microservices, React et le microfrontend, nous avons adopté l'utilisation de Kubernetes et Docker. Ces outils ont permis l'orchestration et la mise à l'échelle efficaces des services, tout en simplifiant le déploiement dans différents environnements.

Mots-clés: Monolithique, Groovy, architecture microservices, AngularJS, React, Delta Lake, Microfrontend, Kubernetes, Docker.

Abstract

This final year project report highlights the different stages of our project to transform a Groovy-based monolithic architecture into a Java-based microservices architecture, while transitioning from AngularJS to React and adopting a microfrontend approach. Our goal was to enhance the flexibility, maintainability, performance, data consistency, and modularity of our application.

Initially, our codebase was developed as a monolith using Groovy. The first step of our project involved analyzing the monolithic architecture and identifying components that could be isolated into independent microservices. Concurrently, we evaluated the benefits of migrating from the outdated AngularJS framework to the more modern and performant React framework.

We proceeded with the design and implementation of these microservices using Spring Boot, ensuring loose coupling between functionalities and effective communication among services. Simultaneously, we gradually migrated our AngularJS code to React, rewriting existing features according to React's best development practices.

To ensure data consistency among microservices, we implemented Delta Lake, an incremental data management technology. Delta Lake guaranteed reliable data and simplified secure read and write operations between services. Java connectors facilitated communication between the React-based frontend and Delta Lake, ensuring robust and secure connectivity. These connectors also enabled efficient read and write operations, ensuring data integrity and consistency.

In parallel, we adopted a microfrontend approach for our frontend architecture, decoupling frontend features into autonomous modules. This provided greater flexibility and the ability to develop and deploy modules independently.

Finally, to facilitate deployment and management of our microservices, React, and microfrontend architecture, we adopted Kubernetes and Docker. These tools enabled efficient orchestration, scaling of services, and simplified deployment across different environments.

In conclusion, our project resulted in significant improvements in flexibility, maintainability, performance, data consistency, and modularity of our application. The transition from a Groovy-based monolithic architecture to a Java-based microservices architecture, combined with the migration from AngularJS to React, the adoption of Delta Lake, Java connectors, microfrontend, Kubernetes, and Docker, collectively optimized our system.

Mots-clés : Monolithique, Groovy, Microservices, AngularJS, React, Delta Lake, Microfrontend, Kubernetes, Docker

---

ملخص

يقدم هذا التقرير تحليلاً متعمقًا للمشروع المعقد الذي تم إجراؤه لتوسيع نطاق مصدر بيانات الشركة. يتطلب المشروع إزالة MariaDB وتنفيذ Delta Lake ، وهو حل تخزين ومعالجة بيانات عالي الأداء. بالإضافة إلى ذلك ، تم تقسيم المونوليث الحالي إلى خدمات مصغرة باستخدام Spring Boot كواجهة خلفية مع موصل مناسب لـ Delta Lake ، وواجهة ReactJS كواجهة أمامية. قدم المشروع العديد من التحديات التي تطلبت تطبيق التقنيات والاستراتيجيات المتقدمة. وشمل ذلك ترحيل البيانات ، وضمان اتساق البيانات ودقتها ، وإدارة تعقيدات الأنظمة الموزعة ، ودمج التقنيات والخدمات المختلفة. يحدد التقرير الحلول المختلفة التي تم تطويرها لمواجهة هذه التحديات ، بما في ذلك استخدام خوارزميات معالجة البيانات المتقدمة ، وبنى الحوسبة الموزعة ، والحاويات. على الرغم من التحديات التي واجهها ، لا يزال المشروع قيد التنفيذ. يقدم التقرير رؤى حول الجهود الجارية لتحسين قابلية المشروع للتوسع والأداء والوظائف العامة

كلمات مفتاحية : المشروع، مصدر البيانات، Delta Lake ، المونوليث، الميكروسيرفس، Springboot ، ReactJS ، الميكروواجهات، الخلفية، الواجهة الأمامية، البيانات، القابلية للتوسع، الأداء. Delta Lake ، Springboot ، ReactJS ، الميكروواجهات.

Contents

Dedication ii

Acknowledgements iii

Résumé iv

Abstract v

Contents vii

List of Figures ix

List of Tables x

Introduction 1

1 General project context 2

Introduction 2

1.1 Introduction 2

1.2 Presentation of the Host Organization 2

1.2.1 IZICAP 2

1.2.2 Strategic Partnerships 3

1.3 Organizational Structure 4

1.4 Project Framework 5

1.5 System Architecture - Smart Data 6

1.6 Problem and Functional Needs 8

1.7 Internship Objectives 9

1.8 Project Implementation Process 9

1.9 Project Planning 10

Conclusion 11

2 Decision Support and Solution Choices 12

Introduction 12

2.1 Difference between a data warehouse, a data lake, a datalakehouse, and Delta Lake . 12

2.1.1 Data Warehouse 13

2.1.2 Data Lake 13

2.1.3 Datalakehouse 13

2.1.4 Delta Lake (Datalakehouse) . . . 14
2.2 Difference between Microservices Architecture and Monolithic Architecture . . . 14
2.2.1 Advantages and Disadvantages of Monolithic Architecture . . . 15
2.2.2 Advantages and Disadvantages of Microservices Architecture . . . 16
2.3 Difference between Microfrontends Architecture and Monolithic Frontend Architecture 17
2.3.1 Advantages and Disadvantages of Monolithic Frontend Architecture . . . 17
2.3.2 Advantages and Disadvantages of Microfrontends Architecture . . . 18
2.4 Solution Approach . . . 19
2.4.1 Data Part . . . 19
2.4.2 Backend Part . . . 19
2.4.3 Frontend Part . . . 19

3 Technologies Used **21**
3.1 Delta Lake . . . 21
3.2 Key Benefits and Features of Delta Lake . . . 22
3.3 Trino . . . 24
3.4 Spring Boot . . . 25
3.5 Keycloak . . . 25
3.6 Kafka . . . 26

4 Solution Implementation **28**
Introduction . . . 28
4.1 Feasibility Study . . . 28
4.2 Benchmark between Trino and Spark . . . 29
4.2.1 Architecture . . . 29
4.2.2 Execution Time . . . 30
4.2.3 Memory Management . . . 31
4.2.4 Ecosystem and Integrations . . . 32
4.2.5 Scalability . . . 32
4.2.6 Support for Delta Lake Features . . . 32
4.2.7 Ease of Use . . . 32
4.2.8 Other Statistics . . . 32
4.3 Solution Design . . . 33
4.4 Advantages . . . 34
4.5 Visualization . . . 35
4.5.1 Trino Console . . . 35
4.5.2 Microservices . . . 37
4.5.3 Delta Lake . . . 41
4.5.4 Izicap UI . . . 42

Conclusion **45**

Bibliography **46**

List of Figures

1.1 Strategic Partnerships . . . 4
1.2 IZICAP Organizational Structure . . . 4
1.3 Smart data Izicap . . . 6
1.4 Current System Architecture . . . 7
1.5 Scrum Methodology . . . 10
1.6 Gantt Chart . . . 11

2.1 Data Warehouse vs Data Lake vs Data Lakehouse . . . 14
2.2 Monolithic Architecture vs Microservices Architecture . . . 17
2.3 Monolithic Frontend Architecture vs Microfrontends Architecture . . . 19

3.1 Delta Lake multi-hop architecture . . . 22
3.2 Overview of Trino architecture with coordinator and workers . . . 24
3.3 Keycloak Overview . . . 26
3.4 Kafka Architecture . . . 27

4.1 TRINO Architecture . . . 29
4.2 SPARK Architecture . . . 30
4.3 Solution Architecture . . . 34
4.4 Trino Console . . . 35
4.5 Trino SELECT Query . . . 36
4.6 Trino command prompt . . . 36
4.7 Swagger UI of Endpoint 1 . . . 37
4.8 Swagger UI of Endpoint 2 . . . 38
4.9 Swagger UI of Endpoint 3 . . . 39
4.10 Swagger UI . . . 40
4.11 Parquet Data . . . 41
4.12 Izicap Portal . . . 42
4.13 Customer Revenue . . . 42
4.14 Customer Base Evolution . . . 43
4.15 Transactions by Amount . . . 43

List of Tables

1 List of Abbreviations xi

4.1 Technical Specifications of "DELL Latitude 5530" PC 31
4.2 Technical Specifications of ‘ASUS ROG G752 VT’ PC 31
4.3 Technical Specifications of ‘ASUS ROG G752 VT’ PC 31

API	Application programming interface
VSC	Visual Studio Code
DL	Delta Lake
REST	RepresEntational State Transfer
AWS	Amazon Web Services
GCP	Google Cloud Platform
AZR	Microsoft Azure
S3	Simple Storage Service
IAM	Identity and Access Management
MinIO	Minimal Object Storage
SQL	Structured Query Language
DB	Database
ACID	Atomicity, Consistency, Isolation, Durability
OLTP	Online Transaction Processing
OLAP	Online Analytical Processing
ReactJS	React JavaScript
SPA	Single Page Application
JS	JavaScript
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
UI	User Interface
API	Application Programming Interface
Spark	Apache Spark
Hadoop	Apache Hadoop
HDFS	Hadoop Distributed File System
YARN	Yet Another Resource Negotiator
MapReduce	MapReduce Programming Model
Metadata	Data about Data
ETL	Extract, Transform, Load
ELT	Extract, Load, Transform
CRM	Customer Relationship Management
RDBMS	Relational Database Management System
SPI	Server Provider Interface

Table 1: List of Abbreviations

General Introduction

In a constantly evolving digital world, businesses face major challenges in keeping their applications at the forefront of technology and meeting the growing expectations of users. Monolithic architecture has limitations in terms of flexibility, maintainability, performance, and modularity, which has led many organizations to adopt microservices-based architectures.

This report highlights the approach taken to transform a monolithic Groovy architecture into a microservices-based architecture using Java, while migrating from AngularJS to React and adopting a microfrontend approach. The main goal of this transformation was to improve the flexibility, maintainability, performance, data consistency, and modularity of the application.

The first step was to analyze the existing monolithic architecture and identify components that could be isolated into independent microservices. In parallel, an assessment of the benefits of migrating from AngularJS to React was conducted.

The design and implementation of the microservices were done using Spring Boot, ensuring decoupling of functionalities and ensuring efficient communication between services. The AngularJS code was gradually migrated to React using best practices in React development.

To ensure data consistency among the microservices, the incremental data management technology Delta Lake was used. Java connectors facilitated communication between the React-based frontend and Delta Lake, ensuring a robust and secure connection.

In parallel, a microfrontend approach was adopted for the frontend, allowing functionalities to be decoupled into autonomous modules.

This report will detail each step of the transformation, focusing on the decisions made, challenges encountered, and results achieved. It will highlight the benefits of microservices-based architecture, the use of React and microfrontend, as well as the use of Kubernetes and Docker. This case study provides valuable insights into application modernization and best development practices in a constantly changing environment.

Chapter 1

General project context

1.1 Introduction

The objective of this section is to present the general context of the project, starting with the host organization, IZICAP, where I am completing my 5-month internship. We will describe their activities, strategy, values, as well as the organizational structure and profiles involved in my project. Then, we will address the work request that expresses the functional requirements on which I based the design and implementation of the solution, as well as the methodology followed to achieve the set objectives.

1.2 Presentation of the Host Organization

1.2.1 IZICAP

IZICAP is a unique and innovative company focused on data analysis, loyalty, and digital marketing for small and medium-sized enterprises (SMEs). Their user-friendly and easy-to-install solution is based on the SaaS model, allowing for the utilization of customers' transactional data from payment cards.

By collaborating with banks and acquirers, IZICAP offers an enhanced value proposition and provides a powerful digital marketing tool to SMEs. Their mission is to support merchants on a daily basis to accelerate the development of their businesses, thereby addressing their specific needs.

IZICAP's story dates back to 2013 when it emerged in an ever-evolving and increasingly complex landscape of financial services. Since then, traditional banking and payment models have been continually threatened by new, more innovative challenger companies. Intense competition means that merchant loyalty can no longer be taken for granted, and often acquirers and payment service providers are compelled to reduce their fees.

After establishing a strong presence in France through partnerships with Groupe BPCE and Crédit Agricole, Izicap formed a partnership with Nexi, the leading acquirer and fintech in Italy, and joined Mastercard's StartPath program with the aim of significantly expanding its global reach. This strategic expansion strengthens their position in the market and confirms their commitment to offering their services to a wider audience.

With Izicap, SMEs have access to an innovative solution that leverages data analysis to optimize their loyalty and marketing strategy. With their expertise and growing international presence, IZICAP is determined to support business growth and help them thrive in an ever-changing commercial environment.

1.2.2 Strategic Partnerships

IZICAP has established strategic partnerships with several key players in the financial industry. Among their notable partners are:

- **Groupe BPCE:** Izicap collaborated with Groupe BPCE, one of the leading banking groups in France, to provide their loyalty and digital marketing solution to merchants.

- **Crédit Agricole:** Izicap partnered with Crédit Agricole, one of the largest French banks, to offer their services to merchants within their network.

- **Nexi:** Izicap formed a partnership with Nexi, the leading acquirer and fintech company in Italy. This collaboration allows Izicap to expand its presence in the Italian market and benefit from Nexi's expertise in the field of payment services.

- **Mastercard StartPath:** Izicap joined Mastercard's StartPath program, which supports startups and innovative companies in the fintech field. This collaboration provides Izicap with an opportunity to increase its visibility and expand internationally.

- **BBVA:** Izicap also established a partnership with BBVA, one of the major banks in Spain and Latin America. This collaboration with BBVA strengthens Izicap's presence in the Spanish market and enables the company to offer its loyalty and digital marketing solutions to merchants affiliated with BBVA. The partnership with BBVA demonstrates Izicap's commitment to expanding its reach and working with major players in the financial industry to support small and medium-sized enterprises in their growth.

- **Ingenico:** Izicap partnered with Ingenico, one of the leading providers of payment solutions and merchant services globally. Through this partnership, Izicap is able to integrate its loyalty and digital marketing solution into Ingenico's payment terminals, providing a comprehensive solution to merchants using Ingenico's services. This collaboration strengthens Izicap's position as a provider of innovative loyalty and marketing solutions, leveraging Ingenico's expertise and global reach in the payments industry.

Figure 1.1: Strategic Partnerships

1.3 Organizational Structure

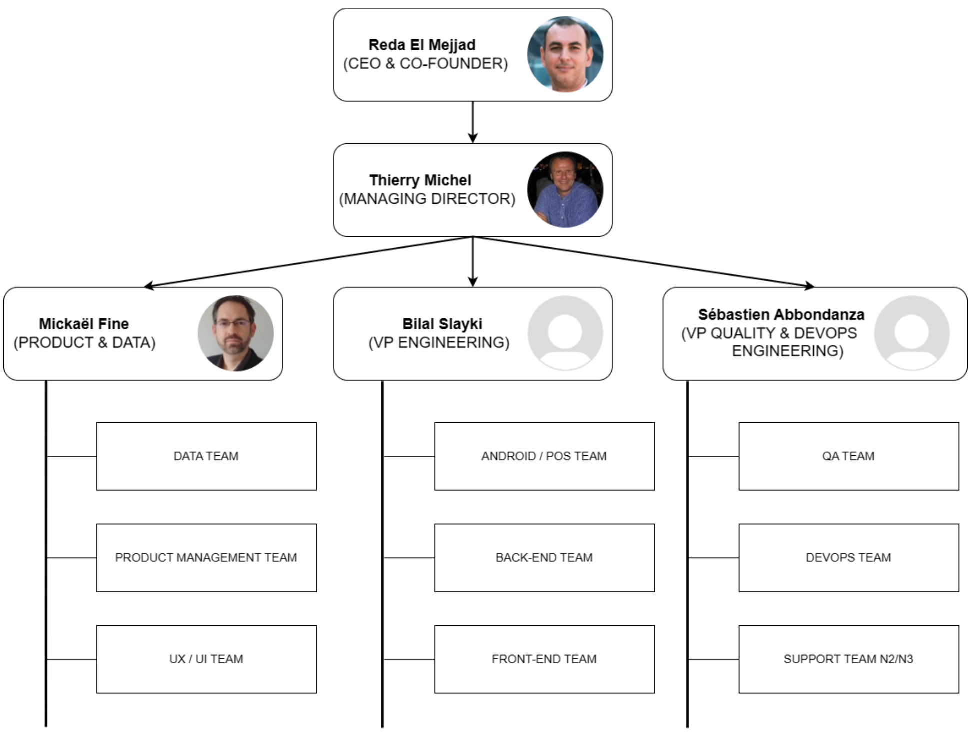

Figure 1.2: IZICAP Organizational Structure

Within the project team, I had the opportunity to work in a versatile manner across different domains. I actively contributed to backend development by participating in the design and implementation of microservices. I also collaborated with the frontend team, bringing my expertise in integrating microfrontends and improving the user experience. Additionally, I was involved in data management, contributing to the implementation of Delta Lake and optimizing data flows. This diverse experience allowed me to gain a holistic view of the project and effectively collaborate with different teams to achieve the set objectives.

1.4 Project Framework

The project involves the revamping of the "Smart data" application, which refers to the process of extracting valuable information from large amounts of data to make informed business decisions.

Smart data allows us to:

1. **Data collection**: Izicap helps businesses collect customer data from various touchpoints such as point-of-sale systems, loyalty programs, online interactions, etc. Ensure the integration of their data collection tools into your existing systems or processes.

2. **Data analysis**: Izicap's smart data solution enables you to analyze the collected data to discover trends, patterns, and customer behaviors. Use their analytics tools and algorithms to gain actionable insights from the data.

3. **Customer segmentation**: Segment your customer base based on their preferences, purchase history, demographic characteristics, or other relevant criteria. Izicap's smart data solution can assist you in identifying different customer segments and creating personalized marketing strategies for each segment.

4. **Personalized marketing**: Utilize the insights derived from Izicap's smart data solution to create targeted marketing campaigns. Send personalized offers, promotions, or recommendations to specific customer segments, thereby increasing the chances of conversion and customer satisfaction.

5. **Customer loyalty**: Leverage Izicap's smart data solution to identify customers who are at risk of churning or no longer making purchases from you. Develop loyalty strategies by offering incentives, loyalty programs, or personalized communications to keep them engaged and loyal to your brand.

6. **Performance tracking**: Regularly monitor the performance of your marketing campaigns and customer engagement efforts. Izicap's solution can provide you with metrics and reports to evaluate the effectiveness of your strategies and make data-driven adjustments when necessary.

7. **Continuous improvement**: Smart data solutions are most effective when used iteratively. Regularly analyze data, adapt your strategies, and refine your approach based on new insights and evolving customer behavior.

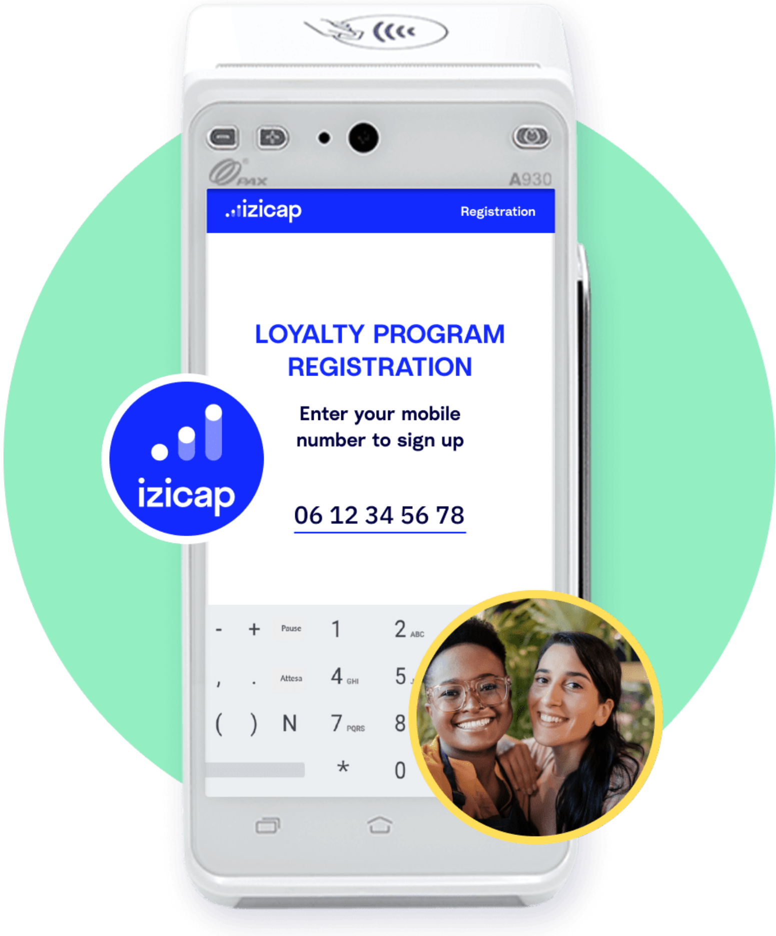

Figure 1.3: Smart data Izicap

1.5 System Architecture - Smart Data

The figure below visualizes the structure and key components of our system, as well as the interactions between them. It forms the foundation of our system and plays a crucial role in providing features and services to our users.

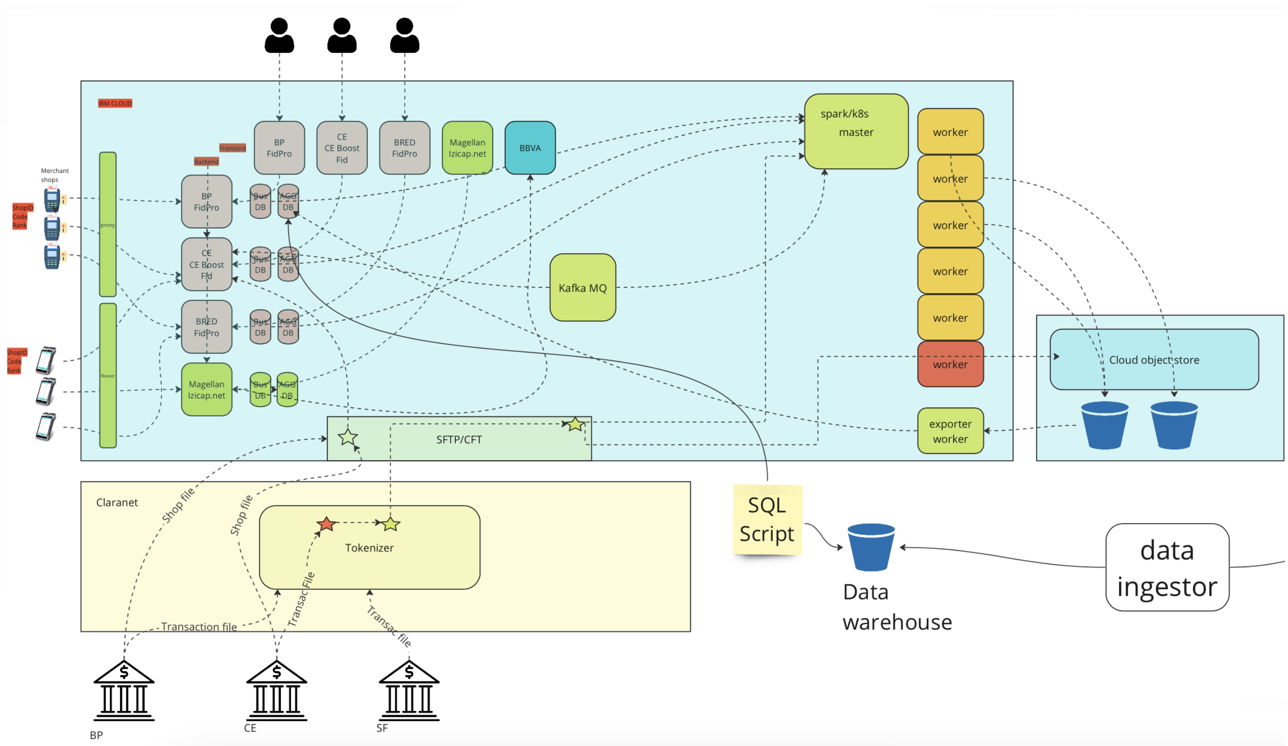

Figure 1.4: Current System Architecture

This diagram represents the functioning of the current monolithic architecture, which has several disadvantages that can hinder the efficiency, flexibility, and maintainability of the system. Here is a detailed description of the main drawbacks:

1. **Complexity and dependencies**: The monolithic architecture involves grouping all modules and functionalities of the system into a single block. This creates strong dependencies between different components, making understanding, management, and updates complex. Changes made to one part of the system can have repercussions on other parts, making testing, deployment, and bug fixing more challenging.

2. **Limited scalability**: Monolithic architecture often struggles to adapt to increased load or growing demand. Since all components are bundled into a single monolith, it is difficult to selectively scale a specific part of the system. This can lead to bottlenecks, reduced performance, and poor resource allocation when dealing with large data volumes or managing an increase in the number of users.

3. **Complex deployments and high risks**: With a monolithic architecture, deployments require updating the entire system, even for minor changes. This increases the risks of errors and regressions, as a single mistake can result in the unavailability of the entire system. Additionally, deployments need careful planning and coordination, which can lead to longer downtimes and service interruptions for users.

4. **Technological choice challenges**: In a monolithic architecture, the technologies used are often tightly coupled and integrated. This can make it difficult to adopt new technologies or integrate specialized components. Upgrading versions or adding new features can be limited by initial technological choices, hindering innovation and adaptation to market changes.

5. **Cohesion and responsibilities**: Monolithic architecture does not facilitate a clear separation of responsibilities and functionalities. Different modules of the system are often tightly coupled and may share common features. This makes isolating issues, module-specific maintenance, and code reuse specific to a domain challenging.

1.6 Problem and Functional Needs

The problem at hand lies in the currently used monolithic architecture, which presents limitations in terms of scalability, flexibility, and maintainability. The constraints imposed by this architecture make it complex to deploy new functionalities and can potentially disrupt the entire system. Additionally, the use of a traditional relational database such as MariaDB proves insufficient for efficiently handling large volumes of transactional data. It is worth noting that the Smart Data application has been in existence for 10 years and has been developed using the Groovy language. This long period of existence attests to the stability and maturity of our system. However, the use of Groovy may present certain constraints in terms of maintainability and scalability, especially when it comes to integrating new technologies and managing more modern architectures.

Functional Needs

To address these challenges, different functional needs have been identified:

- **Scalability**: It is necessary to have an architecture that can easily adapt to a growing number of users, transactions, and data. It is essential that our system can scale harmoniously and maintain its performance even with a significant increase in workload.
- **Flexibility**: We need to be able to introduce new functionalities independently and deploy them without disrupting the entire system. An approach based on microservices and microfrontends will enable us to achieve this flexibility by facilitating the development, testing, and isolated deployment of each component.
- **Improved Performance**: It is crucial to have a data infrastructure capable of efficiently handling large volumes of transactional data. By replacing MariaDB with Delta Lake, we will be able to leverage advanced features such as ACID transaction management and compatibility with powerful analysis tools, significantly improving the performance and reliability of our system.
- **Separation of Responsibilities**: We aim for better separation of responsibilities between different components of our system. Microservices will allow us to decouple functionalities, assign them to specific teams, and promote more effective code management, simplified maintenance, and increased scalability.

1.7 Internship Objectives

The project falls within the scope of renewing the functionalities of the Smart Data application, aiming to improve its scalability, flexibility, and performance. In this context, the objectives of the internship are as follows:

1. Study and analyze the current architecture of the SMART DATA application to understand the constraints and limitations that hinder its growth and evolution.
2. Propose and design a migration strategy from the monolithic architecture to an architecture based on microservices and microfrontends, thus enabling better modularity and greater flexibility in the development and deployment of functionalities.
3. Implement multiple microservices with Spring Boot within the new architecture, using appropriate technology and ensuring its seamless integration with other components of the system.
4. Evaluate the benefits and implications of using Delta Lake as a replacement for the MariaDB database, with a focus on performance, transaction management, and integration with analysis tools.
5. Integrate Trino into the architecture to enable the execution of complex SQL queries and optimize data processing, ensuring effective collaboration with the development team.
6. Set up a deployment and management infrastructure for microservices, using tools such as Kubernetes, Docker, and Jenkins, to facilitate deployment, scaling, and component management.
7. Design and implement unit tests and integration tests to ensure the quality and reliability of the new components developed within the microservices-based architecture.
8. Thoroughly document the migration process and the technological choices made, providing clear instructions for the future maintenance of the architecture and management of updates.
9. Collaborate closely with the existing team to facilitate the transition to the new architecture, providing technical support, guidance, and training on the new technologies used.

1.8 Project Implementation Process

The project implementation process followed several iterations called "sprints," typically lasting two weeks each. Each sprint focused on delivering functional increments of the application, enabling quick tangible results.

Here are the key steps of the project implementation process based on the Scrum methodology:

1. **Product backlog definition:** In collaboration with stakeholders and the development team, we identified and prioritized the features to be developed and integrated into the microservices-based architecture.

2. **Sprint planning:** At the beginning of each sprint, we held a planning meeting to define the specific sprint goals, select tasks to be accomplished, and estimate the required effort.

3. **Iterative development:** The development team worked iteratively on the assigned tasks, focusing on accomplishing the identified features for the current sprint.

4. **Daily stand-up meetings:** Every day, the team gathered for a brief stand-up meeting to share progress, potential obstacles, and coordinate activities.

5. **Sprint review:** At the end of each sprint, we conducted a sprint review to showcase the developed features and gather feedback from stakeholders. This allowed us to validate the achieved results and plan for the next steps.

6. **Sprint retrospective:** After the sprint review, we conducted a retrospective to evaluate the sprint's progress, identify strengths and areas for improvement, and adjust our approach accordingly.

7. **Subsequent iterations:** The planning, iterative development, sprint review, and retrospective process repeated for each subsequent sprint, enabling incremental progress towards the project's objectives.

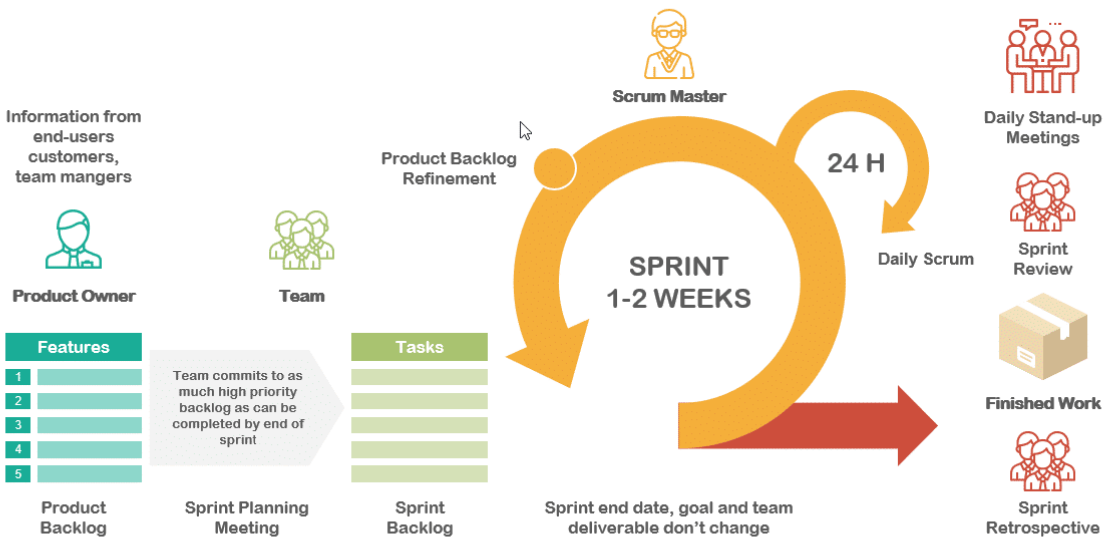

Figure 1.5: Scrum Methodology

1.9 Project Planning

For the project planning, we used the method of modeling the network of task dependencies. This approach allowed us to break down the work into different structured tasks and establish dependency relationships between them.

We also utilized the technique of Gantt chart to graphically represent the project's tasks and resources over time. In the horizontal axis, we listed the different tasks, and in the vertical axis, we defined the days, weeks, or months. Each task was represented by a bar whose length was proportional to the estimated duration of that task.

The Gantt chart enabled us to visualize the distribution of tasks, their duration, and their sequence. Some tasks were performed sequentially, while others could be done in parallel, either partially or entirely. This clear and visual representation helped us plan the project by determining and organizing the different tasks to ensure effective project management.

The following figure presents the Gantt chart detailing the project's planning, with tasks arranged over time and their respective durations. This allowed us to track the project's progress, identify any delays, and take appropriate measures to resolve them.

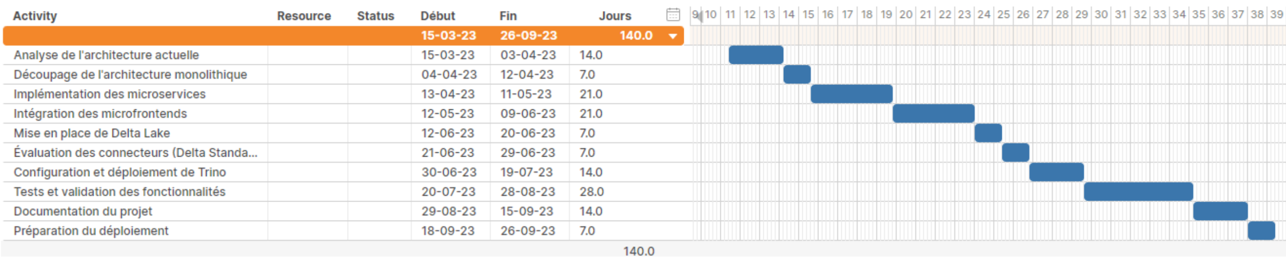

Figure 1.6: Gantt Chart

Conclusion

In conclusion of this section, we have presented the project planning for renewing the functionalities of the SMART DATA application. We developed a schedule based on the modeling of the network of task dependencies, using the technique of the Gantt chart. This chart allowed us to visualize the different activities, their estimated durations, and the required resources.

During the planning, we identified the main project activities, such as the analysis of the current architecture, decomposition of the monolithic architecture, implementation of microservices, integration of microfrontends, implementation of Delta Lake, evaluation of connectors, configuration and deployment of Trino, testing and validation of functionalities, project documentation, deployment preparation, as well as user training and awareness.

We also took into account the total duration of the project, which is estimated to be 5 months, and adjusted the durations of activities accordingly. This provided us with a more precise view of the project's temporal planning.

Chapter 2

Decision Support and Solution Choices

Introduction

Decision Support encompasses the processes, technologies, and tools used to collect, store, analyze, and present data in order to support decision-making and help businesses gain actionable insights. Decision suuport involves transforming raw data into meaningful information and actionable knowledge for decision-makers. In this chapter, we will explore why we have chosen to adopt Delta Lake instead of its counterparts, as well as microservices and microfrontends.

2.1 Difference between a data warehouse, a data lake, a datalakehouse, and Delta Lake

- **Data Warehouse:** A data warehouse is a centralized database specifically designed for reporting and analysis. It stores structured data from different sources, organizes it according to a predefined data model, and optimizes it for analytical queries. Data in a data warehouse is typically consistent, integrated, and historical. However, building and maintaining a data warehouse can be complex and costly.

- **Data Lake:** A data lake is a centralized data repository that stores large amounts of raw, structured, and unstructured data. Unlike a data warehouse, a data lake does not require prior data modeling. It offers great flexibility and scalability for storing heterogeneous data. However, data integration and data quality can be challenging in a data lake.

- **Datalakehouse:** Datalakehouse is an emerging architecture that combines the benefits of a data warehouse and a data lake. It enables storing and processing both raw and structured data in a centralized environment. This hybrid approach provides the flexibility of a data lake and the analytical capabilities of a data warehouse. However, implementing a datalakehouse may require additional efforts to ensure data quality and query efficiency.

- **Delta Lake:** Delta Lake is a technology that integrates with existing data lakes to provide additional features such as ACID (Atomicity, Consistency, Isolation, Durability) transaction management, incremental updates, and data consistency guarantees. Delta Lake is built on Apache Parquet and Apache Arrow, which accelerate analytical queries and improve overall performance. However, using Delta Lake may require additional technical skills and may impact the complexity of the data architecture.

2.1.1 Data Warehouse

Advantages:

1. Consistent and integrated data
2. Predefined data modeling for optimized analysis
3. High performance for analytical queries

Disadvantages:

1. High construction and maintenance costs
2. Complexity of data modeling
3. Limitations for integrating unstructured data

2.1.2 Data Lake

Advantages:

1. Economical storage of large amounts of data
2. Flexibility to integrate raw and unstructured data
3. Ability to process data from different sources

Disadvantages:

1. Difficulty in maintaining data quality and governance
2. Need for advanced tools for data analysis and processing
3. Requires technical skills for efficient data exploitation

2.1.3 Datalakehouse

Advantages:

1. Combines the benefits of a data warehouse and a data lake
2. Flexibility to store and analyze raw and structured data
3. Ability to evolve based on evolving needs

Disadvantages:

1. Requires additional efforts for data quality
2. Increased complexity of the data architecture
3. Requires technical skills for setup and management

2.1.4 Delta Lake (Datalakehouse)

Advantages:

1. ACID transaction management for data consistency
2. Support for incremental updates and stream processing
3. High performance for analytical queries

Disadvantages:

1. Requires specific technical skills
2. Impact on the complexity of the existing data architecture
3. May require adaptations for seamless integration with existing tools

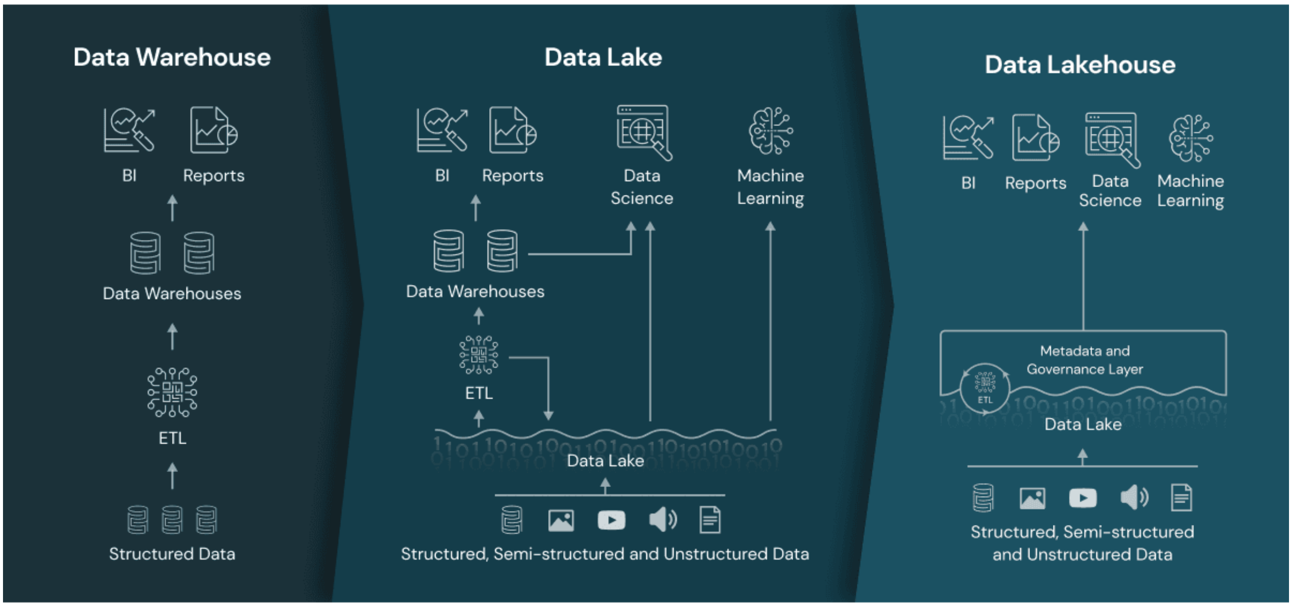

Figure 2.1: Data Warehouse vs Data Lake vs Data Lakehouse

2.2 Difference between Microservices Architecture and Monolithic Architecture

- **Monolithic:** A monolith is an application that is designed as a single, indivisible entity. All the application's functionalities are grouped together in a single code base, sharing the same resources, databases, and deployments. In a monolithic architecture, there is no clear separation of functionalities into independent services. Any modification or evolution of the application requires changes at a global level.

- **Microservice:** A microservice is an architectural approach in which an application is built as a set of independent and autonomous services, each focusing on a specific function of the application. Each microservice is developed, deployed, and managed independently, allowing for easier scalability, flexibility, and maintenance. Microservices communicate with each other through well-defined interfaces, typically based on REST APIs or messaging.

2.2.1 Advantages and Disadvantages of Monolithic Architecture

Advantages:

1. **Simplicity of initial development:** The monolithic approach allows for rapid development of an application by grouping all functionalities into a single code base. This simplifies dependency management and coordination of different parts of the application.

2. **Less operational complexity:** With a monolithic architecture, there are fewer components and services to manage, simplifying deployment, monitoring, and management operations. Everything is bundled within a single deployment, which can be easier to handle for operational teams.

3. **Faster internal communications:** In a monolithic architecture, communications between different parts of the application are faster as they typically occur through internal method calls. This can be beneficial in terms of performance and reduced latency.

Disadvantages:

1. **Difficulty in scaling and maintaining:** With a monolithic architecture, scaling and updates can be more complex since every change needs to be made to the entire application. This can slow down the development process and pose risks of errors during deployments.

2. **Technological rigidity:** A monolithic architecture can result in technological rigidity as all parts of the application must use the same technologies and programming languages. This can limit the opportunities to adopt new technologies or independently evolve specific parts of the application.

3. **Difficulty in isolating issues:** In case of issues or bugs, it can be more challenging to isolate and resolve them in a monolithic architecture. Since all functionalities are grouped in a single code base, pinpointing the exact origin of the problem can be complex.

4. **Less flexibility in terms of scalability:** Monolithic architecture can pose challenges in terms of scalability. If a part of the application requires more resources to handle high load, adjusting that specific part without increasing the overall resources of the application can be difficult.

2.2.2 Advantages and Disadvantages of Microservices Architecture

Advantages:

1. **Scalability and evolvability:** Microservices allow breaking down the application into several autonomous and independent services, facilitating horizontal scalability. Each microservice can be deployed, scaled, and updated independently, efficiently managing load variations and ensuring easy scalability according to business needs.

2. **Technological flexibility:** Microservices offer the ability to use different technologies and programming languages for each service. In our case, using Spring Boot allows us to leverage its rich ecosystem and advanced features for rapid application development. It also enables adopting specific technologies based on the needs of each microservice, promoting technological flexibility.

3. **Independence and autonomy:** Each microservice is designed to operate autonomously, allowing better isolation of functionalities and responsibilities. This facilitates maintenance, testing, and independent deployment of services, reducing the risks of impacting the entire system in case of changes or issues.

4. **Rapid development and deployment:** Microservices enable an agile development approach by emphasizing shorter development cycles. Teams can focus on specific features and develop them independently, accelerating the overall system development. Additionally, microservices can be continuously deployed through the use of automated deployment techniques, facilitating frequent and rapid updates.

5. **Ease of maintenance and debugging:** Due to their modular nature, microservices facilitate system maintenance and debugging. In case of problems or errors, it's easier to identify the specific service involved and resolve the issue without impacting the entire system.

Disadvantages:

1. **Complexity in managing communications:** Microservices involve communication between different services, typically through REST APIs or messaging. Managing these communications can become complex, especially when numerous services are involved. Issues such as latency, data consistency, and error handling may arise.

2. **Increased initial development overhead:** Developing microservices requires additional effort to properly decompose functionalities, define interfaces, and set up appropriate infrastructure for service deployment and communication. This can increase the initial workload and require specific skills in distributed architecture.

3. **Managing data consistency:** With microservices, each service can have its own database or data storage. This can make managing data consistency more complex, particularly when simultaneous updates involving multiple services occur. Techniques such as distributed transactions or asynchronous events may be required to maintain data consistency.

4. **Deployment and management of multiple services:** With microservices, there is an increased number of services to deploy, manage, and monitor. This may require additional skills in automated deployments, container management, or cluster management. Monitoring the performance and behavior of each service can also become more complex.

5. **Infrastructure cost:** Microservices may require more complex infrastructure and additional resources to operate effectively. Each service needs to be deployed and run independently, which can result in increased costs related to computing resources and infrastructure management.

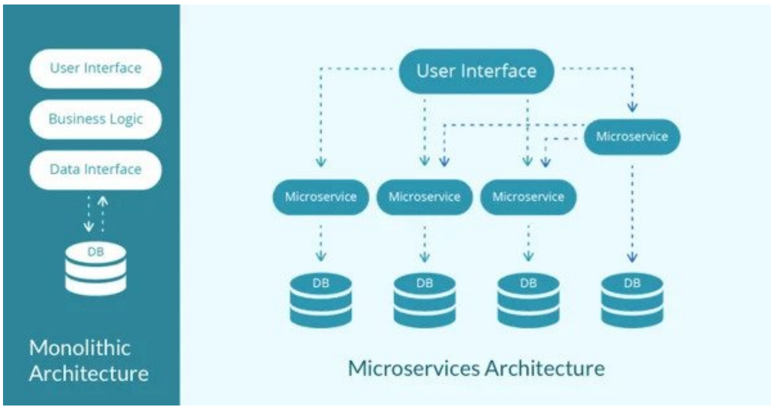

Figure 2.2: Monolithic Architecture vs Microservices Architecture

2.3 Difference between Microfrontends Architecture and Monolithic Frontend Architecture

- **Monolithic Frontend:** A monolithic frontend architecture is an architectural approach where the frontend application is developed as a single entity, typically using a specific framework such as AngularJS. All the features, views, and user interface logic are grouped together in a single code base.

- **Microfrontends:** Microfrontends are an architectural approach where a frontend application is divided into multiple independent micro-applications, each responsible for a specific part of the user interface. Each micro-application can be developed, deployed, and evolved independently, using different frameworks, languages, and technologies.

2.3.1 Advantages and Disadvantages of Monolithic Frontend Architecture

Advantages:

1. **Simplicity of initial development:** The monolithic architecture with AngularJS provides a straightforward approach for the initial development of the frontend application. All the features are grouped together in a single code base, making it easier to coordinate and manage the development.

2. **Ease of communication between components:** In a monolithic architecture, AngularJS components can communicate with each other easily through the AngularJS directives and services system. This allows for quick and efficient communication between different parts of the application.

3. **Interoperability of features:** Since all the features are developed using AngularJS, it is easier to share features and modules between different parts of the application. This promotes code reuse and simplifies maintenance.

Disadvantages:

1. **Difficulty in maintenance and scalability:** As the frontend application becomes more complex, maintaining and evolving the monolithic architecture with AngularJS can become challenging. Changes to one part of the application can have implications on the entire code base, making the development process slower and riskier.

2. **Performance limitations:** In a monolithic architecture, all the features are loaded together, which can lead to performance issues if the application becomes large. Loading times can be longer, and the application may be less responsive for the user.

3. **Limited flexibility:** The monolithic architecture with AngularJS can limit technological flexibility. Since all parts of the application are developed using AngularJS, it can be difficult to introduce new technologies or independently evolve specific parts of the application.

2.3.2 Advantages and Disadvantages of Microfrontends Architecture

Advantages:

1. **Independence of development teams:** Each micro-application can be developed by a separate team, promoting greater autonomy and better collaboration among development teams. Each team can choose the technologies that best suit their needs.

2. **Scalability and ease of maintenance:** Microfrontends architecture allows for independent scaling and maintenance of different parts of the application. Changes to one micro-application do not impact others, making maintenance easier and enabling rapid deployment of new features.

3. **Technological flexibility:** Each micro-application can use the technology, framework, or programming language that best suits its specific domain. This allows for the introduction of new technologies and leveraging the benefits of the latest developments in software engineering.

Disadvantages:

1. **Increased complexity:** Microfrontends architecture introduces some complexity in the development and deployment of the application. Managing interactions and communication between different micro-applications may require additional planning and coordination.

2. **Higher initial development cost:** Developing a microfrontends architecture may require a higher initial investment in terms of resources and time. Developing multiple separate micro-applications and setting up the necessary infrastructure can be more costly than developing a monolithic application.

3. **Network overhead:** Using microfrontends architecture can result in increased network overhead as each micro-application requires separate requests and resource loading. This can impact application performance and require efficient network management.

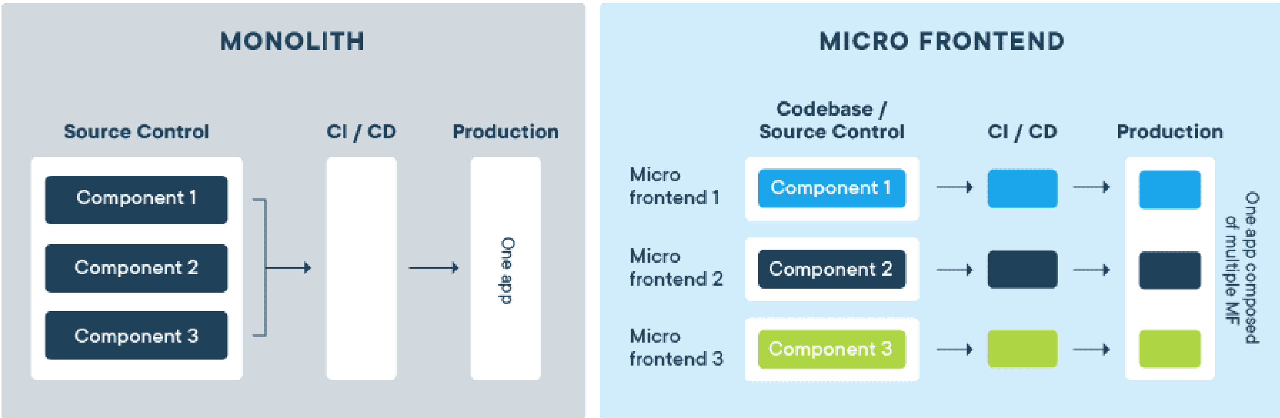

Figure 2.3: Monolithic Frontend Architecture vs Microfrontends Architecture

2.4 Solution Approach

2.4.1 Data Part

In the context of our solution, we have chosen to replace the existing exporter worker and MariaDB databases with the use of a data lakehouse architecture. A data lakehouse is a hybrid approach that combines the benefits of a data warehouse and a data lake, providing a more flexible and scalable solution for data management.

2.4.2 Backend Part

For the implementation of microservices, we have chosen to use Spring Boot, a popular Java framework for application development. By using Spring Boot, we can create standalone microservices that are independent of each other and can be developed, deployed, and scaled individually. Spring Boot also provides features such as data persistence management, security, and REST API creation, making it easier to develop microservices.

2.4.3 Frontend Part

For the implementation of microfrontends, we have primarily chosen to use React. The development team can benefit from the rich and mature React ecosystem, as well as its increasing popularity in the frontend development industry. However, it is also mentioned that Angular can be used later if needed, providing additional flexibility in the choice of technologies.

Conclusion

When evaluating different data architectures for Izicap, it is important to understand the specific needs related to the management of bank files, transaction receipts, and aggregation operations.

A data warehouse could have been a viable option, offering structured and optimized data structures for analytical queries. However, the main drawback of a data warehouse lies in its static nature, which requires pre-modeling of data and rigid transformation before loading. This can pose challenges when integrating new file types or evolving aggregation requirements.

On the other hand, a data lake has advantages in terms of cost-effective storage and flexibility to integrate raw and unstructured data. However, maintaining data quality and governance can be more complex, and specific technical skills are required to effectively leverage data from the data lake.

Therefore, Delta Lake was favored due to its ability to meet Izicap's specific needs for managing bank files, transaction receipts, and aggregation operations. It provides the necessary flexibility and performance while maintaining data integrity, making it a solid choice for the company's data architecture.

By adopting a microservices-based approach, Izicap can improve the flexibility, scalability, and maintenance of its frontend application. This will enable the company to better meet the changing needs of its users, facilitate collaboration between development teams, and adopt innovative technologies to deliver an optimal user experience.

Microfrontends offer several advantages for Izicap. Firstly, the modularity inherent in microfrontends allows for the development, deployment, and maintenance of different parts of the user interface independently. This promotes collaboration between development teams and enables rapid and efficient evolution. Each micro-application can be developed using the framework and technologies that best suit its specific needs, providing essential technological flexibility for Izicap.

In contrast, the monolithic approach has limitations in terms of scalability and technological flexibility. Changes made to one part of the application can impact the entire system, making evolutions more complex and risky. Additionally, introducing new technologies or frameworks can be challenging in a monolithic architecture, limiting opportunities for innovation and modernization.

Chapter 3

Technologies Used

Introduction

In this chapter, we will examine in detail the key technologies that are used in our solution to provide advanced features and meet the specific needs of our project. The three main technologies we will cover are Delta Lake, Trino, and Microfrontends.

3.1 Delta Lake

Delta Lake is a data management technology that enables efficient and reliable storage, management, and analysis of massive volumes of data. It is built on a file-based Parquet architecture and offers advanced features such as ACID (Atomicity, Consistency, Isolation, Durability) transaction management and compatibility with popular analytics tools. Delta Lake also ensures data integrity, query consistency, and supports replication and recovery in case of failures.

The concept of a 'lakehouse' is made possible by Delta Lake. It is a data architecture that combines the benefits of data warehouses and data lakes, providing a unique and consistent approach to data management. Data is stored in Parquet format in the data lake, enabling continuous and batch processing.

- **Enables Lakehouse architecture:** Delta Lake enables a continuous and streamlined data architecture that allows organizations to manage and process massive volumes of data in a continuous and batch manner without the hassle of separately managing and operating streaming, data warehouses, and data lakes.

- **Enables intelligent data management for data lakes:** Delta Lake provides efficient and scalable metadata management, which provides insights into massive data volumes in data lakes. With this information, data governance and management tasks can be performed more effectively.

- **Schema enforcement for improved data quality:** Since data lakes don't have a defined schema, it becomes easy for bad/incompatible data to enter the data systems. Data quality is improved through automatic schema validation, which validates DataFrame and table compatibility before writes.

- **Enables ACID transactions:** Most organizational data architectures involve numerous ETL and ELT movements in and out of data storage, which opens it up to more complexity and failure points. Delta Lake ensures data durability and persistence during ETL and other data operations. Delta Lake captures all data changes during data operations in a transaction log, ensuring data integrity and reliability during data operations.

[3]

3.2 Key Benefits and Features of Delta Lake

With Delta Lake, data is stored in an optimized format, such as Parquet, in a data lake. This format enables efficient query processing regardless of the mode of data access, whether it is streaming or batch processing.

[2]

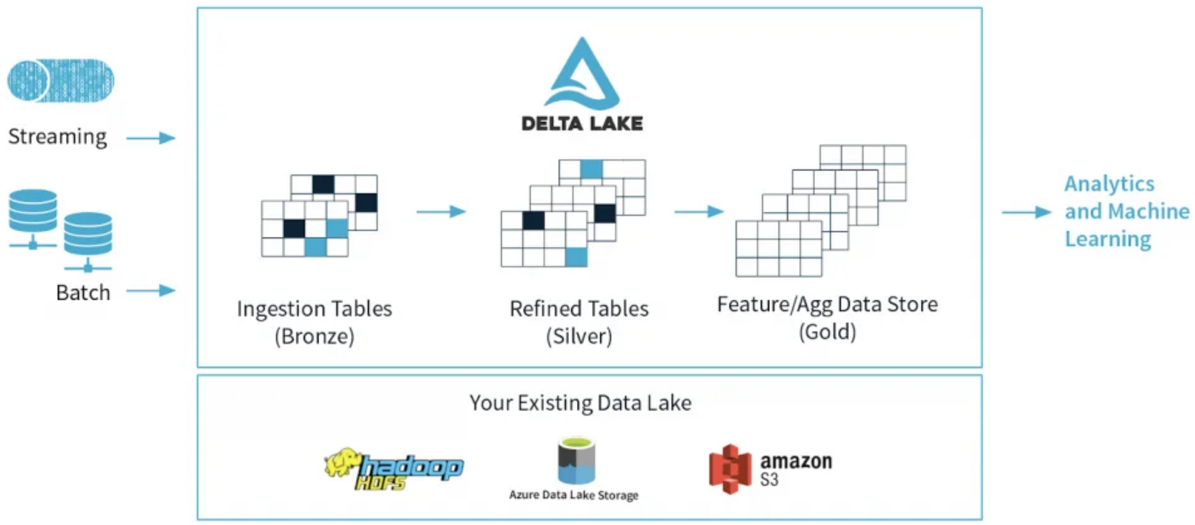

Figure 3.1: Delta Lake multi-hop architecture

- **Audit trails and history:** In Delta Lake, each write exists as a transaction and is sequentially recorded in a transaction log. Therefore, all modifications or validations made to the transaction log are recorded, leaving a complete trail to be used for historical audits, versioning, or time travel purposes. This Delta Lake feature ensures data integrity and reliability for enterprise data operations.

- **Time travel and data versioning:** Since each write creates a new version and stores the old version in the transaction log, users can view/restore old versions of data by providing the timestamp or version number of an existing table or directory to the Spark read API. Using the provided version number, Delta Lake then constructs a complete snapshot of the version with the information provided by the transaction log. Rollbacks and version management play a crucial role in machine learning experimentation, where data scientists iteratively modify hyperparameters to train models and can revert to previous changes if necessary.

- **Unifies batch and stream processing:** Each table in a Delta Lake is both a batch and stream sink. With Structured Streaming in Spark, organizations can efficiently stream and process data. Additionally, with efficient metadata management, scalability, and ACID quality for each transaction, near-real-time analytics becomes possible without using a more complicated two-tier data architecture.

- **Efficient and scalable metadata management:** Delta Lake stores metadata information in the transaction log and leverages the distributed processing power of Spark to quickly process, read, and manage large volumes of data metadata, thereby enhancing data governance.

- **ACID transactions:** Delta Lake ensures that users always see a consistent view of data in a table or directory. It achieves this by capturing every modification made in a transaction log and isolating it at the strongest isolation level, the serializable level. At the serializable level, every existing operation has and follows a serial sequence that, when executed one by one, provides the same result as stated in the table.

- **Data Manipulation Language (DML) operations:** Delta Lake supports DML operations such as updates, deletes, and merges, which play a role in complex data operations such as Change Data Capture (CDC), continuous upserts, and Slowly Changing Dimensions (SCD). Operations like CDC ensure data synchronization across all data systems and minimize time and resources spent on ELT operations. For example, using CDC, instead of ETL-ing all available data, only the recently updated data since the last operation undergoes transformation.

- **Schema enforcement:** Delta Lake performs automatic schema validation by checking a set of rules to determine the compatibility of a DataFrame write to a table. One such rule is the existence of all DataFrame columns in the target table. An occurrence of an extra or missing column in the DataFrame raises an error exception. Another rule is that the DataFrame and the target table must have the same column types, which, if not, triggers an exception. Delta Lake also uses Data Definition Language (DDL) to explicitly add new columns. This data lake feature helps avoid the ingestion of incorrect data, ensuring high data quality.

- **Compatibility with Spark API:** Delta Lake is built on Apache Spark and is fully compatible with the Spark API, enabling the creation of efficient and reliable large-scale data pipelines.

- **Flexibility and integration:** Delta Lake is an open-source storage layer and utilizes the Parquet format for storing data files, which promotes data sharing and facilitates integration with other technologies, fostering innovation.

3.3 Trino

Trino, formerly known as Presto, is a distributed, open-source SQL query engine. It is designed to execute interactive and analytical queries at a large scale on heterogeneous and distributed data. Trino offers great versatility by allowing access to various types of data sources, whether they are relational databases, file systems, real-time data sources, or cloud storage services. With its distributed design, Trino enables high performance and horizontal scalability, making it an essential tool for data analysis in our solution.

[4]

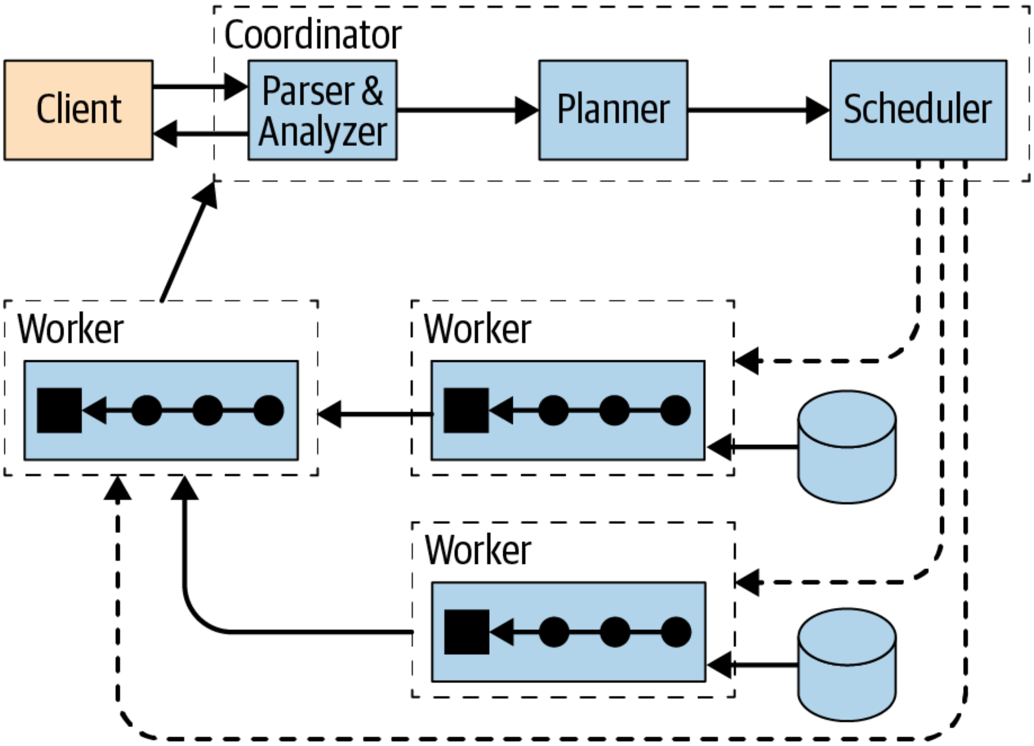

Figure 3.2: Overview of Trino architecture with coordinator and workers

1. A coordinator is a Trino server that handles incoming queries and manages workers to execute the queries.
2. A worker is a Trino server responsible for executing tasks and processing data.
3. The discovery service typically runs on the coordinator and allows workers to register to participate in the cluster.
4. All communications and data transfers between clients, the coordinator, and workers use REST-based interactions over HTTP/HTTPS.

3.4 Spring Boot

Spring Boot is an open-source framework for Java application development. It provides a simplified and opinionated approach to creating standalone, production-ready Java applications without the need for complex configuration.

One of the main advantages of Spring Boot is its ability to reduce boilerplate configuration and simplify application development by providing smart default configuration definitions and automating many development tasks. It also embeds an application server, making it easy to deploy and run the application without requiring an external application server.

Spring Boot follows the annotation-driven programming paradigm, where annotations are used to configure and orchestrate different parts of the application. It offers a wide range of features, such as dependency injection, externalized configuration, error handling, security, data access, etc. These features are bundled into starters, which are pre-defined dependencies that facilitate adding specific functionality to the application.

With its simplified approach, Spring Boot allows developers to focus more on the business logic of their application rather than tedious configuration tasks. It also promotes good development practices, such as separation of concerns and modularity, making applications more maintainable and scalable.

[5]

3.5 Keycloak

Keycloak is an open-source Identity and Access Management (IAM) solution developed by Red Hat. It provides comprehensive features for user management, authentication, authorization, and securing applications.

Keycloak helps centralize and simplify identity management within an IT infrastructure. It offers features such as user registration, multi-factor authentication, role and permission management, session management, and integration with common authentication and authorization protocols like OAuth 2.0 and OpenID Connect.

Keycloak provides functionality for managing roles, administrators, users, and passwords. Here's how Keycloak addresses these aspects:

[6]

1. Keycloak allows defining roles at the realm or application level. Roles can be created and assigned to users to define their permissions and access.

2. Keycloak administrators can create, manage, and assign roles to users through the administration interface or management API.

3. Roles can be used to control access to features, pages, and resources within the application.

4. Keycloak provides specific administration roles like "admin" or "superadmin" that allow users to perform administrative tasks such as managing clients, users, roles, etc.

5. Users can log in using their credentials (username and password) or other supported authentication methods like two-factor authentication, OAuth 2.0, etc.

6. Keycloak supports user-based authentication and provides a registration interface to allow users to create their accounts.

7. Keycloak also offers advanced authentication features such as two-factor authentication, social authentication (via identity providers like Google, Facebook, etc.), and certificate-based authentication.

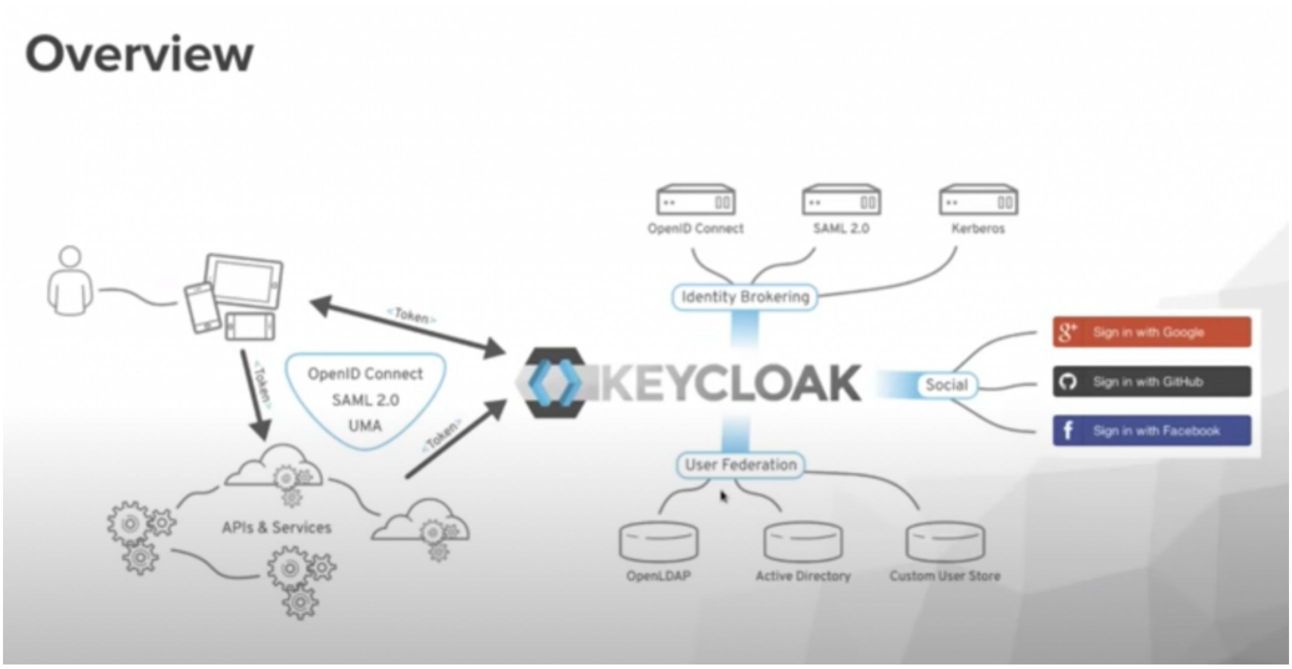

Figure 3.3: Keycloak Overview

3.6 Kafka

Kafka is a distributed and scalable data streaming platform designed to efficiently handle the transmission and processing of real-time data streams. It was developed by Apache Software Foundation.

Kafka is based on a distributed log architecture, where data is stored as streams of messages in "topics". Data producers send messages to specific "topics," while consumers subscribe to those "topics" to retrieve the messages. This allows for asynchronous communication and clear separation between data producers and consumers.

[1]

The key features of Kafka include:

1. Scalability: Kafka is designed to handle large volumes of data and can be horizontally scaled

to meet growing performance needs. It can handle high workloads and process thousands of messages per second.

2. Fault tolerance: Kafka ensures high availability and fault tolerance by replicating data across multiple nodes in the cluster. This ensures data reliability and availability even in case of node failures.

3. Data durability: Messages stored in Kafka are persistent and can be retained for a defined period. This allows for message replay and data recovery when needed, which is crucial for use cases requiring long-term data retention.

4. Stream processing: Kafka is designed for real-time stream processing. It enables applications to consume continuous data streams and process them in real-time, which is critical for use cases requiring real-time analysis, data pipelines, etc.

5. Integration with other tools: Kafka easily integrates with other tools and frameworks such as Spark, Hadoop, Flink, etc. This enables seamless integration with the Big Data ecosystem and facilitates data ingestion, processing, and streaming.

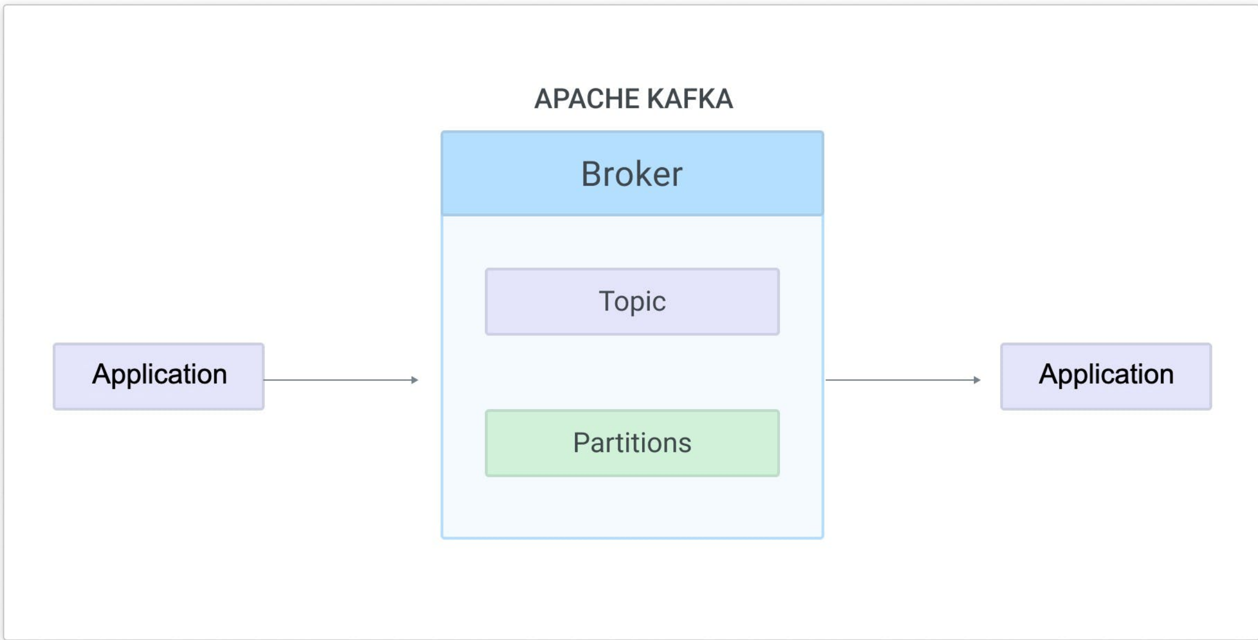

Figure 3.4: Kafka Architecture

Conclusion

these technologies, developers and data professionals can leverage their strengths to build powerful and scalable applications. Trino enables fast and flexible data querying, Spring Boot simplifies application development, Keycloak provides robust IAM capabilities, and Kafka enables real-time data streaming and processing.

Chapter 4

Solution Implementation

Introduction

This chapter focuses on the implementation of the solution, with an emphasis on choosing the connector to manipulate Delta Lake. We will discuss the feasibility study of different connectors and their impact on modularity, performance, and security.

4.1 Feasibility Study

During our feasibility study, we examined several connector options to manipulate Delta Lake. Here is an overview of the connectors we explored:

1. **Delta Standalone:** We attempted to use Delta Standalone, which is a standalone connector for Spark. However, we found that Delta Standalone only supports timestamp snapshots and does not allow queries. Additionally, it can sometimes be heavy when retrieving multiple records, which affects performance.

2. **Delta Sharing Server:** Delta Sharing Server is another option that requires its own server. While it is popular with Rust and Python, we found that using Delta Sharing Server with Java does not provide all the available options and lacks detailed documentation.

3. **Spark:** Spark is a feasible option and already available for manipulating Delta Lake. However, it requires a specific version of Spring Boot (2.7-) with very specific Maven configuration. Although Spark offers many features and is well integrated with the Big Data ecosystem, its use can be complex and requires precise configuration.

4. **Trino:** Trino is another connector we explored. However, it requires its own server and a relational database (Hive). Trino offers several advantages, including the ability to use standard SQL and a Java JDBC connector to access Delta Lake. This provides increased flexibility and ease of use.

4.2 Benchmark between Trino and Spark

Trino is a distributed and fast query engine designed to execute interactive SQL queries on large datasets. Spark, on the other hand, is a popular distributed processing system that also supports querying large-scale data. For this benchmark study, we will evaluate the performance of Trino and Spark on querying data stored in Delta Lake (Amazon S3 in our case), a data storage format optimized for analytical analytics.

4.2.1 Architecture

The following diagrams represent the respective architectures of the two Proof of Concept (POC) projects, SPARK and TRINO.

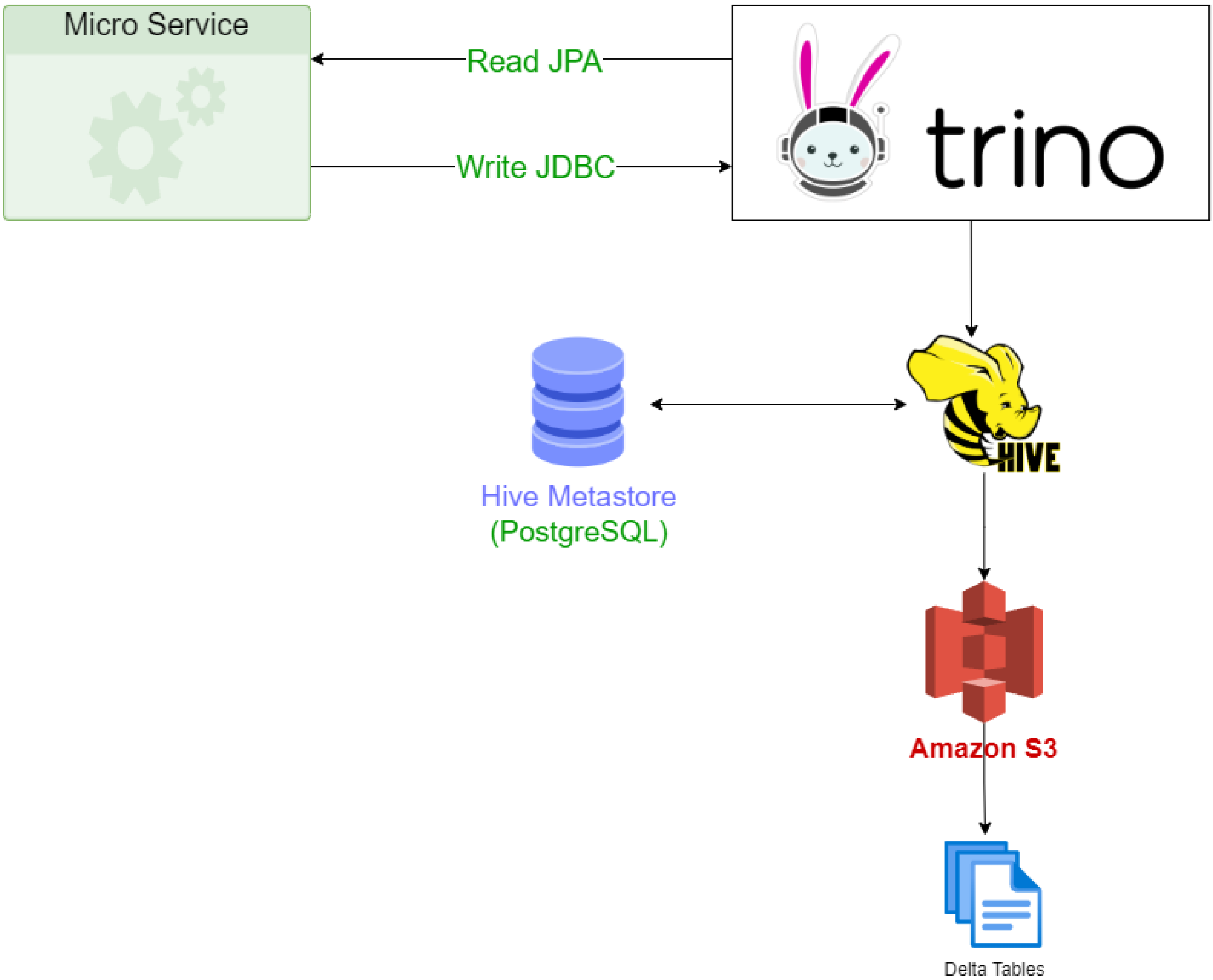

Figure 4.1: TRINO Architecture

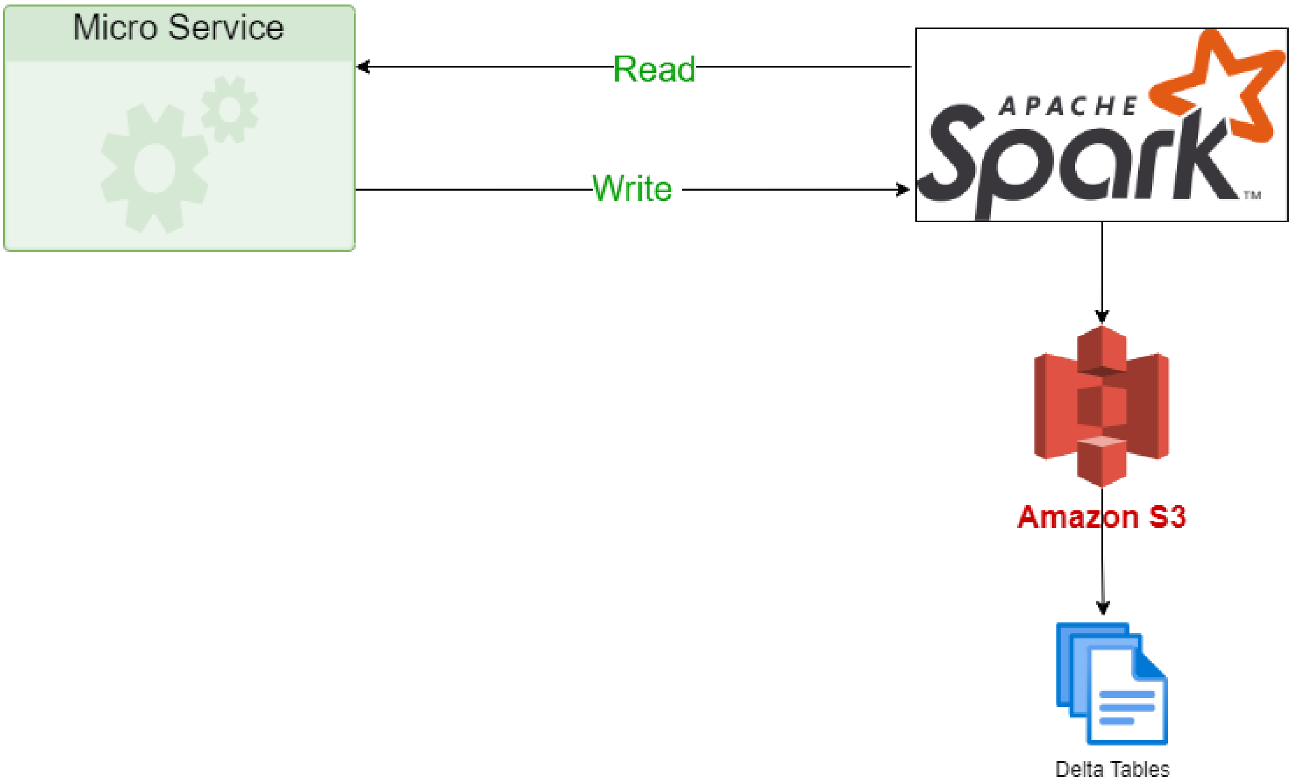

Figure 4.2: SPARK Architecture

4.2.2 Execution Time

- **Trino:** Trino is known for its high execution speed. It uses an optimized architecture to perform SQL queries very quickly by leveraging query parallelization and optimization. Trino can provide short execution times for queries on Delta Lake.

- **Spark:** Spark is also designed to deliver high performance, but it may require more time to read data from disk. However, Spark can offer high performance when used with complex transformation operations, thanks to its disk-based distributed computing model.

We performed a performance test on a 2-gigabyte Delta table using 2 computers:

1. The first PC, 'DELL Latitude 5530' is where we will launch the following Docker images:

    - **Trino:** This is the Trino server image for distributed data querying using SQL.
    - **Hive:** It will be used in conjunction with Trino to query and analyze data stored in Hive, thanks to the compatibility between the two systems and the metadata sharing via the Hive Metastore.
    - **PostgreSQL:** It will be used to store metadata of Delta tables, schemas, and partitions, facilitating management, discovery, and access to data in a distributed environment. It also offers query optimization features and permission management to improve data analysis performance and security.
    - **AWS:** Where the Delta tables are located.

2. The second PC, 'ASUS ROG G752 VT' is where the Spring Boot microservices for SPARK and TRINO will be executed.

The configuration used in the tests is as follows:

Component	**Specifications**
Processor	12th Gen Intel® Core™ i7-1255U 1.70 - 4.8 GHz
Processor Frequency	2.6 - 3.2 GHz
Memory	24 GB
Memory Type	DDR4
Graphics Chip	Intel Iris Xe Graphics
Hard Drive	512GB PCIe NVMe Class 35 SSD
Operating System Type	Windows 11 64-bit

Table 4.1: Technical Specifications of "DELL Latitude 5530" PC

Component	**Specifications**
Processor	Intel Core i7-6700HQ
Processor Frequency	2.6 - 3.2 GHz
Memory	24 GB DDR4
Hard Drive	1TB SSD + 1TB HDD
Memory Type hline Graphics Chip	DDR4-SDRAM Nvidia GeForce GTX 970M
Graphics Memory Quantity	3072 MB dedicated
Operating System Type	Windows 10 64-bit

Table 4.2: Technical Specifications of 'ASUS ROG G752 VT' PC

The results are as follows:

Queries	**Implementation**	
	Spark (s)	**Trino (s)**
SELECT (per page and per page result limit - 100 per page)	1.2	0.43
Aggregation and grouping	2.1	0.8
Sorting	2.2	0.8

Table 4.3: Technical Specifications of 'ASUS ROG G752 VT' PC

4.2.3 Memory Management

- **Trino:** Trino primarily utilizes memory to accelerate query processing. It has an in-memory query engine that optimizes data access and minimizes access times. However, this means that Trino may require a significant amount of memory to process large datasets.

- **Spark:** Spark uses a disk-based distributed computing model, which means data is typically stored on disk and loaded into memory as needed. This allows Spark to handle much larger datasets than what can fit in memory.

4.2.4 Ecosystem and Integrations

- **Trino:** Trino has a growing ecosystem with a wide range of connectors and integrations, allowing access to different data sources, including Delta Lake. It is compatible with various data visualization and processing tools.

- **Spark:** Spark is a mature project with a rich ecosystem of tools, libraries, and connectors. It benefits from a large developer community and numerous learning resources. Spark also supports both stream processing and batch processing of data.

4.2.5 Scalability

- **Trino:** Trino is designed to be highly scalable and can handle large datasets. It can be easily configured to accommodate intensive workloads and significant data volumes.

- **Spark:** Spark is also designed for scalability and can process massive datasets. It has a distributed processing system that can adapt to different cluster configurations and resources.

4.2.6 Support for Delta Lake Features

- **Trino:** Trino has an official connector for Delta Lake, allowing it to read and write data in Delta Lake. It supports essential Delta Lake features such as ACID transaction management, incremental updates, and merge operations.

- **Spark:** Spark also has built-in support for Delta Lake. It offers advanced Delta data management features, including support for ACID transactions, efficient merge operations, and data versioning mechanisms.

4.2.7 Ease of Use

- **Trino:** Trino provides a standard SQL interface, making it easy to use for users familiar with SQL. It also offers user-friendly interactive querying tools and flexible configuration options.

- **Spark:** Spark offers an SQL interface through Spark SQL, but it is also oriented towards data processing through a broader API. Spark may be more complex to use for users less familiar with distributed processing concepts.

4.2.8 Other Statistics

	Spark	**Trino**
Developer Experience	⭐⭐ Spark's syntax requires SMEs to have experience in data engineering	⭐⭐⭐ Standard SQL interface, making it easier to use for users familiar with SQL.

Coût	Team: 🧑‍💻🧑‍💻 Infrastructure: $	Team: 🧑‍💻🧑‍💻 Infrastructure: $
Fiabilité	⭐⭐⭐ Mature product maintained internally and supported by Google Dataproc	⭐⭐ Has experienced a few outages as the internal team learns to maintain and scale
Caractéristiques et Open sources	⭐⭐⭐ Has been supported by the open-source community since 2014.	⭐⭐⭐ Has been supported by the open-source community since 2019. Shopify has made contributions
Usage	~Approximately 800,000 jobs/day	3000 active users/week ~Approximately 200,000 queries/week

4.3 Solution Design

For the solution design, we have chosen an architecture based on microservices, using the following technologies: Spring Data, JPA, Hibernate, JDBC, and Trino.

The microservices are developed using Spring Data, which provides a high-level abstraction for interacting with the database. This allows the use of annotations and interfaces to define entities, queries, and persistence operations. JPA (Java Persistence API) is used as the interface layer for interacting with the database, while Hibernate is used as the persistence provider, enabling the management of Java objects and their mapping to Trino.

Communication between the microservices is facilitated by a Kafka broker. Kafka is a distributed streaming platform that allows microservices to exchange messages in real-time. Microservices publish events to Kafka topics, and other microservices can subscribe to these topics to receive the corresponding events.

For authentication and authorization management, we use Spring Cloud Gateway. This component receives service calls from web, mobile, or desktop applications. These applications send a JSON Web Token (JWT) to Spring Cloud Gateway for authentication. Spring Cloud Gateway verifies the availability of the token by contacting Keycloak, which is responsible for identity and access management. Once the token is validated, Spring Cloud Gateway allows the request to pass through to the microservices.

The microservices communicate with Trino using the JDBC connector. Trino is responsible for querying and processing data from a large number of data sources. It stores metadata in Hive, which is a data warehouse based on Hadoop. Trino also uses the concept of Delta tables to manage incremental data changes. These Delta tables are stored on AWS (Amazon Web Services), providing a scalable and reliable storage solution.

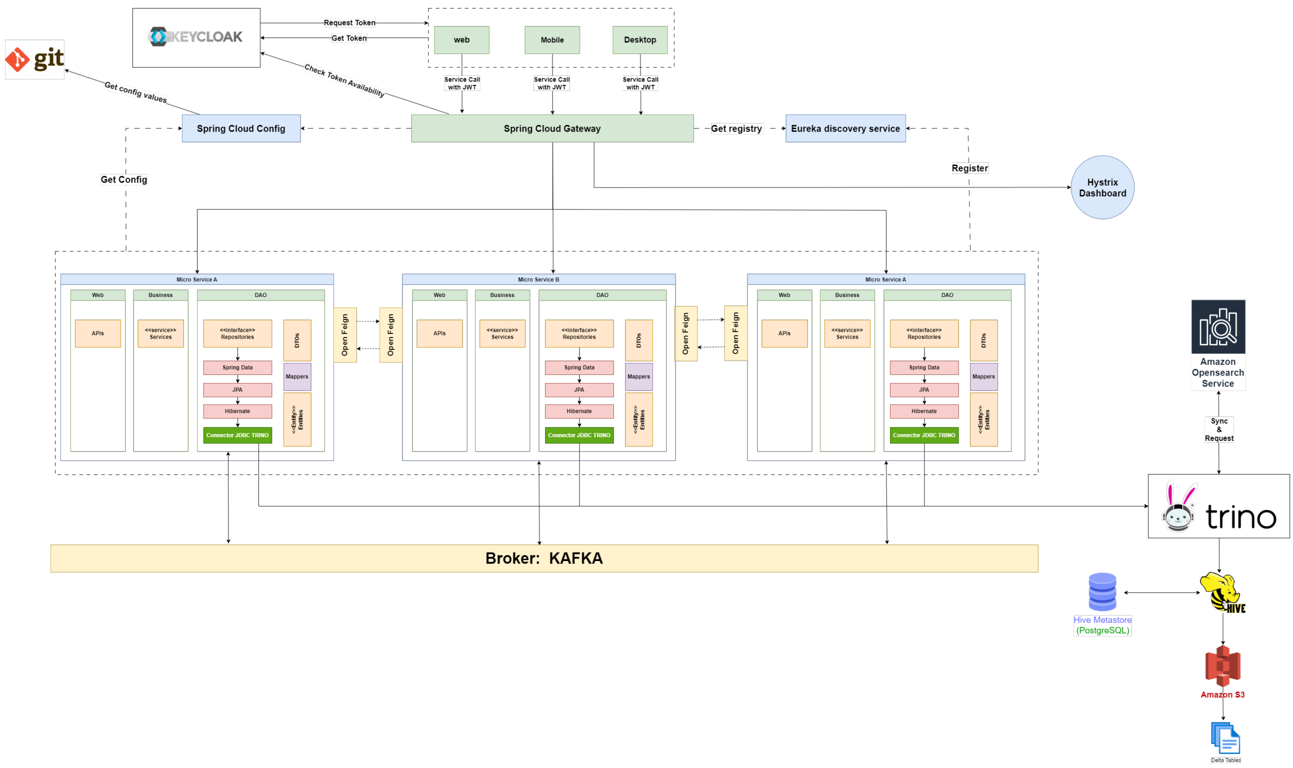

Figure 4.3: Solution Architecture

4.4 Advantages

The chosen architecture offers several significant advantages:

- **Scalability:** The microservices-based architecture allows for horizontal scalability, meaning each microservice can be deployed and scaled independently based on its specific needs. This enables fine-grained capacity and resource adjustments, making it easier to handle high workloads and adapt to growing application requirements.
- **Modularity:** The modular design of microservices enables more efficient development, deployment, and maintenance. Each microservice can be developed independently, facilitating collaboration between teams and promoting code reuse. Additionally, modifications or updates to one microservice do not impact others, reducing the risks of errors and conflicts.
- **Technological Flexibility:** Microservices provide flexibility in terms of technological choices. Each microservice can be developed using the most suitable technologies for its specific functionality. This allows for leveraging the advantages of each technology and selecting the best tools for each component of the architecture.
- **Effective Communication:** Using Kafka as a streaming broker facilitates communication between microservices. Kafka ensures reliability and scalability of real-time event exchanges, enabling smooth communication and efficient synchronization between different system components.

- **Enhanced Security:** Integration of Keycloak for authentication and authorization provides a high level of security. JWT tokens are used for user identification, and Spring Cloud Gateway handles token verification and management. This ensures that only authorized individuals have access to appropriate resources and functionalities, strengthening application security.
- **Optimized Performance:** Utilizing Trino as a distributed query engine allows querying and processing large amounts of data in a parallel and efficient manner. Trino offers high performance and query optimization, enabling faster results and improved user experience.

These advantages contribute to a scalable, modular, flexible, secure, and high-performing architecture that meets the requirements of your project.

4.5 Visualization

In this section, we will present screenshots of the different microservices in our solution, including Swagger for the API, the Trino console for data exploration, and the Izicap website. These screenshots will illustrate the user interface and the functionalities offered by each service.

4.5.1 Trino Console

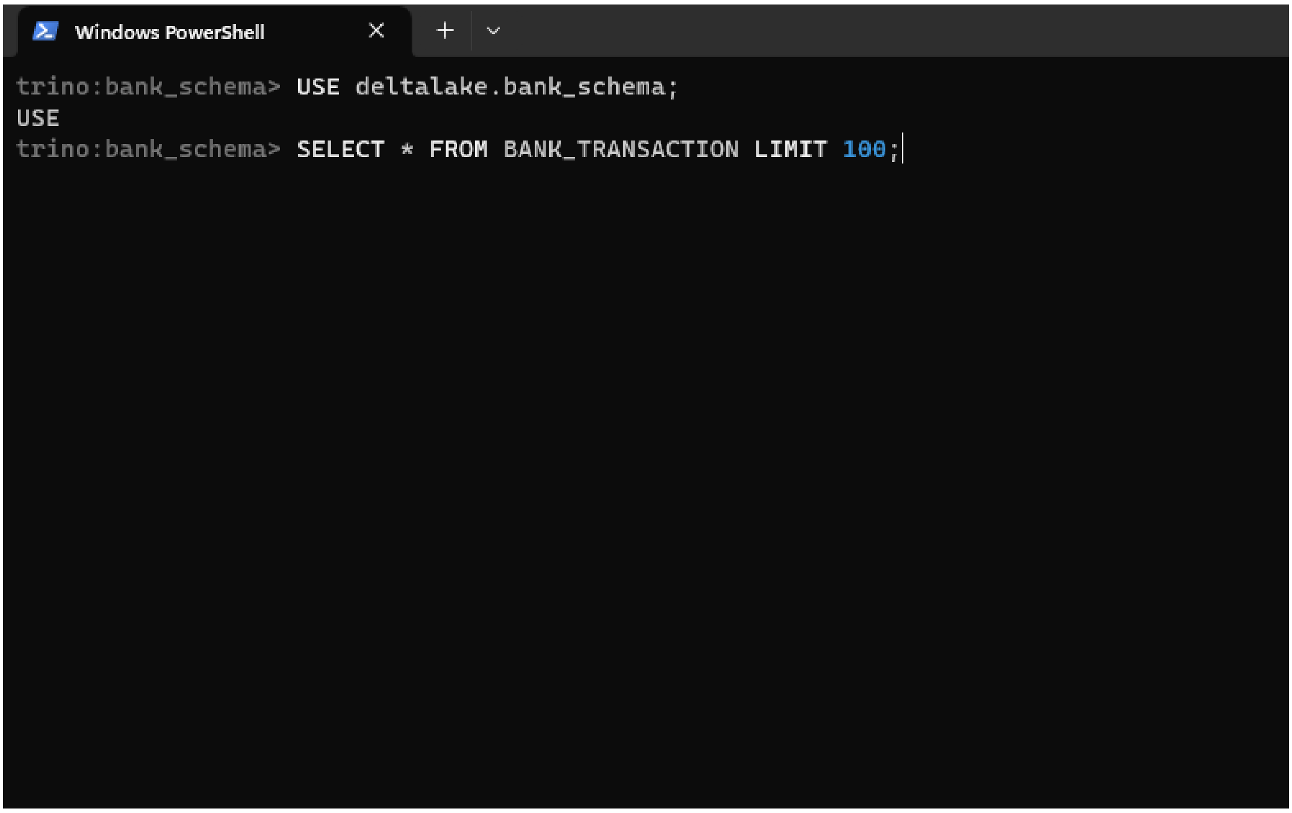

Figure 4.4: Trino Console

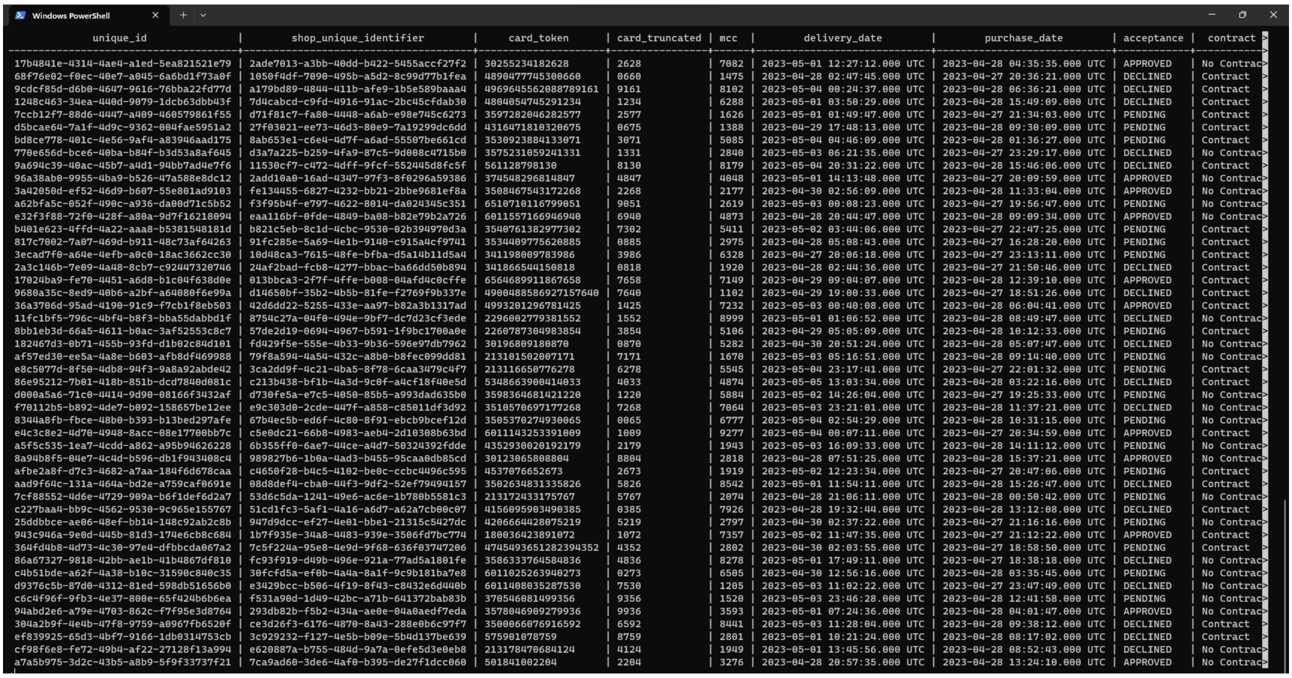

Figure 4.5: Trino SELECT Query

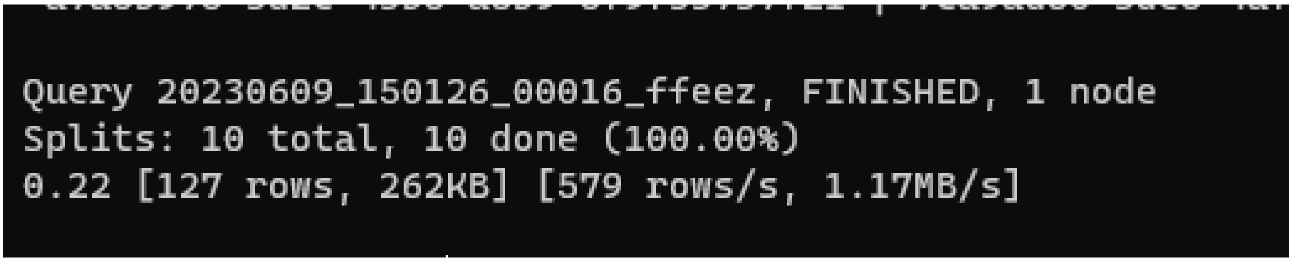

Figure 4.6: Trino command prompt

These screenshots of the Trino console highlight the advanced data querying interface provided by Trino. Developers and analysts can execute SQL queries on the data stored in Trino, explore schemas, preview results, and perform analytical operations to extract valuable insights.

4.5.2 Microservices

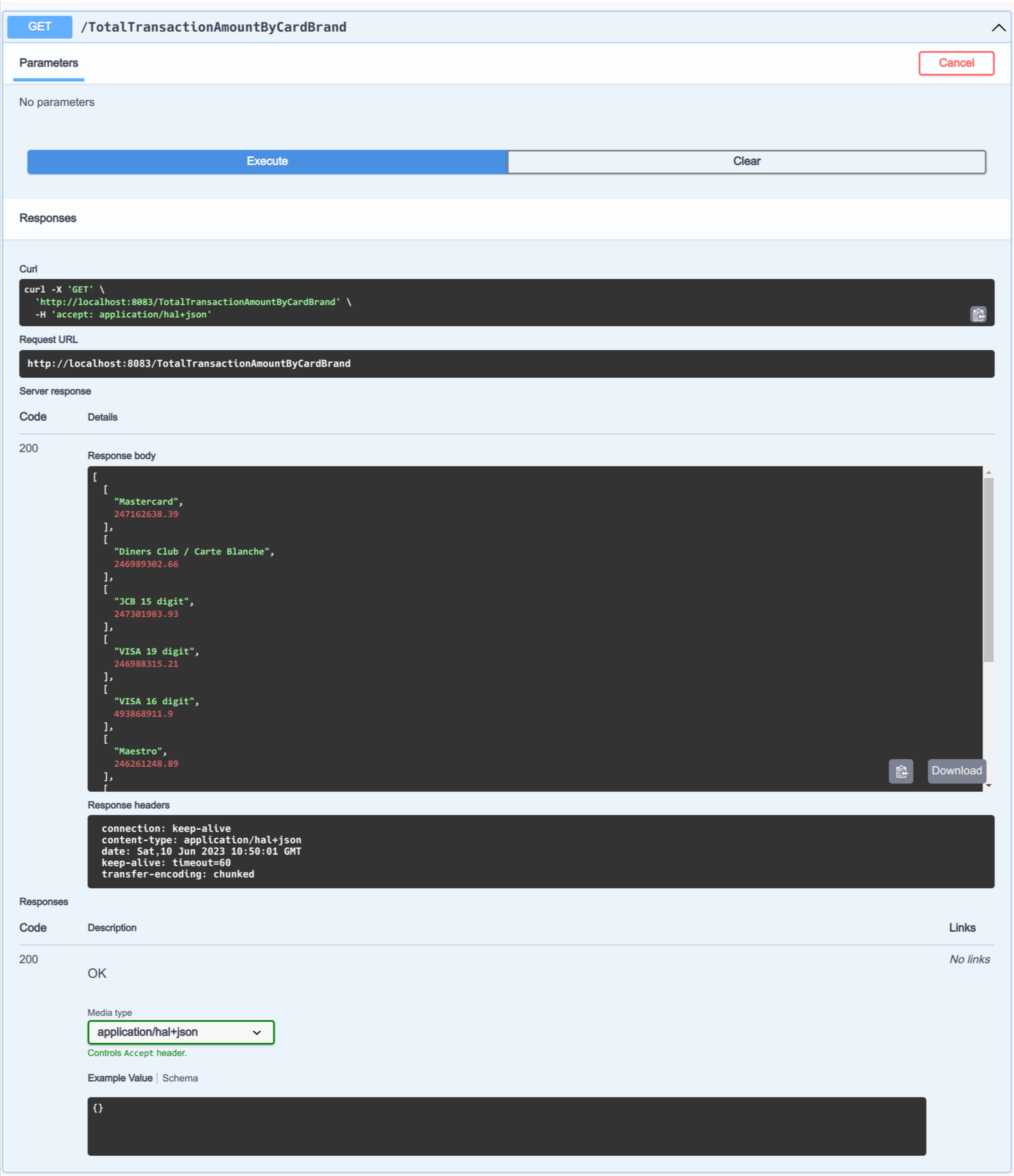

Figure 4.7: Swagger UI of Endpoint 1

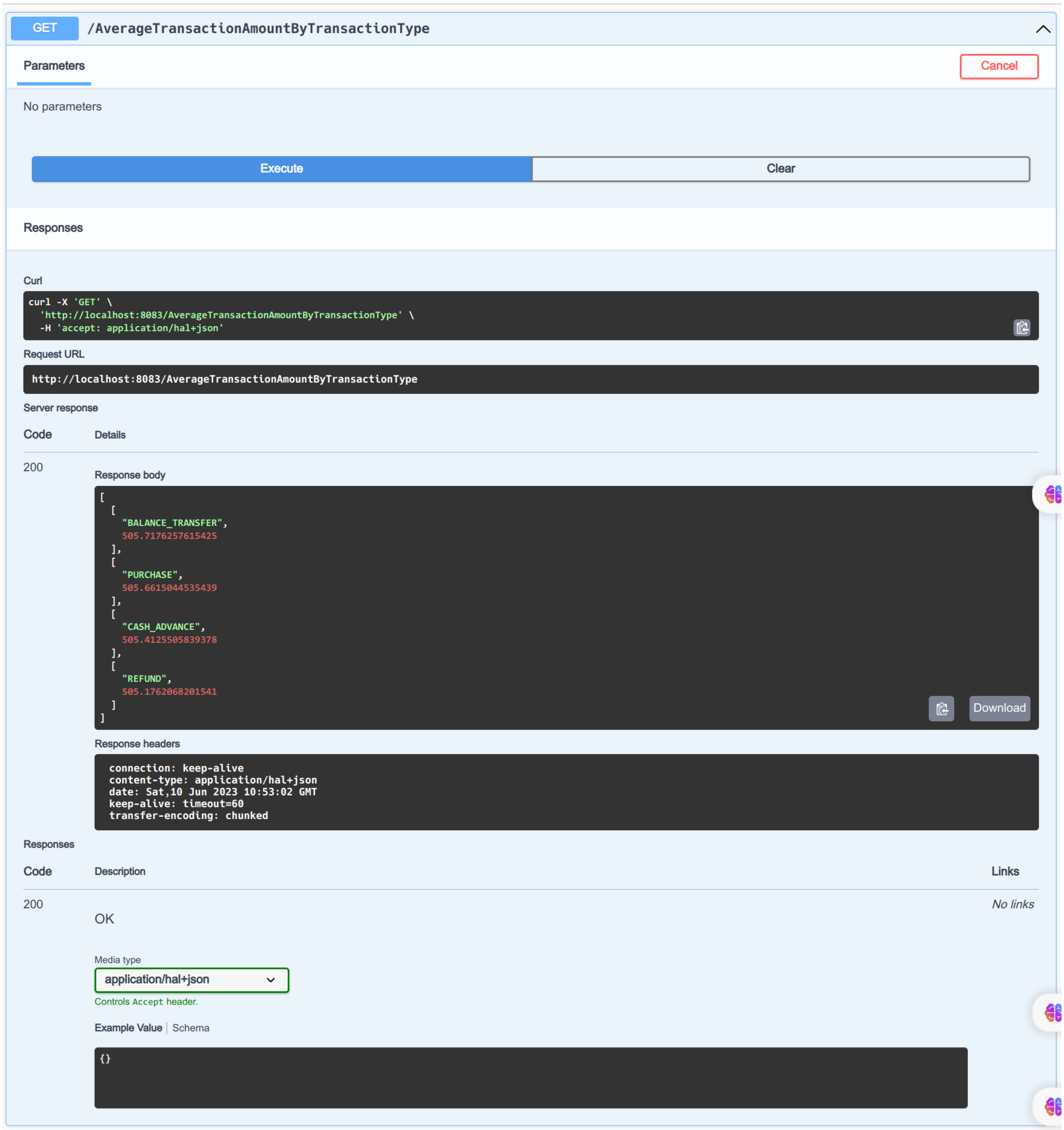

Figure 4.8: Swagger UI of Endpoint 2

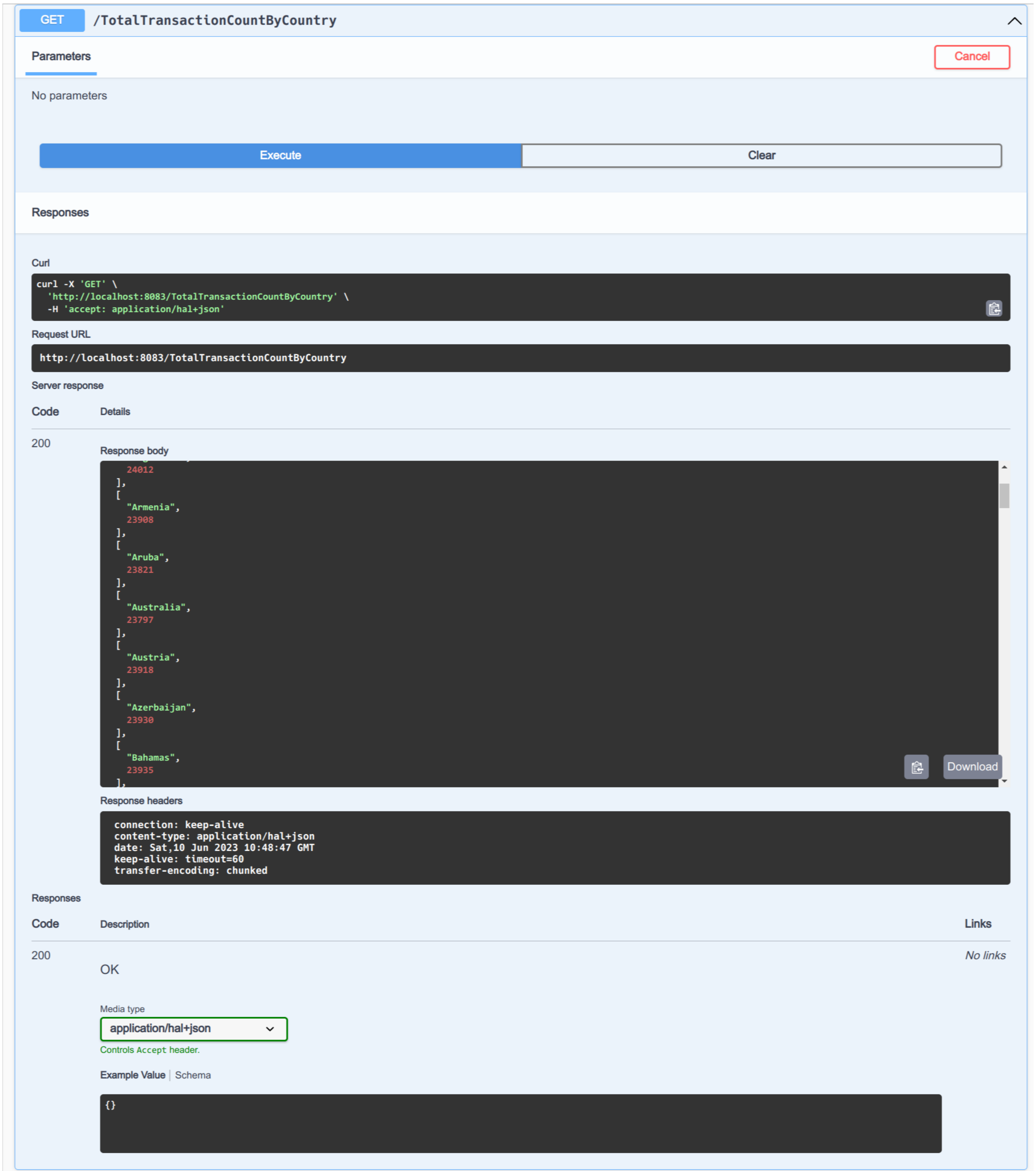

Figure 4.9: Swagger UI of Endpoint 3

Figure 4.10: Swagger UI

These images showcase the detailed documentation of our API using Swagger. This user-friendly interface allows developers to understand the available endpoints, required parameters, expected responses, and usage examples. Swagger facilitates integration and communication with our microservices.

4.5.3 Delta Lake

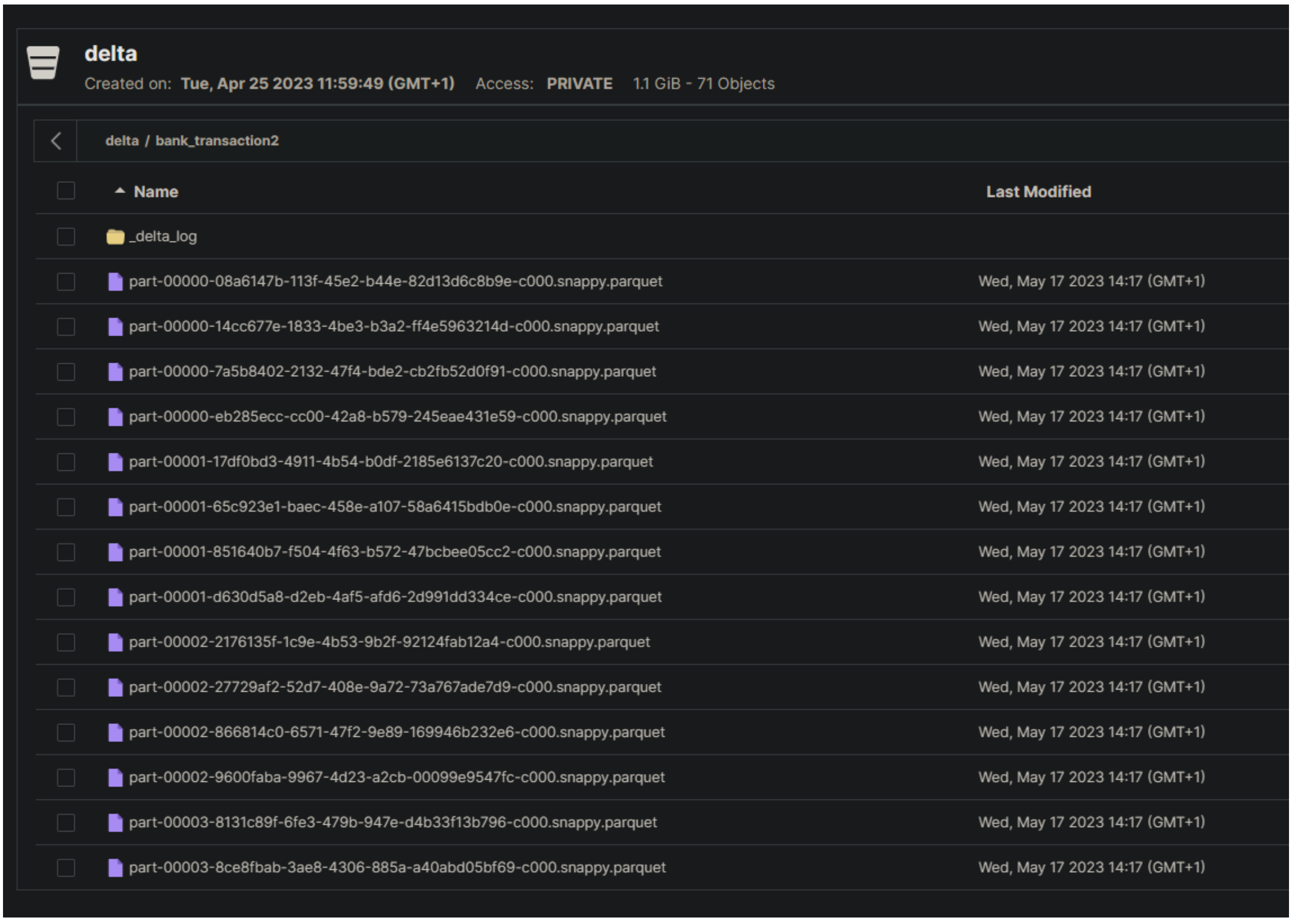

Figure 4.11: Parquet Data

We have data files in the Parquet format. Parquet files are optimized for efficient storage and retrieval of data. They are compressed and structured in a way that allows for fast queries and selective reading of data. These Parquet files contain the actual table data, organized into columns and partitions, making it easy to manipulate and further analyze the data.

4.5.4 Izicap UI

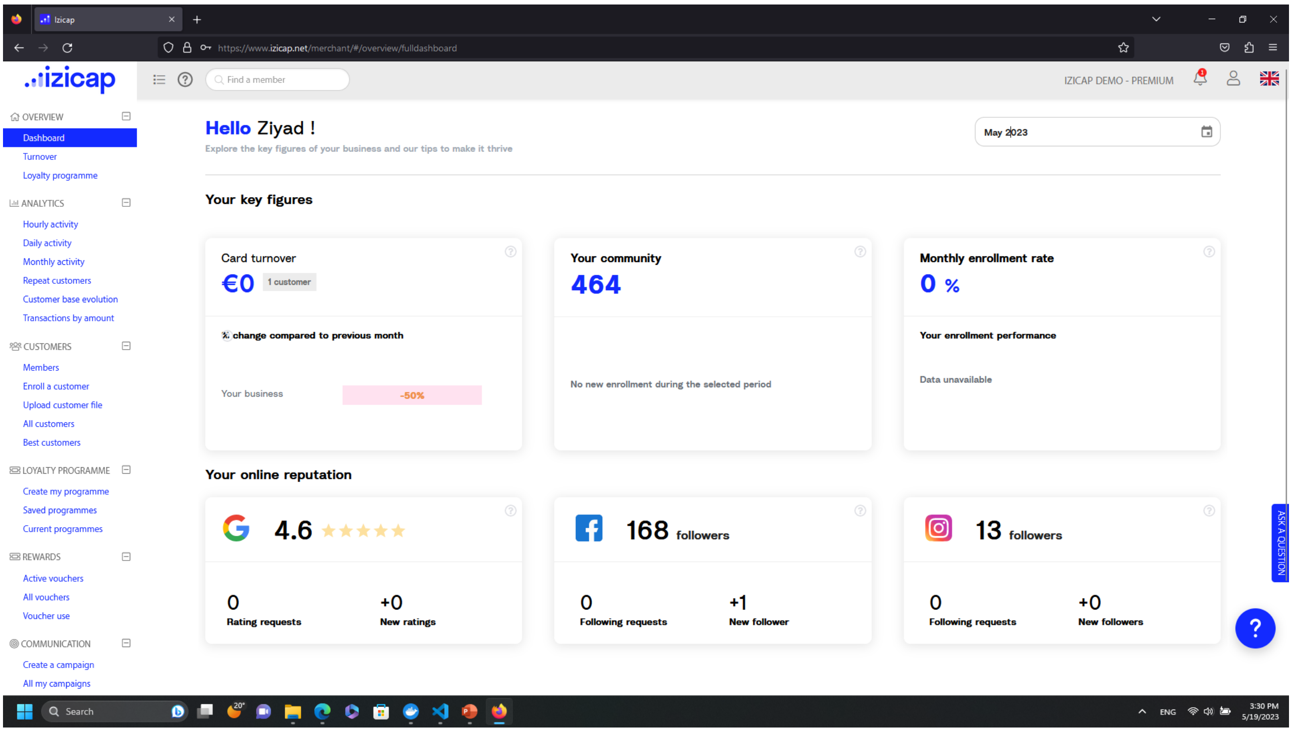

Figure 4.12: Izicap Portal

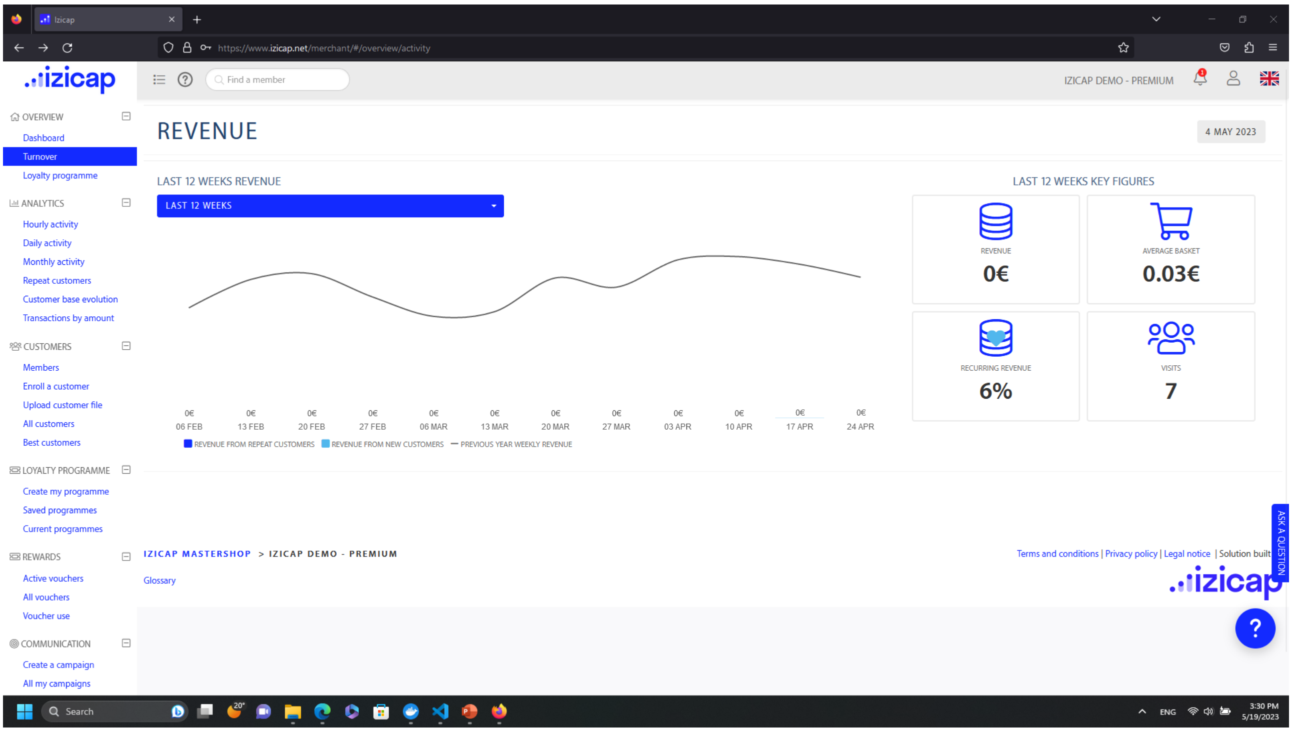

Figure 4.13: Customer Revenue

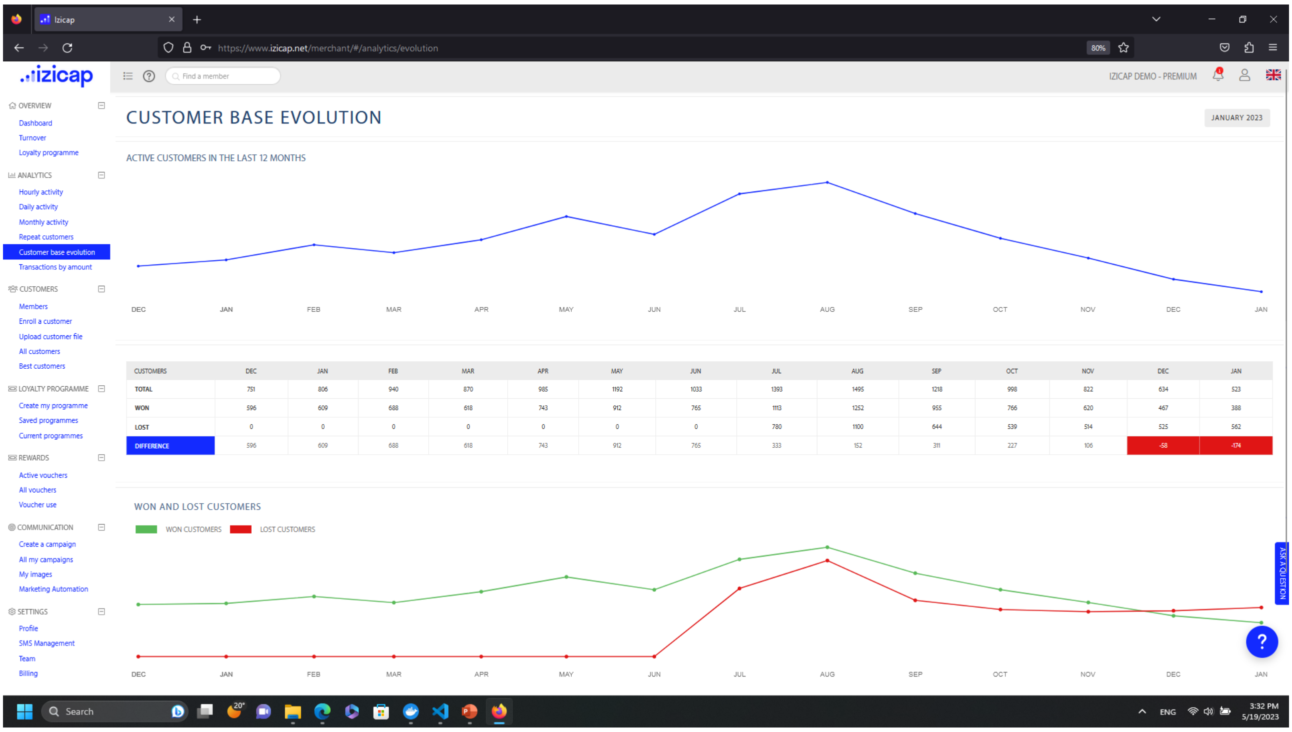

Figure 4.14: Customer Base Evolution

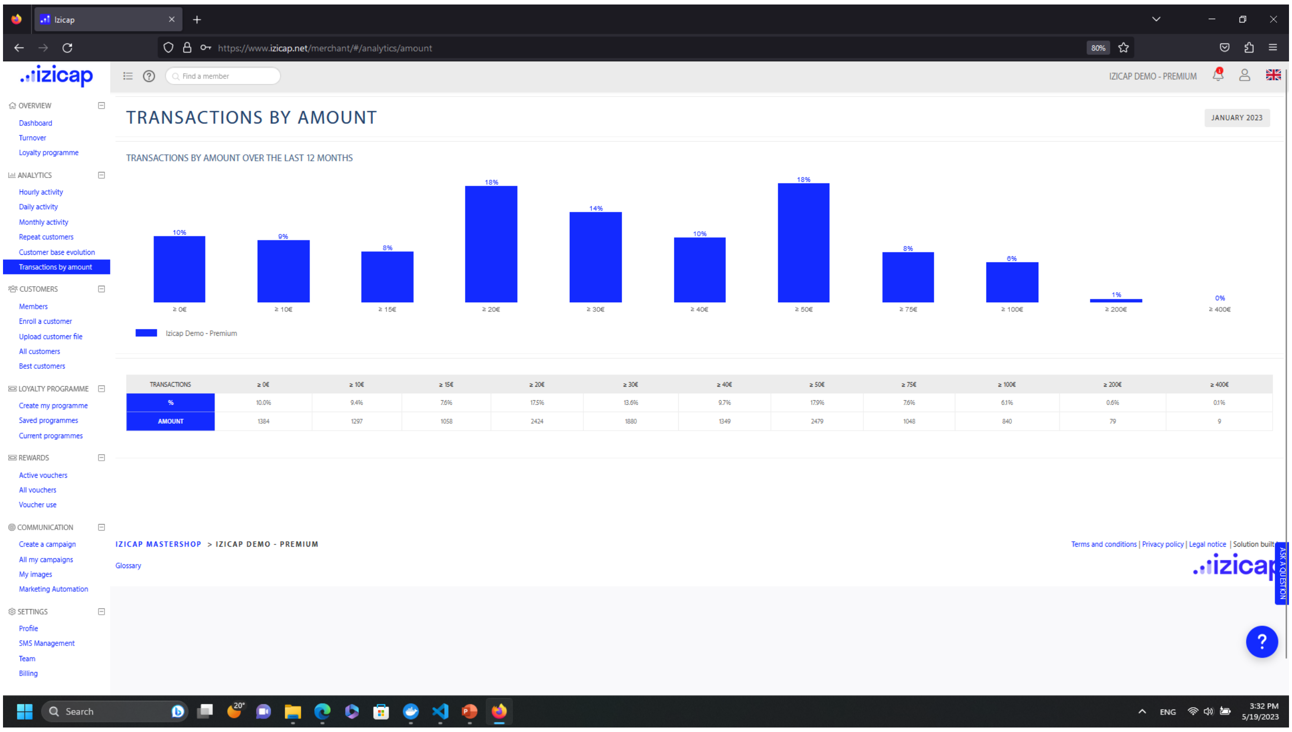

Figure 4.15: Transactions by Amount

The use of micro-frontends allows for splitting the user interface into several independent modules, each being responsible for a specific functionality. This enables clear separation of responsibilities and facilitates the development, maintenance, and evolution of the application.

Conclusion

The feasibility study of connectors for manipulating Delta Lake has allowed us to make informed decisions regarding the implementation of the solution. While Spark is widely used and integrated into the Big Data ecosystem, Trino offers significant advantages in terms of flexibility and performance for SQL queries. Based on our specific needs and technical constraints, we have chosen to use Trino as the connector for manipulating Delta Lake in our solution.

General Conclusion

The internship has been a rewarding experience that allowed exploring various aspects of microservices using technologies such as Delta Lake, Trino, Spring Boot, and Keycloak. The work environment was conducive to learning and implementing these concepts.

The adoption of microservices brings numerous advantages over a monolithic architecture. Microservices offer better scalability and flexibility, allowing independent deployment, development, and scaling of each service. Moreover, communication between microservices through APIs facilitates integration and collaboration among different parts of the system.

Throughout the internship, we learned how to design and implement microservices using Spring Boot, leveraging its features for persistence, security, and REST API creation. We also integrated Keycloak to manage user authentication and authorization in our microservices architecture. The use of Delta Lake ensured data reliability and facilitated handling updates.

By successfully transforming the monolithic architecture into a microservices-based architecture, the application achieved improved flexibility, maintainability, performance, data consistency, and modularity. The adoption of React and the microfrontend approach enhanced the frontend development experience and allowed for more modular and independent functionalities.

The internship provided valuable insights into modern application architecture and development practices. It emphasized the importance of considering scalability, flexibility, and modularity when designing and implementing software solutions. The experience gained will undoubtedly contribute to future projects and endeavors in the field of software development.

Bibliography

[1] Apache. 2014.

[2] Databrick. 2023.

[3] M. Kukreja. *Data Engineering with Apache Spark, Delta Lake, and Lakehouse*. Packt Publishing, 2021.

[4] M. T. Matt Fuller, Manfred Moser. *Trino: The Definitive Guide*. O'Reilly Media, Inc., 2021.

[5] K. S. P. Reddy. *Beginning Spring Boot 2: Applications and Microservices with the Spring Framework*. Apress, 2021.

[6] S. Thorgersen. 2014.