



MINISTRY 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	x
List of Tables	xi
Introduction	1
1 General project context	2
2 Business Intelligence and Solution Choices	11
3 Technologies Used	20
4 Solution Implementation	27
Conclusion	44

# 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 متعمقاً 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 ، وبناء 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

<b>Dedication</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Résumé</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>Introduction</b>	<b>1</b>
<b>1 General project context</b>	<b>2</b>
Introduction . . . . .	2
1.1 Difference between a data warehouse, a data lake, a datalakehouse, and Delta Lake . . . . .	2
1.1.1 Data Warehouse . . . . .	3
1.1.2 Data Lake . . . . .	3
1.1.3 Datalakehouse . . . . .	3
1.1.4 Delta Lake (Datalakehouse) . . . . .	4
1.2 Difference between Microservices Architecture and Monolithic Architecture . . . . .	4
1.2.1 Advantages and Disadvantages of Monolithic Architecture . . . . .	5
1.2.2 Advantages and Disadvantages of Microservices Architecture . . . . .	6
1.3 Difference between Microfrontends Architecture and Monolithic Frontend Architecture . . . . .	7
1.3.1 Advantages and Disadvantages of Monolithic Frontend Architecture . . . . .	7
1.3.2 Advantages and Disadvantages of Microfrontends Architecture . . . . .	8
1.4 Solution Approach . . . . .	9
1.4.1 Data Part . . . . .	9
1.4.2 Backend Part . . . . .	9
1.4.3 Frontend Part . . . . .	9
<b>2 Business Intelligence and Solution Choices</b>	<b>11</b>
Introduction . . . . .	11
2.1 Difference between a data warehouse, a data lake, a datalakehouse, and Delta Lake . . . . .	11

2.1.1	Data Warehouse	12
2.1.2	Data Lake	12
2.1.3	Datalakehouse	12
2.1.4	Delta Lake (Datalakehouse)	13
2.2	Difference between Microservices Architecture and Monolithic Architecture	13
2.2.1	Advantages and Disadvantages of Monolithic Architecture	14
2.2.2	Advantages and Disadvantages of Microservices Architecture	15
2.3	Difference between Microfrontends Architecture and Monolithic Frontend Architecture	16
2.3.1	Advantages and Disadvantages of Monolithic Frontend Architecture	16
2.3.2	Advantages and Disadvantages of Microfrontends Architecture	17
2.4	Solution Approach	18
2.4.1	Data Part	18
2.4.2	Backend Part	18
2.4.3	Frontend Part	18
<b>3</b>	<b>Technologies Used</b>	<b>20</b>
3.1	Delta Lake	20
3.2	Key Benefits and Features of Delta Lake	21
3.3	Trino	23
3.4	Spring Boot	24
3.5	Keycloak	24
3.6	Kafka	25
<b>4</b>	<b>Solution Implementation</b>	<b>27</b>
	Introduction	27
4.1	Feasibility Study	27
4.2	Benchmark between Trino and Spark	28
4.2.1	Architecture	28
4.2.2	Execution Time	29
4.2.3	Memory Management	30
4.2.4	Ecosystem and Integrations	31
4.2.5	Scalability	31
4.2.6	Support for Delta Lake Features	31
4.2.7	Ease of Use	31
4.2.8	Other Statistics	31
4.3	Solution Design	32
4.4	Advantages	33
4.5	Visualization	34
4.5.1	Trino Console	34
4.5.2	Microservices	36
4.5.3	Delta Lake	40
4.5.4	Izicap UI	41

Conclusion	44
------------	----

# List of Figures

1.1	Data Warehouse vs Data Lake vs Data Lakehouse . . . . .	4
1.2	Monolithic Architecture vs Microservices Architecture . . . . .	7
1.3	Monolithic Frontend Architecture vs Microfrontends Architecture . . . . .	9
2.1	Data Warehouse vs Data Lake vs Data Lakehouse . . . . .	13
2.2	Monolithic Architecture vs Microservices Architecture . . . . .	16
2.3	Monolithic Frontend Architecture vs Microfrontends Architecture . . . . .	18
3.1	Delta Lake multi-hop architecture . . . . .	21
3.2	Overview of Trino architecture with coordinator and workers . . . . .	23
3.3	Kafka Architecture . . . . .	26
4.1	TRINO Architecture . . . . .	28
4.2	SPARK Architecture . . . . .	29
4.3	Solution Architecture . . . . .	33
4.4	Trino Console . . . . .	34
4.5	Trino SELECT Query . . . . .	35
4.6	Trino command prompt . . . . .	35
4.7	Swagger UI of Endpoint 1 . . . . .	36
4.8	Swagger UI of Endpoint 2 . . . . .	37
4.9	Swagger UI of Endpoint 3 . . . . .	38
4.10	Swagger UI . . . . .	39
4.11	Parquet Data . . . . .	40
4.12	Izicap Portal . . . . .	41
4.13	Customer Revenue . . . . .	41
4.14	Customer Base Evolution . . . . .	42
4.15	Transactions by Amount . . . . .	42

# List of Tables

1	List of Abbreviations . . . . .	xii
4.1	Technical Specifications of "DELL Latitude 5530" PC . . . . .	30
4.2	Technical Specifications of 'ASUS ROG G752 VT' PC . . . . .	30
4.3	Technical Specifications of 'ASUS ROG G752 VT' PC . . . . .	30

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.

# General project context

## Introduction

Business Intelligence (BI), also known as 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. Business Intelligence 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.

## 1.1 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.

### 1.1.1 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

### 1.1.2 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

### 1.1.3 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

**1.1.4 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

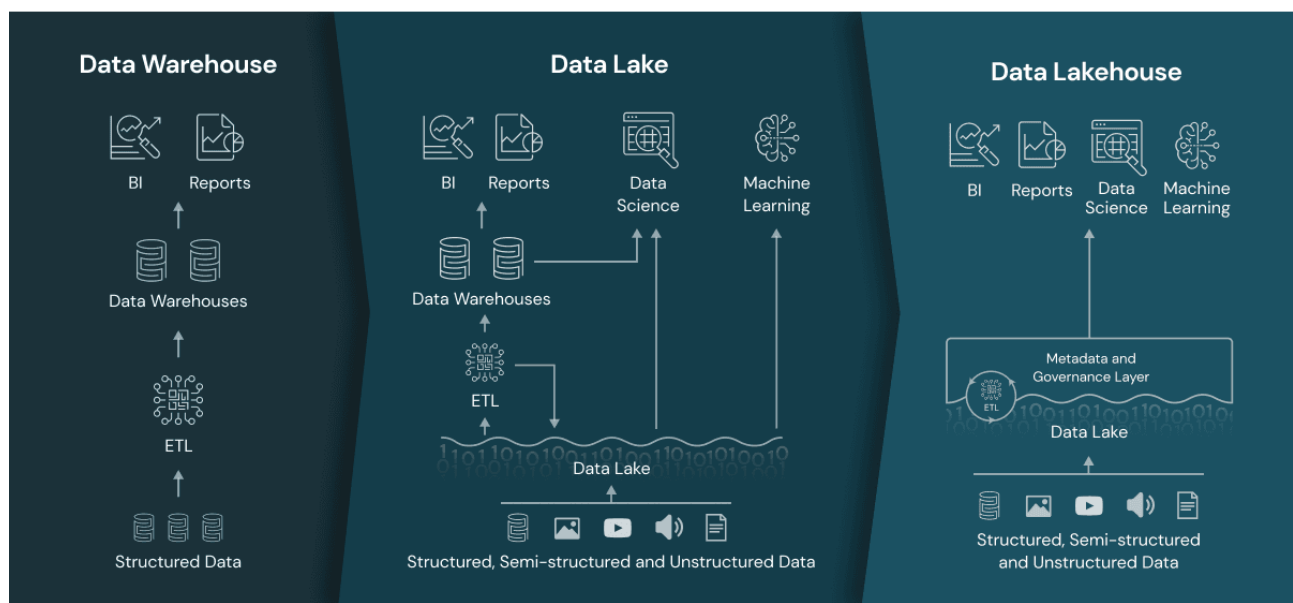


Figure 1.1: Data Warehouse vs Data Lake vs Data Lakehouse

## 1.2 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.

### 1.2.1 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.

## 1.2.2 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.

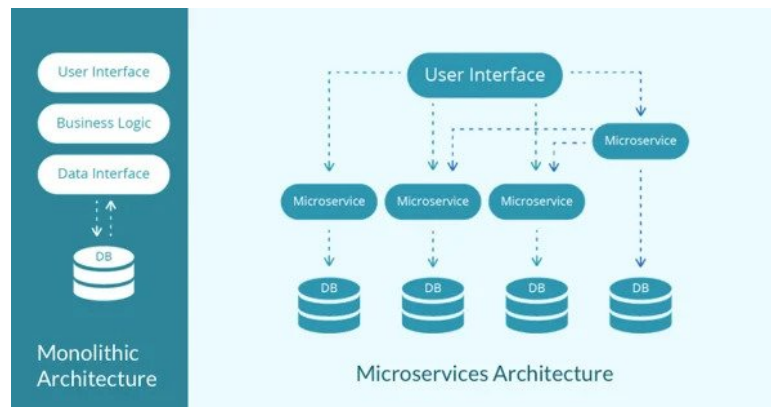


Figure 1.2: Monolithic Architecture vs Microservices Architecture

## 1.3 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.

### 1.3.1 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.

### **1.3.2 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.

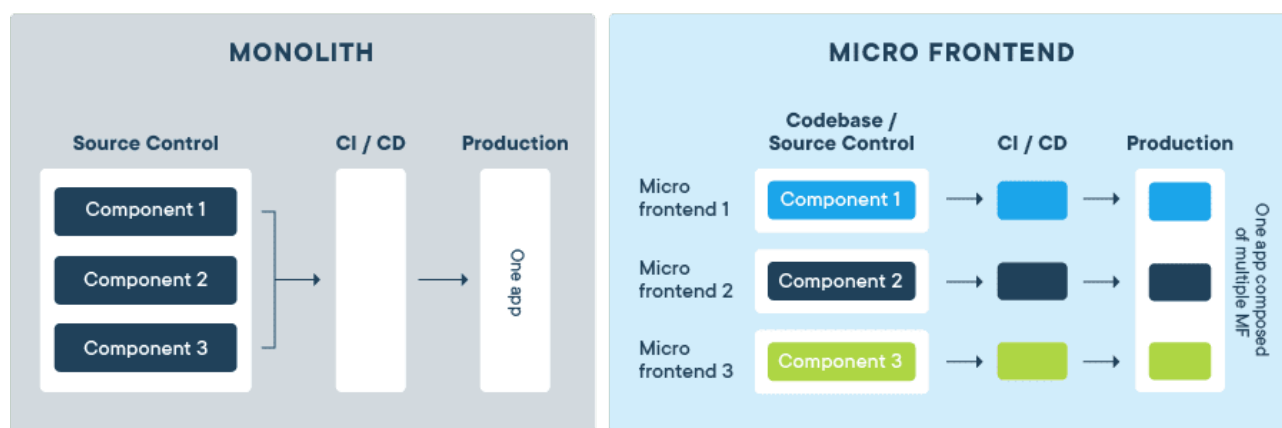


Figure 1.3: Monolithic Frontend Architecture vs Microfrontends Architecture

## 1.4 Solution Approach

### 1.4.1 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.

### 1.4.2 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.

### 1.4.3 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.



# Business Intelligence and Solution Choices

## Introduction

Business Intelligence (BI), also known as 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. Business Intelligence 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

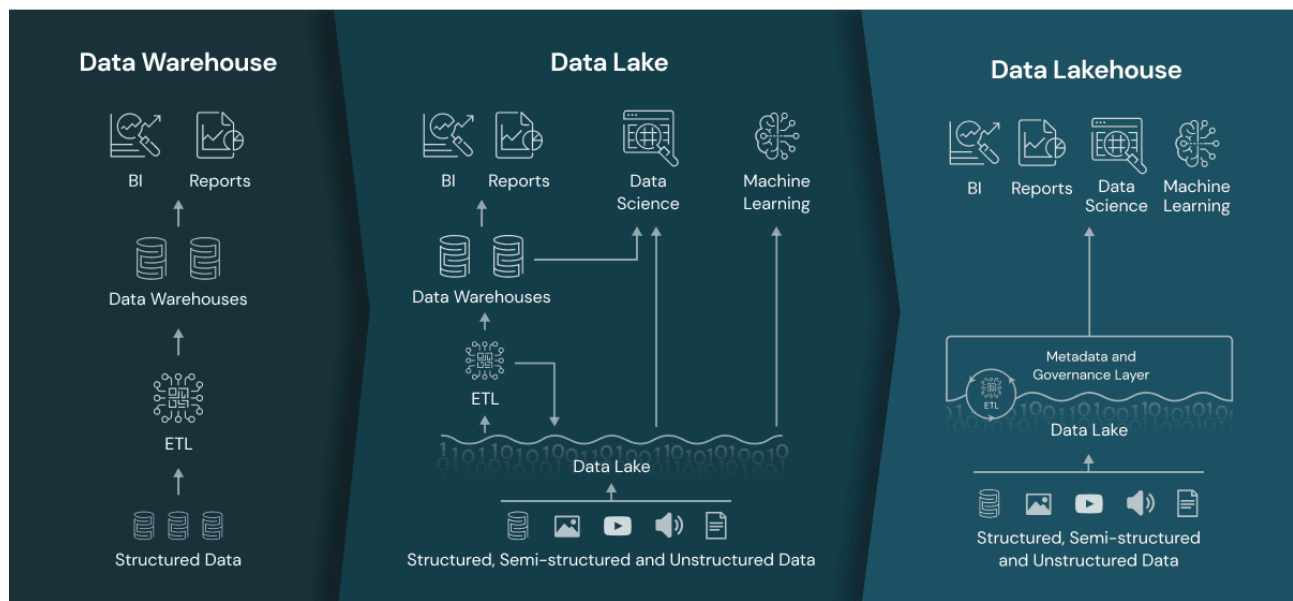


Figure 2.1: Data Warehouse vs Data Lake vs Data Lakehouse

## 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.

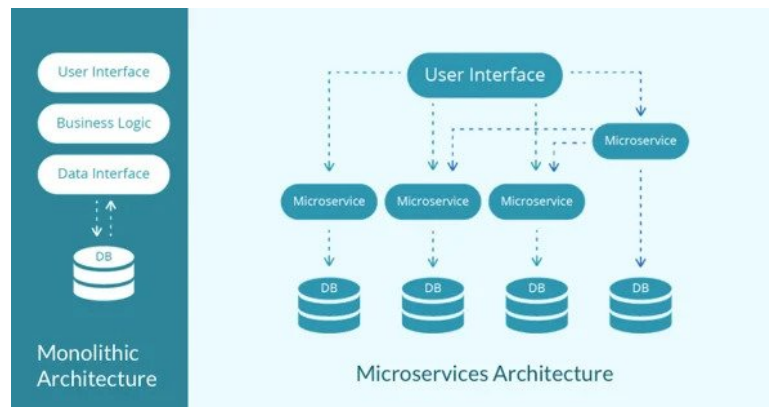


Figure 2.2: Monolithic Architecture vs Microservices Architecture

## 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.

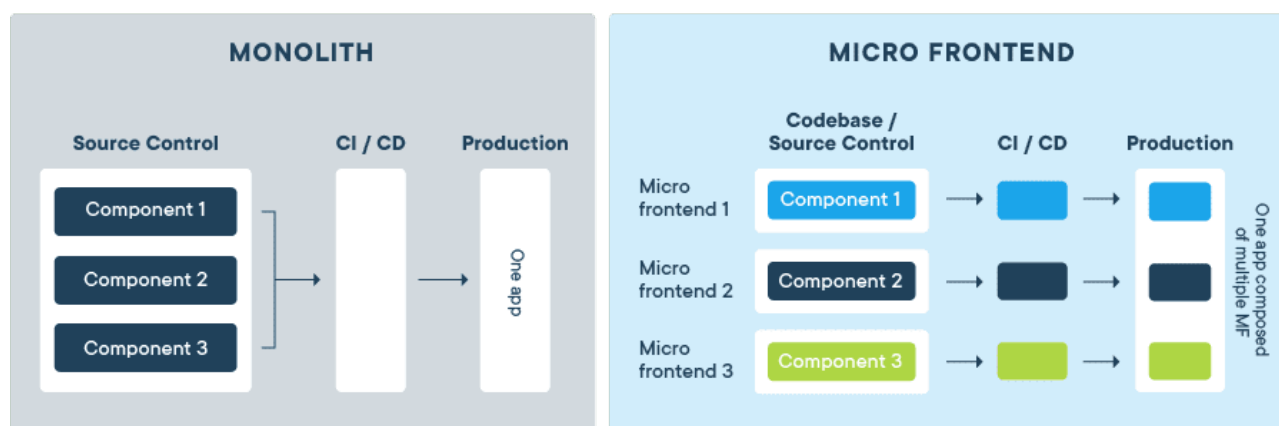


Figure 2.3: Monolithic Frontend Architecture vs Microfrontends Architecture

## 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.

# 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.

## 3.2 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.

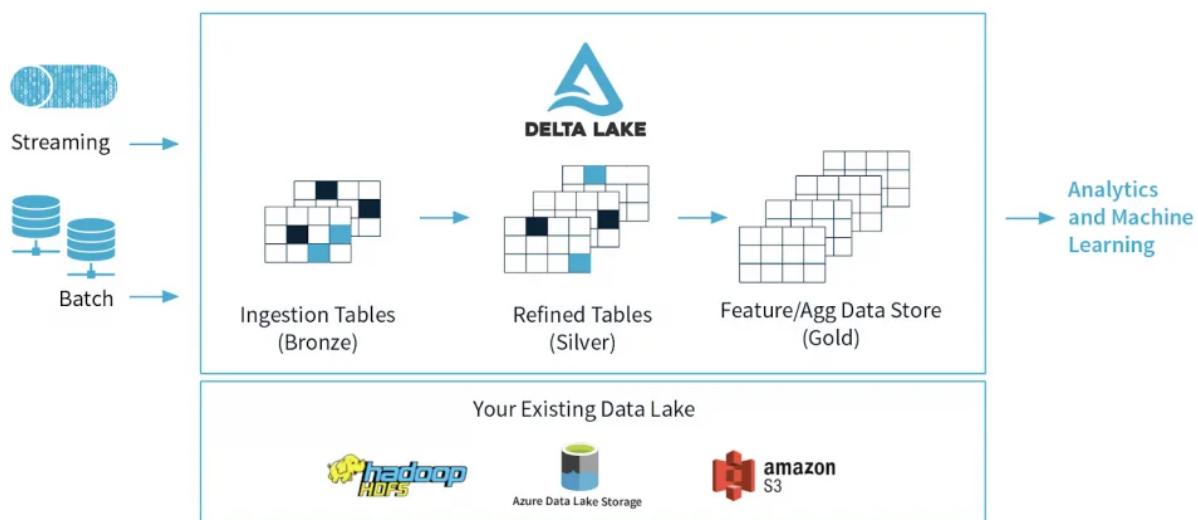


Figure 3.1: Delta Lake multi-hop architecture

- **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.

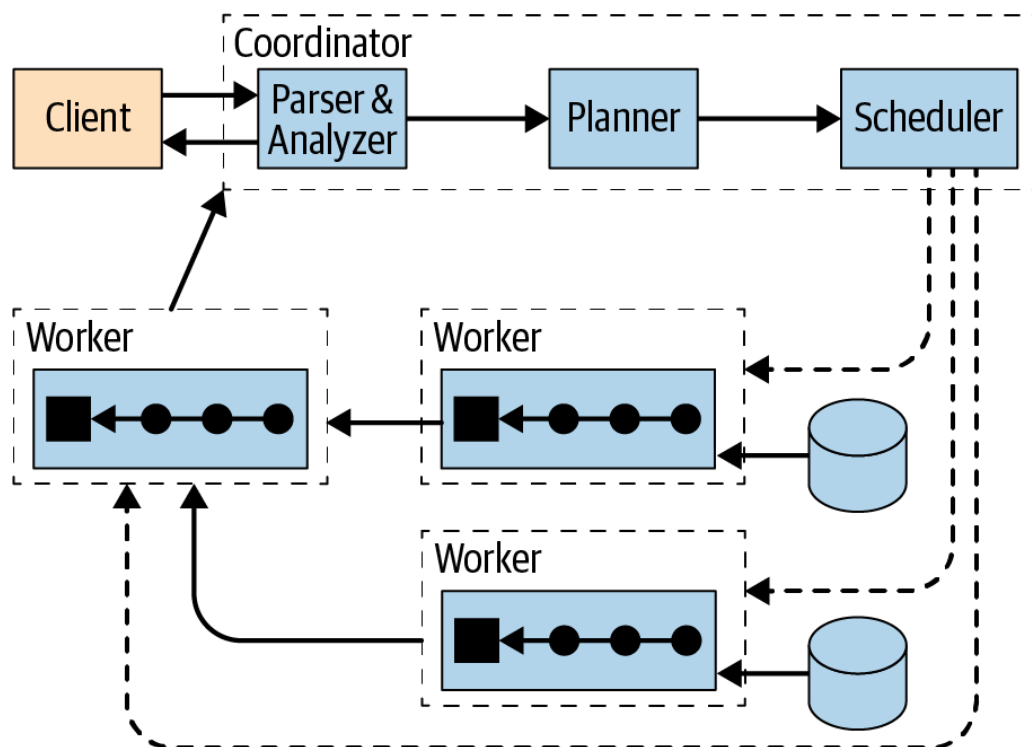


Figure 3.2: Overview of Trino architecture with coordinator and workers

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.

## 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:

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.

## 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.

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.

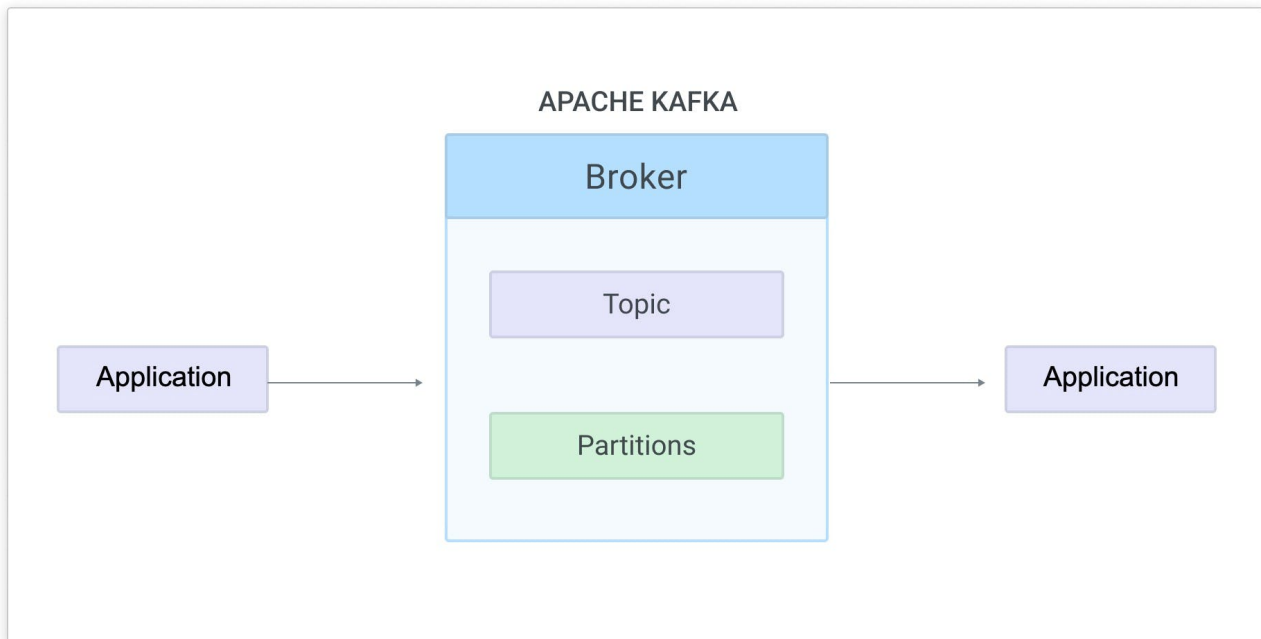


Figure 3.3: Kafka Architecture

## 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.



# 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.

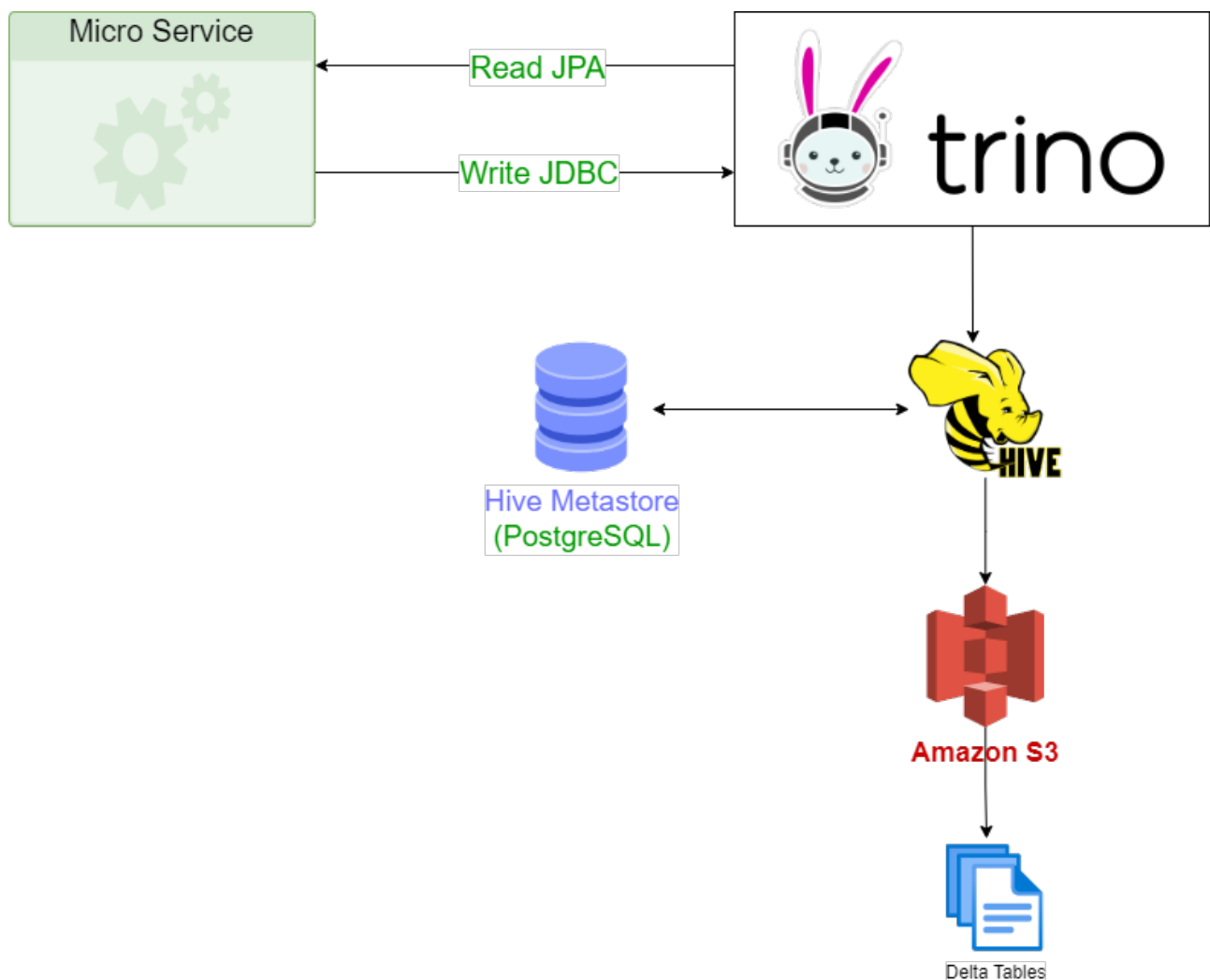


Figure 4.1: TRINO Architecture

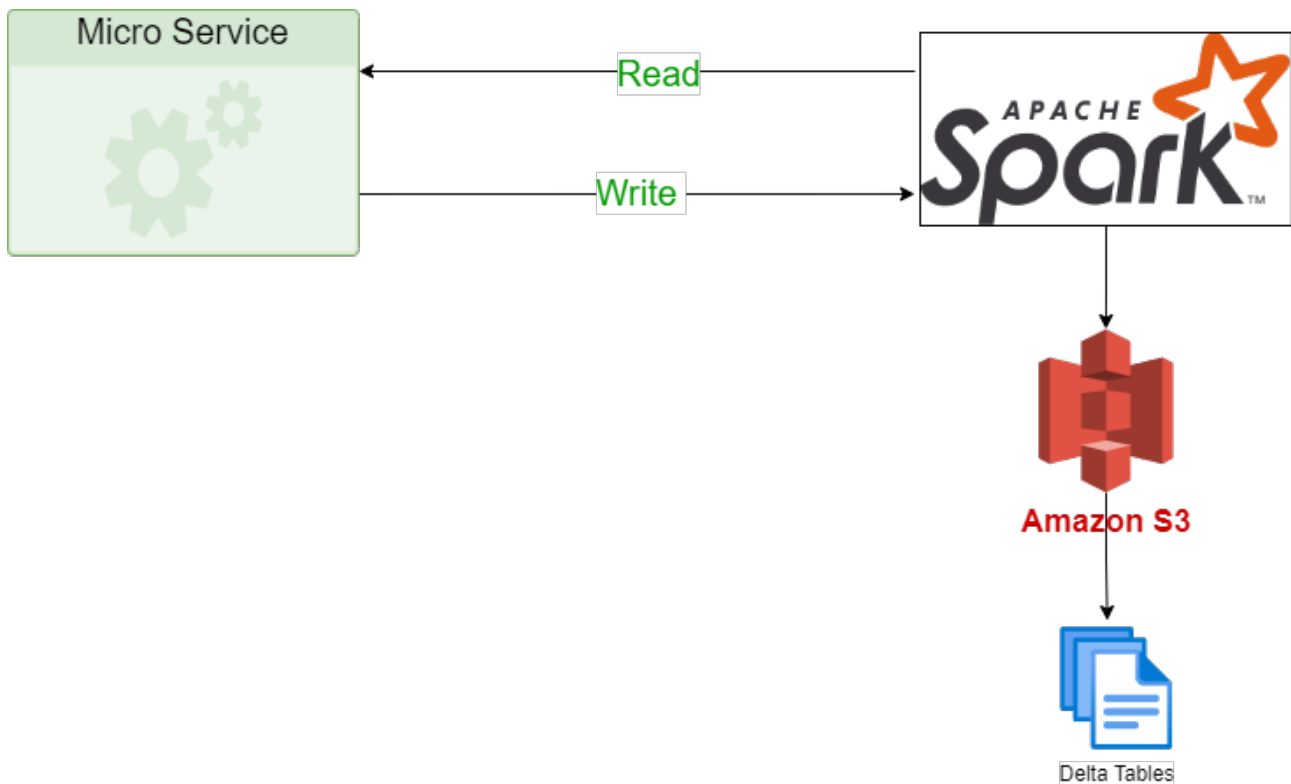


Figure 4.2: SPARK Architecture

#### 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	DDR4-SDRAM
hline Graphics Chip	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


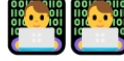




- **Trino:** Trino has an official connector for Delta Lake, allowing it to read and write data in Delta Lake. It supports essential Delta Lake features such as ACID transaction management, incremental updates, and merge operations.
- **Spark:** Spark also has built-in support for Delta Lake. It offers 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	 <p>Spark's syntax requires SMEs to have experience in data engineering</p>	 <p>Standard SQL interface, making it easier to use for users familiar with SQL.</p>

<b>Coût</b>	Team:  Infrastructure: \$	Team:  Infrastructure: \$
<b>Fiabilité</b>	 Mature product maintained internally and supported by Google Dataproc	 Has experienced a few outages as the internal team learns to maintain and scale
<b>Caractéristiques et Open sources</b>	 Has been supported by the open-source community since 2014.	 Has been supported by the open-source community since 2019. Shopify has made contributions
<b>Usage</b>	~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.

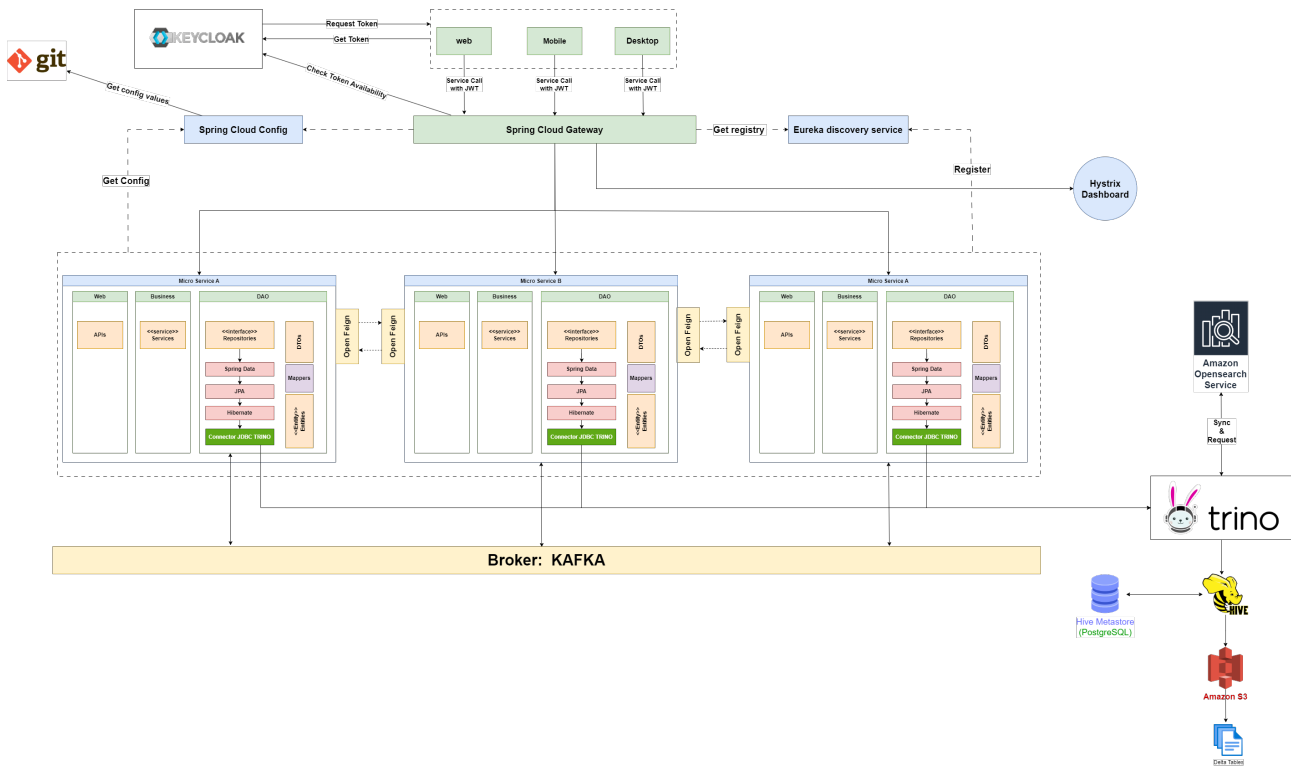


Figure 4.3: Solution Architecture

## 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.

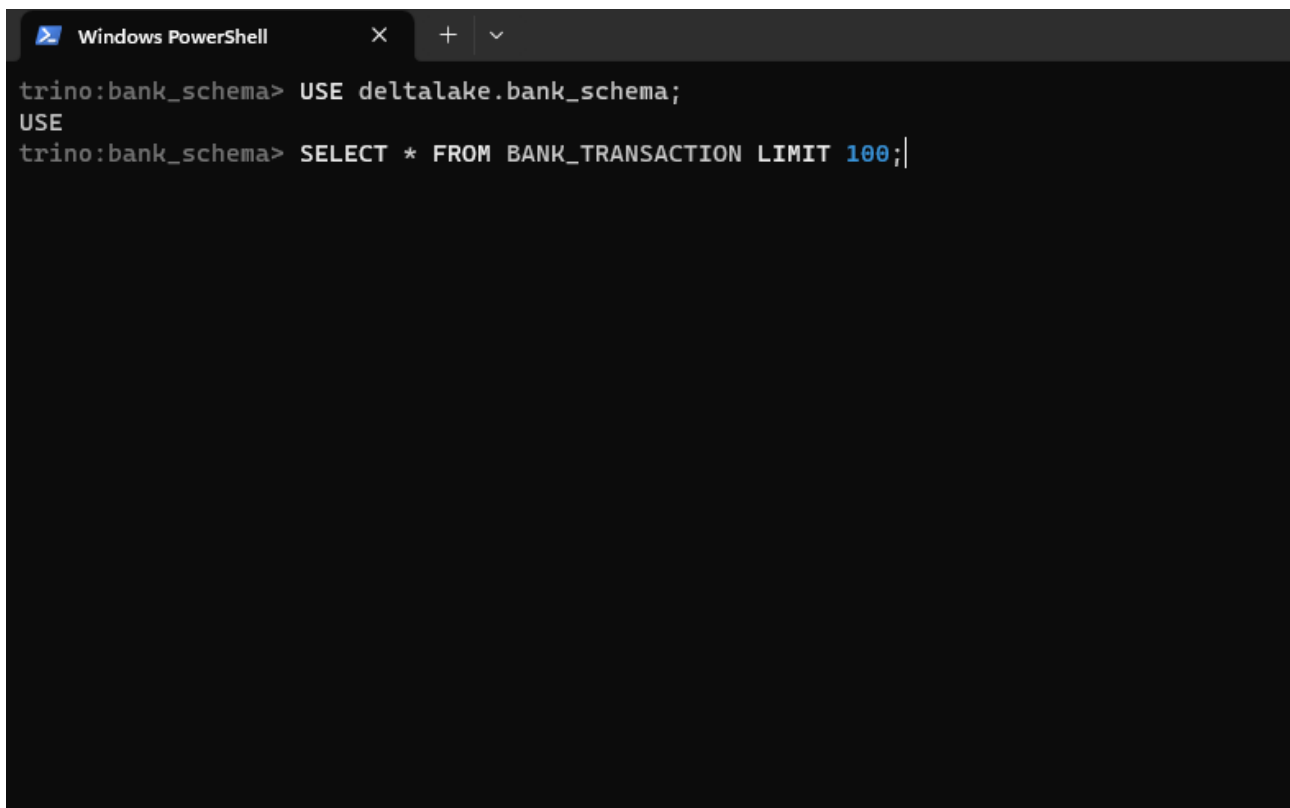
- **Enhanced Security:** Integration of Keycloak for authentication and authorization provides a high level of security. JWT tokens are used for user identification, and Spring Cloud Gateway handles token verification and management. This ensures that only authorized individuals have access to appropriate 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

A screenshot of a Windows PowerShell terminal window. The title bar at the top reads "Windows PowerShell" with standard window controls (minimize, maximize, close). The terminal content shows a Trino session with the prompt "trino:bank\_schema>". The user has entered the command "USE deltalake.bank\_schema;" followed by another "USE" command on a new line. The prompt is now "trino:bank\_schema>". The user has then entered the command "SELECT \* FROM BANK\_TRANSACTION LIMIT 100;".

```
trino:bank_schema> USE deltalake.bank_schema;
USE
trino:bank_schema> SELECT * FROM BANK_TRANSACTION LIMIT 100;|
```

Figure 4.4: Trino Console



unique_id	shop_unique_identifier	card_token	card_truncated	mcc	delivery_date	purchase_date	acceptance	contract
17b4841e-4314-4ae4-a1ed-5ea821521e79	2ade7013-a3bb-48dd-b422-5455accf27f2	30255234182628	2628	7082	2023-05-01 12:27:12.000 UTC	2023-04-28 04:35:35.000 UTC	APPROVED	No Contract
68f76e02-78ec-48e7-a945-6a6bd1f73a0f	1050f4df-7098-495b-a5d2-8c99d77b1fea	4898477745380669	8660	1475	2023-04-28 02:47:45.000 UTC	2023-04-27 20:36:21.000 UTC	DECLINED	Contract
9c9cf85d-d0b0-4d07-9d16-76bba22f7d70	a179b089-484a-411b-afe9-1b5e589baaa0	406964552688789161	9161	8182	2023-05-04 00:20:17.000 UTC	2023-04-28 06:36:21.000 UTC	DECLINED	Contract
12a8c463-34ea-408e-9079-1c6b3dbb4b3f	70dca3cd-c9fd-4216-91ac-2bc45cfab030	40804954745291234	1234	6288	2023-05-01 03:50:19.000 UTC	2023-04-28 15:49:09.000 UTC	DECLINED	Contract
7c3b12f7-88d6-44d7-a909-4d0579861f55	d71f81c7-fa8a-4448-a5ab-e98e745c6273	3597282846282577	2577	1626	2023-05-01 01:49:47.000 UTC	2023-04-27 21:34:03.000 UTC	PENDING	Contract
d5bcae67-7a1f-4d9c-9362-084fae5951a2	27f03021-ee73-46d3-80e9-7a19299dc6dd	4316471810326075	8675	1388	2023-04-29 17:48:13.000 UTC	2023-04-28 09:38:09.000 UTC	PENDING	Contract
db8c778-481c-4e56-9af4-a83946aad175	8ab653e1-c6e4-4d7f-a6ad-55507be61c1d	3538923884133671	3671	5885	2023-05-04 04:46:09.000 UTC	2023-04-28 01:36:27.000 UTC	PENDING	Contract
778e5d6d-bc0c-4d0a-b0af-53453a8af605	d3a7a225-b259-4f49-87c5-9d08ec4715b0	3575211809241331	1331	2040	2023-05-03 06:21:35.000 UTC	2023-04-27 23:29:17.000 UTC	DECLINED	No Contract
9a90c539-0d4c-4b97-a4d1-9abb7ade4cfe	11530677-c072-4dff-9f4f-35204d6d4c5f	561129793130	8130	8179	2023-05-04 20:31:22.000 UTC	2023-04-28 15:46:06.000 UTC	DECLINED	Contract
96a3ab8b-9955-4bd9-b526-47a588edc1c2	2add18a8-16ad-4347-97f3-8f0296a59386	3745482968148047	4807	4048	2023-05-01 10:13:48.000 UTC	2023-04-27 20:09:59.000 UTC	APPROVED	No Contract
3a42850d-ef52-46d9-b607-55e801ad9103	fe134455-6927-4232-bb21-2bbe9681ef8a	3580467543172268	2268	2177	2023-04-30 02:56:09.000 UTC	2023-04-28 11:33:04.000 UTC	APPROVED	No Contract
a62bf45e-052f-490c-a936-da08d71c5b52	f3f9504f-e797-4e22-8014-da24345c351	6510718116799851	9851	2619	2023-05-03 00:08:23.000 UTC	2023-04-27 19:56:47.000 UTC	PENDING	No Contract
e32f3f08-72f6-4d8f-a08a-207f16218094	ea111b0f-8fd6-4809-ba80-b02e79027276	6811537166940800	9300	4073	2023-04-28 20:40:47.000 UTC	2023-04-28 09:09:34.000 UTC	APPROVED	No Contract
b0d16623-4ff4-da22-aaab-b538154081d1	8021c5ab-8c1d-4d0c-9539-92b394970d3a	3540676132977302	7302	5411	2023-05-02 03:44:06.000 UTC	2023-04-27 02:07:25.000 UTC	PENDING	Contract
81c77082-7a07-4b9d-b911-48c73af64263	91fc285e-5a69-4e1b-9140-c915a4cf9701	3534049775620885	8885	2975	2023-04-28 05:08:43.000 UTC	2023-04-27 16:28:20.000 UTC	PENDING	Contract
3ecad7f0-a64e-4efb-a0c9-18ac3662cc30	10d485a3-7615-48fe-bfba-d5a14b11d5a4	34119809783986	3986	6328	2023-04-27 20:06:18.000 UTC	2023-04-27 23:13:11.000 UTC	PENDING	Contract
2a3c14d6-70e9-44a8-8cb7-c924d7320746	24af2bad-fc8b-4277-bbac-ba6dd508b894	341866544158818	8818	1928	2023-04-28 02:44:36.000 UTC	2023-04-27 21:50:46.000 UTC	DECLINED	Contract
1782abaf-f70e-4d51-4e6b-b1c0f4f638d0	813bb3a2-27f7-4ffe-b080-04af4d0c4cfe	6564689911807658	7658	7149	2023-04-29 09:04:07.000 UTC	2023-04-28 12:39:10.000 UTC	APPROVED	Contract
9608a35c-8ed9-48b6-a2df-ac0808f6e99a	01d050bf-35b2-4b5b-01fe-f2769f9b337e	400908858692715760	7600	1182	2023-04-29 19:00:33.000 UTC	2023-04-27 18:51:26.000 UTC	DECLINED	Contract
36a3f96d-95ad-4190-91c9-f7cb1f8e5b03	42d6dd22-5255-433e-aa97-b82a3b1317ad	4995321296781025	1425	7232	2023-05-03 00:40:08.000 UTC	2023-04-28 06:04:41.000 UTC	APPROVED	Contract
11fc1bf5-796c-4bf4-b8f3-bba55dabdbd1f	8754c27a-04f0-494e-9bf7-dc7d23c3f3de	2296062779381552	1552	8999	2023-05-01 01:06:52.000 UTC	2023-04-28 08:49:47.000 UTC	DECLINED	No Contract
8bb1eb3d-66a5-4611-b6ac-3af52553bc87	57d6c2d9-0694-4907-b591-1f9bc1700a0e	2260787394983854	3854	5186	2023-04-29 05:05:09.000 UTC	2023-04-28 10:12:33.000 UTC	PENDING	Contract
1024e7d3-0b71-4b5b-93f0-d1b02c94d011	fd029f5e-555e-4b33-9b36-59d9e70d7962	30186869180876	8076	5202	2023-04-30 20:53:24.000 UTC	2023-04-28 05:07:47.000 UTC	DECLINED	No Contract
af57d43d-e55a-4a8e-b603-afb8d4f60908	79f8a594-4a54-432c-a8b0-b8fec899dd81	2131101582007171	7171	1670	2023-05-03 05:16:51.000 UTC	2023-04-28 09:14:40.000 UTC	PENDING	No Contract
08e5977d-8f50-4d8b-9f3f-9a8a92ad0e1c	3ca2d9ff-4c21-4ba5-8f78-6caa3479c4f7	213116560776278	6278	5545	2023-05-04 23:17:01.000 UTC	2023-04-27 22:01:32.000 UTC	PENDING	Contract
86e95212-70d1-41b8-851b-dcd7840d081c	c213b438-bf1b-43ad-9c0f-adcf184f0e5d	3348663908410833	4033	4074	2023-05-05 13:03:34.000 UTC	2023-04-28 03:22:16.000 UTC	DECLINED	Contract
d080a5a6-71c0-4414-9d08-0816f3432af	d730f85a-e7c5-4850-8b5b-a993dad435b0	3598364681221220	1220	5884	2023-05-02 14:26:04.000 UTC	2023-04-27 19:25:33.000 UTC	PENDING	No Contract
f70112b3-b092-4d4e-b092-1568370e12ee	e9c303d9-2cdc-40ff-a858-c8b9116f3d02	3518970697177268	7268	7064	2023-05-03 23:21:01.000 UTC	2023-04-28 11:37:21.000 UTC	DECLINED	Contract
934aa8f0-fbce-4db0-b393-b13bed297afe	0704e25b-edf0-4c88-8f91-ebc30bc6f12d	3585370274308065	8065	6777	2023-05-04 02:54:29.000 UTC	2023-04-28 10:31:15.000 UTC	PENDING	No Contract
e4c38e2-4d70-4948-8acc-08e17700bb7c	c5e0dc21-60b8-4983-aeb4-2d10380b63bd	6011143253391009	1009	9277	2023-05-04 00:07:11.000 UTC	2023-04-27 20:34:59.000 UTC	APPROVED	Contract
a5f5c535-1ea7-4cdd-a862-a95b94626228	6b355f8f-6ae7-404e-a4d7-50324392fdde	4352933080192179	2179	1943	2023-05-03 16:09:33.000 UTC	2023-04-28 14:11:12.000 UTC	PENDING	No Contract
8a94b8ff-04e7-4c4d-b506-dbf1943080c0	98982706-1b0a-4a43-b455-95caabdb85cd	30123665808804	8804	2018	2023-04-28 07:51:25.000 UTC	2023-04-28 15:37:21.000 UTC	APPROVED	No Contract
af0e2a3f-dfca-4b29-47aa-15af6d6d6c0a	c1650f2e-b402-b48c-c4c6-666366207073	3612866366207073	2073	1019	2023-05-02 12:23:24.000 UTC	2023-04-27 10:47:06.000 UTC	PENDING	Contract
aad9f64a-131a-4604-bd2e-a759cafb0601e	08d8d8fe-cba0-44f3-9df2-52ef79494107	2386234813355826	5826	8542	2023-05-01 11:54:11.000 UTC	2023-04-28 15:26:47.000 UTC	DECLINED	Contract
7c78855d-4d5e-4729-999a-b6f1def6d2a7	53dc65da-1241-496e-ac6e-1b780b5581c3	213127433175767	5767	2074	2023-04-28 21:06:11.000 UTC	2023-04-28 00:50:42.000 UTC	PENDING	No Contract
c227ba5d-bb9c-4562-9530-c9c95e155707	51cd1fc3-5af1-4a16-a6d7-a62a7cb080b7	4156095903490385	8385	7926	2023-04-28 19:32:44.000 UTC	2023-04-28 13:12:08.000 UTC	DECLINED	Contract
25dddbce-a806-40ef-bb14-140c92ab2c80	907d99cc-e72f-4a01-bbe1-21315c0427dc	4286664428075219	5219	2797	2023-04-30 02:37:22.000 UTC	2023-04-27 21:16:16.000 UTC	PENDING	No Contract
90c3c06a-9e0d-4b5b-91d3-174e0c0c0c04	1b7f9f5e-3a0a-4083-939e-3506f070c774	180036023891072	1072	7357	2023-05-02 11:47:35.000 UTC	2023-04-27 21:12:22.000 UTC	APPROVED	Contract
36fd4db8-4d73-4c30-974a-dfbcbcd067a2	7c5cf204-95e8-4e9d-9f68-636f03747206	4745493651282394352	4352	2002	2023-04-30 02:03:55.000 UTC	2023-04-27 18:58:50.000 UTC	PENDING	Contract
86a6732f-9818-42bb-a81b-41b4867df81e	fc93f919-d09b-496e-921a-77ad5a1801fe	3583337345408436	4836	8278	2023-05-01 17:49:11.000 UTC	2023-04-27 18:38:18.000 UTC	DECLINED	No Contract
c4b51bde-a62f-4a38-b10c-31598c80403c	30fcf45a-ef0b-440a-8a1f-9c9b181ba7e8	60110252639408273	8273	6505	2023-04-30 12:56:16.000 UTC	2023-04-28 03:35:45.000 UTC	PENDING	No Contract
09370c0f-8708-9312-81ed-590b051650b0	634290cc-0500-4719-8f43-c0326e0400b0	6011080352075308	7538	1205	2023-05-03 11:02:22.000 UTC	2023-04-27 23:07:49.000 UTC	DECLINED	No Contract
c5c4f96f-9f3d-4e37-8000-65f440b0b0ea	f531a0e4-1d09-420c-471b-601372ba0b30	370546881490356	9356	1520	2023-05-03 23:46:28.000 UTC	2023-04-28 12:41:58.000 UTC	PENDING	No Contract
94a0b26c-479e-4703-862c-f7f95e3d8764	293db82b-f5b2-430a-a08e-04a0a0edf7da	3570846992779936	9936	3593	2023-05-01 07:24:36.000 UTC	2023-04-28 04:01:47.000 UTC	APPROVED	No Contract
30a42b9f-4eb4-4779-9759-a0967fb6520f	ce3d2f3f-6176-4870-8au3-288e0b6c9f77	3590866278016592	6592	8041	2023-05-03 11:28:04.000 UTC	2023-04-28 09:38:12.000 UTC	DECLINED	Contract
ef839925-65d3-4b7f-9166-1db0314753bc	3c929232-f127-4e5b-b09e-5b4d137be039	5759591878759	8759	2801	2023-05-01 10:21:24.000 UTC	2023-04-28 08:17:02.000 UTC	DECLINED	Contract
c9f08f6e-f072-400a-a722-27120f13a994	6e20807a-b755-400d-0a78-0ef5d30e0038	213170470680124	1024	1949	2023-05-01 13:45:56.000 UTC	2023-04-28 08:52:43.000 UTC	DECLINED	No Contract
97a5b975-302c-4b35-a009-5f9f337f7f21	7c0a9d68-3d6e-4af0-b395-de27f1dc0c08	50181082284	2284	3276	2023-04-28 20:57:35.000 UTC	2023-04-28 13:24:10.000 UTC	APPROVED	No Contract

Figure 4.5: Trino SELECT Query

```

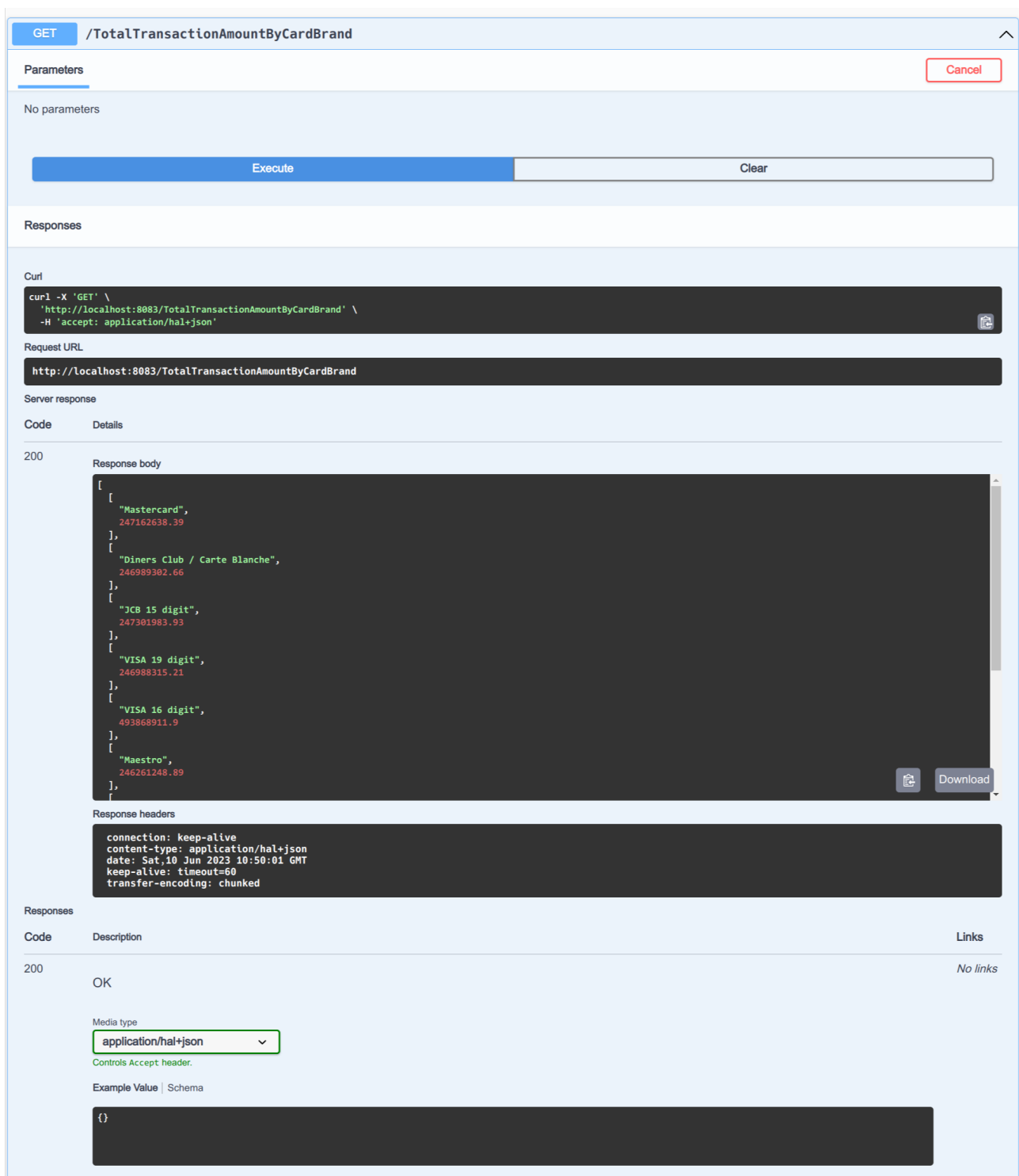
Query 20230609_150126_00016_ffeez, FINISHED, 1 node
Splits: 10 total, 10 done (100.00%)
0.22 [127 rows, 262KB] [579 rows/s, 1.17MB/s]

```

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



The image shows the Swagger UI for an endpoint named `/TotalTransactionAmountByCardBrand`. The interface is divided into several sections:

- GET /TotalTransactionAmountByCardBrand**: The endpoint name and method.
- Parameters**: A section with a "Cancel" button and the text "No parameters".
- Execute**: A blue button to execute the request.
- Clear**: A button to clear the request.
- Responses**: A section showing the response details.
- Curl**: A text area containing the curl command:

```
curl -X 'GET' \
  'http://localhost:8083/TotalTransactionAmountByCardBrand' \
  -H 'accept: application/hal+json'
```
- Request URL**: A text area containing the URL:

```
http://localhost:8083/TotalTransactionAmountByCardBrand
```
- Server response**: A section showing the response details.
- Code**: A table with a single row showing the status code `200`.
- Details**: A section showing the response body and headers.
- Response body**: A text area containing the JSON response:

```
{
  [
    [
      "Mastercard",
      247162638.39
    ],
    [
      "Diners Club / Carte Blanche",
      246989302.66
    ],
    [
      "JCB 15 digit",
      247301983.93
    ],
    [
      "VISA 19 digit",
      246988315.21
    ],
    [
      "VISA 16 digit",
      493868911.9
    ],
    [
      "Maestro",
      246261248.89
    ]
  ]
}
```
- Response headers**: A text area containing the headers:

```
connection: keep-alive
content-type: application/hal+json
date: Sat, 10 Jun 2023 10:50:01 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```
- Responses**: A table with a single row showing the status code `200` and the description `OK`.
- Links**: A section showing the media type `application/hal+json` and the example value `{}`.

Figure 4.7: Swagger UI of Endpoint 1

GET /AverageTransactionAmountByTransactionType

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:8083/AverageTransactionAmountByTransactionType' \
-H 'accept: application/hal+json'
```

Request URL

http://localhost:8083/AverageTransactionAmountByTransactionType

Server response

Code	Details
200	<p>Response body</p> <pre>[   {     "BALANCE_TRANSFER",     505.7176257615425   },   {     "PURCHASE",     505.6615044535439   },   {     "CASH_ADVANCE",     505.4125505839378   },   {     "REFUND",     505.1762060201541   } ]</pre> <p>Response headers</p> <pre>connection: keep-alive content-type: application/hal+json date: Sat, 10 Jun 2023 10:53:02 GMT keep-alive: timeout=60 transfer-encoding: chunked</pre>

Responses

Code	Description	Links
200	OK	No links

Media type

application/hal+json

Controls Accept header.

Example Value | Schema

```
{}
```

Figure 4.8: Swagger UI of Endpoint 2

The image displays the Swagger UI for a REST API endpoint. The endpoint is a GET request to `/TotalTransactionCountByCountry`. The response is a 200 status code with a JSON body containing a list of countries and their transaction counts. The response headers include `connection: keep-alive`, `content-type: application/hal+json`, `date: Sat, 10 Jun 2023 10:48:47 GMT`, `keep-alive: timeout=60`, and `transfer-encoding: chunked`.

**Parameters**

No parameters

**Execute** **Clear**

**Responses**

**Curl**

```
curl -X 'GET' \
  'http://localhost:8083/TotalTransactionCountByCountry' \
  -H 'accept: application/hal+json'
```

**Request URL**

```
http://localhost:8083/TotalTransactionCountByCountry
```

**Server response**

**Code** **Details**

200

**Response body**

```
[
  {
    "country": "Armenia",
    "count": 23908
  },
  {
    "country": "Aruba",
    "count": 23821
  },
  {
    "country": "Australia",
    "count": 23797
  },
  {
    "country": "Austria",
    "count": 23918
  },
  {
    "country": "Azerbaijan",
    "count": 23930
  },
  {
    "country": "Bahamas",
    "count": 23935
  }
]
```

**Response headers**

```
connection: keep-alive
content-type: application/hal+json
date: Sat, 10 Jun 2023 10:48:47 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

**Responses**

Code	Description	Links
200	OK	No links

**Media type**

application/hal+json

**Controls Accept header.**

**Example Value** | **Schema**

```
{}
```

Figure 4.9: Swagger UI of Endpoint 3

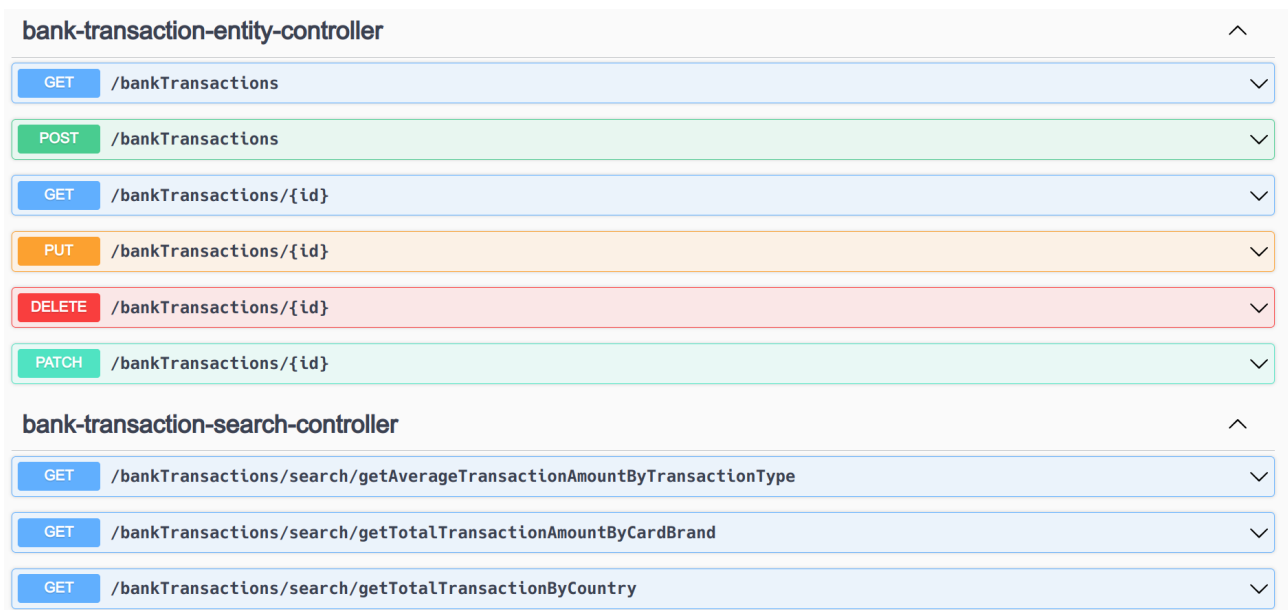
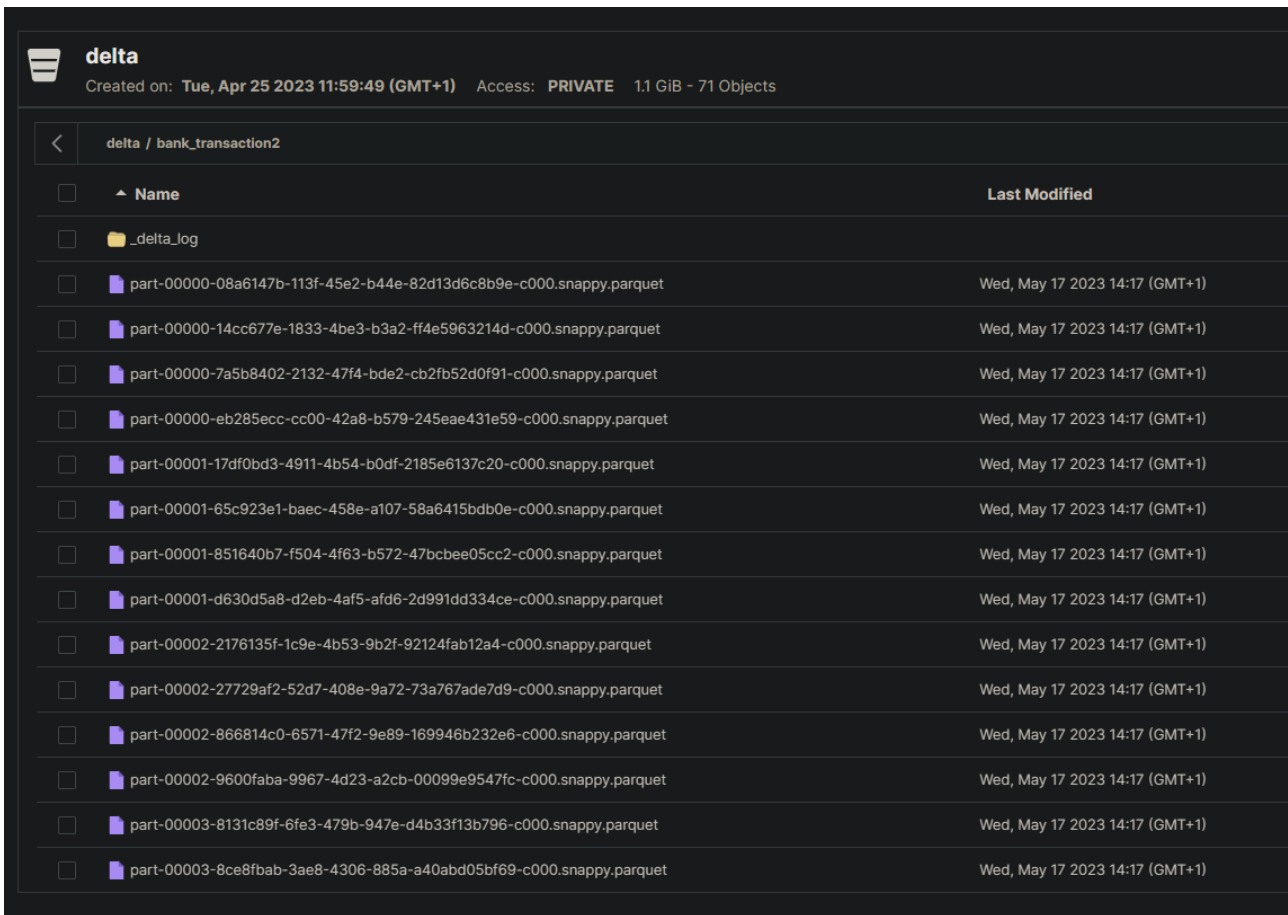


Figure 4.10: Swagger UI

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



delta	
Created on: Tue, Apr 25 2023 11:59:49 (GMT+1) Access: PRIVATE 1.1 GiB - 71 Objects	
delta / bank_transaction2	
Name	Last Modified
_delta_log	
part-00000-08a6147b-113f-45e2-b44e-82d13d6c8b9e-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00000-14cc677e-1833-4be3-b3a2-ff4e5963214d-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00000-7a5b8402-2132-47f4-bde2-cb2fb52d0f91-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00000-eb285ecc-cc00-42a8-b579-245eae431e59-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00001-17df0bd3-4911-4b54-b0df-2185e6137c20-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00001-65c923e1-baec-458e-a107-58a6415bdb0e-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00001-851640b7-f504-4f63-b572-47bcbee05cc2-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00001-d630d5a8-d2eb-4af5-afd6-2d991dd334ce-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00002-2176135f-1c9e-4b53-9b2f-92124fab12a4-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00002-27729af2-52d7-408e-9a72-73a767ade7d9-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00002-866814c0-6571-47f2-9e89-169946b232e6-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00002-9600faba-9967-4d23-a2cb-00099e9547fc-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00003-8131c89f-6fe3-479b-947e-d4b33f13b796-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)
part-00003-8ce8fbab-3ae8-4306-885a-a40abd05bf69-c000.snappy.parquet	Wed, May 17 2023 14:17 (GMT+1)

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

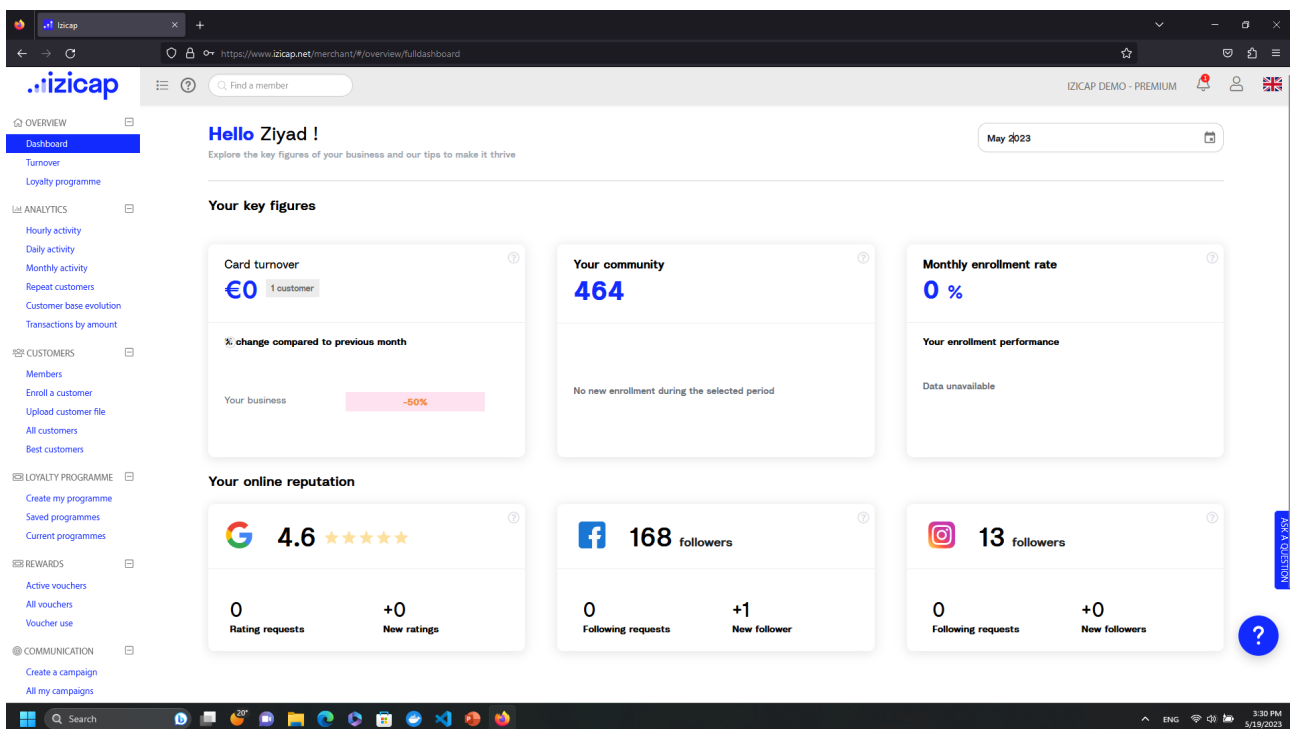


Figure 4.12: Izicap Portal

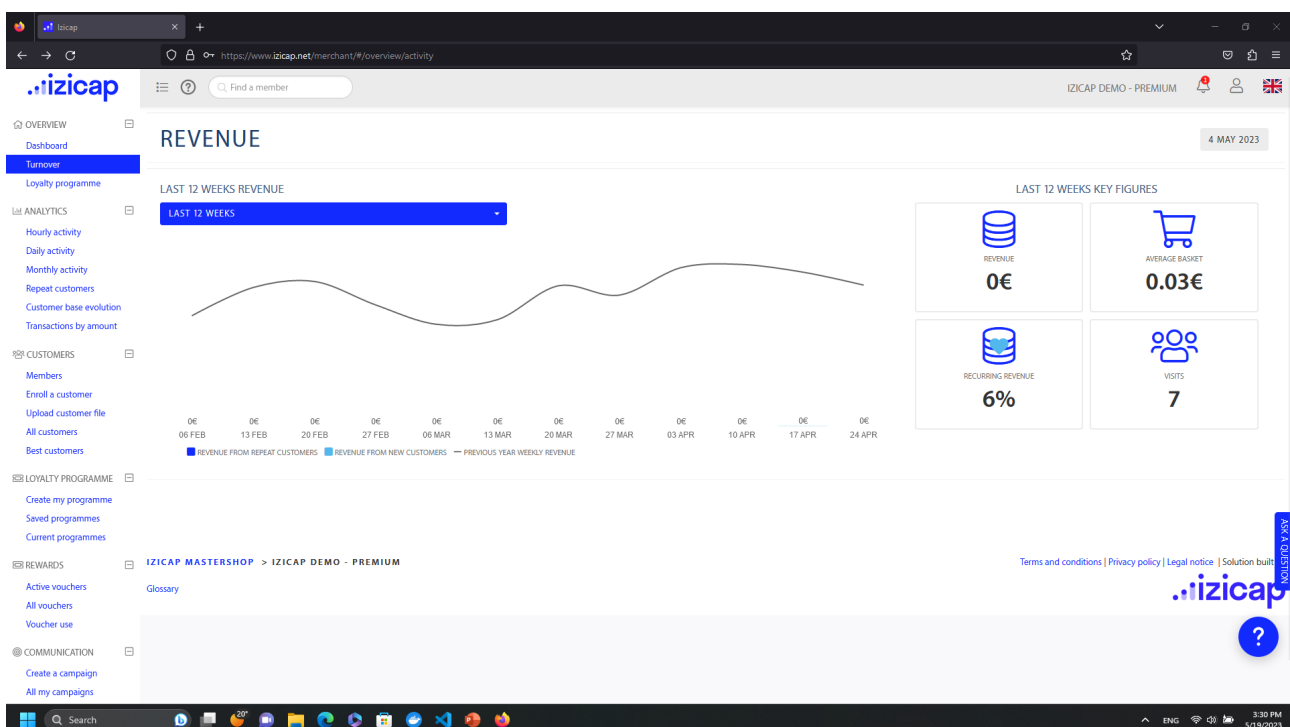


Figure 4.13: Customer Revenue

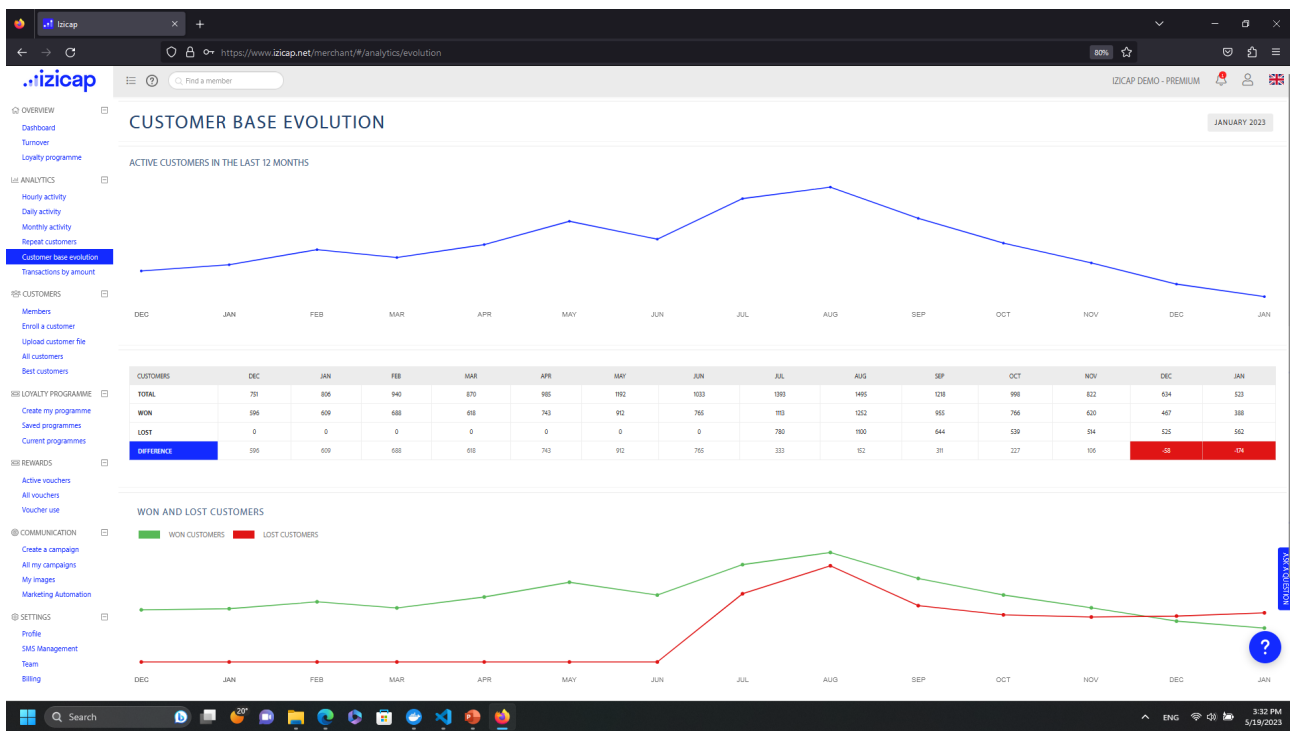


Figure 4.14: Customer Base Evolution

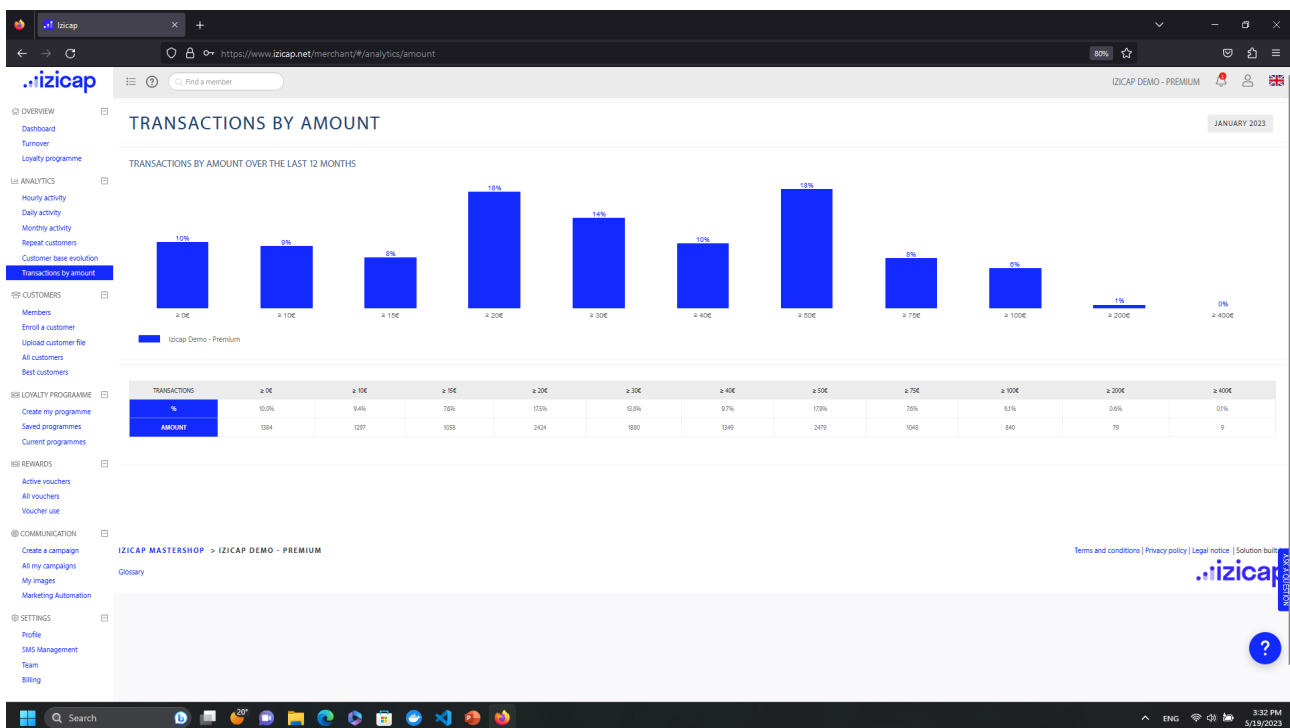


Figure 4.15: Transactions by Amount

The use of micro-frontends allows 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.

---