



MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
DE LA RECHERCHE SCIENTIFIQUE ET DE L'INNOVATION
UNIVERSITÉ SULTAN MOULAY SLIMANE
ECOLE NATIONALE DES SCIENCES APPLIQUÉS
DE KHOURIBGA



Mémoire de fin d'études

pour l'obtention du

Diplôme d'Ingénieur d'État

Filière : Informatique et Ingénierie des Données



Présenté par :

Ziyad RAKIB

Participation à la migration d'une architecture monolithique à une architecture microservices

Sous la supervision :

Abdelghani Ghazdali (interne)

Nidal Lamghari (interne)

Bilal Slayki (externe)

Membres du jury :

OURDOU Amal	Entité	Président
OUMMI Hssaine	Entité	Examineur
Nom et prénoms du rapporteur	Entité	Rapporteur
Nom et prénoms du rapporteur	Entité	Rapporteur

Année Académique : 2023

Sommaire

Dédicace	ii
Remerciements	iii
Résumé	iv
Abstract	v
Table des matières	vii
Table des figures	ix
Liste des tableaux	x
Introduction	1
1 Contexte général du projet	2
2 Informatique décisionnelle et Choix de solutions	13
3 Technologies Utilisées	24
4 Implementation de la solution	31
Conclusion	39

Dédicace

“

À ma mère, qui m'a comblée de son soutien et dévouée moi avec un amour inconditionnel. Tu es pour moi un exemple de courage et sacrifice continuuel. Que cet humble travail porte témoignage de mon affection, de mon attachement éternel et qu'il appelle sur moi ta bénédiction continuelle.,

À mon père, aucune dédicace ne peut exprimer l'amour, l'estime, dévouement et le respect que j'ai toujours eu pour vous. Rien dans le monde vaut les efforts faits jour et nuit pour mon éducation et mon bien-être. Ce travail est le fruit de vos sacrifices que vous avez fait pour mon éducation et ma formation,

À mes chers frères, merci pour votre amour, soutien, et encouragements,

À tous mes chers amis, pour le soutien que vous m'avez apporté, je dis Merci encore une fois à tous ceux qui me sont chers, à vous tous

Merci.

”

- Ziyad

Remerciements

Tout d’abord, je remercie le grand Dieu puissant de nous donné la puissance pour continuer et dépasser toutes les difficultés.

J’adresse mes remerciements à, **Monsieur Abdelghani Ghazdali**, chef de filière Informatique et Ingénierie des Données qui a su assurer le bon déroulement des séances d’encadrement du début à la fin.

Je tiens également à adresser mes sincères remerciements à **Professeur Nidal Lamghari**, pour ses précieux conseils, son expertise et sa disponibilité tout au long de cette expérience. Votre encadrement attentionné m’a permis d’approfondir mes connaissances et de progresser dans mes compétences techniques. Je suis reconnaissant d’avoir pu bénéficier de votre expertise et de votre soutien constant.

Au terme de ce travail, je tiens également à exprimer mes sincères remerciements et ma gratitude envers tous ceux qui, par leur enseignement, par leur soutien et leurs conseils, ont contribué au déroulement de ce projet.

Par ailleurs, je tiens à exprimer ma gratitude envers **IZICAP**, où j’ai eu la chance de réaliser mon projet de fin d’études. Je suis reconnaissant envers toute l’équipe de **IZICAP** pour m’avoir accueilli chaleureusement et pour m’avoir permis de mettre en pratique mes connaissances académiques dans un environnement professionnel. Votre expertise, votre mentorat et vos ressources ont été d’une valeur inestimable pour la réalisation de ce projet.

Je tiens à remercier spécialement **M. Bilal SLAYKI**, mon encadrant en entreprise, pour son soutien constant, ses orientations judicieuses et son accompagnement précieux tout au long de ce projet. Votre expérience, vos conseils éclairés et votre bienveillance m’ont inspiré et ont grandement contribué à ma croissance professionnelle. Je vous suis reconnaissant d’avoir partagé votre expertise avec moi et d’avoir été un mentor exceptionnel.

Finalement, mes remerciements s’adressent aussi à l’ensemble du corps professoral et administratif de l’ENSA Khouribga pour l’effort qu’ils fournissent afin de nous garantir une bonne formation et à l’équipe administrative et technique pour tous les services offerts.

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.

Table des matières

Dédicace	ii
Remerciements	iii
Résumé	iv
Abstract	v
Table des matières	vii
Table des figures	ix
Liste des tableaux	x
Introduction	1
1 Contexte général du projet	2
Introduction	2
1.1 Introduction	2
1.2 Présentation de l'organisme d'accueil	2
1.2.1 IZICAP	2
1.2.2 Partenariats stratégiques	3
1.3 Organigramme	4
1.4 Cadre du projet	5
1.5 L'architecture du système -Smart Data-	6
1.6 Problématique et besoin fonctionnel	8
1.7 Objectives du stages	9
1.8 Processus de réalisation du projet	10
1.9 Planification du projet	11
Conclusion	12
2 Informatique décisionnelle et Choix de solutions	13
Introduction	13
2.1 Différence entre un data warehouse, un data lake, un datalakehouse et un delta lake	13
2.1.1 Data Warehouse	14
2.1.2 Data Lake	14
2.1.3 Datalakehouse	15

2.1.4	Delta Lake (Datalakehouse)	15
2.2	Différence entre une architecture microservices et une architecture monolithique	16
2.2.1	Avantages et Inconvénients de l'architecture monolithique	16
2.2.2	Avantages et Inconvénients de l'architecture microservices	17
2.3	Différence entre une architecture microfrontends et une architecture monolithique frontend	19
2.3.1	Avantages et Inconvénients du monolithe frontend	19
2.3.2	Avantages et Inconvénients de l'architecture microfrontends	20
2.4	Cheminement de la solution	21
2.4.1	Partie Data	21
2.4.2	Partie Backend	21
2.4.3	Partie Frontend	22
	Conclusion	22
3	Technologies Utilisées	24
	Introduction	24
3.1	Delta Lake	24
3.2	Principaux avantages et caractéristiques de Delta Lake	25
3.3	Trino	27
3.4	Springboot	28
3.5	Keycloak	28
3.6	Kafka	29
4	Implementation de la solution	31
	Introduction	31
4.1	Étude de faisabilité	31
4.2	Benchmark entre Trino et Spark	32
4.2.1	Architecture	32
4.2.2	Temps d'exécution	33
4.2.3	Gestion de la mémoire	34
4.2.4	Écosystème et intégrations	35
4.2.5	Évolutivité	35
4.2.6	Support des fonctionnalités Delta Lake	35
4.2.7	Facilité d'utilisation	35
4.2.8	Autres statistiques	36
4.3	Schématisation de la Solution	36
4.4	Avantages	37
	Conclusion	39

Table des figures

1.1	Partenariats stratégiques	4
1.2	Organigramme de Izicap	4
1.3	Smart data Izicap	6
1.4	Architecture du système actuelle	7
1.5	Méthodologie Scrum	11
1.6	Diagramme de Gantt	11
2.1	Data Warehouse vs Data Lake vs Data Lakehouse	15
2.2	Architecture monolithique vs microservice	18
2.3	Architecture monolithique frontend vs microfrontends	21
2.4	Delta lake connectés à des microservices	22
2.5	Schéma des microfrontends	22
3.1	Architecture multi-sauts de Delta Lake	25
3.2	Vue d'ensemble de l'architecture Trino avec le coordinateur et les workers	27
3.3	Architecture Kafka	30
4.1	Architecture TRINO	32
4.2	Architecture SPARK	33
4.3	Architecture de solution	37

Liste des tableaux

1	List of Abbreviations	xi
4.1	Caractéristiques techniques du PC « DEL Latitude 5530 »	34
4.2	Caractéristiques techniques du PC «ASUS ROG G752 VT»	34
4.3	Caractéristiques techniques du PC «ASUS ROG G752 VT»	34

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

Introduction Générale

Dans un monde numérique en constante évolution, les entreprises sont confrontées à des défis majeurs pour maintenir leurs applications à la pointe de la technologie et répondre aux attentes croissantes des utilisateurs. L'architecture monolithique présente des limitations en termes de flexibilité, de maintenabilité, de performances et de modularité, ce qui pousse de nombreuses organisations à adopter des architectures basées sur des microservices.

Ce rapport met en évidence la démarche entreprise pour transformer une architecture monolithique en Groovy vers une architecture basée sur des microservices en utilisant Java, tout en migrant de AngularJS vers React et en adoptant une approche de microfrontend. L'objectif principal de cette transformation était d'améliorer la flexibilité, la maintenabilité, les performances, la cohérence des données et la modularité de l'application.

La première étape a été d'analyser l'architecture monolithique existante et d'identifier les composants pouvant être isolés en microservices indépendants. En parallèle, une évaluation des avantages de migrer de AngularJS vers React a été réalisée.

La conception et la mise en œuvre des microservices ont été réalisées en utilisant Spring-Boot, en veillant à découpler les fonctionnalités et à assurer une communication efficace entre les services. Le code AngularJS a été progressivement migré vers React en utilisant les meilleures pratiques de développement React.

Pour garantir la cohérence des données entre les microservices, la technologie de gestion de données incrémentielle Delta Lake a été utilisée. Des connecteurs en Java ont facilité la communication entre le frontend basé sur React et Delta Lake, assurant une connexion robuste et sécurisée.

En parallèle, une approche de microfrontend a été adoptée pour le frontend, permettant de découpler les fonctionnalités en modules autonomes.

Ce rapport détaillera chaque étape de la transformation, mettant l'accent sur les décisions prises, les défis rencontrés et les résultats obtenus. Il soulignera les avantages de l'architecture basée sur des microservices, l'utilisation de React et du microfrontend, ainsi que l'utilisation de Kubernetes et Docker. Cette étude de cas offre une perspective précieuse sur la modernisation des applications et les meilleures pratiques de développement dans un environnement en constante évolution.

Contexte général du projet

1.1 Introduction

L'objectif de cette partie est de présenter le contexte général du projet, en commençant par l'organisme d'accueil, IZICAP, où j'ai effectué un stage de 5 mois. Nous décrirons leurs activités, leur stratégie, leurs valeurs, ainsi que l'organigramme et les profils impliqués dans mon projet. Ensuite, nous aborderons la demande de travaux qui exprime les besoins fonctionnels sur lesquels je me suis basé pour concevoir et réaliser la solution, ainsi que la méthodologie suivie pour atteindre les objectifs fixés.

1.2 Présentation de l'organisme d'accueil

1.2.1 IZICAP

IZICAP est une société unique et novatrice axée sur l'analyse de données, la fidélisation et le marketing numérique pour les petites et moyennes entreprises (PME). Leur solution conviviale et facile à installer repose sur le modèle SaaS, permettant d'exploiter les données transactionnelles des cartes de paiement des clients.

En collaborant avec les banques et les acquéreurs, IZICAP offre une proposition de valeur renforcée et mettons à leur disposition un outil puissant de marketing numérique destiné aux PME. Leur mission est d'accompagner les commerçants au quotidien pour accélérer le développement de leur entreprise, répondant ainsi à leurs besoins spécifiques.

L'histoire d'IZICAP remonte à 2013, lorsque elle a vu le jour dans un paysage des services financiers en constante évolution et de plus en plus complexe. Depuis lors, les modèles bancaires et de paiement traditionnels sont continuellement menacés par de nouvelles entreprises challengers plus innovantes. Une concurrence agressive signifie que la fidélité des commerçants ne peut plus être considérée comme acquise, et souvent les acquéreurs et les fournisseurs de services de paiement sont contraints de réduire leurs frais.

Après s'être solidement implantée en France grâce à des partenariats avec le Groupe BPCE et le Crédit Agricole, Izicap a établi un partenariat avec Nexi, le principal acquéreur et fintech en Italie, et a rejoint le programme StartPath de Mastercard dans le but d'étendre considérablement sa portée à l'échelle mondiale. Cette expansion stratégique renforce Leur position sur le marché et confirme Leur engagement à offrir leur services à un public plus large.

Avec Izicap, les PME ont accès à une solution innovante qui tire parti de l'analyse de données pour optimiser leur stratégie de fidélisation et de marketing. Grâce à leur expertise et à leur présence internationale croissante, IZICAP est déterminée à soutenir la croissance des entreprises et à les aider à prospérer dans un environnement commercial en constante évolution.

1.2.2 Partenariats stratégiques

IZICAP a établi des partenariats stratégiques avec plusieurs acteurs clés de l'industrie financière. Parmi ses partenaires notables figurent :

- **Groupe BPCE** : Izicap a collaboré avec le Groupe BPCE, l'un des principaux groupes bancaires en France, pour fournir sa solution de fidélisation et de marketing numérique aux commerçants.
- **Crédit Agricole** : Izicap a établi un partenariat avec Crédit Agricole, l'une des plus grandes banques françaises, afin d'offrir ses services aux commerçants de leur réseau.
- **Nexi** : Izicap a noué un partenariat avec Nexi, le principal acquéreur et fintech en Italie. Cette collaboration permet à Izicap d'étendre sa présence sur le marché italien et de bénéficier de l'expertise de Nexi dans le domaine des services de paiement.
- **Mastercard StartPath** : Izicap a rejoint le programme StartPath de Mastercard, qui soutient les start-ups et les entreprises innovantes dans le domaine des technologies financières. Cette collaboration offre à Izicap l'opportunité d'accroître sa visibilité et son expansion internationale.
- **BBVA** : Izicap a également établi un partenariat avec BBVA, l'une des principales banques en Espagne et en Amérique latine. Cette collaboration avec BBVA renforce la présence d'Izicap sur le marché espagnol et permet à la société d'offrir ses solutions de fidélisation et de marketing numérique aux commerçants affiliés à BBVA. Le partenariat avec BBVA témoigne de l'engagement d'Izicap à étendre sa portée et à travailler avec des acteurs majeurs de l'industrie financière pour soutenir les petites et moyennes entreprises dans leur croissance.
- **Ingenico** : Izicap a établi un partenariat avec Ingenico, l'un des principaux fournisseurs de solutions de paiement et de services marchands au niveau mondial. Grâce à ce partenariat, Izicap est en mesure d'intégrer sa solution de fidélisation et de marketing numérique aux terminaux de paiement d'Ingenico, offrant ainsi une solution complète aux commerçants utilisant les services d'Ingenico. Cette collaboration renforce la position d'Izicap en tant que fournisseur de solutions de fidélisation et de marketing innovantes, en s'appuyant sur l'expertise et la portée mondiale d'Ingenico dans le secteur des paiements.



FIGURE 1.1 : Partenariats stratégiques

1.3 Organigramme

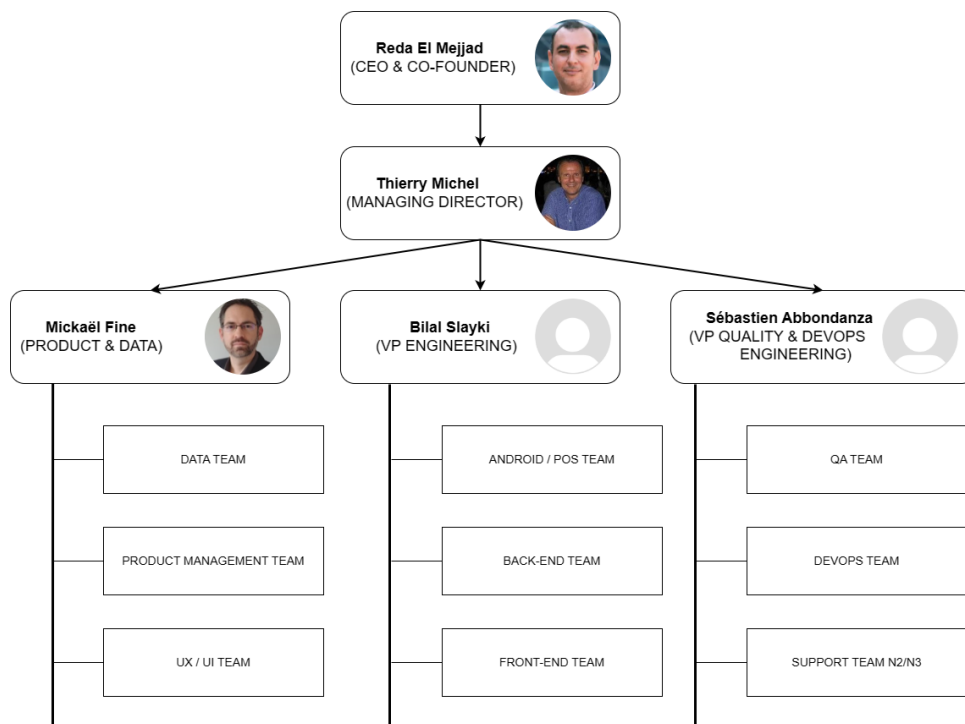


FIGURE 1.2 : Organigramme de Izicap

Au sein de l'équipe de projet, j'ai eu l'opportunité de travailler de manière polyvalente dans différents domaines. J'ai contribué activement au développement du backend en participant à la concep-

tion et à l'implémentation des microservices. J'ai également collaboré avec l'équipe frontend en apportant mon expertise pour l'intégration des microfrontends et l'amélioration de l'expérience utilisateur. De plus, j'ai été impliqué dans la gestion des données en participant à la mise en place de Delta Lake et en contribuant à l'optimisation des flux de données. Cette expérience diversifiée m'a permis d'acquérir une vision globale du projet et de collaborer efficacement avec différentes équipes pour atteindre les objectifs fixés.

1.4 Cadre du projet

Le projet consiste à refondre l'application 'Smart data' qui fait référence au processus d'extraction d'informations précieuses à partir de grandes quantités de données afin de prendre des décisions commerciales éclairées.

La smart data nous permet de :

1. **Collecte des données** : Izicap aide les entreprises à collecter des données clients à partir de différents points de contact tels que les systèmes de point de vente, les programmes de fidélité, les interactions en ligne, etc. Assurez-vous d'intégrer leurs outils de collecte de données dans vos systèmes ou processus existants.
2. **Analyse des données** : La solution de smart data d'Izicap vous permet d'analyser les données collectées pour découvrir des tendances, des modèles et des comportements clients. Utilisez leurs outils d'analyse et leurs algorithmes pour obtenir des informations exploitables à partir des données.
3. **Segmentation des clients** : Segmentez votre base de clients en fonction de leurs préférences, de leur historique d'achat, de leurs caractéristiques démographiques ou d'autres critères pertinents. La solution de smart data d'Izicap peut vous aider à identifier différents segments de clients et à créer des stratégies marketing personnalisées pour chaque segment.
4. **Marketing personnalisé** : Exploitez les informations tirées de la solution de smart data d'Izicap pour créer des campagnes marketing ciblées. Envoyez des offres personnalisées, des promotions ou des recommandations à des segments spécifiques de clients, augmentant ainsi les chances de conversion et de satisfaction client.
5. **Fidélisation de la clientèle** : Utilisez la solution de smart data d'Izicap pour identifier les clients qui risquent de résilier leur abonnement ou de ne plus acheter chez vous. Développez des stratégies de fidélisation en leur offrant des incitations, des programmes de fidélité ou des communications personnalisées afin de les maintenir engagés et fidèles à votre marque.
6. **Suivi des performances** : Surveillez régulièrement les performances de vos campagnes marketing et de vos efforts d'engagement client. La solution d'Izicap peut vous fournir des métriques et des rapports pour évaluer l'efficacité de vos stratégies et apporter des ajustements basés sur les données lorsque nécessaire.

7. **Amélioration continue** : Les solutions de smart data sont les plus efficaces lorsqu'elles sont utilisées de manière itérative. Analysez régulièrement les données, adaptez vos stratégies et peaufinez votre approche en fonction de nouvelles informations et de l'évolution du comportement client.

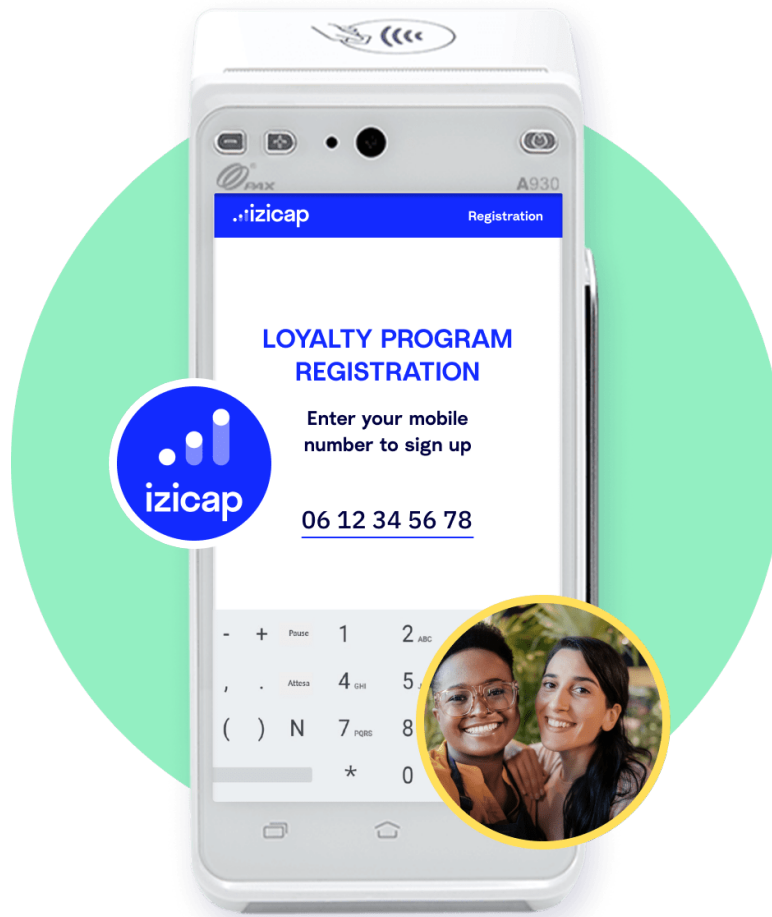


FIGURE 1.3 : Smart data Izicap

1.5 L'architecture du système -Smart Data-

La figure ci-dessous visualise la structure et les composants clés de notre système, ainsi que les interactions entre eux, elle constitue ainsi le fondement de notre système et joue un rôle essentiel dans la fourniture de fonctionnalités et de services à nos utilisateurs.

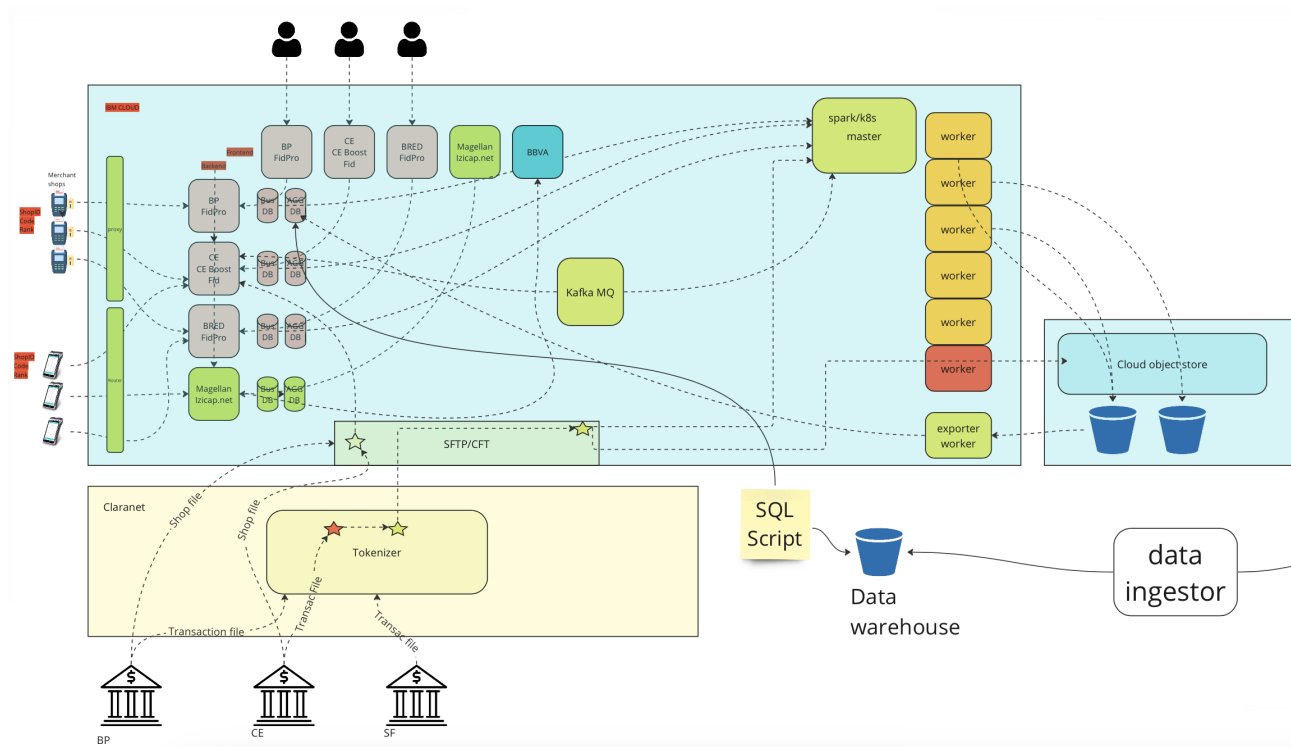


FIGURE 1.4 : Architecture du système actuelle

Ce schéma représente le fonctionnement de l'architecture monolithique actuelle qui présente plusieurs inconvénients qui peuvent entraver l'efficacité, la flexibilité et la maintenabilité du système. Voici une description détaillée des principaux inconvénients :

1. **Complexité et dépendances** : L'architecture monolithique implique que tous les modules et fonctionnalités du système sont regroupés en un seul bloc. Cela crée une forte interdépendance entre les différents composants, rendant la compréhension, la gestion et les mises à jour complexes. Les modifications apportées à une partie du système peuvent avoir des répercussions sur d'autres parties, ce qui rend les tests, les déploiements et les corrections d'erreurs plus difficiles.
2. **Scalabilité limitée** : L'architecture monolithique a souvent des difficultés à s'adapter à une augmentation de la charge ou à une demande croissante. Étant donné que tous les composants sont regroupés dans un seul monolithe, il est difficile de faire évoluer sélectivement une partie spécifique du système. Cela peut entraîner des goulots d'étranglement, des performances réduites et une mauvaise répartition des ressources lorsqu'il s'agit de traiter des volumes de données importants ou de gérer une augmentation du nombre d'utilisateurs.
3. **Déploiements complexes et risques élevés** : Avec une architecture monolithique, les déploiements nécessitent la mise à jour de l'ensemble du système, même pour des modifications mineures. Cela augmente les risques d'erreurs et de régressions, car une seule erreur peut entraîner l'indisponibilité du système tout entier. De plus, les déploiements doivent être soigneusement planifiés et coordonnés, ce qui peut entraîner des temps d'arrêt plus longs et des interruptions de service pour les utilisateurs.

4. **Difficulté de choix technologiques** : Dans une architecture monolithique, les technologies utilisées sont souvent liées et intégrées de manière étroite. Cela peut rendre difficile l'adoption de nouvelles technologies ou l'intégration de composants spécialisés. Les mises à niveau de versions ou les ajouts de nouvelles fonctionnalités peuvent être limités par les choix technologiques initiaux, ce qui peut entraver l'innovation et l'adaptation aux évolutions du marché.
5. **Cohésion et responsabilités** : L'architecture monolithique ne facilite pas une séparation claire des responsabilités et des fonctionnalités. Les différents modules du système sont souvent intimement liés et peuvent partager des fonctionnalités communes. Cela rend difficile l'isolation des problèmes, la maintenance spécifique des modules et la réutilisation de code spécifique à un domaine.

1.6 Problématique et besoin fonctionnel

La problématique qui se pose réside dans l'architecture monolithique actuellement utilisée, laquelle présente des limitations en termes de scalabilité, de flexibilité et de maintenabilité. Les contraintes imposées par cette architecture rendent complexe le déploiement de nouvelles fonctionnalités et entraînent des perturbations potentielles dans l'ensemble du système. De plus, l'utilisation d'une base de données relationnelle traditionnelle, telle que MariaDB, se révèle insuffisante pour traiter efficacement de grands volumes de données transactionnelles, il convient de souligner que l'application Smart Data existe depuis maintenant 10 ans et a été développée en utilisant le langage Groovy. Cette longue période d'existence témoigne de la stabilité et de la maturité de notre système. Cependant, l'utilisation du langage Groovy peut présenter certaines contraintes en termes de maintenabilité et d'évolutivité, notamment lorsqu'il s'agit d'intégrer de nouvelles technologies et de gérer des architectures plus modernes.

Besoin fonctionnel :

Afin de répondre à ces problématiques, différents besoins fonctionnels ont été identifiés :

- **Scalabilité** : Il est nécessaire de disposer d'une architecture qui puisse s'adapter facilement à une croissance du nombre d'utilisateurs, des transactions et des données. Il est essentiel que notre système puisse évoluer harmonieusement et maintenir ses performances, même face à une augmentation significative de la charge de travail.
- **Flexibilité** : Nous devons être en mesure d'introduire de nouvelles fonctionnalités de manière indépendante et de les déployer sans perturber l'ensemble du système. Une approche basée sur des microservices et des microfrontends nous permettra d'atteindre cette flexibilité, en facilitant le développement, les tests et le déploiement isolé de chaque composant.
- **Performances améliorées** : Il est primordial de disposer d'une infrastructure de données capable de gérer efficacement d'importants volumes de données transactionnelles. En remplaçant MariaDB par Delta Lake, nous pourrions tirer parti de fonctionnalités avancées telles que

la gestion des transactions ACID et la compatibilité avec des outils d'analyse performants, ce qui améliorera sensiblement les performances et la fiabilité de notre système.

- **Séparation des responsabilités** : Nous visons une meilleure séparation des responsabilités entre les différents composants de notre système. Les microservices nous permettront de découpler les fonctionnalités, de les attribuer à des équipes spécifiques et de favoriser une gestion du code plus efficace, une maintenance simplifiée et une évolutivité accrue.

1.7 Objectives du stages

Le projet s'inscrit dans le cadre du renouvellement des fonctionnalités de l'application Smart Data, visant à améliorer sa scalabilité, sa flexibilité et ses performances. Dans ce contexte, les objectifs du stage sont les suivants :

1. Étudier et analyser l'architecture actuelle de l'application SMART DATA afin de comprendre les contraintes et les limitations qui entravent sa croissance et son évolution.
2. Proposer et concevoir une stratégie de migration de l'architecture monolithique vers une architecture basée sur des microservices et des microfrontends, permettant ainsi une meilleure modularité et une plus grande flexibilité dans le développement et le déploiement des fonctionnalités.
3. Mettre en œuvre multiples microservices avec Springboot dans le cadre de la nouvelle architecture, en utilisant une technologie appropriée et en veillant à son intégration harmonieuse avec les autres composants du système.
4. Évaluer les avantages et les implications de l'utilisation de Delta Lake en remplacement de la base de données MariaDB, en mettant l'accent sur les performances, la gestion des transactions et l'intégration avec les outils d'analyse.
5. Intégrer Trino dans l'architecture pour permettre l'exécution de requêtes SQL complexes et optimiser le traitement des données, en assurant une collaboration efficace avec l'équipe de développement.
6. Mettre en place une infrastructure de déploiement et de gestion des microservices, en utilisant des outils tels que Kubernetes, Docker et Jenkins, pour faciliter le déploiement, la mise à l'échelle et la gestion des composants.
7. Concevoir et mettre en œuvre des tests unitaires et des tests d'intégration pour assurer la qualité et la fiabilité des nouveaux composants développés dans le cadre de l'architecture basée sur les microservices.
8. Documenter de manière approfondie le processus de migration et les choix technologiques effectués, fournissant des instructions claires pour la maintenance future de l'architecture et la gestion des mises à jour.

9. Collaborer étroitement avec l'équipe existante pour faciliter la transition vers la nouvelle architecture, en offrant un soutien technique, des conseils et des formations sur les nouvelles technologies utilisées.

1.8 Processus de réalisation du projet

Le processus de réalisation du projet s'est déroulé en plusieurs itérations appelées 'sprints', d'une durée généralement fixe de deux semaines. Chaque sprint était axé sur la livraison d'incrément fonctionnels de l'application, permettant ainsi d'obtenir rapidement des résultats tangibles.

Voici les étapes clés du processus de réalisation du projet, basé sur la méthodologie Scrum :

1. **Définition du backlog du produit** : En collaboration avec les parties prenantes et l'équipe de développement, nous avons identifié et priorisé les fonctionnalités à développer et à intégrer dans l'architecture basée sur les microservices.
2. **Planification du sprint** : Au début de chaque sprint, nous avons organisé une réunion de planification pour définir les objectifs spécifiques du sprint, sélectionner les tâches à réaliser et estimer les efforts nécessaires.
3. **Développement itératif** : L'équipe de développement a travaillé de manière itérative sur les tâches assignées, en se concentrant sur la réalisation des fonctionnalités identifiées pour le sprint en cours.
4. **Réunions quotidiennes de stand up** : Chaque jour, l'équipe s'est réunie pour une brève réunion de stand-up afin de partager les progrès, les obstacles éventuels et coordonner les activités.
5. **Revue de sprint** : À la fin de chaque sprint, nous avons organisé une revue de sprint pour présenter les fonctionnalités développées et obtenir des retours des parties prenantes. Cela nous a permis de valider les résultats obtenus et de planifier les prochaines étapes.
6. **Rétrospective de sprint** : Après la revue de sprint, nous avons mené une rétrospective pour évaluer le déroulement du sprint, identifier les points forts et les points à améliorer, et ajuster notre approche en conséquence.
7. **Itérations suivantes** : Le processus de planification, de développement itératif, de revue de sprint et de rétrospective s'est répété pour chaque sprint suivant, permettant ainsi une progression incrémentale vers les objectifs du projet.

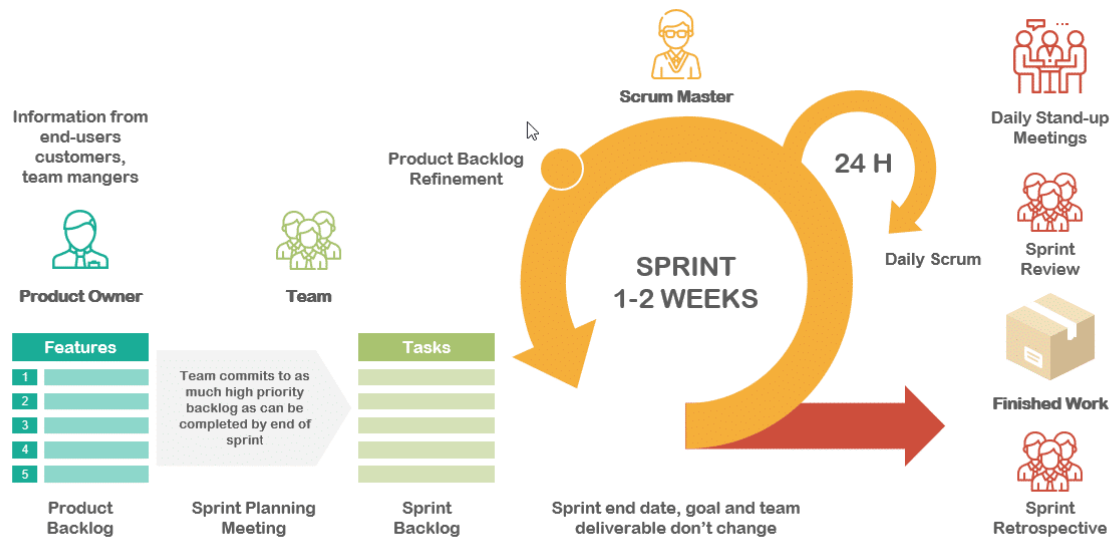


FIGURE 1.5 : Méthodologie Scrum

1.9 Planification du projet

Pour la planification du projet, nous avons utilisé la méthode de modélisation du réseau de dépendance entre les tâches. Cette approche nous a permis de décomposer le travail en différentes tâches structurées et d'établir des relations de dépendance entre elles.

Nous avons également utilisé la technique du diagramme de Gantt pour représenter graphiquement les tâches et les ressources du projet dans le temps. En ligne, nous avons listé les différentes tâches, et en colonne, nous avons défini les jours, les semaines ou les mois. Chaque tâche a été représentée par une barre dont la longueur est proportionnelle à la durée estimée de cette tâche.

Le diagramme de Gantt nous a permis de visualiser la répartition des tâches, leur durée et leur succession. Certaines tâches se sont réalisées en séquence, tandis que d'autres ont pu être réalisées en parallèle, de manière partielle ou totale. Cette représentation claire et visuelle nous a aidés à planifier le projet en déterminant et en organisant les différentes tâches de manière à assurer une gestion efficace du projet.

La figure suivante présente le diagramme de Gantt détaillant la planification du projet, avec les tâches ordonnées dans le temps et leur durée respective. Cela nous a permis de suivre l'avancement du projet, d'identifier les éventuels retards et de prendre les mesures appropriées pour les résoudre.

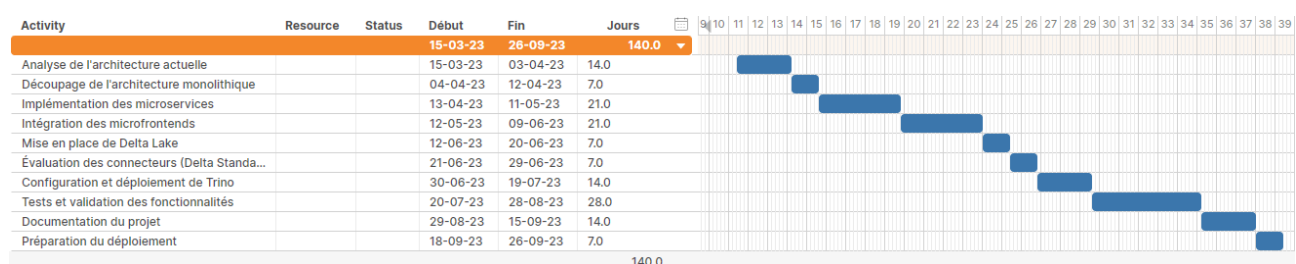


FIGURE 1.6 : Diagramme de Gantt

Conclusion

En conclusion de cette section, nous avons présenté la planification du projet de renouvellement des fonctionnalités de l'application SMART DATA. Nous avons élaboré un planning basé sur la modélisation du réseau de dépendance entre les tâches, en utilisant la technique du diagramme de Gantt. Ce diagramme nous a permis de visualiser les différentes activités, leur durée estimée et les ressources nécessaires.

Au cours de la planification, nous avons identifié les principales activités du projet, telles que l'analyse de l'architecture actuelle, le découpage de l'architecture monolithique, l'implémentation des microservices, l'intégration des microfrontends, la mise en place de Delta Lake, l'évaluation des connecteurs, la configuration et le déploiement de Trino, les tests et la validation des fonctionnalités, la documentation du projet, la préparation du déploiement, ainsi que la formation et la sensibilisation des utilisateurs.

Nous avons également pris en compte la durée totale du projet, qui est estimée à 5 mois, et avons ajusté les durées des activités en conséquence. Cela nous a permis d'avoir une vision plus précise de la planification temporelle du projet.

Informatique décisionnelle et Choix de solutions

Introduction

L'informatique décisionnelle, également connue sous le terme de Business Intelligence (BI), englobe les processus, les technologies et les outils utilisés pour collecter, stocker, analyser et présenter les données dans le but de soutenir la prise de décision et d'aider les entreprises à obtenir des informations exploitables. L'informatique décisionnelle implique la transformation des données brutes en informations significatives et en connaissances exploitables pour les décideurs. On va voir dans ce chapitre pourquoi on a décidé d'opter pour Delta Lake au lieu de ses contre-parts ainsi que les microservice et les microfrontends.

2.1 Différence entre un data warehouse, un data lake, un datalakehouse et un delta lake

- **Data Warehouse** : Un data warehouse est une base de données centralisée qui est spécifiquement conçue pour le reporting et l'analyse. Il stocke les données structurées provenant de différentes sources, les organise selon un modèle de données prédéfini et les optimise pour des requêtes analytiques. Les données dans un data warehouse sont généralement cohérentes, intégrées et historisées. Cependant, la construction et la maintenance d'un data warehouse peuvent être complexes et coûteuses.
- **Data Lake** : Un data lake est un référentiel de données centralisé qui stocke de grandes quantités de données brutes, structurées et non structurées. Contrairement au data warehouse, le data lake ne nécessite pas une modélisation préalable des données. Il offre une grande flexibilité et évolutivité pour stocker des données hétérogènes. Cependant, l'intégration et la qualité des données peuvent être des défis dans un data lake.

- **Datalakehouse** : Le datalakehouse est une architecture émergente qui combine les avantages du data warehouse et du data lake. Il permet de stocker et de traiter à la fois des données brutes et des données structurées dans un environnement centralisé. Cette approche hybride offre la flexibilité d'un data lake et la capacité d'analyse d'un data warehouse. Cependant, la mise en place d'un datalakehouse peut nécessiter des efforts supplémentaires pour garantir la qualité des données et l'efficacité des requêtes.
- **Delta Lake** : Delta Lake est une technologie qui s'intègre aux data lakes existants pour fournir des fonctionnalités supplémentaires, telles que la gestion des transactions ACID (Atomicité, Cohérence, Isolation, Durabilité), la gestion des mises à jour incrémentielles et la garantie de la cohérence des données. Delta Lake est construit sur Apache Parquet et Apache Arrow, ce qui permet d'accélérer les requêtes analytiques et d'améliorer les performances globales. Cependant, l'utilisation de Delta Lake peut nécessiter des compétences techniques supplémentaires et peut avoir un impact sur la complexité de l'architecture de données.

2.1.1 Data Warehouse

Avantages :

1. Données cohérentes et intégrées
2. Modélisation préalable des données pour une analyse optimisée
3. Hautes performances pour les requêtes analytiques

Inconvénients :

1. Coût élevé de construction et de maintenance
2. Complexité de la modélisation des données
3. Limitations pour l'intégration de données non structurées

2.1.2 Data Lake

Avantages :

1. Stockage économique de grandes quantités de données
2. Flexibilité pour intégrer des données brutes et non structurées
3. Capacité à traiter des données de différentes sources

Inconvénients :

1. Difficulté à maintenir la qualité des données et la gouvernance
2. Besoin d'outils avancés pour l'analyse et le traitement des données
3. Requiert des compétences techniques pour l'exploitation efficace des données

2.1.3 Datalakehouse

Avanatges :

1. Combinaison des avantages du data warehouse et du data lake
2. Flexibilité pour stocker et analyser des données brutes et structurées
3. Possibilité d'évoluer en fonction des besoins évolutifs

Inconvénients :

1. Nécessite des efforts supplémentaires pour la qualité des données
2. Complexité accrue de l'architecture de données
3. Besoin de compétences techniques pour la mise en place et la gestion

2.1.4 Delta Lake (Datalakehouse)

Avanatges :

1. Gestion des transactions ACID pour une cohérence des données
2. Prise en charge des mises à jour incrémentielles et du traitement des flux de données
3. Hautes performances pour les requêtes analytiques

Inconvénients :

1. Nécessite des compétences techniques spécifiques
2. Impact sur la complexité de l'architecture de données existante
3. Peut nécessiter des adaptations pour une intégration transparente avec les outils existants

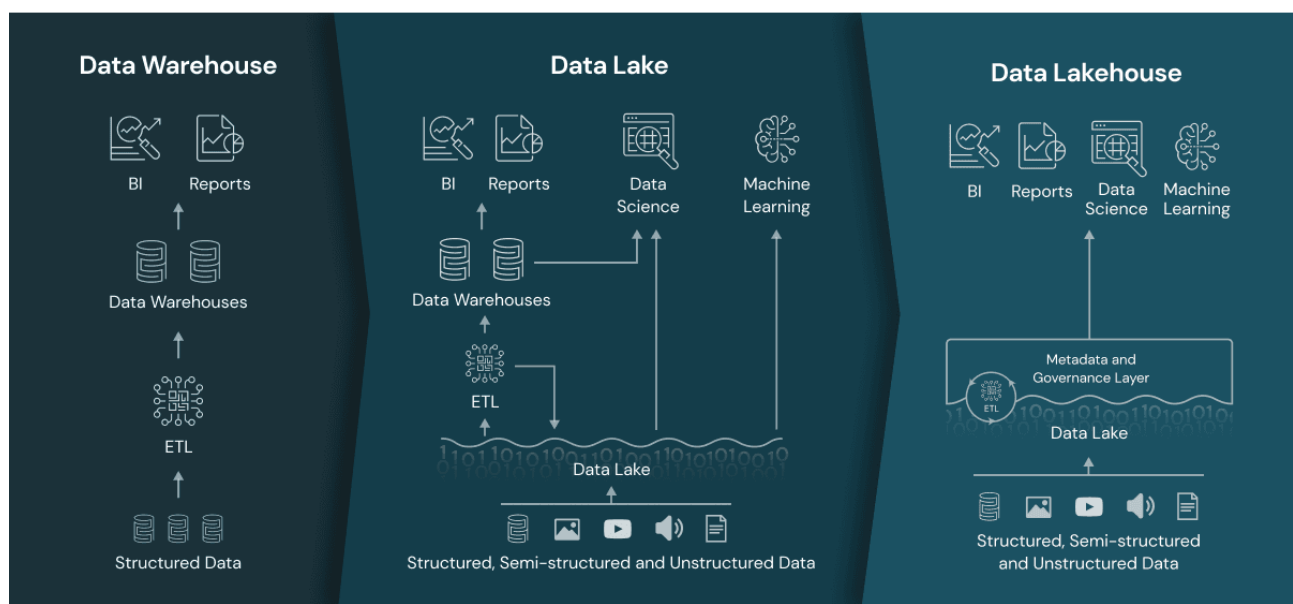


FIGURE 2.1 : Data Warehouse vs Data Lake vs Data Lakehouse

2.2 Différence entre une architecture microservices et une architecture monolithique

- **Monolithe** : un monolithe est une application qui est conçue comme une entité unique et indivisible. Toutes les fonctionnalités de l'application sont regroupées dans un seul code base, partageant les mêmes ressources, bases de données et déploiements. Dans une architecture monolithique, il n'y a pas de découpage clair des fonctionnalités en services indépendants. Toute modification ou évolution de l'application nécessite des changements au niveau global.
- **Microservice** : Un microservice est une approche architecturale dans laquelle une application est construite comme un ensemble de services indépendants et autonomes, chacun se concentrant sur une fonction spécifique de l'application. Chaque microservice est développé, déployé et géré de manière indépendante, ce qui permet une évolutivité, une flexibilité et une maintenance plus faciles. Les microservices communiquent entre eux via des interfaces bien définies, généralement basées sur des API REST ou des messages.

2.2.1 Avantages et Inconvénients de l'architecture monolithique

Avantages :

1. **Simplicité de développement initial** : L'approche monolithique permet de développer rapidement une application en regroupant toutes les fonctionnalités dans un seul code base. Cela facilite la gestion des dépendances et la coordination des différentes parties de l'application.
2. **Moins de complexité opérationnelle** : Avec une architecture monolithique, il y a moins de composants et de services à gérer, ce qui simplifie les opérations de déploiement, de surveillance et de gestion. Tout est regroupé au sein d'un seul déploiement, ce qui peut être plus facile à gérer pour les équipes opérationnelles.
3. **Communications internes plus rapides** : Dans une architecture monolithique, les communications entre les différentes parties de l'application sont plus rapides car elles se font généralement via des appels de méthode internes. Cela peut être bénéfique en termes de performances et de latence réduite.

Inconvénients :

1. **Difficulté à faire évoluer et à maintenir** : Avec une architecture monolithique, les évolutions et les mises à jour peuvent être plus complexes, car chaque changement doit être effectué sur l'ensemble de l'application. Cela peut rendre le processus de développement plus lent et entraîner des risques d'erreurs lors des déploiements.
2. **Rigidité technologique** : Une architecture monolithique peut entraîner une rigidité technologique, car toutes les parties de l'application doivent utiliser les mêmes technologies et langages de programmation. Cela peut limiter les possibilités d'adopter de nouvelles technologies ou de faire évoluer des parties spécifiques de l'application de manière indépendante.

3. **Difficulté à isoler les problèmes** : En cas de problème ou de bug, il peut être plus difficile de les isoler et de les résoudre dans une architecture monolithique. Étant donné que toutes les fonctionnalités sont regroupées dans un seul code base, il peut être complexe de localiser l'origine exacte du problème.
4. **Moins de flexibilité en termes d'évolutivité** : L'architecture monolithique peut poser des défis en termes d'évolutivité. Si une partie de l'application nécessite plus de ressources pour gérer une charge élevée, il peut être difficile d'ajuster cette partie spécifique sans augmenter les ressources globales de l'application.

2.2.2 Avantages et Inconvénients de l'architecture microservices

Avantages :

1. **Scalabilité et évolutivité** : Les microservices permettent de découper l'application en plusieurs services autonomes et indépendants, ce qui facilite la scalabilité horizontale. Chaque microservice peut être déployé, mis à l'échelle et mis à jour indépendamment, ce qui permet de gérer efficacement les variations de charge et de garantir une évolutivité facile en fonction des besoins de l'entreprise.
2. **Flexibilité technologique** : Les microservices offrent la possibilité d'utiliser différentes technologies et langages de programmation pour chaque service. Dans notre cas, l'utilisation de Spring Boot nous permet de profiter de son écosystème riche et de ses fonctionnalités avancées pour le développement rapide d'applications. Cela permet également d'adopter des technologies spécifiques en fonction des besoins de chaque microservice, favorisant ainsi la flexibilité technologique.
3. **Indépendance et autonomie** : Chaque microservice est conçu pour fonctionner de manière autonome, ce qui permet une meilleure isolation des fonctionnalités et des responsabilités. Cela facilite la maintenance, le test et le déploiement des services de manière indépendante, réduisant ainsi les risques d'impact sur l'ensemble du système en cas de modifications ou de problèmes.
4. **Développement et déploiement rapides** : Les microservices permettent une approche de développement agile en favorisant des cycles de développement plus courts. Les équipes peuvent se concentrer sur des fonctionnalités spécifiques et les développer de manière indépendante, ce qui accélère le développement global du système. De plus, les microservices peuvent être déployés de manière continue grâce à l'utilisation de techniques de déploiement automatisées, facilitant ainsi les mises à jour fréquentes et rapides.
5. **Facilité de maintenance et de débogage** : En raison de leur nature modulaire, les microservices facilitent la maintenance et le débogage du système. En cas de problème ou d'erreur, il est plus facile d'identifier le service spécifique concerné et de résoudre le problème sans impacter l'ensemble du système.

Inconvénients :

1. **Complexité de la gestion des communications** : Les microservices impliquent une communication entre différents services, généralement via des API REST ou des messages. La gestion de ces communications peut devenir complexe, en particulier lorsque de nombreux services sont impliqués. Des problèmes tels que la latence, la cohérence des données et la gestion des erreurs peuvent se poser.
2. **Surcharge de développement initial** : Le développement de microservices nécessite un effort supplémentaire pour découper correctement les fonctionnalités, définir les interfaces et mettre en place une infrastructure appropriée pour le déploiement et la communication des services. Cela peut augmenter la charge de travail initiale et nécessiter des compétences spécifiques en matière d'architecture distribuée.
3. **Gestion de la cohérence des données** : Avec des microservices, chaque service peut avoir sa propre base de données ou son propre stockage de données. Cela peut rendre la gestion de la cohérence des données plus complexe, en particulier lorsqu'il y a des mises à jour simultanées impliquant plusieurs services. Des techniques telles que les transactions distribuées ou les événements asynchrones peuvent être nécessaires pour maintenir la cohérence des données.
4. **Déploiement et gestion de plusieurs services** : Avec les microservices, il y a un nombre accru de services à déployer, gérer et surveiller. Cela peut nécessiter des compétences supplémentaires en matière d'automatisation des déploiements, de gestion des conteneurs ou de gestion des clusters. Le suivi des performances et du comportement de chaque service peut également devenir plus complexe.
5. **Coût de l'infrastructure** : Les microservices peuvent nécessiter une infrastructure plus complexe et des ressources supplémentaires pour fonctionner efficacement. Chaque service doit être déployé et exécuté indépendamment, ce qui peut entraîner une augmentation des coûts liés aux ressources informatiques et à la gestion de l'infrastructure.

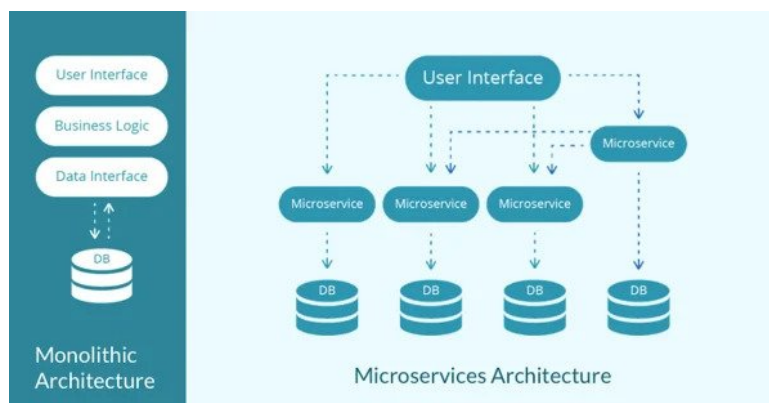


FIGURE 2.2 : Architecture monolithique vs microservice

2.3 Différence entre une architecture microfrontends et une architecture monolithique frontend

- **Monolithe frontend** : Un frontend monolithique est une approche architecturale où l'application frontend est développée comme une seule entité, généralement en utilisant un framework spécifique tel qu'AngularJS. Toutes les fonctionnalités, les vues et les logiques de l'interface utilisateur sont regroupées dans un seul code base.
- **Microfrontends** : Les microfrontends sont une approche architecturale où une application frontend est divisée en plusieurs micro-applications indépendantes, chacune étant responsable d'une partie spécifique de l'interface utilisateur. Chaque micro-application peut être développée, déployée et évoluée de manière autonome, utilisant différents frameworks, langages et technologies.

2.3.1 Avantages et Inconvénients du monolithe frontend

Avantages :

1. **Simplicité de développement initial** : L'architecture monolithique avec AngularJS offre une approche simple pour le développement initial de l'application frontend. Toutes les fonctionnalités sont regroupées dans un seul code base, ce qui facilite la coordination et la gestion du développement.
2. **Facilité de communication entre les composants** : Dans une architecture monolithique, les composants AngularJS peuvent communiquer entre eux facilement via le système de directives et de services d'AngularJS. Cela permet une communication rapide et efficace entre les différentes parties de l'application.
3. **Interopérabilité des fonctionnalités** : Étant donné que toutes les fonctionnalités sont développées en utilisant AngularJS, il est plus facile de partager des fonctionnalités et des modules entre les différentes parties de l'application. Cela favorise la réutilisation du code et simplifie la maintenance.

Inconvénients :

1. **Difficulté à maintenir et à faire évoluer** : À mesure que l'application frontend devient plus complexe, la maintenance et l'évolution de l'architecture monolithique avec AngularJS peuvent devenir difficiles. Les modifications apportées à une partie de l'application peuvent avoir des répercussions sur l'ensemble du code base, ce qui rend le processus de développement plus lent et risqué.
2. **Limitations de performance** : Dans une architecture monolithique, toutes les fonctionnalités sont chargées en même temps, ce qui peut entraîner des problèmes de performance si l'application devient volumineuse. Les temps de chargement peuvent être plus longs et l'application peut être moins réactive pour l'utilisateur.

3. **Flexibilité limitée** : L'architecture monolithique avec AngularJS peut limiter la flexibilité technologique. Étant donné que toutes les parties de l'application sont développées en utilisant AngularJS, il peut être difficile d'introduire de nouvelles technologies ou de faire évoluer certaines parties spécifiques de l'application de manière indépendante.

2.3.2 Avantages et Inconvénients de l'architecture microfrontends

Avantages :

1. **Indépendance des équipes de développement** : Chaque micro-application peut être développée par une équipe distincte, ce qui favorise une plus grande autonomie et une meilleure collaboration entre les équipes de développement. Chaque équipe peut choisir les technologies qui conviennent le mieux à ses besoins.
2. **Évolutivité et facilité de maintenance** : L'architecture microfrontends permet de faire évoluer et de maintenir les différentes parties de l'application de manière indépendante. Les modifications apportées à une micro-application n'ont pas d'impact sur les autres, ce qui facilite la maintenance et permet de déployer rapidement de nouvelles fonctionnalités.
3. **Flexibilité technologique** : Chaque micro-application peut utiliser la technologie, le framework ou le langage de programmation qui convient le mieux à son domaine spécifique. Cela permet d'introduire de nouvelles technologies et d'exploiter les avantages des derniers développements dans le domaine de l'ingénierie logicielle.

Inconvénients :

1. **Complexité accrue** : L'architecture microfrontends introduit une certaine complexité dans le développement et le déploiement de l'application. La gestion des interactions et de la communication entre les différentes micro-applications peut nécessiter une planification et une coordination supplémentaires.
2. **Coût de développement initial** : Le développement d'une architecture microfrontends peut nécessiter un investissement initial plus important en termes de ressources et de temps. Le développement de plusieurs micro-applications distinctes et la mise en place de l'infrastructure nécessaire peuvent être plus coûteux que le développement d'une application monolithique.
3. **Surcharge réseau** : L'utilisation d'une architecture microfrontends peut entraîner une surcharge réseau plus importante, car chaque micro-application nécessite des requêtes et des chargements de ressources distincts. Cela peut avoir un impact sur les performances de l'application et nécessiter une gestion efficace du réseau.

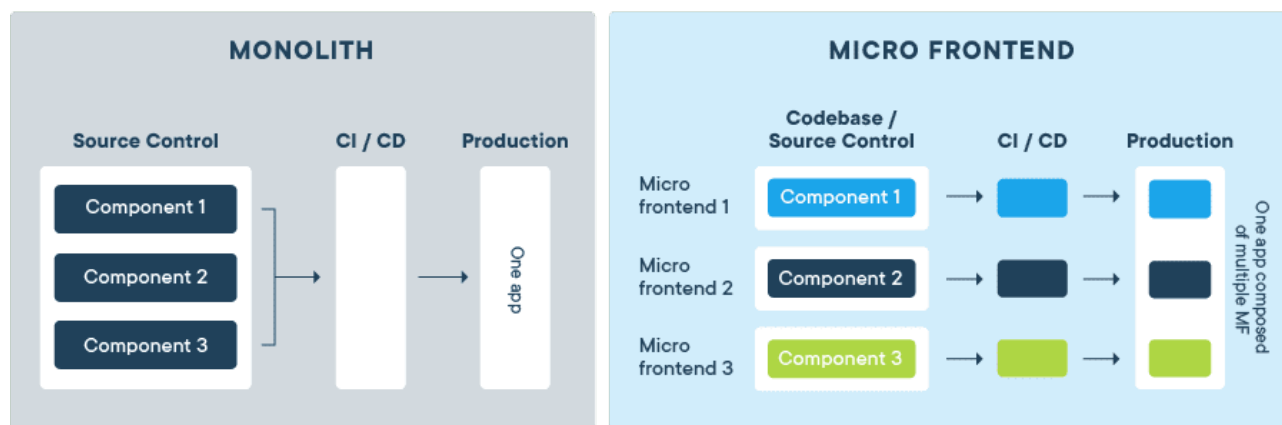


FIGURE 2.3 : Architecture monolithique frontend vs microfrontends

2.4 Cheminement de la solution

2.4.1 Partie Data

Dans le cadre de notre solution, nous avons opté de remplacer l'exporter worker existant et les bases de données MariaDB par de remplacer l'exporter worker existant et les bases de données MariaDB par l'utilisation d'une architecture de type datalakehouse. Une datalakehouse est une approche hybride qui combine les avantages d'un data warehouse et d'un data lake, offrant ainsi une solution plus flexible et évolutive pour la gestion des données.

2.4.2 Partie Backend

Pour la mise en œuvre des microservices, nous avons opté d'utiliser Spring Boot, un framework Java populaire pour le développement d'applications. En utilisant Spring Boot, nous pouvons créer des microservices autonomes, indépendants les uns des autres, qui peuvent être développés, déployés et scalés individuellement. Spring Boot fournit également des fonctionnalités telles que la gestion de la persistance des données, la sécurité et la création d'API REST, ce qui facilite le développement des microservices.

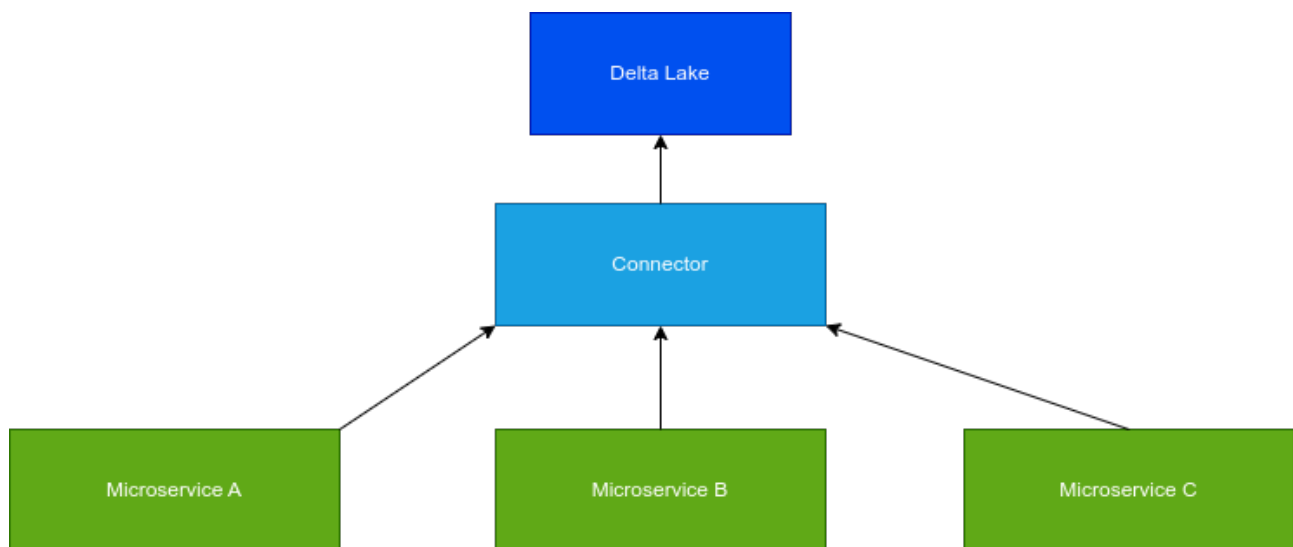


FIGURE 2.4 : Delta lake connectés à des microservices

2.4.3 Partie Frontend

Pour la mise en œuvre des microfrontends, Nous avons choisie d'utiliser principalement React pour les microfrontends, l'équipe de développement peut bénéficier de l'écosystème riche et mature de React, ainsi que de sa popularité croissante dans l'industrie du développement frontend. Cependant, il est également mentionné qu'Angular peut être utilisé ultérieurement si nécessaire, offrant ainsi une flexibilité supplémentaire dans le choix des technologies.

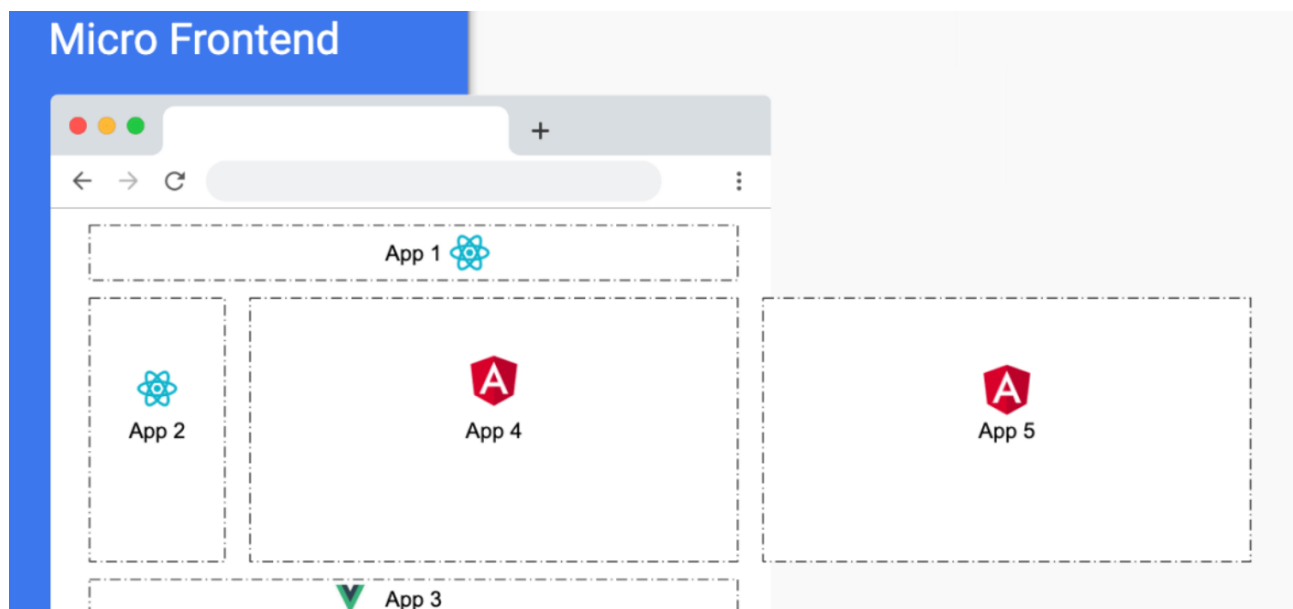


FIGURE 2.5 : Schéma des microfrontends

Conclusion

Lors de l'évaluation des différentes architectures de données pour Izicap, il est important de comprendre les besoins spécifiques liés à la gestion des fichiers bancaires, des reçus de transactions et des

opérations d'agrégation.

Un data warehouse aurait pu être une option envisageable, offrant des structures de données organisées et optimisées pour les requêtes analytiques. Cependant, le principal inconvénient d'un data warehouse réside dans sa nature statique, qui nécessite une modélisation préalable des données et une transformation rigide avant leur chargement. Cela peut poser des défis lors de l'intégration de nouveaux types de fichiers ou de l'évolution des besoins en matière d'agrégation.

D'autre part, un data lake présente des avantages en termes de stockage économique et de flexibilité pour intégrer des données brutes et non structurées. Cependant, il peut être plus complexe de maintenir la qualité des données et la gouvernance, et des compétences techniques spécifiques sont nécessaires pour exploiter efficacement les données du data lake.

Donc, Delta Lake a été privilégié en raison de sa capacité à répondre aux besoins spécifiques d'Izicap en matière de gestion des fichiers bancaires, des reçus de transactions et des opérations d'agrégation. Il offre la flexibilité et la performance nécessaires tout en maintenant l'intégrité des données, ce qui en fait un choix solide pour l'architecture de données de l'entreprise.

En adoptant une approche basée sur les microservices, Izicap peut améliorer la flexibilité, la scalabilité et la maintenance de son application frontend. Cela permettra à l'entreprise de mieux répondre aux besoins changeants de ses utilisateurs, de faciliter la collaboration entre les équipes de développement et d'adopter des technologies innovantes pour offrir une expérience utilisateur optimale.

Les microfrontends offrent plusieurs avantages pour Izicap. Tout d'abord, la modularité inhérente aux microfrontends permet de développer, déployer et maintenir différentes parties de l'interface utilisateur de manière indépendante. Cela favorise la collaboration entre les équipes de développement et permet d'évoluer rapidement et efficacement. Chaque micro-application peut être développée en utilisant le framework et les technologies les plus adaptés à ses besoins spécifiques, ce qui offre une flexibilité technologique essentielle pour Izicap.

En revanche, l'approche monolithique présente des limitations en termes de scalabilité et de flexibilité technologique. Les modifications apportées à une partie de l'application peuvent avoir un impact sur l'ensemble du système, ce qui rend les évolutions plus complexes et risquées. De plus, l'introduction de nouvelles technologies ou frameworks peut être difficile dans une architecture monolithique, ce qui limite les possibilités d'innovation et de modernisation.

Technologies Utilisées

Introduction

Dans ce chapitre, nous examinerons en détail les technologies clés qui sont utilisées dans notre solution pour fournir des fonctionnalités avancées et répondre aux besoins spécifiques de notre projet. Les trois principales technologies que nous aborderons sont Delta Lake, Trino et les Microfrontends.

3.1 Delta Lake

Delta Lake est une technologie de gestion des données qui permet de stocker, gérer et analyser des volumes massifs de données de manière efficace et fiable. Il repose sur une architecture basée sur des fichiers parquet et offre des fonctionnalités avancées telles que la gestion des transactions ACID (Atomicité, Cohérence, Isolation, Durabilité) et la compatibilité avec des outils d'analyse populaires. Delta Lake garantit également l'intégrité des données, la cohérence des requêtes et la prise en charge de la réplication et de la récupération en cas de défaillance.

Le concept de 'lakehouse' est rendu possible grâce à Delta Lake. Il s'agit d'une architecture de données qui combine les avantages des entrepôts de données et des lacs de données, en offrant une approche unique et cohérente pour la gestion des données. Les données sont stockées au format Parquet dans le lac de données, permettant ainsi un traitement continu et par lots.

- **Permet une architecture Lakehouse :** Delta Lake permet une architecture de données continue et simplifiée qui permet aux organisations de gérer et de traiter d'énormes volumes de données en continu et par lots sans les tracas de gestion et d'exploitation liés à la gestion séparée du streaming, des data warehouses et des lacs de données.
- **Permet une gestion intelligente des données pour les lacs de données :** Delta Lake offre une gestion efficace et évolutive des métadonnées, qui fournit des informations sur les volumes de données massifs dans les lacs de données. Grâce à ces informations, les tâches de gouvernance et de gestion des données se déroulent plus efficacement.

- **Application du schéma pour une meilleure qualité des données :** Étant donné que les lacs de données n'ont pas de schéma défini, il devient facile pour les données mauvaises/incompatibles d'entrer dans les systèmes de données. La qualité des données est améliorée grâce à la validation automatique du schéma, qui valide la compatibilité DataFrame et table avant les écritures.
- **Permet la transaction ACID :** La plupart des architectures de données organisationnelles impliquent de nombreux mouvements ETL et ELT entrant et sortant du stockage de données, ce qui l'ouvre à plus de complexité et d'échecs aux points d'entrée des nœuds. Delta Lake garantit la durabilité et la persistance des données pendant l'ETL et d'autres opérations de données. Delta Lake capture toutes les modifications apportées aux données pendant les opérations de données dans un journal des transactions, garantissant ainsi l'intégrité et la fiabilité des données pendant les opérations de données.

3.2 Principaux avantages et caractéristiques de Delta Lake

Avec Delta Lake, les données sont stockées dans un format optimisé, tel que Parquet, dans un lac de données. Ce format facilite le traitement efficace des requêtes, quel que soit le mode d'accès aux données, qu'il s'agisse d'un traitement streaming ou par batch.

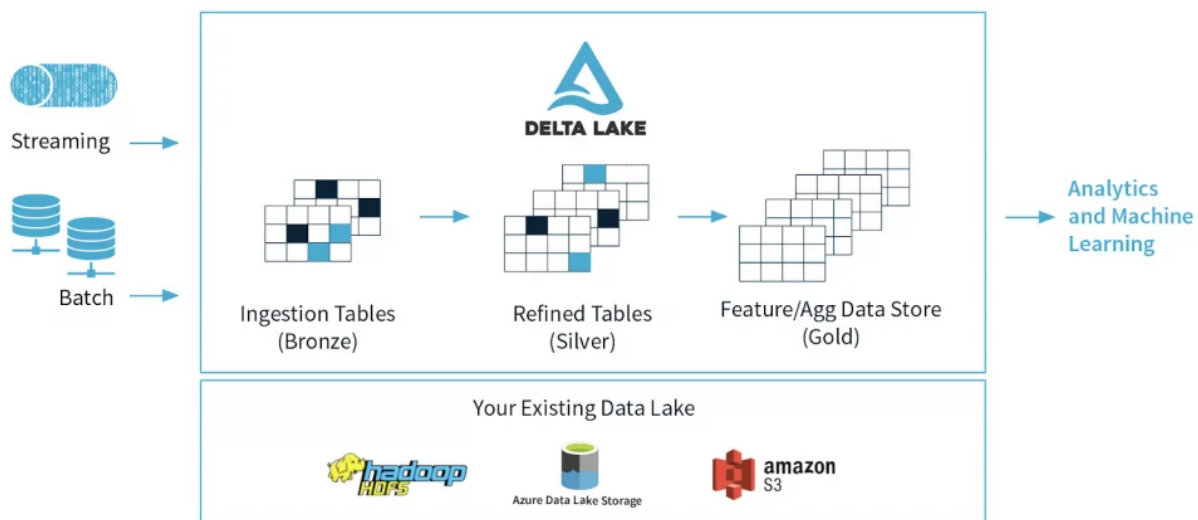


FIGURE 3.1 : Architecture multi-sauts de Delta Lake

- **Pistes d'audit et historique :** Dans Delta Lake, chaque écriture existe en tant que transaction et est enregistrée en série dans un journal des transactions. Par conséquent, toutes les modifications ou validations apportées au journal des transactions sont enregistrées, laissant une trace complète à utiliser dans les audits historiques, la gestion des versions ou à des fins de voyage dans le temps. Cette fonctionnalité de Delta Lake permet de garantir l'intégrité et la fiabilité des données pour les opérations de données d'entreprise.

- **Voyager dans le temps et versionner les données** : Étant donné que chaque écriture crée une nouvelle version et stocke l'ancienne version dans le journal des transactions, les utilisateurs peuvent afficher/restaurer les anciennes versions de données en fournissant l'horodatage ou le numéro de version d'une table ou d'un répertoire existant à l'API de lecture Sparks. À l'aide du numéro de version fourni, Delta Lake construit ensuite un instantané complet de la version avec les informations fournies par le journal des transactions. Les retours en arrière et la gestion des versions jouent un rôle essentiel dans l'expérimentation de l'apprentissage automatique, où les scientifiques des données modifient de manière itérative les hyperparamètres pour former des modèles et peuvent revenir aux modifications si nécessaire.
- **Unifie le traitement par lots et par flux** : Chaque table d'un Delta Lake est un puits de lot et de flux. Avec le streaming structuré Sparks, les organisations peuvent diffuser et traiter efficacement les données de streaming. De plus, grâce à la gestion efficace des métadonnées, à la facilité d'évolutivité et à la qualité ACID de chaque transaction, l'analyse en temps quasi réel devient possible sans utiliser une architecture de données à deux niveaux plus compliquée.
- **Gestion efficace et évolutive des métadonnées** : Delta Lakes stocke les informations de métadonnées dans le journal des transactions et exploite la puissance de traitement distribuée de Spark pour traiter rapidement, lire et gérer efficacement de gros volumes de métadonnées de données, améliorant ainsi la gouvernance des données.
- **transactions ACID** : Delta Lakes garantit que les utilisateurs voient toujours une vue de données cohérente dans une table ou un répertoire. Il garantit cela en capturant chaque modification effectuée dans un journal de transactions et en l'isolant au niveau d'isolation le plus fort, le niveau sérialisable. Au niveau sérialisable, chaque opération existante a et suit une séquence en série qui, lorsqu'elle est exécutée une par une, fournit le même résultat que celui indiqué dans le tableau.
- **Opérations du langage de manipulation de données** : Delta Lakes prend en charge les opérations DML telles que les mises à jour, les suppressions et les fusions, qui jouent un rôle dans les opérations de données complexes telles que la capture de données de modification (CDC), les upserts en continu et la dimension à évolution lente (SCD). Des opérations comme CDC assurent la synchronisation des données dans tous les systèmes de données et minimisent le temps et les ressources consacrés aux opérations ELT. Par exemple, en utilisant le CDC, au lieu d'ETL toutes les données disponibles, seules les données récemment mises à jour depuis la dernière opération subissent une transformation.
- **Schema Enforcement** : Delta Lakes effectue une validation automatique du schéma en vérifiant un ensemble de règles pour déterminer la compatibilité d'une écriture d'un DataFrame vers une table. L'une de ces règles est l'existence de toutes les colonnes DataFrame dans la table cible. Une occurrence d'une colonne supplémentaire ou manquante dans le DataFrame génère une erreur d'exception. Une autre règle est que le DataFrame et la table cible doivent contenir les mêmes types de colonnes, ce qui, sinon, déclenchera une exception. Delta Lake utilise également DDL (Data Definition Language) pour ajouter explicitement de nouvelles colonnes. Cette

fonctionnalité de lac de données permet d'éviter l'ingestion de données incorrectes, garantissant ainsi une qualité élevée des données.

- **Compatibilité avec l'API de Spark :** Delta Lake est basé sur Apache Spark et est entièrement compatible avec l'API Spark, qui permet de créer des pipelines de données volumineuses efficaces et fiables.
- **Flexibilité et intégration :** Delta Lake est une couche de stockage open source et utilise le format Parquet pour stocker des fichiers de données, ce qui favorise le partage de données et facilite l'intégration avec d'autres technologies et stimule l'innovation.

3.3 Trino

Trino, anciennement connu sous le nom de Presto, est un moteur de requêtes SQL distribué et open-source. Il est conçu pour exécuter des requêtes interactives et analytiques à grande échelle sur des données hétérogènes et distribuées. Trino offre une grande polyvalence en permettant l'accès à différents types de sources de données, qu'il s'agisse de bases de données relationnelles, de systèmes de fichiers, de sources de données en temps réel ou de services de stockage cloud. Grâce à sa conception distribuée, Trino permet des performances élevées et une scalabilité horizontale, ce qui en fait un outil essentiel pour l'analyse des données dans notre solution.

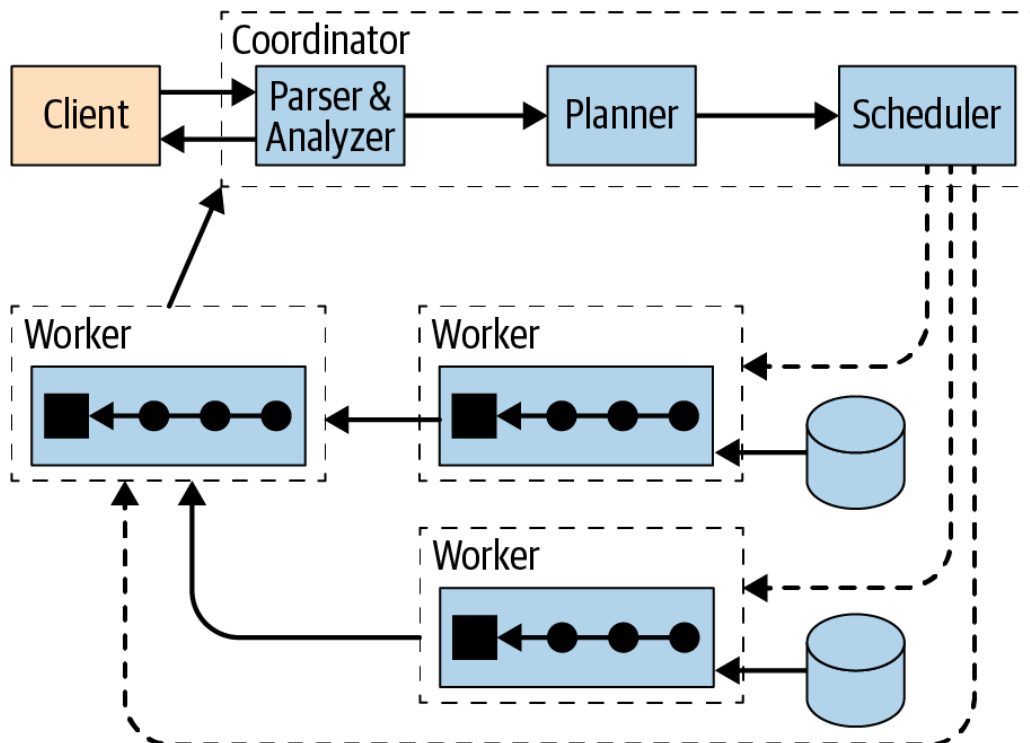


FIGURE 3.2 : Vue d'ensemble de l'architecture Trino avec le coordinateur et les workers

1. Un coordinateur est un serveur Trino qui gère les requêtes entrantes et gère les workers pour exécuter les requêtes.

2. Un worker est un serveur Trino responsable de l'exécution des tâches et du traitement des données.
3. Le service de découverte s'exécute généralement sur le coordinateur et permet aux workers de s'inscrire pour participer au cluster.
4. Toutes les communications et tous les transferts de données entre les clients, le coordinateur et les workers utilisent des interactions basées sur REST sur HTTP/HTTPS.

3.4 Springboot

Spring Boot est un framework open-source pour le développement d'applications Java. Il fournit une approche simplifiée et conventionnelle pour créer des applications Java autonomes, prêtes à être déployées, sans nécessiter une configuration complexe.

L'un des principaux avantages de Spring Boot est sa capacité à réduire la configuration boilerplate et à simplifier le développement d'applications en fournissant des définitions de configuration par défaut intelligentes et en automatisant de nombreux aspects du développement. Il intègre également un serveur d'applications embarqué, ce qui facilite le déploiement et l'exécution de l'application sans avoir besoin d'un serveur d'applications externe.

Spring Boot suit le paradigme de la programmation orientée annotation, où les annotations sont utilisées pour configurer et orchestrer les différentes parties de l'application. Il offre une large gamme de fonctionnalités, telles que l'injection de dépendances, la configuration externe, la gestion des erreurs, la sécurité, l'accès aux données, etc. Ces fonctionnalités sont regroupées dans des starters, qui sont des dépendances prédéfinies facilitant l'ajout de fonctionnalités spécifiques à l'application.

Grâce à son approche simplifiée, Spring Boot permet aux développeurs de se concentrer davantage sur la logique métier de leur application plutôt que sur des tâches de configuration fastidieuses. Il favorise également les bonnes pratiques de développement, telles que la séparation des préoccupations et la modularité, ce qui rend les applications plus maintenables et évolutives.

3.5 Keycloak

Keycloak est une solution open-source de gestion des identités et des accès (Identity and Access Management) développée par Red Hat. Il fournit des fonctionnalités complètes pour la gestion des utilisateurs, l'authentification, l'autorisation et la sécurisation des applications.

Keycloak permet de centraliser et de simplifier la gestion des identités au sein d'une infrastructure informatique. Il offre des fonctionnalités telles que l'inscription des utilisateurs, l'authentification à plusieurs facteurs, la gestion des rôles et des autorisations, la gestion des sessions, ainsi que l'intégration avec des protocoles d'authentification et d'autorisation courants tels que OAuth 2.0 et OpenID Connect.

Keycloak offre un ensemble de fonctionnalités pour gérer les rôles, les administrateurs, les utilisateurs et les mots de passe. Voici comment Keycloak aborde ces aspects :

1. Keycloak permet de définir des rôles au niveau du royaume (realm) ou au niveau de l'application. Les rôles peuvent être créés et attribués aux utilisateurs pour définir leurs autorisations et leurs accès.
2. Les administrateurs Keycloak peuvent créer, gérer et assigner des rôles aux utilisateurs via l'interface d'administration ou via l'API de gestion.
3. Les rôles peuvent être utilisés pour contrôler l'accès aux fonctionnalités, aux pages et aux ressources au sein de l'application.
4. Keycloak propose des rôles d'administration spécifiques tels que "admin" ou "superadmin" qui permettent aux utilisateurs d'effectuer des tâches d'administration, telles que la gestion des clients, des utilisateurs, des rôles, etc.
5. Les utilisateurs peuvent se connecter à l'aide de leurs identifiants (nom d'utilisateur et mot de passe) ou d'autres méthodes d'authentification prises en charge, telles que l'authentification à deux facteurs, OAuth 2.0, etc.
6. Keycloak prend en charge l'authentification basée sur les utilisateurs et fournit une interface d'inscription pour permettre aux utilisateurs de créer leurs comptes.
7. Keycloak offre également des fonctionnalités d'authentification avancées, telles que l'authentification à deux facteurs, l'authentification sociale (via des fournisseurs d'identité tels que Google, Facebook, etc.) et l'authentification basée sur des certificats.

3.6 Kafka

Kafka est une plateforme de streaming de données distribuée et évolutive, conçue pour gérer efficacement la transmission et le traitement de flux de données en temps réel. Elle a été développée par Apache Software Foundation.

Kafka est basé sur une architecture de journal de messages distribué, où les données sont stockées sous forme de flux de messages dans des "topics". Les producteurs de données envoient des messages à des "topics" spécifiques, tandis que les consommateurs s'abonnent à ces "topics" pour récupérer les messages. Cela permet une communication asynchrone et une séparation claire entre les producteurs et les consommateurs de données.

Les principales caractéristiques de Kafka incluent :

1. Scalabilité : Kafka est conçu pour gérer de gros volumes de données et peut être mis à l'échelle horizontalement pour répondre aux besoins de performance croissants. Il peut gérer des charges de travail élevées et traiter des milliers de messages par seconde.

2. Tolérance aux pannes : Kafka garantit une haute disponibilité et une tolérance aux pannes en répliquant les données sur plusieurs nœuds du cluster. Cela garantit la fiabilité et la disponibilité des données, même en cas de défaillance d'un ou plusieurs nœuds.
3. Durabilité des données : Les messages stockés dans Kafka sont persistants et peuvent être conservés pendant une période définie. Cela permet de rejouer les messages et de récupérer les données en cas de besoin, ce qui est essentiel pour les cas d'utilisation nécessitant une conservation à long terme des données.
4. Traitement de flux : Kafka est conçu pour le traitement de flux en temps réel. Il permet aux applications de consommer des flux de données en continu et de les traiter en temps réel, ce qui est crucial pour les cas d'utilisation nécessitant une analyse en temps réel, des pipelines de données, etc.
5. Intégration avec d'autres outils : Kafka s'intègre facilement avec d'autres outils et frameworks tels que Spark, Hadoop, Flink, etc. Cela permet une intégration transparente avec l'écosystème Big Data et facilite l'ingestion, le traitement et la diffusion des données.

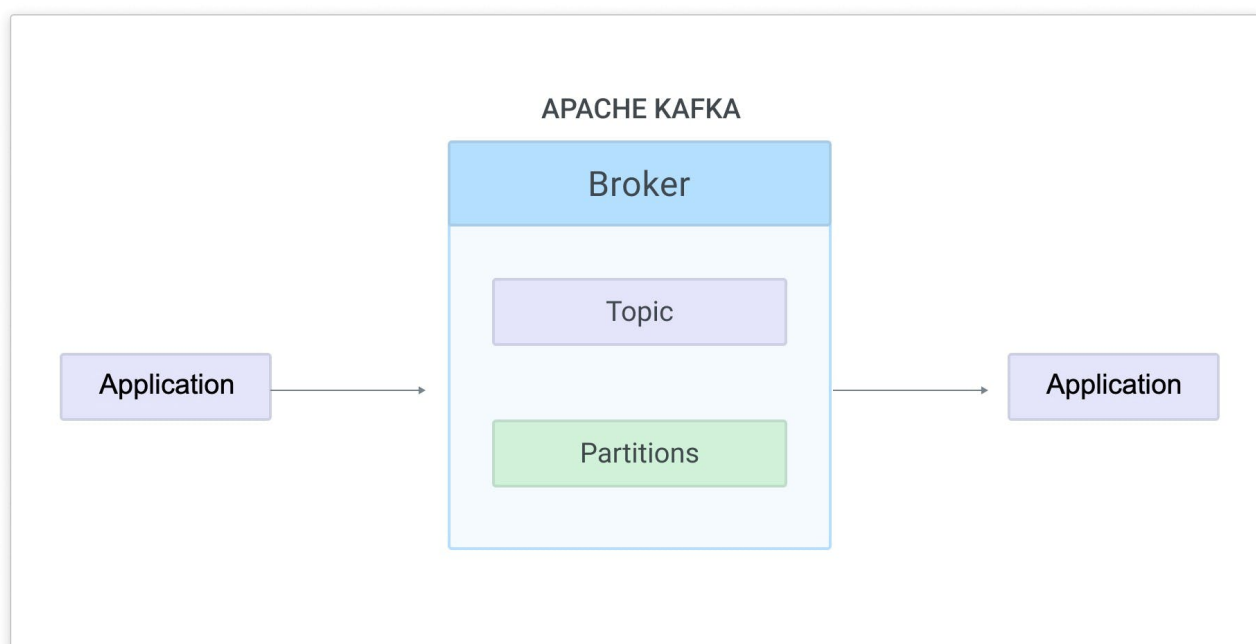


FIGURE 3.3 : Architecture Kafka

Implementation de la solution

Introduction

Ce chapitre se concentre sur l'implémentation de la solution, en mettant l'accent sur le choix du connecteur pour manipuler Delta Lake. Nous discuterons de l'étude de faisabilité des différents connecteurs et de leur impact sur la modularité, les performances et la sécurité.

4.1 Étude de faisabilité

Lors de notre étude de faisabilité, nous avons examiné plusieurs options de connecteurs pour manipuler Delta Lake. Voici un aperçu des connecteurs que nous avons explorés :

1. **Delta Standalone** : Nous avons tenté d'utiliser Delta Standalone, qui est un connecteur indépendant de Spark. Cependant, nous avons constaté que Delta Standalone ne prend en charge que des snapshots de timestamp et ne permet pas les requêtes. De plus, il peut parfois être lourd lors de la récupération de plusieurs enregistrements, ce qui affecte les performances.
2. **Delta Sharing Server** : Delta Sharing Server est une autre option qui nécessite son propre serveur. Bien qu'il soit populaire avec Rust et Python, nous avons constaté que l'utilisation de Delta Sharing Server avec Java ne fournit pas toutes les options disponibles et manque de documentation détaillée.
3. **Spark** : Spark est une option faisable et déjà disponible pour manipuler Delta Lake. Cependant, il nécessite une version spécifique de Spring Boot (2.7-) avec une configuration Maven très spécifique. Bien que Spark offre de nombreuses fonctionnalités et soit bien intégré à l'écosystème de Big Data, son utilisation peut être complexe et nécessite une configuration précise.
4. **Trino** : Trino est un autre connecteur que nous avons exploré. Cependant, il nécessite son propre serveur et une base de données relationnelle (Hive). Trino offre plusieurs avantages, notamment la possibilité d'utiliser du SQL standard et un connecteur Java JDBC pour accéder à Delta Lake. Cela offre une flexibilité et une facilité d'utilisation accrues.

4.2 Benchmark entre Trino et Spark

Trino est un moteur de requêtage distribué et rapide conçu pour exécuter des requêtes SQL interactives sur de grands ensembles de données. Spark, d'autre part, est un système de traitement distribué populaire qui prend également en charge le requêtage de données à grande échelle. Pour cette étude de benchmark, nous allons évaluer les performances de Trino et Spark sur le requêtage de données stockées dans Delta Lake (S3 Amazon dans notre cas), un format de stockage de données optimisé pour les analyses analytiques.

4.2.1 Architecture

Les deux schémas suivant représentent l'architecture respective des deux POC (Proof Of Concept) SPARK et TRINO.

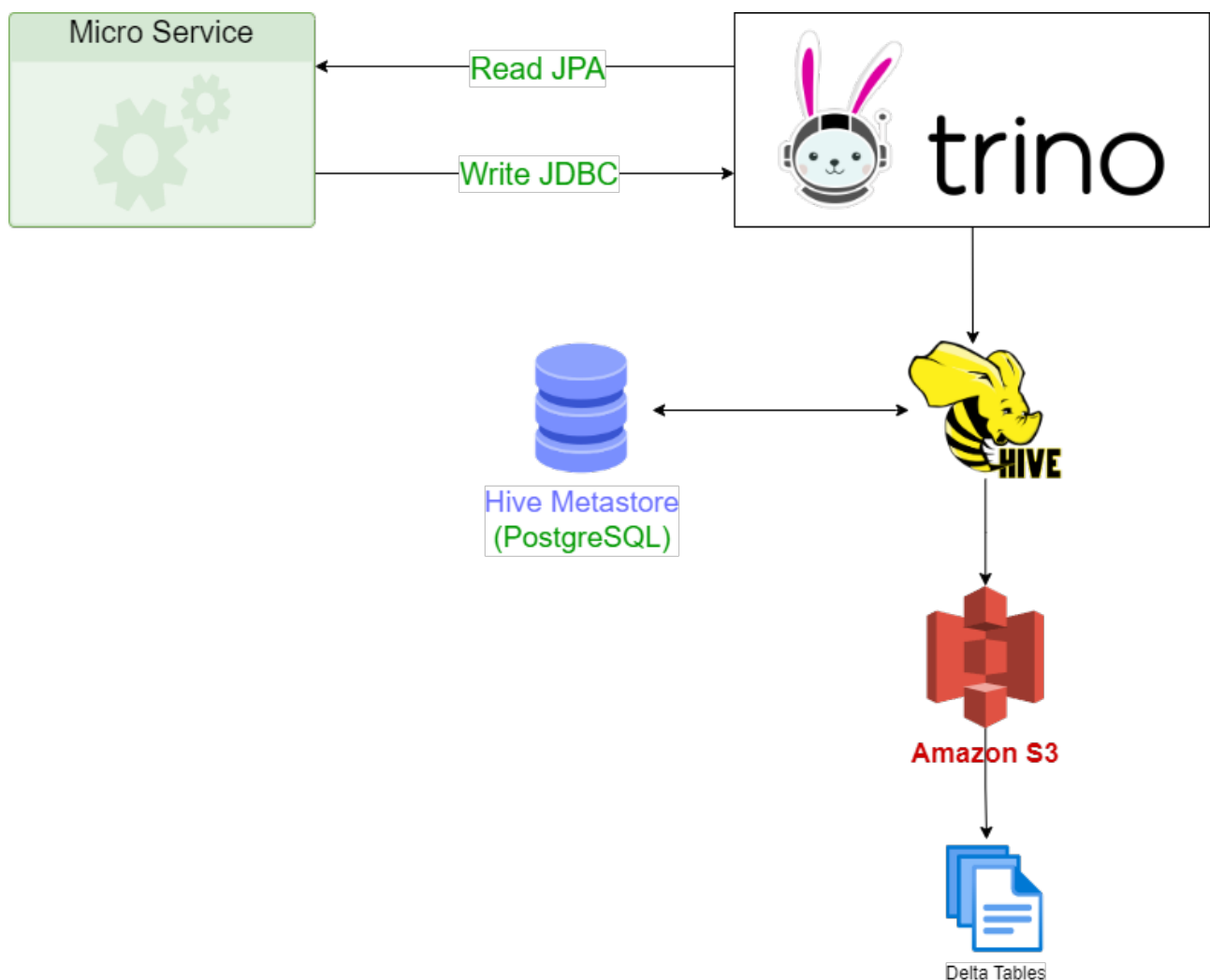


FIGURE 4.1 : Architecture TRINO

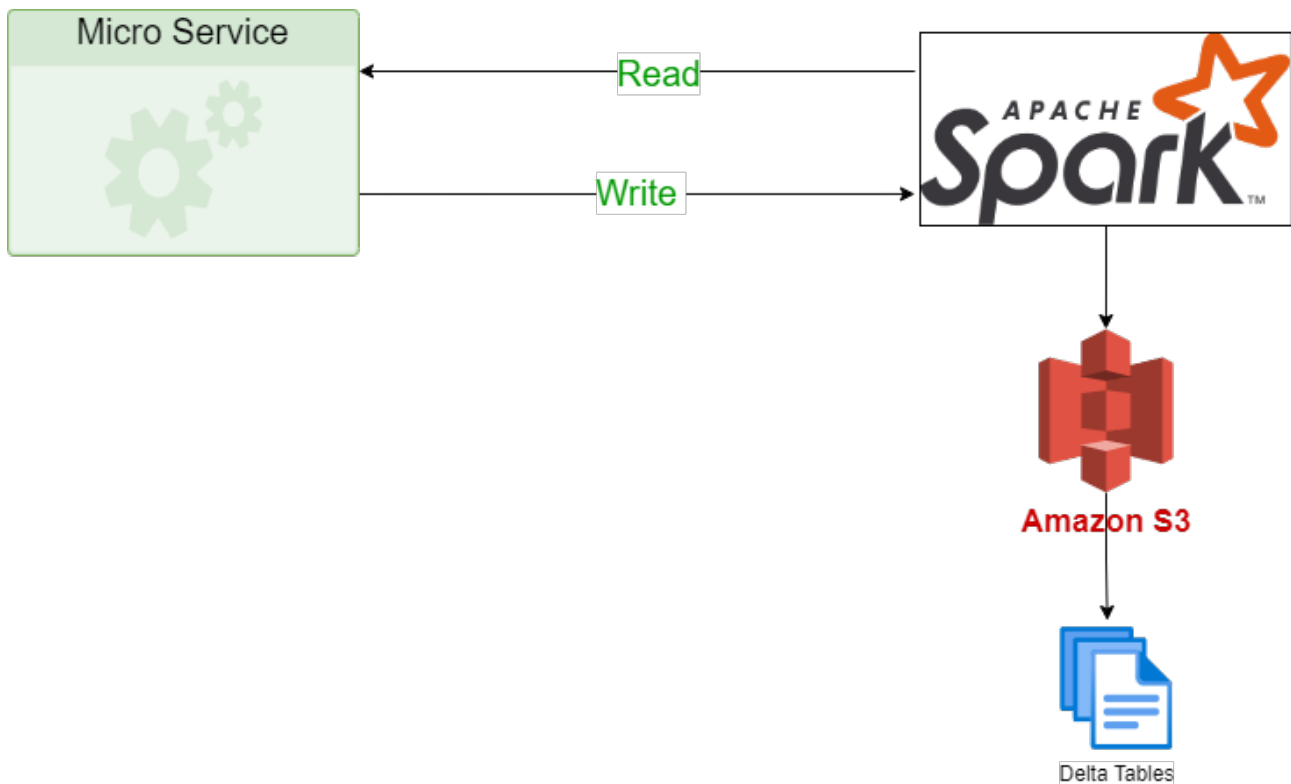


FIGURE 4.2 : Architecture SPARK

4.2.2 Temps d'exécution

- **Trino** : Trino est connu pour sa vitesse d'exécution élevée. Il utilise une architecture optimisée pour exécuter des requêtes SQL de manière très rapide, en exploitant la parallélisation et l'optimisation des requêtes. Trino peut fournir des temps d'exécution courts en ce qui concerne les requêtes sur Delta Lake.
- **Spark** : Spark est également conçu pour offrir des performances élevées, mais il peut nécessiter plus de temps pour lire les données à partir du disque. Cependant, Spark peut offrir des performances élevées lorsqu'il est utilisé avec des opérations de transformation complexes, grâce à son modèle de calcul distribué basé sur le disque.

Nous avons effectué un test de performance sur une table Delta de 2 giga-octets en utilisant 2 ordinateurs :

1. Le premier PC «DELL Latitude 5530» ou nous allons lancer les images Docker suivantes :
2.
 - **Trino** : c'est l'image du serveur Trino pour l'interrogation de données distribué avec le langage SQL
 - **Hive** : Il sera utilisé conjointement avec Trino afin d'interroger et analyser des données stockées dans Hive grâce à la compatibilité entre les deux systèmes et le partage des métadonnées via le Hive Metastore.
 - **PostgreSQL** : Il sera utilisé afin de stocker les métadonnées des tables Delta, des schémas et des partitions, facilitant ainsi la gestion, la découverte et l'accès aux données dans un environnement distribué. Il offre également des fonctionnalités d'optimisation des requêtes

et de gestion des autorisations pour améliorer les performances et la sécurité des analyses de données.

- **AWS** : Où les delta tables sont trouvés.

3. Le Dèxième PC «ASUS ROG G752 VT» c'est là ou sera exécuté les micro-services springboot de SPARK et de TRINO.

La configuration utilisée lors des tests est la suivante :

Composant	Spécifications
Processeur	12th Gen Intel® Core™ i7-1255U 1.70 - 4.8 GHz
Fréquence processeur	2.6 - 3.2 GHz
Mémoire vive	24 GO
Type de mémoire vive	DDR4
Puce graphique	Intel Iris Xe Graphics
Disque dur	512GB PCIe NVMe Class 35 SSD
Type du système d'exploitation	Windows 11 64 bits

TABLE 4.1 : Caractéristiques techniques du PC « DEL Latitude 5530 »

Composant	Spécifications
Processeur	Intel Core i7-6700HQ
Fréquence processeur	2.6 - 3.2 GHz
Mémoire vive	24 GO DDR4
Disque dur	1TO SSD + 1TO HDD
Type de mémoire vive	DDR4-SDRAM
Puce graphique	Nvidia GeForce GTX 970M
Quantité de mémoire graphique	3072 Mo dédiée
Type du système d'exploitation	Windows 10 64 bits

TABLE 4.2 : Caractéristiques techniques du PC «ASUS ROG G752 VT»

Les résultats sont les suivants :

Requêtes	Implémentation	
	Spark (s)	Trino (s)
SELECT (par page et par limite de résultats par page - 100 par page)	1.2	0.43
Agrégation et regroupement	2.1	0.8
Tri	2.2	0.8

TABLE 4.3 : Caractéristiques techniques du PC «ASUS ROG G752 VT»

4.2.3 Gestion de la mémoire

- **Trino** : Trino utilise principalement la mémoire pour accélérer le traitement des requêtes. Il dispose d'un moteur de requêtage en mémoire qui optimise l'accès aux données et minimise les temps d'accès. Cependant, cela signifie que Trino peut nécessiter une quantité de mémoire significative pour traiter de grands ensembles de données.
- **Spark** : Spark utilise un modèle de calcul distribué basé sur le disque, ce qui signifie que les données sont généralement stockées sur le disque et chargées en mémoire selon les besoins.

Cela permet à Spark de gérer des ensembles de données beaucoup plus volumineux que ce qui peut tenir en mémoire.

4.2.4 Écosystème et intégrations

- **Trino** : Trino dispose d'un écosystème en pleine croissance avec une large gamme de connecteurs et d'intégrations, ce qui permet d'accéder à différentes sources de données, y compris Delta Lake. Il est compatible avec divers outils de visualisation et de traitement des données.
- **Spark** : Spark est un projet mature avec un écosystème riche d'outils, de bibliothèques et de connecteurs. Il bénéficie d'une large communauté de développeurs et de nombreuses ressources d'apprentissage. Spark prend également en charge le traitement de données en continu et le traitement par lots.

4.2.5 Évolutivité

- **Trino** : Trino est conçu pour être hautement évolutif et peut gérer de grands ensembles de données. Il peut être facilement configuré pour s'adapter à des charges de travail intensives et à des volumes de données importants.
- **Spark** : Spark est également conçu pour l'évolutivité et peut traiter des ensembles de données massifs. Il dispose d'un système de traitement distribué qui peut s'adapter à différentes configurations de clusters et de ressources.









4.2.6 Support des fonctionnalités Delta Lake

- **Trino** : Trino dispose d'un connecteur officiel pour Delta Lake, ce qui lui permet de lire et d'écrire des données dans Delta Lake. Il prend en charge les fonctionnalités essentielles de Delta Lake, telles que la gestion des transactions ACID, les mises à jour incrémentielles et les opérations de fusion ("merge").
- **Spark** : Spark dispose également d'un support intégré pour Delta Lake. Il offre des fonctionnalités avancées de gestion des données Delta, notamment la prise en charge des transactions ACID, des opérations de fusion performantes et des mécanismes de contrôle de version des données.

4.2.7 Facilité d'utilisation

- **Trino** : Trino offre une interface SQL standard, ce qui facilite son utilisation pour les utilisateurs familiers avec SQL. Il propose également des outils de requêtage interactif conviviaux et des options de configuration flexibles.
- **Spark** : Spark propose une interface SQL via Spark SQL, mais il est également orienté vers le traitement de données par le biais d'un API plus large. Spark peut être plus complexe à utiliser pour les utilisateurs moins familiers avec les concepts du traitement distribué.

4.2.8 Autres statistiques

	Spark	Trino
Expérience Développeur	 La syntaxe de Spark exige que les PME aient une expérience en ingénierie de données	 Une interface SQL standard, ce qui facilite son utilisation pour les utilisateurs familiers avec SQL
Coût	Équipe :  Infrastructure : \$	Équipe :  Infrastructure : \$
Fiabilité	 Produit mature maintenu en interne et pris en charge par Google Dataproc	 A été en panne à quelques reprises car l'équipe interne apprend à maintenir et à évoluer
Caractéristiques et Open sources	 A été soutenu par la communauté open source depuis 2014	 A été soutenu par la communauté open source depuis 2019. Shopify a apporté des contributions.
Usage	~800,000 emplois/jour	3000 utilisateurs actifs/semaine ~200,000 requêtes/semaine

4.3 Schématisation de la Solution

Dans le cadre de la schématisation de la solution, nous avons opté pour une architecture basée sur des microservices, utilisant les technologies suivantes : Spring Data, JPA, Hibernate, JDBC et Trino.

Les microservices sont développés en utilisant Spring Data, qui fournit une abstraction de haut niveau pour interagir avec la base de données. Cela permet d'utiliser des annotations et des interfaces pour définir les entités, les requêtes et les opérations de persistance. JPA (Java Persistence API) est utilisé comme couche d'interface pour interagir avec la base de données, tandis que Hibernate est utilisé comme fournisseur de persistance, permettant la gestion des objets Java et leur mapping vers Trino.

La communication entre les microservices est assurée par un broker Kafka. Kafka est une plateforme de streaming distribuée qui permet aux microservices d'échanger des messages en temps réel. Les microservices publient des événements sur des sujets Kafka, et d'autres microservices peuvent s'abonner à ces sujets pour recevoir les événements correspondants.

Pour gérer l'authentification et l'autorisation, nous utilisons Spring Cloud Gateway. Ce composant reçoit les appels de services provenant des applications Web, mobiles ou de bureau. Ces applications envoient un jeton JWT (JSON Web Token) à Spring Cloud Gateway pour authentification. Ce dernier vérifie la disponibilité du jeton en contactant Keycloak, qui est responsable de la gestion des

identités et des accès. Une fois le jeton validé, Spring Cloud Gateway autorise la demande à passer vers les microservices.

Les microservices communiquent avec Trino en utilisant le connecteur JDBC. Trino est responsable de l'interrogation et du traitement des données sur une grande quantité de sources de données. Il stocke les métadonnées dans Hive, qui est un entrepôt de données basé sur Hadoop. Trino utilise également le concept de tables Delta pour gérer les modifications incrémentielles des données. Ces tables Delta sont stockées sur AWS (Amazon Web Services), offrant une solution de stockage évolutive et fiable.

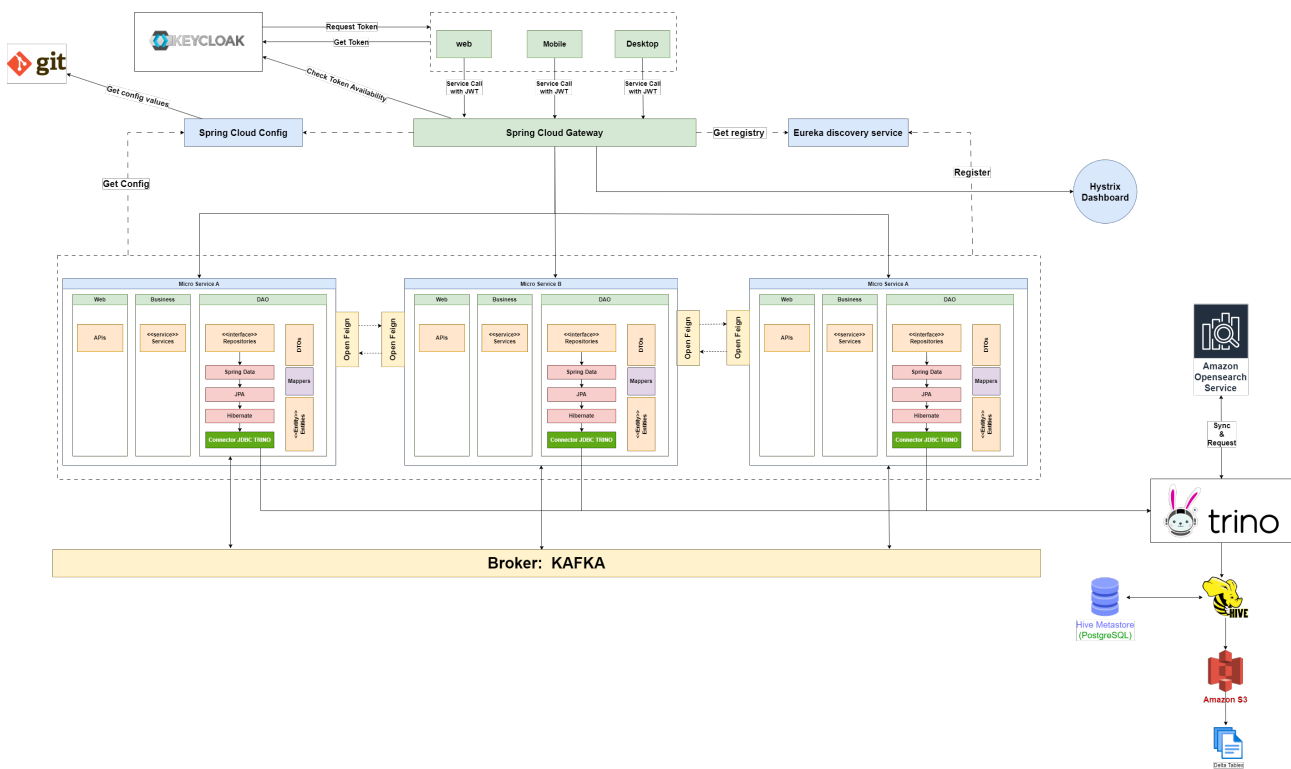


FIGURE 4.3 : Architecture de solution

4.4 Avantages

L'architecture que nous avons choisie présente plusieurs avantages significatifs :

- **Évolutivité** : L'architecture basée sur des microservices permet une évolutivité horizontale, ce qui signifie que chaque microservice peut être déployé et mis à l'échelle indépendamment en fonction de ses besoins spécifiques. Cela permet d'ajuster la capacité et les ressources de manière granulaire, ce qui facilite la gestion des charges de travail élevées et l'adaptation aux besoins croissants de l'application.
- **Modularité** : La conception modulaire des microservices permet une gestion plus efficace du développement, du déploiement et de la maintenance. Chaque microservice peut être développé indépendamment, ce qui facilite la collaboration entre les équipes et favorise la réutilisation du code. De plus, les modifications ou les mises à jour apportées à un microservice n'ont

pas d'impact sur les autres, ce qui réduit les risques d'erreurs et de conflits.

- **Flexibilité technologique :** Les microservices offrent une flexibilité en termes de choix technologiques. Chaque microservice peut être développé en utilisant les technologies les plus adaptées à sa fonctionnalité spécifique. Cela permet d'exploiter au mieux les avantages de chaque technologie et de choisir les outils les mieux adaptés à chaque composant de l'architecture.
- **Communication efficace :** L'utilisation de Kafka comme broker de streaming facilite la communication entre les microservices. Kafka garantit la fiabilité et la scalabilité des échanges d'événements en temps réel, ce qui permet une communication fluide et une synchronisation efficace entre les différents composants du système.
- **Sécurité renforcée :** L'intégration de Keycloak pour l'authentification et l'autorisation offre un niveau élevé de sécurité. Les jetons JWT sont utilisés pour l'identification des utilisateurs, et Spring Cloud Gateway assure la vérification et la gestion de ces jetons. Cela garantit que seules les personnes autorisées ont accès aux ressources et aux fonctionnalités appropriées, renforçant ainsi la sécurité de l'application.
- **Performances optimisées :** L'utilisation de Trino comme moteur de requête distribué permet d'interroger et de traiter de grandes quantités de données de manière parallèle et efficace. Trino offre des performances élevées et une optimisation des requêtes, ce qui permet d'obtenir des résultats plus rapidement et d'améliorer l'expérience utilisateur.

Conclusion

L'étude de faisabilité des connecteurs pour manipuler Delta Lake nous a permis de prendre des décisions éclairées sur l'implémentation de la solution. Bien que Spark soit largement utilisé et intégré à l'écosystème Big Data, Trino offre des avantages significatifs en termes de flexibilité et de performances pour les requêtes SQL. En fonction de nos besoins spécifiques et des contraintes techniques, nous avons opté pour l'utilisation de Trino comme connecteur pour manipuler Delta Lake dans notre solution.

Conclusion Générale

Le stage a été une expérience enrichissante qui a permis d'explorer divers aspects des microservices en utilisant des technologies telles que Delta Lake, Trino, Spring Boot, et Keycloak. L'environnement de travail était propice à l'apprentissage et à la mise en pratique de ces concepts.

L'adoption des microservices présente de nombreux avantages par rapport à une architecture monolithique. Les microservices offrent une meilleure scalabilité et flexibilité, permettant le déploiement, le développement et la mise à l'échelle indépendants de chaque service. De plus, la communication entre les microservices via des API facilite l'intégration et la collaboration entre les différentes parties du système.

Pendant le stage, nous avons appris à concevoir et implémenter des microservices en utilisant Spring Boot, en exploitant ses fonctionnalités de persistance, de sécurité, et de création d'API REST. Nous avons également intégré Keycloak pour gérer l'authentification et l'autorisation des utilisateurs dans notre architecture de microservices. L'utilisation de Delta Lake a permis de garantir la fiabilité des données et de faciliter la gestion des mises à jour.
