École Marocaine des Sciences de l'Ingénieur

Rapport de Projet Académique

Architecture des Composants d'Entreprise

Mise en place d'une Architecture Microservices pour une Banque Digitale

---

Réalisé par :
KHARRAZ Kenza
MOUGHIT Aya

Encadré par :
Pr. Abdelaziz Ettaoufik

Année Universitaire : 2025 - 2026

Remerciements

Nous tenons à exprimer notre profonde gratitude à notre professeur, Monsieur Abdelaziz Ettaoufik, pour son encadrement, ses conseils précieux et son expertise technique tout au long de ce projet. Nous remercions également l'EMSI pour la qualité de l'enseignement dispensé.

Résumé

Ce projet présente la conception et la mise en œuvre d'une application bancaire numérique ("Digital Banking") fondée sur une architecture en microservices. L'objectif premier est de déployer une solution modulaire, scalable et résiliente, apte à gérer la complexité des opérations bancaires en temps réel. L'infrastructure technique s'appuie sur l'écosystème Spring Cloud (Eureka, Gateway, OpenFeign) pour l'orchestration des services, tandis que l'interface utilisateur, développée en React, garantit une expérience client fluide et réactive.

Table des matières

Table des figures

Chapitre 1

Introduction Générale

1.1 Contexte et Enjeux

Le secteur bancaire traverse une mutation profonde, caractérisée par la digitalisation intégrale des services financiers. Face à des usagers exigeant instantanéité, haute disponibilité et sécurité accrue, les architectures monolithiques traditionnelles atteignent leurs limites structurelles en matière d'évolutivité et de maintenance. Dès lors, la transition vers une architecture distribuée s'impose comme un impératif stratégique pour garantir l'agilité et la pérennité des systèmes d'information.

1.2 Présentation de l'Organisme (EMSI)

L'École Marocaine des Sciences de l'Ingénieur (EMSI) se distingue par son excellence académique et sa pédagogie axée sur l'innovation. Ce projet s'inscrit dans le cadre du module "Architecture des Composants d'Entreprise", dont la vocation est de permettre aux futurs ingénieurs de maîtriser les paradigmes logiciels avancés nécessaires à la conception de systèmes complexes.

1.3 Objectifs du Projet

Ce travail vise à atteindre les objectifs techniques et fonctionnels suivants :

- **Architecture :** Concevoir un système découplé reposant sur le paradigme des Microservices pour assurer une scalabilité horizontale.
- **Backend :** Implémenter des services RESTful robustes à l'aide du framework Spring Boot 3.
- **Infrastructure :** Sécuriser et orchestrer les flux via une API Gateway centralisée et un service de découverte Eureka.

— **Frontend :** Développer une interface utilisateur ergonomique, modulaire et moderne basée sur la bibliothèque React.

1.4 Méthodologie de Travail

Pour mener à bien ce projet, nous avons adopté une démarche itérative inspirée de la méthode **Agile/Scrum**. Cette approche nous a permis de diviser le développement en plusieurs cycles (Sprints) :

— **Sprint 1 (Conception & Socle) :** Analyse des besoins, définition de l'architecture et mise en place des serveurs d'infrastructure (Eureka, Config).

— **Sprint 2 (Développement Backend) :** Développement des microservices métier (*Compte-service*, *Transaction-service*) et tests unitaires.

— **Sprint 3 (Frontend & Intégration) :** Création de l'interface React, consommation des API via Axios et tests d'intégration globaux.

Chapitre 2

Analyse et Conception

2.1 Analyse des Besoins Fonctionnels

L'analyse fonctionnelle a permis d'identifier deux acteurs principaux : les Clients, utilisateurs finaux des services, et les Administrateurs, garants de la gestion du système. Le périmètre fonctionnel couvre les processus métier essentiels suivants :

- **Gestion des Clients :** Création, modification et consultation des profils clients au sein de l'agence.
- **Consultation des Comptes :** Accès à la liste exhaustive des comptes bancaires et visualisation détaillée de l'état de chacun (Solde, Type, Date).
- **Opérations Financières :** Exécution sécurisée de transactions (virements de compte à compte, débits, crédits).
- **Traçabilité :** Accès à l'historique complet des transactions pour un audit et un suivi précis.

2.2 Analyse des Besoins Non-Fonctionnels

Au-delà des fonctionnalités métiers, le système doit répondre à des exigences de qualité logicielle :

- **Disponibilité :** Le système doit rester opérationnel même en cas de défaillance d'un service (Gestion via Gateway et Discovery).
- **Performance :** Les temps de réponse des API doivent être optimisés pour garantir une expérience fluide.
- **Scalabilité :** L'architecture doit permettre l'ajout facile de nouvelles instances de services pour absorber une montée en charge.
- **Maintenabilité :** Le code doit être modulaire et documenté pour faciliter les évolutions futures.

2.3 Spécifications Techniques et Prérequis

Le tableau suivant récapitule les prérequis nécessaires au déploiement de la solution :

Composant	**Spécification / Version**
Langage Backend	Java 17 (LTS)
Framework Backend	Spring Boot 3.x
Gestion de dépendances	Apache Maven 3.8+
Framework Frontend	React.js 18.x
Base de données	H2 (In-Memory) / MySQL (Production)
Serveur de Découverte	Spring Cloud Eureka

TABLE 2.1 – Spécifications techniques du projet

2.4 Modélisation UML

La phase de conception s'est appuyée sur le langage UML pour formaliser la structure et le comportement du système.

2.4.1 Diagramme de Classes

Le diagramme de classes ci-dessus représente la structure statique du système, centrée sur le concept de **Compte**, modélisé comme une classe abstraite utilisant une stratégie d'héritage *Single Table* pour ses spécialisations (*CompteCourant* et *CompteEpargne*). L'entité **Client** entretient une relation "un-à-plusieurs" avec ces comptes, permettant une gestion centralisée des produits bancaires. Parallèlement, chaque compte est associé à un historique de **Transaction**s immuables, garantissant une traçabilité complète des opérations financières (débits, crédits, virements) au sein d'une architecture orientée objet modulaire et extensible.

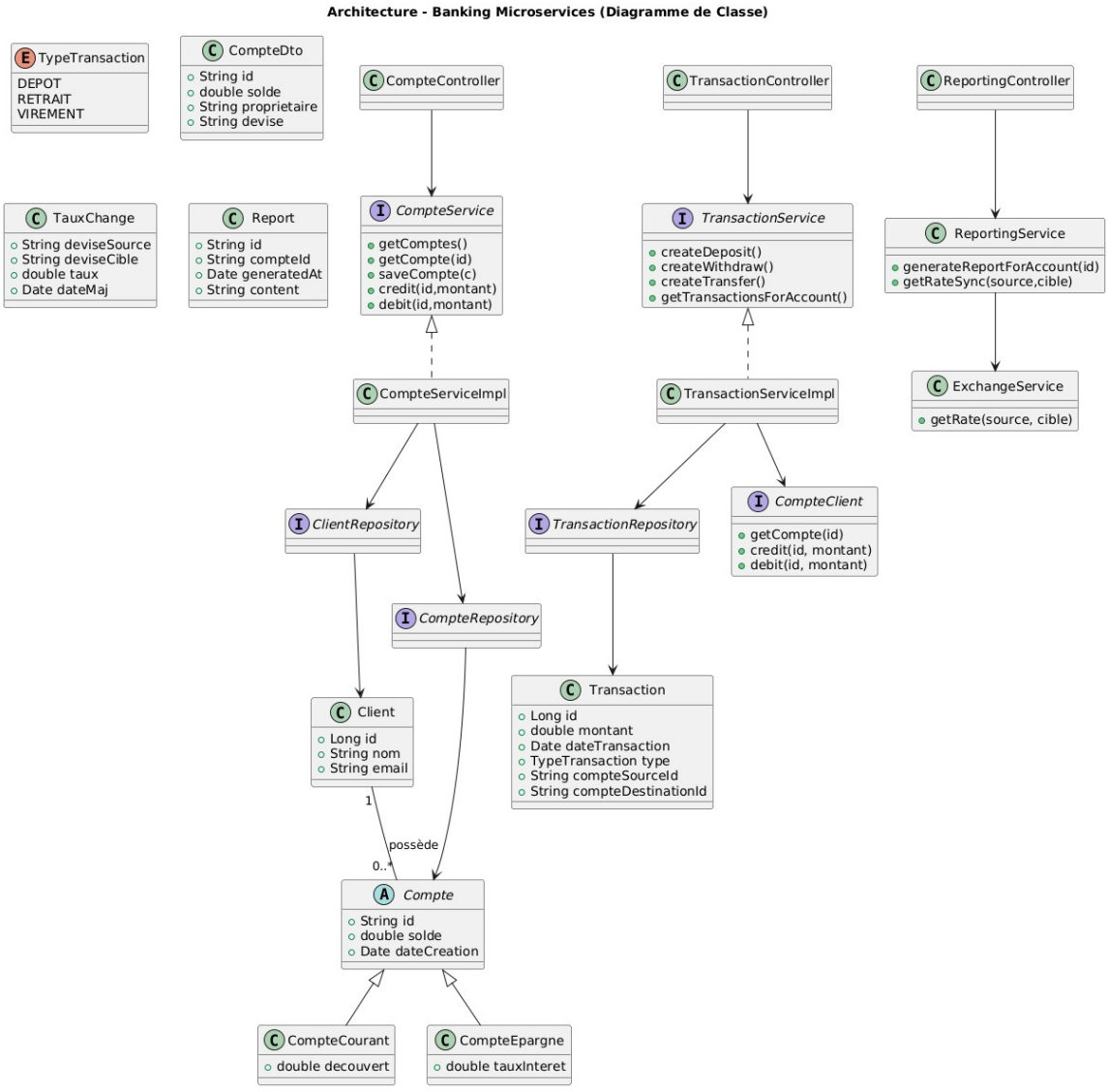

FIGURE 2.1 – Diagramme de Classes de l'application Digital Banking

Chapitre 3

Réalisation et Implémentation

3.1 Environnement Technique

La réalisation de ce projet s'appuie sur une stack technologique moderne et éprouvée :

- **Backend :** Langage Java 17, Framework Spring Boot 3 et écosystème Spring Cloud.
- **Frontend :** React.js pour la construction d'interfaces dynamiques, couplé à Axios pour la consommation d'API et Bootstrap pour le design system.
- **Persistance :** Base de données H2 (in-memory) et Spring Data JPA pour l'abstraction ORM.
- **Outils de Développement :** Maven pour la gestion des dépendances, IntelliJ IDEA comme IDE et Postman pour les tests d'API.

3.1.1 Détail des Outils Utilisés

FIGURE 3.1 – Java 17

Java 17 : Socle de notre développement backend, choisi pour sa stabilité (LTS), ses performances et ses fonctionnalités modernes (Records, Switch expressions) qui facilitent l'écriture d'un code robuste et lisible.

FIGURE 3.2 – Spring Boot

Spring Boot : Accélère le développement de nos microservices grâce à son approche "Convention over Configuration". Il fournit un serveur embarqué et simplifie la gestion des dépendances pour un déploiement rapide.

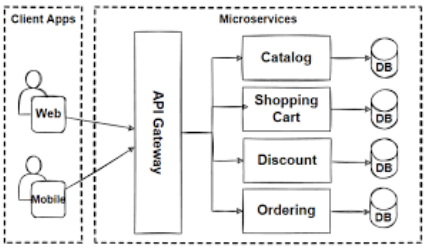

FIGURE 3.3 – Spring Cloud

Spring Cloud : Fournit les briques essentielles pour gérer les interactions dans un système distribué complexe, notamment la découverte de services via Eureka et la configuration centralisée.

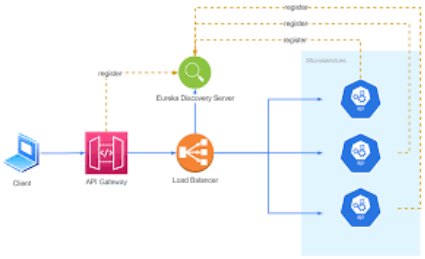

FIGURE 3.4 – Architecture Microservices

Architecture Microservices : Approche architecturale privilégiant le découpage de l'application en services autonomes et faiblement couplés, favorisant ainsi la scalabilité, la maintenance et l'évolutivité du système bancaire.

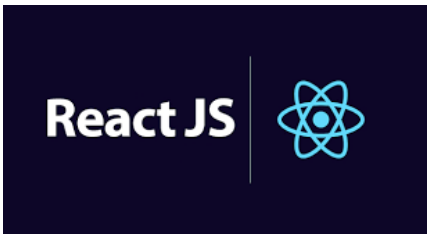

FIGURE 3.5 – React.js

React.js : Bibliothèque JavaScript utilisée pour créer une interface utilisateur dynamique et réactive. Son architecture à base de composants permet une modularité et une réutilisabilité maximale du code frontend.

FIGURE 3.6 – Spring Data JPA

Spring Data JPA : Simplifie la couche d'accès aux données en offrant une abstraction puissante sur JDBC et Hibernate, permettant de manipuler les bases de données via des interfaces Java standardisées.

FIGURE 3.7 – H2 Database

H2 Database : Base de données relationnelle légère fonctionnant en mémoire. Elle est idéale pour le développement et les tests, permettant un prototypage rapide sans configuration d'infrastructure lourde.

FIGURE 3.8 – Maven

Maven : Outil de gestion et d'automatisation de production. Il gère les dépendances, la compilation et le packaging de nos applications Java, assurant la cohérence du cycle de vie du logiciel.

FIGURE 3.9 – Postman

Postman : Plateforme essentielle pour le développement d'API. Elle nous permet de tester, documenter et valider les endpoints de nos microservices, assurant ainsi la fiabilité des échanges de données avant l'intégration frontend.

3.2 Détail des Microservices et Implémentation

Cette section approfondit le rôle et l'implémentation technique des différents services composants notre architecture bancaire.

3.2.1 Discovery Service (Eureka Server)

Le service de découverte est le pivot central de la communication. Il permet l'auto-enregistrement des instances, évitant ainsi le codage en dur des adresses IP.

```
server:
  port: 8761
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

Listing 3.1 – Configuration Eureka Server

3.2.2 Gateway Service (Routage)

La Gateway agit comme un point d'entrée unique. Elle centralise les requêtes et les redirige dynamiquement vers les microservices métier enregistrés sur Eureka.

```
spring:
  cloud:
    gateway:
      routes:
        - id: compte-service
          uri: lb://COMPTE-SERVICE
          predicates:
            - Path=/comptes/**
```

Listing 3.2 – Configuration du routage Gateway

3.2.3 Compte Service (Métier)

Responsable de la gestion des comptes et clients. Il expose des API REST pour la création de comptes et la gestion des profils.

```
public interface AccountRepository extends JpaRepository<BankAccount,
    String> {
    List<BankAccount> findByCustomerId(Long customerId);
}
```

Listing 3.3 – Interface AccountRepository

3.2.4 Transaction Service (Opérations)

Ce service gère les transferts et opérations financières. Il utilise Spring Cloud OpenFeign pour interroger le service de compte de manière synchrone.

```
@FeignClient(name = "COMPTE-SERVICE")
public interface AccountRestClient {
    @GetMapping("/comptes/{id}")
    BankAccountDTO getBankAccount(@PathVariable String id);
}
```

Listing 3.4 – Client Feign pour la communication inter-service

3.3 Communication et Orchestration

L'écosystème repose sur une communication REST pilotée par **OpenFeign**. Cette abstraction permet de traiter les appels distants comme des appels de méthodes Java locaux, garantissant une forte maintenabilité.

3.4 Gestion de la Persistance

Chaque microservice métier dispose de sa propre base de données **H2** (in-memory) pour isoler les données et respecter le paradigme microservices. L'ORM **Spring Data JPA** assure la transition fluide entre les objets métier et les tables relationnelles.

3.5 Architecture Globale de Déploiement

L'architecture modulaire mise en place se décompose en quatre services interconnectés :

1. **Discovery Service (Eureka) :** Assure l'enregistrement et la découverte dynamique des instances de services.
2. **Gateway Service :** Agit comme point d'entrée unique, gérant le routage, l'authentification et le filtrage des requêtes.
3. **Compte Service :** Microservice métier dédié à la gestion du cycle de vie des comptes bancaires.
4. **Transaction Service :** Microservice métier responsable du traitement et de l'historisation des opérations financières.

3.6 Interfaces Utilisateur

Cette section présente les écrans principaux de l'application.

3.6.1 Liste des Comptes

L'interface ci-dessus présente une vue synthétique de tous les comptes détenus par le client. Pour chaque compte, le système affiche son identifiant unique, son solde actuel en temps réel, sa date de création ainsi que son état (ACTIF, SUSPENDU ou BLOQUÉ). Cette vue permet au client d'avoir une situation financière globale instantanée dès sa connexion.

3.6.2 Détails et Opérations

En sélectionnant un compte spécifique, l'utilisateur accède à la vue détaillée illustrée ci-dessus. Cette page reprend les informations essentielles du compte et sert de point de

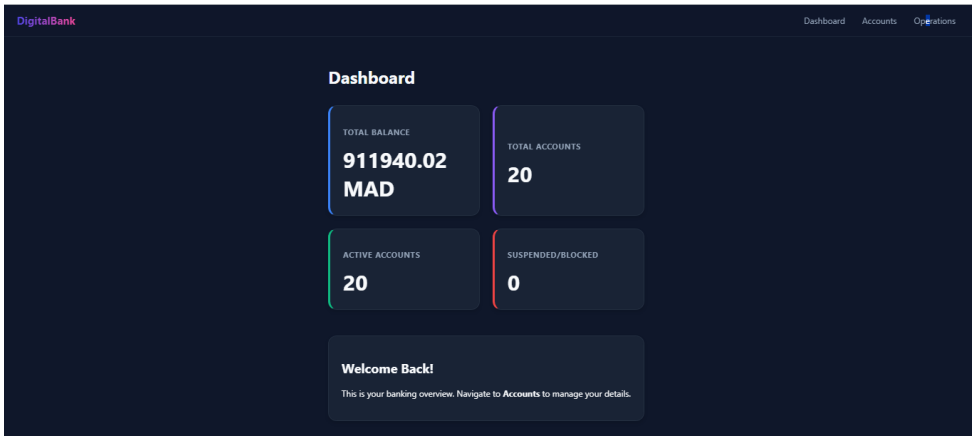

FIGURE 3.10 – Interface de liste des comptes bancaires

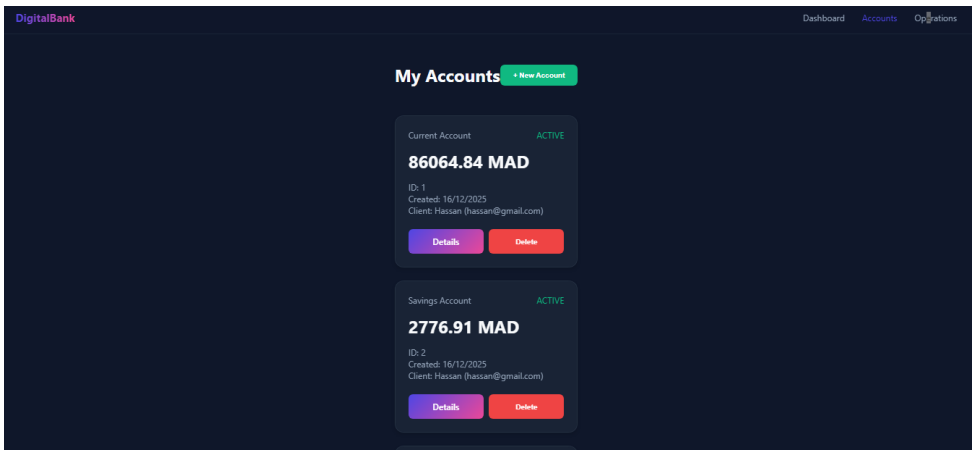

FIGURE 3.11 – Vue détaillée d'un compte

départ pour des actions contextuelles. Le design épuré met en avant le solde pour une lisibilité optimale.

3.6.3 Transactions

La section "Transactions" ci-dessus liste l'historique complet des opérations financières (débits, crédits, virements) effectuées sur le compte. Les données sont présentées sous forme tabulaire, triées par ordre antéchronologique (les plus récentes en premier). Chaque ligne précise l'identifiant de la transaction, le montant impliqué, le type d'opération ainsi que la date d'exécution.

3.6.4 Formulaire de Virement

Pour effectuer un transfert d'argent, l'utilisateur utilise le formulaire dédié ci-dessus. Il doit spécifier le compte source (débiteur), l'identifiant du compte de destination (bénéficiaire) et le montant à transférer. Le système effectue des vérifications de cohérence (solde suffisant, existence du compte destinataire) avant de valider l'opération de manière atomique via le

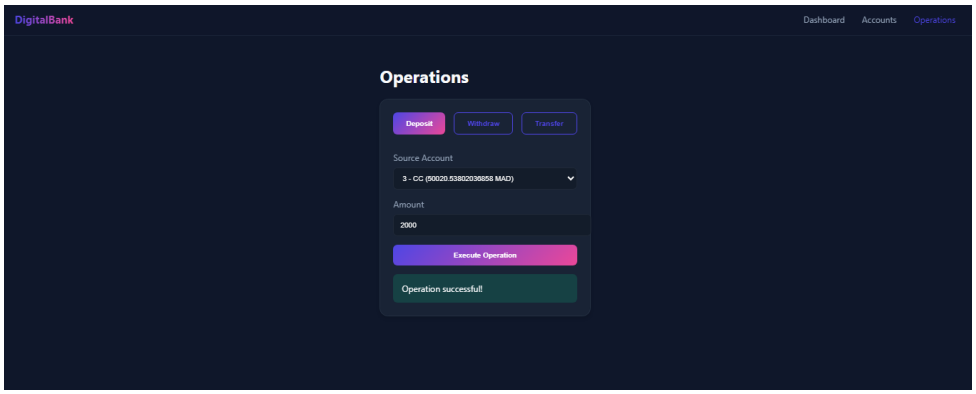

FIGURE 3.12 – Historique des transactions

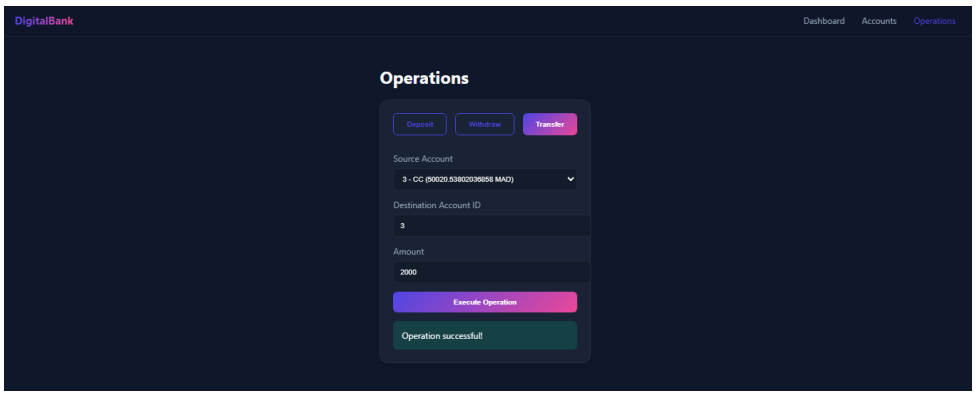

FIGURE 3.13 – Formulaire d'exécution de virement

`Transaction Service`.

Chapitre 4

Guide d'Installation et d'Utilisation

Cette section fournit les instructions nécessaires pour déployer et exécuter l'application localement.

4.1 Prérequis

Avant de commencer, assurez-vous que les composants suivants sont installés sur votre machine :

- **JDK 17** ou version ultérieure.
- **Apache Maven 3.8+** pour le build backend.
- **Node.js & npm** pour le frontend React.
- **Git** pour le contrôle de version.
- Un IDE (IntelliJ IDEA, VS Code ou Eclipse).

4.2 Procédure de Lancement

L'architecture microservices impose un ordre de démarrage précis pour garantir la bonne découverte des services.

4.2.1 Étape 1 : Le Discovery Service

Démarrez le projet `discovery-service`. Attendez que la console affiche que le serveur est prêt sur le port 8761.

```
mvn spring-boot:run
```

4.2.2 Étape 2 : Les Microservices Métier

Lancez simultanément `compte-service` et `transaction-service`. Ils s'enregistreront automatiquement sur Eureka.

4.2.3 Étape 3 : La Gateway

Démarrez `gateway-service`. C'est par ce port (généralement 8888 ou 9999) que transiteront toutes les requêtes frontend.

4.2.4 Étape 4 : Le Frontend React

Accédez au dossier `frontend-react`, installez les dépendances puis lancez le serveur de développement :

```
npm install
npm start
```

Chapitre 5

Tests et Résultats

La phase de test est cruciale pour garantir la fiabilité du système bancaire.

5.1 Tests Unitaires (JUnit & Mockito)

Nous avons implémenté des tests unitaires pour valider la logique métier des services. L'usage de Mockito permet d'isoler les composants en simulant les dépendances (Repositories).

```
@Test
void testTransfer() {
    // Cas de test pour un virement reussi
}
```

Listing 5.1 – Exemple de test unitaire

5.2 Tests d'Intégration avec Postman

Postman a été utilisé pour valider les endpoints REST. Nous avons créé des collections de tests automatisés pour vérifier :

- La création de nouveaux clients.
- L'attribution de comptes à un client existant.
- L'exécution de virements avec vérification du solde.

5.3 Validation du Parcours Utilisateur

L'intégration du frontend avec les microservices a permis de valider des scénarios réels :

1. **Connexion :** Authentification et récupération du profil.

2. **Navigation :** Consultation fluide des différents comptes.
3. **Opération :** Réalisation d'un virement et mise à jour instantanée du solde via les WebServices.

Chapitre 6

Conclusion et Perspectives

Ce projet a permis de démontrer la pertinence de l'approche microservices dans le contexte bancaire, en validant la faisabilité technique d'une architecture distribuée, robuste et évolutive. La séparation des responsabilités et l'usage de technologies standards ont facilité le développement et la maintenance de la solution.

L'expérience acquise lors de ce projet souligne l'importance d'une conception rigoureuse (UML) et d'une orchestration efficace (Spring Cloud) pour gérer la complexité inhérente aux systèmes distribués.

6.1 Perspectives d'Évolutions

Plusieurs axes d'amélioration sont envisagés pour transformer ce prototype en solution de production :

- **Sécurité :** Mise en place de Spring Security avec JWT ou OAuth2 via Keycloak.
- **Conteneurisation :** Utilisation de Docker et Docker-Compose pour simplifier le déploiement multi-services.
- **Architecture Événementielle :** Introduction d'Apache Kafka pour la gestion asynchrone des transactions et les notifications en temps réel.
- **Observabilité :** Intégration de Prometheus et Grafana pour le monitoring des services.

Bibliographie

[1] Spring Cloud Documentation.
`https://spring.io/projects/spring-cloud`

[2] React Official Documentation.
`https://reactjs.org`