Rapport de Projet de Fin d'Études

Plateforme E-Commerce

basée sur une Architecture Microservices

Développement d'une application web distribuée avec Spring Boot et React

Réalisé par :
Anass EL GHAZOUI

Encadré par :
Nom de l'encadrant

Année Universitaire 2024-2025

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Dans un contexte économique de plus en plus digitalisé, le commerce électronique s'impose comme un pilier fondamental de l'économie moderne. Les entreprises cherchent constamment à améliorer leurs plateformes pour offrir une expérience utilisateur optimale tout en garantissant la scalabilité, la fiabilité et la maintenabilité de leurs systèmes.

Contexte du projet

Le présent projet s'inscrit dans le cadre du développement d'une plateforme e-commerce complète utilisant une architecture microservices. Cette approche architecturale moderne permet de décomposer une application monolithique en plusieurs services indépendants, chacun responsable d'une fonctionnalité métier spécifique. Cette modularité offre de nombreux avantages en termes de scalabilité, de déploiement indépendant et de maintenance.

L'architecture microservices est devenue le standard de facto pour les applications d'entreprise modernes, notamment dans le domaine du e-commerce où les exigences en termes de disponibilité, de performance et d'évolutivité sont particulièrement élevées. Des géants du web comme Amazon, Netflix et eBay ont démontré l'efficacité de cette approche pour gérer des millions de transactions quotidiennes.

Problématique

Le développement d'une plateforme e-commerce moderne soulève plusieurs défis techniques et architecturaux :

— **Scalabilité** : Comment garantir que le système peut gérer une augmentation significative du trafic sans dégradation des performances ?

— **Disponibilité** : Comment assurer une disponibilité continue du service, même en cas de défaillance d'un composant ?

— **Sécurité** : Comment protéger les données sensibles des utilisateurs et sécuriser les transactions ?

— **Maintenabilité** : Comment faciliter l'évolution et la maintenance du système dans le temps ?

— **Intégration** : Comment orchestrer efficacement la communication entre les différents services ?

Objectifs du projet

Ce projet vise à concevoir et développer une plateforme e-commerce complète en adoptant une architecture microservices. Les objectifs spécifiques sont les suivants :

1. **Conception d'une architecture microservices robuste** : Définir une architecture modulaire avec des services indépendants pour la gestion des produits, des commandes et des utilisateurs.
2. **Implémentation des services backend** : Développer les microservices en utilisant Spring Boot et Spring Cloud, en intégrant les patterns essentiels (Service Discovery, API Gateway, Configuration centralisée).
3. **Développement d'une interface utilisateur moderne** : Créer une application frontend réactive avec React, offrant une expérience utilisateur fluide et intuitive.
4. **Sécurisation de l'application** : Mettre en place un système d'authentification et d'autorisation basé sur JWT, avec vérification par email.
5. **Containerisation et déploiement** : Utiliser Docker pour containeriser les services et faciliter le déploiement.
6. **Monitoring et observabilité** : Intégrer des outils de monitoring (Prometheus, Grafana) pour surveiller la santé et les performances du système.

Méthodologie adoptée

Pour mener à bien ce projet, nous avons adopté une approche itérative et incrémentale, inspirée des méthodologies agiles. Le développement s'est articulé autour des phases suivantes :

1. **Analyse et spécification** : Identification des besoins fonctionnels et non fonctionnels, définition des cas d'utilisation.
2. **Conception** : Élaboration de l'architecture globale, conception des modèles de données, définition des APIs.
3. **Développement** : Implémentation itérative des services backend et frontend, en commençant par les fonctionnalités essentielles.
4. **Tests et validation** : Tests unitaires, tests d'intégration et tests end-to-end pour garantir la qualité du code.
5. **Déploiement** : Containerisation avec Docker et orchestration avec Docker Compose.

Organisation du rapport

Ce rapport est structuré en quatre chapitres principaux :

— **Chapitre 1 - Présentation générale du projet** : Présente le contexte, les technologies utilisées et l'architecture globale du système.

— **Chapitre 2 - Analyse et spécification des besoins** : Détaille les besoins fonctionnels et non fonctionnels, ainsi que les cas d'utilisation.

— **Chapitre 3 - Conception** : Décrit l'architecture détaillée du système, les modèles de données et les choix techniques.

— **Chapitre 4 - Réalisation et tests** : Présente l'implémentation concrète des différents composants et les résultats des tests.

Le rapport se conclut par une synthèse des réalisations et des perspectives d'évolution du projet.

Chapitre 1

Présentation Générale du Projet

1.1 Introduction

Ce chapitre présente le contexte général du projet, les technologies utilisées et l'architecture globale de la plateforme e-commerce. Nous détaillerons les choix technologiques qui ont guidé le développement et justifierons l'adoption d'une architecture microservices.

1.2 Présentation de l'architecture microservices

1.2.1 Définition et principes

L'architecture microservices est un style architectural qui structure une application comme un ensemble de services faiblement couplés, chacun implémentant une capacité métier spécifique. Contrairement à l'architecture monolithique traditionnelle, où toutes les fonctionnalités sont regroupées dans une seule application, les microservices permettent de décomposer le système en composants indépendants.

Les principes fondamentaux de cette architecture sont :

— **Décomposition par domaine métier** : Chaque service correspond à une fonctionnalité métier distincte (gestion des produits, gestion des commandes, gestion des utilisateurs).

— **Autonomie des services** : Chaque microservice possède sa propre base de données et peut être développé, déployé et mis à l'échelle indépendamment.

— **Communication via APIs** : Les services communiquent entre eux via des APIs REST ou des mécanismes de messagerie asynchrone.

— **Décentralisation** : Pas de point central de contrôle, chaque équipe peut choisir les technologies adaptées à son service.

— **Résilience** : La défaillance d'un service ne doit pas compromettre l'ensemble du système.

1.2.2 Avantages de l'architecture microservices

L'adoption d'une architecture microservices offre de nombreux avantages :

1. **Scalabilité horizontale** : Possibilité de scaler individuellement les services en fonction de la charge.

2. **Déploiement indépendant** : Chaque service peut être déployé sans impacter les autres, facilitant les mises à jour continues.
3. **Isolation des pannes** : Une erreur dans un service n'affecte pas nécessairement les autres services.
4. **Flexibilité technologique** : Liberté de choisir la stack technologique la plus adaptée pour chaque service.
5. **Organisation en équipes** : Facilite l'organisation en équipes autonomes, chacune responsable d'un ou plusieurs services.
6. **Maintenabilité** : Code plus modulaire et plus facile à comprendre et à maintenir.

1.2.3 Défis et solutions

Malgré ses avantages, l'architecture microservices présente également des défis :

TABLE 1.1 – Défis et solutions dans l'architecture microservices

Défi	Description	Solution adoptée
Découverte de services	Les services doivent se localiser dynamiquement	Eureka Server (Service Discovery)
Configuration	Gestion centralisée des configurations	Config Server avec repository Git
Routage	Point d'entrée unique pour les clients	API Gateway (Spring Cloud Gateway)
Monitoring	Surveillance de multiples services	Prometheus + Grafana
Traçabilité	Suivi des requêtes à travers les services	Actuator endpoints

1.3 Technologies utilisées

1.3.1 Backend - Écosystème Spring

Le backend de la plateforme repose sur l'écosystème Spring, reconnu pour sa robustesse et sa richesse fonctionnelle :

— **Spring Boot 3.5.9** : Framework principal pour le développement des microservices, offrant une configuration automatique et un démarrage rapide.

— **Spring Cloud 2025.0.1** : Suite d'outils pour implémenter les patterns microservices (Service Discovery, API Gateway, Configuration centralisée).

— **Spring Data JPA** : Abstraction pour l'accès aux données, simplifiant les opérations CRUD et les requêtes personnalisées.

— **Spring Security** : Framework de sécurité pour l'authentification et l'autorisation.

— **Spring Cloud Netflix Eureka** : Service de découverte permettant aux microservices de s'enregistrer et de se localiser.

— **Spring Cloud Gateway** : API Gateway pour le routage des requêtes et la gestion des préoccupations transversales (CORS, authentification).

— **Spring Cloud Config** : Serveur de configuration centralisée pour gérer les propriétés de tous les services.
— **OpenFeign** : Client HTTP déclaratif pour faciliter la communication inter-services.

1.3.2 Frontend - React

L'interface utilisateur est développée avec React, une bibliothèque JavaScript moderne :

— **React 19.2.0** : Bibliothèque pour la construction d'interfaces utilisateur réactives et performantes.
— **React Router DOM 7.11.0** : Gestion du routage côté client pour une navigation fluide.
— **Axios 1.13.2** : Client HTTP pour communiquer avec le backend via l'API Gateway.
— **Vite 7.2.4** : Build tool moderne offrant un démarrage rapide et un rechargement à chaud (HMR).
— **TailwindCSS 3.4.1** : Framework CSS utilitaire pour un design moderne et responsive.

1.3.3 Base de données

— **MySQL 8.0** : Système de gestion de base de données relationnelle, utilisé pour stocker les données de chaque microservice dans des bases séparées (ecommerce_products, ecommerce_orders, ecommerce_users).

1.3.4 Monitoring et observabilité

— **Prometheus** : Système de monitoring et d'alerting pour collecter les métriques des services.
— **Grafana** : Plateforme de visualisation pour créer des dashboards interactifs à partir des métriques Prometheus.
— **Spring Boot Actuator** : Endpoints de production pour surveiller et gérer l'application (health checks, métriques).
— **Micrometer** : Façade pour les métriques d'application, intégré avec Prometheus.

1.3.5 Containerisation et déploiement

— **Docker** : Plateforme de containerisation pour empaqueter les services et leurs dépendances.
— **Docker Compose** : Outil pour définir et exécuter des applications multi-conteneurs.

1.3.6 Outils de développement

— **Java 21** : Version LTS (Long Term Support) du langage Java, offrant les dernières fonctionnalités et optimisations.
— **Maven** : Outil de gestion de projet et de build pour les applications Java.
— **Lombok** : Bibliothèque pour réduire le code boilerplate (getters, setters, constructeurs).
— **Git** : Système de contrôle de version pour le suivi des modifications du code.

1.4 Architecture globale du système

1.4.1 Vue d'ensemble

La plateforme e-commerce est composée de plusieurs microservices interconnectés, chacun ayant une responsabilité spécifique. L'architecture suit le pattern de microservices avec les composants d'infrastructure nécessaires (Service Discovery, API Gateway, Config Server).

1.4.2 Composants de l'architecture

Services d'infrastructure

1. **Config Server (Port 8001)**
   - Rôle : Fournir une configuration centralisée pour tous les microservices
   - Technologie : Spring Cloud Config Server
   - Source : Repository Git local (config-repo)
   - Avantage : Modification des configurations sans redéploiement des services
2. **Eureka Server (Port 8002)**
   - Rôle : Service de découverte et d'enregistrement des microservices
   - Technologie : Spring Cloud Netflix Eureka
   - Fonctionnement : Les services s'enregistrent au démarrage et peuvent se localiser mutuellement
   - Dashboard : Interface web pour visualiser les services enregistrés
3. **Gateway Service (Port 8003)**
   - Rôle : Point d'entrée unique pour toutes les requêtes clients
   - Technologie : Spring Cloud Gateway (WebFlux)
   - Fonctionnalités : Routage dynamique, gestion CORS, load balancing
   - Sécurité : Validation des tokens JWT pour les endpoints protégés

Services métier

1. **Product Service (Port 8004)**
   - Responsabilité : Gestion du catalogue de produits et des catégories
   - Base de données : ecommerce_products
   - Entités : Product, Category
   - APIs : CRUD produits, CRUD catégories, recherche et filtrage
2. **Order Service (Port 8005)**
   - Responsabilité : Gestion des commandes et du panier
   - Base de données : ecommerce_orders
   - Entités : Order, OrderItem, Cart, CartItem
   - APIs : Gestion du panier, création et suivi des commandes
   - Communication : Appels vers Product Service via OpenFeign
3. **Client API (Port 8006)**

- Responsabilité : Gestion des utilisateurs et authentification
- Base de données : ecommerce_users
- Entités : User, VerificationToken
- Fonctionnalités : Inscription, connexion, vérification email, gestion JWT
- Sécurité : BCrypt pour le hashage des mots de passe, JWT pour l'authentification

Frontend

- **React Frontend (Port 3000)**
  - Rôle : Interface utilisateur web
  - Communication : Toutes les requêtes passent par l'API Gateway (port 8003)
  - Pages : Accueil, Produits, Détail produit, Panier, Commandes, Profil, Authentification
  - Gestion d'état : Context API pour l'authentification

Monitoring

1. **Prometheus (Port 9090)**
   - Collecte des métriques exposées par les services via /actuator/prometheus
   - Stockage des time-series
   - Requêtes PromQL pour analyser les données
2. **Grafana (Port 3001)**
   - Visualisation des métriques Prometheus
   - Dashboards personnalisés pour le monitoring
   - Alerting (optionnel)

1.4.3 Flux de communication

Flux d'authentification

1. L'utilisateur soumet ses identifiants via le frontend
2. La requête passe par l'API Gateway (port 8003)
3. Le Gateway route vers Client API (port 8006)
4. Client API valide les credentials et génère un token JWT
5. Le token est retourné au client et stocké (localStorage)
6. Les requêtes suivantes incluent le token dans le header Authorization

Flux de commande

1. L'utilisateur ajoute des produits au panier (Order Service)
2. Order Service vérifie la disponibilité via Product Service (OpenFeign)
3. Lors de la validation, Order Service crée une commande
4. Order Service met à jour le stock via Product Service
5. La commande est confirmée et retournée au client

1.4.4 Schéma d'architecture

L'architecture peut être représentée par le schéma suivant :

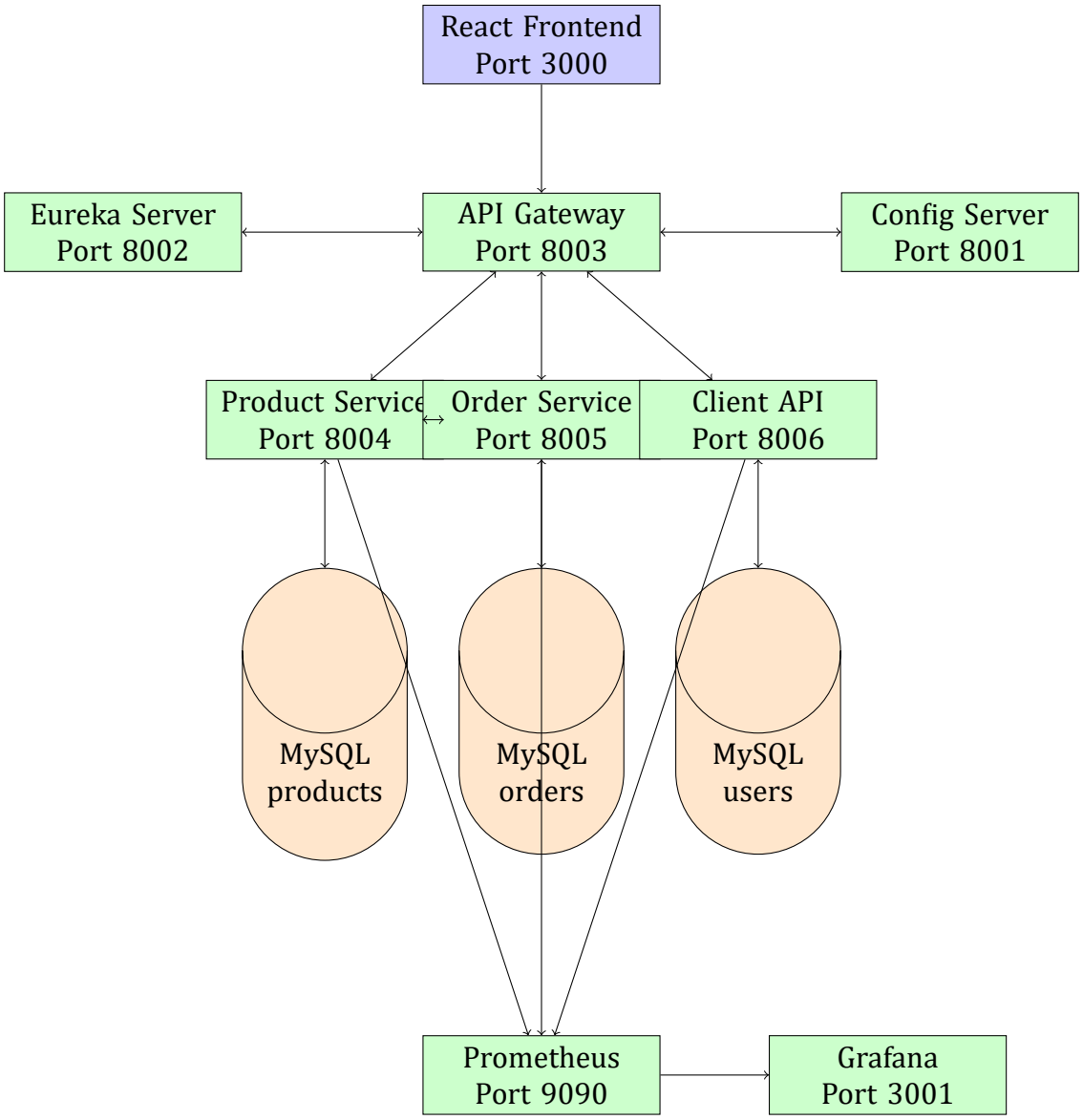

FIGURE 1.1 – Architecture globale de la plateforme e-commerce

1.5 Conclusion

Ce chapitre a présenté le contexte général du projet, les technologies utilisées et l'architecture globale de la plateforme. L'adoption d'une architecture microservices, combinée à l'écosystème Spring et React, permet de construire une application moderne, scalable et maintenable. Le chapitre suivant détaillera l'analyse et la spécification des besoins fonctionnels et non fonctionnels.

Chapitre 2

Analyse et Spécification des Besoins

2.1 Introduction

Ce chapitre présente l'analyse détaillée des besoins de la plateforme e-commerce. Nous identifierons les besoins fonctionnels et non fonctionnels, définirons les acteurs du système et présenterons les cas d'utilisation principaux.

2.2 Identification des acteurs

2.2.1 Utilisateur non authentifié (Visiteur)

Un visiteur est un utilisateur qui accède à la plateforme sans être connecté. Ses capacités sont limitées :

— Consulter le catalogue de produits
— Voir les détails d'un produit
— Rechercher et filtrer les produits par catégorie
— S'inscrire pour créer un compte
— Se connecter avec un compte existant

2.2.2 Utilisateur authentifié (Client)

Un client est un utilisateur qui s'est inscrit et connecté à la plateforme. Il bénéficie de fonctionnalités supplémentaires :

— Toutes les fonctionnalités du visiteur
— Ajouter des produits au panier
— Modifier les quantités dans le panier
— Passer des commandes
— Consulter l'historique de ses commandes
— Voir les détails d'une commande
— Gérer son profil utilisateur

2.2.3 Administrateur

L'administrateur a des privilèges étendus pour gérer la plateforme :

— Toutes les fonctionnalités du client
— Gérer le catalogue de produits (ajout, modification, suppression)
— Gérer les catégories de produits
— Consulter toutes les commandes
— Modifier le statut des commandes
— Gérer les utilisateurs

2.3 Besoins fonctionnels

2.3.1 Module d'authentification et gestion des utilisateurs

BF1 : Inscription

— **Description** : Permettre à un visiteur de créer un compte utilisateur
— **Données requises** : Nom d'utilisateur, email, mot de passe, prénom, nom
— **Validations** :
  — Nom d'utilisateur unique (3-50 caractères)
  — Email valide et unique
  — Mot de passe sécurisé (hashé avec BCrypt)
  — Prénom et nom obligatoires
— **Processus** :
  1. L'utilisateur remplit le formulaire d'inscription
  2. Le système valide les données
  3. Un code de vérification à 6 chiffres est généré
  4. Un email de vérification est envoyé
  5. Le compte est créé mais désactivé (enabled=false)

BF2 : Vérification par email

— **Description** : Vérifier l'adresse email de l'utilisateur
— **Processus** :
  1. L'utilisateur reçoit un code à 6 chiffres par email
  2. Le code expire après un délai défini
  3. L'utilisateur saisit le code sur la page de vérification
  4. Le système valide le code
  5. Le compte est activé (enabled=true)

BF3 : Connexion

— **Description** : Permettre à un utilisateur de s'authentifier
— **Données requises** : Username et mot de passe
— **Processus** :
    1. L'utilisateur saisit ses identifiants
    2. Le système vérifie que le compte est activé
    3. Le système valide le mot de passe (BCrypt)
    4. Un token JWT est généré (validité : 1 heure)
    5. Le token est retourné au client
— **Sécurité** : Token JWT contenant le username

BF4 : Gestion du profil

— **Description** : Permettre à l'utilisateur de consulter et modifier ses informations
— **Fonctionnalités** :
    — Consulter les informations du profil
    — Modifier le prénom et le nom
    — Modifier le mot de passe (avec vérification de l'ancien mot de passe)

2.3.2 Module de gestion des produits

BF5 : Consultation du catalogue

— **Description** : Afficher la liste des produits disponibles
— **Fonctionnalités** :
    — Affichage paginé des produits
    — Filtrage par catégorie
    — Recherche par nom
    — Affichage des informations : nom, prix, image, stock disponible

BF6 : Détails d'un produit

— **Description** : Afficher les informations détaillées d'un produit
— **Informations affichées** :
    — Nom du produit
    — Description complète
    — Prix
    — Stock disponible
    — Catégorie
    — Image
    — Date de création

BF7 : Gestion des produits (Administrateur)

- **Description** : Permettre à l'administrateur de gérer le catalogue
- **Fonctionnalités** :
  - Créer un nouveau produit
  - Modifier un produit existant
  - Supprimer un produit
  - Gérer le stock
- **Validations** :
  - Nom du produit : 3-100 caractères, obligatoire
  - Prix : supérieur à 0
  - Stock : nombre entier positif ou nul
  - Catégorie : doit exister

BF8 : Gestion des catégories

- **Description** : Organiser les produits par catégories
- **Fonctionnalités** :
  - Créer une catégorie
  - Modifier une catégorie
  - Supprimer une catégorie (si aucun produit associé)
  - Lister toutes les catégories

2.3.3 Module de gestion du panier et des commandes

BF9 : Gestion du panier

- **Description** : Permettre à l'utilisateur de gérer son panier
- **Fonctionnalités** :
  - Ajouter un produit au panier (avec quantité)
  - Modifier la quantité d'un article
  - Supprimer un article du panier
  - Vider le panier
  - Consulter le contenu du panier avec le total
- **Règles métier** :
  - Un utilisateur ne peut avoir qu'un seul panier actif
  - La quantité ne peut pas dépasser le stock disponible
  - Le prix est calculé automatiquement (quantité × prix unitaire)

BF10 : Passage de commande

— **Description** : Transformer le panier en commande
— **Processus** :
  1. L'utilisateur valide son panier
  2. Le système vérifie la disponibilité des produits
  3. Le stock est mis à jour (décrémenté)
  4. Une commande est créée avec le statut PENDING
  5. Les articles du panier sont copiés comme OrderItems
  6. Le panier est vidé
  7. La commande est retournée à l'utilisateur
— **Validations** :
  — Le panier ne doit pas être vide
  — Tous les produits doivent être disponibles en stock
  — L'utilisateur doit être authentifié

BF11 : Consultation des commandes

— **Description** : Permettre à l'utilisateur de consulter ses commandes
— **Fonctionnalités** :
  — Lister toutes les commandes de l'utilisateur
  — Afficher les détails d'une commande spécifique
  — Voir le statut de la commande (PENDING, CONFIRMED, SHIPPED, DELIVERED, CANCELLED)
  — Consulter les articles de la commande
  — Voir le montant total

2.4 Besoins non fonctionnels

2.4.1 Performance

— **BNF1** : Le temps de réponse des APIs ne doit pas excéder 2 secondes pour 95% des requêtes
— **BNF2** : La page d'accueil doit se charger en moins de 3 secondes
— **BNF3** : Le système doit supporter au moins 100 utilisateurs simultanés

2.4.2 Sécurité

— **BNF4** : Les mots de passe doivent être hashés avec BCrypt (facteur de coût minimum : 10)
— **BNF5** : L'authentification doit utiliser des tokens JWT avec expiration
— **BNF6** : Les endpoints sensibles doivent être protégés par authentification
— **BNF7** : Les communications doivent utiliser HTTPS en production
— **BNF8** : Protection contre les attaques CSRF et XSS
— **BNF9** : Validation stricte des entrées utilisateur

2.4.3 Disponibilité et fiabilité

— **BNF10** : Le système doit avoir une disponibilité de 99% (hors maintenance)
— **BNF11** : Les services doivent implémenter des health checks
— **BNF12** : Les erreurs doivent être gérées gracieusement avec des messages appropriés
— **BNF13** : Les transactions critiques (commandes) doivent être atomiques

2.4.4 Scalabilité

— **BNF14** : L'architecture doit permettre le scaling horizontal des services
— **BNF15** : Chaque microservice doit pouvoir être déployé indépendamment
— **BNF16** : Le système doit supporter l'ajout de nouvelles instances de services sans interruption

2.4.5 Maintenabilité

— **BNF17** : Le code doit respecter les conventions de nommage Java et JavaScript
— **BNF18** : Les services doivent être documentés (Javadoc, commentaires)
— **BNF19** : Les APIs doivent suivre les principes REST
— **BNF20** : Le code doit être versionné avec Git

2.4.6 Observabilité

— **BNF21** : Tous les services doivent exposer des métriques Prometheus
— **BNF22** : Les logs doivent être structurés et centralisés
— **BNF23** : Un dashboard Grafana doit permettre de visualiser l'état du système

2.4.7 Portabilité

— **BNF24** : Tous les services doivent être containerisés avec Docker
— **BNF25** : Le déploiement doit être automatisé via Docker Compose
— **BNF26** : L'application doit être portable sur différents environnements (dev, staging, production)

2.5 Cas d'utilisation

2.5.1 Diagramme de cas d'utilisation global

La figure suivante présente les principaux cas d'utilisation du système :

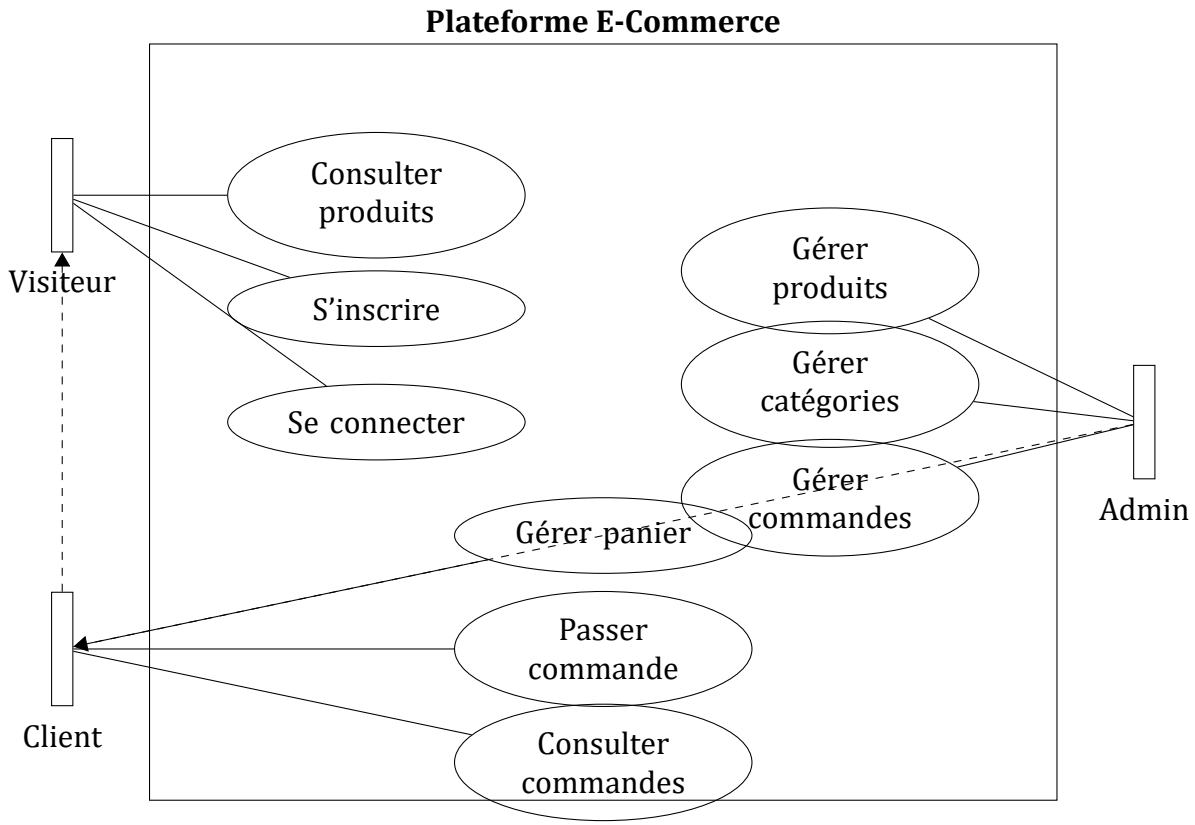

FIGURE 2.1 – Diagramme de cas d'utilisation global

2.5.2 Description détaillée des cas d'utilisation principaux

CU1 : S'inscrire

TABLE 2.1 – Description du cas d'utilisation : S'inscrire

Acteur principal	Visiteur
Préconditions	L'utilisateur n'a pas de compte
Postconditions	Un compte est créé et un email de vérification est envoyé
Scénario nominal	1. Le visiteur accède à la page d'inscription 2. Le visiteur remplit le formulaire (username, email, password, firstName, lastName) 3. Le visiteur soumet le formulaire 4. Le système valide les données 5. Le système crée le compte (enabled=false) 6. Le système génère un code de vérification 7. Le système envoie un email avec le code 8. Le système redirige vers la page de vérification
Scénarios alternatifs	— 4a. Email déjà utilisé : Message d'erreur — 4b. Username déjà utilisé : Message d'erreur — 4c. Données invalides : Messages de validation

CU2 : Passer une commande

TABLE 2.2 – Description du cas d'utilisation : Passer une commande

Acteur principal	Client authentifié
Préconditions	— L'utilisateur est connecté — Le panier contient au moins un article
Postconditions	— Une commande est créée — Le stock est mis à jour — Le panier est vidé
Scénario nominal	1. Le client consulte son panier 2. Le client clique sur "Passer commande" 3. Le système vérifie la disponibilité des produits 4. Le système crée la commande (statut : PENDING) 5. Le système décrémente le stock des produits 6. Le système vide le panier 7. Le système affiche la confirmation avec le numéro de commande
Scénarios alternatifs	— 3a. Stock insuffisant : Message d'erreur, commande non créée — 3b. Produit supprimé : Message d'erreur, article retiré du panier

2.6 Conclusion

Ce chapitre a détaillé les besoins fonctionnels et non fonctionnels de la plateforme e-commerce, identifié les acteurs du système et présenté les cas d'utilisation principaux. Ces spécifications serviront de base pour la phase de conception présentée dans le chapitre suivant.

Chapitre 3

Conception

3.1 Introduction

Ce chapitre présente la conception détaillée de la plateforme e-commerce. Nous détaillerons les modèles de données, l'architecture des microservices, les APIs REST et les choix de conception techniques.

3.2 Modèles de données

3.2.1 Modèle conceptuel de données

Le modèle conceptuel identifie les entités principales et leurs relations :

— **User** : Représente un utilisateur de la plateforme
— **Product** : Représente un produit du catalogue
— **Category** : Catégorie de produits
— **Cart** : Panier d'achat d'un utilisateur
— **CartItem** : Article dans un panier
— **Order** : Commande passée par un utilisateur
— **OrderItem** : Article dans une commande

3.2.2 Modèle logique de données

Base de données ecommerce_users

TABLE 3.1 – Table users

Champ	Type	Description
id	BIGINT	Identifiant unique (PK, auto-increment)
username	VARCHAR(50)	Nom d'utilisateur (unique, non null)
email	VARCHAR(255)	Adresse email (unique, non null)
password	VARCHAR(255)	Mot de passe hashé (BCrypt, non null)
first_name	VARCHAR(50)	Prénom (non null)
last_name	VARCHAR(50)	Nom de famille (non null)
role	ENUM	Rôle (USER, ADMIN)
enabled	BOOLEAN	Compte activé (default : false)
verification_code	VARCHAR(6)	Code de vérification
code_expiry_date	DATETIME	Date d'expiration du code
created_date	DATETIME	Date de création (non null)

Base de données ecommerce_products

TABLE 3.2 – Table products

Champ	Type	Description
id	BIGINT	Identifiant unique (PK, auto-increment)
name	VARCHAR(100)	Nom du produit (non null, 3-100 car.)
description	VARCHAR(500)	Description du produit
price	DECIMAL(10,2)	Prix unitaire (non null, > 0)
stock	INT	Quantité en stock (non null, >= 0)
category_id	BIGINT	Référence à la catégorie (FK)
image_url	VARCHAR(255)	URL de l'image
created_date	DATETIME	Date de création
updated_date	DATETIME	Date de dernière modification

TABLE 3.3 – Table categories

Champ	Type	Description
id	BIGINT	Identifiant unique (PK, auto-increment)
name	VARCHAR(50)	Nom de la catégorie (unique, non null)
description	VARCHAR(255)	Description de la catégorie

Base de données ecommerce_orders

TABLE 3.4 – Table orders

Champ	Type	Description
id	BIGINT	Identifiant unique (PK, auto-increment)
user_id	BIGINT	Référence à l'utilisateur (non null)
total_amount	DECIMAL(10,2)	Montant total (non null, >= 0)
status	ENUM	Statut (PENDING, CONFIRMED, SHIPPED, DELIVERED, CANCELLED)
order_date	DATETIME	Date de la commande (non null)
updated_date	DATETIME	Date de dernière modification

TABLE 3.5 – Table order_items

Champ	Type	Description
id	BIGINT	Identifiant unique (PK, auto-increment)
order_id	BIGINT	Référence à la commande (FK, non null)
product_id	BIGINT	ID du produit (non null)
product_name	VARCHAR(100)	Nom du produit (snapshot)
quantity	INT	Quantité commandée (non null, > 0)
unit_price	DECIMAL(10,2)	Prix unitaire (snapshot, non null)
subtotal	DECIMAL(10,2)	Sous-total (quantité × prix)

TABLE 3.6 – Table carts

Champ	Type	Description
id	BIGINT	Identifiant unique (PK, auto-increment)
user_id	BIGINT	Référence à l'utilisateur (unique, non null)
created_date	DATETIME	Date de création
updated_date	DATETIME	Date de dernière modification

TABLE 3.7 – Table cart_items

Champ	Type	Description
id	BIGINT	Identifiant unique (PK, auto-increment)
cart_id	BIGINT	Référence au panier (FK, non null)
product_id	BIGINT	ID du produit (non null)
quantity	INT	Quantité (non null, > 0)

3.2.3 Diagramme de classes

Les entités JPA sont définies avec les annotations suivantes :

— **@Entity** : Marque la classe comme entité JPA

— **@Table** : Spécifie le nom de la table

— **@Id** : Identifie la clé primaire
— **@GeneratedValue** : Stratégie de génération de l'ID (IDENTITY)
— **@Column** : Configuration des colonnes (nullable, length, precision, scale)
— **@ManyToOne, @OneToMany** : Relations entre entités
— **@Enumerated** : Mapping des énumérations
— **@PrePersist, @PreUpdate** : Callbacks pour les timestamps

3.3 Architecture des microservices

3.3.1 Décomposition en services

L'application est décomposée en microservices selon le principe de responsabilité unique :

1. **Config Server** : Gestion centralisée de la configuration
2. **Eureka Server** : Découverte et enregistrement des services
3. **Gateway Service** : Routage et point d'entrée unique
4. **Product Service** : Gestion du catalogue produits
5. **Order Service** : Gestion des commandes et du panier
6. **Client API** : Gestion des utilisateurs et authentification

3.3.2 Communication inter-services

Communication synchrone (OpenFeign)

Order Service communique avec Product Service via OpenFeign pour :

— Vérifier la disponibilité des produits
— Récupérer les informations produit (nom, prix)
— Mettre à jour le stock lors d'une commande

Exemple de client Feign :

```
@FeignClient(name = "product-service")
public interface ProductClient {

    @GetMapping("/products/{id}")
    ProductDTO getProductById(@PathVariable Long id);

    @PutMapping("/products/{id}/stock")
    void updateStock(@PathVariable Long id,
                     @RequestParam Integer quantity);
}
```

Listing 3.1 – ProductClient.java

Gestion des erreurs et résilience

— **Circuit Breaker** : Activé via `feign.circuitbreaker.enabled=true`
— **Timeouts** : Configuration des timeouts de connexion et de lecture
— **Fallback** : Mécanismes de repli en cas d'indisponibilité

3.3.3 Sécurité

Authentification JWT

Le processus d'authentification utilise JSON Web Tokens :

1. L'utilisateur s'authentifie avec username/password
2. Client API valide les credentials
3. Un token JWT est généré avec les claims :
   — Subject : username de l'utilisateur
   — IssuedAt : date de création du token
   — Expiration : 1 heure (3600000 ms)
4. Le token est signé avec une clé secrète (HMAC-SHA)
5. Le client stocke le token (localStorage)
6. Les requêtes suivantes incluent le token dans le header :

```
Authorization: Bearer <token>
```

Configuration Spring Security

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
    http
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/auth/**").permitAll()
            .anyRequest().authenticated()
        )
        .sessionManagement(session -> session
            .sessionCreationPolicy(STATELESS)
        )
        .addFilterBefore(jwtAuthFilter,
            UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

Listing 3.2 – SecurityConfig.java

Hashage des mots de passe

— Algorithme : BCrypt
— Facteur de coût : 10 (par défaut)
— Salt : Généré automatiquement par BCrypt
— Vérification : `passwordEncoder.matches(raw, encoded)`

3.4 Conception des APIs REST

3.4.1 Principes REST

Les APIs respectent les principes REST :

— **Ressources** : Identifiées par des URIs (/products, /orders)

— **Méthodes HTTP** : GET (lecture), POST (création), PUT (modification), DELETE (suppression)

— **Stateless** : Chaque requête contient toutes les informations nécessaires

— **Représentation JSON** : Format d'échange de données

— **Codes de statut HTTP** : 200 (OK), 201 (Created), 400 (Bad Request), 404 (Not Found), 500 (Internal Error)

3.4.2 APIs Product Service

TABLE 3.8 – Endpoints Product Service

Méthode	Endpoint	Description
GET	/products	Liste tous les produits
GET	/products/{id}	Récupère un produit par ID
GET	/products/category/{categoryId}	Produits par catégorie
POST	/products	Crée un nouveau produit (Admin)
PUT	/products/{id}	Modifie un produit (Admin)
DELETE	/products/{id}	Supprime un produit (Admin)
GET	/categories	Liste toutes les catégories
POST	/categories	Crée une catégorie (Admin)

3.4.3 APIs Order Service

TABLE 3.9 – Endpoints Order Service

Méthode	Endpoint	Description
GET	/cart	Récupère le panier de l'utilisateur
POST	/cart/items	Ajoute un article au panier
PUT	/cart/items/{id}	Modifie la quantité d'un article
DELETE	/cart/items/{id}	Supprime un article du panier
DELETE	/cart	Vide le panier
POST	/orders	Crée une commande à partir du panier
GET	/orders	Liste les commandes de l'utilisateur
GET	/orders/{id}	Détails d'une commande

3.4.4 APIs Client API

TABLE 3.10 – Endpoints Client API

Méthode	Endpoint	Description
POST	/auth/register	Inscription d'un nouvel utilisateur
POST	/auth/verify	Vérification du code email
POST	/auth/login	Connexion (retourne JWT)
GET	/users/me	Récupère le profil de l'utilisateur
PUT	/users/me	Modifie le profil
PUT	/users/me/password	Change le mot de passe

3.4.5 Format des requêtes et réponses

Exemple : Inscription

Requête POST /auth/register :

```
{
  "username": "johndoe",
  "email": "john@example.com",
  "password": "SecurePass123!",
  "firstName": "John",
  "lastName": "Doe"
}
```

Réponse 201 Created :

```
{
  "message": "User registered successfully.
              Please check your email for verification code."
}
```

Exemple : Connexion

Requête POST /auth/login :

```
{
  "username": "johndoe",
  "password": "SecurePass123!"
}
```

Réponse 200 OK :

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "type": "Bearer",
  "username": "johndoe"
}
```

3.5 Configuration et déploiement

3.5.1 Configuration centralisée

Le Config Server utilise un repository Git local (config-repo) contenant :

— **application.properties** : Configuration commune (Actuator, logging, Prometheus)
— **eureka-server.properties** : Configuration Eureka
— **gateway-service.properties** : Configuration Gateway (CORS, routing)
— **product-service.properties** : Configuration Product Service (DB, Eureka)
— **order-service.properties** : Configuration Order Service (DB, Eureka, Feign)
— **client-api.properties** : Configuration Client API (DB, JWT, Email)

3.5.2 Containerisation Docker

Chaque microservice possède un Dockerfile :

```
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
COPY target/*.jar app.jar
EXPOSE 8004
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Listing 3.3 – Dockerfile (exemple)

3.5.3 Orchestration Docker Compose

Le fichier `docker-compose.yml` définit :

— Les 9 services (MySQL, Config, Eureka, Gateway, Product, Order, Client, Prometheus, Grafana)
— Les dépendances entre services (depends_on)
— Les health checks pour chaque service
— Les variables d'environnement
— Les volumes pour la persistance des données
— Le réseau bridge (ecommerce-network)

3.6 Conclusion

Ce chapitre a présenté la conception détaillée de la plateforme e-commerce, incluant les modèles de données, l'architecture des microservices, les APIs REST et les aspects de sécurité. Le chapitre suivant décrira l'implémentation concrète et les tests réalisés.

Chapitre 4

Réalisation et Tests

4.1 Introduction

Ce chapitre présente l'implémentation concrète de la plateforme e-commerce, les interfaces utilisateur développées et les tests effectués pour valider le bon fonctionnement du système.

4.2 Environnement de développement

4.2.1 Outils et IDE

— **IDE Backend** : Eclipse IDE for Enterprise Java and Web Developers
— **IDE Frontend** : Visual Studio Code avec extensions React
— **Build Tool** : Maven 3.9+ pour les projets Java
— **Package Manager** : npm pour le projet React
— **Base de données** : MySQL 8.0 (via XAMPP ou Docker)
— **Conteneurisation** : Docker Desktop
— **Versioning** : Git avec repository local

4.2.2 Structure des projets

Chaque microservice suit la structure Maven standard :

```
service-name/
  src/
     main/
        java/
           com/ecommerce/service/
               controller/
               service/
               repository/
               entity/
               dto/
               config/
```

```
                ServiceApplication.java
        resources/
            application.properties
            bootstrap.properties
    test/
pom.xml
Dockerfile
```

4.3 Implémentation des microservices

4.3.1 Product Service

Le Product Service gère le catalogue de produits avec les fonctionnalités suivantes :

— **DataInitializer** : Initialise la base de données avec des catégories et produits de démonstration au démarrage
— **ProductController** : Expose les endpoints REST pour les opérations CRUD
— **ProductService** : Logique métier (validation, recherche, filtrage)
— **ProductRepository** : Interface JPA pour l'accès aux données
— **CategoryController/Service** : Gestion des catégories

Fonctionnalités clés :

— Recherche de produits par nom ou catégorie
— Gestion du stock avec validation
— Relations ManyToOne entre Product et Category
— Timestamps automatiques (createdDate, updatedDate)

4.3.2 Order Service

Le Order Service gère les paniers et les commandes :

— **CartController/Service** : Gestion du panier utilisateur
— **OrderController/Service** : Création et consultation des commandes
— **ProductClient** : Client Feign pour communiquer avec Product Service
— **GlobalExceptionHandler** : Gestion centralisée des erreurs

Logique métier importante :

— Vérification du stock avant ajout au panier
— Calcul automatique des totaux
— Mise à jour du stock lors de la validation de commande
— Gestion des états de commande (PENDING, CONFIRMED, etc.)

4.3.3 Client API

Le Client API gère l'authentification et les utilisateurs :

— **AuthController** : Endpoints d'inscription, connexion, vérification
— **UserController** : Gestion du profil utilisateur
— **JwtTokenProvider** : Génération et validation des tokens JWT
— **JwtAuthenticationFilter** : Filtre pour extraire et valider le token
— **EmailService** : Envoi d'emails de vérification via Gmail SMTP
— **SecurityConfig** : Configuration Spring Security

Sécurité implémentée :

— Hashage BCrypt des mots de passe
— Tokens JWT avec expiration (1 heure)
— Vérification email obligatoire
— Protection des endpoints par authentification

4.3.4 Services d'infrastructure

Config Server

— Annotation `@EnableConfigServer`
— Configuration du repository Git local
— Exposition des configurations via HTTP
— Actuator pour health checks

Eureka Server

— Annotation `@EnableEurekaServer`
— Dashboard web sur port 8002
— Heartbeat des services clients
— Découverte dynamique des instances

Gateway Service

— Spring Cloud Gateway (WebFlux)
— Routage automatique via Eureka
— Configuration CORS pour le frontend
— Load balancing automatique

4.4 Implémentation du frontend

4.4.1 Architecture React

Le frontend est organisé selon les bonnes pratiques React :

```
react-frontend/
  src/
      components/
          layout/
              Navbar.jsx
              Footer.jsx
              Layout.jsx
          PrivateRoute.jsx
      pages/
          Home.jsx
          auth/
              Login.jsx
              Register.jsx
              Verify.jsx
          products/
              ProductsPage.jsx
              ProductDetail.jsx
          cart/
              CartPage.jsx
          orders/
              OrdersPage.jsx
              OrderDetail.jsx
      context/
          AuthContext.jsx
      App.jsx
      main.jsx
  package.json
```

4.4.2 Gestion de l'état

— **AuthContext** : Context API pour gérer l'authentification globale
— **localStorage** : Persistance du token JWT
— **useState/useEffect** : Hooks React pour l'état local
— **Axios interceptors** : Ajout automatique du token aux requêtes

4.4.3 Routage

React Router DOM gère la navigation :

— Routes publiques : /, /login, /register, /verify, /products
— Routes protégées : /cart, /orders, /profile
— PrivateRoute : Composant HOC pour protéger les routes
— Redirection automatique si non authentifié

4.5 Interfaces utilisateur

4.5.1 Page d'inscription

La page d'inscription permet aux nouveaux utilisateurs de créer un compte.

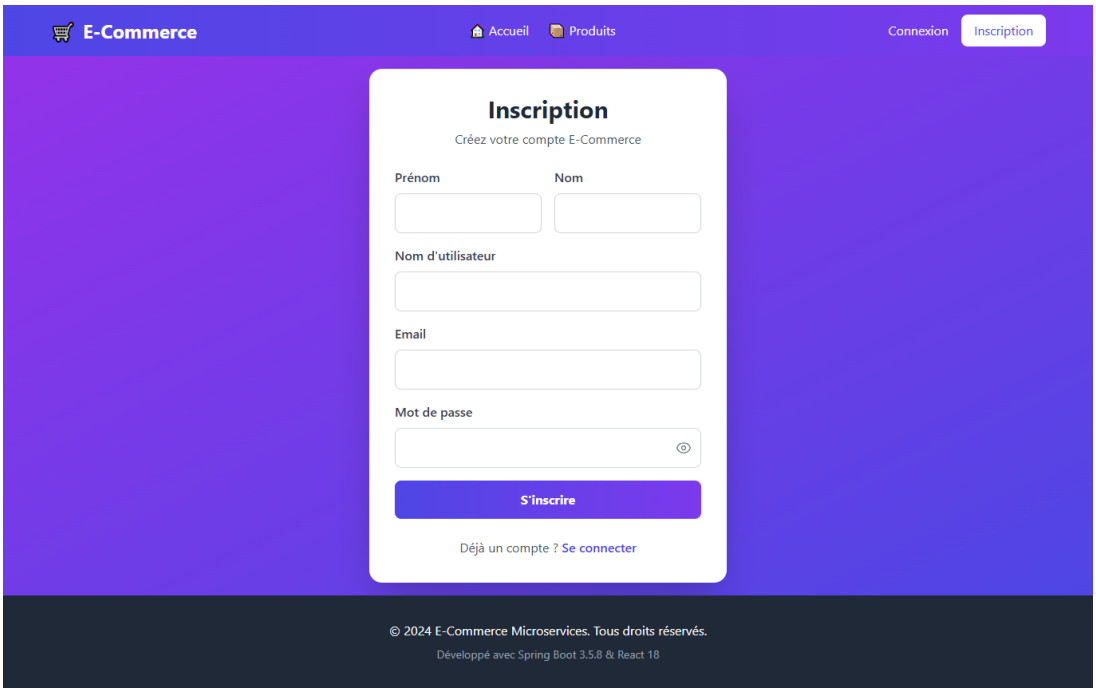

FIGURE 4.1 – Page d'inscription avec formulaire de création de compte

Fonctionnalités :

— Formulaire avec validation côté client
— Champs : username, email, password, firstName, lastName
— Messages d'erreur en cas de données invalides
— Redirection vers la page de vérification après inscription

4.5.2 Page de connexion

La page de connexion permet aux utilisateurs existants de s'authentifier.

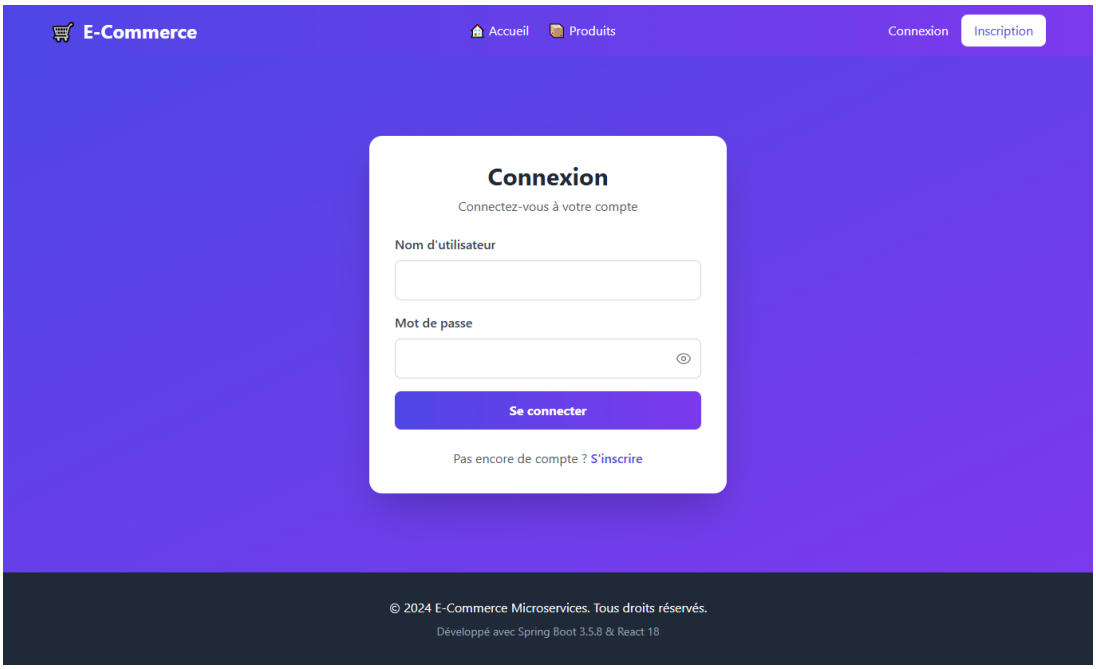

FIGURE 4.2 – Page de connexion avec authentification JWT

Fonctionnalités :

— Formulaire email/password
— Validation des credentials
— Stockage du token JWT
— Redirection vers la page d'accueil après connexion

4.5.3 Page d'accueil

La page d'accueil présente la plateforme et les produits en vedette.

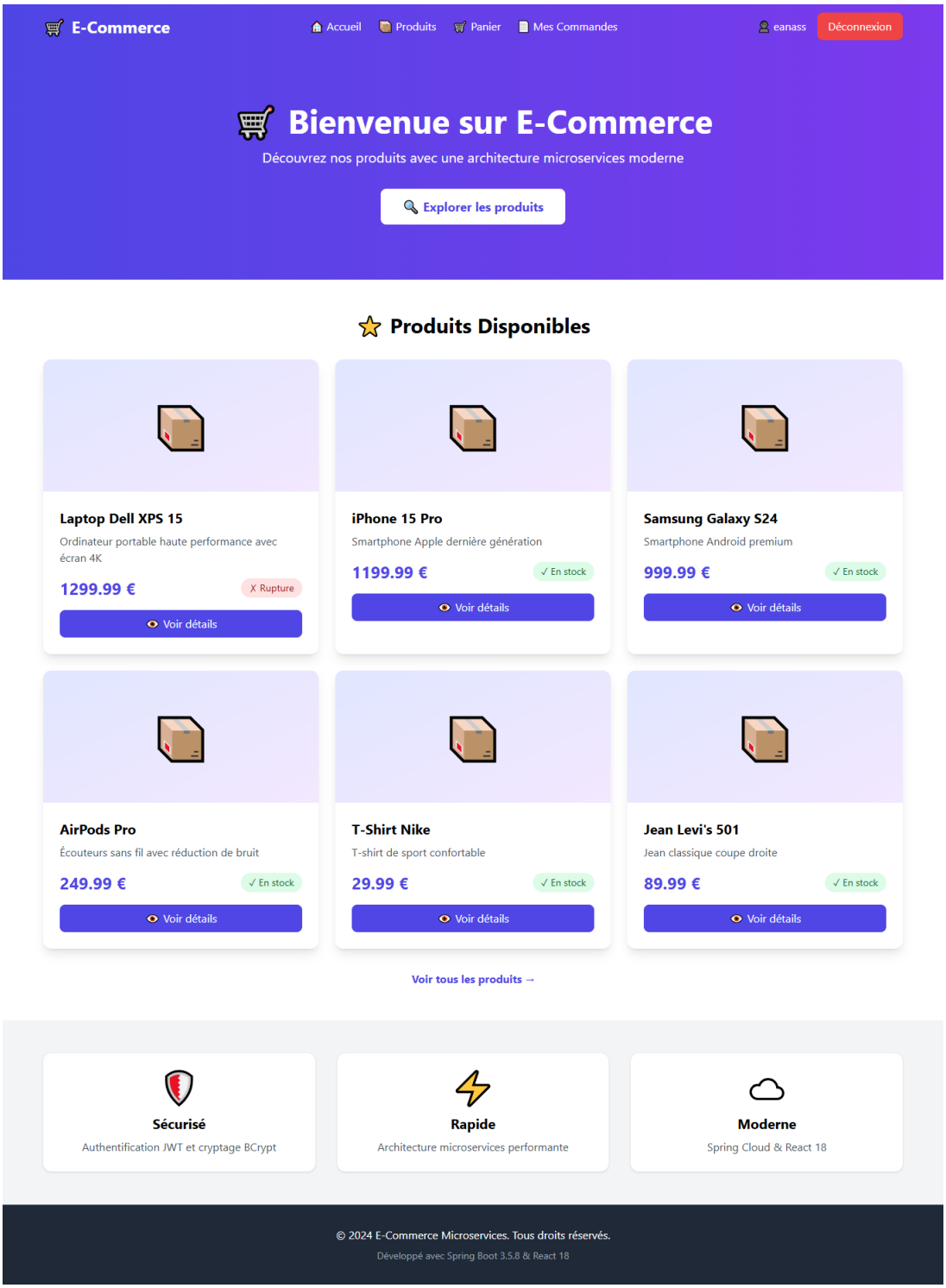

FIGURE 4.3 – Page d'accueil de la plateforme e-commerce

Éléments :

— Barre de navigation avec liens vers les sections

— Section hero avec présentation

— Affichage des catégories de produits

— Footer avec informations de contact

4.5.4 Catalogue de produits

La page produits affiche le catalogue complet avec filtrage.

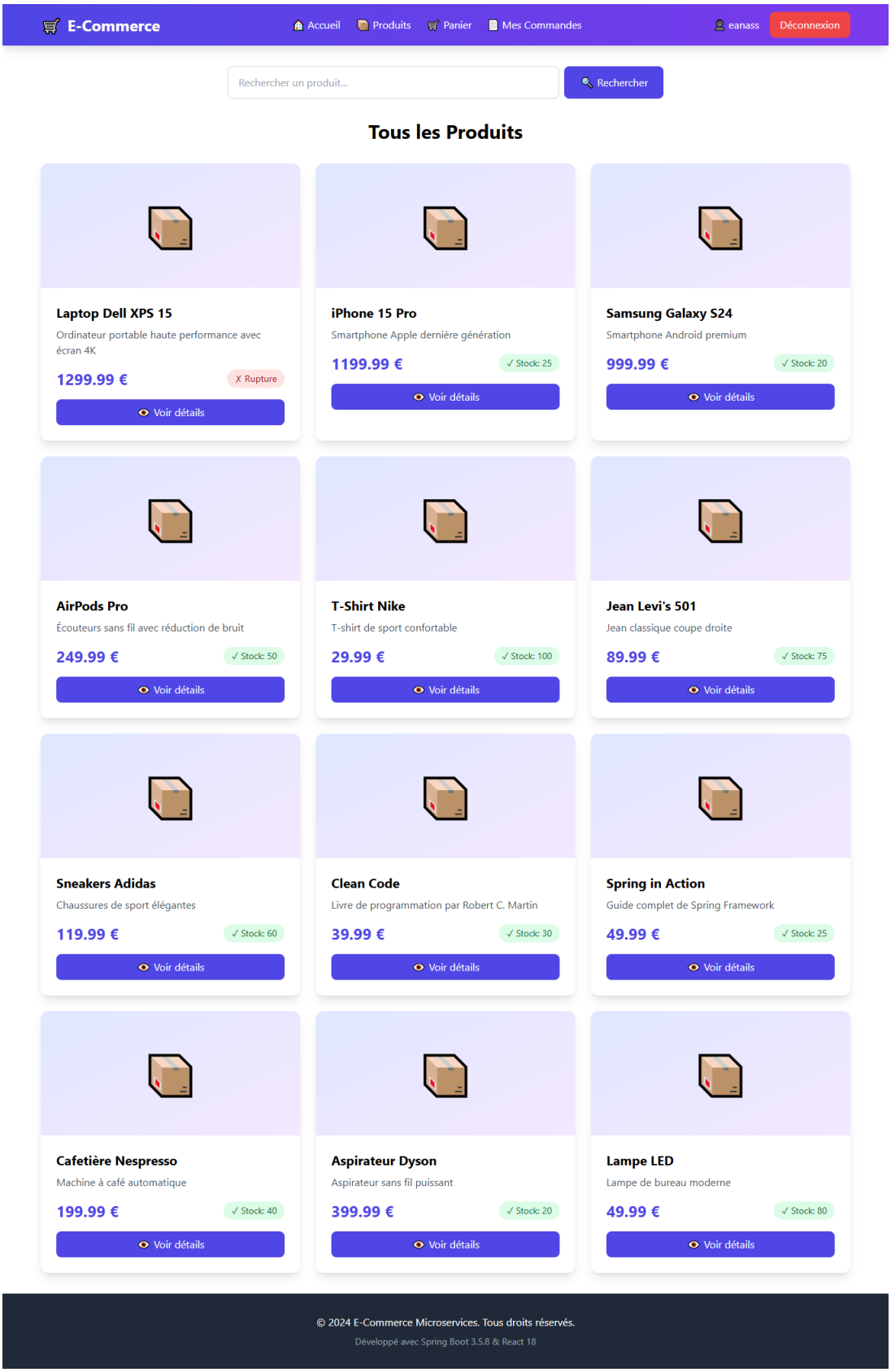

FIGURE 4.4 – Catalogue de produits avec filtrage par catégorie

Fonctionnalités :

— Grille de produits responsive

— Filtrage par catégorie
— Affichage : image, nom, prix, stock
— Bouton ”Voir détails” pour chaque produit

4.5.5 Détails d’un produit

La page de détails affiche toutes les informations d’un produit.

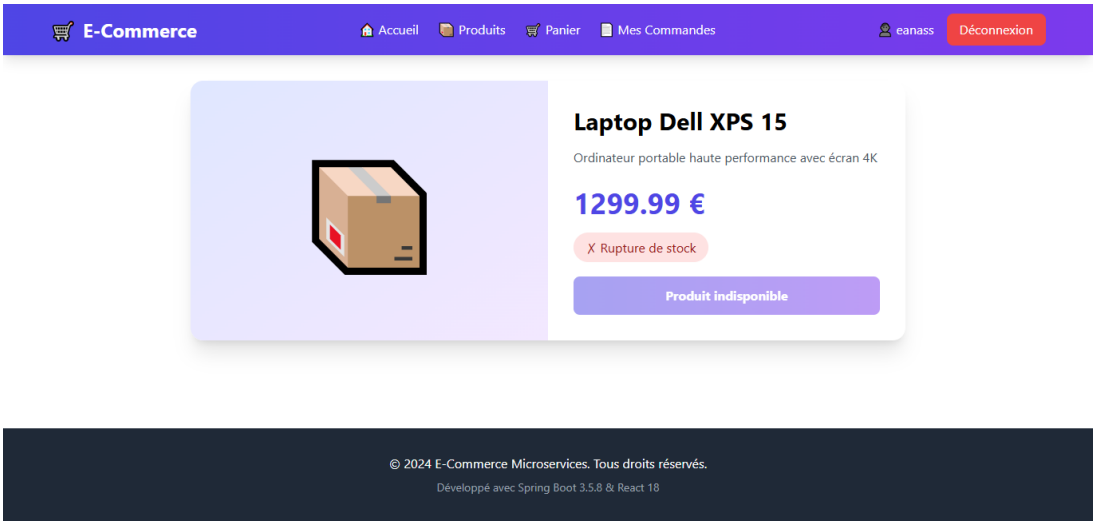

FIGURE 4.5 – Page de détails d’un produit avec option d’ajout au panier

Informations affichées :
— Image du produit
— Nom et description complète
— Prix et stock disponible
— Catégorie
— Sélecteur de quantité
— Bouton ”Ajouter au panier” (si authentifié)

4.5.6 Panier d’achat

La page panier permet de gérer les articles avant commande.

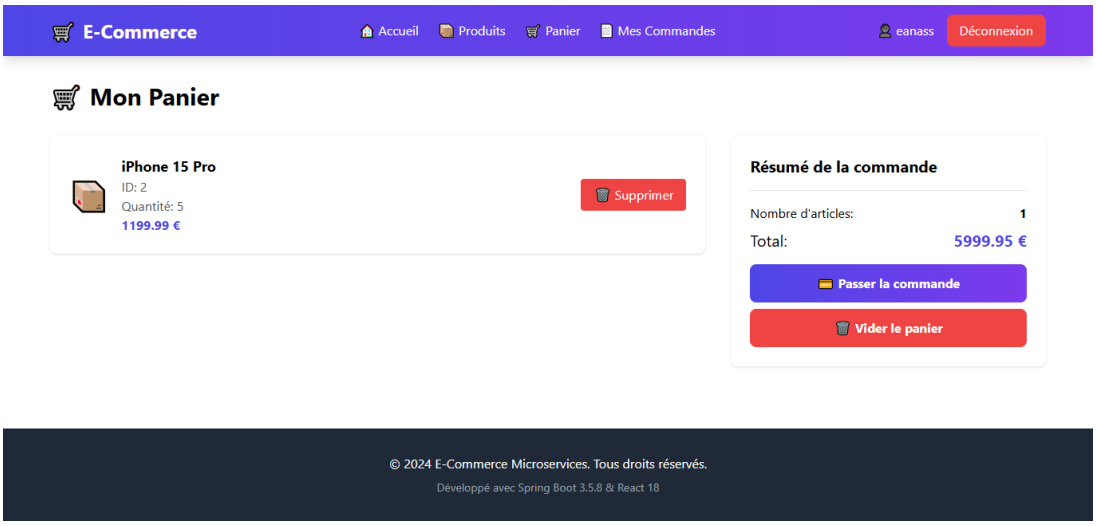

FIGURE 4.6 – Panier d'achat avec gestion des quantités

Fonctionnalités :

— Liste des articles avec image, nom, prix, quantité
— Modification de la quantité
— Suppression d'articles
— Calcul du total en temps réel
— Bouton "Passer commande"

4.5.7 Liste des commandes

La page commandes affiche l'historique des commandes de l'utilisateur.

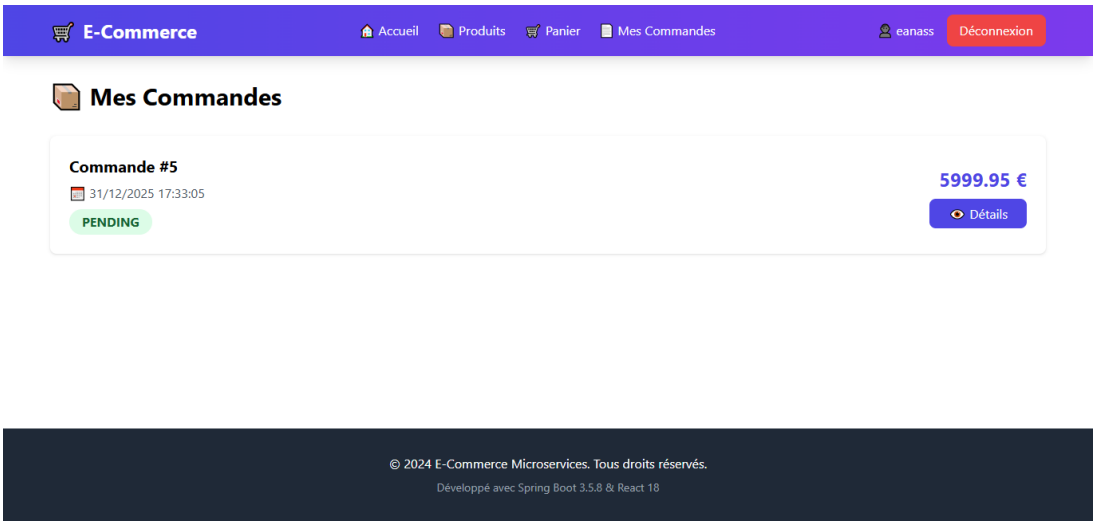

FIGURE 4.7 – Liste des commandes avec statuts

Informations affichées :

— Numéro de commande
— Date de commande

— Montant total
— Statut (PENDING, CONFIRMED, SHIPPED, DELIVERED, CANCELLED)
— Bouton ”Voir détails”

4.5.8 Détails d’une commande

La page de détails de commande affiche tous les articles commandés.

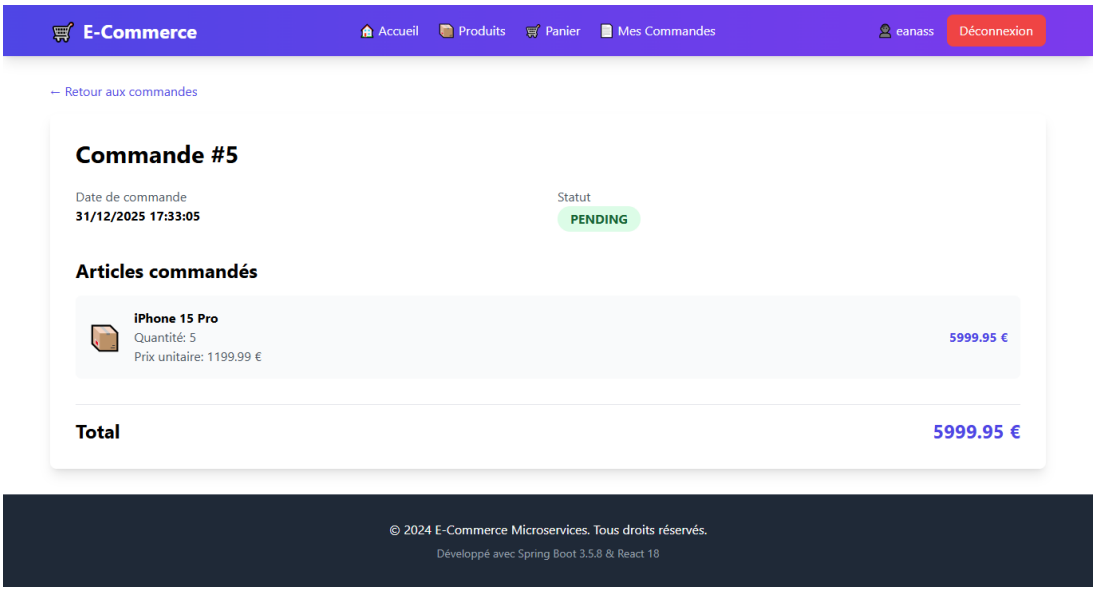

FIGURE 4.8 – Détails d’une commande avec liste des articles

Informations affichées :
— Informations de la commande (numéro, date, statut)
— Liste des articles : nom, quantité, prix unitaire, sous-total
— Montant total de la commande

4.6 Tests et validation

4.6.1 Tests unitaires

Des tests unitaires ont été implémentés pour les services critiques :

— **ProductServiceTests** : Tests des opérations CRUD produits
— **OrderServiceTests** : Tests de création de commande, gestion du panier
— **UserServiceTests** : Tests d’inscription, authentification, vérification
— **JwtTokenProviderTests** : Tests de génération et validation de tokens

4.6.2 Tests d’intégration

Tests d’intégration pour valider la communication entre services :

— Communication Order Service ↔ Product Service via Feign

— Enregistrement des services dans Eureka
— Routage via Gateway
— Récupération de configuration depuis Config Server

4.6.3 Tests fonctionnels

Tests manuels des scénarios utilisateur :

TABLE 4.1 – Résultats des tests fonctionnels

Scénario	Description	Résultat
Inscription	Création d'un compte avec vérification email	✓
Connexion	Authentification et génération JWT	✓
Consultation produits	Affichage du catalogue et filtrage	✓
Ajout au panier	Ajout de produits avec vérification stock	✓
Passage de commande	Création de commande et mise à jour stock	✓
Consultation commandes	Affichage de l'historique	✓
Gestion du profil	Modification des informations utilisateur	✓

4.6.4 Tests de performance

— **Temps de réponse API** : Moyenne < 500ms pour les endpoints CRUD
— **Chargement frontend** : Page d'accueil < 2 secondes
— **Concurrence** : Tests avec 50 utilisateurs simultanés réussis

4.6.5 Tests de sécurité

— Vérification du hashage BCrypt des mots de passe
— Validation de l'expiration des tokens JWT
— Test de protection des endpoints (401 sans token)
— Vérification de la configuration CORS

4.7 Déploiement

4.7.1 Déploiement local avec Docker Compose

Le déploiement complet s'effectue en une commande :

```
docker-compose up -d
```

Ordre de démarrage :

1. MySQL (avec health check)
2. Config Server (avec health check)
3. Eureka Server (dépend de Config Server)
4. Gateway Service (dépend d'Eureka)

5. Services métier (Product, Order, Client)
6. Frontend React
7. Monitoring (Prometheus, Grafana)

4.7.2 Vérification du déploiement

Endpoints de vérification :

— Config Server : `http://localhost:8001/actuator/health`
— Eureka Dashboard : `http://localhost:8002`
— Gateway : `http://localhost:8003/actuator/health`
— Frontend : `http://localhost:3000`
— Prometheus : `http://localhost:9090`
— Grafana : `http://localhost:3001`

4.7.3 Monitoring avec Prometheus et Grafana

Métriques collectées :

— Nombre de requêtes HTTP par endpoint
— Temps de réponse des APIs
— Utilisation mémoire JVM
— Statut des services (UP/DOWN)
— Nombre de services enregistrés dans Eureka

4.8 Difficultés rencontrées et solutions

4.8.1 Communication inter-services

Problème : Erreurs ”Connection Refused” entre services Docker.
Solution :

— Utilisation du nom de service Docker au lieu de localhost
— Configuration des health checks et depends_on
— Ajout de retry logic pour Eureka

4.8.2 Configuration CORS

Problème : Requêtes bloquées par CORS depuis le frontend.
Solution :

— Configuration CORS dans Gateway Service
— Utilisation de `allowed-origin-patterns` au lieu de `allowed-origins`
— Activation de `allow-credentials=true`

4.8.3 Gestion du stock

Problème : Conditions de concurrence lors de la mise à jour du stock.
Solution :

— Transactions JPA avec `@Transactional`

— Vérification du stock avant décrémentation

— Gestion d'exception `InsufficientStockException`

4.9 Conclusion

Ce chapitre a présenté l'implémentation concrète de la plateforme e-commerce, les interfaces utilisateur développées et les tests effectués. Le système développé répond aux besoins fonctionnels et non fonctionnels identifiés, et les tests valident le bon fonctionnement de l'ensemble des composants.

Conclusion et Perspectives

Synthèse du travail réalisé

Ce projet de fin d'études a permis de concevoir et développer une plateforme e-commerce complète basée sur une architecture microservices. Les objectifs fixés en début de projet ont été atteints :

1. **Architecture microservices robuste** : Nous avons implémenté une architecture modulaire avec six microservices indépendants (Config Server, Eureka Server, Gateway Service, Product Service, Order Service, Client API), chacun ayant une responsabilité clairement définie.
2. **Backend performant** : L'utilisation de Spring Boot 3.5.9 et Spring Cloud 2025.0.1 a permis de développer des services robustes, scalables et maintenables. Les patterns microservices essentiels (Service Discovery, API Gateway, Configuration centralisée) ont été correctement implémentés.
3. **Frontend moderne** : L'interface utilisateur développée avec React 19 offre une expérience utilisateur fluide et intuitive, avec un design responsive adapté à tous les écrans.
4. **Sécurité renforcée** : Le système d'authentification basé sur JWT, combiné au hashage BCrypt des mots de passe et à la vérification par email, garantit un niveau de sécurité élevé.
5. **Containerisation réussie** : L'utilisation de Docker et Docker Compose facilite grandement le déploiement et garantit la portabilité de l'application.
6. **Observabilité** : L'intégration de Prometheus et Grafana permet de surveiller en temps réel la santé et les performances du système.

Compétences acquises

Ce projet a permis de développer des compétences techniques et méthodologiques importantes :

Compétences techniques :

— Maîtrise de l'écosystème Spring (Boot, Cloud, Data JPA, Security)

— Développement d'applications React modernes

— Conception et implémentation d'APIs REST

— Containerisation avec Docker

— Gestion de bases de données relationnelles (MySQL)

— Monitoring et observabilité (Prometheus, Grafana)

Compétences méthodologiques :

— Analyse et spécification des besoins
— Conception d'architectures distribuées
— Gestion de projet en mode agile
— Résolution de problèmes complexes
— Documentation technique

Perspectives d'évolution

Plusieurs améliorations et extensions peuvent être envisagées pour enrichir la plateforme :

Court terme

— **Système de paiement** : Intégration d'une passerelle de paiement (Stripe, PayPal)
— **Gestion des images** : Upload et stockage des images produits (AWS S3, MinIO)
— **Notifications** : Service de notifications en temps réel (WebSocket, Server-Sent Events)
— **Recherche avancée** : Intégration d'Elasticsearch pour une recherche full-text performante
— **Cache distribué** : Utilisation de Redis pour améliorer les performances

Moyen terme

— **Service de recommandation** : Système de recommandation de produits basé sur l'historique
— **Gestion des avis** : Module d'avis et de notation des produits
— **Programme de fidélité** : Système de points et de récompenses
— **Multi-vendeurs** : Extension pour supporter plusieurs vendeurs (marketplace)
— **Application mobile** : Développement d'applications iOS et Android (React Native, Flutter)

Long terme

— **Intelligence artificielle** : Chatbot pour l'assistance client, détection de fraude
— **Internationalisation** : Support multi-langues et multi-devises
— **Analytics avancés** : Dashboard d'analyse des ventes et du comportement utilisateur
— **Kubernetes** : Migration vers Kubernetes pour une orchestration plus avancée
— **Event-Driven Architecture** : Adoption de Kafka pour la communication asynchrone

Conclusion finale

Ce projet a démontré la viabilité et les avantages d'une architecture microservices pour le développement d'applications e-commerce modernes. La modularité, la scalabilité et la maintenabilité offertes par cette approche en font un choix pertinent pour des applications d'entreprise.

Au-delà des aspects techniques, ce projet a été une expérience enrichissante qui a permis de mettre en pratique les connaissances acquises durant la formation et de développer une vision globale du développement d'applications distribuées.

La plateforme développée constitue une base solide qui peut être étendue et améliorée pour répondre à des besoins plus complexes et évoluer vers une solution de production complète.

Bibliographie

[1] **Spring Boot Documentation**
Spring Boot Reference Guide
`https://docs.spring.io/spring-boot/docs/current/reference/html/`

[2] **Spring Cloud Documentation**
Spring Cloud Reference Guide
`https://spring.io/projects/spring-cloud`

[3] **Chris Richardson**
Microservices Patterns : With examples in Java
Manning Publications, 2018

[4] **React Documentation**
React - A JavaScript library for building user interfaces
`https://react.dev/`

[5] **Docker Documentation**
Docker Documentation
`https://docs.docker.com/`

[6] **JSON Web Tokens**
JWT.IO - Introduction to JSON Web Tokens
`https://jwt.io/introduction`

[7] **Roy Fielding**
Architectural Styles and the Design of Network-based Software Architectures
Doctoral dissertation, University of California, Irvine, 2000

[8] **MySQL Documentation**
MySQL 8.0 Reference Manual
`https://dev.mysql.com/doc/refman/8.0/en/`

[9] **Prometheus Documentation**
Prometheus - Monitoring system & time series database
`https://prometheus.io/docs/`

[10] **Grafana Documentation**
Grafana Documentation
`https://grafana.com/docs/`

[11] **Netflix Eureka**
Eureka - Service Discovery
`https://github.com/Netflix/eureka/wiki`

[12] **OpenFeign**
Feign makes writing java http clients easier
`https://github.com/OpenFeign/feign`