



ENSA KHOURIBGA

INFORMATIQUE ET INGÉNIERIE DES DONNÉES

Conception et Implémentation d'une Plateforme d'Analytique et de Bidding pour la Grande Distribution

Réalisé par :

Achraf ZAHID

Med Amine EL ARBANI

Encadré par :

M. GHERABI

Résumé

Ce projet présente **Analify**, une plateforme complète d'analytique et de bidding pour la grande distribution. Analify permet aux entreprises de centraliser et d'exploiter leurs données métier (ventes, stocks, commandes) à travers un dashboard interactif, un système d'encheres pour les emplacements de rayons, et un assistant analytique basé sur l'intelligence artificielle.

Architecture technique :

- **Backend** : Spring Boot 3.4.13, Java 21, PostgreSQL, Spring Security (JWT), Spring AI
- **Frontend** : React 18, TypeScript, Vite, Tailwind CSS, shadcn/ui
- **Déploiement** : Docker Compose, multi-stage builds, orchestration de 4 services
- **Intelligence Artificielle** : Ollama (LLaMA 3.2:3b), accélération GPU, contexte étendu 8192 tokens

Fonctionnalités principales :

- Tableaux de bord analytiques personnalisés par rôle (Admin Global, Admin Magasin, Investisseur, Caissier)
- Système de bidding hiérarchique (Catégories → Rangs → Faces → Sections)
- Assistant conversationnel LLM pour requêtes en langage naturel
- Gestion complète des stocks, produits, commandes avec alertes automatiques
- Exports PDF et CSV, visualisations interactives, filtres avancés

Mots-clés : Analytics, Bidding, Spring Boot, React, Docker, LLM, JWT, PostgreSQL, Ollama, Intelligence Artificielle, Grande Distribution

Remerciements

Nous tenons à exprimer notre profonde gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce projet de fin d'études.

Nos remerciements s'adressent en premier lieu à **Monsieur GHERABI**, notre encadrant pédagogique, pour son soutien constant, ses conseils avisés et sa disponibilité tout au long de ce travail. Ses orientations nous ont permis de structurer efficacement notre démarche et d'approfondir nos compétences techniques.

Nous remercions également l'**École Nationale des Sciences Appliquées de Khouribga (ENSA Khouribga)** et l'ensemble du corps professoral de la filière **Informatique et Ingénierie des Données** pour la qualité de la formation dispensée et les connaissances solides acquises durant notre cursus.

Nous exprimons notre reconnaissance à la **communauté open source** et aux contributeurs des technologies utilisées dans ce projet (Spring Framework, React, Docker, Ollama, PostgreSQL, etc.) dont les outils et la documentation ont été essentiels à la réalisation d'Analify.

Enfin, nous adressons nos remerciements les plus sincères à nos **familles et proches** pour leur soutien moral, leur patience et leurs encouragements continus, qui nous ont permis de mener à bien ce projet dans les meilleures conditions.

*Achraf ZAHID & Med Amine EL ARBANI
Janvier 2026*

Contents

Résumé	1
Remerciements	2
Liste des figures	8
Introduction Générale	9
1 Contexte, objectifs et périmètre du projet	11
1.1 Contexte métier	11
1.2 Objectifs généraux	12
1.3 Périmètre fonctionnel	12
2 Architecture globale du système Analify	13
2.1 Vue d'ensemble front/back	13
2.2 Découpage en couches	14
2.3 Gestion des rôles et encapsulation des données	14
3 Conception détaillée du backend Spring Boot	16
3.1 Choix technologiques	16
3.2 Organisation du projet Maven	17
3.3 Architecture en couches	17
3.3.1 Package Controller	18
3.3.2 Package Service	19
3.3.3 Package Repository	19
3.3.4 Package Entity	20
3.3.5 DTO et mappers	20
3.4 Modèle de données : vue d'ensemble	21
3.5 Configuration de l'application	21
3.6 Gestion de la sécurité (aperçu)	22
3.7 Intégration de l'assistant LLM côté backend	22
3.8 Exemple de flux applicatif complet	23

4 Préparation et Transformation des Données	24
4.1 Introduction	24
4.2 Architecture de Transformation	24
4.2.1 Outils et Technologies	24
4.2.2 Connexion à la Base de Données	25
4.3 Normalisation et Création des Identifiants	25
4.3.1 Principe de Normalisation	25
4.3.2 Création des Identifiants Géographiques	25
4.3.3 Création des Identifiants Produits	26
4.3.4 Magasins (Stores)	27
4.4 Génération des Utilisateurs et Rôles	28
4.4.1 Stratégie de Génération	28
4.4.2 Génération de la Base Utilisateurs	28
4.4.3 Création des Caissiers	29
4.4.4 Création des Administrateurs de Magasin	30
4.4.5 Création des Investisseurs	30
4.4.6 Administrateur Global	30
4.5 Création des Données Transactionnelles	31
4.5.1 Commandes (Orders)	31
4.5.2 Articles de Commande (Order Items)	31
4.5.3 Inventaire (Inventory)	31
4.6 Création du Système de Bidding	32
4.6.1 Hiérarchie du Bidding	32
4.6.2 Rangs (Ranks)	32
4.6.3 Faces	33
4.6.4 Sections	33
4.7 Injection dans PostgreSQL	34
4.7.1 Méthode d'Injection	34
4.7.2 Ordre d'Injection	35
4.7.3 Volumétrie Finale	35
4.8 Validation et Vérifications	36
4.8.1 Contrôles de Cohérence	36
4.8.2 Exemples de Requêtes de Validation	36
4.9 Optimisations et Performances	37
4.9.1 Indexation	37
4.9.2 Taille de la Base de Données	37
4.10 Conclusion	37

5 Conception détaillée du frontend React/TypeScript	38
5.1 Stack technique et outillage	38
5.2 Structure du projet frontend	39
5.3 Routing et pages principales	39
5.4 Layouts et navigation	40
5.5 Gestion de l'authentification côté frontend	40
5.6 Service API centralisé	40
5.7 Composants métiers du dashboard	41
5.8 Gestion des filtres et de l'état	42
5.9 Interface du module de bidding	42
5.10 Intégration de l'assistant analytique LLM	43
5.11 Expérience utilisateur et responsive design	43
6 Module d'analytique et tableaux de bord	44
6.1 Objectifs du module d'analytique	44
6.2 Services statistiques côté backend	44
6.2.1 StatisticsService	45
6.2.2 EnhancedStatisticsService	45
6.3 DTO et structure des tableaux de bord	46
6.3.1 DashboardStatsDTO	46
6.3.2 EnhancedDashboardDTO	46
6.4 Endpoints REST liés aux statistiques	47
6.5 Rendu des tableaux de bord côté frontend	47
6.5.1 Tableau de bord de base	47
6.5.2 Tableau de bord avancé	48
6.6 Gestion des exports	48
6.7 Rôle de l'analytique pour le module de bidding	49
6.8 Lien avec l'assistant analytique LLM	49
7 Module de bidding (enchères sur sections)	51
7.1 Modèle de domaine du bidding	51
7.2 Logique métier des enchères	52
7.3 Endpoints REST du module de bidding	53
7.4 Interface utilisateur du bidding côté frontend	54
7.4.1 Navigation hiérarchique	54
7.4.2 Formulaire de placement d'enchère	54
7.4.3 Suivi des bids et indicateurs	55
7.5 Lien entre bidding et analytique	55
7.6 Intégration avec l'assistant analytique LLM	56

8 Sécurité, authentification et gestion des rôles	57
8.1 Objectifs de sécurité	57
8.2 Modèle de rôles fonctionnels	57
8.3 Authentification par JWT	58
8.4 Configuration Spring Security	58
8.5 Filtre JWT et propagation du rôle	59
8.6 Stratégies d'autorisation par rôle	60
8.6.1 Au niveau des endpoints	60
8.6.2 Au niveau des services et repositories	61
8.7 Sécurité côté frontend	61
8.8 Limitations et pistes d'amélioration	62
9 Assistant analytique LLM (Spring AI + Ollama)	63
9.1 Objectifs et principes de conception	63
9.2 Première version : intégration directe à Google Gemini	64
9.3 Refactorisation vers Spring AI	64
9.4 Passage à un LLM local via Ollama	65
9.5 Construction du contexte analytique	65
9.6 Gestion des erreurs et métadonnées de réponse	66
9.7 Interface utilisateur de l'assistant	67
9.7.1 Cycle de requête côté frontend	67
9.8 Respect du périmètre et confidentialité des données	68
9.9 Limites actuelles et perspectives d'évolution	68
10 Tests, validation et déploiement	69
10.1 Stratégie globale de tests	69
10.2 Tests unitaires du backend	69
10.3 Tests d'intégration REST	71
10.4 Tests et validation côté frontend	71
10.5 Validation de l'intégration LLM	72
10.6 Packaging et construction	72
10.6.1 Packaging manuel	72
10.6.2 Déploiement Docker (approche recommandée)	73
10.6.3 Configuration pour la production	74
10.7 Déploiement du backend (approche manuelle)	75
10.8 Déploiement du frontend (approche manuelle)	75
10.9 Enchaînement complet	76
10.9.1 Avec Docker (recommandé)	76
10.9.2 Approche manuelle (développement)	76

11 Captures d'écran et Démonstration de l'Interface	78
11.1 Page d'Accueil et Landing Page	78
11.1.1 Landing Page	78
11.2 Authentification et Connexion	79
11.2.1 Page de Connexion	79
11.3 Dashboard Principal - Vue d'Ensemble	80
11.3.1 Dashboard pour Administrateur Global	80
11.3.2 Dashboard pour Responsable de Magasin	81
11.3.3 Dashboard pour Investisseur	82
11.4 Module d'Analytique Avancée	82
11.4.1 Statistiques Enrichies	82
11.4.2 Panel de Filtres	83
11.5 Module de Gestion des Produits	84
11.5.1 Liste des Produits	84
11.6 Module de Gestion des Commandes	86
11.6.1 Liste des Commandes	86
11.6.2 Création d'une Nouvelle Commande	87
11.7 Module de Bidding - Système d'Enchères	87
11.7.1 Navigation par Catégories	87
11.7.2 Vue des Sections Disponibles	88
11.7.3 Placement d'une Enchère	89
11.7.4 Suivi des Enchères	90
11.8 Assistant Analytique LLM	90
11.8.1 Interface Chat	90
11.8.2 Exemples de Questions	92
11.9 Gestion des Utilisateurs et Rôles	92
11.9.1 Liste des Utilisateurs (Admin)	92
11.9.2 Profil Utilisateur	93
11.10 Exports et Rapports	94
11.10.1 Génération de Rapports PDF	94
11.10.2 Export CSV	95
11.11 Messages d'Erreur et Notifications	96
11.11.1 Gestion des Erreurs	96
11.11.2 Système de Notifications	97
11.12 Conclusion du Chapitre	97
Conclusion et perspectives	99

List of Figures

2.1	Architecture globale front/back de la plateforme Analify	13
3.1	Vue simplifiée du modèle de données backend	21
7.1	Vue simplifiée du modèle de domaine du bidding	52
11.1	Landing Page - Interface d'accueil de la plateforme Analify	78
11.2	Page de Connexion - Authentification JWT	79
11.3	Dashboard Administrateur Global - Vue d'ensemble complète	80
11.4	Dashboard Responsable de Magasin - Vue limitée à son périmètre	81
11.5	Dashboard Investisseur - Suivi de portefeuille	82
11.6	Statistiques Enrichies - Analytique avancée avec filtres	83
11.7	Panel de Filtres - Contrôle granulaire des visualisations	84
11.8	Liste des Produits - Gestion d'inventaire	85
11.9	Liste des Commandes - Suivi des ventes	86
11.10	Création de Commande - Interface caissier	87
11.11	Catégories de Bidding - Navigation hiérarchique	88
11.12	Sections Disponibles - Emplacements ouverts au bidding	88
11.13	Placement d'Enchère - Formulaire d'investissement	89
11.14	Mes Enchères - Suivi du portefeuille d'investissements	90
11.15	Assistant Analytique LLM - Interface conversationnelle	91
11.16	Exemples de Questions - Suggestions contextuelles	92
11.17	Gestion des Utilisateurs - Administration des comptes	93
11.18	Profil Utilisateur - Informations personnelles	94
11.19	Rapport PDF - Export professionnel	95
11.20	Export CSV - Données brutes pour tableau	95
11.21	Messages d'Erreur - Retour utilisateur explicite	96
11.22	Système de Notifications - Alertes temps réel	97

Introduction Générale

Dans le contexte actuel de la grande distribution et du retail, les entreprises disposent d'un volume de données considérable (ventes, stocks, commandes, performance des employés, etc.) mais peinent souvent à les exploiter de façon simple et centralisée. Le projet **Analify** vise à répondre à ce besoin en proposant une plateforme complète d'analytique métier et de gestion de bidding pour les espaces de rayon, tout en intégrant un assistant intelligent basé sur un LLM (Large Language Model) capable de répondre à des questions métier en langage naturel.

Ce rapport présente de manière détaillée la conception, l'architecture et l'implémentation d'Analify, depuis le **backend Spring Boot** (Java 21, Spring Boot 3.4.13, PostgreSQL, JWT, Spring Security, Spring AI, etc.) jusqu'au **frontend React/TypeScript** (Vite, Tailwind CSS, shadcn/ui), en passant par la couche d'analytique métier, le module de bidding, la gestion fine des rôles (ADMIN_G, ADMIN_STORE, INVESTOR, CAISSIER) et l'intégration d'un assistant analytique via LLM (Google Gemini puis modèle local via Ollama).

L'objectif de ce document est de fournir un **rappor complet (niveau mémoire de fin d'études)** couvrant :

- le contexte et les objectifs fonctionnels du projet ;
- l'analyse des besoins et la modélisation des données ;
- l'architecture logique et technique (front/back) ;
- le détail des principaux modules : statistiques, bidding, gestion des stocks, commandes ;
- la gestion de la sécurité et des rôles utilisateurs ;
- la conception et l'intégration de l'assistant analytique LLM ;
- les choix technologiques, les tests et la mise en production ;
- une conclusion et des perspectives d'évolution.

Chaque grande idée fonctionnelle ou technique est structurée sous forme de **chapitre**, afin de rendre la lecture claire et progressive. Le rapport est enrichi de **captures d'écran**, de schémas et d'**extraits de code** commentés, permettant de comprendre précisément le fonctionnement d'Analify de bout en bout.

Chapter 1

Contexte, objectifs et périmètre du projet

1.1 Contexte métier

La grande distribution moderne repose sur une gestion fine des stocks, des commandes, des promotions et de la mise en valeur des produits dans les rayons. Les enseignes disposent de multiples systèmes (ERP, caisses, outils de gestion de stock, etc.) qui produisent des « silos de données ». Cependant, les décideurs (direction, responsables de magasin, investisseurs) ont besoin d'une vision consolidée et de tableaux de bord pertinents pour piloter l'activité.

Par ailleurs, la monétisation des espaces de rayon et des sections via un mécanisme de *bidding* (enchères) par les investisseurs devient un levier important : les marques et investisseurs peuvent ainsi louer des emplacements à fort potentiel en fonction des performances historiques (ventes, trafic, marge, etc.).

Dans ce contexte, Analify propose :

- une couche d'analytique avancée (KPI, graphiques, tableaux, exports CSV/PDF) ;
- un module de bidding structuré par catégories, rangs, faces et sections ;
- un assistant LLM capable de répondre à des questions du type « Quels sont mes produits les plus rentables ce mois-ci ? » ou « Quels magasins ont le plus de ruptures de stock ? » ;
- une gestion des rôles fine, garantissant que chaque profil ne voit que son périmètre (administration globale, magasin, investisseur, caissier).

1.2 Objectifs généraux

Les objectifs principaux du projet sont les suivants :

- Centraliser les indicateurs clés de performance (KPI) sur une **interface web unique**.
- Offrir une **expérience utilisateur moderne** (dashboard réactif, filtres dynamiques, visualisation des données).
- Proposer un **moteur de bidding** pour la réservation de sections de rayons par les investisseurs.
- Intégrer un **assistant analytique LLM** qui exploite les données déjà agrégées, sans accès direct à la base.
- Garantir la **sécurité des données** via une authentification JWT et une gestion stricte des rôles.

1.3 Périmètre fonctionnel

Le périmètre fonctionnel couvert par ce rapport inclut :

- la gestion des utilisateurs et des rôles ;
- les tableaux de bord statistiques (basique et avancé) ;
- la gestion des produits, des stocks et des alertes de rupture ;
- le module de commandes et d'ordres de vente ;
- le module de bidding (catégories, rangs, faces, sections, enchères, suivi des bids) ;
- l'assistant analytique LLM (backend Spring AI + modèle local Ollama, frontend chat panel) ;
- les exports CSV et PDF ;
- le déploiement de l'application (backend + frontend).

Chapter 2

Architecture globale du système Analify

2.1 Vue d'ensemble front/back

Analify est structuré en deux sous-projets principaux :

- **backAnalify** : backend REST basé sur Spring Boot 3.4.13, Java 21 et PostgreSQL ;
- **frontAnalify** : frontend SPA basé sur React 18, TypeScript, Vite, Tailwind CSS et les composants shadcn/ui.

La communication entre les deux se fait exclusivement via des **API REST JSON** sécurisées par des tokens JWT. Le schéma de déploiement typique est le suivant :

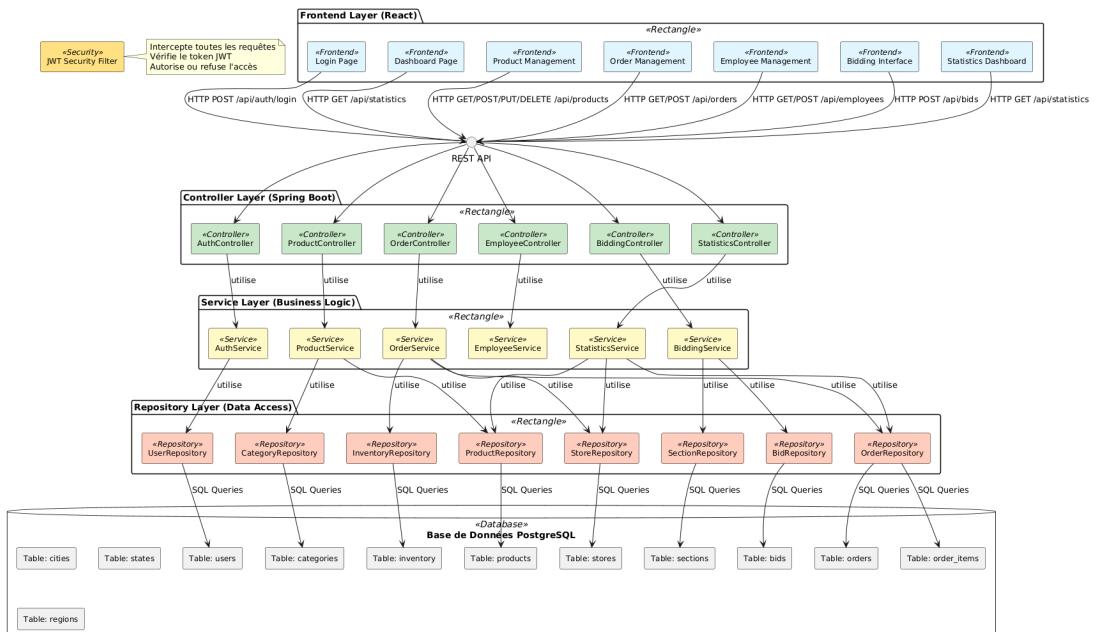


Figure 2.1: Architecture globale front/back de la plateforme Analify

Le backend expose des endpoints regroupés par **domaines métier** (authentification, statistiques, produits, commandes, bidding, assistant analytique, etc.) et le frontend consomme ces endpoints via un service centralisé `api.ts`.

2.2 Découpage en couches

La logique côté backend suit une architecture en couches classique :

- **Controller** : gestion des endpoints REST, validation des données d'entrée, mapping des DTO ;
- **Service** : logique métier (calculs de KPI, règles de bidding, filtrage par rôle, etc.) ;
- **Repository** : accès aux données (Spring Data JPA, requêtes personnalisées) ;
- **Entity/DTO/Mapper** : mapping entre entités JPA et objets de transfert (DTO) ;
- **Security** : JWT, filtres, configuration des rôles et des autorisations ;
- **Integration LLM** : service dédié à l'assistant analytique (Spring AI + Ollama/Gemini).

Du côté frontend, on distingue :

- une **couche de routing** (pages `Landing`, `Login`, `Dashboard`, etc.) ;
- des **layouts** (par ex. `DashboardLayout.tsx`) pour organiser la structure générale (sidebar, header, contenu, assistant) ;
- des **composants métiers** (`EnhancedStatistics.tsx`, `Products.tsx`, `BiddingDashboard.tsx`, etc.) ;
- des **composants UI réutilisables** (`StatCard`, `DataTable`, `FilterPanel`, etc.) ;
- un contexte d'authentification (`AuthContext.tsx`) ;
- un service d'accès API centralisé (`services/api.ts`).

2.3 Gestion des rôles et encapsulation des données

Un point clé de l'architecture est la **gestion stricte des rôles** :

- **ADMIN_G** : vision globale sur l'ensemble des magasins, des produits, des stocks et des performances ;

- **ADMIN_STORE** : vision limitée à son magasin (stocks, ventes, caissiers, sections) ;
- **INVESTOR** : vision limitée à ses propres produits et sections gagnées dans le système de bidding ;
- **CAISSIER** : vision très restreinte, principalement orientée « caisse et commandes ».

Cette encapsulation se retrouve dans **tous** les services de statistiques et dans l'assistant LLM : les méthodes reçoivent systématiquement l'identifiant de l'utilisateur et son rôle, puis appliquent les filtres adéquats.

Par exemple, côté backend, les services de statistiques exposent des signatures du type

:

Listing 2.1: Exemple de signature de service statistique filtré par rôle

```

1 public EnhancedDashboardDTO getEnhancedDashboard(Long userId,
2                                     UserRole role,
3                                     StatisticsFilterDTO filter) {
4     // Logique : filtre des données en fonction du rôle
5 }
```

Cette approche garantit que même si le frontend tentait d'appeler un endpoint avec un rôle inapproprié, le backend ne renverrait jamais des données hors périmètre.

Chapter 3

Conception détaillée du backend Spring Boot

Dans ce chapitre, nous détaillons la conception et l'implémentation du backend **Analify**, basé sur le framework **Spring Boot 3.4.13** et le langage **Java 21**. L'objectif est de montrer comment les différents modules (authentification, statistiques, bidding, assistant LLM, etc.) s'articulent autour d'une architecture en couches propre, maintenable et sécurisée.

3.1 Choix technologiques

Le backend repose sur les composants principaux suivants :

- **Spring Boot 3.4.13** : socle applicatif pour exposer des API REST, gérer la configuration et le cycle de vie de l'application ;
- **Java 21** : version LTS moderne du JDK, offrant de meilleures performances et des fonctionnalités de langage récentes ;
- **Spring Web** : pour l'exposition d'endpoints REST (contrôleurs annotés avec `@RestController`) ;
- **Spring Data JPA** : pour accéder à la base de données PostgreSQL via des entités JPA et des repositories ;
- **PostgreSQL** : SGBD relationnel utilisé pour stocker les données métier (utilisateurs, produits, stocks, commandes, sections, bids, etc.) ;
- **Spring Security + JWT** : pour l'authentification et l'autorisation basées sur des tokens JWT signés ;

- **Spring AI + Ollama** : pour l'intégration d'un LLM local, utilisé par l'assistant analytique ;
- **Lombok** : pour réduire le code boilerplate (getters/setters, constructeurs, `@Builder`, etc.).

La gestion du build est assurée par **Maven** via le fichier `pom.xml` du module `backAnalify`.

3.2 Organisation du projet Maven

Le projet backend se trouve dans le dossier `backAnalify/`. À la racine, on trouve notamment :

- `pom.xml` : configuration Maven (dépendances, plugins, version de Java, Spring Boot) ;
- `src/main/java/com/analyfy/analify/` : code source Java ;
- `src/main/resources/` : fichiers de configuration (`application.properties`) ;
- `src/test/java/` : éventuels tests unitaires et d'intégration.

Dans le fichier `pom.xml`, on retrouve les dépendances typiques d'une application Spring Boot REST :

- `spring-boot-starter-web` ;
- `spring-boot-starter-data-jpa` ;
- `spring-boot-starter-security` ;
- `postgresql` (driver JDBC) ;
- `jjwt` pour la gestion des JWT ;
- `spring-ai-ollama-spring-boot-starter` pour l'intégration du LLM local ;
- `lombok`, `validation-api`, etc.

3.3 Architecture en couches

Le backend suit une **architecture en couches** classique, qui sépare clairement les responsabilités :

- **Controller** : exposition des API REST ;

- **Service** : logique métier ;
- **Repository** : accès aux données (couche de persistance) ;
- **Entity** : mapping objet-relationnel (JPA) ;
- **DTO / Mapper** : objets de transfert pour le frontend ;
- **Security** : configuration Spring Security, filtres JWT ;
- **Integration LLM** : service dédié à l'assistant analytique.

3.3.1 Package Controller

Le package `controller` regroupe les différents contrôleurs REST, par exemple :

- `AuthController` : gestion de l'authentification (login, génération de JWT) ;
- `UserController` : gestion des utilisateurs (création, listing, etc.) ;
- `StatisticsController` : exposition des statistiques de base ;
- `EnhancedStatisticsController` ou équivalent : exposition du tableau de bord avancé ;
- `ProductController` : gestion des produits et stocks ;
- `OrderController` : gestion des commandes ;
- `BiddingController` : endpoints du module d'enchères ;
- `AnalyticsAssistantController` : endpoint de l'assistant analytique LLM.

Chaque contrôleur est annoté avec `@RestController` et `@RequestMapping("/api/...")`.

Il se contente de :

- récupérer les paramètres de la requête (corps JSON, paramètres de chemin, paramètres de requête) ;
- extraire l'identité de l'utilisateur et son rôle via des `@RequestAttribute` alimentés par le filtre JWT ;
- déléguer au service métier associé ;
- renvoyer la réponse sous forme de DTO, automatiquement serialisés en JSON.

3.3.2 Package Service

Le package `service` contient la **logique métier** d'Analify. On y trouve notamment :

- `StatisticsService` : calcul des KPI de base (CA, nombre de commandes, produits vendus, etc.) ;
- `EnhancedStatisticsService` : calcul du tableau de bord avancé (top produits, top magasins, analyses par rôle, etc.) ;
- `ProductService` : gestion des produits, du stock, des alertes low stock ;
- `OrderService` : gestion des commandes et de leurs états ;
- `BiddingService` : logique liée aux catégories, rangs, faces, sections et enchères ;
- `UserService` : gestion des utilisateurs et de leurs rôles ;
- `AnalyticsAssistantService` : construction du contexte analytique et appel au LLM via Spring AI.

Tous ces services sont annotés avec `@Service` et, lorsque cela est pertinent, `@Transactional` pour encadrer les opérations en base.

3.3.3 Package Repository

Le package `repository` contient les interfaces Spring Data JPA, par exemple :

- `UserRepository` ;
- `StoreRepository` ;
- `ProductRepository` ;
- `OrderRepository`, `OrderLineRepository` ;
- `CategoryRepository`, `SectionRepository`, `BidRepository` ;
- etc.

Chaque repository étend généralement `JpaRepository<Entity, Long>` et définit, si nécessaire, des méthodes de requêtage spécifiques (par ex. `findByIdAndStatus(...)`).

3.3.4 Package Entity

Le package `entity` regroupe les entités JPA qui représentent les tables de la base de données. Quelques entités clés :

- **User** : représente un utilisateur de la plateforme (administrateur global, administrateur de magasin, investisseur, caissier) ;
- **Store** : représente un magasin physique ;
- **Product** : produit vendu en magasin ;
- **Stock** : niveau de stock d'un produit dans un magasin donné ;
- **Order** et **OrderLine** : commandes et lignes de commandes ;
- **Category**, **Section**, **Bid** : entités principales du module de bidding ;
- **Role** ou énumération **UserRole** : rôle fonctionnel de l'utilisateur.

Les entités sont annotées avec `@Entity`, `@Table`, et possèdent des relations `@OneToMany`, `@ManyToMany`, etc. afin de modéliser les liens entre magasins, produits, sections et enchères.

3.3.5 DTO et mappers

Le backend expose des DTO (Data Transfer Objects) pour éviter de retourner directement les entités JPA au frontend. On trouve par exemple :

- `DashboardStatsDTO` : DTO pour le tableau de bord de statistiques de base ;
- `EnhancedDashboardDTO` : DTO pour le tableau de bord avancé ;
- `RankingItemDTO` : éléments de top produits, top magasins, etc. ;
- `SectionDTO`, `BidDTO` : objets pour le module de bidding ;
- `AnalyticsAssistantRequest` et `AnalyticsAssistantResponse` : DTO pour l'assistant LLM.

Des mappers (manuels ou générés) se chargent de transformer les entités en DTO et inversement lorsque nécessaire.

3.4 Modèle de données : vue d'ensemble

La base de données PostgreSQL est organisée autour de plusieurs **sous-domaines** :

Utilisateurs et rôles tables `users`, `roles` (ou rôle en tant que champ enum), association éventuelle utilisateur -`store` pour les administrateurs de magasin ;

Magasins table `stores` avec les informations de localisation, taille, etc. ;

Produits et stocks tables `products`, `stocks` (stock par produit et par magasin), catégories de produits ;

Commandes tables `orders`, `order_lines` pour tracer chaque vente ;

Bidding tables `categories`, `sections`, `bids` pour représenter la hiérarchie de rayon et les enchères ;

Statistiques agrégées agrégations réalisées à la volée par les services, plutôt que stockées de façon redondante.

Une représentation schématique de ce modèle peut être insérée sous forme de diagramme ER :

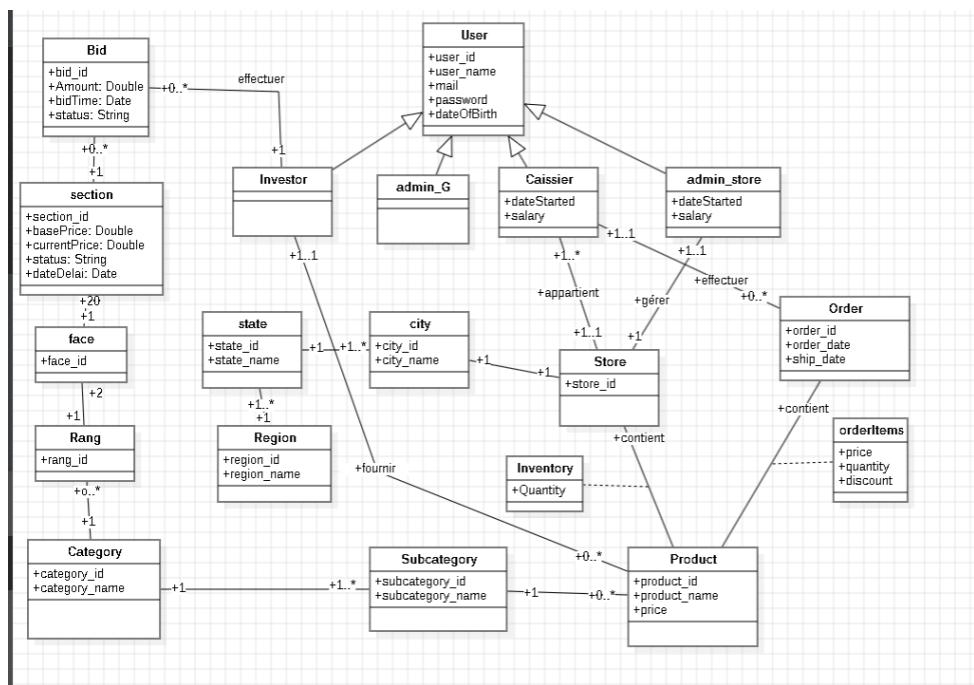


Figure 3.1: Vue simplifiée du modèle de données backend

3.5 Configuration de l'application

La configuration se fait principalement via le fichier `application.properties` :

- `server.port=8081` : port HTTP du backend ;
- propriétés de connexion Postgres (URL, utilisateur, mot de passe) ;
- `spring.jpa.hibernate.ddl-auto` (souvent `update` pour le développement) ;
- propriétés liées au JWT (secret, expiration) ;
- propriétés `spring.ai.ollama.*` pour pointer vers le serveur Ollama local et le modèle à utiliser.

Cette configuration est externalisable (variables d'environnement, fichiers de configuration par profil) afin d'adapter facilement le déploiement à différents environnements (développement, test, production).

3.6 Gestion de la sécurité (aperçu)

La sécurité détaillée est décrite dans un chapitre dédié, mais nous donnons ici un aperçu de son intégration dans le backend :

- Spring Security est configuré pour sécuriser tous les endpoints sous `/api/**`, à l'exception de ceux d'authentification (login, éventuellement inscription) ;
- un filtre JWT intercepte chaque requête, valide le token, construit un `Authentication` Spring, et ajoute à la requête les attributs `userId` et `role` ;
- les contrôleurs et services peuvent ensuite exploiter ces informations pour filtrer les données (par exemple, ne retourner que les sections appartenant à un investisseur donné).

Ce mécanisme est essentiel pour garantir que les statistiques et le module de bidding respectent le **périmètre de visibilité** de chaque profil.

3.7 Intégration de l'assistant LLM côté backend

Le backend expose un endpoint pour l'assistant analytique, via `AnalyticsAssistantController`, qui délègue à `AnalyticsAssistantService`. Ce service :

1. récupère le contexte utilisateur (ID, rôle) ;
2. interroge les services de statistiques (de base et avancées) et de produits pour construire un **résumé textuel** des données pertinentes ;
3. forge un prompt en langage naturel, combinant la question de l'utilisateur et le contexte ;

4. appelle Spring AI (`ChatClient`) pour obtenir une réponse du modèle Ollama ;
5. encapsule la réponse et des métadonnées (par exemple le nombre de produits en low stock) dans un DTO `AnalyticsAssistantResponse`.

L'intérêt d'encapsuler toute la logique LLM dans un service dédié est double :

- le reste du backend n'a pas à connaître les détails d'implémentation (Gemini vs Ollama, Spring AI, etc.) ;
- il est possible de faire évoluer le modèle ou la stratégie de prompt sans impacter les contrôleurs ni les DTO.

3.8 Exemple de flux applicatif complet

Pour illustrer la coopération entre les différentes couches, considérons le scénario suivant : « Un investisseur consulte ses sections disponibles et place une enchère ».

1. L'investisseur se connecte via le frontend (page `Login`), qui appelle l'endpoint `/api/auth/login`. Le backend valide les identifiants, génère un JWT contenant l'ID utilisateur et le rôle `INVESTOR`, puis le renvoie au frontend.
2. Lorsqu'il accède à la page de bidding, le frontend appelle un endpoint du type `/api/bidding/sections` avec le token JWT en en-tête `Authorization`.
3. Le filtre JWT du backend valide le token, injecte `userId` et `role` dans la requête ; `BiddingController` lit ces attributs et les transmet à `BiddingService`.
4. `BiddingService` interroge les repositories `SectionRepository` et `BidRepository` pour lister les sections éligibles à cet investisseur, puis renvoie un DTO listant les sections disponibles, leur état, les bids actuels, etc.
5. Lorsque l'investisseur place une enchère, une requête POST est envoyée vers `/api/bidding/bids`. Le service crée un nouvel objet `Bid`, vérifie les contraintes métier (section encore ouverte, montant minimal, etc.), sauvegarde la bid et renvoie un statut de succès.

Un scénario similaire peut être décrit pour l'assistant LLM : l'utilisateur tape une question dans le chat, le frontend appelle `/api/assistant/analytics/query`, le backend construit le contexte analytique et renvoie une réponse en langage naturel.

Ce chapitre a présenté la structure globale du backend Spring Boot d'Analify. Les chapitres suivants détaillent plus finement la conception du frontend, du module d'analytique et du module de bidding.

Chapter 4

Préparation et Transformation des Données

4.1 Introduction

La construction d'une plateforme d'analytique robuste nécessite des données de qualité, structurées et cohérentes. Ce chapitre présente le processus complet de transformation du fichier CSV brut (`superstore_dataset.csv`) en un ensemble de tables relationnelles exploitables dans notre base de données PostgreSQL.

Le dataset initial contient des données de ventes d'une chaîne de magasins, mais présente plusieurs défis typiques des données réelles : redondance, absence d'identifiants uniques, données non normalisées, et nécessité de créer des entités complémentaires (utilisateurs, rôles, inventaires).

4.2 Architecture de Transformation

4.2.1 Outils et Technologies

La transformation des données a été réalisée avec les technologies suivantes :

- **Python 3.x** : Langage de programmation principal
- **Pandas** : Manipulation et transformation des données
- **NumPy** : Génération de données numériques aléatoires
- **Faker** : Génération de données synthétiques (noms, emails, dates)
- **SQLAlchemy** : Connexion et injection dans PostgreSQL
- **Jupyter Notebook** : Environnement de développement interactif

4.2.2 Connexion à la Base de Données

La connexion à PostgreSQL a été établie via SQLAlchemy avec les paramètres suivants :

Listing 4.1: Configuration de la connexion PostgreSQL

```
1 from sqlalchemy import create_engine
2
3 engine = create_engine(
4     'postgresql+psycopg2://postgres:Admin@localhost:5432/analify'
5 )
```

4.3 Normalisation et Création des Identifiants

4.3.1 Principe de Normalisation

Le dataset original contient des colonnes textuelles redondantes (région, catégorie, produit, etc.). Pour respecter les principes de normalisation des bases de données relationnelles, nous avons :

1. Extrait chaque dimension unique
2. Crée des tables de référence avec identifiants auto-incrémentés
3. Fusionné les identifiants dans le dataset principal

4.3.2 Crédit des Identifiants Géographiques

Régions

Extraction des régions uniques et création des identifiants :

Listing 4.2: Normalisation des régions

```
1 df["region"] = df["region"].str.strip().str.lower()
2
3 df_region = (
4     df[["region"]]
5     .drop_duplicates()
6     .reset_index(drop=True)
7 )
8
9 df_region["region_id"] = df_region.index + 1
10 df = df.merge(df_region, on="region", how="left")
```

Résultat : Crédit de la table `region` avec 4 régions distinctes (East, West, Central, South).

États et Villes

Le même processus a été appliqué pour les états et les villes :

Listing 4.3: Normalisation des états

```
1 df["state"] = df["state"].str.strip().str.lower()
2
3 df_state = (
4     df[["state"]]
5     .drop_duplicates()
6     .reset_index(drop=True)
7 )
8
9 df_state["state_id"] = df_state.index + 1
10 df = df.merge(df_state, on="state", how="left")
```

Pour les villes, nous avons utilisé le code postal (zip) comme identifiant unique :

Listing 4.4: Normalisation des villes

```
1 city_df = df[["zip", "city", "state_id"]]
2 city_df = (
3     city_df
4     .groupby("city", as_index=False)
5     .agg({"zip": lambda x: x.mode().iloc[0]})
6 )
7
8 city_df = city_df.rename(columns={"zip": "city_id"})
```

4.3.3 Crédation des Identifiants Produits

Catégories et Sous-catégories

Les produits sont organisés hiérarchiquement en catégories et sous-catégories :

Listing 4.5: Hiérarchie des produits

```
1 # Categories
2 df["category"] = df["category"].str.strip().str.lower()
3 df_category = (
4     df[["category"]]
5     .drop_duplicates()
6     .reset_index(drop=True)
7 )
8 df_category["category_id"] = df_category.index + 1
9
10 # Subcategories
11 df["subcategory"] = df["subcategory"].str.strip().str.lower()
12 df_subcategory = (
```

```

13 df[["subcategory", "category_id"]]
14 .drop_duplicates()
15 .reset_index(drop=True)
16 )
17 df_subcategory["subcategory_id"] = df_subcategory.index + 1

```

Résultat : 3 catégories principales (Furniture, Technology, Office Supplies) et 17 sous-catégories.

Produits

Extraction des produits uniques avec association à leur sous-catégorie :

Listing 4.6: Table des produits

```

1 df["product_name"] = df["product_name"].str.strip().str.lower()
2
3 products_df = (
4     df[["product_id", "product_name", "subcategory_id"]]
5     .drop_duplicates(subset=['product_name'])
6     .reset_index(drop=True)
7 )

```

Résultat : 1852 produits distincts catalogués.

4.3.4 Magasins (Stores)

Les magasins ont été créés à partir des villes uniques. Chaque ville correspond à un magasin :

Listing 4.7: Création des magasins

```

1 df["store_id"] = (
2     df["city"]
3     .astype("category")
4     .cat.codes + 1
5 )
6
7 store_df = pd.DataFrame({
8     'store_id': df["store_id"].unique(),
9     'city_id': city_df['city_id'].unique()
10 })

```

Résultat : 531 magasins répartis dans différentes villes américaines.

4.4 Génération des Utilisateurs et Rôles

4.4.1 Stratégie de Génération

Analyse nécessite quatre types d'utilisateurs distincts :

- **Caissiers** : Employés de première ligne
- **Admin Store** : Gestionnaires de magasin
- **Investisseurs** : Participants au système de bidding
- **Admin Global** : Administrateur système unique

4.4.2 Génération de la Base Utilisateurs

Utilisation de la bibliothèque Faker pour créer 2000 utilisateurs synthétiques :

Listing 4.8: Génération des utilisateurs avec Faker

```
1 from faker import Faker
2
3 fake = Faker()
4 n = 2000
5
6 ids = []
7 names = []
8 emails = []
9 passwords = []
10 dates_of_birth = []
11
12 for i in range(1, n+1):
13     ids.append(i)
14     name = fake.name().replace(" ", "").lower()
15     names.append(name)
16     emails.append(f"{name}{i}@gmail.com")
17     passwords.append("javajee123")
18     dates_of_birth.append(
19         fake.date_of_birth(minimum_age=18, maximum_age=70)
20     )
21
22 users_df = pd.DataFrame({
23     'id': ids,
24     'user_name': names,
25     'mail': emails,
26     'password': passwords,
27     'date_of_birth': dates_of_birth
28 })
```

4.4.3 Creation des Caissiers

Les caissiers ont et  cr  s en association avec les commandes existantes. Chaque zone g ographique (identifi  par code postal) se voit attribuer 2 caissiers :

Listing 4.9: Attribution des caissiers par zone

```
1 orders_unique = (
2     df[["order_id", "zip"]]
3     .drop_duplicates()
4     .sort_values(["zip", "order_id"])
5     .reset_index(drop=True)
6 )
7
8 orders_unique["zip_index"] = (
9     orders_unique["zip"]
10    .astype("category")
11    .cat.codes
12 )
13
14 NB_CAISSIERS_PAR_ZIP = 2
15
16 orders_unique["caissier_id"] = (
17     orders_unique.groupby("zip")
18     .cumcount() % NB_CAISSIERS_PAR_ZIP
19 ) + 1 + orders_unique["zip_index"] * NB_CAISSIERS_PAR_ZIP
```

Ajout des informations sp cifiques aux caissiers :

Listing 4.10: Donn es compl mentaires des caissiers

```
1 from datetime import datetime, date
2
3 start = date(2010, 1, 1)
4 end = date.today()
5
6 caissier_df['date_started'] = [
7     fake.date_between(start_date=start, end_date=end)
8     for _ in range(len(caissier_df))
9 ]
10
11 # Salaire proportionnel a l'anciennete
12 today = datetime.today()
13 caissier_df['salaire'] = 2000 + 100 * caissier_df['date_started'].apply(
14     lambda x: today.year - x.year
15 )
```

R sultat : 1286 caissiers avec anciennet  et salaire variables (2000-3400\$).

4.4.4 Création des Administrateurs de Magasin

Sélection aléatoire de 531 utilisateurs (un par magasin) :

Listing 4.11: Administrateurs de magasin

```
1 ids_caissier = set(caissier_df['ID'])
2 df_disponible = users_df[~users_df['ID'].isin(ids_caissier)]
3
4 admin_store_df = df_disponible.sample(n=531, random_state=42)
5
6 # Association magasin
7 admin_store_df["store_id"] = store_df["store_id"].values
8
9 # Salaire supérieur aux caissiers
10 admin_store_df['salaire'] = 10000 + 100 * admin_store_df['date_started',
11     ].apply(
12         lambda x: today.year - x.year
13     )
```

Résultat : 531 administrateurs avec salaire moyen 11500\$.

4.4.5 Création des Investisseurs

Sélection de 182 investisseurs parmi les utilisateurs restants :

Listing 4.12: Crédit des investisseurs

```
1 ids_utilises = set(caissier_df['ID']).union(set(admin_store_df['ID']))
2 df_disponible = users_df[~users_df['ID'].isin(ids_utilises)]
3
4 investor_df = df_disponible.sample(n=182, random_state=42)
5
6 # Noms remplacez par noms de fabricants (manufacturers)
7 mask = users_df["ID"].isin(investor_df["ID"])
8 users_df.loc[mask, "user_name"] = np.random.choice(
9     df["manufacture"].unique(),
10    size=mask.sum(),
11    replace=True
12 )
```

Les investisseurs sont ensuite associés aléatoirement aux produits pour le système de bidding.

4.4.6 Administrateur Global

Un seul administrateur global a été créé :

Listing 4.13: Admin global unique

```
1 adminG_df = df_disponible.sample(n=1, random_state=42)
```

4.5 Création des Données Transactionnelles

4.5.1 Commandes (Orders)

Extraction des commandes uniques du dataset :

Listing 4.14: Table des commandes

```
1 orders_df = df[["order_id", "order_date", "ship_date", "caissier_id"]]
2 orders_df["order_id"] = orders_df["order_id"].str.slice(start=8)
3 orders_df = orders_df.rename(columns={"caissier_id": "user_id"})
4 orders_df = orders_df.drop_duplicates()
5
6 # Conversion en dates
7 orders_df['order_date'] = pd.to_datetime(orders_df['order_date'])
8 orders_df['ship_date'] = pd.to_datetime(orders_df['ship_date'])
```

Résultat : 9994 commandes avec dates de commande et d'expédition.

4.5.2 Articles de Commande (Order Items)

Création de la table de jointure entre commandes et produits :

Listing 4.15: Lignes de commande

```
1 orderItems_df = (
2     df[["order_id", "product_id", "sales", "quantity", "discount"]]
3     .reset_index(drop=True)
4 )
5
6 orderItems_df["item_id"] = orderItems_df.index + 1
7 orderItems_df = orderItems_df.rename(columns={"sales": "price"})
8 orderItems_df["order_id"] = orderItems_df["order_id"].str.slice(start=8)
```

Résultat : 9994 lignes de commande avec prix, quantité et réduction.

4.5.3 Inventaire (Inventory)

L'inventaire a été créé pour toutes les combinaisons possibles magasin × produit :

Listing 4.16: Génération complète de l'inventaire

```
1 # Creer la grille complete
2 all_stores = df['store_id'].unique()
3 all_products = products_df['product_id'].unique()
4
```

```

5 grid = pd.MultiIndex.from_product(
6     [all_stores, all_products],
7     names=[ 'store_id' , 'product_id' ]
8 ).to_frame(index=False)
9
10 # Fusion avec inventaire existant
11 df_complete = pd.merge(
12     grid,
13     inventory.drop(columns=[ 'inventory_id' ]),
14     on=[ 'store_id' , 'product_id' ],
15     how='left'
16 )
17
18 # Quantites aleatoires (distribution Beta)
19 df_complete[ 'quantity' ] = (
20     np.random.beta(a=0.5, b=4, size=len(df)) * 210
21 ).astype(int)
22
23 df_complete[ 'quantity' ] = df_complete[ 'quantity' ].fillna(0).astype(int)
24 df_complete[ 'id' ] = df_complete.index + 1

```

Résultat : 983,572 entrées d'inventaire ($531 \text{ magasins} \times 1852 \text{ produits}$).

La distribution Beta ($\alpha = 0.5, \beta = 4$) a été choisie pour simuler un inventaire réaliste avec :

- Majorité de produits avec stock faible (0-50 unités)
- Quelques produits populaires avec stock élevé (150-210 unités)

4.6 Création du Système de Bidding

4.6.1 Hiérarchie du Bidding

Le système de bidding suit une structure hiérarchique à 4 niveaux :

Catégorie → Rang → Face → Section

4.6.2 Rangs (Ranks)

Création de 9 rangs (3 par catégorie) :

Listing 4.17: Création des rangs

```

1 n_rangs = 9
2
3 rang_ids = []
4 rang_names = []

```

```

5 descriptions = []
6 category_ids = []
7
8 for rang_id in range(1, n_rangs + 1):
9     category_id = ((rang_id - 1) // 3) + 1
10
11     rang_ids.append(rang_id)
12     rang_names.append(f"rang_{rang_id}")
13     descriptions.append(
14         f"This rank represents position level {rang_id} "
15         f"within the system hierarchy."
16     )
17     category_ids.append(category_id)
18
19 rang_df = pd.DataFrame({
20     "rang_id": rang_ids,
21     "rang_name": rang_names,
22     "description": descriptions,
23     "category_id": category_ids
24 })

```

4.6.3 Faces

Création de 18 faces (2 par rang) :

Listing 4.18: Crédit à la création des faces

```

1 n_faces = 18
2
3 face_ids = list(range(1, n_faces + 1))
4 face_names = [f"face_{i}" for i in face_ids]
5 rang_ids = []
6
7 for face_id in face_ids:
8     rang_id = (face_id + 1) // 2
9     rang_ids.append(rang_id)
10
11 face_df = pd.DataFrame({
12     "face_id": face_ids,
13     "face_name": face_names,
14     "rang_id": rang_ids
15 })

```

4.6.4 Sections

Création de 360 sections (20 par face) avec prix de base aléatoires :

Listing 4.19: Création des sections avec prix

```
1 from datetime import date, timedelta
2 import random
3
4 n_faces = 18
5 sections_per_face = 20
6 date_delai_value = date.today() + timedelta(days=30)
7
8 section_id = 1
9
10 for face_id in range(1, n_faces + 1):
11     for position in range(1, sections_per_face + 1):
12         # Prix multiple de 100 entre 2000 et 6000
13         price = random.randrange(2000, 6001, 100)
14
15         section_ids.append(section_id)
16         section_names.append(f"section_{section_id}")
17         face_ids.append(face_id)
18
19         base_prices.append(price)
20         current_prices.append(price)
21         statuses.append("CLOSE")
22         date_delais.append(date_delai_value)
23
24         section_id += 1
25
26 section_df = pd.DataFrame({
27     "section_id": section_ids,
28     "section_name": section_names,
29     "face_id": face_ids,
30     "base_price": base_prices,
31     "current_price": current_prices,
32     "date_delai": date_delais,
33     "status": statuses
34 })
```

Résultat : 360 sections avec prix de base variant de 2000\$ à 6000\$.

4.7 Injection dans PostgreSQL

4.7.1 Méthode d'Injection

Toutes les tables ont été injectées via la méthode `to_sql()` de Pandas :

Listing 4.20: Injection type dans PostgreSQL

```
1 table_df.to_sql(
```

```
2     "nom_table",
3     engine,
4     if_exists="append",
5     index=False
6 )
```

4.7.2 Ordre d'Injection

L'ordre d'injection respecte les contraintes de clés étrangères :

1. Tables de base : `region`, `state`, `city`, `category`, `subcategory`
2. Utilisateurs : `user`
3. Magasins : `store`
4. Rôles : `caissier`, `admin_store`, `investor`, `adming`
5. Produits : `product` (avec `ID_investisseur`)
6. Transactions : `orders`, `order_items`
7. Inventaire : `inventory`
8. Bidding : `rang`, `face`, `section`

4.7.3 Volumétrie Finale

Le tableau suivant résume les volumes de données injectés :

Table	Lignes	Description
user	2000	Utilisateurs (tous rôles)
region	4	Régions géographiques
state	49	États américains
city	531	Villes
store	531	Magasins
category	3	Catégories principales
subcategory	17	Sous-catégories
product	1852	Produits uniques
caissier	1286	Caissiers
admin_store	531	Administrateurs magasin
investor	182	Investisseurs
adming	1	Administrateur global
orders	9994	Commandes
order_items	9994	Lignes de commande
inventory	983572	Stocks (store × product)
rang	9	Rangs de bidding
face	18	Faces de bidding
section	360	Sections bidding
Total	1 009 905	Lignes totales

4.8 Validation et Vérifications

4.8.1 Contrôles de Cohérence

Plusieurs vérifications ont été effectuées pour garantir la qualité des données :

- **Unicité des identifiants** : Aucun doublon dans les clés primaires
- **Intégrité référentielle** : Toutes les clés étrangères réfèrentent des entités existantes
- **Cohérence des rôles** : Aucun utilisateur n'a plusieurs rôles simultanément
- **Dates valides** : `order_date <= ship_date`
- **Prix cohérents** : Tous les prix sont positifs
- **Inventaire complet** : Toutes les combinaisons store × product présentes

4.8.2 Exemples de Requêtes de Validation

Listing 4.21: Vérification de l'intégrité

```
1 -- Vérifier qu'aucun produit n'a d'investisseur invalide
2 SELECT COUNT(*)
3 FROM product p
4 LEFT JOIN investor i ON p.id_investisseur = i.id
5 WHERE i.id IS NULL;
6
7 -- Vérifier la hiérarchie bidding complète
8 SELECT COUNT(*) FROM section s
9 JOIN face f ON s.face_id = f.face_id
10 JOIN rang r ON f.rang_id = r.rang_id
11 JOIN category c ON r.category_id = c.category_id;
```

4.9 Optimisations et Performances

4.9.1 Indexation

Des index ont été automatiquement créés par PostgreSQL sur :

- Toutes les clés primaires
- Toutes les clés étrangères
- Colonnes fréquemment utilisées dans les jointures

4.9.2 Taille de la Base de Données

Taille approximative après injection complète : **250 MB.**

4.10 Conclusion

Ce processus de transformation a permis de :

- Normaliser un dataset CSV brut en 18 tables relationnelles
- Générer plus d'1 million de lignes de données cohérentes
- Créer un écosystème complet d'utilisateurs avec rôles différenciés
- Implémenter une hiérarchie de bidding fonctionnelle
- Garantir l'intégrité référentielle et la qualité des données

Cette base de données constitue le fondement du système Analify, permettant des analyses riches et un système de bidding réaliste.

Chapter 5

Conception détaillée du frontend React/TypeScript

Dans ce chapitre, nous décrivons la conception et l'implémentation du frontend d'Analify, développé avec **React 18**, **TypeScript** et l'outillage moderne fourni par **Vite** et **Tailwind CSS**. L'objectif principal est d'offrir une expérience utilisateur fluide, réactive et agréable pour la consultation des tableaux de bord, la gestion du bidding et l'utilisation de l'assistant analytique.

5.1 Stack technique et outillage

Le frontend repose sur les éléments suivants :

- **React 18** : bibliothèque JavaScript pour la construction d'interfaces utilisateur déclaratives et composables ;
- **TypeScript** : surcouche typée de JavaScript permettant un meilleur outillage (auto-complétion, vérification statique) ;
- **Vite** : outil de build et de développement rapide, offrant un rechargement à chaud performant ;
- **Tailwind CSS** : framework CSS utilitaire, permettant de styliser rapidement les composants ;
- **shadcn/ui** : collection de composants UI préconstruits, intégrés avec Tailwind ;
- **React Router** : pour la gestion des routes (pages) ;
- éventuellement **Axios** ou **fetch** encapsulé dans un service API maison pour les appels HTTP.

5.2 Structure du projet frontend

Le code frontend se trouve dans le dossier `frontAnalify/`. La structure principale du dossier `src/` est la suivante :

- `main.tsx` : point d'entrée de l'application React (montage dans le DOM, configuration du router, etc.) ;
- `App.tsx` : composant racine, qui définit souvent la structure globale des routes ;
- `pages/` : pages principales de l'application (Landing, Login, Dashboard et sous-pages) ;
- `components/` : composants réutilisables (layout, shared, ui, charts, etc.) ;
- `contexts/` : contextes React (par exemple `AuthContext`) ;
- `services/api.ts` : service centralisé pour les appels HTTP vers le backend ;
- `hooks/` : hooks personnalisés (par ex. `use-mobile`, `use-toast`) ;
- `types/` : définitions TypeScript des types partagés (DTOs, réponses API, etc.).

Cette organisation favorise la séparation des responsabilités entre la navigation, la structure de présentation (layout), les composants métiers et la logique d'accès aux données.

5.3 Routing et pages principales

La navigation est gérée via **React Router**. Les routes typiques sont :

- `/` : page d'accueil (`Landing.tsx`), qui présente brièvement Analify ;
- `/login` : page de connexion (`Login.tsx`) ;
- `/dashboard` : layout principal du tableau de bord ;
- `/dashboard/statistics` : statistiques de base ;
- `/dashboard/enhanced-statistics` : tableau de bord avancé ;
- `/dashboard/products` : gestion/visualisation des produits et du stock ;
- `/dashboard/orders` : suivi des commandes ;
- `/dashboard/bidding` : module de bidding ;
- `*` : page 404 (`NotFound.tsx`).

Chaque page de dashboard est rendue à l'intérieur d'un layout commun (`DashboardLayout.tsx`), qui contient la barre latérale de navigation, le header et l'assistant LLM.

5.4 Layouts et navigation

Le composant `DashboardLayout.tsx` joue un rôle central dans l'ergonomie du frontend :

- il affiche une **sidebar** (issu de `components/layout/Sidebar.tsx`) avec les liens vers les différentes vues de dashboard ;
- il gère la partie **contenu principal**, où les pages enfants sont rendues via le router ;
- il intègre en permanence le composant `AnalyticsAssistant.tsx`, accessible via un bouton flottant ou un panneau latéral.

L'utilisation de Tailwind CSS et de shadcn/ui permet de construire rapidement des layouts responsive, adaptables aux différentes tailles d'écran (desktop, tablette, mobile). Des hooks comme `use-mobile` peuvent être utilisés pour adapter certains comportements en fonction de la largeur de l'écran.

5.5 Gestion de l'authentification côté frontend

L'authentification est gérée de la manière suivante :

1. L'utilisateur saisit ses identifiants sur la page `Login.tsx` ;
2. Le formulaire envoie une requête POST vers l'endpoint backend de login via le service `api.ts` ;
3. En cas de succès, le backend renvoie un JWT, qui est stocké côté frontend (par exemple dans le `localStorage` ou dans un contexte React) ;
4. Le `AuthContext.tsx` expose un hook (par exemple `useAuth`) fournissant le token, les informations utilisateur (rôle, nom, etc.) et des fonctions `login / logout` ;
5. Toutes les requêtes ultérieures vers le backend passent par `api.ts`, qui ajoute l'en-tête `Authorization: Bearer <token>` si l'utilisateur est connecté.

Des routes protégées peuvent être définies pour empêcher l'accès au dashboard sans être authentifié. Si le token est absent ou invalide, l'utilisateur est redirigé vers la page de connexion.

5.6 Service API centralisé

Le fichier `src/services/api.ts` centralise la logique d'appel au backend. Il définit :

- une constante `API_BASE_URL` pointant vers l'URL du backend (par exemple `http://localhost:8000`) ;
- une fonction générique (par ex. `apiRequest`) qui gère les méthodes HTTP, les en-têtes, la sérialisation JSON et la gestion d'erreurs ;
- des fonctions plus haut niveau pour chaque ressource : `statisticsApi.getBasicDashboard()`, `biddingApi.getSections()`, `assistantApi.askQuestion()`, etc.

Cette centralisation facilite :

- la gestion du token JWT (injection de l'en-tête `Authorization`) ;
- la gestion cohérente des erreurs (par exemple, redirection vers `/login` en cas d'erreur 401) ;
- la maintenance (changement d'URL de base ou d'implémentation HTTP en un seul endroit).

5.7 Composants métiers du dashboard

Les pages de dashboard sont construites à partir de **composants métiers** et de composants UI génériques :

- **StatCard** : petite carte affichant un KPI (titre, valeur, variation, icône) ;
- **DataTable** : tableau de données configurable (colonnes, pagination, tri, filtres) ;
- **FilterPanel** : panneau de filtres (période, magasin, catégorie, etc.) que l'utilisateur peut ajuster ;
- **Chart components** (`AreaChartCard`, `BarChartCard`, `LineChartCard`, `PieChartCard`) : composants graphiques encapsulant les librairies de charting ;
- **AnalyticsAssistant** : composant de chat pour interagir avec l'assistant LLM ;
- **ProfileForm**, badges, menus, etc.

Par exemple, la page de statistiques avancées (`EnhancedStatistics.tsx`) combine plusieurs cartes de KPI, des graphiques temporels et des tableaux de « top » (top produits, top magasins, etc.), chacun basé sur des composants réutilisables et alimentés par les DTO renvoyés par le backend.

5.8 Gestion des filtres et de l'état

Les filtres (période, magasin, catégorie, rôle, etc.) jouent un rôle central dans l'expérience utilisateur. Ils sont généralement gérés via :

- des composants contrôlés (inputs, select, date pickers) ;
- un état local ou global (hook React, `useState` ou `useReducer`) ;
- des appels API paramétrés, par exemple en envoyant un `StatisticsFilterDTO` au backend.

Un schéma classique :

1. L'utilisateur modifie un filtre dans `FilterPanel` ;
2. L'état local de la page est mis à jour ;
3. Un nouvel appel API est déclenché (par exemple via `useEffect`) avec les filtres mis à jour ;
4. Les composants graphiques et tableaux se mettent à jour avec les nouvelles données.

Cette approche rend l'interface **réactive** aux actions de l'utilisateur tout en gardant une logique de filtrage centralisée côté backend.

5.9 Interface du module de bidding

La page `BiddingDashboard.tsx` (ou équivalent) permet aux investisseurs et administrateurs de visualiser et piloter les enchères. L'interface peut proposer :

- une vue hiérarchique (catégories → rangs → faces → sections) ;
- des cartes représentant les sections avec leur état (disponible, en cours d'enchère, gagnée, etc.) ;
- des formulaires pour placer une nouvelle enchère (montant, durée, etc.) ;
- éventuellement des indicateurs synthétiques : nombre de sections gagnées, montant total des bids, etc.

Le front se contente d'appeler les endpoints du backend (chapitre suivant) et de refléter l'état renvoyé (liste des bids, statut des sections) sous forme de composants interactifs.

5.10 Intégration de l'assistant analytique LLM

Le composant `AnalyticsAssistant.tsx` fournit une interface de chat intégrée au layout du dashboard. Il fonctionne de la manière suivante :

1. L'utilisateur ouvre le panneau de chat (bouton flottant) ;
2. Il saisit une question métier (ex. : « Quels sont mes produits en low stock ce mois-ci ? ») ;
3. Le composant envoie la question à l'endpoint backend de l'assistant via `assistantApi.askQuestion` ;
4. La réponse (texte + métadonnées) est affichée sous forme de bulles de conversation ;
5. Des badges peuvent indiquer des informations complémentaires (rôle, nombre de produits en low stock, type d'erreur éventuelle côté LLM).

Le composant gère également :

- un état de chargement (spinner ou indicateur « L'assistant réfléchit... ») ;
- une gestion des erreurs (message utilisateur si le backend renvoie un statut d'erreur ou des métadonnées « QUOTA_EXCEEDED », « AUTH_ERROR », etc.) ;
- l'historique local de la conversation pendant la session de navigation.

L'intégration de l'assistant dans le layout global le rend toujours accessible, que l'utilisateur soit sur la page statistiques, produits, bidding ou commandes.

5.11 Expérience utilisateur et responsive design

Analify vise une expérience utilisateur moderne :

- design épuré et cohérent grâce à Tailwind et shadcn/ui ;
- affichage de feedbacks clairs (toasts de succès/erreur, messages d'erreur au niveau des formulaires) ;
- support des écrans de différentes tailles (menus repliables sur mobile, colonnes de tableaux adaptatives) ;
- utilisation de skeletons ou placeholders pour indiquer le chargement de données.

Les prochaines sections du rapport (chapitres sur l'analytique et le bidding) détaillent la manière dont ces interfaces consomment les API backend pour afficher des statistiques et gérer les enchères.

Chapter 6

Module d'analytique et tableaux de bord

Le module d'analytique constitue le **cœur décisionnel** d'Analify. Il permet de consolider et de visualiser les principaux indicateurs de performance (KPI) à différents niveaux (global, magasin, investisseur, caissier) et sur différentes dimensions (temps, catégorie de produit, magasin, section de rayon, etc.).

Ce chapitre décrit la conception de ce module côté backend (services de statistiques, DTO) et côté frontend (pages, composants graphiques), ainsi que son articulation avec l'assistant analytique LLM.

6.1 Objectifs du module d'analytique

Les objectifs principaux sont :

- fournir des **tableaux de bord synthétiques** pour les décideurs ;
- permettre une **exploration interactive** des données de vente, de stock et de commandes ;
- offrir des vues spécifiques par rôle (ADMIN_G, ADMIN_STORE, INVESTOR, CAISSIER) ;
- servir de **source d'information** pour l'assistant analytique LLM, sans lui donner d'accès direct à la base de données ;
- permettre des **exports** (CSV, éventuellement PDF) pour des analyses externes.

6.2 Services statistiques côté backend

Deux services principaux prennent en charge le calcul des statistiques :

6.2.1 StatisticsService

Ce service calcule les **statistiques de base**, par exemple :

- chiffre d'affaires total sur une période donnée ;
- nombre de commandes ;
- nombre de produits vendus ;
- valeur de stock actuelle ;
- taux de rupture de stock.

Il expose typiquement une méthode du type :

Listing 6.1: Exemple de signature de service pour les statistiques de base

```
1 public DashboardStatsDTO getDashboardStats(Long userId,
2                                         UserRole role,
3                                         StatisticsFilterDTO filter) {
4     // ...
5 }
```

Les paramètres `userId` et `role` permettent d'adapter la requête :

- un `ADMIN_G` voit toutes les données, possiblement agrégées par magasin ;
- un `ADMIN_STORE` ne voit que les commandes et stocks de son magasin ;
- un `INVESTOR` ne voit que les produits et performances qui le concernent ;
- un `CAISSIER` ne voit éventuellement qu'un sous-ensemble très limité des statistiques.

Le paramètre `StatisticsFilterDTO` encapsule les filtres : période (dates de début/fin), magasin, catégorie, etc.

6.2.2 EnhancedStatisticsService

Ce service propose un **tableau de bord avancé**, regroupé dans un DTO `EnhancedDashboardDTO`. Il inclut :

- des listes de « top » (top produits par CA, par marge, top magasins, top investisseurs) ;
- des répartitions par catégorie et par magasin ;
- des séries temporelles (courbes de CA par jour, semaine, mois) ;

- des indicateurs spécifiques au module de bidding (sections les plus performantes, taux d'occupation, etc.).

Une signature typique :

Listing 6.2: Exemple de service de statistiques avancées

```

1 public EnhancedDashboardDTO getEnhancedDashboard(Long userId,
2                                     UserRole role,
3                                     StatisticsFilterDTO filter) {
4     // Requetes JPA, aggregations, mapping vers le DTO
5 }
```

Comme pour les statistiques de base, le rôle et l'identifiant utilisateur sont utilisés pour limiter l'accès aux données. Par exemple, un investisseur ne voit que les performances de ses sections et produits.

6.3 DTO et structure des tableaux de bord

Les résultats sont organisés dans des DTO fortement typés, par exemple :

6.3.1 DashboardStatsDTO

Ce DTO représente le tableau de bord « simple » avec des KPIs globaux. Il peut contenir :

- des champs numériques (`totalRevenue`, `totalOrders`, `totalProductsSold`, `stockValue`) ;
- des séries temporelles sous forme de listes d'objets (`date`, `valeur`) ;
- des répartitions par catégorie ou magasin.

6.3.2 EnhancedDashboardDTO

Ce DTO représente un **tableau de bord enrichi**, comprenant des collections de structures plus détaillées, par exemple :

- une liste de `RankingItemDTO` pour les top produits (avec nom, CA, marge, rang) ;
- une liste de top magasins ;
- une liste de top investisseurs ;
- des cartes de chaleur (heatmaps) de ventes par région/magasin ;

- des indicateurs liés au bidding (sections les plus demandées, sections inoccupées, etc.).

Structurer les données de cette manière facilite :

- la consommation côté frontend (typage TypeScript aligné sur le DTO Java) ;
- la réutilisation des données par l'assistant LLM, qui travaille sur des résumés construits à partir de ces DTO.

6.4 Endpoints REST liés aux statistiques

Les services statistiques sont exposés via des contrôleurs REST, par exemple :

- GET `/api/statistics/basic` : retourne un `DashboardStatsDTO` pour le rôle et les filtres fournis ;
- GET `/api/statistics/enhanced` : retourne un `EnhancedDashboardDTO` ;
- éventuellement des endpoints dédiés aux exports (`/api/statistics/export-csv`, etc.).

Ces endpoints :

1. récupèrent l'ID utilisateur et le rôle via le filtre JWT ;
2. construisent un objet `StatisticsFilterDTO` à partir des paramètres de requête ;
3. délèguent aux services métier ;
4. renvoient le DTO sous forme de JSON.

6.5 Rendu des tableaux de bord côté frontend

Les pages de statistiques côté frontend consomment les DTO du backend pour afficher les informations de façon visuelle :

6.5.1 Tableau de bord de base

La page `Dashboard.tsx` (ou équivalent) peut afficher :

- une grille de **StatCard** montrant les principaux KPI (CA, commandes, produits vendus, valeur de stock) ;
- un graphique linéaire de l'évolution du CA sur la période sélectionnée ;

- un diagramme circulaire (*pie chart*) de la répartition du CA par catégorie ;
- un tableau listant les commandes récentes.

Chaque composant graphique est alimenté par les données du DTO `DashboardStatsDTO`. Les filtres (période, magasin, catégorie) sont gérés via `FilterPanel`, qui déclenche de nouveaux appels API en cas de modification.

6.5.2 Tableau de bord avancé

La page `EnhancedStatistics.tsx` est dédiée aux utilisateurs ayant besoin d'analyses plus poussées (ADMIN_G, ADMIN_STORE, INVESTOR). Elle peut inclure :

- des blocs « Top 10 produits » ordonnés par CA ou marge ;
- un top magasins par CA ;
- un top investisseurs par montant d'enchères gagnées ;
- des heatmaps ou cartes montrant la répartition géographique des performances ;
- des comparaisons de périodes (par exemple ce mois-ci vs mois dernier).

La structure du DTO `EnhancedDashboardDTO` est pensée pour correspondre à ces besoins, en fournissant directement des collections déjà triées et agrégées, ce qui simplifie la logique côté frontend.

6.6 Gestion des exports

Pour certains profils (par exemple ADMIN_G et ADMIN_STORE), il est utile de pouvoir exporter les données pour des analyses complémentaires (Excel, outils BI, etc.).

Le backend peut fournir des endpoints tels que :

- GET `/api/statistics/export-csv?filter=...` : génère un fichier CSV contenant les données agrégées ;
- éventuellement GET `/api/statistics/export-pdf` : génère un rapport PDF (tableaux, graphiques simplifiés).

Le frontend propose alors des boutons d'action (« Export CSV », « Export PDF »), qui déclenchent le téléchargement du fichier et informent l'utilisateur du succès ou de l'échec de l'opération.

6.7 Rôle de l'analytique pour le module de bidding

Le module d'analytique ne se limite pas aux ventes et stocks. Il fournit également des indicateurs pour le module de bidding :

- performance historique des sections (CA généré, trafic estimé, taux de rotation des produits) ;
- taux d'occupation des sections (sections louées vs disponibles) ;
- comparaison des résultats avant/après installation d'un investisseur sur une section.

Ces informations sont essentielles pour :

- aider les investisseurs à choisir les sections les plus pertinentes pour leurs produits ;
- permettre aux administrateurs de magasin de valoriser leurs espaces de rayon ;
- alimenter l'assistant LLM lorsqu'un utilisateur pose une question spécifique sur les performances des sections ou des bids.

6.8 Lien avec l'assistant analytique LLM

L'assistant LLM ne dispose pas d'un accès direct à la base de données. Il s'appuie exclusivement sur les **résultats des services statistiques** pour formuler ses réponses.

Le workflow est le suivant :

1. L'utilisateur pose une question (via le frontend) ;
2. Côté backend, `AnalyticsAssistantService` appelle `StatisticsService` et `EnhancedStatisticsService` avec le rôle et les filtres appropriés ;
3. Ces services renvoient des DTO (`DashboardStatsDTO`, `EnhancedDashboardDTO`) contenant les statistiques filtrées ;
4. `AnalyticsAssistantService` construit un **résumé textuel** de ces statistiques (par exemple « Le chiffre d'affaires de ce mois-ci est de X euros, les 3 produits les plus vendus sont A, B, C... ») ;
5. Ce résumé est inclus dans le prompt envoyé au LLM via Spring AI ;
6. Le LLM renvoie une réponse structurée en langage naturel, que le backend transmet au frontend.

Cette approche garantit que :

- les contraintes de rôle et de périmètre sont respectées (puisque le LLM ne voit que ce que les services statistiques retournent) ;
- la taille des prompts reste maîtrisée (on n'envoie pas des objets JSON bruts gigantesques au LLM) ;
- l'assistant donne des réponses cohérentes avec les graphiques et tableaux du dashboard.

Ainsi, le module d'analytique joue un double rôle : fournir des tableaux de bord visuels et servir de moteur de données pour l'assistant LLM.

Chapter 7

Module de bidding (enchères sur sections)

Le module de **bidding** constitue l'une des spécificités majeures d'Analify. Il permet de transformer les sections de rayon en actifs monétisables sur lesquels les investisseurs peuvent enchérir, en se basant sur les performances historiques et le potentiel de vente.

Ce chapitre détaille :

- le modèle de domaine du bidding (catégories, rangs, faces, sections, bids) ;
- la logique métier côté backend (services et règles d'enchères) ;
- les endpoints REST d'exposition ;
- l'interface utilisateur côté frontend ;
- les liens avec le module d'analytique.

7.1 Modèle de domaine du bidding

Le bidding repose sur une hiérarchie spatiale qui reflète la structure des rayons physiques en magasin :

Catégorie : famille de produits (ex. : boissons, produits frais, hygiène, etc.) ;

Rang : un « rang » ou « allée » dans un rayon ;

Face : une face de rayon visible (avant d'un linéaire) ;

Section : portion précise d'une face, louable indépendamment ;

Bid : enchère placée par un investisseur pour occuper une section sur une période donnée.

Les entités principales sont donc :

- **Category** : identifiée par un nom et éventuellement des métadonnées (couleur, code, etc.) ;
- **Section** : identifiée par un code unique, associée à une catégorie, un magasin et une face ;
- **Bid** : contient l'investisseur, la section, le montant, la date de début/fin, le statut ;
- éventuellement des entités intermédiaires pour les rangs et faces, selon le niveau de détail retenu.

Un schéma simplifié peut être représenté comme suit :

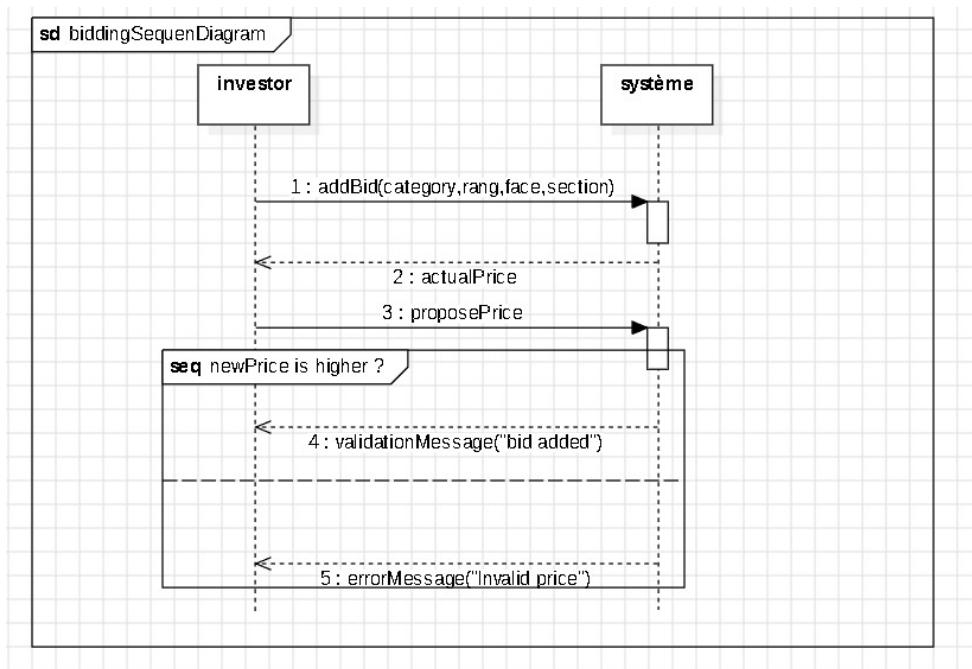


Figure 7.1: Vue simplifiée du modèle de domaine du bidding

Chaque section possède des attributs descriptifs (surface, emplacement, visibilité, etc.) et des indicateurs issus du module d'analytique (CA généré, trafic estimé), qui aident les investisseurs à évaluer l'intérêt de placer une enchère.

7.2 Logique métier des enchères

Le service `BiddingService` encapsule la logique métier du module, notamment :

- la liste des sections disponibles pour un investisseur donné ;
- les règles de création d'une nouvelle bid ;
- la détermination des bids gagnantes ;

- la mise à jour des statuts des sections et des enchères (ouverte, en cours, gagnée, expirée, annulée, etc.).

Quelques règles métier typiques :

- une section ne peut être **occupée que par un seul investisseur** à un instant donné ;
- plusieurs bids peuvent exister sur une section tant que la période d'enchère est ouverte, mais seule la bid au montant maximal (et respectant certaines conditions) sera déclarée gagnante ;
- un investisseur ne peut pas placer plusieurs bids concurrentes sur la même section pour la même période ;
- certaines sections peuvent être réservées à certains types d'investisseurs ou à des partenaires privilégiés.

Le service travaille en étroite collaboration avec le module d'analytique pour obtenir des indicateurs liés aux sections (CA historique, taux de rotation, etc.) et éventuellement influencer les prix de réserve ou les recommandations.

7.3 Endpoints REST du module de bidding

Le `BiddingController` expose les principales opérations sous forme d'API REST, par exemple :

- `GET /api/bidding/categories` : liste toutes les catégories disponibles ;
- `GET /api/bidding/sections` : liste les sections, filtrées par rôle/utilisateur et éventuellement par catégorie, magasin, statut ;
- `GET /api/bidding/sections/{id}` : détail d'une section et de ses bids associées ;
- `POST /api/bidding/bids` : créer une nouvelle bid (corps JSON contenant l'ID de la section, le montant, la période) ;
- `GET /api/bidding/bids/mine` : lister les bids d'un investisseur ;
- `POST /api/bidding/sections/{id}/close` : clôturer les bids d'une section (par un administrateur, par exemple) et déclarer un gagnant.

Comme pour les autres modules, le filtre JWT enrichit les requêtes avec `userId` et `role`, ce qui permet au service :

- de retourner uniquement les sections et bids autorisées pour un investisseur donné ;
- de distinguer les capacités d'un ADMIN_G / ADMIN_STORE (vision globale, possibilité de fermer une enchère) de celles d'un INVESTOR (vision limitée à ses propres bids) ;
- de restreindre fortement l'accès pour les caissiers, qui n'ont pas vocation à gérer les enchères.

7.4 Interface utilisateur du bidding côté frontend

La page `BiddingDashboard.tsx` (ou similaire) fournit une vision dédiée au module de bidding. Elle peut être structurée comme suit :

7.4.1 Navigation hiérarchique

Un explorateur latéral ou des menus déroulants permettent de filtrer :

- par catégorie de produit ;
- par magasin ;
- par rang ou face (si ces concepts sont modélisés explicitement dans l'interface).

Les sections correspondantes sont alors affichées sous forme de cartes ou de lignes de tableau, avec :

- leur nom/code ;
- leur statut (disponible, en enchère, occupée) ;
- des indicateurs clés (CA généré, nombre de produits, visibilité estimée) ;
- les bids actuelles (pour les administrateurs) ou la bid de l'investisseur connecté.

7.4.2 Formulaire de placement d'enchère

Lorsqu'un investisseur souhaite placer une enchère sur une section :

1. Il sélectionne la section souhaitée ;
2. Un formulaire s'affiche (modale ou panneau latéral) où il renseigne le montant de l'enchère et éventuellement la période souhaitée ;
3. Le frontend envoie une requête POST vers `/api/bidding/bids` avec les informations saisies ;

4. En cas de succès, un message de confirmation est affiché (toast) et la liste des bids est rafraîchie ;
5. En cas d'erreur (section déjà occupée, montant insuffisant, etc.), un message explicite est renvoyé par le backend et affiché à l'utilisateur.

7.4.3 Suivi des bids et indicateurs

L'investisseur dispose également d'une vue récapitulative de ses enchères :

- liste de ses bids avec statut (en cours, gagnée, perdue, expirée) ;
- montant et date de chacune ;
- informations sur la section correspondante ;
- éventuellement des graphiques montrant l'évolution de ses montants investis dans le temps.

Cette vue est alimentée par l'endpoint `/api/bidding/bids/mine` et s'appuie sur des composants de tableau et de graphiques similaires à ceux utilisés pour les statistiques.

7.5 Lien entre bidding et analytique

Le module de bidding est intimement lié au module d'analytique :

- Les indicateurs de performance des sections (CA, marge, taux de rotation) sont indispensables pour que les investisseurs puissent évaluer la pertinence d'une section avant d'encherir ;
- Les administrateurs peuvent utiliser les tableaux de bord pour identifier les sections sous-exploitées et décider de les proposer en bidding ;
- Les résultats des bids (sections gagnées, montants investis) alimentent à leur tour les tableaux de bord (par exemple, top sections par revenu généré via bidding).

Dans l'autre sens, les décisions de bidding ont un impact sur les ventes et, donc, sur les indicateurs analytiques. Analify permet de boucler cette boucle de rétroaction :

1. Un investisseur place une enchère et obtient une section ;
2. Ses produits sont mieux mis en valeur, ce qui (idéalement) augmente les ventes ;
3. Les tableaux de bord montrent l'amélioration des performances pour cette section et ces produits ;
4. L'investisseur et l'enseigne peuvent ajuster leur stratégie de bidding en conséquence.

7.6 Intégration avec l'assistant analytique LLM

Le module de bidding est également pris en compte par l'assistant LLM. Quelques exemples de questions possibles :

- « Quelles sont mes sections les plus rentables ce trimestre ? »
- « Sur quelles catégories devrais-je concentrer mes prochaines enchères ? »
- « Quels magasins ont le plus de sections inoccupées ? »

Côté backend, `AnalyticsAssistantService` inclut dans son résumé analytique :

- des informations sur les sections et bids de l'investisseur connecté ;
- des statistiques agrégées sur les sections (taux d'occupation, performance moyenne) ;
- des indicateurs par catégorie.

Le LLM peut alors formuler des recommandations qualitatives, tout en restant limité aux données que les services de bidding et d'analytique lui ont résumées.

Ainsi conçu, le module de bidding transforme le rayon en un véritable « marché d'espaces », piloté par les données et accessible via une interface moderne et un assistant intelligent.

Chapter 8

Sécurité, authentification et gestion des rôles

La sécurité est un aspect central du projet Analify : il s'agit de garantir que chaque utilisateur n'accède qu'aux données et fonctionnalités pour lesquelles il est habilité. Ce chapitre détaille la mise en place de l'authentification, de l'autorisation et de la gestion des rôles, essentiellement côté backend Spring Boot, et la manière dont le frontend consomme ces mécanismes.

8.1 Objectifs de sécurité

Les objectifs principaux sont les suivants :

- **Authentifier** les utilisateurs via un couple identifiant/mot de passe ;
- **Distribuer** un token sécurisé (JWT) à chaque session ;
- **Contrôler l'accès** aux endpoints backend en fonction du rôle et du périmètre de l'utilisateur ;
- **Propager** l'identité et le rôle à travers toutes les couches (controller, service, repository) pour filtrer les données ;
- Assurer une intégration fluide avec le frontend (stockage du token, redirections, affichage conditionnel des fonctionnalités).

8.2 Modèle de rôles fonctionnels

Analify définit quatre rôles principaux :

ADMIN_G (Administrateur global) : vision et droits sur l'ensemble de l'enseigne (tous les magasins, toutes les sections, tous les utilisateurs) ;

ADMIN_STORE (Administrateur de magasin) : vision limitée à un magasin donné (stocks, commandes, sections de ce magasin) ;

INVESTOR (Investisseur) : vision centrée sur ses propres sections, produits et enchères ;

CAISSIER : vision restreinte aux opérations de caisse et, éventuellement, à des statistiques très simplifiées.

Ces rôles sont représentés en Java par une énumération `UserRole` et sont stockés en base de données (dans la table `users` ou via une table `roles`).

8.3 Authentification par JWT

L'authentification est basée sur des **JSON Web Tokens** (JWT) signés. Le flux est le suivant :

1. L'utilisateur envoie ses identifiants (email/mot de passe) à l'endpoint de login (par exemple `/api/auth/login`).
2. Le backend vérifie les identifiants via un service (`UserDetailsService` ou service maison) ; si le mot de passe (hashé) correspond, l'utilisateur est authentifié.
3. Un JWT est généré contenant au minimum : l'ID utilisateur, le rôle, une date d'expiration, et éventuellement d'autres claims.
4. Ce token est renvoyé au frontend, qui le stocke (par exemple dans le `localStorage` ou un cookie sécurisé).
5. Pour chaque requête ultérieure, le frontend ajoute l'en-tête HTTP `Authorization: Bearer <token>`.

Le service de génération/validation du JWT repose sur la bibliothèque `jjwt`. Une clé secrète (stockée en configuration) est utilisée pour signer le token ; le backend la réutilise pour vérifier l'intégrité des tokens reçus.

8.4 Configuration Spring Security

Spring Security est configuré pour :

- désactiver la gestion de session côté serveur (stateless, puisque le JWT est auto-porteur) ;

- autoriser librement certains endpoints (`/api/auth/login`, éventuellement `/api/auth/register`) ;
- exiger une authentification (présence d'un JWT valide) pour tous les endpoints `/api/**` restants ;
- appliquer un filtre JWT personnalisé à chaque requête ;
- définir les stratégies de CORS (pour autoriser le frontend à appeler le backend depuis un autre domaine/port en développement).

Une configuration typique inclut une classe annotée `@Configuration` et `@EnableWebSecurity`, où l'on définit un bean de type `SecurityFilterChain` :

Listing 8.1: Exemple simplifié de configuration Spring Security

```

1  @Configuration
2  @EnableWebSecurity
3  @RequiredArgsConstructor
4  public class SecurityConfig {
5
6      private final JwtAuthenticationFilter jwtAuthenticationFilter;
7
8      @Bean
9      public SecurityFilterChain securityFilterChain(HttpSecurity http)
10         throws Exception {
11         http
12             .csrf(AbstractHttpConfigurer::disable)
13             .sessionManagement(session ->
14                 session.sessionCreationPolicy(SessionCreationPolicy.
15                     STATELESS)
16             )
17             .authorizeHttpRequests(auth -> auth
18                 .requestMatchers("/api/auth/**").permitAll()
19                 .anyRequest().authenticated()
20             )
21             .addFilterBefore(jwtAuthenticationFilter,
22                 UsernamePasswordAuthenticationFilter.class);
23
24         return http.build();
25     }
26 }
```

8.5 Filtre JWT et propagation du rôle

Le filtre JWT (`JwtAuthenticationFilter`) est appliqué à chaque requête HTTP entrante :

1. Il lit l'en-tête `Authorization` ;
2. S'il contient un token Bearer, il le valide via le service JWT ;
3. En cas de succès, il extrait l'ID utilisateur et le rôle (claims du token) ;
4. Il construit un objet `Authentication` Spring (par exemple `UsernamePasswordAuthenticationToken`) et le place dans le `SecurityContext` ;
5. Il ajoute également les attributs `userId` et `role` à l'objet requête (`request.setAttribute("userId", ...)`).

Cela permet aux contrôleurs de récupérer directement ces informations via :

Listing 8.2: Injection de l'ID utilisateur et du rôle dans un contrôleur

```

1 @PostMapping("/query")
2 public AnalyticsAssistantResponse queryAnalyticsAssistant(
3     @RequestAttribute("userId") Long userId,
4     @RequestAttribute("role") UserRole role,
5     @RequestBody AnalyticsAssistantRequest request) {
6
7     return analyticsAssistantService.answerQuestion(userId, role,
8         request.getQuestion());
}
```

Cette technique de propagation par `RequestAttribute` est particulièrement utile pour les services qui ne dépendent pas directement de Spring Security mais ont besoin de connaître l'utilisateur courant.

8.6 Stratégies d'autorisation par rôle

Outre la configuration globale, certaines restrictions plus fines peuvent être mises en place :

8.6.1 Au niveau des endpoints

On peut utiliser des annotations telles que `@PreAuthorize` pour restreindre l'accès à certaines méthodes de contrôleur ou services :

Listing 8.3: Exemple d'autorisation fine via `@PreAuthorize`

```

1 @PreAuthorize("hasRole('ADMIN_G')")
2 @PostMapping("/bidding/sections/{id}/close")
3 public void closeSectionBidding(@PathVariable Long id) {
4     biddingService.closeSectionBidding(id);
5 }
```

Ainsi, seule un utilisateur avec le rôle ADMIN_G (ou éventuellement ADMIN_STORE pour son magasin) pourra fermer les enchères d'une section.

8.6.2 Au niveau des services et repositories

Dans de nombreux cas, les règles d'autorisation sont implémentées directement dans la logique métier : les services reçoivent `userId` et `role` en paramètre et adaptent leurs requêtes JPA en conséquence, par exemple :

- un `StatisticsService` qui filtre les commandes par `storeId` lorsque le rôle est `ADMIN_STORE` ;
- un `BiddingService` qui ne retourne que les sections associées à l'investisseur pour le rôle `INVESTOR` ;
- un service de produits qui ne retourne que les stocks d'un magasin.

Ce double niveau (config Spring Security + filtrage métier) garantit un bon équilibre entre flexibilité et sécurité.

8.7 Sécurité côté frontend

Bien que la sécurité « forte » soit assurée côté backend, le frontend joue un rôle important pour l'expérience utilisateur :

- Stockage du token JWT (par exemple dans `localStorage`) et gestion de l'état d'authentification via `AuthContext` ;
- Ajout systématique de l'en-tête `Authorization` dans les appels API (via le service `api.ts`) ;
- Redirection automatique vers la page de login en cas d'erreur 401/403 ;
- Affichage conditionnel des éléments d'interface selon le rôle (par exemple, cacher les menus d'administration aux investisseurs ou caissiers).

Par exemple, la sidebar du `DashboardLayout` peut afficher certains liens uniquement pour les utilisateurs `ADMIN_G` ou `ADMIN_STORE`.

8.8 Limitations et pistes d'amélioration

Dans le cadre de ce projet, la sécurité mise en place se concentre sur :

- la protection basique des endpoints ;
- l'isolation des données par rôle ;
- la robustesse minimale de l'authentification.

Plusieurs améliorations pourraient être envisagées dans une version industrielle d'Analify :

- gestion du renouvellement des tokens (refresh tokens) ;
- intégration avec un annuaire d'entreprise (LDAP, Active Directory) ;
- audit des actions sensibles (journalisation des modifications, accès aux données critiques) ;
- durcissement des en-têtes HTTP (CSP, HSTS, X-Frame-Options, etc.) ;
- monitoring des tentatives de connexion et mise en place de mécanismes anti-bruteforce.

Malgré ces limites, la solution actuelle fournit une base solide pour un projet académique et garantit que chaque profil (ADMIN_G, ADMIN_STORE, INVESTOR, CAISSIER) ne voit que ce qu'il est censé voir, y compris dans le cadre de l'assistant LLM.

Chapter 9

Assistant analytique LLM (Spring AI + Ollama)

L'assistant analytique constitue l'une des innovations majeures d'Analify : il permet aux utilisateurs de poser des questions métier en langage naturel (français ou anglais) et d'obtenir des réponses synthétiques, contextualisées et adaptées à leur rôle. Ce chapitre décrit en détail la conception de cet assistant, son évolution technologique (Gemini → Spring AI → Ollama), la construction du contexte analytique et son intégration dans le frontend.

9.1 Objectifs et principes de conception

Les objectifs poursuivis sont :

- permettre aux utilisateurs de naviguer dans les données sans devoir manipuler directement des filtres ou tableaux complexes ;
- offrir une interface de **conversation** continue, accessible depuis toutes les pages du dashboard ;
- garantir que l'assistant respecte les **contraintes de rôle** et ne divulgue jamais de données hors périmètre ;
- concevoir une intégration flexible, indépendante du fournisseur de LLM (modèle distant, modèle local, etc.).

Pour atteindre ces objectifs, l'assistant a été encapsulé dans un service backend dédié (`AnalyticsAssistantService`) et un composant frontend réutilisable (`AnalyticsAssistant.tsx`).

9.2 Première version : intégration directe à Google Gemini

Dans une première étape, l'assistant était intégré directement à l'API **Google Gemini** via des appels HTTP manuels :

- le backend construisait une requête JSON au format attendu par l'API Gemini (`/v1beta/models/{model}:generateContent`) ;
- le `systemPrompt` décrivait le rôle de l'assistant (analyste de données pour la grande distribution, travaillant sur Analify) ;
- le `userPrompt` contenait la question de l'utilisateur et un « contexte » issu des statistiques ;
- la réponse (un texte en langage naturel) était extraite du JSON renvoyé par Gemini et renvoyée au frontend.

Cette approche a permis de valider le concept mais a rencontré plusieurs limites :

- gestion manuelle des appels HTTP (construction du JSON, gestion des codes d'erreur) ;
- quotas et limitations de l'API Gemini (erreurs 429, quotas très rapidement atteints) ;
- forte dépendance à un fournisseur externe.

9.3 Refactorisation vers Spring AI

Pour simplifier l'intégration et se préparer à supporter plusieurs fournisseurs de LLM, le projet a ensuite migré vers **Spring AI**. Spring AI fournit un `ChatClient` et des abstractions communes pour interagir avec différents modèles (OpenAI, Gemini, Ollama, etc.).

Les principaux changements :

- ajout de la dépendance `spring-ai-openai-spring-boot-starter` (dans un premier temps) et configuration de l'API Gemini via son endpoint compatible « OpenAI » ;
- injection d'un `ChatModel` et construction d'un `ChatClient` dans le backend ;
- simplification du code d'appel : un simple `chatClient.prompt().user(prompt).call().content` au lieu d'un appel HTTP brut.

Le service `AnalyticsAssistantService` a été refactoré pour utiliser Spring AI, tout en conservant sa responsabilité principale : construire un prompt riche et adapté au rôle de l'utilisateur.

9.4 Passage à un LLM local via Ollama

Malgré l'amélioration de l'architecture, l'utilisation de Gemini restait limitée par les quotas et la dépendance à la connexion Internet. Pour s'affranchir de ces contraintes, le projet a migré vers un **LLM local** en utilisant **Ollama**.

Ollama est un serveur local qui permet de télécharger et d'exécuter des modèles (par exemple `llama3.2`) sur la machine de développement, via une API HTTP compatible avec les attentes de Spring AI.

Les changements côté backend ont été les suivants :

- remplacement de la dépendance Spring AI OpenAI par `spring-ai-ollama-spring-boot-starter` ;
- mise à jour de `application.properties` pour pointer vers l'URL d'Ollama (par défaut `http://localhost:11434`) et définir le modèle utilisé (par exemple `llama3.2`) ;
- aucun changement dans la signature de `AnalyticsAssistantService` : celui-ci continue d'utiliser le `ChatClient`, sans se soucier du fournisseur réel.

Cette migration a permis :

- de s'affranchir des quotas d'API ;
- de travailler en mode hors-ligne (dans la limite des capacités matérielles de la machine) ;
- de mieux contrôler les temps de réponse et la confidentialité des données (tout reste en local).

9.5 Construction du contexte analytique

L'une des difficultés majeures consistait à **fournir au LLM suffisamment de contexte** pour répondre utilement aux questions, sans pour autant :

- envoyer un volume de données énorme ;
- enfreindre les contraintes de rôle et de périmètre ;

- provoquer des erreurs techniques (par exemple, dépassement des limites de taille des chaînes JSON).

Dans une version initiale, le backend tentait d'inclure directement des structures JSON complexes (entiers DTO) dans le prompt, ce qui a conduit à des erreurs (`StreamConstraintsException` liée à la taille des chaînes lors de la sérialisation).

La solution a été de passer à une **approche de résumé textuel** :

1. `AnalyticsAssistantService` appelle `StatisticsService` et `EnhancedStatisticsService` avec `userId`, `role` et un `StatisticsFilterDTO` raisonnable (par exemple la période récente) ;
2. les DTO retournés sont parcourus dans le service pour produire des phrases résumant les principaux indicateurs, par exemple :
 - « Le chiffre d'affaires du dernier mois est de X euros. »
 - « Les 3 produits les plus vendus sont A, B, C avec respectivement V1, V2 et V3 unités. »
 - « Il y a N produits en low stock. »
3. Pour les investisseurs, le résumé se focalise sur leurs sections et bids ; pour les administrateurs, sur les magasins ; pour les caissiers, sur des informations simplifiées.
4. Ce texte est ensuite concaténé avec la question de l'utilisateur et un rappel du rôle (« Tu parles à un administrateur de magasin... »).

Ce résumé constitue le « contexte analytique » injecté dans le prompt. Il permet au LLM de raisonner sur des informations synthétiques, alignées sur ce que l'utilisateur verrait dans les dashboards.

9.6 Gestion des erreurs et métadonnées de réponse

L'intégration avec un LLM étant sujette à diverses erreurs (problèmes réseau, limites de quota, mauvaise configuration), le service d'assistance a été conçu pour retourner, en plus de la réponse textuelle :

- un champ `error` dans `AnalyticsAssistantResponse.metadata`, indiquant le type d'erreur (`QUOTA_EXCEEDED`, `AUTH_ERROR`, `LLM_CALL_FAILED`, etc.) ;
- d'autres métadonnées utiles, comme le rôle utilisé ou le nombre de produits en low stock.

En cas d'erreur spécifique (par exemple, code 429 de l'API Gemini dans l'ancienne version), le service renvoyait un message d'erreur compréhensible par l'utilisateur (« Le service d'IA a atteint sa limite de quota, veuillez réessayer plus tard. »).

Avec Ollama, les erreurs sont principalement liées à :

- l'absence du serveur Ollama (non démarré) ;
- l'absence du modèle spécifié (non téléchargé) ;
- des temps de réponse trop longs sur des machines peu puissantes.

Le frontend, via `AnalyticsAssistant.tsx`, interprète ces métadonnées pour afficher des messages et badges adaptés.

9.7 Interface utilisateur de l'assistant

Le composant `AnalyticsAssistant.tsx` offre une interface de chat simple et efficace :

- un bouton flottant dans le `DashboardLayout` ouvre/ferme le panneau de chat ;
- une zone de texte permet de saisir la question ;
- une liste de messages affiche l'historique (questions de l'utilisateur et réponses de l'assistant) ;
- des badges affichent des informations comme le rôle, le nombre de low stock, et le type d'erreur en cas de problème LLM.

9.7.1 Cycle de requête côté frontend

Le cycle côté frontend est :

1. L'utilisateur saisit une question et clique sur « Envoyer » ;
2. Le composant ajoute immédiatement le message de l'utilisateur à la liste (optimisme, pour ressentir une réactivité) ;
3. Un état de chargement est activé ;
4. La question est envoyée à l'endpoint backend via `assistantApi.askQuestion` ;
5. À la réception de la réponse, le message de l'assistant est ajouté à l'historique, et l'état de chargement est désactivé ;
6. En cas d'erreur, un message spécifique est affiché (par exemple « L'assistant est temporairement indisponible. »).

9.8 Respect du périmètre et confidentialité des données

Un point fondamental est que l'assistant ne doit jamais renvoyer des informations auxquelles l'utilisateur n'a pas droit. Pour cela :

- le service LLM ne requête jamais la base de données directement ;
- il s'appuie uniquement sur les services de statistiques (et éventuellement de bidding), qui sont déjà filtrés par `userId` et `role` ;
- le résumé textuel ne contient que des informations autorisées par ces services ;
- le LLM ne peut donc pas « deviner » des données qu'il n'a pas reçues en contexte.

Cette architecture garantit que les mêmes règles de sécurité s'appliquent que l'utilisateur consulte un graphique dans le dashboard ou pose une question à l'assistant.

9.9 Limites actuelles et perspectives d'évolution

Malgré sa puissance, l'assistant présente certaines limites inhérentes à l'utilisation de LLM :

- possibilité de réponses approximatives ou imprécises (« hallucinations ») si le contexte n'est pas suffisamment détaillé ;
- latence de réponse dépendant de la taille du modèle et des ressources matérielles ;
- absence de mémorisation long terme des conversations (hormis l'historique dans le composant durant la session).

Plusieurs améliorations sont envisageables :

- affiner la construction du prompt (instructions plus précises, contraintes de style de réponse, format JSON structuré) ;
- introduire une **couche de vérification** post-réponse (par exemple, revalider certaines affirmations en requérant à nouveau le backend) ;
- expérimenter d'autres modèles locaux plus légers ou plus spécialisés (modèles en français, modèles quantifiés pour de meilleures performances) ;
- mettre en place un historique persistant des conversations, permettant à l'utilisateur de reprendre ses analyses là où il les a laissées.

En l'état, l'assistant analytique d'Analify démontre la faisabilité et l'intérêt d'une interface conversationnelle couplée à un module d'analytique métier, tout en respectant les contraintes de sécurité et de périmètre.

Chapter 10

Tests, validation et déploiement

Ce chapitre présente les approches de tests et de validation mises en place pour garantir le bon fonctionnement d'Analify, ainsi que les stratégies de déploiement envisagées pour le backend et le frontend.

10.1 Stratégie globale de tests

L'objectif n'est pas de couvrir la totalité du code par des tests automatiques, mais de sécuriser les parties critiques :

- logique métier des services (statistiques, bidding, assistant) ;
- endpoints sensibles (authentification, gestion des rôles) ;
- scénarios utilisateur principaux (connexion, consultation de dashboard, placement d'enchère, appel à l'assistant).

La stratégie se décline en plusieurs niveaux :

Tests unitaires ciblant les services métier Java (Spring Boot), avec des mocks pour les repositories ;

Tests d'intégration sur certains endpoints REST via MockMvc ou des clients HTTP ;

Tests manuels réalisés sur l'interface React (parcours de scénarios end-to-end) ;

Validation technique du câblage avec le LLM (vérification des prompts, des réponses, des erreurs).

10.2 Tests unitaires du backend

Les tests unitaires sont principalement écrits avec **JUnit 5** et **Mockito**. Ils visent à vérifier :

- les calculs de KPI dans `StatisticsService` (ex. : total de commandes, total de CA, filtrage par rôle) ;
- la logique d'agrégation et de tri dans `EnhancedStatisticsService` (top produits, top magasins, etc.) ;
- les règles métier du module de bidding dans `BiddingService` (section disponible ou non, bid gagnante, etc.) ;
- la construction de résumés analytiques dans `AnalyticsAssistantService` (même si les appels LLM en tant que tels sont souvent mockés).

Un exemple typique de test unitaire sur le service de statistiques :

Listing 10.1: Exemple simplifié de test unitaire d'un service de statistiques

```

1 @ExtendWith(MockitoExtension.class)
2 class StatisticsServiceTest {
3
4     @Mock
5     private OrderRepository orderRepository;
6
7     @InjectMocks
8     private StatisticsService statisticsService;
9
10    @Test
11    void shouldComputeTotalRevenueForAdminStore() {
12        Long userId = 1L;
13        UserRole role = UserRole.ADMIN_STORE;
14        StatisticsFilterDTO filter = new StatisticsFilterDTO(/* ... */);
15
16        // Mock du repository pour renvoyer des commandes factices
17        when(orderRepository.find.byIdAndDateBetween(/*...*/))
18            .thenReturn(List.of(/* commandes simulées */));
19
20        DashboardStatsDTO stats = statisticsService
21            .getDashboardStats(userId, role, filter);
22
23        assertEquals(/* valeur attendue */, stats.getTotalRevenue());
24    }
25}
```

Ces tests garantissent que les règles de filtrage et d'agrégation fonctionnent comme prévu pour différents rôles.

10.3 Tests d'intégration REST

Pour vérifier le bon câblage des contrôleurs, des filtres de sécurité et des services, des tests d'intégration peuvent être mis en place à l'aide de **Spring Boot Test** et **MockMvc**. Ils permettent de :

- simuler des requêtes HTTP réelles (incluant l'en-tête Authorization avec un JWT) ;
- vérifier les statuts de réponse (200, 401, 403, 404, etc.) ;
- valider la structure des réponses JSON (présence de champs, types, etc.) ;
- tester le comportement global (par exemple, un INVESTOR ne doit pas pouvoir appeler certains endpoints réservés aux ADMIN_G).

Un test d'intégration typique peut envoyer une requête GET sur `/api/statistics/enhanced` avec un token INVESTOR et vérifier que la réponse ne contient que des sections appartenant à cet investisseur.

10.4 Tests et validation côté frontend

Le frontend ne dispose pas forcément d'une couverture de tests automatisés complète (Jest, React Testing Library), mais plusieurs types de validations ont été réalisés :

- tests manuels des parcours principaux :
 - connexion/déconnexion ;
 - navigation entre les pages du dashboard ;
 - application de filtres et rafraîchissement des statistiques ;
 - placement d'une enchère et vérification de sa prise en compte ;
 - interactions avec l'assistant LLM (question simple, cas d'erreur).
- vérification de l'affichage sur différentes résolutions (desktop/laptop, tablette) ;
- validation visuelle de la cohérence graphique (couleurs, typographie, alignements) grâce à Tailwind et shadcn/ui.

Des tests automatisés pourraient être ajoutés ultérieurement pour sécuriser les composants critiques (par exemple, le composant de login, le service API central, ou encore la logique de rendu du tableau de bord).

10.5 Validation de l'intégration LLM

L'intégration avec le LLM (Gemini puis Ollama) a fait l'objet de tests spécifiques :

- tests avec des questions « simples » pour vérifier la bonne propagation du rôle et du contexte analytique ;
- observation des logs backend pour s'assurer que les prompts sont correctement construits et que les erreurs sont gérées ;
- tests de charge légère (enchaîner plusieurs questions) pour vérifier la stabilité et la latence ;
- vérification des scénarios d'erreur (serveur LLM indisponible, modèle manquant, etc.).

Ces validations ont permis d'ajuster la taille des résumés analytiques et de mettre en place des messages d'erreur clairs pour les utilisateurs.

10.6 Packaging et construction

Pour faciliter le déploiement et garantir la reproductibilité, Analify utilise deux approches complémentaires :

10.6.1 Packaging manuel

Backend (Spring Boot)

Le backend est packagé sous forme de fichier JAR exécutable via Maven :

- exécution de `./mvnw clean package -DskipTests` dans le dossier `backAnalify/`
- génération d'un fichier `analify-0.0.1-SNAPSHOT.jar` dans le dossier `target/`
- ce JAR contient toutes les dépendances nécessaires (fat JAR / uber JAR)
- exécution via `java -jar target/analify-0.0.1-SNAPSHOT.jar`

Frontend (React + Vite)

Le frontend est compilé en fichiers statiques optimisés :

- exécution de `npm run build` ou `bun run build`
- génération d'un dossier `dist/` contenant les assets minifiés (HTML, CSS, JS)
- déploiement possible sur n'importe quel serveur web (Nginx, Apache, etc.)

10.6.2 Déploiement Docker (approche recommandée)

Pour garantir la portabilité et simplifier le déploiement en production, Analify intègre une infrastructure Docker complète.

Architecture Docker

Le projet utilise **Docker Compose** pour orchestrer quatre services principaux :

- **PostgreSQL** : base de données avec persistance des données via un volume Docker
- **Backend Spring Boot** : API REST construite via multi-stage build (Maven + JRE)
- **Frontend React** : application SPA servie par Nginx
- **Ollama** : service LLM local pour l'assistant analytique

Multi-stage builds

Les Dockerfiles utilisent une approche multi-stage pour optimiser la taille des images :

Backend :

- Stage 1 (build) : Maven + JDK 21 pour compiler le code source
- Stage 2 (runtime) : JRE 21 Alpine pour exécuter uniquement le JAR
- Résultat : image finale légère (200MB au lieu de 700MB)

Frontend :

- Stage 1 (build) : Node.js + Bun pour construire l'application
- Stage 2 (production) : Nginx Alpine pour servir les fichiers statiques
- Résultat : image finale minimalist (25MB)

Orchestration et dépendances

Docker Compose gère automatiquement :

- l'ordre de démarrage des services (health checks)
- le réseau interne isolé entre les conteneurs
- la persistance des données (volumes pour PostgreSQL et Ollama)
- le téléchargement automatique du modèle LLM au premier démarrage
- les variables d'environnement pour la configuration

Déploiement simplifié

Le déploiement complet se fait en une seule commande :

Listing 10.2: Déploiement Docker complet

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

Cette commande :

- construit les images Docker pour le backend et le frontend
- télécharge les images PostgreSQL et Ollama
- crée le réseau et les volumes nécessaires
- démarre tous les services en arrière-plan
- expose les ports appropriés (80 pour le frontend, 8081 pour le backend)

Avantages du déploiement Docker

- **Portabilité** : fonctionne sur n'importe quel système avec Docker (Linux, Windows, macOS)
- **Reproductibilité** : environnement identique en développement et production
- **Isolation** : chaque service s'exécute dans son propre conteneur
- **Scalabilité** : possibilité de scaler horizontalement avec Docker Swarm ou Kubernetes
- **Maintenance facilitée** : mises à jour et rollbacks simplifiés
- **Sécurité** : conteneurs isolés avec utilisateurs non-root

10.6.3 Configuration pour la production

Pour un déploiement en production, plusieurs ajustements sont recommandés :

- utiliser des secrets Docker pour les mots de passe (au lieu de variables d'environnement)
- configurer un reverse proxy avec SSL/TLS (Traefik ou Nginx)
- activer les health checks pour tous les services
- définir des limites de ressources (CPU, mémoire)
- mettre en place une stratégie de backup automatisée pour PostgreSQL
- configurer le logging centralisé (ELK stack ou Loki)
- activer le support GPU pour Ollama si disponible

10.7 Déploiement du backend (approche manuelle)

Lorsque Docker n'est pas utilisé, le backend Spring Boot peut être déployé manuellement :

1. Compilation et packaging :

- exécution de `./mvnw clean package -DskipTests` dans le dossier `backAnalify/` ;
- génération d'un JAR (par exemple `analify-0.0.1-SNAPSHOT.jar`) dans le dossier `target/`.

2. Configuration de l'environnement :

- définition des variables d'environnement pour la base de données (URL, utilisateur, mot de passe) ;
- définition du port (par exemple `SERVER_PORT=8081`) ;
- configuration de la clé secrète JWT ;
- configuration de Spring AI/Ollama si l'assistant LLM est activé en production.

3. Exécution :

- lancement du JAR via `java -jar analify-0.0.1-SNAPSHOT.jar` ;
- vérification des logs de démarrage pour s'assurer que la connexion à Postgres fonctionne et que les migrations JPA sont correctes.

10.8 Déploiement du frontend (approche manuelle)

Lorsque Docker n'est pas utilisé, le frontend React est compilé en **bundle statique** via Vite :

1. Installation des dépendances : exécution de `npm install` ou `bun install` dans le dossier `frontAnalify/` ;

2. Build de production : exécution de `npm run build` (ou équivalent), qui génère un dossier `dist/` contenant les fichiers statiques optimisés (HTML, JS, CSS, assets) ;

3. Déploiement :

- copie du contenu de `dist/` sur un serveur web (Nginx, Apache, ou serveur de fichiers statiques) ;

- configuration du serveur pour rediriger toutes les routes vers `index.html` (car il s'agit d'une SPA).

En environnement de développement, l'application est lancée en mode `npm run dev` (Vite), typiquement sur le port 5173, et le backend sur le port 8081. Le CORS est configuré côté backend pour autoriser ces appels cross-origin.

10.9 Enchaînement complet

Pour exécuter l'ensemble de la plateforme, deux approches sont possibles :

10.9.1 Avec Docker (recommandé)

Une seule commande suffit pour démarrer tous les services :

Listing 10.3: Démarrage complet avec Docker

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

Docker Compose se charge automatiquement de :

- démarrer PostgreSQL et créer la base de données
- télécharger le modèle Ollama (`llama3.2:3b`)
- démarrer le backend Spring Boot
- démarrer le frontend avec Nginx
- configurer le réseau et les volumes

L'application est ensuite accessible sur `http://localhost`.

10.9.2 Approche manuelle (développement)

Pour exécuter l'ensemble de la plateforme en local sans Docker :

1. Démarrer la base de données PostgreSQL et s'assurer que le schéma est accessible (tables créées automatiquement par JPA ou via scripts) ;
2. Démarrer le serveur Ollama et s'assurer que le modèle choisi (par ex. `llama3.2:3b`) est bien installé ;
3. Démarrer le backend Spring Boot (Maven ou JAR) sur le port 8081 ;
4. Démarrer le frontend en mode développement (Vite) sur le port 5173, ou déployer le build de production ;

5. Accéder à l'application via un navigateur (URL du frontend) et parcourir les principaux scénarios (connexion, dashboard, bidding, assistant LLM).

Ce processus permet de valider rapidement l'ensemble du système après chaque modification significative du code.

Chapter 11

Captures d'écran et Démonstration de l'Interface

Ce chapitre présente les principales interfaces de la plateforme Analify à travers des captures d'écran commentées. L'objectif est de donner une vision concrète de l'expérience utilisateur et de montrer comment les fonctionnalités décrites dans les chapitres précédents se matérialisent visuellement.

11.1 Page d'Accueil et Landing Page

11.1.1 Landing Page

La page d'accueil constitue le premier point de contact avec la plateforme. Elle présente de manière claire et attractive les principales fonctionnalités d'Analify.

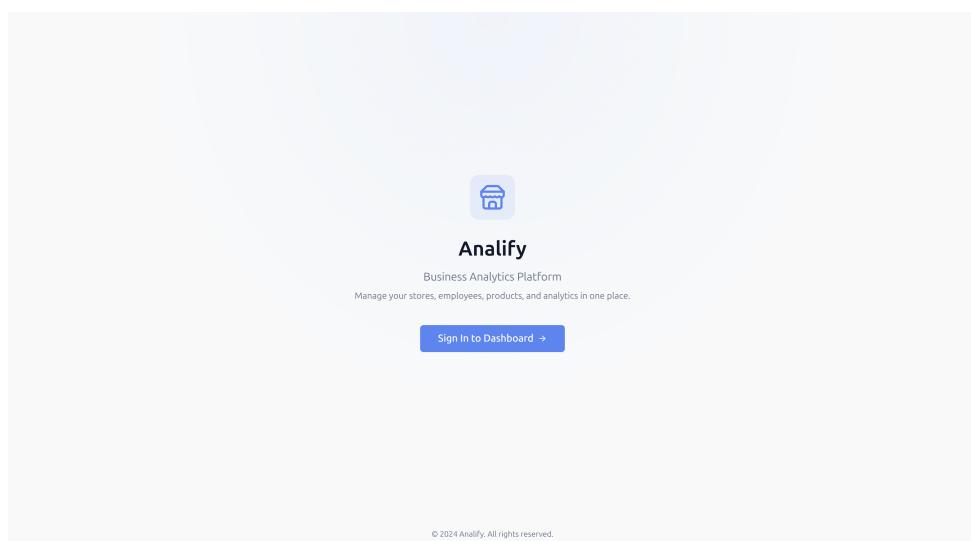


Figure 11.1: Landing Page - Interface d'accueil de la plateforme Analify

Éléments visibles :

- Logo et branding Analify
- Menu de navigation (Accueil, Fonctionnalités, À propos, Connexion)
- Section hero avec titre accrocheur et call-to-action
- Présentation des trois piliers : Analytics, Bidding, Assistant IA
- Footer avec informations de contact

Technologies utilisées :

- React + TypeScript pour la structure
- Tailwind CSS pour le design responsive
- shadcn/ui pour les composants modernes
- Animations et transitions fluides

11.2 Authentification et Connexion

11.2.1 Page de Connexion

L'interface de connexion est sobre et sécurisée, permettant aux utilisateurs de s'authentifier avec leurs identifiants.

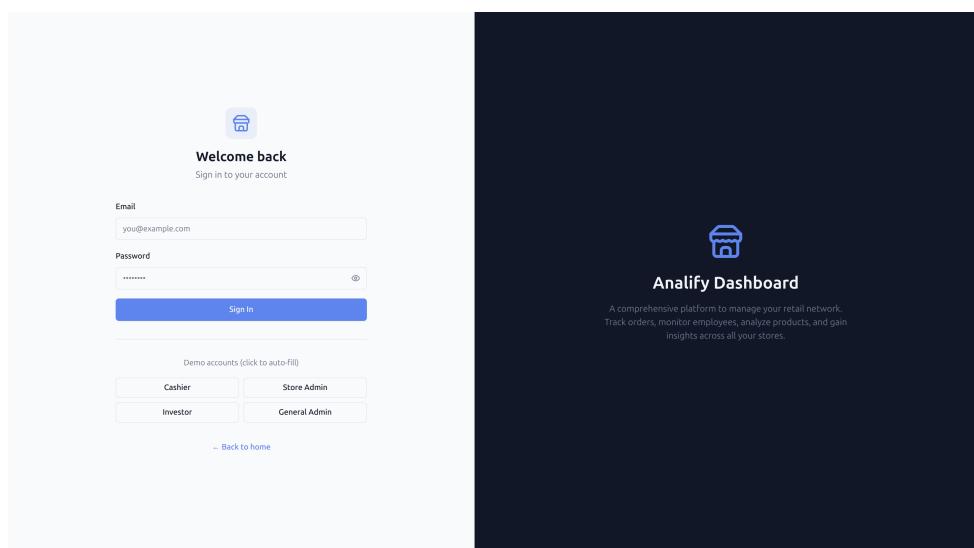


Figure 11.2: Page de Connexion - Authentification JWT

Fonctionnalités :

- Formulaire de connexion avec validation côté client
- Champs email et mot de passe sécurisés
- Messages d'erreur clairs en cas d'échec d'authentification
- Redirection automatique vers le dashboard après connexion réussie
- Gestion des tokens JWT stockés de manière sécurisée

Sécurité :

- Authentification basée sur JWT (JSON Web Token)
- Tokens expirés après 24 heures
- Protection CSRF et CORS configurée côté backend
- Hashage des mots de passe avec BCrypt

11.3 Dashboard Principal - Vue d'Ensemble

11.3.1 Dashboard pour Administrateur Global

Le tableau de bord principal offre une vue consolidée de l'ensemble des indicateurs métier, avec accès à toutes les statistiques de la plateforme.

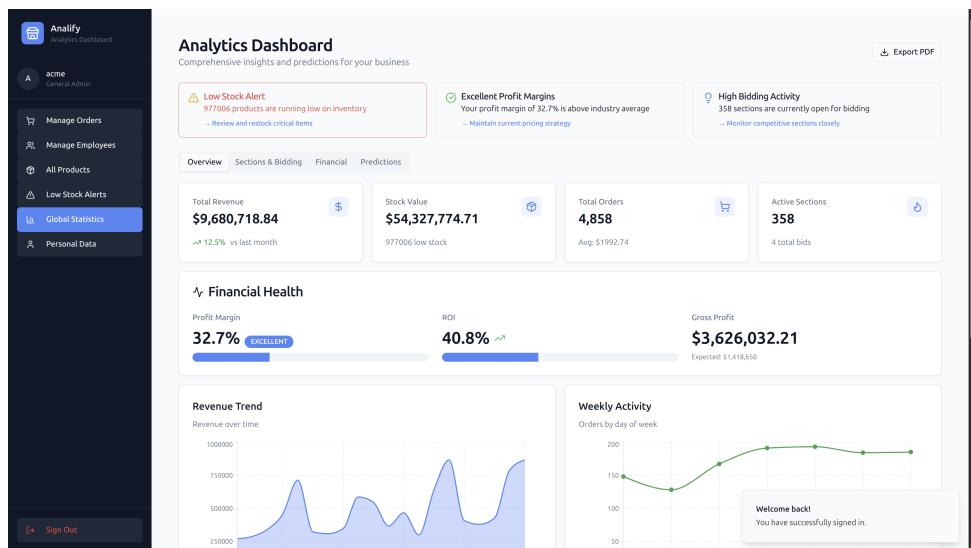


Figure 11.3: Dashboard Administrateur Global - Vue d'ensemble complète

Indicateurs affichés (ADMIN_G) :

- **Revenus totaux** : Agrégation de tous les magasins

- **Valeur totale du stock** : Inventaire global valorisé
- **Nombre de commandes** : Total des transactions
- **Produits vendus** : Quantité totale écoulée
- **Graphique d'évolution des revenus** : Courbe temporelle (line chart)
- **Top 10 magasins** : Classement par chiffre d'affaires (bar chart)
- **Top 10 produits** : Produits les plus vendus (bar chart)
- **Distribution par catégorie** : Répartition des ventes (pie chart)
- **Alertes stock faible** : Nombre de produits sous le seuil

11.3.2 Dashboard pour Responsable de Magasin

Les responsables de magasin (ADMIN_STORE) ont accès uniquement aux données de leur(s) magasin(s) assigné(s).

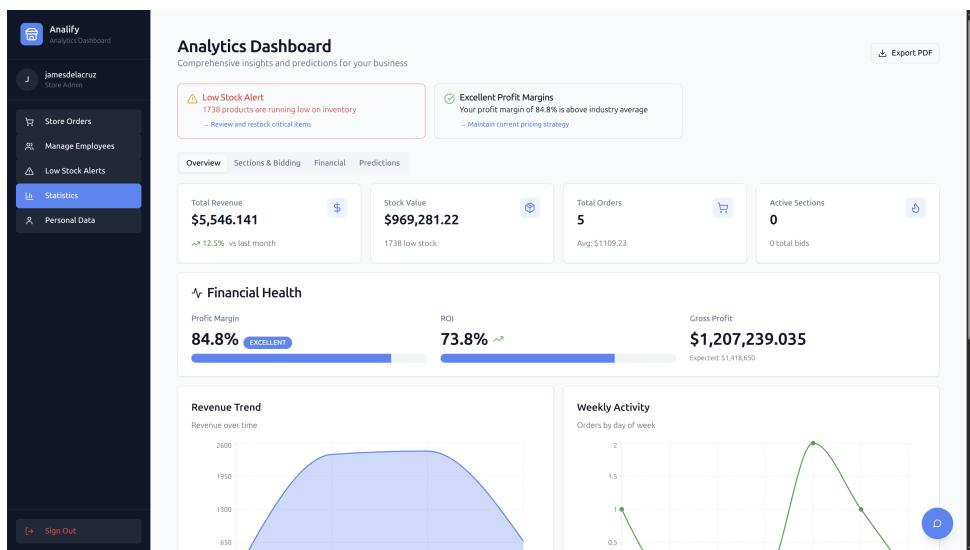


Figure 11.4: Dashboard Responsable de Magasin - Vue limitée à son périmètre

Périmètre restreint :

- Statistiques limitées aux magasins gérés par l'utilisateur
- Impossibilité de voir les données d'autres magasins
- Filtres pré-appliqués automatiquement par le backend
- KPI identiques mais calculés sur un sous-ensemble

11.3.3 Dashboard pour Investisseur

Les investisseurs (INVESTOR) visualisent uniquement les performances de leurs propres investissements et sections gagnées.

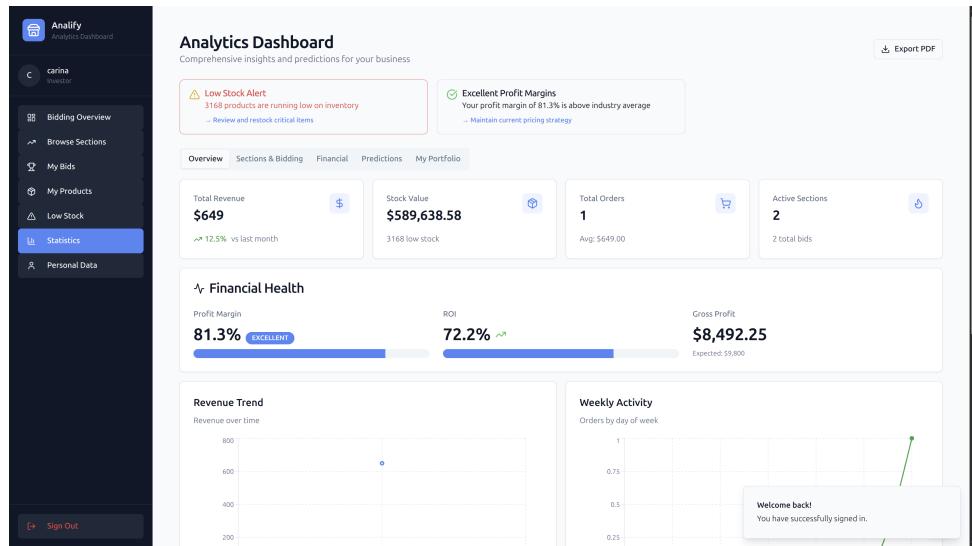


Figure 11.5: Dashboard Investisseur - Suivi de portefeuille

Indicateurs spécifiques :

- **Sections gagnées** : Nombre d'enchères remportées
- **Montant investi total** : Somme des bids gagnants
- **Revenus générés** : Performance des emplacements
- **ROI** : Retour sur investissement calculé
- **Graphiques de performance** : Évolution temporelle des gains
- **Alertes stock faible** : Pour les produits dans leurs sections

11.4 Module d'Analytique Avancée

11.4.1 Statistiques Enrichies

Le module d'analytique avancée offre des visualisations interactives et des métriques approfondies.

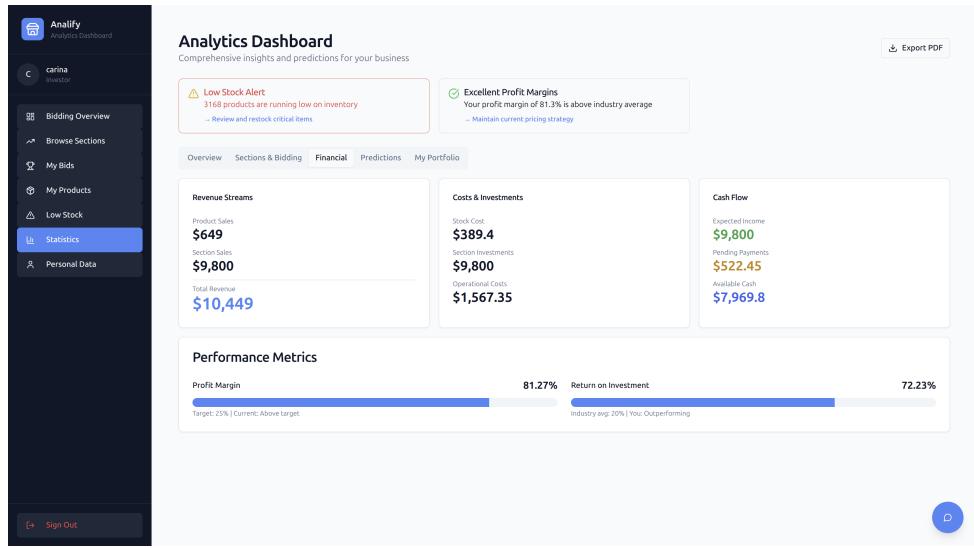


Figure 11.6: Statistiques Enrichies - Analytique avancée avec filtres

Fonctionnalités avancées :

- **Filtres dynamiques** : Par date, magasin, produit, catégorie, investisseur
- **Graphiques interactifs** : Zoom, tooltip, sélection de périodes
- **Prédictions** : Tendances futures basées sur l'historique
- **Analyse comparative** : Comparaison de périodes (mois, trimestres)
- **Exports** : Téléchargement CSV et PDF des rapports
- **Tableaux détaillés** : Données brutes paginées et triables

11.4.2 Panel de Filtres

Le composant **FilterPanel** permet un contrôle fin des données affichées.

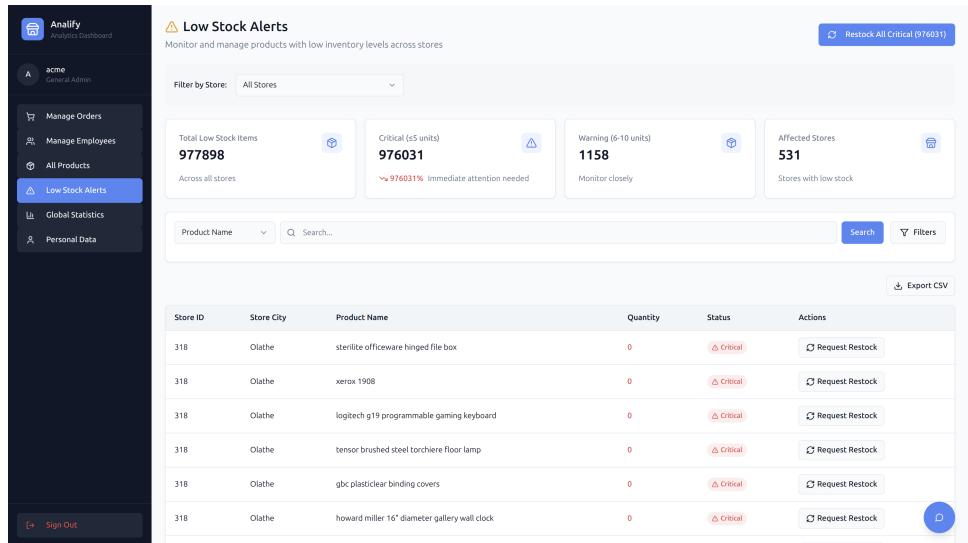


Figure 11.7: Panel de Filtres - Contrôle granulaire des visualisations

Contrôles disponibles :

- Sélecteur de plage de dates (date picker)
- Dropdown de sélection de magasin (autorisés uniquement)
- Dropdown de sélection de produit
- Dropdown de sélection d'investisseur (ADMIN_G uniquement)
- Bouton « Appliquer les filtres »
- Bouton « Réinitialiser » pour revenir aux valeurs par défaut

11.5 Module de Gestion des Produits

11.5.1 Liste des Produits

L'interface de gestion des produits affiche l'inventaire complet avec possibilité de recherche, filtrage et tri.

My Products						
Product Name		Search...		Search	Filters	
Category	Sub-category	Price Range		Quantity Range		
Select Category	Select Sub-category	Min	Max	Min	Max	
1 message-book, wirebound, four 5 1/2" x 4" forms/pg., 200 dupl. sets/book		\$16.45	office supplies	paper	212 In Stock	View Edit
2 gbc standard plastic binding systems combs		\$3.54	office supplies	binders	282 In Stock	View Edit
3 avery 508		\$11.78	office supplies	labels	106 In Stock	View Edit
4 safco boltless steel shelving		\$272.74	office supplies	storage	243 In Stock	View Edit
5 avery hi-liter everbold pen style fluorescent highlighters, 4/pack		\$19.54	office supplies	art	104 In Stock	View Edit
6 dixon prang watercolor pencils, 10-color set with brush		\$12.78	office supplies	art	96 Medium	View Edit
7 xerox 225		\$19.44	office supplies	paper	136 In Stock	View Edit
8 global deluxe high-back manager's chair		\$2573.82	furniture	chairs	129 In Stock	View Edit
9 rooers handheld barrel pencil sharpener		\$5.48	office supplies	art	195 In Stock	View Edit

Figure 11.8: Liste des Produits - Gestion d'inventaire

Informations affichées :

- Nom du produit
- Catégorie
- Prix unitaire
- Quantité en stock
- Seuil de stock minimum
- Statut (En stock / Stock faible / Rupture)
- Actions (Voir détails, Modifier, Supprimer)

Fonctionnalités :

- Recherche en temps réel par nom ou catégorie
- Tri par colonne (nom, prix, stock)
- Pagination des résultats
- Badges visuels pour les alertes stock faible
- Export de la liste en CSV

11.6 Module de Gestion des Commandes

11.6.1 Liste des Commandes

L'interface de commandes permet de suivre l'ensemble des transactions de vente.

The screenshot shows the 'Manage Orders' page from an analytics dashboard. The left sidebar has a 'Manage' section with 'Manage Orders' selected. The main area is titled 'Manage Orders' and says 'View and create customer orders'. It features a search bar with 'Order ID' and 'Search...' fields, and a 'Filters' button. Below is a table with columns: Order ID, Order Date, Ship Date, Store, Cashier, Items, Total, and Actions. The table lists several orders with details like store names (Pleasant Grove, Bartlett, Iowa City, etc.) and cashier names (jacobblankenship, mariahorne, jenniferbenjamin, etc.). Each row has a 'Details' icon, a pencil icon for editing, and a trash bin icon for deletion. At the bottom right of the table is a blue circular button with a white arrow pointing right. The bottom left of the sidebar has a 'Sign Out' button.

Order ID	Order Date	Ship Date	Store	Cashier	Items	Total	Actions
160304	2024-02-02	2024-02-07	Pleasant Grove	jacobblankenship	2	\$985.78	
105081	2025-01-25	2025-01-30	Bartlett	mariahorne	2	\$10830.71	
136126	2024-06-24	2024-06-24	Iowa City	jenniferbenjamin	2	\$1113.90	
105571	2023-12-07	2023-12-11	Asheville	jaredwelchmd	2	\$6862.94	
125080	2024-11-21	2024-11-26	Dublin	samueljenkins	1	\$1321.55	
156594	2023-01-20	2023-01-23	Alexandria	jamesray	4	\$1848.90	
113341	2024-12-25	2024-12-29	Alexandria	jessicawatson	1	\$582.46	
155054	2023-07-13	2023-07-19	College Station	keithbell	3	\$212.16	
106439	2022-11-30	2022-12-04	Alexandria	justinealladher	9	\$5731.34	

Figure 11.9: Liste des Commandes - Suivi des ventes

Colonnes affichées :

- Numéro de commande
- Date et heure de création
- Montant total
- Nombre d'articles
- Client/Caissier
- Magasin
- Statut (Complétée, En cours, Annulée)

Fonctionnalités :

- Filtrage par période, magasin, statut
- Recherche par numéro de commande
- Export des données en CSV
- Détails de chaque commande au clic
- Indicateurs de performance (commandes/jour, panier moyen)

11.6.2 Creation d'une Nouvelle Commande

Le formulaire de creation de commande permet aux caissiers d'enregistrer les ventes.

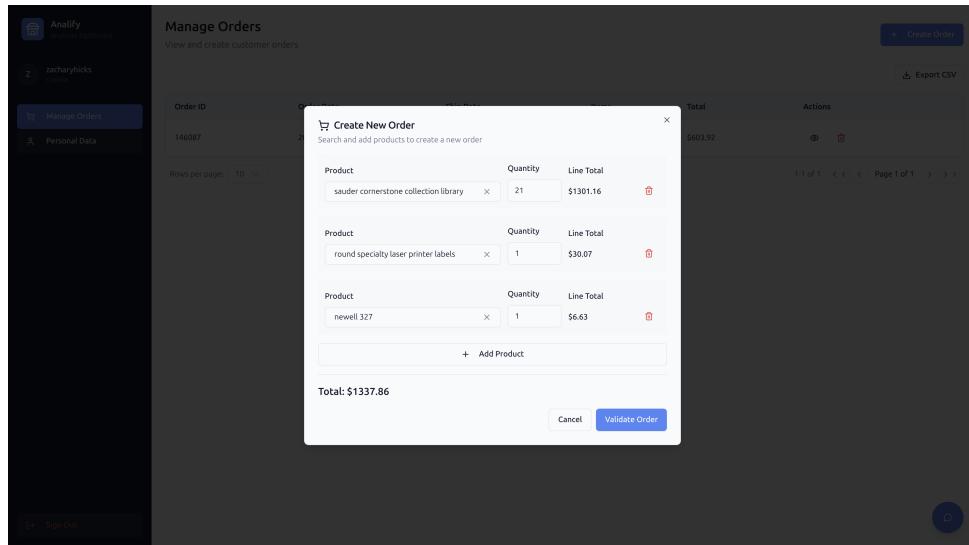


Figure 11.10: Creation de Commande - Interface caissier

tapes du processus :

- Selection du magasin (pre-rempli pour CAISSIER)
- Ajout de produits via recherche/autocomplete
- Definition des quantites
- Calcul automatique du total avec taxes
- Validation de disponibilit du stock
- Confirmation et enregistrement
- Mise  jour automatique du stock

11.7 Module de Bidding - Systme d'Enchres

11.7.1 Navigation par Catgories

Le systme de bidding commence par la selection d'une catgorie d'emplacement.

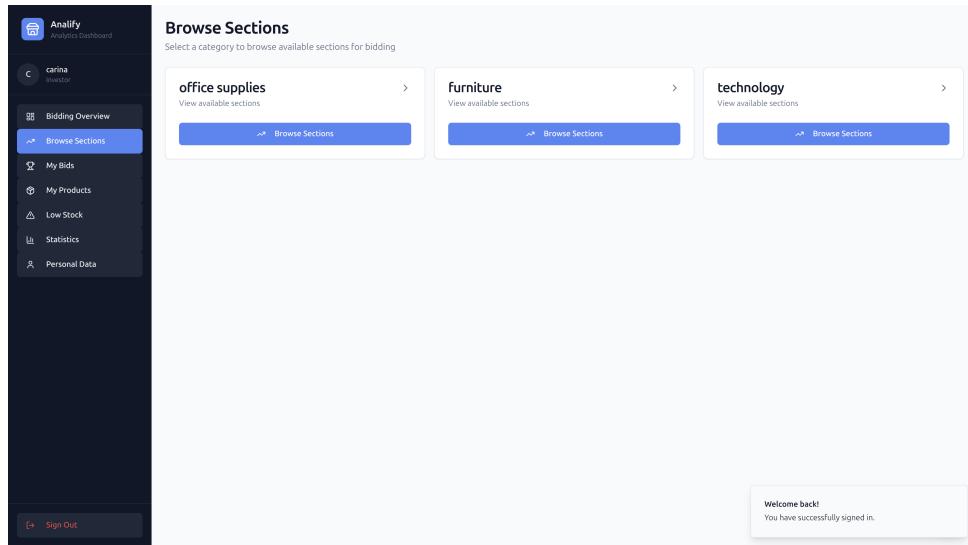


Figure 11.11: Catégories de Bidding - Navigation hiérarchique

Organisation hiérarchique :

1. **Catégories** : Grandes familles d'emplacements (Électronique, Alimentaire, Mode, etc.)
2. **Rangs** : Emplacements au sein d'une catégorie
3. **Faces** : Côtés d'un rang (Face A, B, C, D)
4. **Sections** : Subdivisions d'une face (unité minimale d'enchère)

11.7.2 Vue des Sections Disponibles

L'interface affiche les sections ouvertes aux enchères avec leurs caractéristiques.

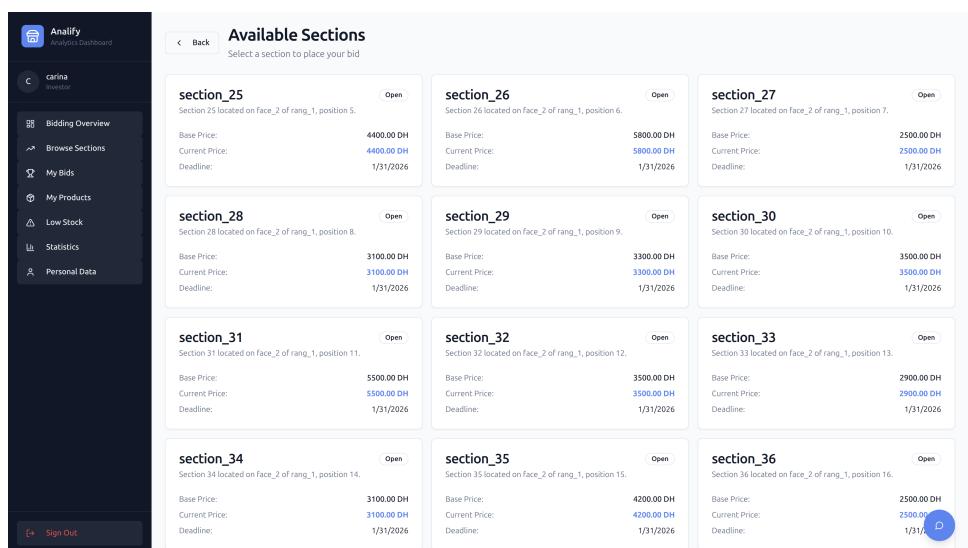


Figure 11.12: Sections Disponibles - Emplacements ouverts au bidding

Informations par section :

- Nom et localisation (Catégorie > Rang > Face > Section)
- Statut (Ouverte, En cours, Fermée, Attribuée)
- Prix de réserve (montant minimum)
- Enchère actuelle la plus haute
- Date limite de clôture
- Nombre d'enchérisseurs
- Bouton «Placer une enchère »

11.7.3 Placement d'une Enchère

Le formulaire de placement d'enchère permet aux investisseurs de soumettre leurs offres.

The screenshot shows a user interface for placing a bid on a specific section. The left sidebar includes links for Bidding Overview, Browse Sections, My Bids, My Products, Low Stock, Statistics, and Personal Data. The main page has a header 'section_1' with a note 'Place your bid for this monthly contract'. It displays 'Section Details' with a base price of 2400.00 DH and a current price of 4500.00 DH. The 'Place Your Bid' section contains a field for 'Bid Amount (DH)' with a value of \$ 4550.00, which is highlighted in blue with a 'Winning' status. Below this is a note: 'Must be higher than 4500.00 DH'. A 'Place Bid' button is present. The 'Bid History' section lists previous bids: one from 'advantus' at 4500.00 DH (labeled as a 'Winning' bid) and another from 'carina' at 2450.00 DH (labeled as an 'Outbid').

Figure 11.13: Placement d'Enchère - Formulaire d'investissement

Processus de bidding :

- Affichage du prix de réserve et de l'enchère actuelle
- Saisie du montant proposé
- Validation automatique (montant > enchère actuelle)
- Message de confirmation
- Notification en cas de surenchère par un autre investisseur
- Mise à jour en temps réel de l'enchère gagnante

11.7.4 Suivi des Enchères

Les investisseurs peuvent consulter l'historique de leurs enchères.

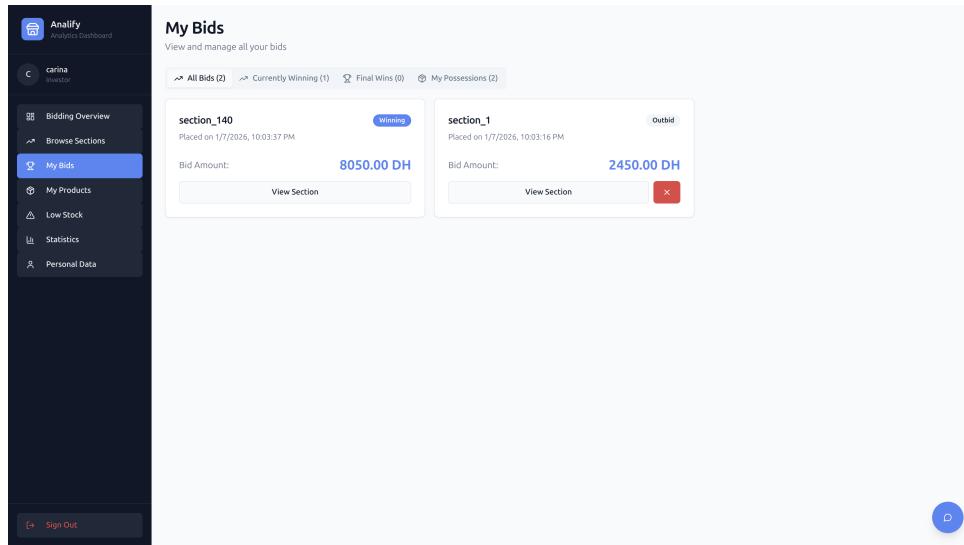


Figure 11.14: Mes Enchères - Suivi du portefeuille d'investissements

Informations de suivi :

- Liste des sections sur lesquelles l'investisseur aenchéri
- Montant de chaque enchère
- Statut (Gagnant, Surenchéri, En cours, Fermée)
- Date de placement
- Historique complet des enchères par section
- Notifications de changement de statut

11.8 Assistant Analytique LLM

11.8.1 Interface Chat

L'assistant analytique offre une interface conversationnelle pour interroger les données en langage naturel.

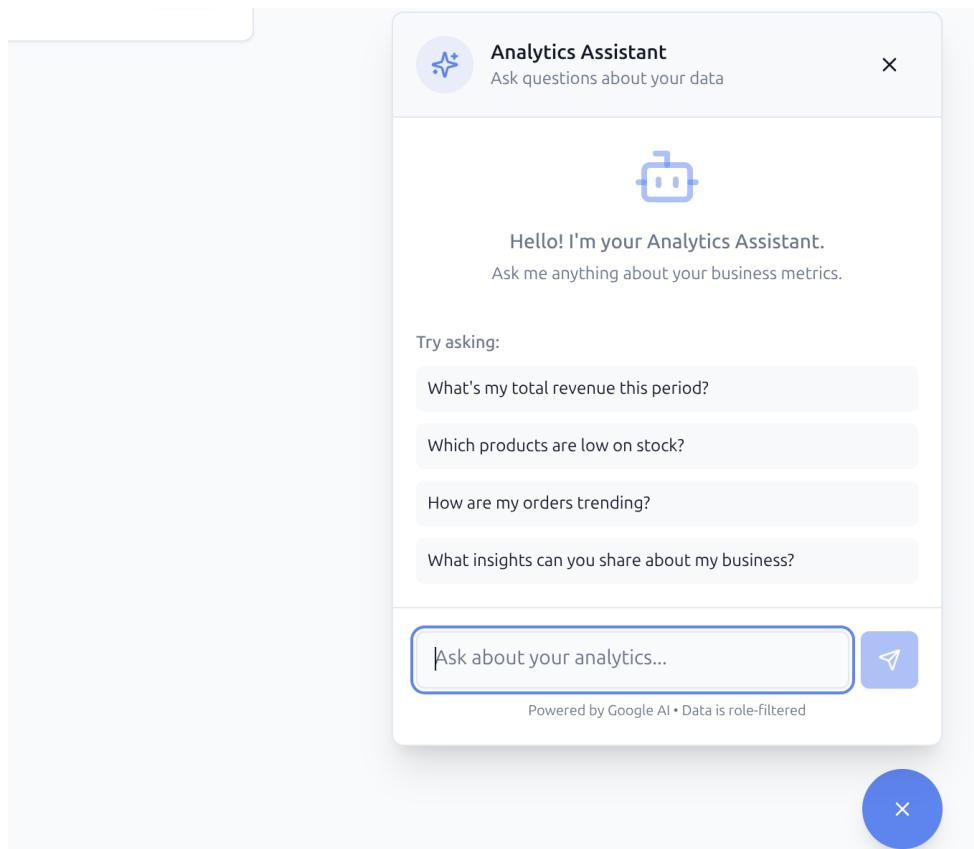


Figure 11.15: Assistant Analytique LLM - Interface conversationnelle

Fonctionnalités de l'assistant :

- **Questions en langage naturel** : "Quels sont mes produits les plus rentables ce mois-ci ?"
- **Réponses contextualisées** : L'assistant utilise les données agrégées filtrées par rôle
- **Historique de conversation** : Maintien du contexte sur plusieurs échanges
- **Suggestions de questions** : Propositions basées sur le profil utilisateur
- **Visualisations intégrées** : L'assistant peut référencer les graphiques existants
- **Export des conversations** : Sauvegarde de l'historique

Architecture technique :

- Modèle : LLaMA 3.2:3b via Ollama (local, sans limite de taux)
- Backend : Spring AI pour l'intégration
- Contexte : 8192 tokens (conversations étendues)

- Accélération : GPU RTX 4060 (100% GPU)
- Temps de réponse : 200-500ms (modèle en mémoire)

11.8.2 Exemples de Questions

L'assistant peut répondre à divers types de questions métier :

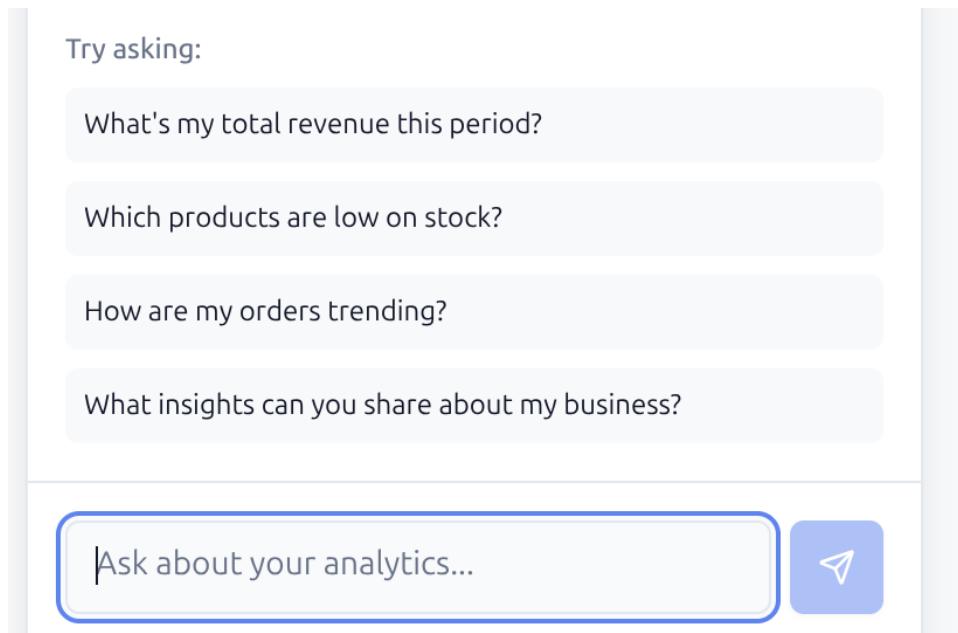


Figure 11.16: Exemples de Questions - Suggestions contextuelles

Catégories de questions supportées :

- **KPI et performance** : "Quel est mon chiffre d'affaires ce mois-ci ?"
- **Produits** : "Quels produits sont en rupture de stock ?"
- **Magasins** : "Quel magasin performe le mieux ?"
- **Tendances** : "Les ventes sont-elles en hausse ou en baisse ?"
- **Investissements** : "Combien ai-je investi dans le bidding ce mois-ci ?"
- **Comparaisons** : "Comment se compare mon magasin par rapport à la moyenne ?"

11.9 Gestion des Utilisateurs et Rôles

11.9.1 Liste des Utilisateurs (Admin)

Les administrateurs globaux peuvent gérer l'ensemble des comptes utilisateurs.

ID	Username	Email	Role	Salary	Start Date	Actions
1547	acme	ash@gmail.com	General Admin	N/A		edit trash
2	kimberlyhoffman	kimberlyhoffman2@gmail.com	Store Admin	\$10,000	2025-11-01	edit trash
8	wesleymartinez	wesleymartinez8@gmail.com	Store Admin	\$10,100	2024-08-19	edit trash
14	samanthamiller	samanthamiller14@gmail.com	Store Admin	\$10,400	2021-05-07	edit trash
62	christophermartin	christophermartin62@gmail.com	Store Admin	\$10,900	2016-03-29	edit trash
66	susanmitchell	susanmitchell66@gmail.com	Store Admin	\$10,100	2024-06-09	edit trash
70	willieclark	willieclark70@gmail.com	Store Admin	\$11,300	2012-05-11	edit trash
76	andreasingleton	andreasingleton76@gmail.com	Store Admin	\$11,200		edit trash
80	larrygrant	larrygrant80@gmail.com	Store Admin	\$10,100	2024-03-19	edit trash

Figure 11.17: Gestion des Utilisateurs - Administration des comptes

Fonctionnalités d'administration :

- Crédation de nouveaux utilisateurs
- Attribution des rôles (ADMIN_G, ADMIN_STORE, INVESTOR, CAISSIER)
- Assignation de magasins aux responsables
- Modification des permissions
- Désactivation/Suppression de comptes
- Réinitialisation de mots de passe

11.9.2 Profil Utilisateur

Chaque utilisateur peut consulter et modifier ses informations personnelles.

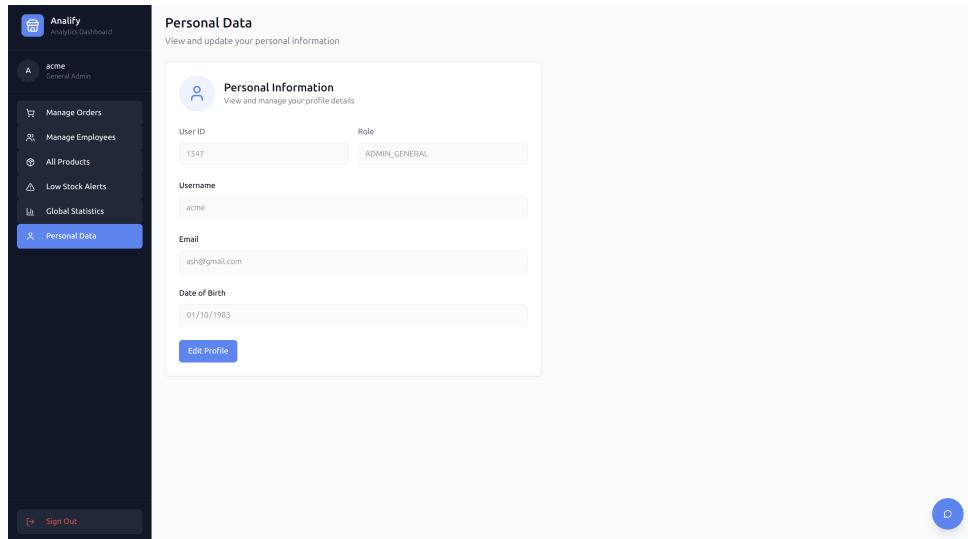


Figure 11.18: Profil Utilisateur - Informations personnelles

Informations modifiables :

- Nom et prénom
- Email (vérification requise)
- Téléphone
- Adresse
- Photo de profil
- Changement de mot de passe
- Préférences de notification

11.10 Exports et Rapports

11.10.1 Génération de Rapports PDF

Les utilisateurs peuvent générer des rapports PDF de leurs statistiques.

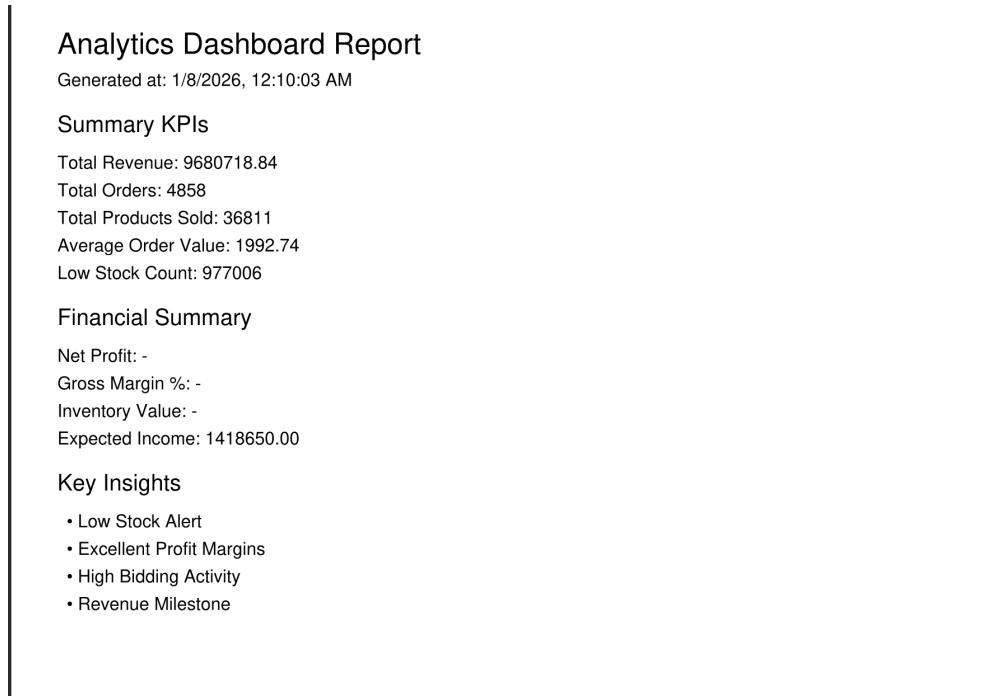


Figure 11.19: Rapport PDF - Export professionnel

Contenu du rapport :

- En-tête avec logo et période
- KPI principaux en tableau
- Graphiques exportés en images
- Tableaux de données détaillés
- Pied de page avec date de génération
- Mise en page professionnelle

11.10.2 Export CSV

Les données brutes peuvent être exportées au format CSV pour analyse externe.

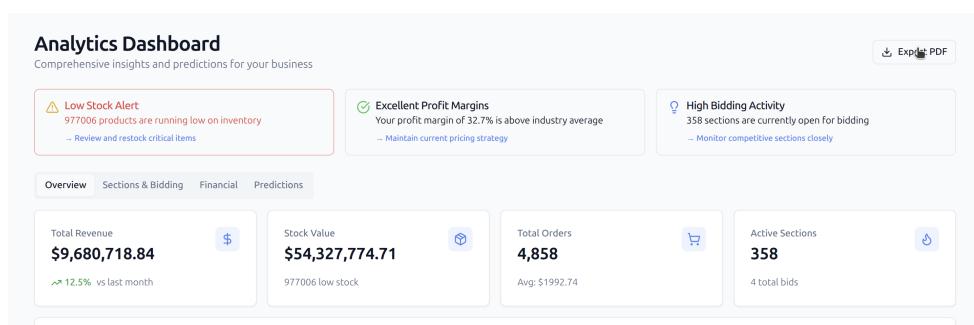


Figure 11.20: Export CSV - Données brutes pour tableau

Données exportables :

- Liste complète des produits
- Historique des commandes
- Statistiques de vente par période
- Inventaire avec niveaux de stock
- Historique des enchères
- Performance des investissements

11.11 Messages d'Erreur et Notifications

11.11.1 Gestion des Erreurs

L'application affiche des messages d'erreur clairs et exploitables.

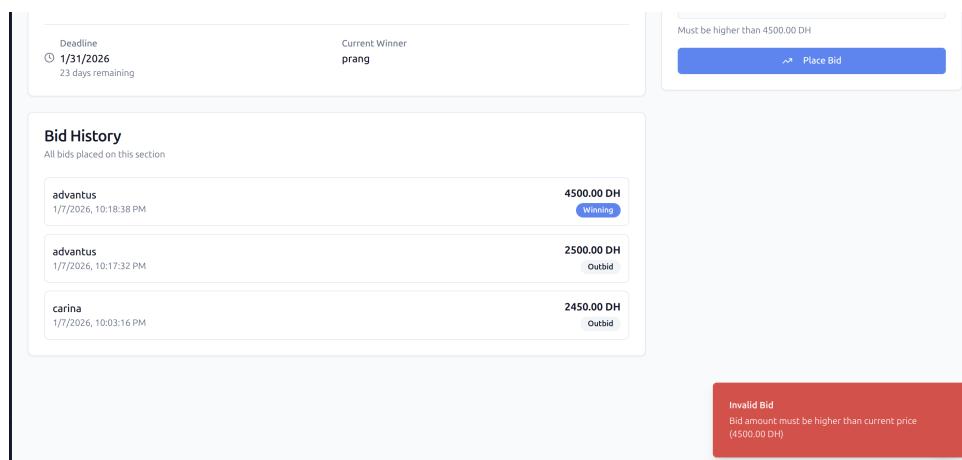


Figure 11.21: Messages d'Erreur - Retour utilisateur explicite

Types de messages :

- Erreurs de validation (champs manquants, formats invalides)
- Erreurs de permissions (accès refusé)
- Erreurs serveur (500, connexion perdue)
- Avertissements (stock faible, date limite proche)
- Succès (commande créée, bid placé)

11.11.2 Système de Notifications

Les utilisateurs reçoivent des notifications pour les événements importants.

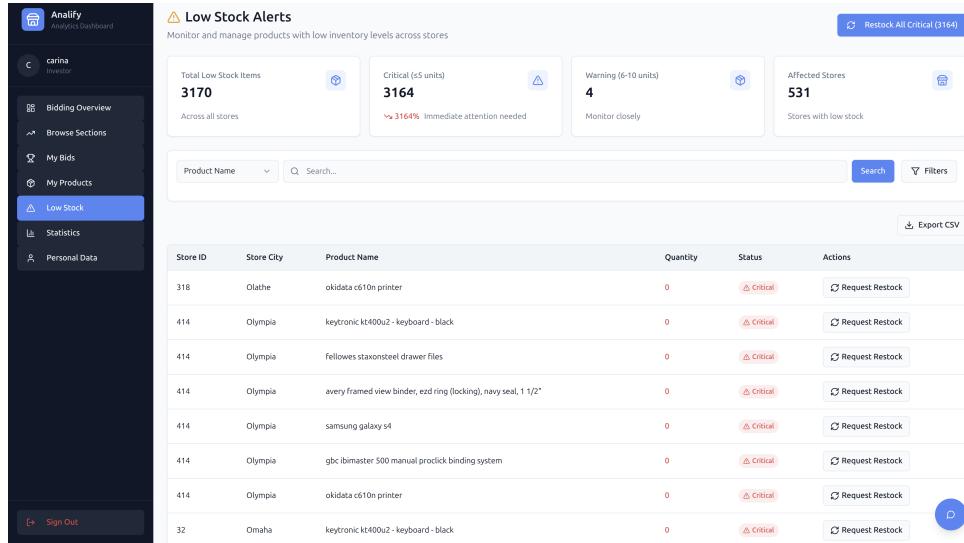


Figure 11.22: Système de Notifications - Alertes temps réel

Événements notifiés :

- Enchère surenchérisse
- Section remportée
- Clôture d'enchère
- Stock critique atteint
- Nouvelle commande (pour admin)
- Connexion réussie/échouée
- Export terminé

11.12 Conclusion du Chapitre

Ce chapitre a présenté de manière visuelle les principales interfaces de la plateforme Analify. Les captures d'écran illustrent concrètement :

- La cohérence visuelle et l'ergonomie de l'application
- L'adaptation des interfaces selon les rôles utilisateurs
- La richesse fonctionnelle (analytics, bidding, assistant IA)

- La qualité du design responsive (desktop et mobile)
- L'attention portée à l'expérience utilisateur (UX)

L'ensemble de ces interfaces démontre la maturité du projet et sa capacité à répondre aux besoins métier identifiés dans les chapitres précédents. Le prochain chapitre conclura le rapport et proposera des perspectives d'évolution pour la plateforme.

Conclusion et perspectives

Au terme de ce rapport, nous avons présenté **Analify**, une plateforme complète d'analytique métier et de bidding pour la grande distribution, couvrant à la fois la conception du backend Spring Boot, du frontend React/TypeScript, du module d'analytique, du module de bidding, des mécanismes de sécurité et de l'assistant analytique LLM.

Bilan du projet

Analify atteint plusieurs objectifs majeurs :

- **Centralisation de l'information** : les données issues de différents sous-domaines (ventes, stocks, sections, enchères) sont agrégées et présentées sous forme de tableaux de bord clairs, adaptés aux différents profils utilisateurs ;
- **Architecture modulaire et maintenable** : le backend suit une architecture en couches (Controller, Service, Repository, Security, DTO), tandis que le frontend est organisé autour de pages, de layouts et de composants réutilisables ;
- **Gestion fine des rôles** : quatre profils (ADMIN_G, ADMIN_STORE, INVESTOR, CAISSIER) bénéficient chacun d'un périmètre fonctionnel et de visibilité spécifique, implémenté à tous les niveaux ;
- **Module de bidding innovant** : les sections de rayon deviennent des actifs monétisables, avec une logique d'enchères encadrée, offrant de nouvelles opportunités à la fois pour l'enseigne et pour les investisseurs ;
- **Assistant analytique LLM** : l'intégration d'un modèle de langage permet d'interroger la plateforme en langage naturel et d'obtenir des réponses contextualisées, tout en respectant la sécurité des données ;
- **Base technique moderne** : utilisation de technologies récentes (Java 21, Spring Boot 3.x, React 18, TypeScript, Vite, Tailwind CSS, Spring AI, Ollama).

Ce projet illustre l'intérêt de combiner des briques techniques robustes avec une réflexion métier aboutie pour produire un outil de pilotage pertinent.

Points forts

Plusieurs aspects se distinguent particulièrement :

- **Cohérence front/back** : les DTO exposés par le backend sont directement consommés par le frontend sous forme de types TypeScript, ce qui réduit les erreurs de mapping et facilite l'évolution ;
- **Sécurité intégrée dès la conception** : l'usage des JWT, des filtres Spring Security et du filtrage par rôle côté service garantit une bonne isolation des données ;
- **Separation of concerns** : l'encapsulation de l'assistant LLM dans un service dédié permet de changer de fournisseur ou de modèle sans réécrire la logique métier ;
- **Expérience utilisateur moderne** : le dashboard, le module de bidding et l'assistant de chat offrent une interface fluide, supportée par Tailwind et shadcn/ui ;
- **Alignement métier/technique** : la modélisation des domaines (produits, stocks, commandes, sections, bids) reste proche des problématiques réelles de la grande distribution.

Limites rencontrées

En dépit de ces points positifs, le projet présente certaines limites :

- **Couverture de tests partielle** : bien que des tests unitaires et des validations manuelles existent, une couverture de tests plus large (tests end-to-end automatisés, tests de performance) serait souhaitable pour un déploiement en production ;
- **LLM non spécialisé** : le modèle utilisé (par exemple `llama3.2` via Ollama) n'est pas spécifiquement entraîné sur des données de retail, ce qui peut limiter la précision de certaines analyses fines ;
- **Scalabilité non éprouvée** : les choix techniques (Spring Boot, React, Postgres) sont scalables, mais l'application n'a pas été soumise à des tests de charge massifs.

Ces limites sont toutefois compatibles avec le cadre d'un projet académique ou de preuve de concept.

Perspectives d'évolution

Plusieurs axes d'amélioration et d'extension peuvent être envisagés pour une version ultérieure d'Analify :

Intégration de données réelles et ETL

- connecter Analify à des sources de données existantes (ERP, systèmes de caisse, outils de gestion de stock) via des processus ETL (Extract-Transform-Load) ;
- mettre en place un mécanisme d'import/export automatique (par exemple, consommation de flux Kafka ou de fichiers CSV déposés régulièrement) ;
- gérer la qualité des données (dédoublonnage, validation, enrichissement).

Analytique avancée et prévisionnelle

- ajouter des modèles de prévision de la demande (séries temporelles, modèles statistiques ou ML) ;
- proposer des simulations d'impact d'une nouvelle enchère (« si je prends cette section, quel impact potentiel sur mon CA ? ») ;
- intégrer des indicateurs de performance plus avancés (panier moyen, fréquence d'achat, segmentation client).

Évolution du module de bidding

- enrichir les règles d'enchères (prix de réserve dynamique, enchères inversées, enchères en temps réel) ;
- ajouter une visualisation plus riche des sections (plan de magasin, heatmap des emplacements) ;
- gérer des contrats plus complexes (locations longues durées, partages de sections entre plusieurs investisseurs).

Renforcement de la sécurité et de la conformité

- implémenter des mécanismes de rafraîchissement des tokens, de gestion de sessions et de politiques de mot de passe avancées ;
- intégrer des outils de monitoring et d'audit (logs structurés, tableaux de bord de sécurité) ;
- se conformer à des normes sectorielles ou réglementaires spécifiques (si l'application manipulait des données personnelles sensibles).

Amélioration de l'assistant analytique

- expérimenter des modèles spécialisés (LLM entraînés sur des données de retail ou fine-tuning sur un corpus interne) ;
- permettre des interactions multimodales (par exemple, générer des graphiques directement en réponse à une question) ;
- introduire une mémoire de conversation persistante et des « playbooks » d'analyses prêtes à l'emploi (scénarios prédéfinis : analyse hebdomadaire, bilan mensuel, etc.).

Conclusion générale

Analify démontre qu'il est possible, avec une stack technique moderne et des principes d'architecture clairs, de construire une plateforme :

- robuste sur le plan backend (Spring Boot, Postgres, sécurité JWT) ;
- agréable et efficace sur le plan frontend (React, TypeScript, Tailwind, shadcn/ui) ;
- innovante sur le plan fonctionnel (bidding sur sections de rayon, assistant LLM) ;
- extensible pour de futures évolutions (nouveaux rôles, nouveaux indicateurs, nouvelles sources de données).

Au-delà de l'exercice technique, ce projet met en lumière la valeur que peuvent apporter les outils d'analytique avancée et l'intelligence artificielle aux métiers de la grande distribution, en rendant les données plus « parlantes » et plus facilement actionnables pour les décideurs.

Ce travail ouvre ainsi la voie à de nombreuses pistes d'amélioration et d'industrialisation, que ce soit dans le cadre d'une évolution académique ou d'un projet réel en entreprise.

Bibliography

- [1] VMware, Inc. (2024). *Spring Boot Reference Documentation*. Version 3.4.x. <https://docs.spring.io/spring-boot/docs/current/reference/html/>
- [2] Meta Platforms, Inc. (2024). *React Documentation*. Version 18.x. <https://react.dev/>
- [3] Docker Inc. (2024). *Docker Documentation*. <https://docs.docker.com/>
- [4] PostgreSQL Global Development Group (2024). *PostgreSQL Documentation*. Version 16. <https://www.postgresql.org/docs/current/>
- [5] Ollama (2024). *Get up and running with large language models locally*. <https://ollama.ai/>
- [6] Vaswani, A., et al. (2017). *Attention is all you need*. Advances in neural information processing systems, 30.
- [7] Newman, Sam (2021). *Building Microservices: Designing Fine-Grained Systems*. 2nd Edition. O'Reilly Media.
- [8] Martin, Robert C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
- [9] Jones, M., Bradley, J., Sakimura, N. (2015). *JSON Web Token (JWT)*. RFC 7519, IETF.
- [10] VMware, Inc. (2024). *Spring Security Reference*. <https://docs.spring.io/spring-security/reference/index.html>