Rapport Arc42 Lab6 - Système de Gestion de Magasin

Liens vers les dépôts Github

- Lab0 : https://github.com/Minh-Khoi-Le/log430-lab0.git
- Lab1 : https://github.com/Minh-Khoi-Le/log430-lab1-cli.git
- Lab2 : https://github.com/Minh-Khoi-Le/log430-lab2.git
- Lab3 : https://github.com/Minh-Khoi-Le/log430-lab2/releases/tag/lab3
- Lab4 : https://github.com/Minh-Khoi-Le/log430-lab4.git
- Lab5 : https://github.com/Minh-Khoi-Le/log430-lab5.git
- Lab6 : https://github.com/Minh-Khoi-Le/log430-lab6.git

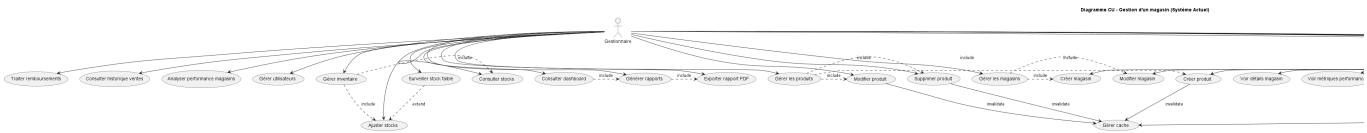
Table des Matières

---

1. Introduction et Objectifs

1.1 Aperçu des Exigences

Le système de gestion de magasin de détail développé pour le LOG430 Lab 6 est une application complète basée sur une architecture microservices. Il fournit une solution complète pour la gestion des opérations de vente au détail incluant l'authentification des utilisateurs, le catalogue de produits, le suivi des stocks, les transactions de vente distribuées avec pattern Saga, et les fonctionnalités administratives.



Le diagramme de cas d'usage ci-dessus présente les principales fonctionnalités du système organisées par acteur :

Utilisateur Client :

- Authentification et gestion de profil
- Consultation du catalogue de produits
- Effectuer des achats
- Consulter l'historique des transactions

Administrateur :

- Gestion des produits et du catalogue
- Gestion des magasins et des stocks
- Traitement des remboursements
- Génération de rapports et analytics

1.2 Objectifs de Qualité

Priorité	Objectif de Qualité	Motivation	Réalisation
1	**Observabilité**	Surveillance complète avec Prometheus et Grafana pour comprendre le comportement du système	Monitoring des Four Golden Signals, métriques personnalisées, dashboards temps réel
2	**Performance**	Optimisation des temps de réponse et gestion de la charge	Caching Redis multiniveau, architecture centralisée de base de données, tests de charge k6
3	**Évolutivité**	Architecture microservices permettant la mise à l'échelle indépendante des services	Décomposition par domaine métier, API Gateway Kong, conteneurisation Docker
4	**Maintenabilité**	Code TypeScript avec architecture DDD et patterns Clean Architecture	Structure modulaire, séparation des responsabilités, documentation Arc42 complète
5	**Sécurité**	Authentification JWT et contrôle d'accès via Kong Gateway	API Keys, rate limiting, validation des requêtes, CORS configuré
6	**Fiabilité**	Système stable avec gestion d'erreurs et validation robuste	Validation cross-domain, transactions ACID, gestion gracieuse des pannes

1.3 Parties Prenantes

Rôle	Attentes	Responsabilités
Développeur	Code maintenable, architecture claire	Implémentation des fonctionnalités
Administrateur Système	Déploiement facile, surveillance	Gestion de l'infrastructure

Rôle	Attentes	Responsabilités
Utilisateur Client	Interface intuitive, performances	Utilisation des fonctionnalités e-commerce
Utilisateur Admin	Outils d'analyse, rapports	Gestion du magasin et des produits

2. Contraintes d'Architecture

2.1 Contraintes Techniques

Contrainte	Description	Impact
Environnement Docker	Tous les services doivent être containerisés	Standardisation du déploiement
PostgreSQL	Base de données relationnelle imposée	Cohérence des données ACID
Node.js + TypeScript	Stack technologique backend	Écosystème JavaScript unifié
Windows	Environnement de développement	Scripts .bat pour l'automatisation

2.2 Contraintes Organisationnelles

- **Projet Académique** : Ressources limitées, focus sur l'apprentissage
- **Délai de Développement** : Développement en temps limité
- **Équipe Solo** : Un seul développeur pour l'implémentation complète

2.3 Contraintes Conventionnelles

- **Patterns DDD** : Implémentation obligatoire des concepts Domain-Driven Design
- **Architecture Microservices** : Décomposition en services indépendants
- **Surveillance** : Monitoring complet avec les "Four Golden Signals"

---

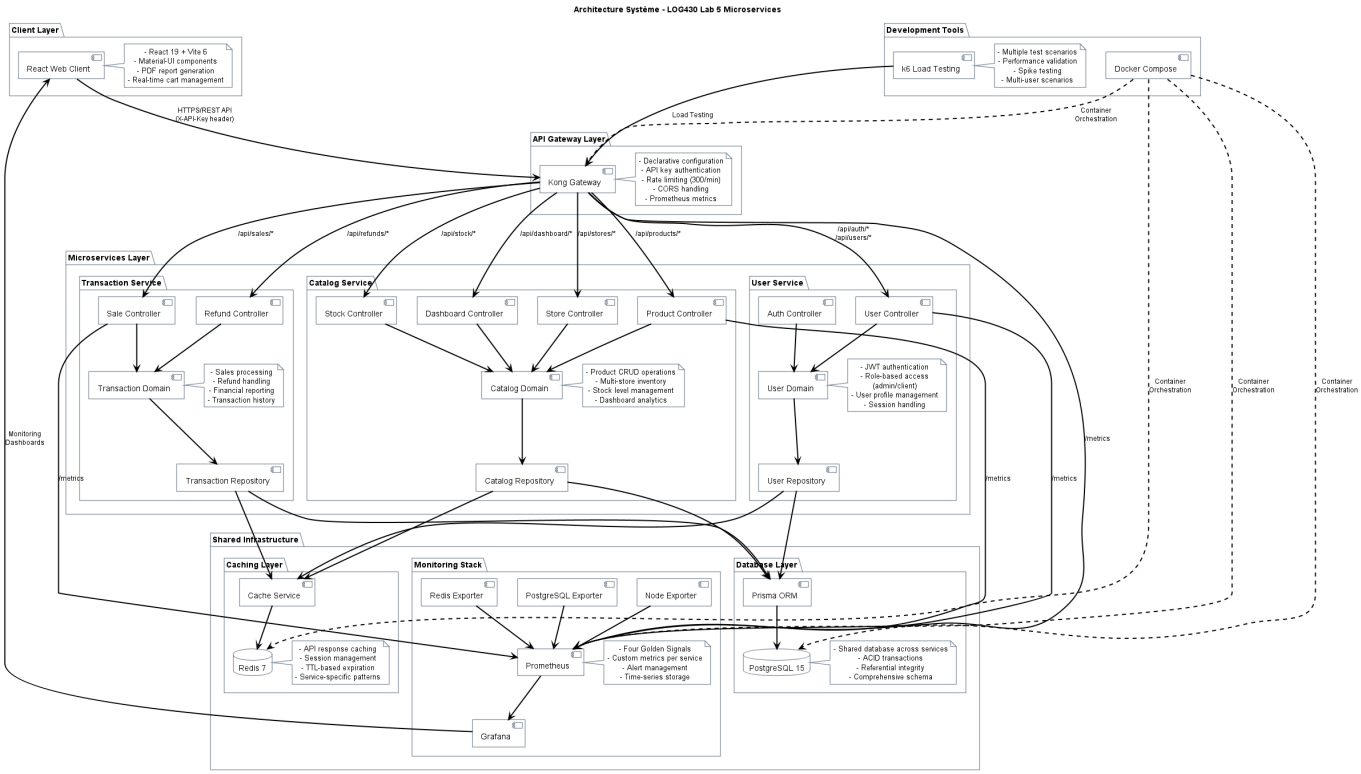
3. Contexte et Portée du Système

3.1 Contexte Métier

Le système adresse les besoins d'une chaîne de magasins de détail nécessitant :

- **Gestion Centralisée** : Catalogue de produits unifié
- **Inventaire Multi-Magasins** : Suivi des stocks par localisation
- **Transactions Sécurisées** : Ventes et remboursements avec audit
- **Analyse des Performances** : Métriques et rapports pour la prise de décision

3.2 Diagramme de Contexte



Le diagramme ci-dessus illustre l'architecture globale du système de gestion de magasin avec ses composants principaux :

- **Client Web** (React 19 + Material-UI) sur localhost:5173
- **Kong Gateway** (API Gateway) sur localhost:8000 avec rate limiting
- **Microservices** : User Service (:3001), Catalog Service (:3002), Transaction Service (:3003), Saga Orchestrator Service (:3004)
- **Infrastructure** : PostgreSQL 15 (:5432), Redis 7 (:6379), Monitoring (Prometheus + Grafana)
- **Tests & Validation** : k6 Load Testing, Unit Tests (Jest), Integration Tests

3.3 Frontières du Système

Inclus dans le système :

- Gestion des utilisateurs et authentification
- Catalogue de produits et inventaire
- Transactions de vente et remboursements
- Surveillance et métriques
- Interface web cliente

Exclus du système :

- Intégration avec des systèmes de paiement externes
- Gestion des fournisseurs
- Logistique et livraison
- Systèmes comptables externes

---

4. Stratégie de Solution

4.1 Approche Architecturale

L'architecture suit une approche **microservices moderne** avec infrastructure centralisée, intégrant les principes suivants :

1. **Décomposition par Domaine Métier** : Chaque service gère un domaine spécifique (User, Catalog, Transaction, Saga Orchestration)
2. **Infrastructure de Base de Données Centralisée** : PostgreSQL partagé avec frontières de domaine strictes et validation cross-domain
3. **API Gateway Centralisé** : Kong pour la gestion des préoccupations transversales (sécurité, monitoring, rate limiting)
4. **Surveillance Complète** : Observabilité avec Prometheus, Grafana et Four Golden Signals
5. **Stratégie de Cache Multiniveau** : Redis avec invalidation automatique et stratégies spécifiques par service
6. **Tests de Performance Intégrés** : Suite k6 complète avec scénarios multiples (spike, stress, e2e)

4.2 Patterns Architecturaux Principaux

Pattern	Application	Bénéfices	Implémentation
Microservices	Décomposition en 4 services indépendants	Évolutivité, maintenabilité, déploiement indépendant	User, Catalog, Transaction, Saga Orchestrator services avec domaines métier bien définis
API Gateway	Kong pour le routage centralisé	Sécurité, surveillance, gestion du trafic	Rate limiting (300/min), API keys, métriques Prometheus
Domain-Driven Design	Structure interne des services	Cohérence métier, séparation des responsabilités	Entities, Use Cases, Repositories par domaine
Clean Architecture	Séparation des couches	Testabilité, flexibilité, indépendance des frameworks	Domain → Application → Infrastructure
CQRS Léger	Séparation lecture/écriture	Performance, optimisation des requêtes	Repositories spécialisés avec cache pour les lectures
Saga Orchestration	Gestion transactions distribuées	Cohérence finale, compensation automatique	Workflows de vente avec états validés et compensation en cas d'échec
Centralized Database	Base de données partagée avec frontières	Performance optimisée, cohérence ACID	Connection pooling, validation cross-domain
Cache-Aside	Stratégie de mise en cache	Réduction latence, amélioration throughput	Redis avec TTL et invalidation automatique

4.3 Technologies Clés

Frontend & Interface Utilisateur :

- **React 19** avec **Vite 6** pour un développement moderne et performant
- **Material-UI v7** pour une interface utilisateur cohérente et accessible
- **React Router DOM v7** pour la navigation client-side
- **Recharts** pour la visualisation de données et l'analytics
- **jsPDF** pour la génération de rapports PDF

Backend & Microservices :

- **Node.js 18+** avec **TypeScript** pour la robustesse et la maintenabilité
- **Express.js** comme framework web léger et performant
- **Domain-Driven Design (DDD)** pour l'organisation du code métier
- **Clean Architecture** avec injection de dépendances
- **Prisma ORM v5** pour l'accès type-safe à la base de données

Infrastructure & Données :

- **PostgreSQL 15** comme base de données relationnelle ACID
- **Redis 7** pour le cache distribué et la gestion de session
- **Kong Gateway** pour l'API Gateway avec sécurité intégrée
- **Docker & Docker Compose** pour la conteneurisation

Observabilité & Monitoring :

- **Prometheus** pour la collecte de métriques
- **Grafana** pour la visualisation et les dashboards
- **Four Golden Signals** (Latency, Traffic, Errors, Saturation)
- **Exporters** spécialisés (Node, PostgreSQL, Redis)

Tests & Qualité :

- **k6** pour les tests de charge et performance
- **Jest** pour les tests unitaires et d'intégration
- **Scénarios multiples** : spike, stress, endurance, e2e

---

5. Vue d'Ensemble de l'Architecture

5.1 Décomposition en Microservices

5.1.1 User Service (Port 3001)

Responsabilités :

- Authentification et autorisation avec JWT
- Gestion des profils utilisateur et rôles (admin/client)
- Validation des tokens et sessions
- Contrôle d'accès basé sur les rôles

- Interface avec l'infrastructure de base de données centralisée

API Endpoints :

- `POST /api/auth/login` - Connexion utilisateur avec génération JWT
- `POST /api/auth/register` - Inscription utilisateur avec validation
- `GET /api/users/profile` - Profil utilisateur authentifié
- `GET /api/users` - Liste des utilisateurs (admin seulement)
- `PUT /api/users/:id` - Mise à jour profil utilisateur

Accès aux Données :

- **Direct** : Entités User uniquement
- **Cross-Domain** : Aucun accès direct aux autres domaines

5.1.2 Catalog Service (Port 3002)

Responsabilités :

- Gestion du catalogue de produits complet
- Gestion des informations de magasin et localisations
- Suivi des stocks d'inventaire en temps réel
- Réservation de stock pour les ventes
- Optimisation des requêtes avec cache Redis
- Analytics et rapports d'inventaire

API Endpoints :

- `GET /api/products` - Liste des produits (avec cache)
- `POST /api/products` - Création de produit (admin)
- `GET /api/products/search` - Recherche de produits
- `GET /api/stores` - Liste des magasins (endpoint public)
- `GET /api/stock/store/:storeId` - Stock par magasin
- `POST /api/stock/reserve` - Réservation de stock
- `POST /api/stock/adjust` - Ajustement d'inventaire

Accès aux Données :

- **Direct** : Entités Product, Store, Stock
- **Cross-Domain** : Validation User pour les opérations admin

5.1.3 Transaction Service (Port 3003)

Responsabilités :

- Traitement des transactions de vente avec validation complète
- Gestion des remboursements avec règles métier
- Historique des transactions par utilisateur et magasin
- Communication avec Catalog Service pour mise à jour stock
- Génération de rapports financiers et analytics
- Validation cross-domain pour utilisateurs, produits et magasins

API Endpoints :

- `POST /api/sales` - Nouvelle vente avec validation complète
- `GET /api/sales/user/:userId` - Historique des ventes utilisateur
- `GET /api/sales/store/:storeId` - Ventes par magasin
- `GET /api/sales/summary` - Résumé des ventes avec métriques
- `POST /api/refunds` - Demande de remboursement
- `PUT /api/sales/:id/status` - Mise à jour statut vente

Accès aux Données :

- **Direct** : Entités Sale, SaleLine, Refund, RefundLine
- **Cross-Domain** : Validation User, Product, Store via ICrossDomainQueries

5.1.4 Saga Orchestrator Service (Port 3004)

Responsabilités :

- Orchestration des workflows de vente distribuée multi-services
- Gestion des états de saga avec machine à états validée
- Exécution automatique des compensations en cas d'échec
- Coordination des étapes : vérification stock → réservation → paiement → confirmation
- Monitoring et observabilité des transactions distribuées
- Gestion des timeouts et retry policies

API Endpoints :

- `POST /api/sagas/sales` - Démarrage d'un workflow de vente orchestrée
- `GET /api/sagas/:correlationId` - Statut et historique d'un workflow saga
- `GET /api/sagas` - Informations sur le service et santé
- `GET /health` - Health check simple
- `GET /health/detailed` - Health check détaillé avec dépendances
- `GET /metrics` - Métriques Prometheus pour workflows saga

Workflow de Vente Orchestrée :

1. **Stock Verification Step** - Vérification de disponibilité via Catalog Service
2. **Stock Reservation Step** - Réservation temporaire avec timeout (30s)
3. **Payment Processing Step** - Traitement paiement via Transaction Service
4. **Order Confirmation Step** - Création vente finale et confirmation

Gestion des États :

```
enum SagaState {
  INITIATED,                    // Saga initiée
  STOCK_VERIFYING,              // Vérification stock en cours
  STOCK_VERIFIED,               // Stock vérifié et disponible
  STOCK_RESERVING,              // Réservation stock en cours
  STOCK_RESERVED,               // Stock réservé temporairement
  PAYMENT_PROCESSING,           // Traitement paiement en cours
```

```
  PAYMENT_PROCESSED,           // Paiement traité avec succès
  ORDER_CONFIRMING,            // Confirmation commande en cours
  SALE_CONFIRMED,              // Vente confirmée (état final succès)
  COMPENSATING_STOCK,          // Compensation stock en cours
  COMPENSATING_PAYMENT,        // Compensation paiement en cours
  COMPENSATED,                 // Compensation terminée
  FAILED                       // Échec final après compensation
}
```

Stratégie de Compensation :

- **Stock réservé mais paiement échoué** → Libération automatique du stock réservé
- **Paiement traité mais confirmation échouée** → Remboursement ou crédit client
- **Timeout de workflow** → Compensation complète selon l'étape atteinte
- **Erreur service externe** → Retry automatique puis compensation si échec persistant

Accès aux Données :

- **Direct** : Entités Saga, SagaStepLog pour traçabilité complète
- **Cross-Domain** : Communication avec Catalog Service et Transaction Service
- **Cache** : États temporaires en Redis pour performance et résilience

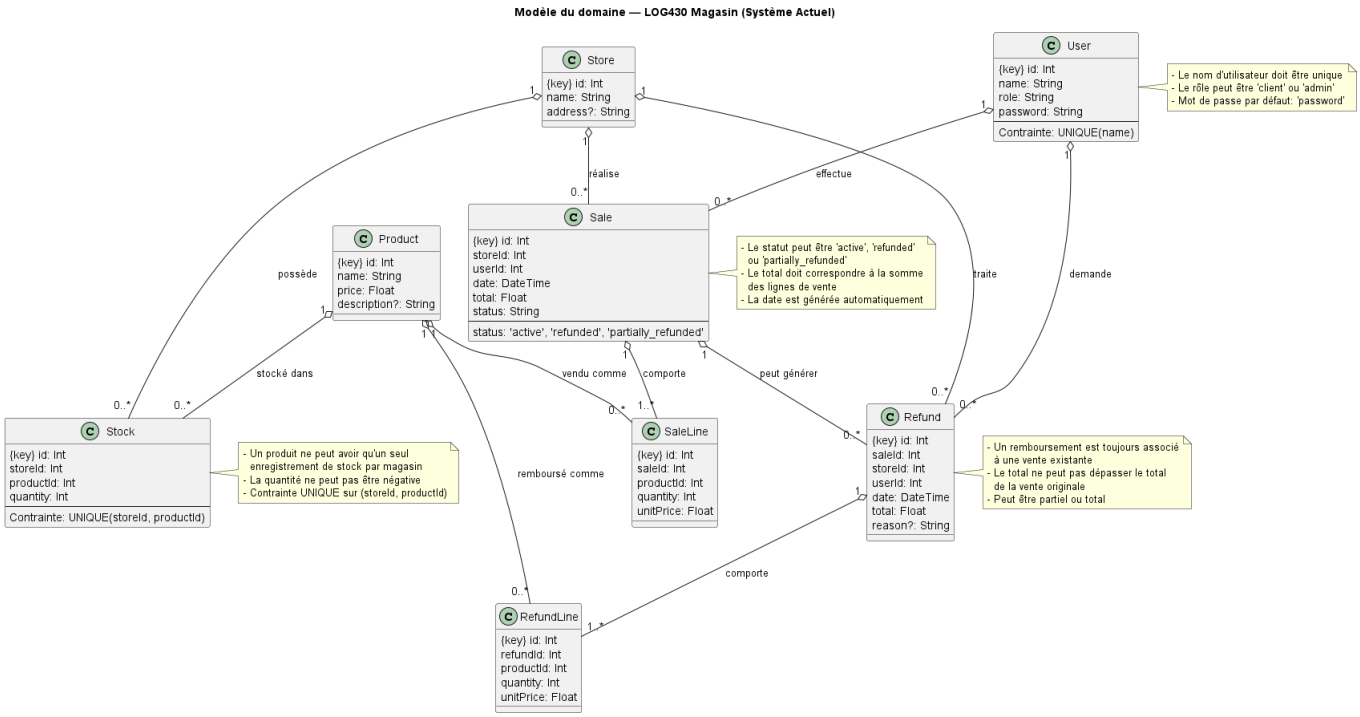
5.2 Architecture de Base de Données Centralisée

5.2.1 Principe d'Architecture

L'implémentation utilise une **infrastructure de base de données centralisée** avec les caractéristiques suivantes :

- **Connection Pool Partagé** : Une seule instance Prisma Client optimisée
- **Frontières de Domaine Strictes** : Accès contrôlé par service via repositories spécialisés
- **Validation Cross-Domain** : Interface `ICrossDomainQueries` pour les validations inter-services
- **Performance Optimisée** : Réduction de la surcharge de connexion et optimisation des requêtes

5.2.2 Modèle de Données Complet



Le modèle de domaine ci-dessus illustre les entités principales et leurs relations dans le système de gestion de magasin. Le schéma Prisma correspondant est le suivant :

```
// Utilisateur du système avec rôles
model User {
  id       Int     @id @default(autoincrement())
  name     String  @unique
  role     String  @default("client") // "admin" | "client"
  password String  // Hash bcrypt
  sales    Sale[]   // Relation vers les ventes
  refunds  Refund[] // Relation vers les remboursements
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

// Magasin physique avec localisation
model Store {
  id      Int      @id @default(autoincrement())
  name    String   @unique
  address String?  // Adresse physique du magasin
  stocks  Stock[]  // Inventaire du magasin
  sales   Sale[]   // Ventes dans ce magasin
  refunds Refund[] // Remboursements traités dans ce magasin
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

// Produit du catalogue
model Product {
  id          Int           @id @default(autoincrement())
  name        String        @unique
  price       Float         // Prix unitaire
```

```
  description String?       // Description détaillée
  stocks      Stock[]       // Stock dans différents magasins
  saleLines   SaleLine[]    // Lignes de vente
  refundLines RefundLine[]  // Lignes de remboursement
  createdAt   DateTime      @default(now())
  updatedAt   DateTime      @updatedAt
}

// Inventaire par magasin (table de jonction)
model Stock {
  id        Int     @id @default(autoincrement())
  store     Store   @relation(fields: [storeId], references: [id])
  storeId   Int
  product   Product @relation(fields: [productId], references: [id])
  productId Int
  quantity  Int     @default(0) // Quantité disponible

  @@unique([storeId, productId]) // Un produit par magasin
  @@index([storeId])
  @@index([productId])
}

// Transaction de vente principale
model Sale {
  id         Int        @id @default(autoincrement())
  date       DateTime   @default(now())
  total      Float      // Total calculé
  status     String     @default("active") // "active" | "completed" | "refunded"
| "partially_refunded"
  user       User       @relation(fields: [userId], references: [id])
  userId     Int
  store      Store      @relation(fields: [storeId], references: [id])
  storeId    Int
  saleLines  SaleLine[] // Lignes de détail
  refunds    Refund[]   // Remboursements associés
  createdAt  DateTime   @default(now())
  updatedAt  DateTime   @updatedAt

  @@index([userId])
  @@index([storeId])
  @@index([date])
}

// Ligne de détail de vente
model SaleLine {
  id        Int     @id @default(autoincrement())
  sale      Sale    @relation(fields: [saleId], references: [id], onDelete:
Cascade)
  saleId    Int
  product   Product @relation(fields: [productId], references: [id])
  productId Int
  quantity  Int     // Quantité vendue
  unitPrice Float   // Prix unitaire au moment de la vente
  lineTotal Float   // Total de la ligne (quantity * unitPrice)
```

```
  @@index([saleId])
  @@index([productId])
}

// Transaction de remboursement
model Refund {
  id          Int          @id @default(autoincrement())
  date        DateTime     @default(now())
  total       Float        // Total remboursé
  reason      String       // Raison du remboursement
  user        User         @relation(fields: [userId], references: [id])
  userId      Int
  store       Store        @relation(fields: [storeId], references: [id])
  storeId     Int
  sale        Sale         @relation(fields: [saleId], references: [id])
  saleId      Int          // Vente originale
  refundLines RefundLine[] // Lignes de détail du remboursement
  createdAt   DateTime     @default(now())
  updatedAt   DateTime     @updatedAt

  @@index([userId])
  @@index([storeId])
  @@index([saleId])
  @@index([date])
}

// Ligne de détail de remboursement
model RefundLine {
  id        Int     @id @default(autoincrement())
  refund    Refund  @relation(fields: [refundId], references: [id], onDelete:
Cascade)
  refundId  Int
  product   Product @relation(fields: [productId], references: [id])
  productId Int
  quantity  Int     // Quantité remboursée
  unitPrice Float   // Prix unitaire remboursé
  lineTotal Float   // Total de la ligne

  @@index([refundId])
  @@index([productId])
}

// Saga pour orchestration des transactions distribuées
model Saga {
  id              Int             @id @default(autoincrement())
  correlationId   String          @unique
  state           String          // État actuel de la saga
  currentStep     String?         // Étape en cours d'exécution
  context         Json            // Contexte de la saga (données métier)
  createdAt       DateTime        @default(now())
  updatedAt       DateTime        @updatedAt
  completedAt     DateTime?       // Date de fin (succès ou échec)
  errorMessage    String?         // Message d'erreur si échec
```

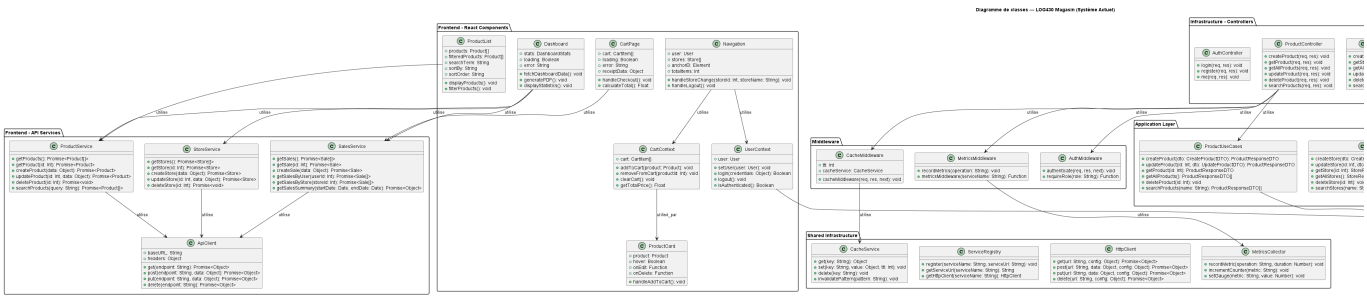
```
  compensationData Json?          // Données pour compensation
  stepLogs        SagaStepLog[]   // Logs détaillés des étapes

  @@index([correlationId])
  @@index([state])
  @@index([createdAt])
}

// Logs détaillés des étapes de saga pour monitoring
model SagaStepLog {
  id            Int       @id @default(autoincrement())
  saga          Saga      @relation(fields: [sagaId], references: [id])
  sagaId        Int
  stepName      String    // Nom de l'étape (StockVerification, Payment, etc.)
  state         String    // État de l'étape (STARTED, COMPLETED, FAILED)
  startedAt     DateTime  @default(now())
  completedAt   DateTime? // Date de fin d'exécution
  duration      Int?      // Durée d'exécution en millisecondes
  success       Boolean?  // Succès ou échec de l'étape
  errorMessage  String?   // Message d'erreur détaillé
  stepData      Json?     // Données spécifiques à l'étape

  @@index([sagaId])
  @@index([stepName])
  @@index([startedAt])
}
```

5.3 Architecture Interne des Services



Chaque microservice suit le pattern **Clean Architecture** avec infrastructure partagée, comme illustré dans le diagramme de classes ci-dessus :

```
services/[service-name]/
├── domain/                    # Logique métier pure
│   ├── entities/              # Entités du domaine
│   ├── repositories/          # Interfaces de persistance
│   └── aggregates/            # Agrégats métier
├── application/               # Cas d'usage applicatifs
│   ├── use-cases/            # Implémentation des cas d'usage
│   └── dtos/                 # Objets de transfert de données
├── infrastructure/           # Préoccupations externes
│   ├── database/             # Repositories avec shared infrastructure
│   │   ├── shared-*.repository.ts  # Implémentations partagées
│   │   └── *.repository.ts         # Adaptations spécifiques
│   ├── http/                 # Contrôleurs REST
│   │   └── *.controller.ts
│   └── services/             # Services externes (HTTP calls)
└── server.ts                 # Point d'entrée et configuration
```

5.3.1 Infrastructure Partagée

```
src/shared/infrastructure/
├── database/                 # Gestion centralisée de la base
│   ├── database-manager.ts   # Gestionnaire Prisma centralisé
│   ├── base-repository.ts    # Repository de base générique
│   └── cross-domain-queries.ts # Validation inter-domaines
├── caching/                  # Cache Redis partagé
│   ├── cache-service.ts      # Service de cache
│   ├── redis-client.ts       # Client Redis configuré
│   └── middleware.ts         # Middleware de cache HTTP
├── logging/                  # Logging centralisé
│   └── logger.ts             # Configuration Winston
├── metrics/                  # Métriques Prometheus
│   └── metrics.ts            # Collection de métriques
└── http/                     # Client HTTP partagé
    └── http-client.ts        # Client pour communication inter-services
```

---

6. Vues Runtime

6.1 Scénario : Authentification Utilisateur

Le diagramme de séquence ci-dessus illustre le processus d'authentification utilisateur :

1. Le client web envoie une requête de connexion via Kong Gateway
2. Kong route la requête vers le User Service
3. Le User Service vérifie les credentials dans PostgreSQL
4. Génération et cache du JWT dans Redis
5. Retour de la réponse d'authentification avec le token

6.2 Scénario : Achat de Produit (Workflow Saga Orchestrée)

Le processus de vente utilise désormais le pattern Saga Orchestration pour coordonner les opérations distribuées :

Workflow de Vente Orchestrée :

1. **Initiation** - Client web envoie requête de vente via Kong Gateway
2. **Routage** - Kong route vers Saga Orchestrator Service (port 3004)
3. **Création Saga** - Orchestrateur crée une saga avec correlationId unique
4. **Étape 1: Stock Verification** - Vérification disponibilité via Catalog Service
5. **Étape 2: Stock Reservation** - Réservation temporaire (timeout 30s)

6. **Étape 3: Payment Processing** - Traitement paiement via Transaction Service
7. **Étape 4: Order Confirmation** - Création vente finale et confirmation stock
8. **Completion** - Saga terminée avec succès ou compensation automatique si échec

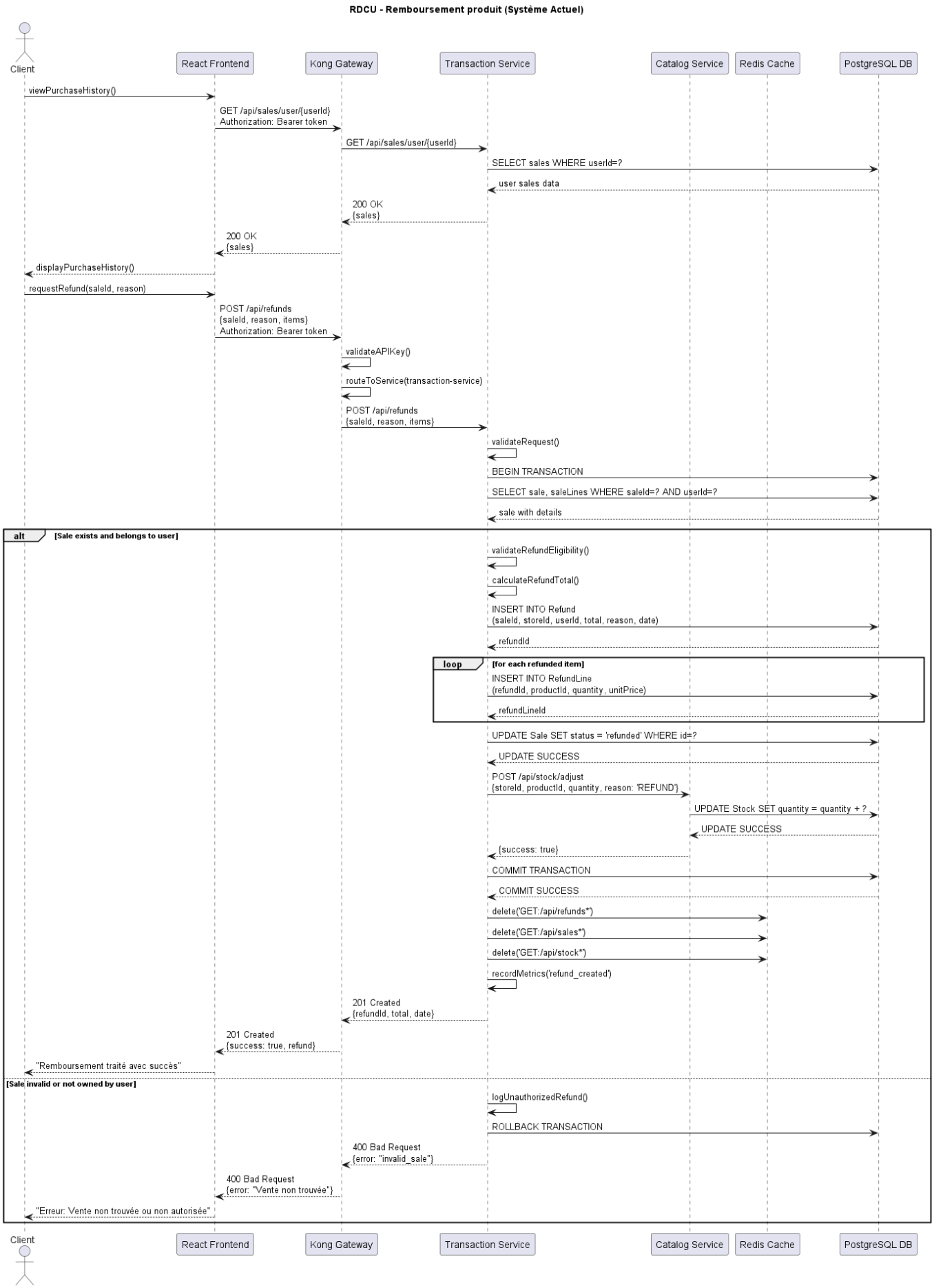
Gestion des Échecs :

- **Échec Stock** → Réponse immédiate au client
- **Échec Réservation** → Libération automatique des quantités
- **Échec Paiement** → Compensation stock + notification client
- **Échec Confirmation** → Compensation paiement + libération stock

Observabilité :

- Chaque étape loggée dans SagaStepLog avec timestamps et métriques
- Corrélation ID pour traçabilité end-to-end
- Métriques Prometheus pour monitoring temps réel

6.3 Scénario : Génération de Rapport



Le processus de génération de rapport pour les administrateurs :

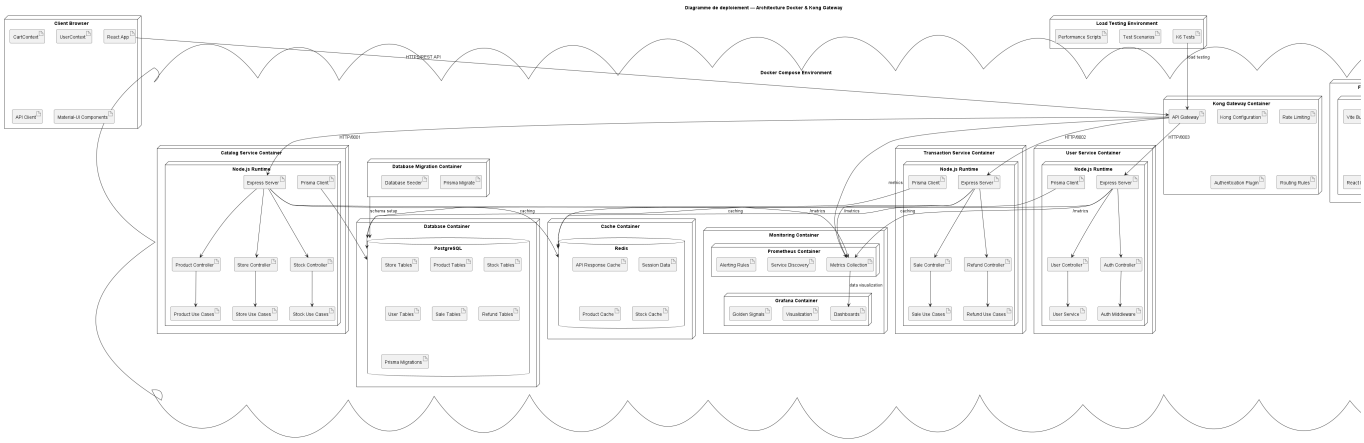
1. Requête dashboard via Kong Gateway

2. Routage vers Catalog Service
3. Agrégation des données de vente et transaction
4. Compilation et retour du rapport complet

---

7. Vues de Déploiement

7.1 Architecture de Déploiement



Le diagramme de déploiement ci-dessus montre l'organisation des conteneurs Docker :

Niveau Frontend :

- Client Web (React) en développement local sur localhost:5173
- Kong Gateway exposé sur localhost:8000 comme point d'entrée unique

Niveau Microservices :

- User Service sur port 3001 (authentification et gestion utilisateurs)
- Catalog Service sur port 3002 (produits et inventaire)
- Transaction Service sur port 3003 (ventes et remboursements)
- Saga Orchestrator Service sur port 3004 (orchestration workflows distribuées)

Niveau Infrastructure :

- PostgreSQL 15 sur port 5432 (base de données centralisée)
- Redis 7 sur port 6379 (cache et sessions)

Niveau Monitoring :

- Prometheus sur port 9090 (collecte de métriques)
- Grafana sur port 3004 (dashboards et visualisation)

7.2 Configuration Docker Compose

Le déploiement utilise Docker Compose avec les composants suivants :

7.2.1 Services Métier

```
services:
  user-service:
    build: ./services/user-service
    ports: ["3001:3000"]
    environment:
      - DATABASE_URL=postgresql://user:password@postgres:5432/retail
      - REDIS_URL=redis://redis:6379
    depends_on: [postgres, redis]

  catalog-service:
    build: ./services/catalog-service
    ports: ["3002:3000"]
    environment:
      - DATABASE_URL=postgresql://user:password@postgres:5432/retail
      - REDIS_URL=redis://redis:6379
    depends_on: [postgres, redis]

  transaction-service:
    build: ./services/transaction-service
    ports: ["3003:3000"]
    environment:
      - DATABASE_URL=postgresql://user:password@postgres:5432/retail
      - REDIS_URL=redis://redis:6379
    depends_on: [postgres, redis]
```

7.2.2 Infrastructure

```
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: retail
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7
    ports: ["6379:6379"]
    volumes:
      - redis_data:/data

  kong:
    image: kong:3.4
    environment:
      KONG_DATABASE: "off"
      KONG_DECLARATIVE_CONFIG: /kong/declarative/kong.yml
    volumes:
      - ./api-gateway/kong/kong.yml:/kong/declarative/kong.yml
    ports:
```

```
      - "8000:8000"
      - "8001:8001"
```

7.2.3 Monitoring

```
  prometheus:
    image: prom/prometheus:latest
    ports: ["9090:9090"]
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'

  grafana:
    image: grafana/grafana:latest
    ports: ["3004:3000"]
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - ./monitoring/grafana:/etc/grafana/provisioning
```

7.3 Stratégies de Déploiement

7.3.1 Déploiement de Développement

- **Script** : `quick-start.bat`
- **Approche** : Déploiement complet avec une commande
- **Avantages** : Simplicité, reproductibilité

7.3.2 Déploiement de Production (Recommandé)

- **Orchestration** : Kubernetes avec Helm Charts
- **Monitoring** : Prometheus Operator
- **Sécurité** : Secrets management, RBAC
- **Évolutivité** : Horizontal Pod Autoscaler

---

8. Concepts Transversaux

8.1 Sécurité

8.1.1 Authentification et Autorisation

- **JWT Tokens** : Tokens stateless pour l'authentification
- **API Keys** : Clés d'API pour l'accès aux services via Kong
- **RBAC** : Contrôle d'accès basé sur les rôles (admin/client)

8.1.2 Sécurité des Communications

- **HTTPS** : Chiffrement des communications (production)
- **CORS** : Configuration appropriée pour les requêtes cross-origin
- **Rate Limiting** : Protection contre les attaques DDoS

8.2 Performance et Mise en Cache

8.2.1 Stratégie de Cache Redis

```
// Cache des produits
const cacheKey = `products:${storeId}`;
const cachedProducts = await redis.get(cacheKey);
if (cachedProducts) {
  return JSON.parse(cachedProducts);
}

// Invalidation lors des mises à jour
await redis.del(`products:${storeId}`);
await redis.del(`stock:${storeId}`);
```

8.2.2 Optimisations Base de Données

- **Index** : Index sur les clés étrangères et colonnes de recherche
- **Requêtes Optimisées** : Utilisation d'agrégations SQL
- **Connection Pooling** : Pool de connexions Prisma

8.3 Observabilité

8.3.1 Four Golden Signals

1. **Latence** : Temps de réponse des requêtes
2. **Trafic** : Nombre de requêtes par seconde
3. **Erreurs** : Taux d'erreur par endpoint
4. **Saturation** : Utilisation des ressources

8.3.2 Métriques Collectées

```
// Métriques Prometheus
const httpRequestDuration = new prometheus.Histogram({
  name: 'http_request_duration_seconds',
  help: 'Duration of HTTP requests in seconds',
  labelNames: ['method', 'route', 'status_code']
});

const httpRequestTotal = new prometheus.Counter({
  name: 'http_requests_total',
  help: 'Total number of HTTP requests',
```

```
  labelNames: ['method', 'route', 'status_code']
});
```

8.4 Gestion des Erreurs

8.4.1 Stratégies de Résilience

- **Circuit Breaker** : Protection contre les cascades de pannes
- **Retry Logic** : Tentatives automatiques en cas d'échec
- **Timeout** : Délais d'attente configurables

8.4.2 Logging Structuré

```
// Logging avec structure JSON
logger.info('User login attempt', {
  userId: user.id,
  timestamp: new Date().toISOString(),
  ip: req.ip,
  userAgent: req.headers['user-agent']
});
```

---

9. Décisions d'Architecture

9.1 ADR-001 : Architecture Microservices

Statut : ACCEPTÉ

Décision : Utiliser une architecture microservices avec trois services principaux (User, Catalog, Transaction).

Justification :

- Évolutivité indépendante des services
- Isolation des pannes
- Flexibilité technologique
- Équipes autonomes (simulation)

Conséquences :

- Complexité de déploiement accrue
- Latence réseau entre services
- Nécessité de monitoring distribué

9.2 ADR-002 : Kong API Gateway

Statut : ACCEPTÉ

Décision : Utiliser Kong Gateway pour le routage et les préoccupations transversales.

Justification :

- Centralisation de la sécurité
- Surveillance unifiée
- Gestion du trafic (rate limiting)
- Documentation API

Conséquences :

- Point de défaillance unique
- Courbe d'apprentissage
- Configuration additionnelle

9.3 ADR-003 : Base de Données Partagée

Statut : ACCEPTÉ

Décision : Utiliser une base de données PostgreSQL partagée entre les services.

Justification :

- Simplicité de déploiement
- Cohérence des données ACID
- Requêtes cross-domaines efficaces
- Contraintes de ressources académiques

Conséquences :

- Couplage entre services
- Pas de technologie de base de données spécialisée
- Évolutivité limitée

9.4 ADR-004 : Monitoring Prometheus/Grafana

Statut : ACCEPTÉ

Décision : Utiliser Prometheus pour la collecte de métriques et Grafana pour la visualisation.

Justification :

- Standard de l'industrie
- Four Golden Signals
- Alerting intégré
- Écosystème riche

Conséquences :

- Courbe d'apprentissage
- Ressources additionnelles
- Configuration complexe

9.5 ADR-005 : Stratégie de Cache Redis

Statut : ACCEPTÉ

Décision : Utiliser Redis pour la mise en cache des réponses API.

Justification :

- Amélioration des performances
- Réduction de la charge DB
- Support des sessions
- Simplicité d'intégration

Conséquences :

- Complexité de gestion du cache
- Cohérence des données
- Ressource additionnelle

9.6 ADR-006 : Infrastructure de Base de Données Centralisée

Statut : ACCEPTÉ

Décision : Refactoriser l'architecture pour utiliser une infrastructure de base de données centralisée avec des frontières de domaine claires.

Justification :

- Standardisation des patterns d'accès aux données
- Optimisation des connexions et performances
- Respect des frontières de domaine via des interfaces repository
- Maintien de la compatibilité API existante
- Amélioration de l'observabilité et du monitoring

Conséquences :

- Dépendance vers l'infrastructure partagée
- Complexité de migration temporaire
- Amélioration de la maintenabilité à long terme
- Patterns d'accès aux données plus cohérents

9.7 ADR-007 : Saga Orchestration Pattern

Statut : ACCEPTÉ

Décision : Implémentation du pattern Saga Orchestration pour la gestion des transactions distribuées avec un service dédié `saga-orchestrator-service`.

Contexte :

Le processus de vente nécessite des opérations coordonnées à travers plusieurs services (vérification stock, réservation, paiement, confirmation) avec besoin de cohérence transactionnelle et gestion des échecs partiels.

Justification :

- **Cohérence transactionnelle** : Garantie de cohérence finale avec compensation automatique
- **Observabilité** : Traçabilité complète des workflows avec SagaStepLog
- **Résilience** : Gestion gracieuse des pannes avec retry et compensation
- **Performance** : Workflow asynchrone non-bloquant avec cache Redis
- **Maintenabilité** : Pattern extensible pour nouveaux workflows

Alternatives rejetées :

- **Choreography Pattern** : Manque d'observabilité centralisée
- **Two-Phase Commit (2PC)** : Blocage possible et point de défaillance unique
- **Transactions Distribuées XA** : Complexité excessive pour le contexte

Conséquences :

- Nouveau service à maintenir (saga-orchestrator-service:3004)
- Complexité accrue de la machine à états
- Latence supplémentaire (~50-100ms) acceptable pour la cohérence
- Eventual consistency nécessitant adaptation UI
- Gains majeurs en fiabilité et observabilité

---

10. Exigences de Qualité

10.1 Scénarios de Qualité

10.1.1 Performance

Scénario	Métrique	Objectif	Mesure
Temps de réponse API	Latence p95	< 200ms	Tests k6
Débit transactions	TPS	> 100 req/s	Tests de charge
Temps de chargement UI	First Contentful Paint	< 1s	Lighthouse

10.1.2 Disponibilité

Scénario	Métrique	Objectif	Mesure
Uptime système	Disponibilité	> 99%	Prometheus
Récupération panne	MTTR	< 5min	Monitoring
Détection problème	MTTD	< 1min	Alerting

10.1.3 Sécurité

Scénario	Métrique	Objectif	Mesure
Authentification	Taux d'échec	< 0.1%	Logs
Autorisation	Accès non autorisé	0	Audit

Scénario	Métrique	Objectif	Mesure
Rate limiting	Requêtes bloquées	> 95%	Kong métriques

10.2 Architecture de Test

10.2.1 Tests de Charge k6

```
// Scénario de montée en charge
export let options = {
  stages: [
    { duration: '5m', target: 50 },   // Montée graduelle
    { duration: '10m', target: 100 }, // Charge normale
    { duration: '5m', target: 200 },  // Pic de charge
    { duration: '10m', target: 100 }, // Retour normal
    { duration: '5m', target: 0 },    // Arrêt graduel
  ],
  thresholds: {
    'http_req_duration': ['p95<200'],
    'http_req_failed': ['rate<0.1'],
  },
};
```

10.2.2 Tests End-to-End

```
// Parcours utilisateur complet
export default function() {
  // 1. Login
  let loginResponse = http.post(`${BASE_URL}/api/auth/login`, {
    name: 'client',
    password: 'client123'
  });

  // 2. Consultation catalogue
  let products = http.get(`${BASE_URL}/api/products`);

  // 3. Achat
  let sale = http.post(`${BASE_URL}/api/sales`, {
    storeId: 1,
    items: [{ productId: 1, quantity: 2 }]
  });

  check(sale, {
    'sale successful': (r) => r.status === 201,
    'response time OK': (r) => r.timings.duration < 500,
  });
}
```

11. Risques et Dettes Techniques

11.1 Risques Identifiés

11.1.1 Risques Techniques

Risque	Probabilité	Impact	Mitigation
Panne Kong Gateway	Moyenne	Élevé	Health checks, load balancer
Surcharge PostgreSQL	Élevée	Moyen	Connection pooling, cache
Épuisement Redis	Faible	Moyen	Eviction policies, monitoring
Latence réseau	Élevée	Faible	Optimisation requêtes, cache

11.1.2 Risques Opérationnels

Risque	Probabilité	Impact	Mitigation
Complexité déploiement	Élevée	Moyen	Scripts automatisés, documentation
Debugging distribué	Élevée	Élevé	Tracing distribué, logging centralisé
Gestion des secrets	Moyenne	Élevé	Gestionnaire de secrets, rotation

11.2 Dettes Techniques

11.2.1 Dettes Architecturales

- **Base de données partagée** : Migration vers DB par service
- **Communication synchrone** : Introduction de messaging asynchrone
- **Absence de service discovery** : Implémentation Consul/Eureka
- **Pas de circuit breaker** : Ajout de Hystrix/Resilience4j

11.2.2 Dettes de Code

- **Tests unitaires manquants** : Couverture < 50%
- **Validation d'entrée** : Schemas de validation incomplets
- **Documentation API** : Spécifications OpenAPI manquantes
- **Logging standardisé** : Format et niveaux incohérents

11.2.3 Dettes d'Infrastructure

- **Secrets en dur** : Variables d'environnement en clair
- **Absence de backup** : Stratégie de sauvegarde manquante
- **SSL/TLS** : HTTPS non configuré
- **Monitoring alerting** : Règles d'alerte basiques

11.3 Plan de Remédiation

11.3.1 Priorité Élevée (0-3 mois)

1. **Implémentation des tests unitaires** (Coverage > 80%)
2. **Configuration HTTPS** pour la production
3. **Alerting avancé** avec Prometheus AlertManager

---

12. Glossaire

12.1 Termes Métier

Terme	Définition
Catalogue	Ensemble des produits disponibles à la vente
Inventaire	Stock de produits disponibles par magasin
Transaction	Opération de vente ou de remboursement
Ligne de vente	Article individuel dans une transaction
Tableau de bord	Interface d'analyse pour les administrateurs

12.2 Termes Techniques

Terme	Définition
API Gateway	Point d'entrée unique pour toutes les requêtes client
Circuit Breaker	Pattern de résilience pour prévenir les cascades de pannes
DDD	Domain-Driven Design - Approche de conception centrée sur le domaine
Four Golden Signals	Latence, Trafic, Erreurs, Saturation - Métriques clés
JWT	JSON Web Token - Standard pour l'authentification
RBAC	Role-Based Access Control - Contrôle d'accès basé sur les rôles
SLI	Service Level Indicator - Indicateur de niveau de service
SLO	Service Level Objective - Objectif de niveau de service

12.3 Acronymes

Acronyme	Signification
ACID	Atomicity, Consistency, Isolation, Durability
CORS	Cross-Origin Resource Sharing
CQRS	Command Query Responsibility Segregation
DTO	Data Transfer Object
HTTP	HyperText Transfer Protocol

Acronyme	Signification
JSON	JavaScript Object Notation
MTTR	Mean Time To Recovery
MTTD	Mean Time To Detection
ORM	Object-Relational Mapping
REST	Representational State Transfer
SQL	Structured Query Language
TPS	Transactions Per Second
UI	User Interface
UX	User Experience

Annexes

Diagrammes Disponibles

Tous les diagrammes de ce rapport sont disponibles sous forme d'images PNG dans le dossier `docs/diagrams/png/` :

- **Architecture Système.png** : Vue d'ensemble de l'architecture complète du système
- **Diagramme CU.png** : Diagramme de cas d'usage montrant les fonctionnalités par acteur
- **Diagramme de classes.png** : Structure des classes et relations dans l'architecture Clean
- **Diagramme de deploiement.png** : Organisation des conteneurs Docker et déploiement
- **MDD Magasin.png** : Modèle de domaine métier avec entités et relations
- **RDCU Remboursement.png** : Séquence de traitement des remboursements
- **RDCU Vente.png** : Séquence de traitement des ventes

Les sources PlantUML correspondantes sont également disponibles dans `docs/diagrams/` pour modification.

Conclusion

Ce rapport Arc42 présente une architecture microservices complète pour un système de gestion de magasin de détail. L'architecture privilégie l'observabilité, la performance et la maintenabilité tout en respectant les contraintes du projet académique.

Les décisions architecturales prises (Kong Gateway, base de données partagée, monitoring Prometheus/Grafana) offrent un bon équilibre entre complexité et fonctionnalité pour un projet d'apprentissage.

Les risques identifiés et les dettes techniques fournissent une roadmap claire pour l'évolution future du système vers une architecture de production robuste.

Auteur : Minh Khoi Le **Date** : 2025-08-06 **Version** : 4.0