

# Rapport Arc42 Lab7 - 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>
- Lab7 : <https://github.com/Minh-Khoi-Le/log430-lab7.git>

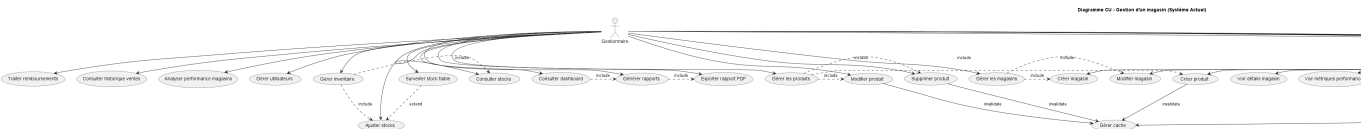
## Table des Matières

1. Introduction et Objectifs
2. Contraintes d'Architecture
3. Contexte et Portée du Système
4. Stratégie de Solution
5. Vue d'Ensemble de l'Architecture
6. Vues Runtime
7. Vues de Déploiement
8. Concepts Transversaux
9. Décisions d'Architecture
10. Exigences de Qualité
11. Risques et Dettes Techniques
12. Glossaire

## 1. Introduction et Objectifs

### 1.1 Aperçu des Exigences

Le système de gestion de magasin de détail développé pour le LOG430 Lab 7 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

Rôle	Attentes	Responsabilités
Administrateur Système	Déploiement facile, surveillance	Gestion de l'infrastructure
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"

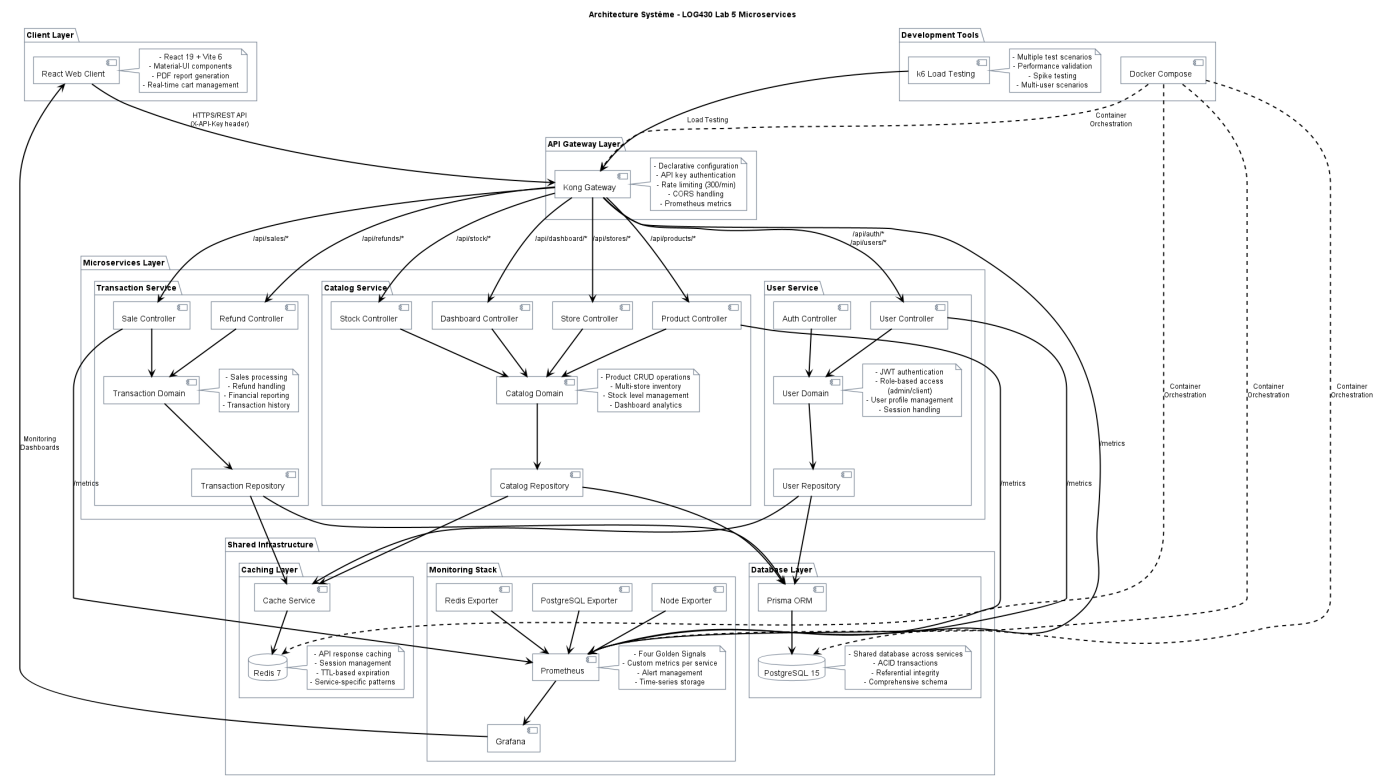
## 3. Contexte et Portée du Système

### 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

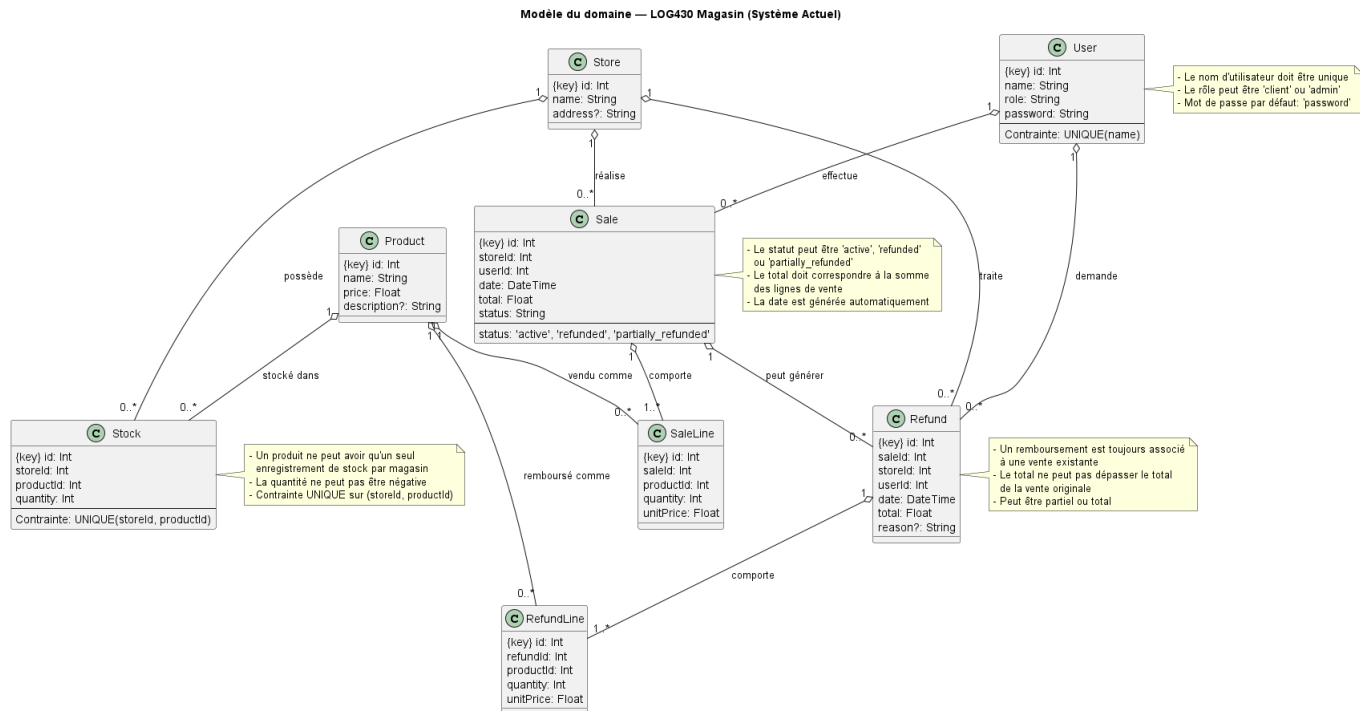
## 5.2 Architecture de Base de Données Centralisée

### 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
}

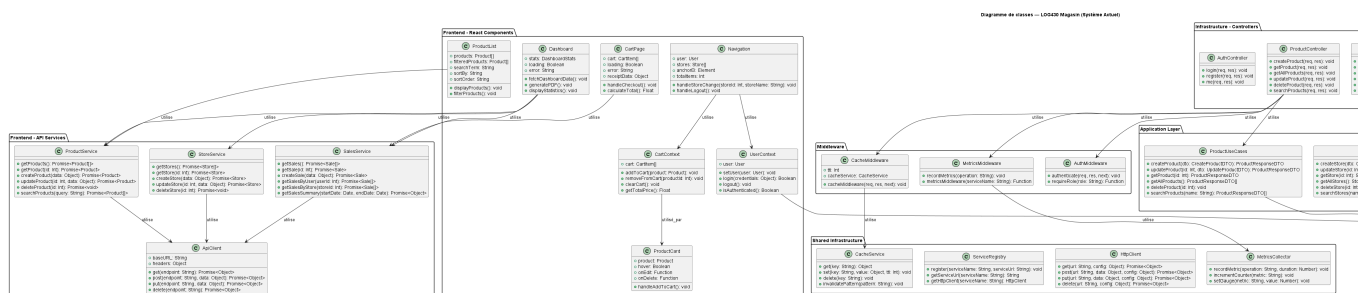
```

```
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
}
```

### 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
```

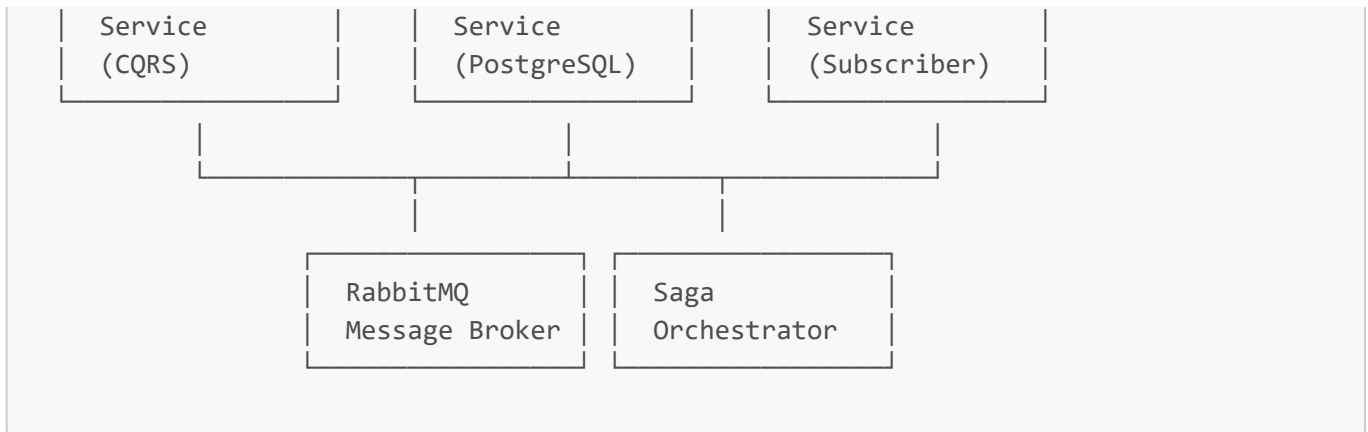
5.5 Architecture Événementielle et CQRS

5.5.1 Vue d'Ensemble de l'Architecture Événementielle

Le système implémente une **architecture événementielle complète** basée sur les patterns Event Sourcing, CQRS (Command Query Responsibility Segregation), et Saga Orchestration. Cette architecture assure le découplage des services, la traçabilité complète, et la cohérence finale des données distribuées.

Composants Principaux :





### 5.5.2 Event Store Service (Port 3008)

#### Responsabilités :

- Persistance immuable de tous les événements business dans PostgreSQL
- Reconstruction d'état des agrégats via replay d'événements
- Contrôle de concurrence optimiste avec versioning
- API de requête flexible pour l'analyse et l'audit
- Support pour les projections et les vues matérialisées

#### Schéma Event Store PostgreSQL :

```

CREATE TABLE event_store (
  event_id UUID PRIMARY KEY,
  event_type VARCHAR(255) NOT NULL,
  aggregate_id VARCHAR(255) NOT NULL,
  aggregate_type VARCHAR(255) NOT NULL,
  event_data JSONB NOT NULL,
  metadata JSONB NOT NULL,
  version INTEGER NOT NULL,
  created_at TIMESTAMP DEFAULT NOW()
);

-- Indexes pour performance
CREATE INDEX idx_event_store_aggregate ON event_store(aggregate_id, version);
CREATE INDEX idx_event_store_type ON event_store(event_type);
CREATE INDEX idx_event_store_created ON event_store(created_at);

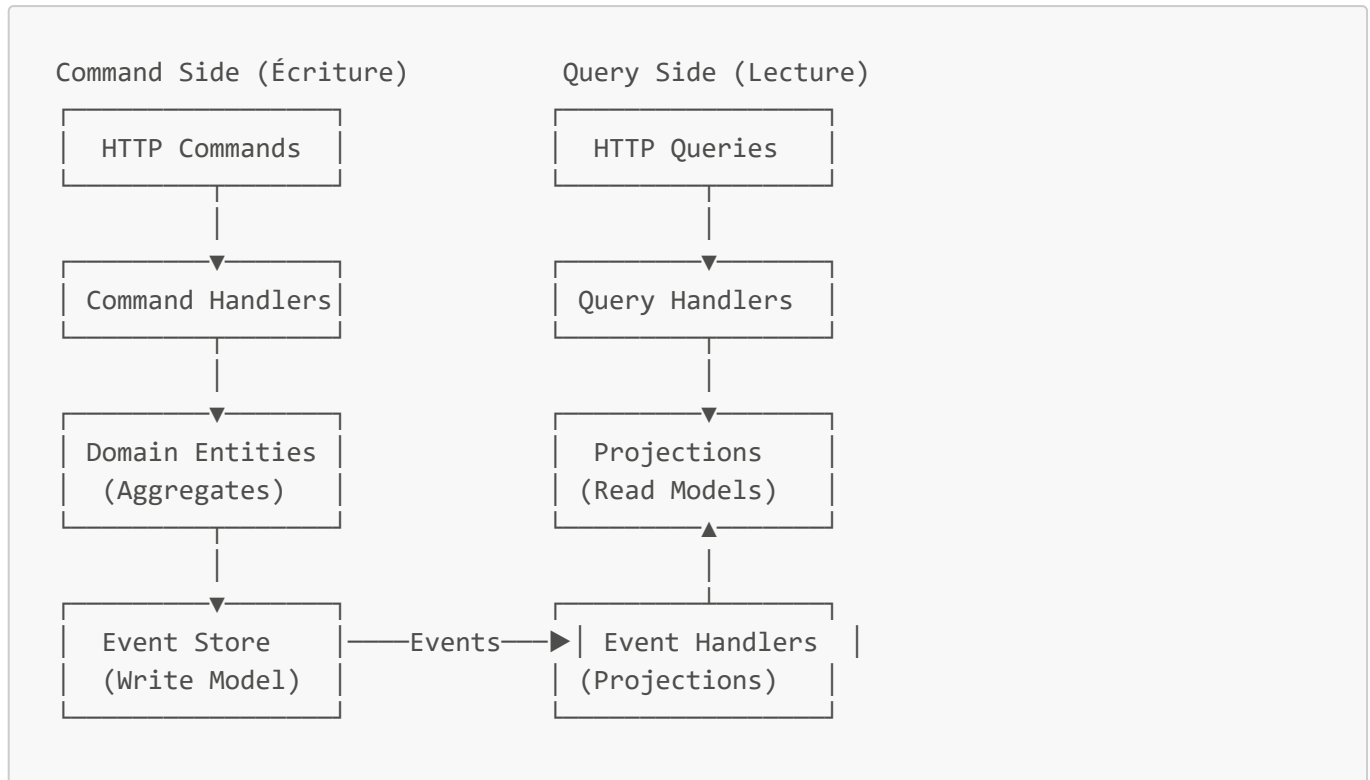
```

#### API Endpoints :

- **POST** /api/events/append/:streamId - Ajouter événements à un stream
- **GET** /api/events/stream/:streamId - Récupérer événements d'un stream
- **GET** /api/events/query - Requête d'événements avec filtres
- **POST** /api/events/replay - Replay d'événements selon critères
- **GET** /api/events/statistics - Statistiques de l'event store

### 5.5.3 Complaint Service - Implémentation CQRS Complète

## Architecture CQRS :



### Command Handlers :

- `CreateComplaintHandler` - Création de nouvelle réclamation
- `AssignComplaintHandler` - Assignment à un agent
- `StartProcessingHandler` - Démarrage du traitement
- `ResolveComplaintHandler` - Résolution de la réclamation
- `CloseComplaintHandler` - Fermeture définitive

### Query Handlers :

- `GetComplaintsHandler` - Liste avec filtres et pagination
- `GetComplaintDetailsHandler` - Détail d'une réclamation
- `GetComplaintTimelineHandler` - Timeline des événements
- `GetComplaintStatisticsHandler` - Statistiques et métriques

### Projections (Read Models) :

```

// Projection optimisée pour l'affichage
interface ComplaintProjection {
    id: string;
    title: string;
    description: string;
    status: ComplaintStatus;
    priority: Priority;
    assignedTo?: string;
    assignedToName?: string;
    createdAt: Date;
    updatedAt: Date;
    resolvedAt?: Date;
  }
  
```



```

    customerInfo: {
      userId: string;
      userName: string;
      email: string;
    };
  }

  // Timeline pour l'audit et le suivi
  interface ComplaintTimeline {
    complaintId: string;
    events: ComplaintTimelineEvent[];
  }

```

#### 5.5.4 RabbitMQ Message Broker

##### Configuration des Exchanges :

- `complaints.events` - Événements du domaine des réclamations
- `audit.events` - Événements d'audit système
- `saga.events` - Événements de coordination des workflows
- `notifications.events` - Événements de notification

##### Queues par Service :

- `complaint.commands` - Commandes pour le service réclamations
- `audit.all.events` - Tous les événements pour audit complet
- `saga.orchestration` - Coordination des workflows distribués
- `notification.events` - Notifications utilisateur

##### Configuration Docker :

```

rabbitmq:
  image: rabbitmq:3-management-alpine
  ports:
    - "5672:5672"    # AMQP port
    - "15672:15672"  # Management UI
  environment:
    RABBITMQ_DEFAULT_USER: admin
    RABBITMQ_DEFAULT_PASS: admin123

```

#### 5.5.5 Audit Service (Port 3007) - Souscripteur Universel

##### Responsabilités :

- Souscription automatique à TOUS les événements système
- Création d'audit logs pour conformité réglementaire
- Suivi des trails d'audit pour workflows complexes
- Corrélation distribuée avec correlation IDs
- Statistiques et rapports d'audit

## Souscriptions d'Événements :

```
// Souscription universelle à tous les événements
await eventBus.subscribeToAll('audit-service-all-events', {
  handle: async (event) => await auditHandlers.handleDomainEvent(event)
});

// Souscriptions spécialisées par domaine
await eventBus.subscribe('audit-service-complaint-events', 'COMPLAINT_*');
await eventBus.subscribe('audit-service-saga-events', 'SAGA_*');
await eventBus.subscribe('audit-service-security-events', 'USER_*');
```

## Modèle de Données Audit :

```
-- Logs d'audit pour tous les événements
CREATE TABLE audit_logs (
  audit_id UUID PRIMARY KEY,
  event_type VARCHAR(255) NOT NULL,
  entity_type VARCHAR(100),
  entity_id VARCHAR(255),
  event_data JSONB NOT NULL,
  correlation_id UUID,
  user_id VARCHAR(255),
  timestamp TIMESTAMP NOT NULL,
  source_service VARCHAR(100),
  metadata JSONB
);

-- Trails d'audit pour workflows
CREATE TABLE audit_trails (
  trail_id UUID PRIMARY KEY,
  correlation_id UUID NOT NULL,
  trail_type VARCHAR(100) NOT NULL,
  status VARCHAR(50) NOT NULL,
  started_at TIMESTAMP NOT NULL,
  completed_at TIMESTAMP,
  metadata JSONB
);
```

### 5.5.6 Flux d'Événements Typique

#### Scénario : Création et Traitement d'une Réclamation

1. HTTP POST /api/complaints  
↓
2. CreateComplaintCommand → ComplaintCommandHandler  
↓
3. Complaint.create() → ComplaintCreatedEvent

```

↓
4. Event Store ← PostgreSQL (persistence)
↓
5. RabbitMQ ← Event Publication
↓
6. Event Consumption:
  ├── Audit Service → Audit Log Creation
  ├── Projection Handlers → Read Model Update
  ├── Notification Service → User Alert
  └── Saga Orchestrator → Workflow Trigger (si applicable)

```

### 5.5.7 Monitoring et Métriques Événementielles

#### Métriques Prometheus Spécialisées :

```

// Publication d'événements
events_published_total{service, event_type, exchange, status}
event_publishing_duration_seconds{service, event_type}

// Consommation d'événements
events_consumed_total{service, event_type, queue, status}
event_consumption_duration_seconds{service, event_type}

// CQRS Métriques
command_executions_total{service, command_type, status}
query_executions_total{service, query_type, status}
projection_updates_total{service, projection_type, status}

// Event Store
event_store_operations_total{service, operation, status}
event_store_event_count{service, aggregate_type}
event_store_size_bytes{service}

// Sagas
saga_executions_total{service, saga_type, status}
saga_step_executions_total{service, saga_type, step_name}
saga_compensations_total{service, saga_type}

```

#### Dashboard Grafana "Event-Driven Architecture" :

- **Event Publishing Rate** : Taux de publication par service et type
- **Event Consumption Latency** : Latence end-to-end des événements
- **CQRS Performance** : Métriques commandes vs queries
- **Projection Lag** : Retard dans la mise à jour des projections
- **Saga Monitoring** : Workflows actifs et compensations
- **Error Rates** : Taux d'erreur par type d'opération

### 5.5.8 Avantages de l'Architecture Événementielle

**Techniques :**

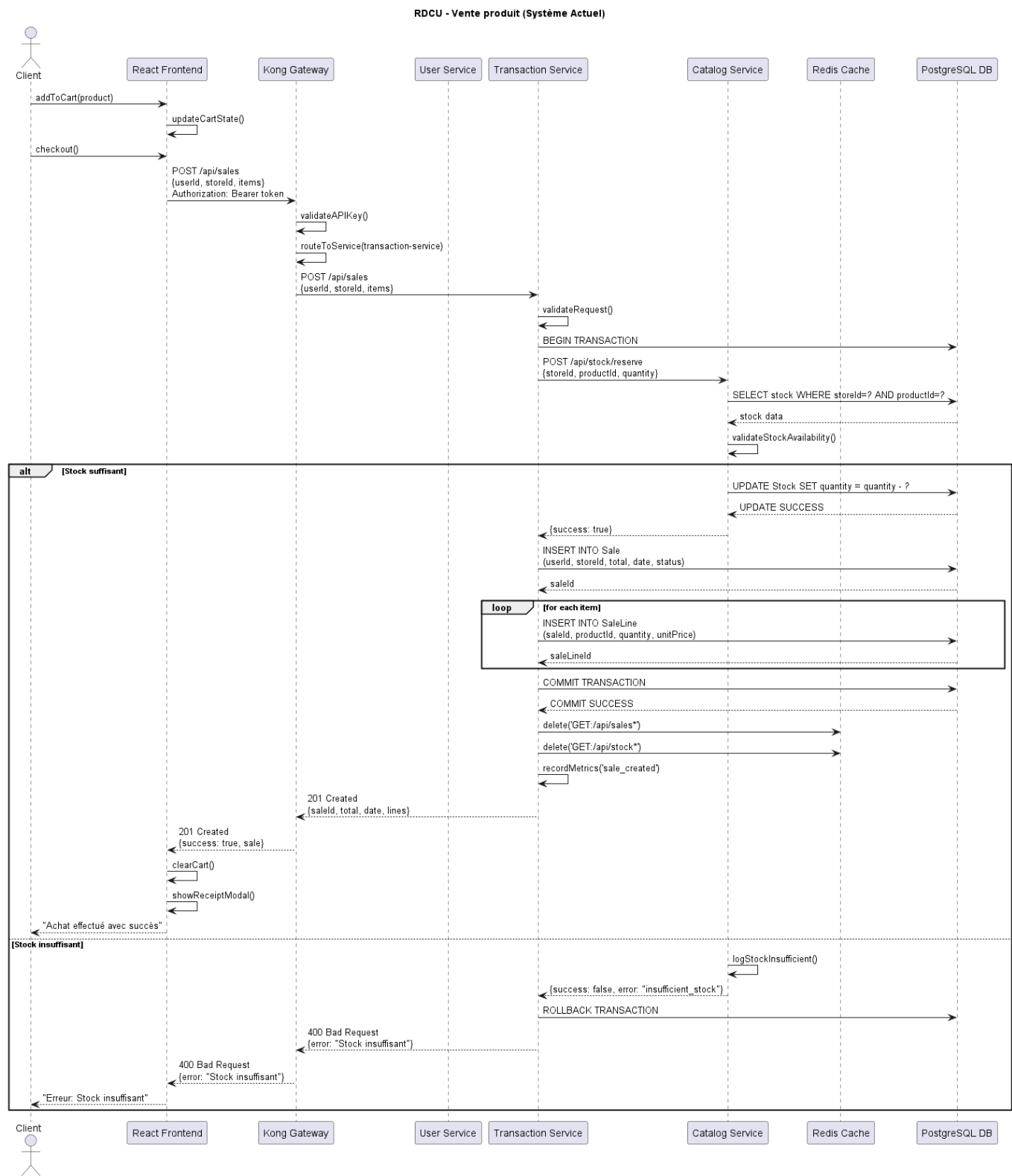
- **Découplage** : Services indépendants communiquant via événements
- **Scalabilité** : Séparation optimisée lecture/écriture avec CQRS
- **Résilience** : Retry logic, dead letter queues, compensations
- **Traçabilité** : Historique complet immuable dans l'Event Store
- **Consistency** : Eventual consistency via propagation d'événements

**Métier :**

- **Audit Complet** : Conformité réglementaire avec trails d'audit
  - **Analyse Historique** : Reconstruction d'état et analytics temporelles
  - **Workflows Complexes** : Orchestration via patterns Saga
  - **Temps Réel** : Réactions immédiates aux événements business
- 

## 6. Vues Runtime

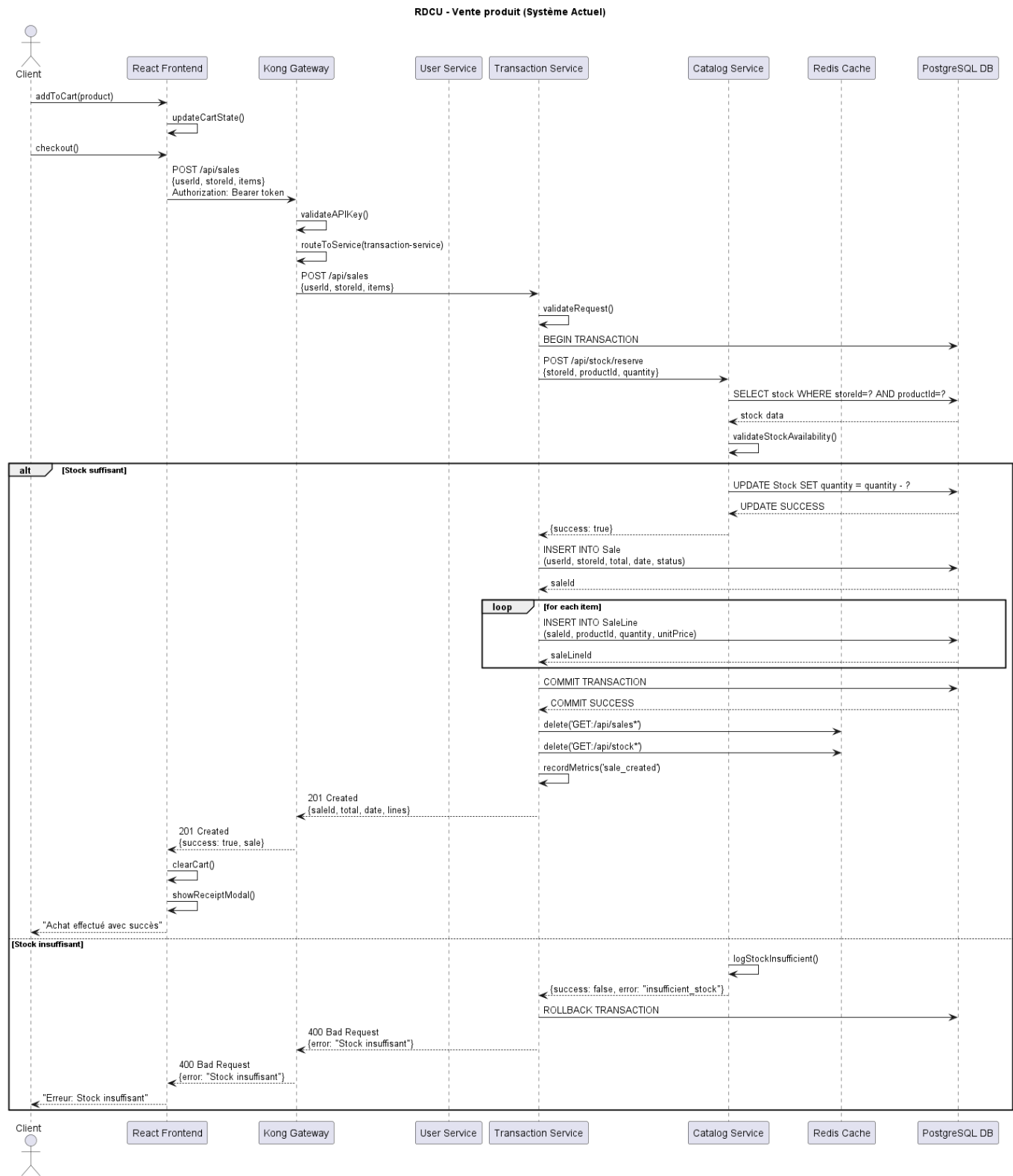
### 6.1 Scénario : Authentification Utilisateur



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. **Routing** - 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

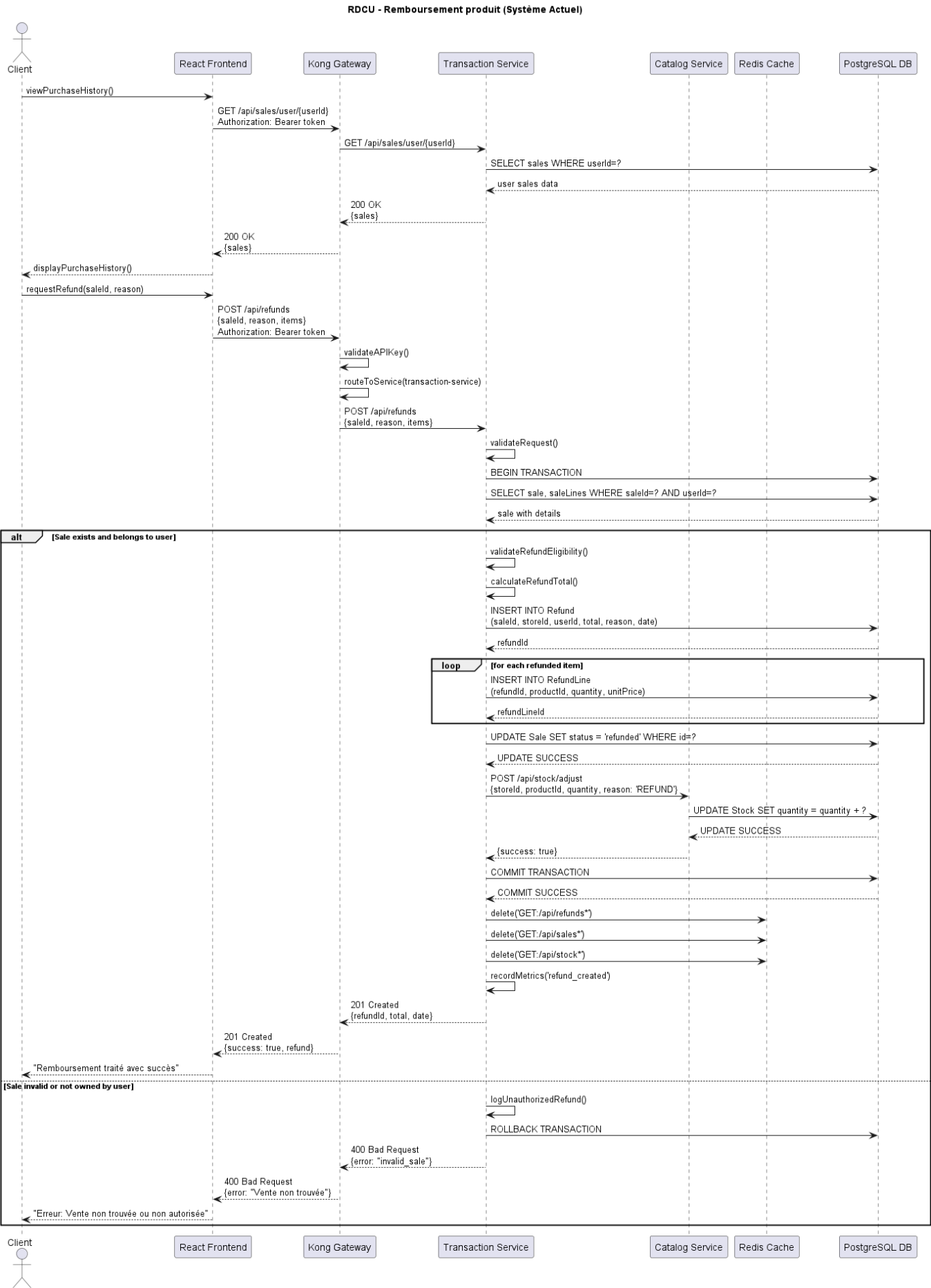
#### Gestion des Échecs :

- **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

## 6.4 Scénario : Workflow Événementiel - Traitement d'une Réclamation CQRS

### Participants :

- Client (Web UI)
- Kong Gateway
- Complaint Service (CQRS)
- Event Store Service
- RabbitMQ (Message Broker)
- Audit Service
- Notification Service

### Flux d'Événements Détaillé :

1. [Client] HTTP POST /api/complaints  
↓
2. [Kong] Route → complaint-service:3005  
↓
3. [Complaint Service - Command Side]
  - ComplaintCommandHandler.createComplaint()
  - Validation métier
  - Complaint.create() → Événement COMPLAINT\_CREATED
 ↓
4. [Event Store Service:3008]
  - PostgreSQL: INSERT INTO event\_store
  - Versioning et concurrence optimiste
 ↓
5. [RabbitMQ] Publication événement
  - Exchange: complaints.events
  - Routing: complaint.event.created
 ↓
6. [Consommation Parallèle]
  - ├ [Audit Service:3007]
    - Création audit\_log automatique
    - Trail de conformité
  - ├ [Complaint Service - Query Side]
    - ComplaintProjectionHandler.onComplaintCreated()
    - Mise à jour projection dénormalisée
  - └ [Notification Service:3006]
    - Envoi notification client
    - Alerte assigné (si applicable)
 ↓
7. [Client] GET /api/complaints/:id  
↓
8. [Complaint Service - Query Side]
  - ComplaintQueryHandler.getComplaintDetails()

- Lecture projection optimisée
- Retour données enrichies

### Caractéristiques Techniques :

- **Eventual Consistency** : Les projections sont mises à jour asynchronement
- **Correlation IDs** : Traçabilité end-to-end dans tous les services
- **Retry Logic** : RabbitMQ avec exponential backoff
- **Monitoring** : Métriques temps réel pour chaque étape

### Métriques de Performance Typiques :

- **Command Execution** : 10-50ms (écriture + event store)
- **Event Publishing** : 1-5ms (RabbitMQ)
- **Event Consumption** : 5-20ms par consommateur
- **Projection Update** : 5-15ms (lecture/écriture projection)
- **Query Execution** : 2-10ms (lecture projection optimisée)

### Gestion d'Erreurs :

- **Échec Command** : Transaction rollback, pas d'événement publié
- **Échec Event Store** : Retry automatique, alerting si persistant
- **Échec Publication** : Retry avec dead letter queue après 3 tentatives
- **Échec Projection** : Retry indépendant, projection lag monitoring

## 6.5 Scénario : Event Replay et Reconstruction d'État

**Cas d'Usage** : Reconstruction de l'état d'une réclamation ou audit forensique

1. [Admin] HTTP GET /api/event-store/replay?aggregateId=complaint-123  
↓
2. [Event Store Service]
  - SELECT \* FROM event\_store WHERE aggregate\_id = 'complaint-123' ORDER BY version
  - Retour séquence chronologique des événements↓
3. [Reconstruction État]
  - ComplaintAggregate.fromEvents(events)
  - Application séquentielle des événements
  - État final reconstitué↓
4. [Analyse/Debug]
  - Comparaison avec projection actuelle
  - Détection d'éventuelles inconsistances
  - Audit trail complet disponible

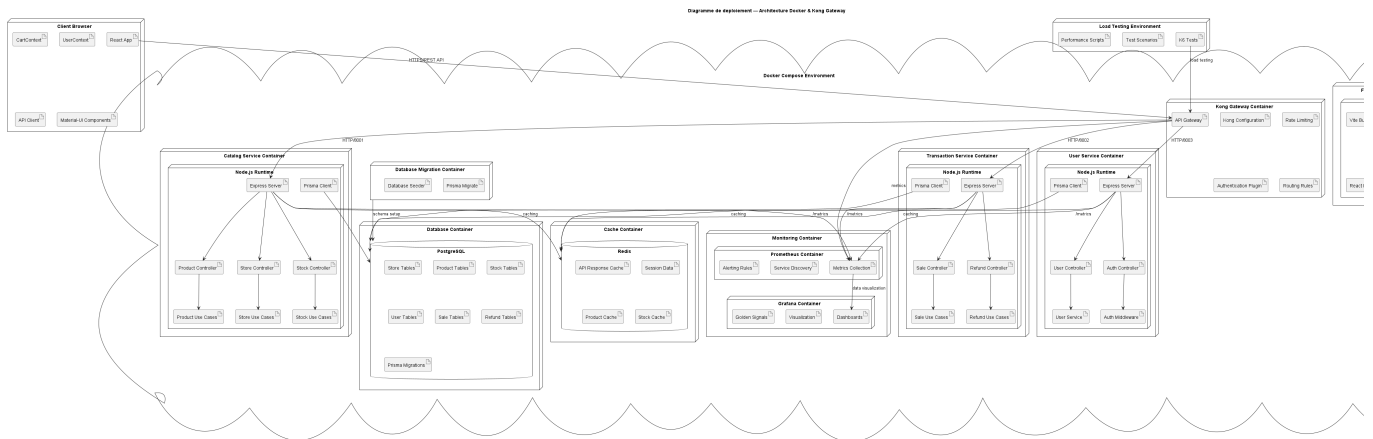
### Utilisation pour l'Audit :

- **Conformité RGPD** : Historique complet des modifications

- **Forensique** : Reconstruction exacte de l'état à tout moment
- **Debug** : Analyse des anomalies via replay
- **Migration** : Reconstruction après changement de schéma

## 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)
- Complaint Service sur port 3005 (gestion réclamations avec CQRS)
- Notification Service sur port 3006 (notifications temps réel)
- Audit Service sur port 3007 (audit universel événementiel)
- Event Store Service sur port 3008 (persistance événements avec PostgreSQL)

#### Niveau Infrastructure :

- PostgreSQL 15 sur port 5432 (base de données centralisée + event store)
- Redis 7 sur port 6379 (cache et sessions)
- RabbitMQ 3 sur ports 5672/15672 (message broker événementiel)

#### Niveau Monitoring :

- Prometheus sur port 9090 (collecte de métriques)
- Grafana sur port 3004 (dashboards et visualisation)
- Node Exporter sur port 9100 (métriques système)
- PostgreSQL Exporter sur port 9187 (métriques base de données)

- Redis Exporter sur port 9121 (métriques cache)

## 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://postgres:postgres@postgres:5432/log430_store
      - REDIS_URL=redis://redis:6379
    depends_on: [postgres, redis]

  catalog-service:
    build: ./services/catalog-service
    ports: ["3002:3000"]
    environment:
      - DATABASE_URL=postgresql://postgres:postgres@postgres:5432/log430_store
      - REDIS_URL=redis://redis:6379
    depends_on: [postgres, redis]

  transaction-service:
    build: ./services/transaction-service
    ports: ["3003:3000"]
    environment:
      - DATABASE_URL=postgresql://postgres:postgres@postgres:5432/log430_store
      - REDIS_URL=redis://redis:6379
    depends_on: [postgres, redis]

  saga-orchestrator-service:
    build: ./services/saga-orchestrator-service
    ports: ["3004:3000"]
    environment:
      - DATABASE_URL=postgresql://postgres:postgres@postgres:5432/log430_store
      - REDIS_URL=redis://redis:6379
    depends_on: [postgres, redis]

  complaint-service:
    build: ./services/complaint-service
    ports: ["3005:3000"]
    environment:
      - DATABASE_URL=postgresql://postgres:postgres@postgres:5432/log430_store
      - REDIS_URL=redis://redis:6379
      - RABBITMQ_URL=amqp://admin:admin123@rabbitmq:5672
    depends_on: [postgres, redis, rabbitmq]

  notification-service:
    build: ./services/notification-service
```

```

    ports: ["3006:3000"]
    environment:
      - RABBITMQ_URL=amqp://admin:admin123@rabbitmq:5672
    depends_on: [rabbitmq]

audit-service:
  build: ./services/audit-service
  ports: ["3007:3000"]
  environment:
    - DATABASE_URL=postgresql://postgres:postgres@postgres:5432/log430_store
    - REDIS_URL=redis://redis:6379
    - RABBITMQ_URL=amqp://admin:admin123@rabbitmq:5672
  depends_on: [postgres, redis, rabbitmq]

event-store-service:
  build: ./services/event-store-service
  ports: ["3008:3000"]
  environment:
    - DATABASE_URL=postgresql://postgres:postgres@postgres:5432/log430_store
    - REDIS_URL=redis://redis:6379
    - RABBITMQ_URL=amqp://admin:admin123@rabbitmq:5672
  depends_on: [postgres, redis, rabbitmq]

```

## 7.2.2 Infrastructure

```

postgres:
  image: postgres:15-alpine
  environment:
    POSTGRES_DB: log430_store
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  ports: ["5432:5432"]
  volumes:
    - postgres_data:/var/lib/postgresql/data
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 30s
    timeout: 10s
    retries: 3

redis:
  image: redis:7-alpine
  ports: ["6379:6379"]
  volumes:
    - redis_data:/data
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 30s
    timeout: 10s
    retries: 3

```

```

rabbitmq:
  image: rabbitmq:3-management-alpine
  ports:
    - "5672:5672"      # AMQP port
    - "15672:15672"    # Management UI
  environment:
    RABBITMQ_DEFAULT_USER: admin
    RABBITMQ_DEFAULT_PASS: admin123
    RABBITMQ_DEFAULT_VHOST: /
  volumes:
    - rabbitmq_data:/var/lib/rabbitmq
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "ping"]
    interval: 30s
    timeout: 10s
    retries: 3

kong:
  image: kong:latest
  environment:
    KONG_DATABASE: "off"
    KONG_DECLARATIVE_CONFIG: "/etc/kong/kong.yml"
    KONG_PROXY_LISTEN: "0.0.0.0:8000"
    KONG_ADMIN_LISTEN: "0.0.0.0:8001"
  volumes:
    - ./api-gateway/kong/kong.yml:/etc/kong/kong.yml
  ports:
    - "8000:8000"      # Public API
    - "8001:8001"      # Admin API
  depends_on:
    - user-service
    - catalog-service
    - transaction-service
    - complaint-service
    - audit-service
    - notification-service
    - event-store-service

```

### 7.2.3 Monitoring

```

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

```

```

- '--web.console.templates=/etc/prometheus/consoles'
- '--storage.tsdb.retention.time=200h'
- '--web.enable-lifecycle'

grafana:
  image: grafana/grafana:latest
  ports: ["3004:3000"]
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=admin
    - GF_USERS_ALLOW_SIGN_UP=false
  volumes:
    - grafana_data:/var/lib/grafana
    - ./monitoring/grafana/provisioning:/etc/grafana/provisioning
    - ./monitoring/grafana/dashboards:/var/lib/grafana/dashboards
  depends_on:
    - prometheus

node-exporter:
  image: prom/node-exporter:latest
  ports: ["9100:9100"]
  command:
    - '--web.listen-address=0.0.0.0:9100'

postgres-exporter:
  image: prometheuscommunity/postgres-exporter:latest
  ports: ["9187:9187"]
  environment:
    DATA_SOURCE_NAME:
      "postgres://postgres:postgres@postgres:5432/log430_store?sslmode=disable"
  depends_on:
    - postgres

redis-exporter:
  image: oliver006/redis_exporter:latest
  ports: ["9121:9121"]
  environment:
    REDIS_ADDR: "redis:6379"
  depends_on:
    - redis

```

### 7.2.4 Dashboards Grafana Spécialisés

Le système inclut des dashboards Grafana préconfigurés pour l'architecture événementielle :

1. **Golden Signals Dashboard** : Vue d'ensemble des quatre signaux dorés
2. **Event-Driven Architecture Dashboard** : Métriques événementielles spécialisées
3. **Saga Monitoring Dashboard** : Suivi des workflows distribués
4. **System Overview Dashboard** : Vue globale de l'infrastructure

### Configuration des Targets Prometheus :

```
# prometheus.yml
scrape_configs:
  - job_name: 'user-service'
    static_configs:
      - targets: ['user-service:3000']
    metrics_path: '/metrics'
    scrape_interval: 5s

  - job_name: 'catalog-service'
    static_configs:
      - targets: ['catalog-service:3000']

  - job_name: 'transaction-service'
    static_configs:
      - targets: ['transaction-service:3000']

  - job_name: 'saga-orchestrator-service'
    static_configs:
      - targets: ['saga-orchestrator-service:3000']

  - job_name: 'complaint-service'
    static_configs:
      - targets: ['complaint-service:3000']

  - job_name: 'audit-service'
    static_configs:
      - targets: ['audit-service:3000']

  - job_name: 'event-store-service'
    static_configs:
      - targets: ['event-store-service:3000']

  - job_name: 'notification-service'
    static_configs:
      - targets: ['notification-service:3000']

  - job_name: 'kong'
    static_configs:
      - targets: ['kong:8000']
    metrics_path: '/metrics'
```

```
- 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

```
```javascript
// 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. **Traffic** : 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']
});
```

### 8.3.3 Monitoring Événementiel Avancé

#### Métriques Spécialisées pour l'Architecture Événementielle :

```
// Événements - Publication et Consommation
const eventPublishingTotal = new prometheus.Counter({
  name: 'events_published_total',
  help: 'Total number of events published',
  labelNames: ['service', 'event_type', 'exchange', 'routing_key', 'status']
});

const eventConsumptionDuration = new prometheus.Histogram({
  name: 'event_consumption_duration_seconds',
  help: 'Duration of event consumption processing in seconds',
  labelNames: ['service', 'event_type', 'queue', 'status'],
  buckets: [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 2, 5, 10]
});

// CQRS - Commands et Queries
const commandExecutionTotal = new prometheus.Counter({
  name: 'command_executions_total',
  help: 'Total number of command executions',
  labelNames: ['service', 'command_type', 'status']
});
```

```
const queryExecutionDuration = new prometheus.Histogram({
  name: 'query_execution_duration_seconds',
  help: 'Duration of query executions in seconds',
  labelNames: ['service', 'query_type', 'status']
});

// Event Store - Opérations et Performance
const eventStoreOperationsTotal = new prometheus.Counter({
  name: 'event_store_operations_total',
  help: 'Total number of event store operations',
  labelNames: ['service', 'operation', 'status']
});

const eventStoreEventCount = new prometheus.Gauge({
  name: 'event_store_event_count',
  help: 'Total number of events in the event store',
  labelNames: ['service', 'aggregate_type']
});

// Sagas - Orchestration et Compensations
const sagaExecutionTotal = new prometheus.Counter({
  name: 'saga_executions_total',
  help: 'Total number of saga executions',
  labelNames: ['service', 'saga_type', 'status']
});

const activeSagas = new prometheus.Gauge({
  name: 'active_sagas',
  help: 'Number of currently active sagas',
  labelNames: ['service', 'saga_type', 'status']
});
```

## Dashboards Grafana Spécialisés :

### 1. Dashboard "Event-Driven Architecture" :

- Taux de publication/consommation d'événements par service
- Latence end-to-end des événements (émission → consommation)
- Distribution des types d'événements
- Taux d'erreur de traitement événementiel

### 2. Dashboard "CQRS Performance" :

- Ratio commands vs queries par service
- Performance des projections (lag de mise à jour)
- Throughput des opérations read vs write
- Cache hit rates des projections

### 3. Dashboard "Saga Monitoring" :

- Workflows actifs par type
- Taux de succès vs compensations

- Durée moyenne des workflows
- Étapes les plus fréquemment échouées

#### 4. Dashboard "Event Store Analytics" :

- Croissance du volume d'événements
- Performance des opérations de replay
- Distribution des types d'agrégats
- Utilisation de l'espace de stockage

#### Alerting Proactif :

```
# Exemples d'alertes Prometheus
groups:
- name: event-driven-alerts
  rules:
    - alert: HighEventProcessingLatency
      expr: histogram_quantile(0.95,
rate(event_consumption_duration_seconds_bucket[5m])) > 1
      for: 2m
      labels:
        severity: warning
      annotations:
        summary: "Latence élevée de traitement d'événements"
        description: "P95 latence > 1s pour {{ $labels.service }}"

    - alert: EventProcessingErrors
      expr: rate(event_processing_errors_total[5m]) > 0.1
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: "Taux d'erreur élevé dans le traitement d'événements"

    - alert: ProjectionLag
      expr: (time() - projection_last_update_timestamp) > 300
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "Retard dans la mise à jour des projections"

    - alert: SagaCompensationHigh
      expr: rate(saga_compensations_total[10m]) /
rate(saga_executions_total[10m]) > 0.1
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "Taux de compensation élevé pour les sagas"
```

## 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

### 9.8 ADR-008 : Architecture Événementielle avec Event Sourcing et CQRS

**Statut** : ACCEPTÉ (Implémenté décembre 2024)

**Décision** : Implémentation d'une architecture événementielle complète avec Event Sourcing, CQRS et message broker RabbitMQ.

#### Contexte :

Le système nécessite une traçabilité complète des opérations business, un découplage efficace entre services, et une scalabilité optimisée pour les opérations de lecture vs écriture.

#### Justification :

- **Event Sourcing** : Historique immuable et complet des événements business
- **CQRS** : Optimisation séparée des modèles de lecture et d'écriture
- **RabbitMQ** : Bus d'événements fiable avec retry logic et dead letter queues
- **Audit automatique** : Traçabilité complète via souscription universelle aux événements
- **Résilience** : Eventual consistency et compensation automatique

#### Composants implémentés :

- **Event Store Service** (Port 3008) : PostgreSQL pour persistance événements
- **Complaint Service** : Implémentation CQRS complète avec projections
- **Audit Service** (Port 3007) : Souscripteur universel aux événements
- **RabbitMQ** : Message broker avec échanges typés par domaine

#### Conséquences :

- Complexité accrue mais gains en observabilité et résilience
- Eventual consistency nécessitant adaptation des UIs
- Nouveaux services à maintenir (event-store, audit)
- Monitoring spécialisé pour l'architecture événementielle

### 9.9 ADR-009 : Implémentation CQRS avec Event Sourcing



**Statut** : ACCEPTÉ (Implémenté décembre 2024)

**Décision** : Implémentation du pattern CQRS avec Event Sourcing pour le service de réclamations (Complaint Service).

**Contexte** :

Le service de réclamations nécessite des opérations complexes de lecture et d'écriture avec besoins différents, ainsi qu'une traçabilité complète du cycle de vie des réclamations.

**Justification** :

- **Séparation Command/Query** : Optimisation indépendante des opérations
- **Projections spécialisées** : Read models optimisés par cas d'usage
- **Event Sourcing intégré** : Historique complet via replay d'événements
- **Performance** : Cache-friendly avec projections dénormalisées
- **Audit trail** : Traçabilité métier automatique

**Architecture technique** :



**Patterns implémentés** :

- Command Handlers avec validation métier
- Domain Aggregates avec logique business
- Event Handlers pour mise à jour projections
- Query Handlers pour requêtes optimisées

**Métriques spécialisées** :

- `command_executions_total` : Exécutions de commandes
- `query_executions_total` : Exécutions de requêtes
- `projection_updates_total` : Mises à jour projections
- Latence P95 pour toutes les opérations

**Conséquences** :

- Architecture plus complexe mais plus maintenable
- Eventual consistency entre commands et queries
- Performance améliorée pour les requêtes fréquentes
- Capacité d'analytics avancées sur l'historique

---

## 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
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

Terme	Définition
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
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 déploiement.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** : 2025-08-08 **Version** : 4.0