Rapport Architecture ARC42 - Système Microservices Multi-Magasins

Évolution Labo 3 à 5 - LOG430

Auteur : Talip Koyluoglu **Date** : 14 juillet 2025
Projet : Architecture Microservices avec Domain-Driven Design et API Gateway Kong
Évolution : Monolithe Django → Architecture Microservices DDD

1. Introduction et Objectifs

1.1 Aperçu du Système

Ce rapport documente l'évolution architecturale d'une application Django monolithique (Labo 3-4) vers une architecture microservices complète avec Domain-Driven Design, API Gateway Kong et communication inter-services HTTP.

1.2 Objectifs Architecturaux

Objectifs Fonctionnels :

- Décomposer l'application monolithique en 5 microservices orientés domaine métier (DDD)
- Centraliser l'accès via Kong API Gateway avec load balancing
- Maintenir la cohérence des données avec bases PostgreSQL dédiées par service
- Ajouter des fonctionnalités e-commerce (panier, clients, checkout)
- Fournir une interface unifiée via le frontend orchestrateur

Objectifs Non-Fonctionnels :

- **Scalabilité** : Services indépendants avec load balancing Kong
- **Performance** : Communication HTTP optimisée et architecture DDD
- **Disponibilité** : 99.9% uptime avec monitoring et health checks
- **Maintenabilité** : Déploiements indépendants par microservice
- **Observabilité** : Logs centralisés et APIs documentées Swagger

1.3 Stakeholders

Rôle	Responsabilités	Préoccupations
Employés Magasin	Interface unifiée multi-services	Performance, facilité d'utilisation
Clients E-commerce	Panier, commandes, comptes	Expérience utilisateur fluide
Gestionnaires	Rapports, stocks, décisions	Données cohérentes et fiables
Équipe DevOps	Déploiement, monitoring	Observabilité, scalabilité
Architecte Système	Évolution technique	Cohérence DDD, dette technique

2. Contraintes

2.1 Contraintes Techniques

- **Plateforme** : APIs REST avec Django + DRF (Python 3.11)
- **Bases de Données** : PostgreSQL dédiées par microservice (5 bases)
- **Communication** : HTTP/REST via Kong API Gateway
- **Déploiement** : Docker Compose avec services isolés
- **API Gateway** : Kong Gateway pour routage, load balancing et sécurité
- **Frontend** : Django avec clients HTTP vers microservices

2.2 Contraintes Organisationnelles

- **Architecture** : Domain-Driven Design obligatoire
- **Équipe** : Équipe unique gérant tous les microservices
- **Infrastructure** : Docker Compose (pas de Kubernetes)
- **Délai** : Migration progressive du monolithe vers microservices

2.3 Contraintes Règlementaires

- **Audit** : Traçabilité des transactions via logs structurés
- **Sécurité** : Authentification centralisée Kong avec clés API

---

3. Contexte

3.1 Contexte Métier

Domaine : E-commerce et retail multi-magasins unifiés
Modèle : Plateforme intégrée pour ventes physiques et en ligne
Enjeux : Scalabilité, maintenabilité, séparation des préoccupations métier

3.2 Évolution Architecturale

Labo 3-4 : Application Django Monolithique

```
[Frontend Django] ↔ [Application Monolithique] ↔ [PostgreSQL Unique]
```

- Application Django unique avec tous les use cases
- Base de données centralisée
- Architecture simple mais non scalable

Labo 5 : Architecture Microservices DDD

```
[Frontend Orchestrateur] ↔ [Kong Gateway] ↔ [5 Microservices DDD] ↔ [5
PostgreSQL Dédiées]
```

- 5 microservices indépendants suivant les principes DDD
- Kong Gateway avec load balancing et routing
- Bases de données dédiées par bounded context
- Communication HTTP avec clients dédiés

3.3 Contraintes d'Évolution

Éléments Conservés :

- Logique métier existante (ventes, gestion stock, reporting)
- Modèles de données essentiels adaptés en entités DDD
- Interface utilisateur unifiée

Éléments Transformés :

- Architecture monolithique → microservices DDD
- Base unique → 5 bases dédiées par domaine
- Appels directs → communication HTTP via Kong
- Services techniques → bounded contexts métier

Éléments Ajoutés :

- API Gateway Kong centralisé
- Services e-commerce (panier, clients, checkout)
- Architecture DDD complète (entités, use cases, value objects)

---

4. Stratégie de Solution

4.1 Approche Architecturale

Principe Directeur : *Domain-Driven Microservices with API Gateway*

J'ai adopté une approche **Domain-Driven Design (DDD)** pour décomposer le monolithe en 5 microservices alignés sur les domaines métier, orchestrés par Kong Gateway.

4.2 Patterns Architecturaux Appliqués

1. **Microservices Pattern** : Décomposition par bounded context DDD
2. **API Gateway Pattern** : Point d'entrée unique avec Kong
3. **Database per Service** : Isolation complète des données par domaine
4. **Domain-Driven Design** : Entités riches, use cases, value objects
5. **HTTP Communication Pattern** : Communication synchrone inter-services
6. **Frontend Orchestration Pattern** : Interface unifiée avec clients HTTP

4.3 Analyse Domain-Driven Design

Identification des Bounded Contexts

1. Bounded Context : Catalogue Management

```
Microservice: service-catalogue (Port 8001)
Database: produits_db (Port 5434)
Entités: Produit, Categorie
Value Objects: NomProduit, PrixMonetaire, ReferenceSKU
Use Cases: RechercherProduitsUseCase, AjouterProduitUseCase,
ModifierPrixUseCase
Endpoints: /api/ddd/catalogue/produits/, /api/ddd/catalogue/categories/
Responsabilités: Gestion catalogue, recherche produits, prix
```

2. Bounded Context : Inventory Management

```
Microservice: service-inventaire (Port 8002)
Database: inventaire_db (Port 5435)
Entités: StockCentral, StockLocal, DemandeReapprovisionnement
Value Objects: Quantite, SeuIlStock, ProduitId, MagasinId
Use Cases: GererStocksUseCase, CreerDemandeUseCase, ObtenirStockUseCase
Endpoints: /api/ddd/inventaire/stocks/, /api/ddd/inventaire/demandes/
Responsabilités: Stocks centraux/locaux, demandes réapprovisionnement
```

3. Bounded Context : Sales Management

```
Microservice: service-commandes (Port 8003)
Database: commandes_db (Port 5436)
Entités: Vente, Magasin
Value Objects: StatutVente, LigneVenteVO, CommandeVente
Use Cases: EnregistrerVenteUseCase, GenererRapportUseCase,
GenererIndicateursUseCase
Endpoints: /api/ddd/ventes/, /api/ddd/rapports/, /api/ddd/indicateurs/
Responsabilités: Ventes magasin, rapports consolidés, indicateurs
performance de chaque magasin
```

4. Bounded Context : Supply Chain Management

```
Microservice: service-supply-chain (Port 8004)
Database: supply_chain_db (Port 5437)
Entités: WorkflowValidation, DemandeReapprovisionnement
Value Objects: DemandeId, MotifRejet, LogValidation
Use Cases: ListerDemandesUseCase, ValiderDemandeUseCase,
RejeterDemandeUseCase
Endpoints: /api/ddd/supply-chain/demandes/, /api/ddd/supply-chain/valider/
Responsabilités: Workflow validation demandes, approbation/rejet,
transferts stock
```

5. Bounded Context : E-commerce Management

```
Microservice: service-ecommerce (Port 8005)
Database: ecommerce_db (Port 5438)
Entités: Client, Panier, ProcessusCheckout
Value Objects: AdresseLivraison, StatutCheckout, CommandeEcommerce
Use Cases: GererClientUseCase, GererPanierUseCase, ValiderCheckoutUseCase
Endpoints: /api/clients/, /api/panier/, /api/checkout/
Responsabilités: Comptes clients, panier d'achat, validation commandes e-commerce
```

Relations entre Bounded Contexts

Communication Inter-Services (HTTP) :

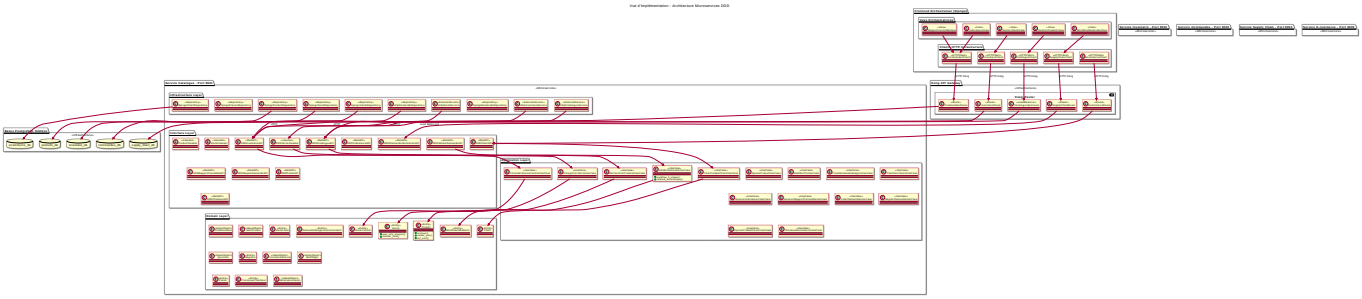
- **service-commandes** → **service-inventaire** : Réduction stock lors des ventes
- **service-supply-chain** → **service-inventaire** : Validation et transfert stock
- **service-ecommerce** → **service-catalogue** : Informations produits pour panier
- **service-ecommerce** → **service-commandes** : Création commandes finales
- **Tous** → **service-catalogue** : Validation existence produits et récupération d'informations des produits

---

5. Vue des Blocs de Construction

5.1 Architecture Microservices Complète

Diagramme Architecture - Vue d'implémentation



5.2 Vue Physique - Infrastructure de Déploiement

Architecture de Déploiement Docker

Kong Gateway (Point d'entrée unique)

- **Port principal** : 8080 (proxy) pour tous les clients
- **Port admin** : 8081 pour configuration Kong
- **Base Kong** : PostgreSQL kong_data (Port 5433)
- **Load Balancing** : Round-robin automatique

Microservices DDD Déployés

1. service-catalogue (Catalogue Load Balancé)

- **3 instances** : catalogue-service-1, catalogue-service-2, catalogue-service-3
- **Ports externes** : 8001, 8006, 8007
- **Port interne** : 8000 pour chaque instance
- **Base dédiée** : PostgreSQL produits_db (Port 5434)
- **Load balancing** : Round-robin via Kong

2. service-inventaire

- **1 instance** : inventaire-service
- **Port externe** : 8002
- **Port interne** : 8000
- **Base dédiée** : PostgreSQL inventaire_db (Port 5435)

3. service-commandes

- **1 instance** : commandes-service
- **Port externe** : 8003
- **Port interne** : 8000
- **Base dédiée** : PostgreSQL commandes_db (Port 5436)
- **Communication** : HTTP vers service-catalogue et service-inventaire

4. service-supply-chain

- **1 instance** : supply-chain-service
- **Port externe** : 8004
- **Port interne** : 8000
- **Base dédiée** : PostgreSQL supply_chain_db (Port 5437)
- **Communication** : HTTP vers service-inventaire pour workflow

5. service-ecommerce

- **1 instance** : ecommerce-service
- **Port externe** : 8005
- **Port interne** : 8005
- **Base dédiée** : PostgreSQL ecommerce_db (Port 5438)
- **Communication** : HTTP vers service-catalogue, service-inventaire, service-commandes

Frontend Orchestrateur

- **magasin/** : Interface Django unifiée
- **Ports** : 80 (NGINX), 8000 (Django)
- **Base** : PostgreSQL lab4db (Port 5432) - données UI uniquement
- **Communication** : HTTP clients vers Kong Gateway

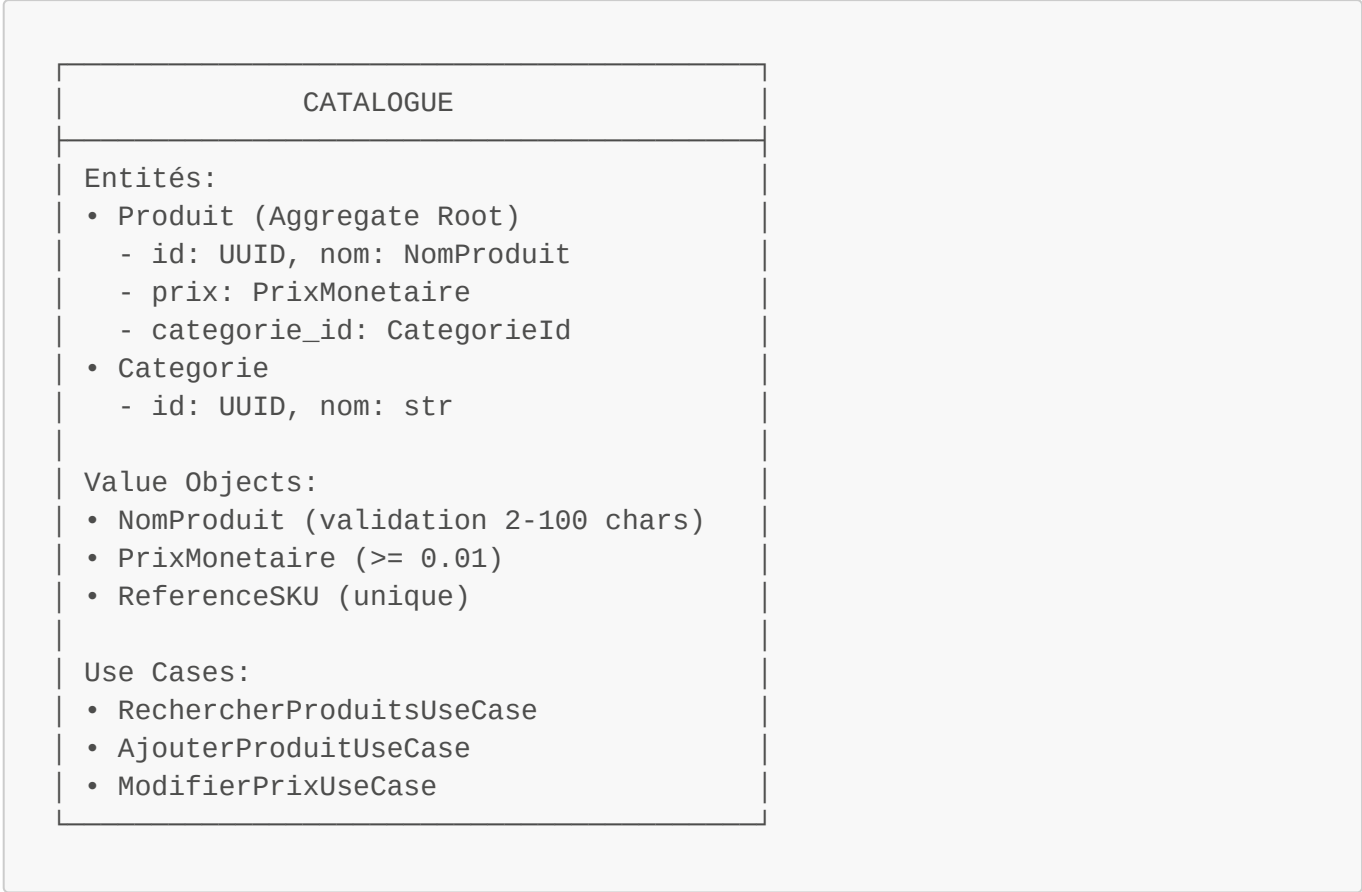
Infrastructure Support

- **Redis** : Cache (Port 6379)
- **Prometheus** : Métriques (Port 9090)
- **Grafana** : Monitoring (Port 3000)

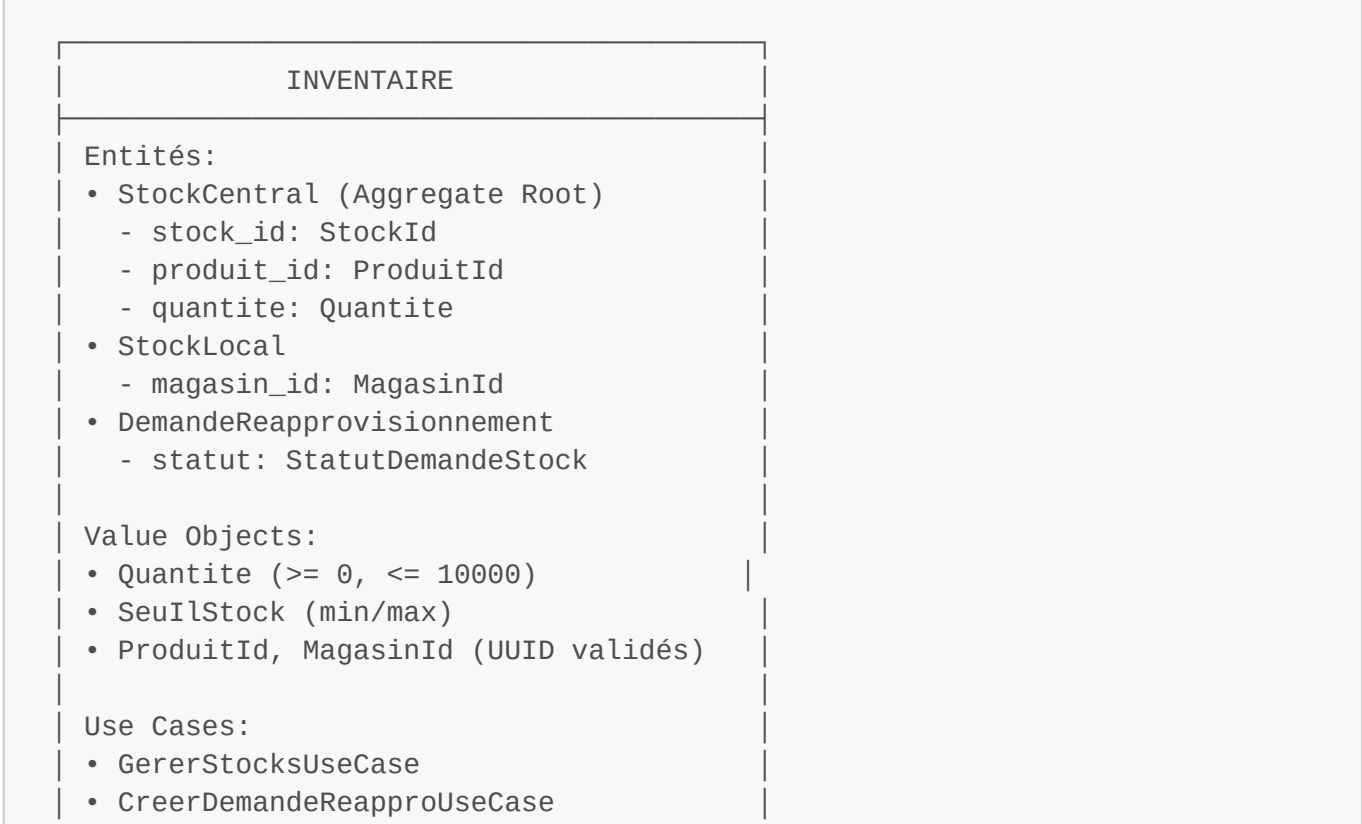
5.3 Vue Logique - Modèle DDD par Domaine

Diagramme Entités Métier par Bounded Context

1. Catalogue Domain (service-catalogue)



2. Inventory Domain (service-inventaire)



```
| • ObtenirStockUseCase                          |
└────────────────────────────────────────────────┘
```

3. Sales Domain (service-commandes)

```
┌────────────────────────────────────────────────┐
|              VENTES                            |
├────────────────────────────────────────────────┤
| Entités:                                       |
| • Vente (Aggregate Root)                       |
|   - id: UUID                                   |
|   - magasin_id: MagasinId                      |
|   - date_vente: datetime                       |
|   - statut: StatutVente                        |
| • Magasin                                      |
|   - id: UUID, nom: str                         |
|                                                |
| Value Objects:                                 |
| • LigneVenteVO (produit, quantité, prix)       |
| • StatutVente (ACTIVE, ANNULEE)                |
| • CommandeVente (validation complète)          |
|                                                |
| Use Cases:                                     |
| • EnregistrerVenteUseCase                      |
| • GenererRapportUseCase                        |
| • GenererIndicateursUseCase                    |
└────────────────────────────────────────────────┘
```

4. Supply Chain Domain (service-supply-chain)

```
┌────────────────────────────────────────────────┐
|           SUPPLY CHAIN                         |
├────────────────────────────────────────────────┤
| Entités:                                       |
| • WorkflowValidation (Aggregate Root)          |
|   - workflow_id: UUID                          |
|   - etapes_validees: List[str]                 |
| • DemandeReapprovisionnement (Remote)          |
|   - statut workflow validation                 |
|                                                |
| Value Objects:                                 |
| • DemandeId (UUID validé)                      |
| • MotifRejet (min 5 chars)                     |
| • LogValidation (étapes horodatées)            |
|                                                |
| Use Cases:                                     |
| • ListerDemandesUseCase                        |
| • ValiderDemandeUseCase                        |
```

```
| • RejeterDemandeUseCase                    |
└────────────────────────────────────────────┘
```

5. E-commerce Domain (service-ecommerce)

```
┌──────────────────────────────────────────────┐
│            E-COMMERCE                        │
├──────────────────────────────────────────────┤
│ Entités:                                     │
│ • Client (Aggregate Root)                    │
│   - id: UUID, email: EmailValue              │
│   - adresses: List[AdresseLivraison]         │
│ • Panier                                     │
│   - client_id: UUID                          │
│   - articles: List[ArticlePanier]            │
│ • ProcessusCheckout                          │
│   - statut: StatutCheckout                   │
│                                              │
│ Value Objects:                               │
│ • EmailValue (validation format)             │
│ • AdresseLivraison (complète)                │
│ • CommandeEcommerce (validation totale)      │
│                                              │
│ Use Cases:                                   │
│ • GererClientUseCase                         │
│ • GererPanierUseCase                         │
│ • ValiderCheckoutUseCase                     │
└──────────────────────────────────────────────┘
```

Relations Inter-Domaines :

- **service-commandes** → **service-inventaire** : Réduction stock via HTTP
- **service-supply-chain** → **service-inventaire** : Workflow validation via HTTP
- **service-ecommerce** → **service-catalogue** : Informations produits via HTTP
- **service-ecommerce** → **service-commandes** : Création commandes via HTTP

---

6. Vue d'Exécution

6.1 Vue Processus - Workflows Métier

Workflow 1 : Enregistrement Vente Magasin

```
Employé Magasin → Frontend → Kong → service-commandes → service-inventaire
                                          ↓                    ↓
                                [EnregistrerVenteUseCase] [RéduireStockUseCase]
                                          ↓                    ↓
                                   [DB commandes]      [DB inventaire]
```

Étapes détaillées :

1. **Employé** saisit vente dans interface magasin/
2. **Frontend** valide formulaire et appelle CommandesClient
3. **Kong Gateway** route vers service-commandes:8003
4. **service-commandes** exécute EnregistrerVenteUseCase :
   - Validation des données métier
   - Appel HTTP vers service-inventaire pour réduire stock
   - Création entité Vente avec lignes
   - Persistance en base commandes_db
5. **service-inventaire** exécute RéduireStockUseCase :
   - Validation stock disponible
   - Mise à jour StockLocal pour le magasin
   - Retour confirmation
6. **Frontend** affiche confirmation de vente

Workflow 2 : Validation Demande Réapprovisionnement

```
Gestionnaire → Frontend → Kong → service-supply-chain → service-inventaire
                                        ↓                       ↓
                          [ValiderDemandeUseCase]    [TransfererStockUseCase]
                                        ↓                       ↓
                              [Workflow 3 étapes]    [StockCentral → StockLocal]
```

Étapes détaillées :

1. **Gestionnaire** consulte demandes en attente via interface
2. **Frontend** appelle SupplyChainClient.lister_demandes_en_attente()
3. **Kong** route vers service-supply-chain:8004
4. **service-supply-chain** exécute ListerDemandesUseCase via HTTP vers service-inventaire
5. **Gestionnaire** clique "Valider" sur une demande
6. **service-supply-chain** exécute ValiderDemandeUseCase (workflow 3 étapes) :
   - **Étape 1** : Validation règles métier demande
   - **Étape 2** : Appel HTTP service-inventaire pour vérifier stock central
   - **Étape 3** : Appel HTTP service-inventaire pour transférer stock
   - **Rollback automatique** si échec à une étape
7. **service-inventaire** met à jour StockCentral (-) et StockLocal (+)
8. **Frontend** affiche confirmation ou erreur détaillée

Workflow 3 : Processus Checkout E-commerce

```
Client Web → Frontend → Kong → service-ecommerce → service-catalogue
                                      ↓                    ↓
                        [ValiderCheckoutUseCase]  [Validation produits]
                                      ↓
```

```
                        service-commandes
                              ↓
                    [Création commande finale]
```

Étapes détaillées :

1. **Client** valide son panier sur interface e-commerce
2. **service-ecommerce** exécute ValiderCheckoutUseCase :
   - Validation entité ProcessusCheckout
   - Appel HTTP service-catalogue pour valider produits
   - Calcul totaux avec frais livraison
   - Appel HTTP service-commandes pour créer commande finale
3. **service-commandes** crée Vente avec réduction stock automatique
4. **Confirmation** remontée jusqu'au client

6.2 Communication Inter-Services

Clients HTTP Dédiés

1. Frontend → Kong Gateway

```
# magasin/infrastructure/
CatalogueClient()    → http://kong:8080/api/catalogue/
InventaireClient()   → http://kong:8080/api/inventaire/
CommandesClient()    → http://kong:8080/api/commandes/
SupplyChainClient()  → http://kong:8080/api/supply-chain/
EcommerceClient()    → http://kong:8080/api/ecommerce/
```

2. Service-to-Service (via Kong)

```
# service-commandes → service-inventaire
HttpStockService()   → http://inventaire-service:8000/api/ddd/

# service-supply-chain → service-inventaire
HttpDemandeRepository() → http://inventaire-service:8000/api/ddd/

# service-ecommerce → service-catalogue
CatalogueService()   → http://catalogue-service:8000/api/ddd/
```

3. Configuration Kong Routes

```
# Kong routing vers microservices
/api/catalogue/*     → service-catalogue:8000
/api/inventaire/*    → service-inventaire:8000
/api/commandes/*     → service-commandes:8000
```

```
/api/supply-chain/*  → service-supply-chain:8000
/api/ecommerce/*     → service-ecommerce:8005
```

---

7. Vue de Déploiement

7.1 Configuration Docker Compose

Architecture de Déploiement Complète

```yaml
# Frontend + Load Balancer
nginx:
  ports: ["80:80"]
  depends_on: [app]

app:
  build: .
  environment:
    DJANGO_SETTINGS_MODULE: config.settings
  depends_on:
    db: {condition: service_healthy}

# Kong API Gateway
kong:
  image: kong:3.6
  ports: ["8080:8000", "8081:8001"]
  environment:
    KONG_DATABASE: postgres
    KONG_PG_HOST: kong-database
  depends_on:
    kong-migration: {condition: service_completed_successfully}

# Microservices avec Load Balancing
catalogue-service-1:
  build: ./service-catalogue
  ports: ["8001:8000"]
  environment:
    POSTGRES_HOST: produits-db
    INSTANCE_ID: catalogue-1

catalogue-service-2:
  build: ./service-catalogue
  ports: ["8006:8000"]
  environment:
    INSTANCE_ID: catalogue-2

catalogue-service-3:
  build: ./service-catalogue
  ports: ["8007:8000"]
  environment:
    INSTANCE_ID: catalogue-3
```

```
inventaire-service:
  build: ./service-inventaire
  ports: ["8002:8000"]
  environment:
    POSTGRES_HOST: inventaire-db

commandes-service:
  build: ./service-commandes
  ports: ["8003:8000"]
  environment:
    POSTGRES_HOST: commandes-db
    PRODUCT_SERVICE_URL: http://catalogue-service-1:8000
    STOCK_SERVICE_URL: http://inventaire-service:8000

supply-chain-service:
  build: ./service-supply-chain
  ports: ["8004:8000"]
  environment:
    POSTGRES_HOST: supply-chain-db
    STOCK_SERVICE_URL: http://inventaire-service:8000

ecommerce-service:
  build: ./service-ecommerce
  ports: ["8005:8005"]
  environment:
    POSTGRES_HOST: ecommerce-db
    CATALOGUE_SERVICE_URL: http://catalogue-service-1:8000

# Bases de données dédiées
produits-db:
  image: postgres:15
  ports: ["5434:5432"]
  environment:
    POSTGRES_DB: produits_db

inventaire-db:
  image: postgres:15
  ports: ["5435:5432"]
  environment:
    POSTGRES_DB: inventaire_db

commandes-db:
  image: postgres:15
  ports: ["5436:5432"]
  environment:
    POSTGRES_DB: commandes_db

supply-chain-db:
  image: postgres:15
  ports: ["5437:5432"]
  environment:
    POSTGRES_DB: supply_chain_db
```

```
  ecommerce-db:
    image: postgres:15
    ports: ["5438:5432"]
    environment:
      POSTGRES_DB: ecommerce_db
```

7.2 Configuration Kong Gateway

Load Balancing et Routing

```
# Script setup-kong.sh - Configuration automatique
# Upstream pour service-catalogue avec 3 instances
create_upstream_if_not_exists "catalogue-upstream" "round-robin"
create_target_if_not_exists "catalogue-upstream" "catalogue-service-1:8000"
100
create_target_if_not_exists "catalogue-upstream" "catalogue-service-2:8000"
100
create_target_if_not_exists "catalogue-upstream" "catalogue-service-3:8000"
100

# Services Kong avec routing
create_service_if_not_exists "catalogue-service" "http://catalogue-
upstream"
create_service_if_not_exists "inventaire-service" "http://inventaire-
service:8000"
create_service_if_not_exists "commandes-service" "http://commandes-
service:8000"
create_service_if_not_exists "supply-chain-service" "http://supply-chain-
service:8000"
create_service_if_not_exists "ecommerce-service" "http://ecommerce-
service:8005"

# Routes avec chemins spécifiques
create_route_if_not_exists "catalogue-route" "catalogue-service"
"/api/catalogue"
create_route_if_not_exists "inventaire-route" "inventaire-service"
"/api/inventaire"
create_route_if_not_exists "commandes-route" "commandes-service"
"/api/commandes"
create_route_if_not_exists "supply-chain-route" "supply-chain-service"
"/api/supply-chain"
create_route_if_not_exists "ecommerce-route" "ecommerce-service"
"/api/ecommerce"
```

---

8. Concepts Transversaux

8.1 Architecture Domain-Driven Design

8.1.1 Structure DDD par Service

Chaque microservice suit rigoureusement l'architecture DDD :

```
service-*/
├── domain/                    # Cœur métier pur
│   ├── entities.py           # Entités riches avec logique
│   ├── value_objects.py      # Value Objects avec validation
│   └── exceptions.py         # Exceptions domaine spécifiques
├── application/               # Orchestration use cases
│   ├── use_cases/            # Use Cases métier
│   ├── repositories/         # Interfaces abstraites
│   └── services/             # Services domaine
├── infrastructure/            # Implémentation technique
│   ├── django_*_repository.py
│   └── external_services/    # Communication HTTP
└── interfaces/               # Points d'entrée (API REST)
    └── *_views.py           # Controllers DDD
```

8.1.2 Exemple Concret : service-catalogue

Domain Layer - Logique métier pure :

```python
class Produit:  # Entité Aggregate Root
    def modifier_prix(self, nouveau_prix: PrixMonetaire):
        if nouveau_prix <= 0:
            raise PrixInvalideError("Le prix doit être positif")
        ancien_prix = self._prix
        self._prix = nouveau_prix
        self._historique_prix.append(nouveau_prix)
        self._date_modification = datetime.now()
```

Application Layer - Use Cases :

```python
class RechercherProduitsUseCase:
    def execute(self, criteres: CritereRecherche) -> Dict[str, Any]:
        # Orchestration avec validation métier
        self._valider_criteres(criteres)
        produits = self._produit_repo.rechercher(criteres)
        return self._formater_resultats(produits)
```

Infrastructure Layer - Implémentation :

```python
class DjangoProduitRepository(ProduitRepository):
    def rechercher(self, criteres: CritereRecherche) -> List[Produit]:
        # Conversion DDD ↔ Django ORM
```

```
        django_produits = ProduitModel.objects.filter(...)
        return [self._to_domain_entity(p) for p in django_produits]
```

Interface Layer - API REST :

```
class CatalogueViews(APIView):
    def post(self, request):
        # Controller DDD - orchestration pure
        use_case = AjouterProduitUseCase(self.produit_repo)
        result = use_case.execute(request.data)
        return Response(result)
```

8.2 Communication Inter-Services

8.2.1 Clients HTTP Structurés

Pattern Client HTTP par service :

```
class InventaireClient:
    def __init__(self, base_url: str = "http://kong:8080/api/inventaire"):
        self.session = requests.Session()
        self.session.headers.update({
            'Content-Type': 'application/json',
            'X-API-Key': 'magasin-secret-key-2025'
        })

    def creer_demande_reapprovisionnement(self, produit_id, magasin_id,
quantite):
        response = self.session.post(
            f"{self.base_url}/api/ddd/inventaire/demandes/",
            json={'produit_id': produit_id, 'magasin_id': magasin_id,
'quantite': quantite}
        )
        return response.json()
```

8.2.2 Gestion d'Erreurs Distribuées

Stratégies de résilience :

- **Timeout Configuration** : 5-30 secondes selon le service
- **Retry Logic** : 3 tentatives avec backoff exponentiel
- **Circuit Breaker** : Dégradation gracieuse si service indisponible
- **Fallback Response** : Données par défaut en cas d'erreur

```
def appel_service_avec_resilience(self, url, data):
    for tentative in range(3):
```

```
        try:
            response = requests.post(url, json=data, timeout=10)
            if response.status_code == 200:
                return response.json()
        except requests.RequestException as e:
            if tentative == 2:  # Dernière tentative
                logger.error(f"Service indisponible après 3 tentatives:
{e}")
                return {"success": False, "error": "Service temporairement
indisponible"}
            time.sleep(2 ** tentative)  # Backoff exponentiel
```

8.3 Sécurité et Authentification

8.3.1 Authentification Kong API Gateway

Authentification centralisée par clés API :

```
# Headers requis pour tous les appels
X-API-Key: magasin-secret-key-2025
Content-Type: application/json
Accept: application/json
```

Configuration Kong - Clés API :

```
# Création consumer
curl -X POST http://kong:8001/consumers/ --data "username=magasin-frontend"

# Attribution clé API
curl -X POST http://kong:8001/consumers/magasin-frontend/key-auth \
     --data "key=magasin-secret-key-2025"

# Plugin key-auth sur tous les services
curl -X POST http://kong:8001/services/catalogue-service/plugins \
     --data "name=key-auth"
```

8.3.2 Sécurité Inter-Services

Communication interne sécurisée :

- **Network isolation** : Services dans réseau Docker privé
- **No direct database access** : Chaque service a sa base dédiée
- **API-only communication** : Pas d'accès direct aux données
- **Validation systematique** : Value Objects avec validation stricte

8.4 Logging et Observabilité

8.4.1 Logging Structuré DDD

Logging par couche DDD :

```
# Domain layer - Événements métier
logger.info(f"Produit {produit.id} - Prix modifié: {ancien_prix} →
{nouveau_prix}")

# Application layer - Use cases
logger.info(f"[{use_case_name}] Début exécution avec params: {params}")
logger.info(f"[{use_case_name}] Résultat: {result}")

# Infrastructure layer - Calls externes
logger.debug(f"HTTP {method} {url} - Status: {status_code} - Durée:
{duration}ms")
```

Corrélation des logs inter-services :

- **Request-ID** propagé dans headers HTTP
- **Service-ID** identifiant unique par microservice
- **Use-Case-ID** traçage des workflows métier

8.4.2 Health Checks et Monitoring

Endpoints de santé par service :

```
# Chaque service expose /api/ddd/{service}/health-check/
class HealthCheckView(APIView):
    def get(self, request):
        return Response({
            "service": "catalogue",
            "status": "healthy",
            "database": self._check_database(),
            "external_services": self._check_external_services(),
            "timestamp": datetime.now().isoformat()
        })
```

Métriques métier DDD :

- **Nombre d'exécutions par use case**
- **Temps de réponse par bounded context**
- **Erreurs par domaine métier**
- **Communication inter-services (latence, erreurs)**

8.5 Performance et Scalabilité

8.5.1 Load Balancing Kong

Configuration load balancing :

- **service-catalogue** : 3 instances avec round-robin
- **Autres services** : 1 instance (scalable horizontalement)
- **Health checks** : Kong vérifie automatiquement la santé des instances
- **Failover automatique** : Retrait des instances défaillantes

8.5.2 Optimisations DDD

Performances par couche :

- **Domain** : Entités légères, logique métier optimisée
- **Application** : Use cases sans état, parallélisation possible
- **Infrastructure** : Connection pooling, requêtes optimisées
- **Interface** : Responses cachables, pagination automatique

Cibles de performance :

- **Latence p95** : < 500ms pour les workflows complexes
- **Throughput** : > 100 req/s par service
- **Disponibilité** : 99.9% avec monitoring automatique

---

9. Décisions

9.1 Architectural Decision Records (ADR)

ADR-001 : Domain-Driven Design comme Principe Architectural

Status : Accepted
Context : Besoin de décomposer un monolithe en microservices cohérents et maintenables
Decision : Adopter DDD avec bounded contexts, entités riches, use cases et value objects

Alternatives considérées :

- Architecture CRUD par entité technique
- Services orientés base de données
- Architecture en couches traditionnelle

Consequences :

- **Positives** : Séparation claire des préoccupations, logique métier centralisée, évolutivité
- **Négatives** : Courbe d'apprentissage, complexité initiale plus élevée
- **Métriques** : Code coverage >80%, séparation stricte des couches

ADR-002 : Kong API Gateway comme Point d'Entrée Unique

Status : Accepted
Context : Besoin d'orchestrer 5 microservices avec load balancing et sécurité
Decision : Kong Gateway avec routing automatique, load balancing et authentification par clés API

Alternatives considérées :

- NGINX + configuration manuelle
- HAProxy
- Service mesh (Istio)
- API Gateway AWS/Cloud

Consequences :

- **Positives** : Configuration déclarative, load balancing intégré, plugins riches
- **Négatives** : Point de défaillance unique, dépendance externe
- **Implémentation** : 3 instances service-catalogue load-balancées, routing automatique

ADR-003 : Communication HTTP Synchrone Inter-Services

Status : Accepted
Context : Besoin de communication cohérente entre microservices DDD
Decision : HTTP/REST synchrone avec clients dédiés et gestion d'erreurs avancée

Alternatives considérées :

- Message queues (RabbitMQ, Kafka)
- Event sourcing
- GraphQL fédéré
- gRPC

Consequences :

- **Positives** : Simplicité, debug facile, compatibilité universelle
- **Négatives** : Couplage temporel, latence réseau
- **Implémentation** : Clients HTTP avec retry, timeout et circuit breaker

ADR-004 : Base de Données par Service (Database per Service)

Status : Accepted
Context : Isolation complète des données par bounded context DDD
Decision : PostgreSQL dédiée par microservice avec schémas optimisés par domaine

Alternatives considérées :

- Base de données partagée avec schémas
- NoSQL par service
- Event store centralisé
- Vues matérialisées dédiées

Consequences :

- **Positives** : Isolation complète, évolution indépendante, performance optimisée
- **Négatives** : Pas de transactions ACID inter-services, gestion de cohérence complexe
- **Implémentation** : 5 bases PostgreSQL + 2 bases infrastructure (Kong, Frontend)

ADR-005 : Frontend Orchestrateur avec Clients HTTP

Status : Accepted
Context : Besoin d'interface unifiée malgré l'architecture microservices
Decision : Frontend Django avec clients HTTP dédiés par service via Kong

Alternatives considérées :

- Frontend découplé (React/Angular) avec API aggregation
- BFF (Backend for Frontend) dédié
- Micro-frontends
- Server-side composition

Consequences :

- **Positives** : Interface cohérente, transition douce du monolithe, control total
- **Négatives** : Frontend couplé aux APIs, pas d'indépendance équipes
- **Implémentation** : magasin/ Django avec clients HTTP structurés

9.2 Alternatives Écartées

Event-Driven Architecture : Complexité excessive pour les besoins métier
NoSQL par service : Pas de besoin de scalabilité extrême, PostgreSQL suffit
Kubernetes : Infrastructure trop complexe pour l'équipe et le projet
Micro-frontends : Équipe unique, interface cohérente préférée

---

10. Scénarios de Qualité

10.1 Performance

Scenario P1 : Charge Normale E-commerce

- **Source** : 50 utilisateurs simultanés sur interface + 20 commandes e-commerce/min
- **Stimulus** : Mix de consultation catalogue, ajout panier, checkout, ventes magasin
- **Environment** : 5 microservices + Kong + 3 instances catalogue
- **Response** : Load balancing équitable, communication HTTP fluide
- **Measure** : Latence p95 < 500ms, taux d'erreur < 2%, throughput > 100 req/s

Architecture Response :

- Kong load balancing rond-robin sur 3 instances catalogue
- Clients HTTP avec timeout 10s et retry automatique
- Health checks Kong toutes les 10 secondes

10.2 Disponibilité

Scenario A1 : Panne d'un Microservice

- **Source** : service-inventaire crash ou réseau coupé
- **Stimulus** : Tentatives d'accès stock et demandes réapprovisionnement

- **Environment** : Autres services opérationnels, Kong actif
- **Response** : Dégradation gracieuse avec messages d'erreur explicites
- **Measure** : Frontend continue de fonctionner, 95% des fonctionnalités disponibles

Architecture Response :

- Clients HTTP avec circuit breaker et messages fallback
- Interface utilisateur affiche statut services indisponibles
- Logs automatiques pour intervention rapide

10.3 Modifiabilité

Scenario M1 : Ajout Nouveau Service

- **Source** : Équipe développement ajoute service-notifications
- **Stimulus** : Nouveau bounded context pour notifications push
- **Environment** : Architecture DDD existante
- **Response** : Intégration sans impact sur services existants
- **Measure** : 2 jours de développement + déploiement

Architecture Response :

- Nouveau service suit template DDD existant
- Configuration Kong déclarative avec nouvelle route
- Communication via clients HTTP existants

10.4 Sécurité

Scenario S1 : Tentative d'Accès Non Autorisé

- **Source** : Attaquant tente d'accéder directement aux microservices
- **Stimulus** : Appels directs sans clé API Kong
- **Environment** : Kong Gateway + authentification active
- **Response** : Blocage automatique des requêtes non autorisées
- **Measure** : 100% des appels non autorisés bloqués

Architecture Response :

- Kong refuse toute requête sans X-API-Key valide
- Services internes accessibles uniquement via Kong
- Logs des tentatives d'intrusion

---

11. Risques et Dette Technique

11.1 Risques Techniques Identifiés

RISK-001 : Latence Communication Inter-Services

Probabilité : Moyenne
Impact : Élevé

Description : Workflows complexes (checkout e-commerce) nécessitent 3-4 appels HTTP séquentiels
Mitigation :

- Parallélisation des appels indépendants
- Cache intelligents aux niveaux appropriés
- Monitoring latence avec alertes
- Optimisation des requêtes base de données

RISK-002 : Panne Kong Gateway (SPOF)

Probabilité : Faible
Impact : Critique
Description : Kong Gateway point de défaillance unique pour tous les services
Mitigation :

- Health checks automatiques Kong
- Redémarrage automatique Docker Compose
- Plan de basculement vers accès direct services (urgence)
- Monitoring Kong avec alertes critiques

RISK-003 : Cohérence Données Distribuées

Probabilité : Moyenne
Impact : Élevé
Description : Pas de transactions ACID inter-services, risque d'incohérence
Mitigation :

- Patterns de compensation (rollback) dans use cases complexes
- Validation stricte via value objects DDD
- Monitoring cohérence avec rapports automatiques
- Tests d'intégration end-to-end robustes

11.2 Dette Technique

DEBT-001 : Communication HTTP Synchrone Exclusive

Description : Tous les appels inter-services sont HTTP synchrones, pas d'asynchrone
Impact : Latence cumulée sur workflows complexes, couplage temporel
Plan de résolution :

- **Phase 1** : Identifier use cases bénéficiant d'événements asynchrones
- **Phase 2** : Implémenter message queue (Redis Streams ou RabbitMQ)
- **Phase 3** : Migration progressive des appels non-critiques

DEBT-002 : Gestion d'Erreurs Inter-Services Basique

Description : Retry simple et timeout, pas de circuit breaker sophistiqué
Impact : Cascading failures possibles, pas de dégradation intelligente
Plan de résolution :

- **Phase 1** : Audit patterns de failure actuels
- **Phase 2** : Implémentation circuit breaker avancé (bibliothèque dédiée)
- **Phase 3** : Dégradation gracieuse par service

DEBT-003 : Configuration Kong Manuelle

Description : Setup Kong via script bash, pas d'IaC ou configuration versionnée
Impact : Risque de drift entre environnements, reproduction difficile
Plan de résolution :

- **Phase 1** : Migration vers configuration Kong déclarative (YAML)
- **Phase 2** : Intégration dans CI/CD
- **Phase 3** : Validation automatique configuration

---

12. Glossaire

Termes Architecturaux

API Gateway : Point d'entrée unique pour toutes les requêtes client vers microservices
Bounded Context : Frontière claire d'un domaine métier dans DDD
Circuit Breaker : Pattern de protection contre les pannes en cascade
Load Balancing : Répartition des requêtes entre plusieurs instances
Microservice : Service indépendant avec base de données et logique métier dédiées

Termes Domain-Driven Design

Aggregate Root : Entité racine contrôlant l'accès à un agrégat
Entity : Objet avec identité unique et cycle de vie
Use Case : Fonctionnalité métier orchestrant les interactions domaine
Value Object : Objet défini par ses attributs, sans identité
Repository : Interface d'accès aux données pour les entités

Termes Techniques Kong

Consumer : Entité Kong représentant un utilisateur/application
Plugin : Extension Kong (auth, rate limiting, logging)
Route : Configuration de routage Kong vers services backend
Service : Définition Kong d'un service backend
Upstream : Pool de targets Kong pour load balancing

Termes Métier

Checkout : Processus de validation et finalisation d'une commande e-commerce
Demande Réapprovisionnement : Demande de transfert stock central → local
Stock Central : Stock principal dans entrepôt central
Stock Local : Stock disponible dans un magasin spécifique
Workflow Validation : Processus d'approbation des demandes réapprovisionnement

Métriques et KPIs

Bounded Context Coverage : % de logique métier dans bounded contexts appropriés
Database per Service Ratio : Ratio services avec base dédiée vs partagée
DDD Layer Separation : Respect strict séparation Domain/Application/Infrastructure
Inter-Service Communication Latency : Latence moyenne appels HTTP inter-services
Use Case Completion Rate : % use cases exécutés avec succès

Fin du rapport ARC42
Document de référence pour l'architecture microservices DDD - LOG430 Labo5

Révisions :

- v1.0 : Architecture microservices DDD initiale
- v1.1 : Ajout service-ecommerce et patterns communication
- v1.2 : Documentation Kong Gateway et load balancing
- v1.3 : Finalisation patterns DDD et workflows métier