

# 1. Introduction

---

Ce rapport documente le troisième laboratoire du cours LOG430 – Architecture Logicielle (Été 2025) à l’ÉTS. L’objectif est de faire évoluer une application de caisse initialement locale vers une architecture logicielle plus modulaire, scalable et distribuée, capable de répondre aux besoins croissants d’une entreprise multi-magasins.

Au fil des laboratoires précédents, nous avons :

- Construit une base logicielle conteneurisée avec Docker (Lab 0)
- Développé une application 2-tier avec persistance PostgreSQL (Lab 1)
- Évolué vers une architecture **3-tier** avec :
  - Un backend FastAPI
  - Une base de données PostgreSQL
  - Une interface web React pour les tableaux de bord

Ce document, structuré selon le modèle **arc42**, détaille les décisions architecturales prises, les cas d'utilisation réalisés, les diagrammes UML, les stratégies de tests, l'intégration CI/CD, ainsi que les difficultés rencontrées et les solutions mises en place.

## 2. Objectifs et exigences

---

### 2.1 Objectifs pédagogiques du laboratoire

- Identifier les limites d'une architecture 2-tier dans un contexte multi-magasins
- Proposer une architecture logicielle distribuée adaptée à des contraintes réelles
- Appliquer des principes d'architecture : séparation des responsabilités, DDD, ADR, conteneurisation
- Démontrer la viabilité de l'architecture par un prototype fonctionnel
- Intégrer les pratiques DevOps (tests, CI/CD, Docker)

### 2.2 Exigences fonctionnelles (Méthode MoSCoW)

#### Must Have

- **UC1** : Générer un rapport consolidé des ventes par magasin (API + React)
- **UC2** : Consulter le stock central et initier une demande de réapprovisionnement
- **UC3** : Visualiser les performances des magasins dans un tableau de bord interactif (React)

#### Should Have

- **UC4** : Modifier les produits à partir de la maison mère (synchronisation)
- **UC6** : Approvisionner un magasin à partir du centre logistique

#### Could Have

- **UC7** : Générer une alerte automatique en cas de seuil critique de stock
- **UC8** : Fournir une interface web légère pour les gestionnaires à distance

## 2.3 Contraintes techniques et d'organisation

- Architecture déployable sur VM (via Docker)
- Modules réutilisables et testables individuellement
- API RESTful proprement définie et documentée (Swagger / OpenAPI)
- Code organisé en modules métiers (produits, ventes, réapprovisionnements, etc.)
- Réutilisation des modules précédents (Lab 0 et Lab 1) dans une architecture 3-tier

## 3. Contexte et limitations de l'architecture précédente

---

### 3.1 Contexte initial (Lab 1)

Lors du laboratoire 1, nous avons développé une application 2-tier composée de :

- Un client console (Python)
- Une base de données PostgreSQL conteneurisée
- Une logique métier directement couplée à l'interface CLI

Cette application permettait d'effectuer des ventes, de consulter les produits, et de gérer les stocks d'un seul magasin.

### 3.2 Limitations identifiées

L'approche 2-tier présente plusieurs **limitations majeures** dans un contexte multi-sites :

- **X Couplage fort** entre l'interface et la logique métier : difficilement réutilisable pour une interface web ou mobile.
- **X Pas d'accès distant** : impossible d'interagir avec l'application à travers un réseau (ex. : maison mère ou autres magasins).
- **X Monopoint de traitement** : toutes les actions sont faites localement, aucun mécanisme de centralisation ou de synchronisation.
- **X Non-scalable horizontalement** : un seul processus gère tout, difficilement réplicable.
- **X Pas de séparation claire des responsabilités** : difficile à tester, maintenir ou faire évoluer.

### 3.3 Besoins émergents justifiant une architecture 3-tier

L'entreprise souhaitant désormais gérer **plusieurs magasins**, un **centre logistique**, et une **maison mère**, nous avons identifié les besoins suivants :

- Centralisation des rapports de ventes
- Synchronisation cohérente des stocks
- Consultation à distance (via API)
- Visualisation par interface web
- Possibilité future d'héberger sur le cloud ou de développer une version mobile

Ces objectifs ne pouvaient être atteints sans refonte de l'architecture, d'où le passage à une solution **3-tier distribuée**.

## 4. Architecture proposée

---

L'architecture mise en place repose sur une approche **3-tier** distribuée, avec une séparation claire entre :

1. **Client** (frontend React)
2. **Serveur applicatif** (API FastAPI)
3. **Base de données** (PostgreSQL)

Ce découplage permet une meilleure évolutivité, une modularité accrue et une possibilité de déploiement en environnement distribué.

### 4.1 Vue logique (architecture métier)

- **Domaines fonctionnels :**
  - **Produit** : gestion du catalogue, informations des produits
  - **Vente** : traitement des ventes, annulation, calcul des totaux
  - **Réapprovisionnement** : demandes d'approvisionnement inter-magasins
  - **Rapport** : génération de rapports consolidés
- **Responsabilités bien séparées :**
  - `models/` → entités ORM SQLAlchemy
  - `schemas.py` → validation Pydantic
  - `services/` → logique métier
  - `routers/` → endpoints API REST

### 4.2 Vue de développement (organisation technique)

Arborescence du projet **Lab 2** :

```

app/
├── main.py          # Point d'entrée FastAPI
├── db.py            # Connexion PostgreSQL (SQLAlchemy)
├── models/          # Entités : Produit, Vente, etc.
├── services/        # Logique métier par domaine
├── routers/         # Routes FastAPI pour chaque use case
├── schemas.py       # Modèles Pydantic (entrée/sortie API)
└── tests/           # Tests unitaires (pytest)

front/
├── App.tsx          # Frontend React (UC3)
├── components/      # Composants réutilisables
└── services/api.ts  # Appels à l'API FastAPI

```

## 4.3 Vue de déploiement (infrastructure conteneurisée)

- **Docker Compose** orchestre les trois services :

- **api** : conteneur FastAPI
- **db** : conteneur PostgreSQL
- **front** : conteneur React

```
services:
  api:
    build: ./app
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      - POSTGRES_HOST=db
  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data
  front:
    build: ./front
    ports:
      - "3000:3000"

volumes:
  postgres_data:
```

## 4.4 Vue des processus (flux d'interactions)

### Exemple : UC1 – Générer un rapport consolidé

1. L'utilisateur (gestionnaire) accède au dashboard React.
2. React appelle **GET /ventes/rapport**.
3. FastAPI traite la logique via **services/rapport\_service.py**.
4. Les données sont récupérées depuis PostgreSQL, agrégées puis renvoyées au client.

Diagrammes disponibles :

- **sequence\_rapport.puml** (UC1)
- **sequence\_demande.puml** (UC2)

## 4.5 Vue des cas d'utilisation (Use Cases)

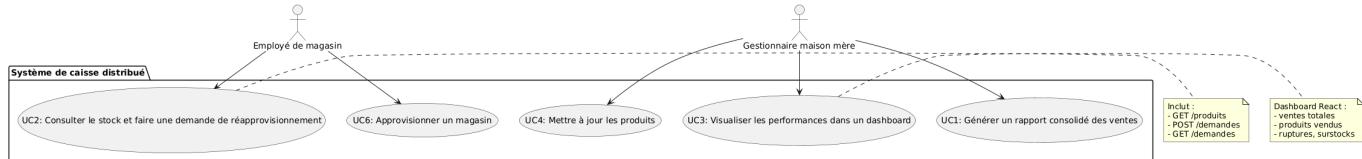
Voici les cas modélisés dans le diagramme **cas\_utilisation.puml** :

- **UC1** : Générer un rapport consolidé des ventes
- **UC2** : Consulter les stocks et déclencher un réapprovisionnement

- **UC3** : Visualiser les performances dans le dashboard
- **UC4** : Mise à jour d'un produit depuis la maison mère
- **UC6** : Approvisionner un magasin depuis le centre logistique

## 4.6 Diagrammes UML

### 4.6.1 Diagramme de cas d'utilisation

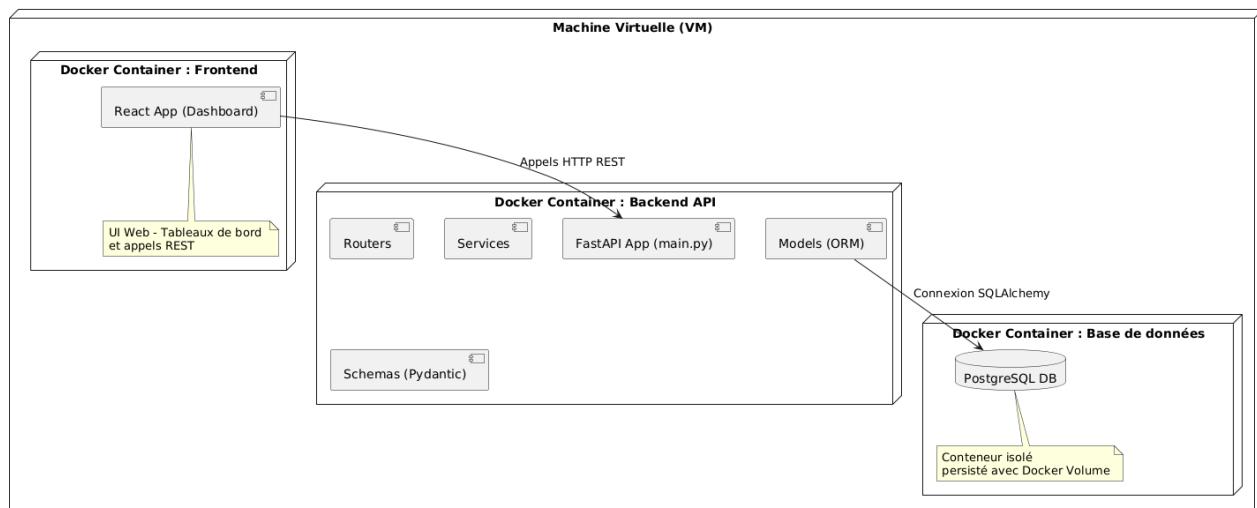


### 4.6.2 Diagramme de classes

### 4.6.3 Diagrammes de déploiement

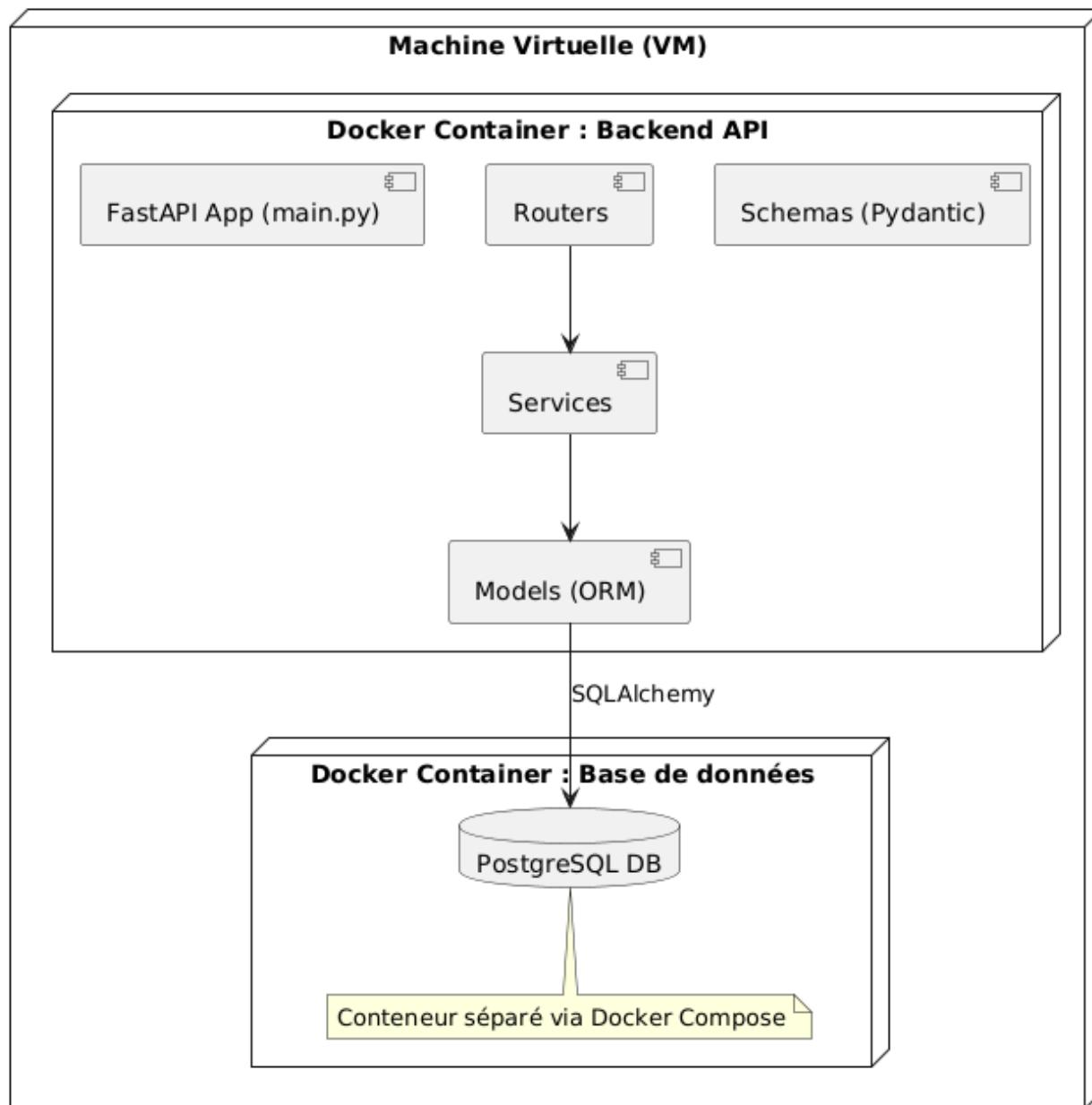
- **Déploiement 3-tier :**

Vue de déploiement - Architecture 3-tier (FastAPI + PostgreSQL + React)



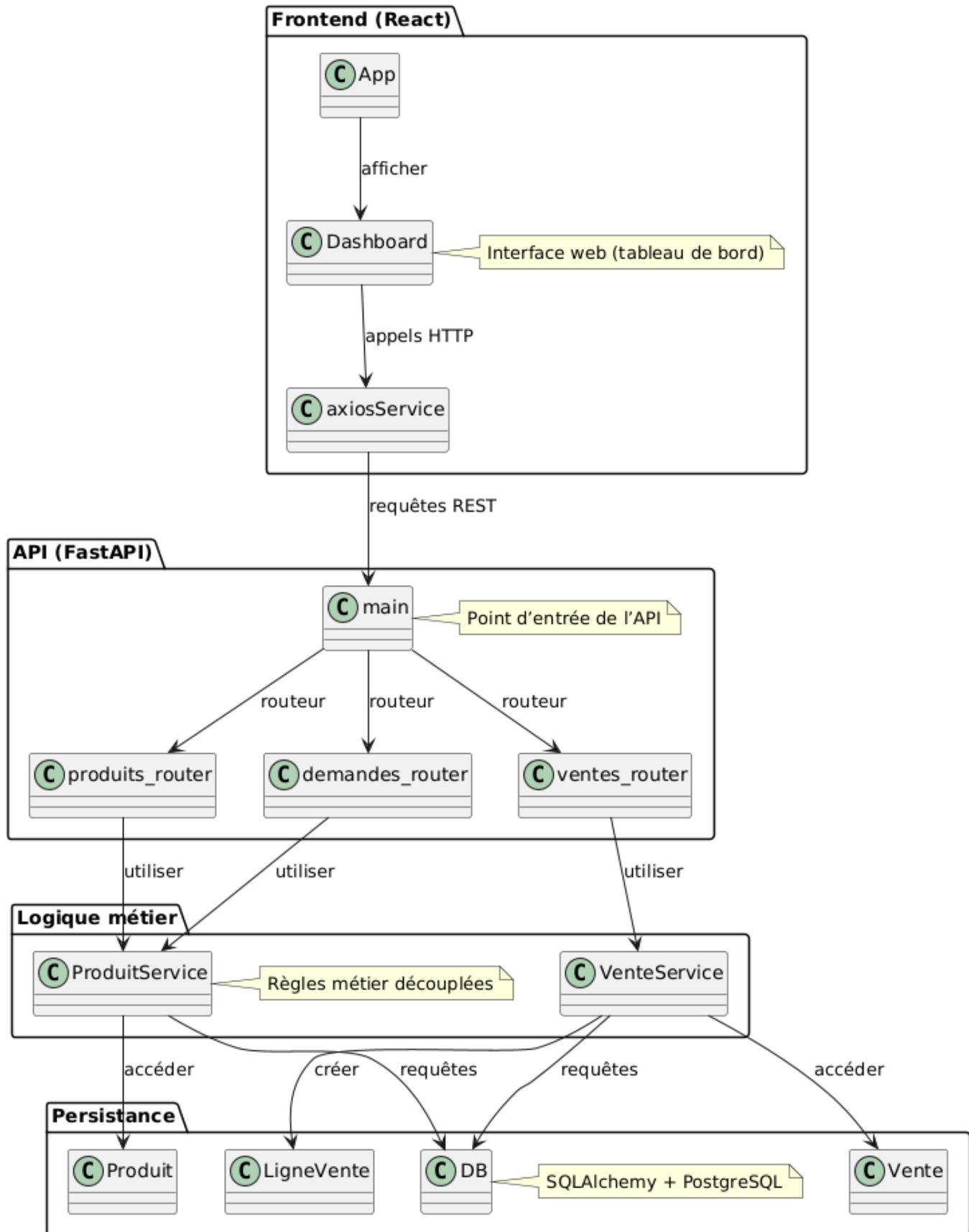
- **Déploiement PostgreSQL :**

**Vue de déploiement - Architecture 3-tier (API + PostgreSQL uniquement)**



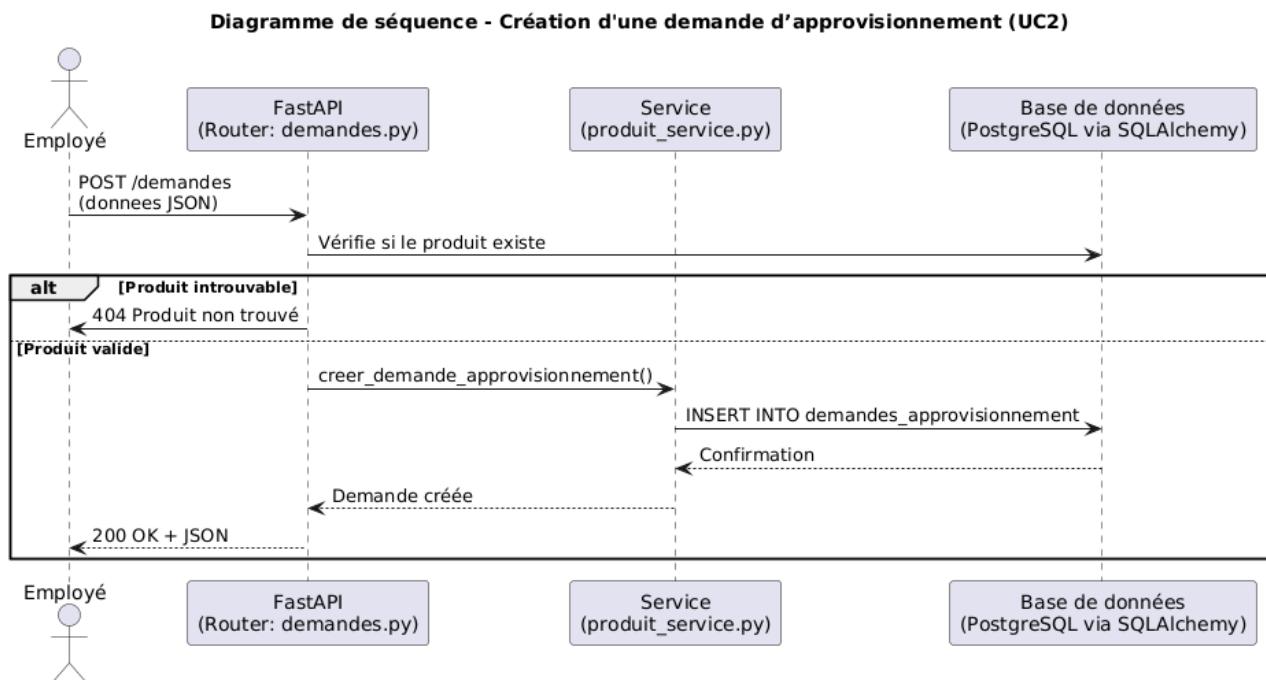
#### 4.6.4 Diagramme d'implémentation

### Vue d'implémentation - Architecture 3-tier (FastAPI + React)

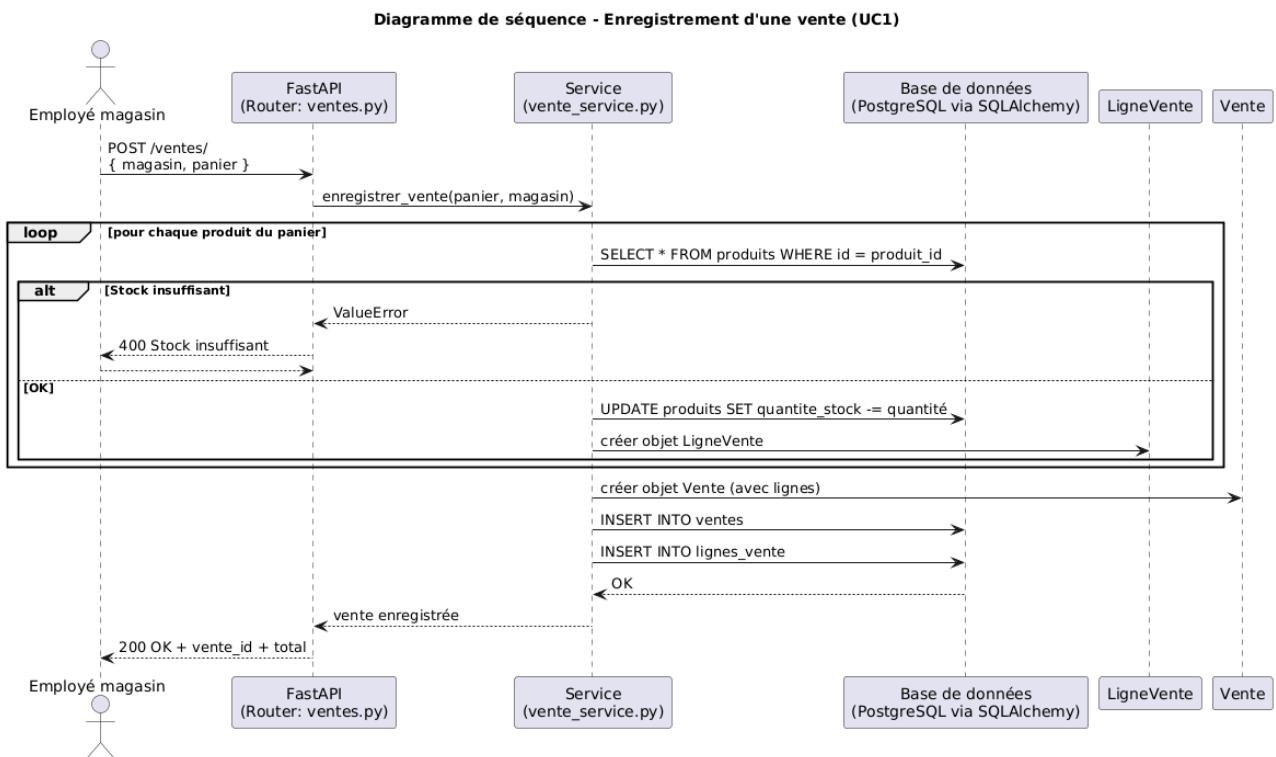


#### 4.6.5 Diagrammes de séquence

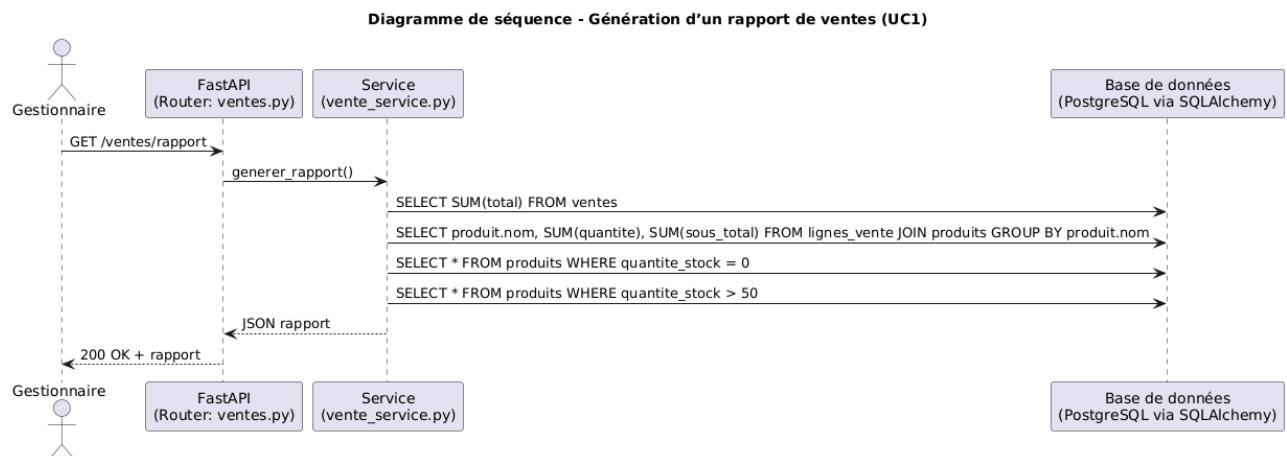
- **Création de demande :**



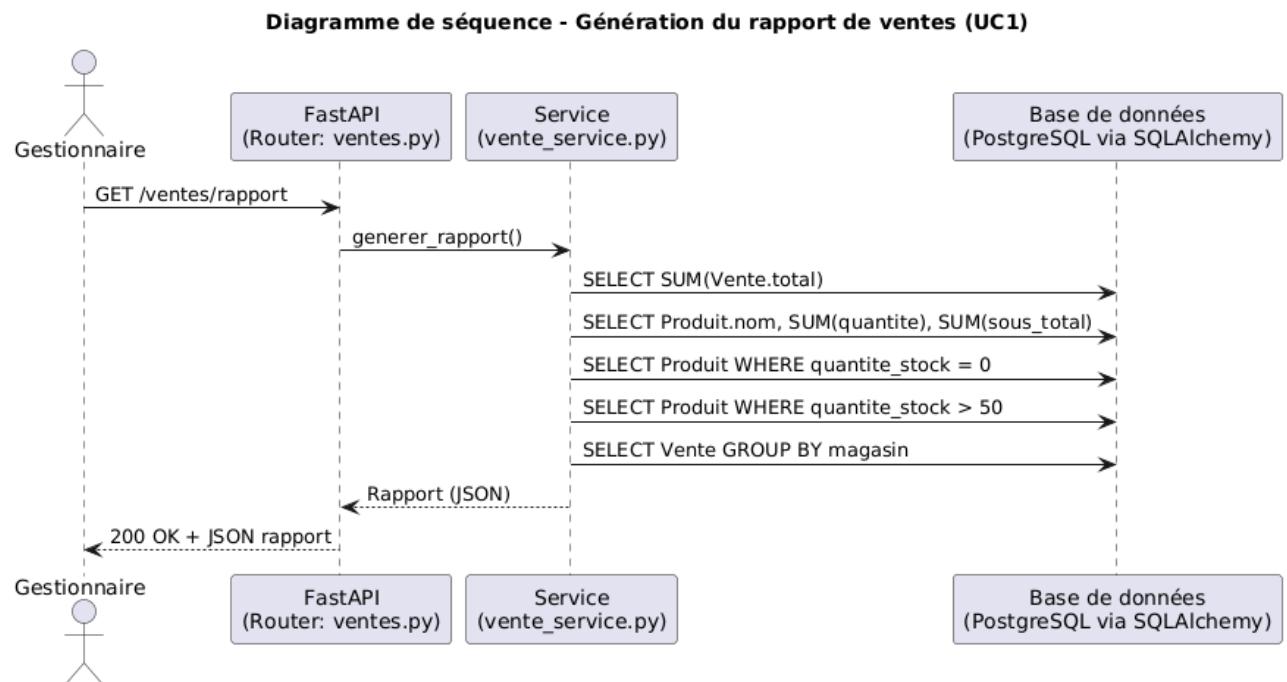
- **Enregistrement de vente :**



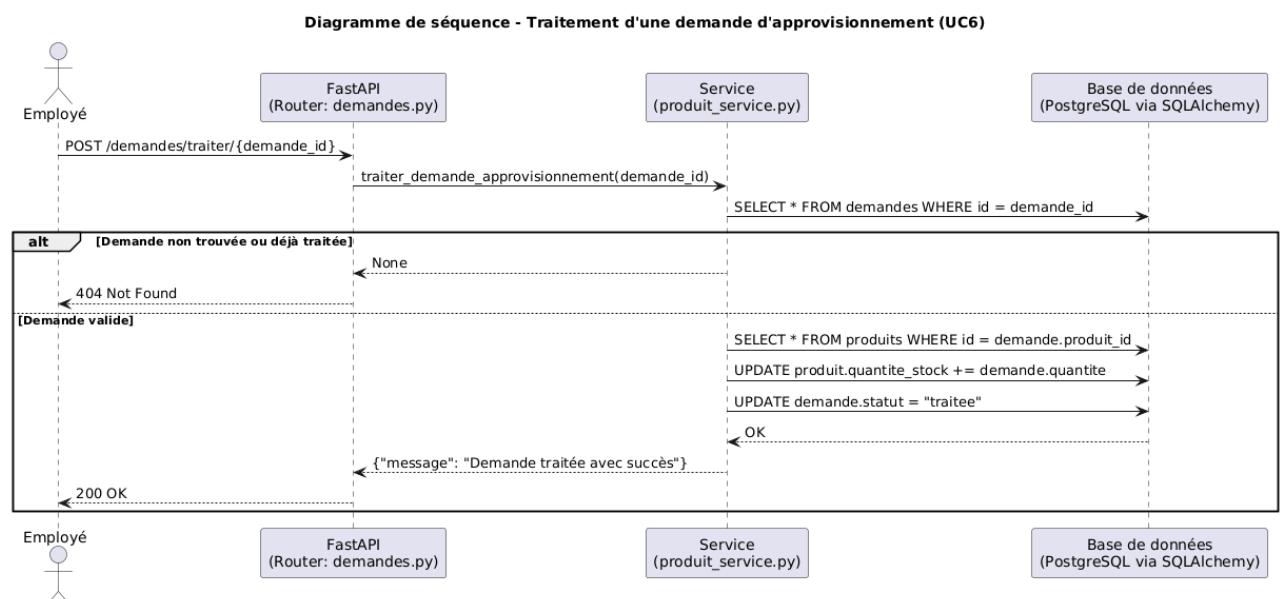
- Génération de rapport :**



- Rapport global :**



- Traitement de demande :**



## 5. Justification des choix technologiques (ADR)

Les choix technologiques et architecturaux majeurs ont été documentés via des **Architectural Decision Records (ADR)**, qui décrivent les décisions critiques, leurs justifications, alternatives rejetées et implications.

### 5.1 Choix du framework backend – FastAPI

#### [ADR-001 – Choix de FastAPI](#)

- **Motivation :**

- Performant et asynchrone
- Documentation automatique via Swagger
- Intégration fluide avec Pydantic pour la validation
- Typage statique pour une meilleure maintenabilité

- **Alternatives considérées :**

- Flask : trop minimaliste, moins typé
- Django : trop monolithique pour une architecture 3-tier

- **Conséquence :**

- Meilleure structuration des endpoints REST
- Gain de productivité via Swagger et typage strict

### 5.2 Choix de l'architecture 3-tier distribuée

#### [ADR-002 – Choix d'une architecture 3-tier](#)

- **Motivation :**

- Support d'interfaces multiples (console, web, mobile)
- Déploiement modulaire (frontend, backend, BDD)
- Séparation claire des responsabilités

- **Alternatives considérées :**

- Rester en 2-tier avec une simple couche API interne
- Utiliser un framework monolithique (Django fullstack)

- **Conséquence :**

- Infrastructure conteneurisée claire avec Docker Compose
- Scalabilité horizontale facilitée
- Intégration de CI/CD plus simple

### 5.3 Justification des autres choix techniques

Élément	Choix
retenus	Justification succincte

**ÉlémentChoix****retenuJustification succincte**

<b>Langage</b>	Python 3.11	Cohérent avec FastAPI, communauté active, facile à tester
<b>ORM</b>	SQLAlchemy	Intégration native avec FastAPI, support typage, compatible avec PostgreSQL
<b>BDD</b>	PostgreSQL	Moteur SQL robuste, open-source, supporte transactions, conteneurisé facilement
<b>Frontend</b>	React (JS/TS)	Flexible, écosystème riche, idéal pour dashboard interactif (UC3)
<b>CI/CD</b>	GitHub Actions	Intégré à GitHub, facile à configurer, exécution automatique de tests/lint/build
<b>Conteneurisation</b>	Docker Compose	Orchestration simple des 3 services (API, BD, Frontend)

## 6. Application de Domain-Driven Design (DDD)

Le projet a intégré les concepts fondamentaux du **Domain-Driven Design (DDD)** pour organiser la logique métier autour de sous-domaines explicites, améliorer la lisibilité du code, et favoriser une évolution structurée du système.

### 6.1 Identification des sous-domaines (Subdomains)

Le domaine métier de l'application a été divisé en **trois sous-domaines principaux** :

**Sous-domaineRôle**

<b>Vente</b>	Enregistre les ventes, permet leur annulation, calcule le chiffre d'affaires
<b>Stock / Logistique</b>	Gère les niveaux de stock, les ruptures, les réapprovisionnements
<b>Administration</b>	Permet de modifier les produits, consulter les indicateurs de performance

### 6.2 Bounded Contexts

Chaque sous-domaine possède son propre **contexte borné (Bounded Context)**, implémenté via des modules Python séparés dans `services/` et `models/`.

**Bounded ContextDescription technique**

<code>services/vente.py</code>	Gère les ventes, interactions avec la base
<code>services/stock.py</code>	Gère le stock et les demandes d'approvisionnement
<code>services/rapport.py</code>	Agrège les données de vente

Chaque contexte possède :

- Son propre modèle (`models/*.py`)
- Ses schémas de validation (`schemas.py`)
- Son service métier (`services/*.py`)
- Son routeur API (`routers/*.py`)

## 6.3 Entités et objets de valeur

### Entité principaleAttributs clés

Produit	<code>id, nom, prix, stock, seuil_rupture</code>
Vente	<code>id, produit_id, quantité, date, total</code>
DemandeApprovisionnement	<code>id, produit_id, quantité, magasin_source, magasin_cible, statut</code>

Les entités sont implémentées en SQLAlchemy dans le dossier `models/`.

## 6.4 Services métier (Domain Services)

La logique métier complexe est centralisée dans des services métier, par exemple :

- `traiter_vente(produit_id, quantite)` dans `services/vente.py`
- `generer_rapport()` dans `services/rapport.py`
- `verifier_rupture(produit)` dans `services/stock.py`

Cela permet :

- De **tester** la logique indépendamment
- De respecter le principe **Single Responsibility**
- De **réutiliser** la logique dans différents endpoints

## 6.5 Respect des principes DDD

### Principe DDDMise en œuvre dans le projet

Séparation des préoccupations	UI / API / Services / Modèles clairement séparés
Modèle riche orienté domaine	Services métiers dédiés pour chaque action complexe
Contexte borné	Modules isolés par fonction métier
Langage omniprésent (Ubiquitous)	Nommage cohérent : <code>Produit</code> , <code>Vente</code> , <code>Demande</code>

## 7. Intégration CI/CD et conteneurisation

Le projet met en œuvre une chaîne d'intégration et de déploiement continue (CI/CD) automatisée à l'aide de **GitHub Actions**, combinée à une architecture conteneurisée via **Docker** et **Docker Compose**.

### 7.1 Conteneurisation Docker

## Objectif

- Isoler les composants (API, base de données, frontend)
- Garantir la reproductibilité de l'environnement
- Faciliter le déploiement local ou sur le cloud

## Structure

Le fichier `docker-compose.yml` orchestre les services suivants :

### ServiceDescriptionPort

api	Application FastAPI	8000
db	PostgreSQL (base de données)	5432
front	Interface React (tableau de bord UC3)	3000

Chaque service possède son propre `Dockerfile`. Les volumes, ports et variables d'environnement sont clairement définis.

## 7.2 Pipeline CI/CD (GitHub Actions)

\*\* Fichier : \*\*``

Le pipeline s'exécute automatiquement à chaque `push` ou `pull request`. Il contient les étapes suivantes :

### ÉtapeOutil /

### commandeObjectif

<input checked="" type="checkbox"/> Linting	<code>black --check .</code>	Vérifie la conformité du code
<input checked="" type="checkbox"/> Tests	<code>pytest</code>	Exécute les tests unitaires
<input checked="" type="checkbox"/> Build Docker	<code>docker build .</code>	Valide la construction des images
 Push (opt.)	<code>docker push monuser/monimage</code>	Pousse l'image Docker sur Docker Hub

## Exemple d'exécution locale

```
docker-compose up --build          # Build complet de l'app
docker-compose run --rm api pytest # Exécution des tests dans le conteneur
```

## 7.3 Avantages obtenus

- Déploiement reproductible : l'environnement local est identique à l'environnement CI
- Tests automatisés à chaque modification
- Meilleure collaboration : chaque contributeur teste dans le même environnement
- Séparation claire des responsabilités : le frontend, l'API et la base ne se bloquent pas mutuellement

Cette configuration renforce la robustesse du système tout en respectant les bonnes pratiques modernes de DevOps.

## 8. Interface utilisateur – Tableau de bord (UC3)

---

Le cas d'utilisation **UC3** consiste à fournir aux gestionnaires un **tableau de bord visuel** leur permettant de suivre les performances des magasins : ventes, stocks, alertes, produits populaires, etc.

### 8.1 Choix du framework : React

- Choix motivé par sa flexibilité, sa large communauté et sa capacité à construire des interfaces interactives.
- Intégré dans le projet via un dossier `front/`, construit et servi en parallèle de l'API.

### 8.2 Fonctionnalités du tableau de bord

Le dashboard, accessible via `http://localhost:3000`, affiche dynamiquement des données provenant de l'API FastAPI :

Indicateur	Description
⌚ Chiffre d'affaires total	Agrégation des ventes par magasin
📊 Produits les plus vendus	Représentation en <b>bar chart</b>
⚠️ Produits en rupture de stock	Liste ou alertes visuelles
📦 Produits en surstock	Liste des produits avec stock excessif
📈 Répartition par magasin	Représentation en <b>pie chart</b>
🕒 Actualisation périodique	Requêtes <code>fetch</code> régulières vers l'API pour mise à jour en temps réel

### 8.3 Intégration frontend ↔ backend

- Les données sont récupérées via des appels à l'API FastAPI (`GET /ventes/rapport`, `GET /produits`).
- La gestion des appels se fait dans `front/services/api.ts`.
- Le composant principal `Dashboard.tsx` gère l'affichage des indicateurs.

### 8.4 Technologies utilisées

Technologie	Rôle
React	Interface utilisateur
Chart.js / Recharts	Graphiques interactifs
Axios / Fetch API	Requêtes HTTP vers l'API FastAPI
CSS Modules / Tailwind	Mise en forme

### 8.5 Exemple de capture d'écran

## 8.6 Observations

- Le dashboard fonctionne bien localement grâce à Docker Compose.
- Quelques défis ont été surmontés, notamment :
  - Problèmes de CORS entre frontend et backend
  - Synchronisation des données backend/frontend en temps réel

## 9. Tests automatisés

---

Les tests jouent un rôle central dans l'assurance qualité de l'architecture. Le projet intègre aujourd'hui des **tests unitaires** exécutés localement et dans la CI, avec l'objectif de mettre en place à terme une couverture exhaustive et des tests d'intégration.

### 9.1 Outils et organisation

Outil	Rôle
pytest	Exécution des tests unitaires et d'intégration légère
TestClient (FastAPI)	Simulation d'appels HTTP vers l'API
Docker Compose	Isolation des environnements de test via conteneurs

Les tests sont organisés dans le dossier `tests/` et nommés selon le module testé (produits, ventes, demandes, etc.).

### 9.2 Couverture actuelle

```
tests/
├── test_app.py      # Scénarios génériques de l'API (ping, produits, ventes,
                     # demandes)
└── test_ping.py    # Vérification de l'endpoint racine `/`
```

- test\_app.py** couvre :
  - `GET /produits/`, `GET /produits/recherche`, `POST /produits/`
  - `POST /ventes/` (cas valide et invalide)
  - `GET /ventes/rapport`
  - `POST /demandes/`
- test\_ping.py** : `GET /` retourne 200 OK

### 9.3 Exécution des tests

- Local** :

```
docker-compose run --rm api pytest
```

- **CI (GitHub Actions)** : intégré dans `.github/workflows/ci.yml` (lint, tests, build)

## 9.4 Perspectives et plan d'extension

- **Tests unitaires** : ajouter des modules de test pour chaque route de `produits.py`, `ventes.py`, `demandes.py`.
- **Tests d'intégration** : connexion à une base de test PostgreSQL pour valider les flux FastAPI ↔ BDD.
- **Tests E2E** : scénarios bout-à-bout sur le frontend React (Playwright ou Cypress).
- **Mesure de couverture** : intégrer `coverage.py` et reporting automatisé.

**Note** : la prochaine implémentation visera à atteindre une couverture unitaire de > 90 % et à intégrer les tests d'intégration et E2E.

# 10. Difficultés rencontrées et solutions apportées et solutions apportées\*\*

Tout au long du développement du laboratoire 2 et de la migration vers une architecture 3-tier distribuée, plusieurs défis techniques ont émergé. Voici les principaux obstacles rencontrés et les solutions apportées.

## 10.1 Problèmes liés à Docker et au rechargement automatique

- **Symptôme** : Le backend FastAPI ne se rechargait pas correctement après modification du code source.
- **Cause** : Unicorn dans Docker ne détectait pas les changements (problème de `volume mount`).
- **Solution** :
- Ajout de `--reload` dans la commande `CMD`
- Montage correct du volume : `./app:/code`
- Spécification du `working_dir` dans le Dockerfile

## 10.2 Intégration FastAPI ↔ React (CORS, fetch)

- **Symptôme** : Le frontend React ne pouvait pas faire de requêtes vers le backend (`CORS error`).
- **Solution** :

```
from fastapi.middleware.cors import CORSMiddleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Dev uniquement
    allow_methods=["*"],
    allow_headers=["*"]
)
```

- Vérification que le port de l'API (8000) est bien exposé et accessible.

## 10.3 Problèmes de dépendances circulaires FastAPI + SQLAlchemy

- **Symptôme** : ImportErrors ou erreurs au démarrage de l'app (ex. : module utilisé avant sa définition).
- **Solution** :
  - Refactorisation de `models/`, `services/`, et `schemas.py` pour éviter l'import croisé
  - Utilisation de `depends()` et `Depends(get_db)` pour injection propre des dépendances

## 10.4 Complexité des tests avec base de données isolée

- **Symptôme** : Les tests modifiaient la base réelle, ou échouaient à cause de l'état partagé.
- **Solution** :
  - Mise en place d'une **base de test dédiée**
  - Utilisation de `override_get_db()` pour injecter un `SessionLocal` temporaire avec rollback

## 10.5 Difficultés front-end (état, composants, graphiques)

- **Symptôme** : L'état des composants React ne reflétait pas toujours les données en temps réel.
- **Solution** :
  - Gestion centralisée de l'état dans `Dashboard.tsx`
  - Appels `fetch()` dans un `useEffect()` avec intervalle de rafraîchissement
  - Intégration progressive de bibliothèques comme `Recharts` pour les graphiques

## 10.6 Autres ajustements techniques

Problème	Solution technique apportée
Requêtes lentes à l'initialisation	Indexation des colonnes fréquentes dans PostgreSQL
Formatage automatique du code	Utilisation de <code>black</code> (via CI) pour uniformiser le style Python
Gestion des erreurs API (404, 422)	Ajout de gestionnaires d'erreurs ( <code>HTTPException</code> , <code>status_code</code> )

Ces ajustements ont permis de stabiliser l'architecture, d'assurer la fluidité entre les composants, et de garantir un déploiement fiable de bout en bout.

## 11. Instructions d'exécution

Cette section détaille comment **lancer, tester et utiliser** le système complet en local via Docker Compose, ainsi que via l'environnement d'intégration continue GitHub Actions.

### 11.1 Prérequis

- Docker installé (v20+ recommandé)
- Docker Compose (ou intégré via Docker Desktop)
- Git (pour cloner le dépôt)

⬇ Cloner le dépôt :

```
git clone https://github.com/ton_user/projet-log430-lab2.git  
cd projet-log430-lab2
```

## 11.2 Lancer tous les services (API + BD + Frontend)

```
docker-compose up --build
```

Cela démarre :

- Le backend FastAPI sur <http://localhost:8000>
- Le frontend React sur <http://localhost:3000>
- La base PostgreSQL (exposée en interne)

## 11.3 Tester l'API manuellement (Swagger)

Accessible via : <http://localhost:8000/docs>

Tu peux y tester les endpoints REST pour :

- `/produits` → consulter/modifier les produits
- `/ventes` → créer, annuler, lister les ventes
- `/demandes` → faire une demande d'approvisionnement
- `/ventes/rapport` → générer le rapport pour le tableau de bord

## 11.4 Accéder au tableau de bord (UC3)

Accessible via : <http://localhost:3000>

Fonctionnalités disponibles :

- Visualisation graphique des ventes
- Alerte sur les stocks
- Vue consolidée multi-magasins

## 11.5 Exécuter les tests

```
docker-compose run --rm api pytest
```

Les tests couvrent les modules :

- vente
- produit
- stock
- rapport

## 11.6 Formatter automatiquement le code (Python)

```
python -m black .
```

## 11.7 Pipeline CI/CD

Le pipeline se déclenche à chaque push ou pull request sur GitHub. Il exécute :

- Lint avec Black
- Tests avec Pytest
- Build Docker

Fichier de configuration : [.github/workflows/ci.yml](#)

## 11.8 Arrêter et nettoyer les conteneurs

```
docker-compose down
```

# 12. Conclusion et leçons tirées

Ce laboratoire a permis de transformer une application locale en une architecture logicielle distribuée, robuste, et évolutive. Grâce à une démarche itérative et structurée, le système est désormais prêt à supporter une entreprise multi-magasins avec des besoins réels en termes de visualisation, synchronisation et évolutivité.

## 12.1 Résumé des réalisations

- Migration réussie vers une architecture **3-tier distribuée**
- Intégration d'une API REST bien structurée avec **FastAPI**
- Mise en place d'un **dashboard React** dynamique (UC3)
- Conteneurisation complète avec **Docker + Docker Compose**
- Tests automatisés et pipeline **CI/CD via GitHub Actions**
- Application concrète des principes d'architecture logicielle : **DDD, ADR, modèle 4+1**

## 12.2 Leçons apprises

## Enseignement cléDétail

 Séparation des responsabilités	Clarifie l'architecture, simplifie les tests et la maintenance
 Structuration modulaire	Facilite la montée en charge et la collaboration
 Tests et CI dès le départ	Diminue les régressions, améliore la qualité continue
 Interopérabilité frontend/backend	Les problématiques CORS, format JSON, etc., doivent être gérées avec rigueur
 Infrastructure conteneurisée	Nécessite de bien comprendre le cycle de vie des services Docker
 Documentation architecturale (ADR, UML)	Utile pour justifier et tracer les décisions critiques

## 12.3 Améliorations possibles (Lab 3 ou projet futur)

- Ajouter un **système d'authentification et autorisation** (JWT, OAuth) pour sécuriser les endpoints.
- Centraliser la **gestion des erreurs** via un middleware global ([ErrorHandler](#)).
- Mettre en place une **base de test dédiée** et des **fixtures** pour isoler les tests d'intégration.
- Déployer un **pipeline de tests** automatisé avec **coverage** générant des rapports visibles dans la CI.
- Intégrer des **tests E2E** sur l'interface React (Playwright/Cypress) pour valider les workflows critiques.
- Ajouter une **infrastructure multitenant** ou **mode multi-lieux** (réplication/synchronisation) pour la production.
- Évoluer vers une **architecture à microservices** si les besoins métiers se complexifient (services indépendants).
- Documenter et versionner les **API** avec **OpenAPI/Swagger** et générer des SDK clients.
- Automatiser le **déploiement** sur un environnement cloud (Docker Swarm, Kubernetes).