

Rapport final LOG430 – Laboratoire 3 : API RESTful pour système multi-magasins

1. Introduction et objectifs

Ce rapport documente la refonte d'une application console en une API RESTful multi-magasins, dans le cadre du cours LOG430 – Architecture Logicielle (Été 2025) à l'ÉTS. L'objectif est de fournir une architecture distribuée, maintenable, testée et documentée, capable de gérer les opérations de caisse, le stock, les ventes, les demandes d'approvisionnement et la génération de rapports consolidés pour un dashboard React.

2. Contraintes et exigences

2.1 Objectifs pédagogiques

- Concevoir une API RESTful robuste, maintenable et sécurisée
- Appliquer les principes REST (stateless, URIs, statuts HTTP)
- Documenter l'API avec Swagger / OpenAPI
- Implémenter pagination, filtrage, gestion des erreurs
- Intégrer tests automatisés et CI/CD (GitHub Actions)

2.2 Exigences fonctionnelles (MoSCoW)

2.2 Exigences fonctionnelles – Priorisation MoSCoW

La méthodologie **MoSCoW** a été utilisée pour hiérarchiser les exigences fonctionnelles du laboratoire. Cette section présente à la fois les exigences prévues dans l'énoncé et celles réellement réalisées dans le projet.

☒ **Must Have – Implémentées avec succès**

Ces fonctionnalités étaient obligatoires et ont été livrées dans leur totalité :

- **UC1 – Générer un rapport consolidé des ventes**
 - Endpoint REST : `GET /api/ventes/rapport`
 - Agrégation des ventes, ruptures de stock, tendances
 - Utilisé par le dashboard React
- **UC2 – Consulter le stock d'un magasin**
 - Endpoint : `GET /api/magasins/{id}/stock`
 - Affiche les produits disponibles magasin par magasin
- **UC3 – Visualiser les performances globales**
 - Dashboard React (`frontend/dashboard`)
 - Données dynamiques issues de l'API REST
- **UC4 – Modifier un produit**
 - Endpoint : `PUT /api/produits/{id}`
 - Mise à jour du nom, prix, stock, catégorie
- **Architecture 3-tier** avec séparation nette : backend FastAPI, base PostgreSQL, frontend React
- **Tests unitaires** avec `pytest` et `TestClient`

- **CI/CD** : linting, test et build via GitHub Actions
-

● Should Have – Implémentées partiellement ou volontairement simplifiées

Ces éléments étaient souhaitables mais non critiques. Ils ont été livrés en version minimale ou optionnelle :

- **Authentification par token statique**
 - Implémentée via dépendance FastAPI (`x-token`)
 - Présente dans Swagger (champ "Authorize")
 - **Documentation complète Swagger/OpenAPI**
 - Routes annotées, `response_model`, exemples
 - Swagger enrichi via `custom_openapi()`
 - **Gestion des erreurs structurée**
 - Format JSON standardisé avec logs horodatés
 - Handlers FastAPI personnalisés
 - **Logs persistants et formatés**
 - Fichier `logs/erreurs.log` avec rotation journalière
 - **Séparation des responsabilités**
 - Dossiers `routers/`, `services/`, `models/`, `schemas.py`
-

● Could Have – Bonus réalisés

Ces améliorations étaient optionnelles mais ont été livrées comme bonus :

- **Pagination, tri et filtrage des produits**
 - Endpoint : `GET /api/produits/filtrer`
 - Paramètres : `page`, `size`, `sort`, `categorie`
 - **Fichier `.http`** pour tests manuels dans VSCode
 - **Simulation d'erreurs (404, 422, token manquant)** pour démonstration
-

✗ Won't Have – Non réalisés par choix ou hors scope

- **Authentification avancée (JWT, OAuth2)**
 - Considérée pour une version future, non exigée dans ce laboratoire
- **Rôles et permissions utilisateur**
 - Non requis dans les cas d'utilisation du Lab 3
- **Déploiement cloud (Heroku, AWS)**
 - Hors du périmètre demandé

2.3 Contraintes techniques


- FastAPI ≥ 0.110 , PostgreSQL, React, Docker Compose
- Respect des standards REST et HTTP
- CI/CD obligatoire (lint, test, build Docker)
- Swagger intégré (Pydantic)
- Sécurité minimale (token statique)

3. Architecture globale (arc42 + UML 4+1)

3.1 Vue contextuelle

Le système s'inscrit dans un contexte multi-magasins :

- **Acteurs** : gérants, caissiers, responsables logistiques
- **Interfaces** : API REST (FastAPI), dashboard React
- **Flux** : chaque requête HTTP correspond à un cas d'utilisation exposé

 Fig. 1 – Cas d'utilisation

3.2 Vue logique


Le système est organisé en cinq domaines fonctionnels : Produit, Vente, Magasin, Demande, Rapport global.

- **Responsabilités** :
 - Produit : gestion du catalogue, recherche, filtrage
 - Vente : enregistrement, annulation, rapport
 - Magasin : gestion du stock, consultation
 - Demande : création, traitement d'approvisionnement
 - Rapport : agrégation des données pour le dashboard

 Fig. 2 – Diagramme de classes

3.3 Vue de développement


- **Découpage technique** :
 - `app/main.py` : point d'entrée FastAPI
 - `routers/` : endpoints REST
 - `services/` : logique métier
 - `models/` : ORM SQLAlchemy
 - `schemas.py` : schémas Pydantic
 - `securite.py` : gestion du token
 - `logs/` : journalisation structurée
- **Frontend** : `frontend/dashboard/` (React)

 Fig. 3 – Diagramme de composants

3.4 Vue de déploiement

- **Orchestration** : Docker Compose
 - FastAPI (8000), PostgreSQL (5432), React (3000)
- **Variables d'environnement** : token, credentials DB
- **Volumes** : persistance des données PostgreSQL, logs

 Fig. 4 – Déploiement 3-tiers

 Fig. 5 – Déploiement PostgreSQL

3.5 Vue dynamique

Chaque scénario d'exécution est décrit par un diagramme de séquence :

- Authentification
- Création et traitement de demandes
- Enregistrement de ventes
- Génération de rapports
- Tri et pagination

 Fig. 6 – Séquence Authentification

 Fig. 7 – Séquence Création Demande

 Fig. 8 – Séquence Traitement Demande




 Fig. 9 – Séquence Enregistrement Vente

 Fig. 10 – Séquence Génération Rapport

 Fig. 11 – Séquence Rapport Global

 Fig. 12 – Séquence Tri & Pagination

4. Cas d'utilisation principaux

- Génération de rapports consolidés (GET `/api/ventes/rapport`)
- Consultation et mise à jour du stock (GET/PUT `/api/magasins/{id}/stock`)
- Enregistrement et gestion des ventes (POST/DELETE `/api/ventes/`)
- Création et traitement des demandes d'approvisionnement (POST `/api/demandes/`, POST `/api/demandes/traiter/{id}`)
- Visualisation des performances (GET `/api/performance/global`, dashboard React)

5. Catalogue des endpoints

Méthode	Endpoint	Description
GET	<code>/api/produits/</code>	Liste l'ensemble des produits
POST	<code>/api/produits/</code>	Ajoute un produit via paramètres de requête
GET	<code>/api/produits/recherche?critere=</code>	Recherche un produit par nom, catégorie ou identifiant
PUT	<code>/api/produits/{produit_id}</code>	Met à jour les détails d'un produit
GET	<code>/api/ventes/</code>	Liste les ventes enregistrées
POST	<code>/api/ventes/</code>	Enregistre une nouvelle vente
GET	<code>/api/ventes/rapport</code>	Génère un rapport consolidé des ventes
DELETE	<code>/api/ventes/{vente_id}</code>	Annule une vente
GET	<code>/api/demandes/</code>	Liste les demandes d'approvisionnement
POST	<code>/api/demandes/</code>	Crée une nouvelle demande

Méthode	Endpoint	Description
POST	<code>/api/demandes/traiter/{demande_id}</code>	Traite et valide une demande d'approvisionnement
GET	<code>/api/magasins/</code>	Récupère la liste des magasins
POST	<code>/api/magasins/</code>	Ajoute un nouveau magasin
GET	<code>/api/magasins/{id}/stock</code>	Affiche le stock d'un magasin spécifique
PUT	<code>/api/magasins/{id}</code>	Modifie les informations d'un magasin
DELETE	<code>/api/magasins/{id}</code>	Supprime un magasin
GET	<code>/api/stock/</code>	Récupère le stock global de tous les magasins
GET	<code>/api/performance/global</code>	Affiche les indicateurs globaux de performance
GET	<code>/api/rapports/ventes?date_debut=&date_fin=</code>	Génère un rapport de ventes sur une période spécifiée
GET	<code>/</code>	Ping / vérification de service

6. Documentation API

- **OpenAPI/Swagger** accessible via `/docs`
- Chaque route: résumé, description, schéma de réponse (`response_model`)
- Sécurité documentée (`x-token`)
- Champs `example=` dans les DTOs
- Swagger personnalisé via `custom_openapi()`
- Exemples de requêtes:

```
GET /api/produits/  
x-token: mon-token-secret  
  
POST /api/ventes/  
Content-Type: application/json  
x-token: mon-token-secret  
{  
  "magasin_id": 1,  
  "date": "2025-07-08",  
  "panier": [{"produit_id": 1, "quantite": 2}]  
}
```

7. Sécurité et gestion des erreurs

- Authentification via header `x-token` (dépendance FastAPI)
- Token défini dans `.env` ou `settings.py`

- Swagger intègre un bouton Authorize pour tester les endpoints protégés
 - Endpoints sensibles protégés (ventes, rapport, modification produit)
 - Logs structurés dans `logs/erreurs.log` (format JSON)
 - Gestion centralisée des erreurs (handlers FastAPI)
 - Codes HTTP standardisés, messages d'erreur normalisés
-

8. Pagination, filtrage et tri

- Endpoint: `GET /api/produits/filtrer`
 - Paramètres: `page`, `size`, `sort` (ex: `nom,asc`), `categorie`
 - Exemple d'appel: `/api/produits/filtrer?page=1&size=5&sort=prix,desc&categorie=fruit`
 - Pagination et tri gérés côté SQL (ORM)
-

9. Qualité, tests et CI/CD

9.1 Qualité logicielle

- Respect des bonnes pratiques REST
- Séparation des responsabilités (services, routers, models)
- Validation systématique (Pydantic)
- Logs et gestion des erreurs centralisés
- Couverture de tests sur les cas d'usage principaux

9.2 Tests

- `pytest` + `TestClient` pour les endpoints
- Fichier `.http` pour tests manuels (Postman compatible)
- Base de test dédiée pour l'intégration
- Tests couvrant: produits, ventes, demandes, rapports

9.3 CI/CD

- Lint (`black`)
 - Tests (`pytest`)
 - Build et test de l'image Docker
 - Pipeline GitHub Actions à chaque push/pull request
-

10. Difficultés rencontrées

1. Gestion asynchrone FastAPI + SQLAlchemy

L'intégration entre FastAPI (asynchrone) et SQLAlchemy (majoritairement synchrone) a posé des problèmes liés à la session et au blocage des threads.

☑ **Solution** : utilisation de `run_in_threadpool()` pour encapsuler les appels bloquants, assurant une exécution fiable dans un contexte async.

2. Injection du token dans Swagger

Swagger ne permettait pas par défaut de tester les endpoints protégés.

☑ **Solution** : ajout d'un schéma `apiKey` dans le header (`x-token`) via `custom_openapi()`, activant le bouton **Authorize** dans l'interface Swagger.

3. Configuration du cache Docker et des migrations

Des conflits de version et de cache empêchaient le bon rechargement du code dans Docker.

☑ **Solution** : ajout de `--reload` à Uvicorn, configuration correcte des volumes, et amélioration des scripts de migration (Alembic ou manuel).

↑ 4. Pagination et tri dynamique

Le parsing du paramètre `sort=champ,ordre` nécessitait une validation fine.

☑ **Solution** : parseur Pydantic avec vérification du champ et de l'ordre (`asc` / `desc`) + retour d'erreur explicite en cas de mauvaise syntaxe.

5. Structuration et rotation des logs

Les logs plats en texte étaient difficiles à analyser.

☑ **Solution** : mise en place d'un logger JSON avec rotation quotidienne (`TimedRotatingFileHandler`), facilitant l'analyse via des outils tiers.

6. Problèmes de CORS entre React et FastAPI

Le frontend React rencontrait des erreurs CORS (403 / blocages navigateur).

☑ **Solution** : ajout du middleware `CORSMiddleware` avec `allow_origins=["*"]` en développement, puis restriction par domaine en production.

7. Tests avec base de données isolée

L'environnement de test modifiait parfois la base réelle.

☑ **Solution** : override du `get_session()` pour injecter une base isolée (SQLite in-memory ou PostgreSQL de test) avec rollback automatique.

11. Améliorations possibles

- Authentification JWT (tokens dynamiques, gestion d'utilisateurs)
- Gestion fine des permissions (rôles, RBAC)
- Dashboard enrichi (graphiques temps réel, alertes)
- Déploiement cloud (Azure, AWS, GCP)
- Tests d'intégration bout-en-bout (frontend + backend)

- Versionnement de l'API (ex: `/api/v1/`)
 - Documentation automatisée des schémas et SDK clients
-

12. Glossaire

- **DTO**: Data Transfer Object (schéma Pydantic)
 - **ORM**: Object-Relational Mapping (SQLAlchemy)
 - **CI/CD**: Intégration et déploiement continus
 - **RBAC**: Role-Based Access Control
-

13. Conclusion

Ce laboratoire a permis de construire une **API RESTful professionnelle**, sécurisée et testée, dans une architecture **3-tier distribuée**.

J'ai appris à :

- Séparer proprement les responsabilités (router, service, modèle)
- Appliquer les principes REST (verbes, statuts, pagination...)
- Sécuriser l'API via des dépendances injectées
- Documenter l'API avec Swagger / OpenAPI enrichi
- Mettre en place des tests unitaires et manuels
- Utiliser Docker et GitHub Actions pour industrialiser le pipeline CI/CD

Les choix techniques, les patrons utilisés (ADR, DDD, 4+1), ainsi que les bonnes pratiques DevOps ont permis d'obtenir un projet **robuste**, **scalable**, et **maintenable**.

Fin du rapport – Été 2025