Rapport final – Laboratoire 5 (LOG430 - Été 2025)

1. Introduction

Ce rapport retrace l'évolution du projet multi-magasins réalisé dans le cadre du cours LOG430, lors de l'été 2025. À travers ce cinquième laboratoire, nous avons entrepris la migration complète d'un système monolithique, initialement structuré en 3 tiers, vers une architecture orientée microservices. Cette transformation vise à mieux répondre aux exigences de flexibilité, de scalabilité et de maintenabilité, tout en intégrant des mécanismes modernes d'observabilité et de gestion de la charge.

L'architecture cible repose sur une série de microservices spécialisés, chacun responsable d'un sous-domaine métier : gestion des produits, du stock, des ventes, des clients, des magasins, du panier d'achat et du processus de validation de commande (checkout). Chacun de ces microservices est conteneurisé, isolé, et possède sa propre base de données PostgreSQL. La communication entre les services s'effectue via des appels HTTP, orchestrés par une API Gateway centralisée : **KrakenD**. Enfin, un mécanisme de **load balancing** a été mis en place sur le `stock-service`, accompagné d'outils d'**observabilité** comme **Prometheus** et **Grafana** pour le suivi en temps réel du comportement du système.

---

2. Objectifs et exigences

Le principal objectif pédagogique de ce laboratoire était de pousser plus loin la modélisation logicielle en abordant la complexité réelle d'une architecture distribuée. Contrairement aux laboratoires précédents centrés sur la séparation logique en trois couches (présentation, logique métier, persistance), ce laboratoire introduit une fragmentation physique de l'application en plusieurs services autonomes. Cette démarche oblige à penser la communication entre composants, la robustesse des intégrations, la tolérance aux pannes, et les mécanismes de supervision nécessaires à une production viable.

Sur le plan fonctionnel, le système devait être capable d'exécuter les opérations classiques d'un mini-système e-commerce : création, mise à jour et consultation des produits, gestion du stock par magasin, enregistrement des ventes, suivi des clients, gestion du panier d'achat, et processus de commande. Le tout devait fonctionner de manière fluide et cohérente, même en présence d'une charge répartie sur plusieurs instances de certains services critiques comme le `stock-service`.

Les exigences non fonctionnelles occupaient également une place centrale. Il s'agissait notamment d'assurer une **scalabilité horizontale**, en instanciant plusieurs services identiques derrière un **proxy NGINX**. L'intégration de **Prometheus** permettait de collecter des métriques sur les temps de réponse et les performances, tandis que **Grafana** offrait une visualisation conviviale de l'état du système. D'autres aspects comme la **documentation Swagger**, la gestion centralisée des erreurs, et l'uniformisation des formats JSON contribuaient à améliorer la qualité globale du produit livré, tant du point de vue technique que de l'expérience développeur.

---

3. Contexte

Le projet s'appuie sur des technologies modernes, accessibles et bien intégrées entre elles. Le langage Python, couplé à **FastAPI**, a permis une construction rapide et lisible des différents services, tout en favorisant l'usage

de standards comme OpenAPI pour la documentation et Pydantic pour la validation des données. Chaque service repose sur sa propre base de données **PostgreSQL**, ce qui renforce l'indépendance des modules et limite les effets de bord lors des déploiements ou des pannes locales.

Tous les composants sont conteneurisés à l'aide de **Docker**, permettant une orchestration simple via `docker-compose`. Cette approche facilite non seulement les tests locaux mais ouvre également la porte à un futur déploiement dans des environnements cloud natifs. La **API Gateway KrakenD** joue un rôle fondamental : elle agit comme point d'entrée unique vers les microservices, filtre et réécrit les requêtes, applique des règles de quota, et centralise les logs.

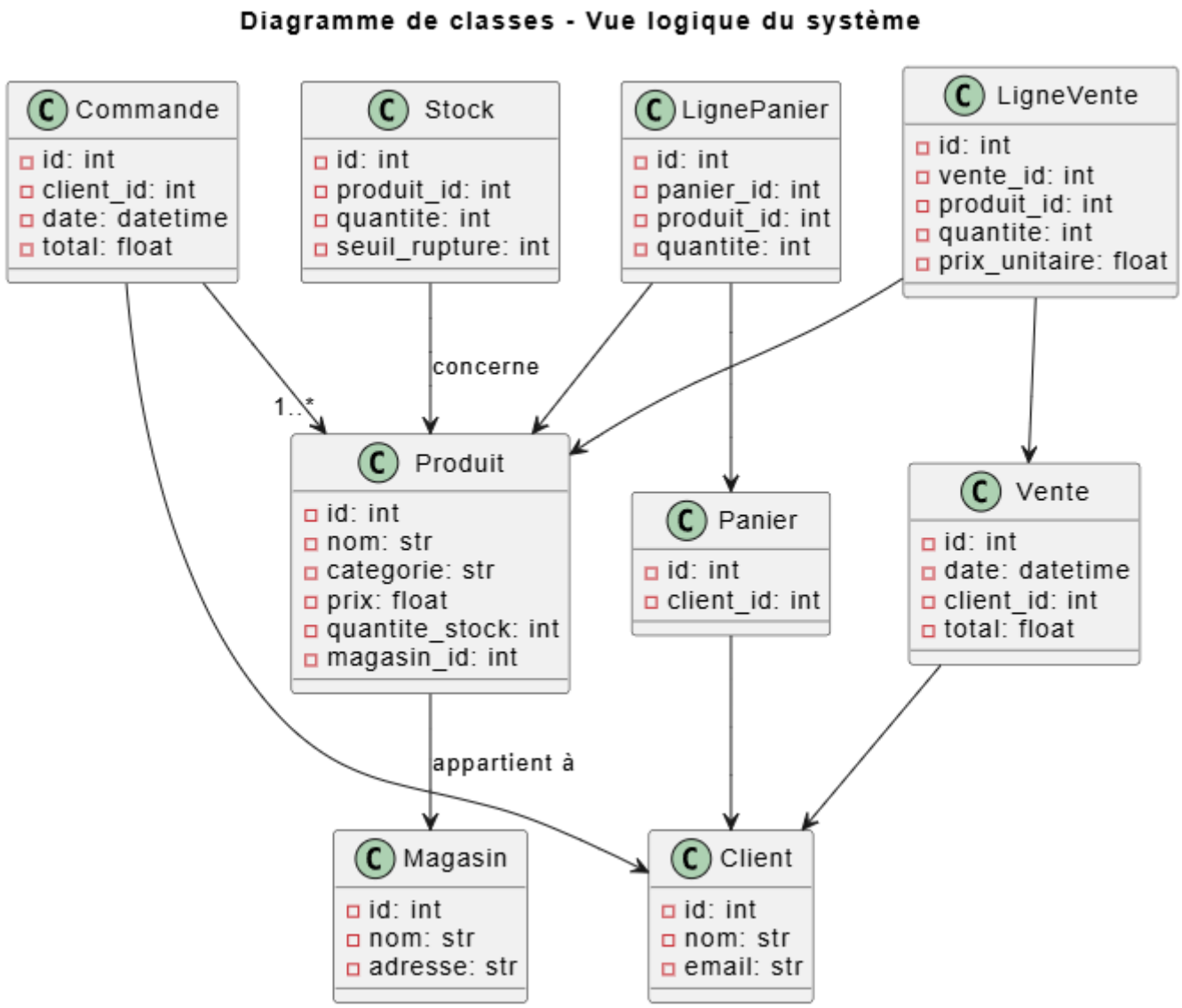
Enfin, les interconnexions entre services ont été pensées de manière explicite. Les appels HTTP entre les modules sont directs et bien définis. À titre d'exemple, la validation d'un panier implique un enchaînement d'opérations réparties entre les microservices `panier-service`, `stock-service`, et `ventes-service`, chacune jouant un rôle précis dans la transaction globale.

---

4. Architecture logique (Vue de développement)

Technologies utilisées

Couches	Composants
Frontend	React.js
API Gateway	KrakenD
Microservices	FastAPI, Pydantic, PostgreSQL
Observabilité	prometheus_fastapi_instrumentator, Grafana
Orchestration	Docker, Docker Compose



---

5. Architecture physique (Vue de déploiement)

Tous les services sont conteneurisés et communiquent via le bridge `log430-network`. Le `stock-service` est répliqué en deux instances, avec NGINX en frontal comme **load balancer**.

```
Utilisateur → KrakenD → Microservices (via réseau Docker)
                                ↓
                ↪ Prometheus ↪ Grafana
```

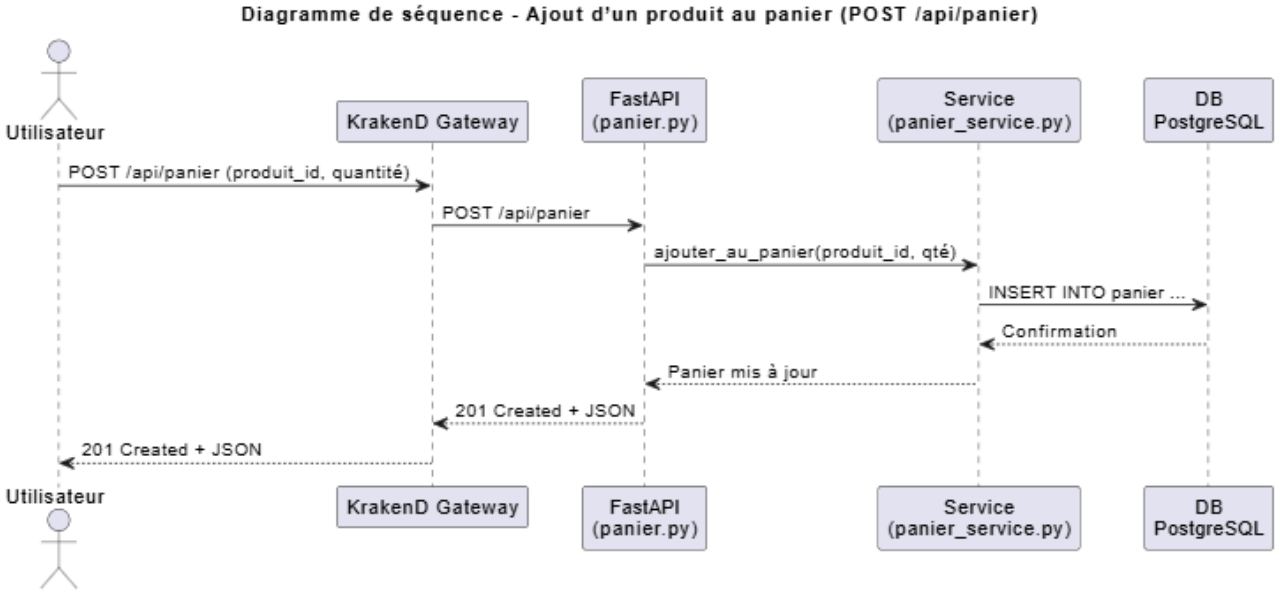
---

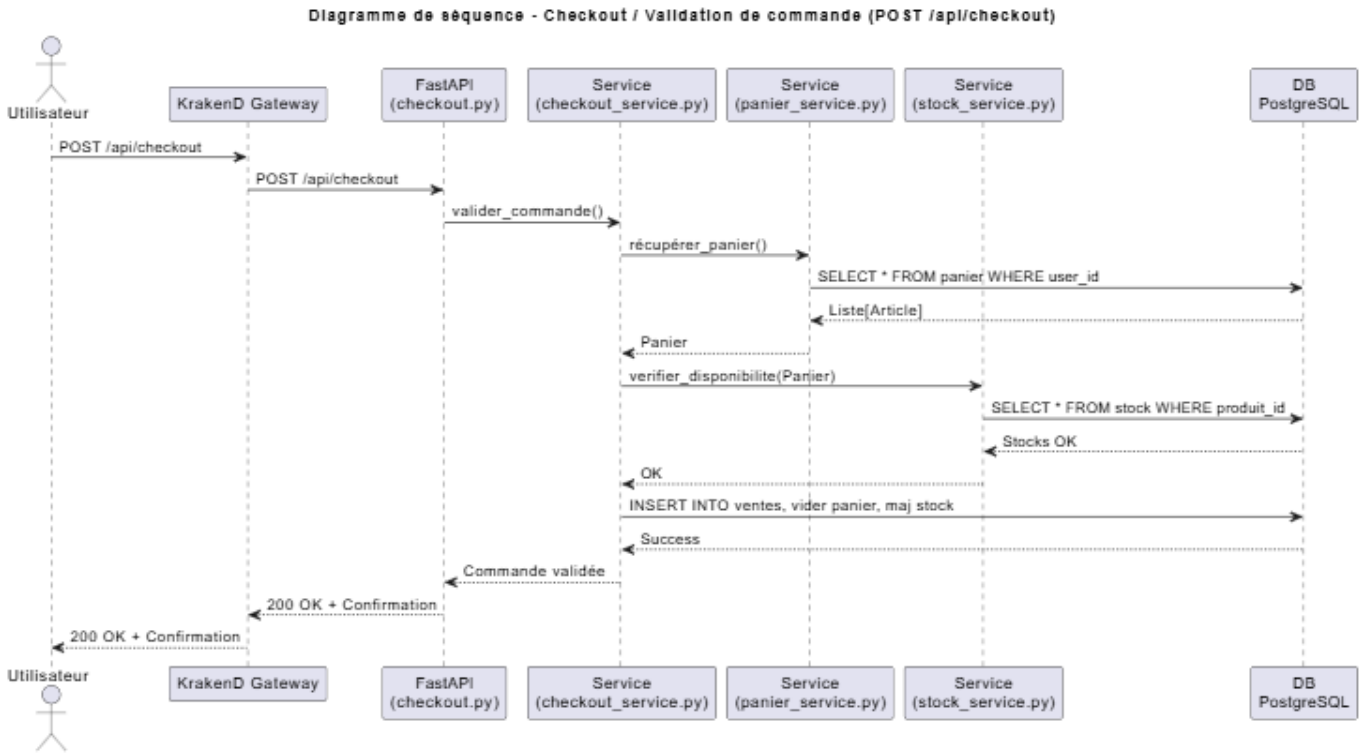
6. Architecture des cas d'utilisation (Vue logique)

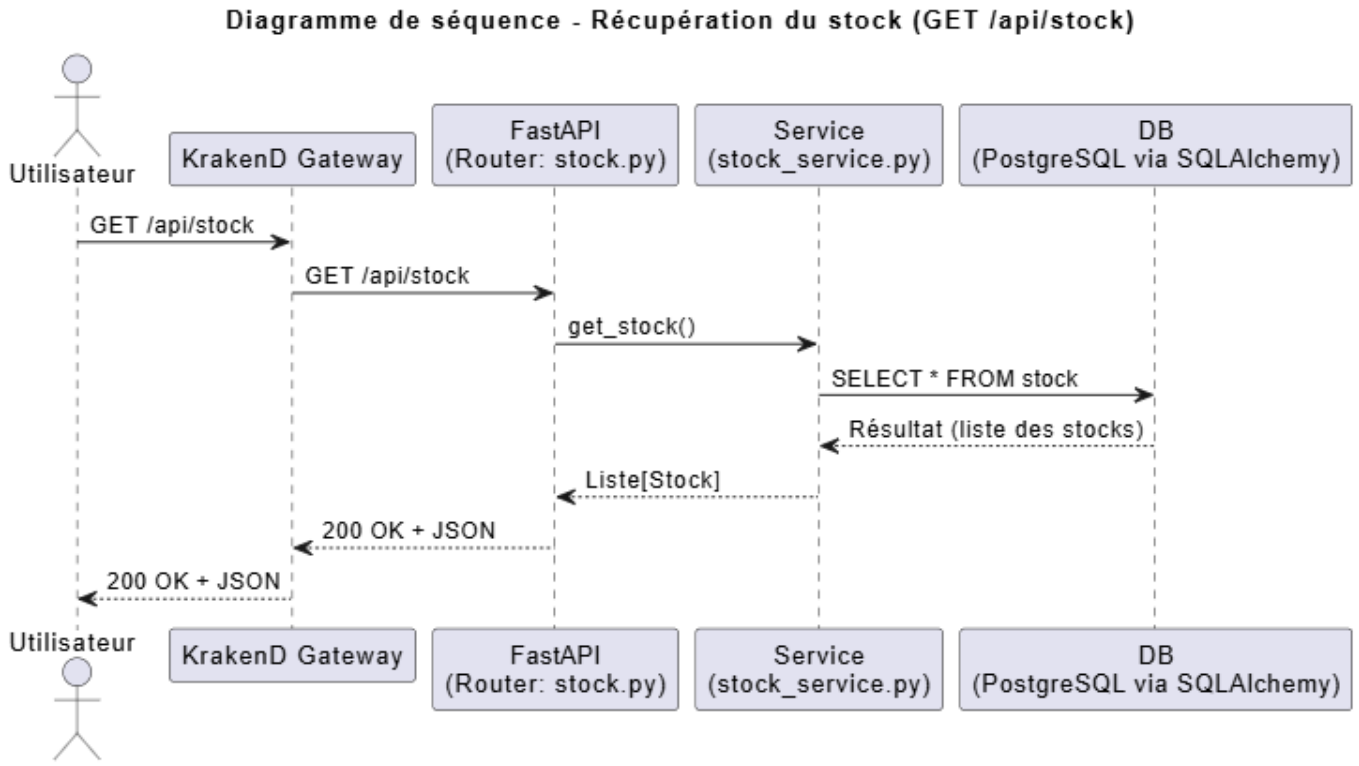
Services principaux

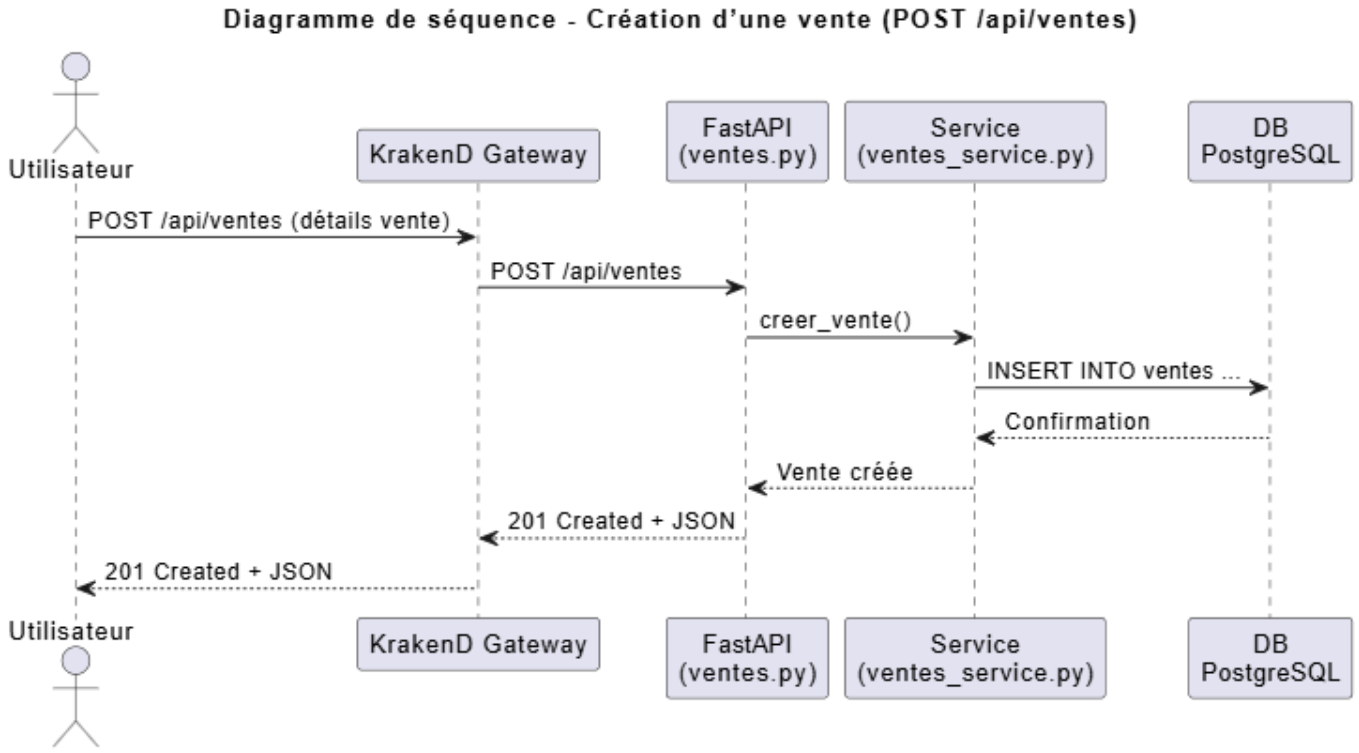
- `produits-service` : CRUD Produits

- `magasin-service` : CRUD Magasins
- `stock-service` : consultation/modification du stock
- `panier-service` : gestion du panier utilisateur
- `checkout-service` : validation de commande
- `ventes-service` : enregistrement des ventes





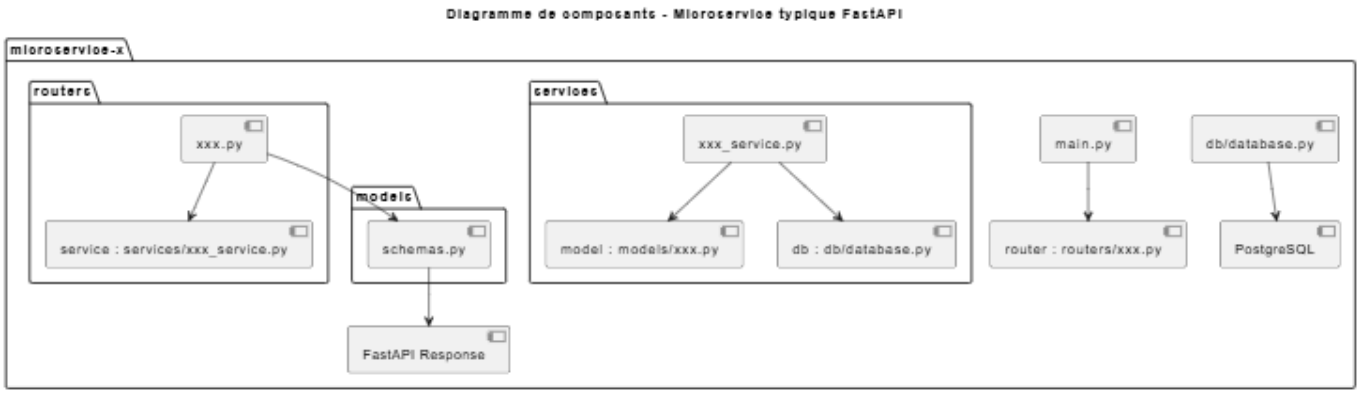




7. Architecture des composants (Vue d'implémentation)

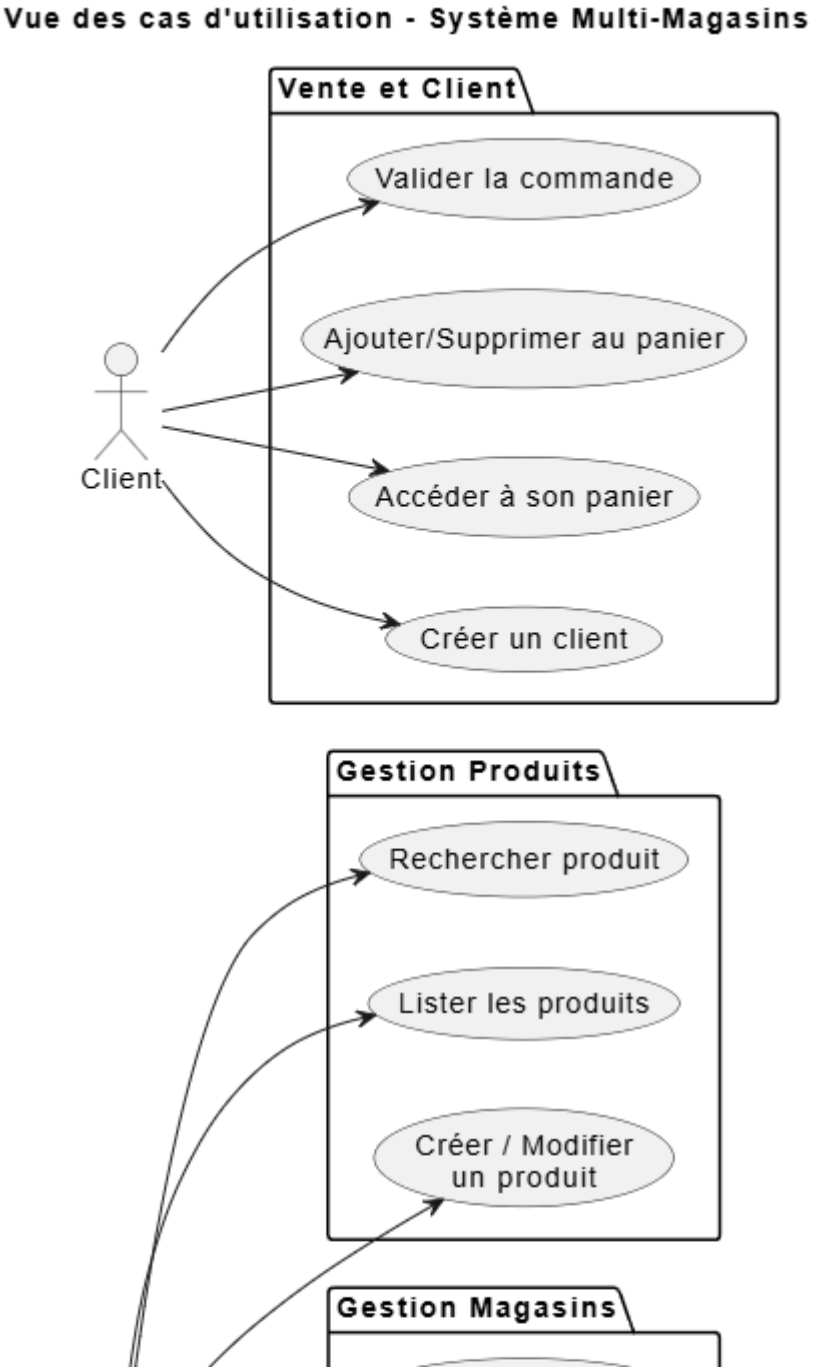
Chaque microservice est structuré selon une architecture **3-tiers** :

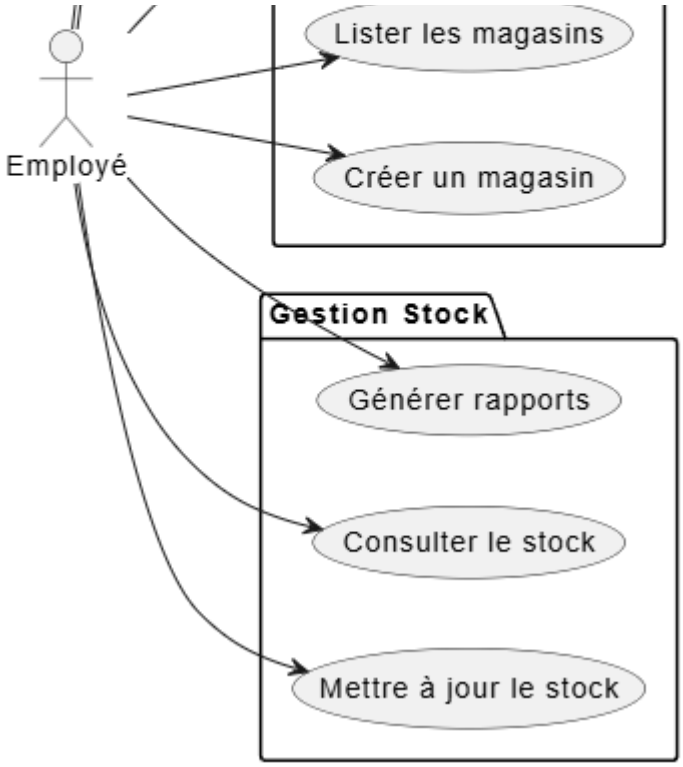
- `routers/` (FastAPI endpoints)
- `services/` (logique métier)
- `dao/` ou `repositories/` (accès BD via SQLAlchemy)
- `models/` (ORM + Pydantic)



8. Vue des processus (Vue dynamique)

Les interactions entre services sont basées sur des appels HTTP internes via leur nom DNS (ex: `http://produits-service:8000`). Les requêtes passent d'abord par KrakenD, qui applique le routing et le quota.





---

9. Observabilité & Load Balancing

- `stock-service` est dupliqué (`stock-service-1`, `stock-service-2`)
- `nginx` répartit la charge entre les deux via `upstream`
- `prometheus_fastapi_instrumentator` expose les métriques
- `Grafana` permet d'analyser : temps de réponse, charge, erreurs

---

10. Sécurité & throttling (KrakenD)

KrakenD applique une politique CORS et un **throttling** global :

- `max_rate`: 5 requêtes/s
- `client_max_rate`: 2 requêtes/s par client

Implémenté via `github_com/devopsfaith/krakend/ratelimit`

---

11. Architectural Decision Records (ADR)

ADR-001: Choix de l'architecture microservices

Statut

Acceptée – 2025-07-16

Contexte

Le système initial, construit dans les laboratoires précédents, suivait une architecture 3-tiers avec un backend FastAPI monolithique. Cependant, la complexité du domaine e-commerce (produits, stock, ventes, clients,

panier, validation) justifie une séparation claire des responsabilités. Le besoin de scalabilité, de répartition de charge, et de déploiement indépendant renforce cette nécessité.

Décision

Nous avons choisi de migrer vers une **architecture microservices**, en découpant l'application en services indépendants :

- `produits-service`
- `stock-service` (x2, derrière un NGINX load balancer)
- `ventes-service`
- `magasin-service`
- `client-service`
- `panier-service`
- `checkout-service`
- Chaque service possède sa propre base de données PostgreSQL.

Une **API Gateway** (KrakenD) centralise les appels et applique du throttling, des logs et des règles de routage.

Conséquences

☑ Positives

- Meilleure séparation des responsabilités métier
- Déploiement indépendant de chaque service
- Scalabilité horizontale facilitée
- Facilité d'intégration avec des observateurs (Prometheus/Grafana)

❌ Négatives

- Complexité accrue : orchestration Docker, configurations, surveillance
- Montée en charge de la CI/CD
- Cohésion entre services à maintenir manuellement (pas de service discovery automatisé)

Alternatives envisagées

- Garder un monolithe FastAPI mais avec modularisation stricte
- Utiliser une architecture hexagonale au lieu de microservices

ADR-002: Choix de KrakenD comme API Gateway

Statut

Acceptée – 2025-07-16

Contexte

Une fois le système découpé en microservices, un besoin de centraliser les appels frontaux est apparu : pour uniformiser les URL, gérer le CORS, appliquer des quotas, et simplifier le point d'entrée du client.

Décision

Nous avons choisi **KrakenD** comme API Gateway pour son approche déclarative, sa légèreté, et sa capacité à gérer :

- les règles de routage vers les services
- le CORS, les quotas, et le throttling
- des logs structurés
- une configuration JSON simple à intégrer dans notre `docker-compose`

Conséquences

☑ Positives

- Déploiement rapide et sans code (config JSON uniquement)
- Intégration naturelle avec Docker
- Compatible avec Prometheus et Grafana
- Bon support du load balancing (directement ou via NGINX intermédiaire)

❌ Négatives

- Moins dynamique qu'une gateway comme Kong ou Traefik (pas de service discovery)
- Gestion manuelle des chemins JSON pour chaque endpoint
- Pas de support avancé OAuth/JWT sans plugins additionnels

Alternatives envisagées

- Utiliser Traefik (plus flexible mais plus complexe à configurer)
- Développer une API gateway maison avec FastAPI (trop long pour le contexte du labo)

---

12. Difficultés rencontrées

- 🔌 Problèmes de DNS Docker pour certains services (`no such host`)
- ⚠ Erreurs de timeout `context deadline exceeded` corrigées via `depends_on` + `timeout` ajusté
- 🔀 Problèmes de redirection NGINX vs KrakenD (ports, host, etc.)

---

13. Conclusion

Ce laboratoire a marqué une étape décisive dans l'évolution de notre système vers une architecture logicielle moderne, modulaire et robuste. En migrant d'un système 3-tiers vers une architecture microservices, nous avons pu :Découpler les responsabilités métier (produits, stock, ventes, clients, panier, checkout, etc.),Isoler les bases de données par domaine, favorisant la cohérence et la résilience, Mettre en place une gateway API centralisée (KrakenD), facilitant l'uniformisation des appels clients, Introduire la répartition de charge (load balancing) avec NGINX pour le service critique de stock, Configurer une base pour l'observabilité via Prometheus et Grafana.

Difficultés rencontrées

Plusieurs défis ont été surmontés tout au long de l'implémentation :

- La duplication de services pour le load balancing a nécessité une attention particulière sur la configuration réseau (bridge, alias, ports), le docker-compose et le proxy NGINX.
- L'intégration de KrakenD a nécessité de comprendre sa syntaxe JSON, les règles de routage, le CORS, le logging, et l'ajout d'un quota (throttling).
- La cohésion interservices, via httpx en Python, a exigé un design propre et résilient, surtout pour le microservice checkout-service qui interagit avec plusieurs services en cascade.
- La structure de projet et les chemins relatifs entre services, configurations et volumes a parfois causé des erreurs subtiles mais critiques (ex: mauvais volume monté ou chemin nginx erroné).
- Le versioning des dépendances (ex: pydantic v2) a causé des conflits, comme le changement de orm_mode vers from_attributes, nécessitant une veille attentive.

Recommandations architecturales

Basé sur notre expérience :

- Factoriser certains comportements transversaux (logs, erreurs, cache) dans des middlewares ou bibliothèques partagées entre microservices.
- Ajouter un service de discovery ou registry si le nombre de microservices continue à croître.
- Renforcer l'observabilité : exporter plus de métriques métier dans Prometheus, créer des alertes pertinentes.
- Prévoir l'usage de circuit breakers ou de timeouts explicites pour éviter les effets en cascade en cas de panne.
- Envisager une authentification centralisée (OAuth2/JWT) au niveau de la gateway pour un contrôle d'accès plus fin.

Bilan

Au final, cette migration nous a permis de mieux comprendre les bénéfices concrets de l'approche microservices :

- scalabilité, indépendance des déploiements, robustesse, testabilité. Malgré une charge de configuration initiale élevée, les gains à long terme justifient pleinement cette approche.

Ce projet a aussi permis de se familiariser avec des outils modernes de production comme Docker, KrakenD, Prometheus, NGINX, FastAPI, et l'écriture de tests intégrés dans un contexte distribué.

Nous avons construit non seulement un système fonctionnel, mais également une base solide pour des évolutions futures réalistes dans un contexte e-commerce professionnel.