Store Manager L08 - Documentation d'Architecture

Ce document, basé sur le modèle arc42, décrit une API REST de gestion de magasin avec capacités GraphQL pour le Labo 08, LOG430.

1. Introduction et Objectifs

Panorama des exigences

L'application « Store Manager » est un système avec architecture microservices pour la gestion des utilisateurs, articles, commandes et stock dans un petit magasin. Elle sert de projet éducatif pour démontrer :

- L'implémentation d'une architecture API REST avec Flask
- L'intégration GraphQL pour des requêtes de données flexibles
- La fonctionnalité de gestion de stock avec cache Redis
- Le support multi-bases de données (MySQL et Redis)
- La comparaison des approches REST vs GraphQL
- L'optimisation des jointures SQL avec SQLAlchemy
- L'utilisation d'un API Gateway (KrakenD) pour le contrôle de timeout et rate limiting
- L'utilisation d'un broker Kafka pour la communication asynchrone avec des microservices
- La transition vers une architecture microservices avec séparation des responsabilités
- L'implémentation de Sagas chorégraphiées pour la coordination distribuée des opérations
- L'application du patron CQRS (Command Query Responsibility Segregation)
- L'utilisation du patron Outbox pour assurer la tolérance aux pannes en architecture event-driven

Nous ferons évoluer ce projet tout au long du cours LOG430, en intégrant de nouvelles fonctionnalités et en faisant évoluer notre architecture pour répondre aux nouvelles exigences. La progression va d'une architecture monolithique vers une architecture microservices avec saga chorégraphiée event-driven.

Objectifs qualité

Priorité	Objectif qualité	Scénario
1	**Extensibilité**	Ajout facile de nouveaux endpoints API et clients grâce aux principes REST
2	**Flexibilité**	Support GraphQL permet aux clients de requêter exactement les données nécessaires
3	**Performance**	Cache Redis pour les données de stock et optimisation des jointures SQL
4	**Maintenabilité**	Séparation claire des responsabilités via les patrons MVC+CQRS et architecture microservices
5	**Fiabilité**	Rate limiting et timeout via KrakenD pour protéger les services backend ; Saga chorégraphiée avec Outbox Pattern pour garantir la cohérence des transactions distribuées

Priorité	Objectif qualité	Scénario
6	**Scalabilité**	Architecture microservices permet la mise à l'échelle indépendante des services ; Kafka permet une communication asynchrone découplée
7	**Tolérance aux pannes**	Patron Outbox et architecture event-driven permettent la récupération des opérations après redémarrage des services

Parties prenantes (Stakeholders)

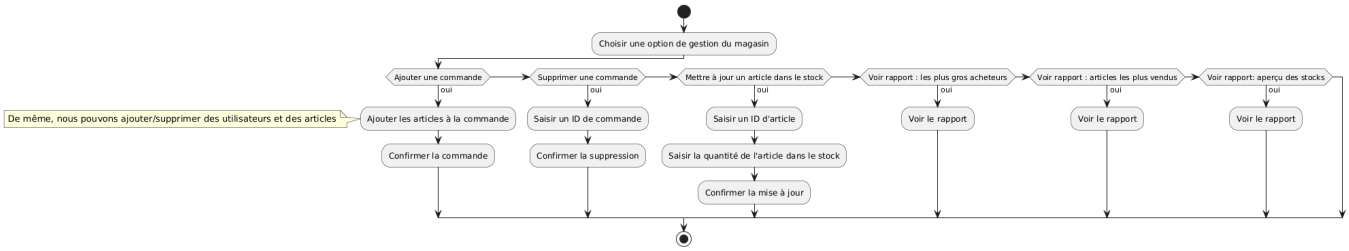
- **Développeuses et développeurs** : Apprendre/enseigner l'architecture API REST, GraphQL, microservices et les patrons de services web modernes
- **Employées et employés du magasin** : Utilisatrices et utilisateurs gérant les articles, commandes et stock via l'API
- **Clientes et clients du magasin** : Utilisatrices et utilisateurs finales servis par l'application (indirectement via les interactions avec les employé·es)
- **Fournisseurs** : Partenaires externes utilisant l'endpoint GraphQL pour vérifier l'état du stock et envoyer des données de réapprovisionnement

2. Contraintes d'architecture

Contrainte	Description
Technologie	Utilisation de Python 3, Flask, MySQL, Redis, KrakenD, Kafka et Docker
Déploiement	Déploiement en conteneurs Docker avec communication réseau entre services via un réseau Docker partagé
Communication	Communication asynchrone entre microservices via Kafka comme event broker ; communication synchrone uniquement via Outbox Pattern pour les opérations critiques
Éducatif	L'application doit clairement démontrer les principes REST vs GraphQL, la gestion de stock, les microservices, les Sagas chorégraphiées, CQRS et les bonnes pratiques API
Conception API	Doit suivre les principes RESTful et fournir une alternative GraphQL pour les requêtes flexibles
API Gateway	KrakenD comme point d'entrée unique pour tous les appels API avec contrôle de rate limiting et timeout
Tolérance aux pannes	Patron Outbox pour assurer la cohérence des transactions dans une architecture event-driven distribuée

3. Portée et contexte du système

Contexte métier



Le système permet aux employé·es du magasin de :

- Gérer les comptes utilisateurs (employé·es et clientes)
- Gérer les articles vendus par le magasin
- Traiter les commandes des clientes
- Suivre et gérer les niveaux de stock des articles
- Générer des rapports de stock
- Permettre aux fournisseurs de consulter l'état du stock via GraphQL

Contexte technique

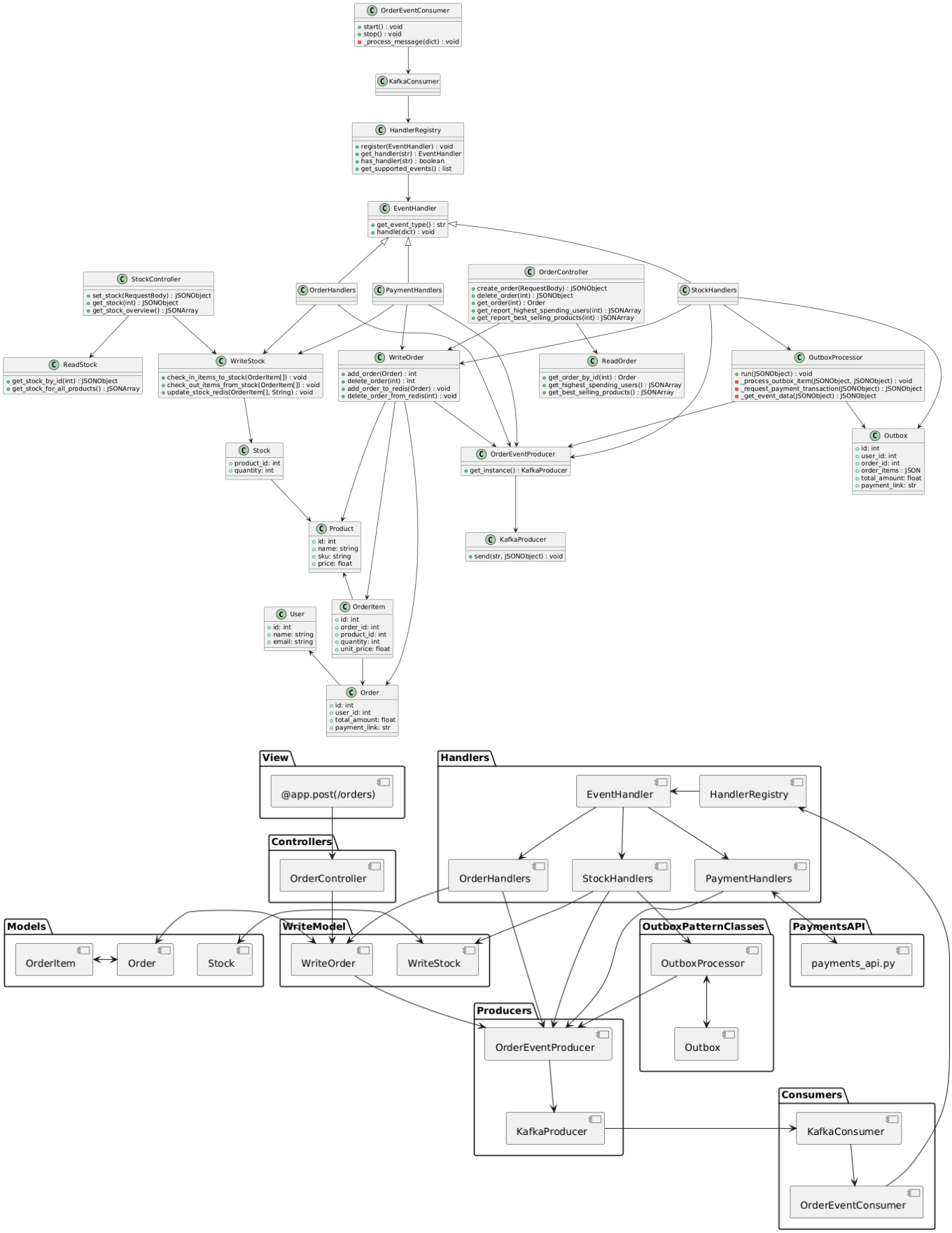
- **Applications clientes** : Postman, applications fournisseurs, potentiels frontends web/mobiles
- **Couche API** : API REST Flask avec endpoint GraphQL
- **Couche base de données** : Backend MySQL avec cache Redis
- **API Gateway** : KrakenD pour le routage, rate limiting et timeout
- **Communication** : Requêtes HTTP/HTTPS entre clients, gateway et microservices

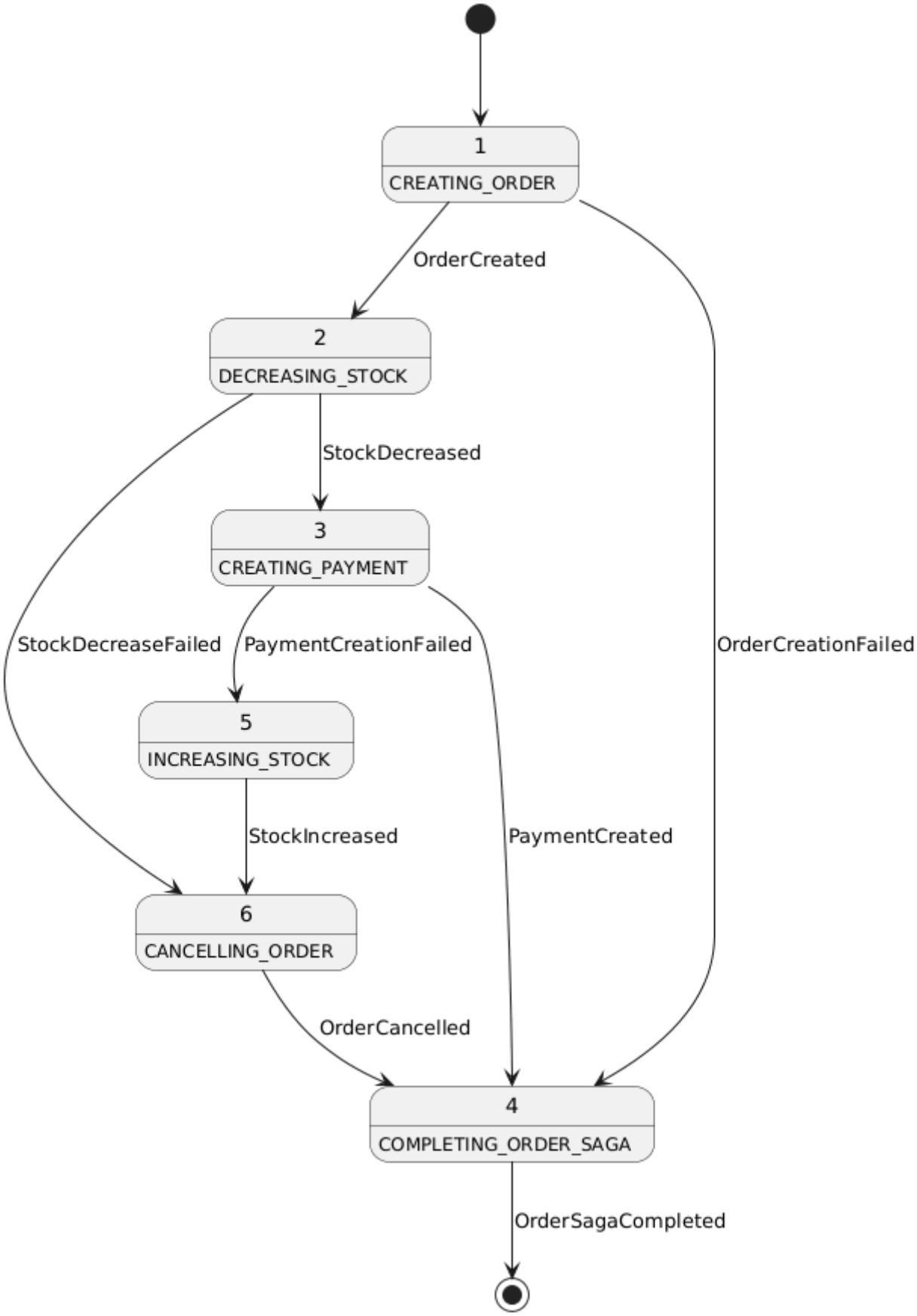
4. Stratégie de solution

Problème	Approche de solution
Standardisation API	Principes REST avec méthodes HTTP et nommage de ressources appropriés
Requêtes de données flexibles	Endpoint GraphQL pour les clients nécessitant des champs de données spécifiques
Gestion de stock	Mises à jour automatiques du stock lors de création/suppression de commandes
Performance	Cache Redis pour les données de stock fréquemment accédées
Support multi-BD	MySQL pour l'écriture, Redis pour la lecture (rapports)
Protection des services	KrakenD API Gateway pour rate limiting et timeout
Séparation des responsabilités	Microservice dédié pour la gestion des commandes ; Kafka comme event broker
Scalabilité	Architecture microservices permettant la mise à l'échelle indépendante
Coordination distribuée	Saga chorégraphiée : chaque service réagit aux événements Kafka sans orchestrateur central

Problème	Approche de solution
Tolérance aux pannes	Patron Outbox pour persister les événements avant envoi à Kafka ; OutboxProcessor pour relancer les opérations après redémarrage
Cohérence transactionnelle	CQRS pour séparer les opérations d'écriture (Commands) des opérations de lecture (Queries)

5. Vue des blocs de construction


OrderEventConsumer
+ start() : void
+ stop() : void
- _process_message(dict) : void
KafkaConsumer
HandlerRegistry
+ register(EventHandler) : void
+ get_handler(str) : EventHandler
+ has_handler(str) : boolean
+ get_supported_events() : list
EventHandler
+ get_event_type() : str
+ handle(dict) : void
StockController
+ set_stock(RequestBody) : JSONObject
+ get_stock(int) : JSONObject
+ get_stock_overview() : JSONArray
OrderHandlers
PaymentHandlers
OrderController
+ create_order(RequestBody) : JSONObject
+ delete_order(int) : JSONObject
+ get_order(int) : Order
+ get_report_highest_spending_users(int) : JSONArray
+ get_report_best_selling_products(int) : JSONArray
StockHandlers
ReadStock
+ get_stock_by_id(int) : JSONObject
+ get_stock_for_all_products() : JSONArray
WriteStock
+ check_in_items_to_stock(OrderItem[]) : void
+ check_out_items_from_stock(OrderItem[]) : void
+ update_stock_redis(OrderItem[], String) : void
WriteOrder
+ add_order(Order) : int
+ delete_order(int) : int
+ add_order_to_redis(Order) : void
+ delete_order_from_redis(int) : void
ReadOrder
+ get_order_by_id(int) : Order
+ get_highest_spending_users() : JSONArray
+ get_best_selling_products() : JSONArray
OutboxProcessor
+ run(JSONObject) : void
- _process_outbox_item(JSONObject, JSONObject) : void
- _request_payment_transaction(JSONObject) : JSONObject
- _get_event_data(JSONObject) : JSONObject
Outbox
+ id: int
+ user_id: int
+ order_id: int
+ order_items : JSON
+ total_amount: float
+ payment_link: str
Stock
+ product_id: int
+ quantity: int
OrderEventProducer
+ get_instance() : KafkaProducer
Product
+ id: int
+ name: string
+ sku: string
+ price: float
KafkaProducer
+ send(str, JSONObject) : void
User
+ id: int
+ name: string
+ email: string
OrderItem
+ id: int
+ order_id: int
+ product_id: int
+ quantity: int
+ unit_price: float
Order
+ id: int
+ user_id: int
+ total_amount: float
+ payment_link: str
View
@app.post(/orders)
Handlers
EventHandler
HandlerRegistry
Controllers
OrderController
OrderHandlers
StockHandlers
PaymentHandlers
Models
OrderItem
Order
Stock
WriteModel
WriteOrder
WriteStock
OutboxPatternClasses
OutboxProcessor
Outbox
PaymentsAPI
payments_api.py
Producers
OrderEventProducer
KafkaProducer
Consumers
KafkaConsumer
OrderEventConsumer

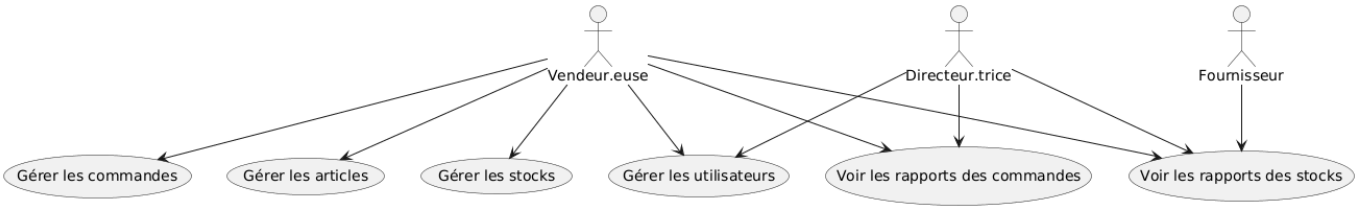


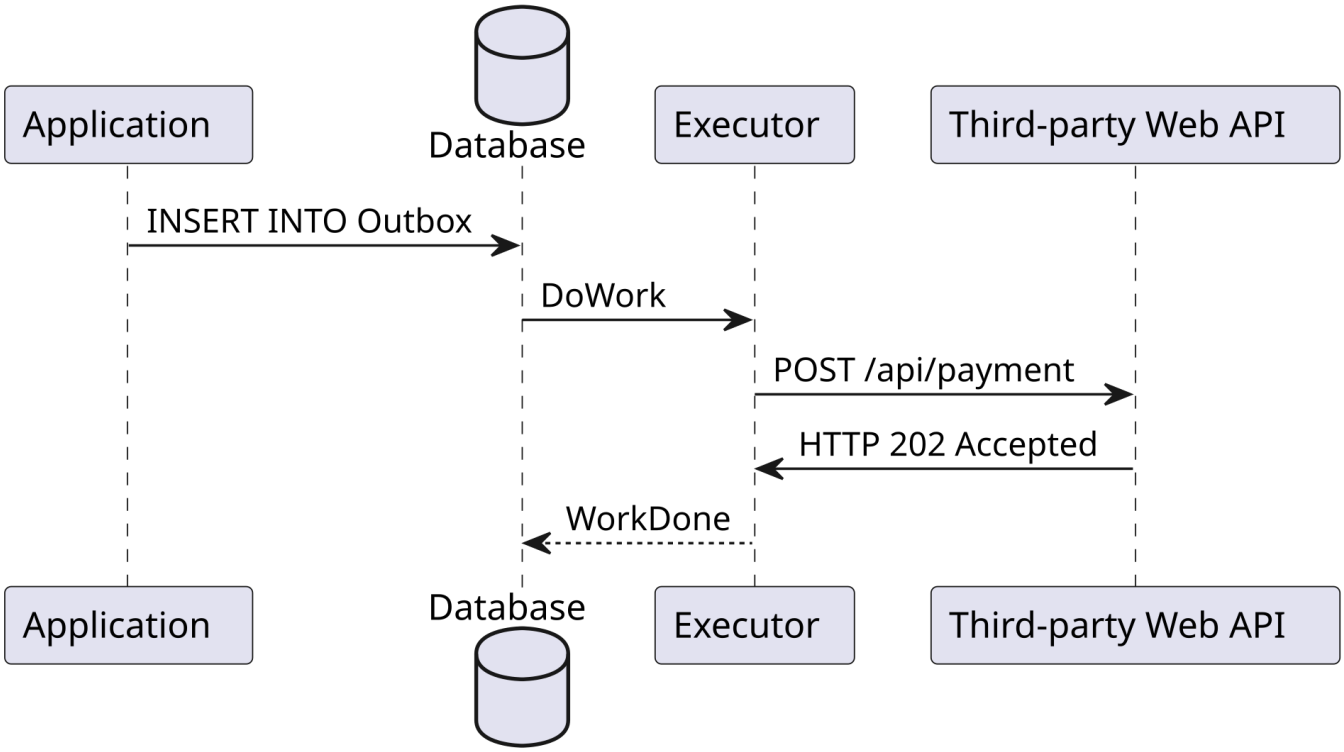
Composants clés :

- **Contrôleurs API** : Gèrent les requêtes et réponses HTTP
- **Couche logique métier** : Traitement des commandes, gestion de stock
- **Models avec SQLAlchemy** : Abstraction de l'accès aux bases de données
- **Cache Redis** : Mise en cache des données de stock pour la performance

- **Schéma GraphQL** : Interface de requête flexible pour les fournisseurs
- **Kafka Producer** : Producteur d'événements qui envoie les messages à Kafka pour notification des microservices
- **Kafka Consumer** : Consommateur d'événements qui réagit aux messages publiés sur les topics Kafka
- **Event Handlers** : Réagissent aux événements Kafka et exécutent les opérations métier ou les compensations
- **Outbox Table** : Stocke les événements à envoyer pour assurer la tolérance aux pannes
- **OutboxProcessor** : Lit la table Outbox et relance l'envoi des événements après redémarrage
- **HandlerRegistry** : Registre centralisé mappant les types d'événements à leurs handlers correspondants

6. Vue d'exécution



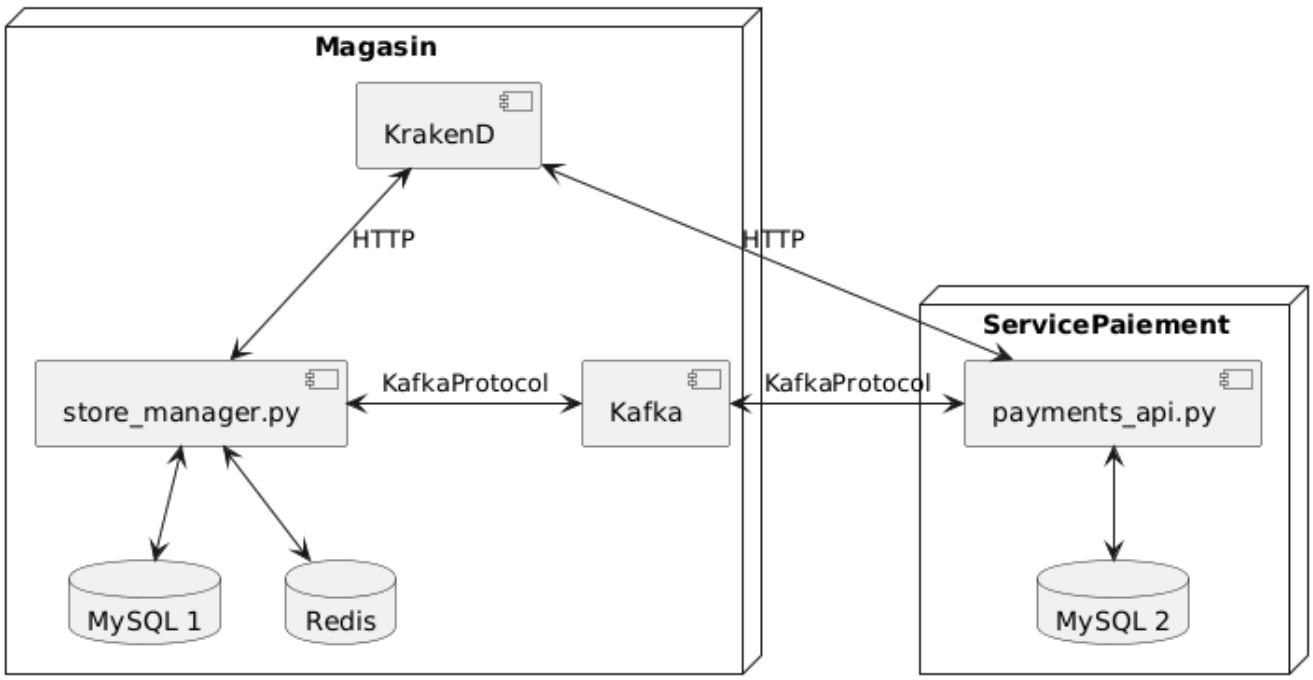


Scénarios clés :

1. **Traitement des commandes** : Saga chorégraphiée où chaque microservice réagit de manière autonome aux événements Kafka
2. **Compensation automatique** : En cas d'erreur, les handlers déclenchent les événements de compensation (ex: StockIncreased)
3. **Rapports de stock** : Génération de rapports de stock complets avec détails des articles
4. **Requêtes GraphQL** : Fournisseurs interrogent des champs de données de stock spécifiques
5. **Réapprovisionnement de stock** : Ajout de quantités de stock aux articles existants

6. **Récupération après panne** : OutboxProcessor redémarre les opérations non complétées après redémarrage du service

7. Vue de déploiement



Architecture conteneurs :

- **Conteneur API et bases de données** : Application Flask, MySQL, Redis.
- **Conteneur Payment Service** : API Flask dédiée aux paiements
- **KrakenD** : API Gateway
- **Réseau Docker** : Permet la communication entre conteneurs

8. Concepts transversaux

- **Principes REST** : Client-serveur, sans état, interface uniforme, système en couches
- **GraphQL** : Langage de requête pour la récupération flexible de données
- **Architecture Event-Driven** : Microservices communiquent via des événements asynchrones à travers Kafka
- **Saga Chorégraphiée** : Coordination distribuée sans orchestrateur central ; chaque service réagit aux événements et déclenche les événements suivants
- **CQRS (Command Query Responsibility Segregation)** : Séparation des opérations d'écriture (Commands) et de lecture (Queries)
- **Outbox Pattern** : Garantit la durabilité des événements en les stockant dans une table locale avant envoi à Kafka
- **Stratégie de cache** : Redis pour l'optimisation de la lecture des données de stock
- **Patrons de base de données** : ORM avec SQLAlchemy, gestion de sessions
- **Communication conteneurs** : Réseaux Docker pour la découverte de services
- **API Gateway Pattern** : Point d'entrée unique avec KrakenD
- **Microservices Pattern** : Séparation des responsabilités par domaine métier
- **Rate Limiting** : Protection contre les abus et surcharge

- **Circuit Breaker** : Timeout control pour éviter les blocages

9. Décisions d'architecture

Veuillez consulter le fichier `/docs/adr/adr001.md`.

10. Exigences qualité

Extensibilité

- Ajout facile de nouveaux endpoints REST suivant les patrons établis
- Le schéma GraphQL peut être étendu pour de nouvelles exigences de données
- Nouveaux event handlers peuvent être enregistrés dans HandlerRegistry sans modification du code existant

Flexibilité

- GraphQL permet aux clients de demander exactement les données dont ils ont besoin
- Support pour plusieurs backends de base de données (MySQL, Redis)
- Architecture microservices permet des changements indépendants
- Nouveaux événements peuvent être ajoutés à la saga sans modification de l'orchestration

Performance

- Cache Redis pour les données de stock réduit la charge de base de données
- Optimisation des jointures SQL pour les requêtes complexes
- Communication asynchrone via Kafka réduit le couplage entre services

Maintenabilité

- Séparation claire des préoccupations avec les patrons MVC+CQRS
- Conventions de nommage cohérentes à travers toutes les couches
- Conteneurisation Docker pour des environnements cohérents
- Microservices permettent des déploiements indépendants
- Handlers dédiés pour chaque transition d'état de la saga

Fiabilité

- Rate limiting via KrakenD protège contre les abus
- Timeout control évite les blocages
- Isolation des pannes via l'architecture microservices
- Patron Outbox garantit la persistence des événements même en cas de panne
- OutboxProcessor automatise la récupération après redémarrage

Scalabilité

- Microservices peuvent être mis à l'échelle indépendamment
- KrakenD peut gérer la distribution de charge
- Base de données et cache séparés permettent une mise à l'échelle horizontale
- Communication asynchrone permet une meilleure répartition de charge

Tolérance aux pannes

- Événements sont persistés dans la table Outbox avant envoi à Kafka
- Saga chorégraphiée peut continuer même si certains services sont temporairement indisponibles
- OutboxProcessor retraite les opérations non complétées lors du redémarrage
- Compensation automatique en cas d'erreur dans une étape de la saga

11. Risques et dettes techniques

Voici un résumé des risques et des dettes techniques liés à l'application de gestion de magasin. S'agissant d'une application pédagogique conçue pour explorer les concepts architecturaux, nous assumons ces risques, qui n'ont pas besoin d'être abordés lors des travaux pratiques.

Risque	Impact	Mitigation
Cohérence du cache	Les données de stock dans Redis peuvent devenir incohérentes avec la base de données	Implémenter des stratégies appropriées d'invalidation de cache
Securité des informations de stock	Aucun de nos endpoints d'API n'est authentifié. Si cela peut ne pas poser de problème majeur si l'application est utilisée uniquement au sein du réseau d'un seul magasin, cela devient problématique dès que nous ouvrons certains endpoints, comme le point de terminaison GraphQL, à des utilisateurs externes.	Créez un système d'authentification pour le point de terminaison GraphQL, ou pour tous les endpoints pour plus de sécurité
Cohérence des stocks par rapport à la réalité	Dans l'implémentation actuelle, il n'existe aucun contrôle des stocks maximum/minimum. Les stocks peuvent être négatifs, par exemple.	Implémenter des vérifications de limites de stock
Défaillance du event broker	Si l'event broker Kafka cesse de fonctionner, la communication entre les services et l'exécution des sagas sera immédiatement compromise.	Utiliser un cluster Kafka avec plusieurs instances du broker, surveiller les instances et leur fournir les ressources informatiques nécessaires pour qu'elles fonctionnent toujours avec des performances élevées.
Cohérence des commandes en cas d'échec à un point spécifique de la saga	Si le Store Manager s'arrête avant la création de l'enregistrement dans la table Outbox, la commande restera dans la base de données sans un payment_id. À moins que nous supprimions ou modifiions manuellement la commande, elle ne sera pas mise à jour.	Inclure les informations de la commande dans la table Outbox immédiatement après la création de la commande.

12. Glossaire

Terme	Définition
API	Application Programming Interface : interface de programmation d'applications
CQRS	Command Query Responsibility Segregation : patron séparant les opérations d'écriture (Commands) des opérations de lecture (Queries)
GraphQL	Langage de requête pour API permettant aux clients de demander des données spécifiques
REST	Representational State Transfer : style architectural pour les services web
RESTful	APIs qui adhèrent aux principes REST