Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de Projet

Étudiants	Look Gabriel
Cours	LOG430
Session	Automne 2025
Groupe	02
Projet	Phase 1
Chargé de laboratoire	
Date	28 Septembre 2025

Document Arc42

1. Introduction et Objectifs

Panorama des exigences

L'application "BrokerX" est une plateforme de courtage en ligne pour investisseurs particuliers. Phase 1 : prototype/POC monolithique robuste mais conçu pour évoluer vers microservices / architecture événementielle.

Objectifs du document

- Documenter architecture robuste (production-ready roadmap).
- Fournir 4+1 views et Arc42.
- Livrer ADRs, UC Must textuels, artifacts pour démarrer le POC.

Parties Prenantes (Stakeholders)

- Clients : utilisateurs via interface web/mobile.
- Opérations Back-Office : gestion des règlements, supervision.
- Conformité / Risque : surveillance pré- et post-trade.
- Fournisseurs de données de marché : cotations en temps réel.
- Bourses externes : simulateurs de marché pour routage d'ordres.

2. Contraintes d'architecture

Contraintes techniques & réglementaires

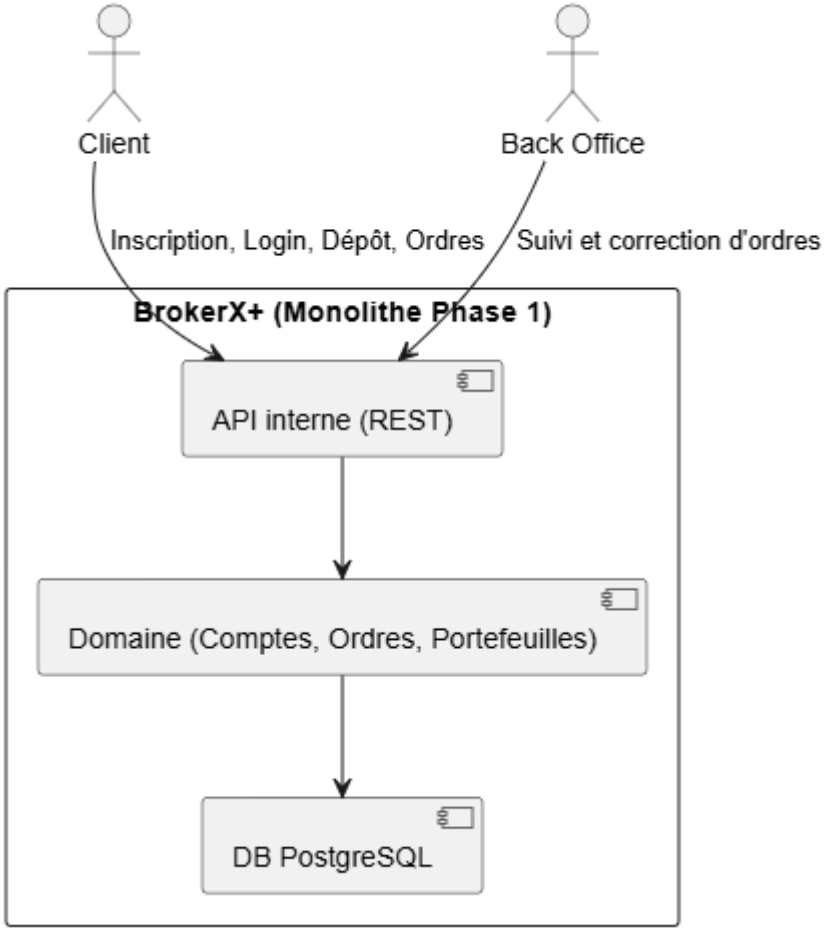
- Langages privilégiés : Java (Spring Boot) — choisi pour faciliter POC et solide écosystème.
- Technologie: Utilisation de Java, SpringBoot, PostgreSQL, Grafana, Prometheus
- Déploiement Cible: Conteneurs Docker, base PostgreSQL
- CI/CD: Github Workflow/Actions, reproductible
- Journal d'audit immuable requis.
- Idempotence des requêtes mutatives.
- Observabilité minimale (logs, metrics); traces pour phase ultérieure.

Exigences non-fonctionnelles

- Latence P95 ordre→ACK ≤ 500 ms (monolithe).

- Throughput ≥ 300 ordres/s (phase-1 cible).
- Disponibilité ~90% (phase-1).
- Audit immuable, exactly-/effectively-once pour règlements.

3. Portée et contexte du système



Contexte métier

Le système BrokerX+ est une plateforme de courtage en ligne qui permet aux investisseurs particuliers d'interagir avec un moteur d'ordres simplifié.
Le système permet aux utilisateurs :

- De s'inscrire et activer un compte afin d'accéder à la plateforme.
- De s'authentifier et maintenir une session sécurisée.
- D'alimenter leur portefeuille par des dépôts virtuels.
- De placer des ordres d'achat ou de vente (marché ou limite).
- De consulter l'état de leurs ordres et de leurs positions.
- Le système permet au Back Office (employés internes) :
    - Suivre les ordres placés par les clients.

    - De vérifier et corriger certains cas particuliers (ex. annulation d'ordres ou ajustements).

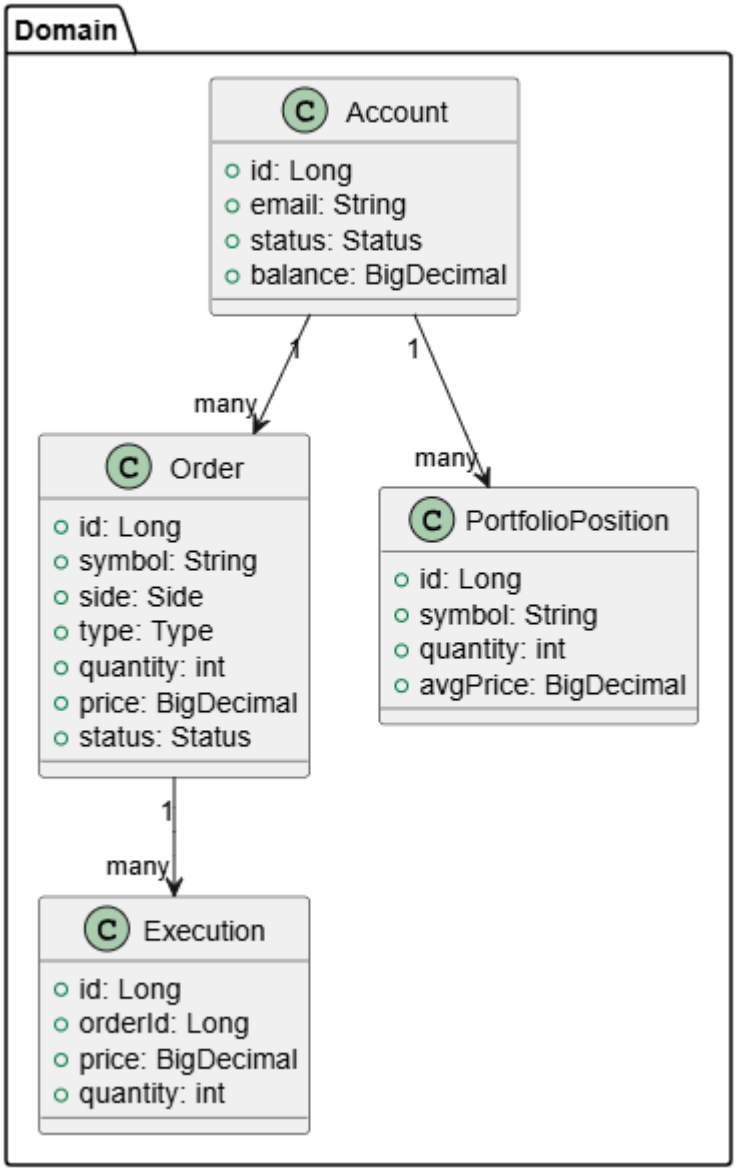
Contexte technique

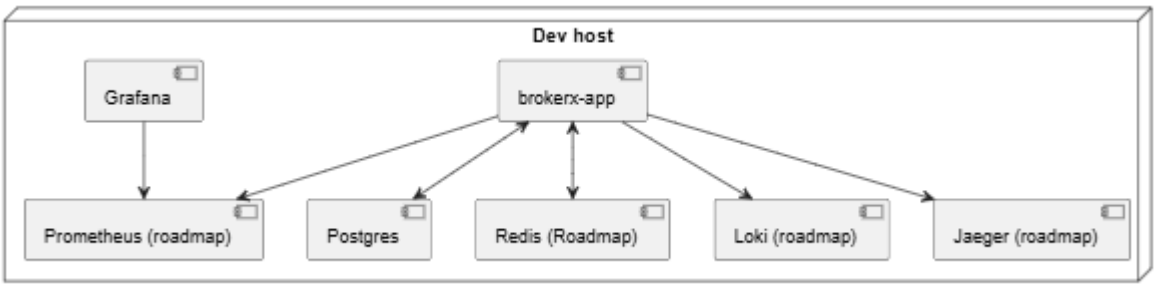
- Interface : Application Web Java (Spring Boot) exposant des endpoints REST internes.
- Architecture : Monolithe en couches, inspiré de l'architecture hexagonale (séparation domaine / application / adapters).
- Concepts DDD : Définition d'agrégats (Account, Order, Portfolio Position), usage de Value Objects et Repositories pour séparer logique métier et persistance.
- Persistance : Base relationnelle PostgreSQL gérée via JPA/Hibernate.
- Transactions : gestion ACID pour garantir la cohérence des ordres et des soldes.
- Observability minimale : Spring Actuator (healthcheck, logs JSON).

4. Stratégie de solution

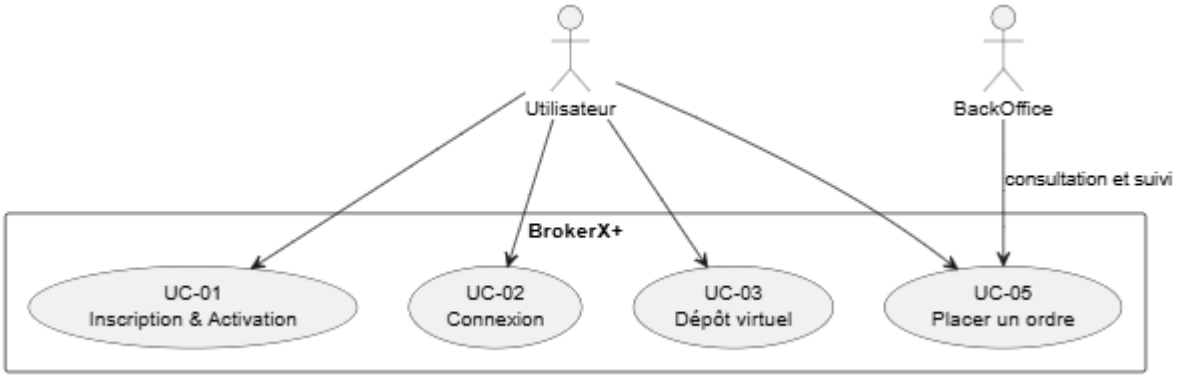
- **Architecture hexagonale** (domaine indépendant des frameworks).
- **Postgres** pour cohérence transactionnelle.
- **Kafka (roadmap)** pour audit immuable et scalabilité des consommateurs.
- **Redis (roadmap)** pour cache et réduction de latence sur lectures fréquentes.
- **Observability** : Prometheus + Grafana (metrics), Loki (logs), Jaeger (traces).
- **Sécurité** : Auth/MFA, RBAC, chiffrement repos/transit.

5. Vue des blocs de construction

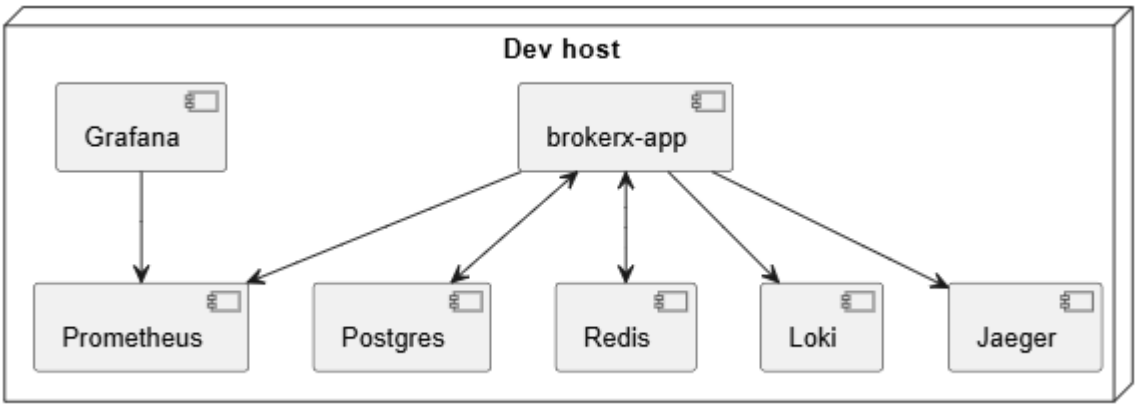




6. Vue d'exécution



7. Vue de déploiement



8. Concepts transverseaux

- **Sécurité** : MFA, RBAC.
- **Audit** : table audit_events + Kafka (future).
- **Idempotence** : Idempotency-Key obligatoire pour opérations critiques.
- **Gestion erreurs** : exceptions transformées en codes REST + logging structuré.
- **Observabilité** : metrics exposées via /actuator/prometheus.
- **Tests qualité** : ArchUnit pour règles d’archi, JUnit pour domain, Gatling/k6 pour perfs.

9. Décisions d’architecture

Veuillez consulter les ADR dans la section “ADR” plus bas

10. Exigences qualité

- **Performance** : 300 ordres/s, P95 latence <500ms.
- **Disponibilité** : 90% phase-1, HA roadmap.
- **Sécurité** : MFA, logs d’audit, RBAC, chiffrement TLS.
- **Observabilité** : logs structurés, metrics Prometheus, traces Jaeger.
- **Évolutivité** : Kafka pour scaling.

11. Risques et dettes techniques

12. Glossaire

Terme	Définition
UC	Use Case
Kafka (roadmap)	système distribué de diffusion d’événements.
Portfolio	ensemble de positions et de liquide d’un utilisateur.
Execution Report	confirmation partielle ou complète d’un ordre.
Matching Engine	moteur d’appariement des ordres.
RBAC	Role Based Access Control.
Observability	capacité à monitorer logs, métriques, traces.
Roadmap	À être implémenté dans le futur.

UC Must

UC-01 — Inscription et Activation

L'inscription sert à créer un nouveau compte sur la plateforme. L'utilisateur entre son courriel et un mot de passe, et le système crée un compte en attente. Un jeton d'activation est généré et doit être validé par l'utilisateur pour rendre le compte actif. Sans cette étape, il ne peut pas utiliser la plateforme.

UC-02 — Authentification (Connexion)

La connexion permet à un utilisateur déjà inscrit d'accéder à son compte. Il entre son courriel et son mot de passe, et si tout est bon et que son compte est actif, le système lui donne un jeton de session. Ce jeton lui permet d'utiliser les fonctions de la plateforme de façon sécurisée.

UC-03 — Dépôt virtuel

Le dépôt virtuel permet d'ajouter de l'argent fictif dans son compte. L'utilisateur envoie un montant et le système l'ajoute à son solde. On utilise une clé spéciale pour éviter que le même dépôt soit compté deux fois.

UC-05 — Placement d’un ordre

Placer un ordre, c'est envoyer une demande d'achat ou de vente d'un titre. L'utilisateur choisit le symbole, le type d'ordre, la quantité et éventuellement un prix limite. Le système vérifie s'il a assez de fonds et enregistre l'ordre. Si une contrepartie existe, l'ordre est exécuté, sinon il reste en attente.

MoSCoW

MUST HAVE

UC-01, UC-02, UC-03, UC-05

SHOULD HAVE

UC-06, UC-08

COULD HAVE

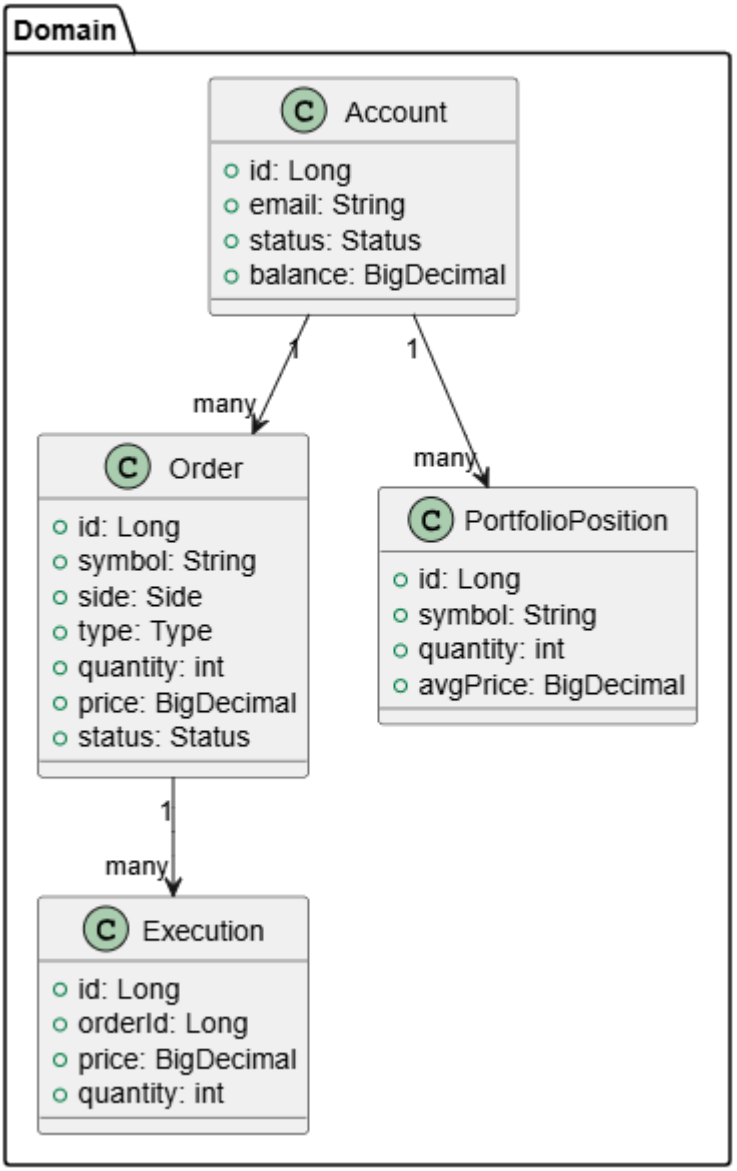
UC-07

WON’T HAVE

UC-04

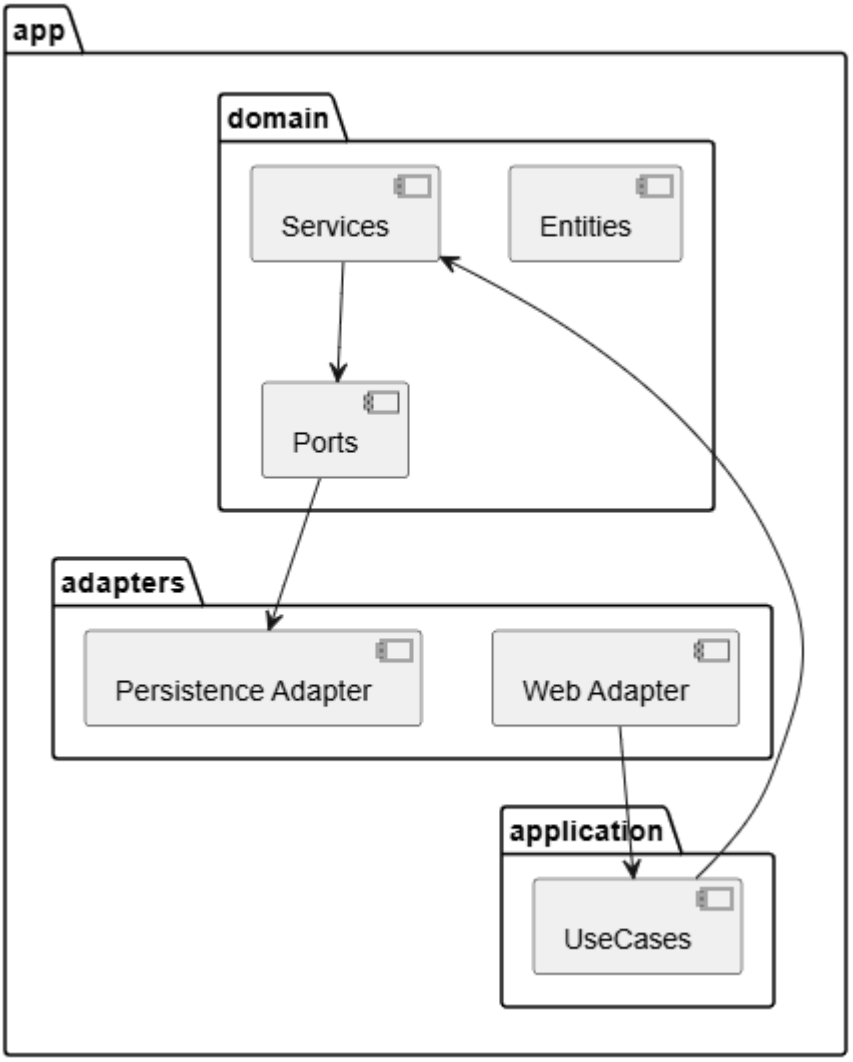
Vues 4+1

Vue Logique

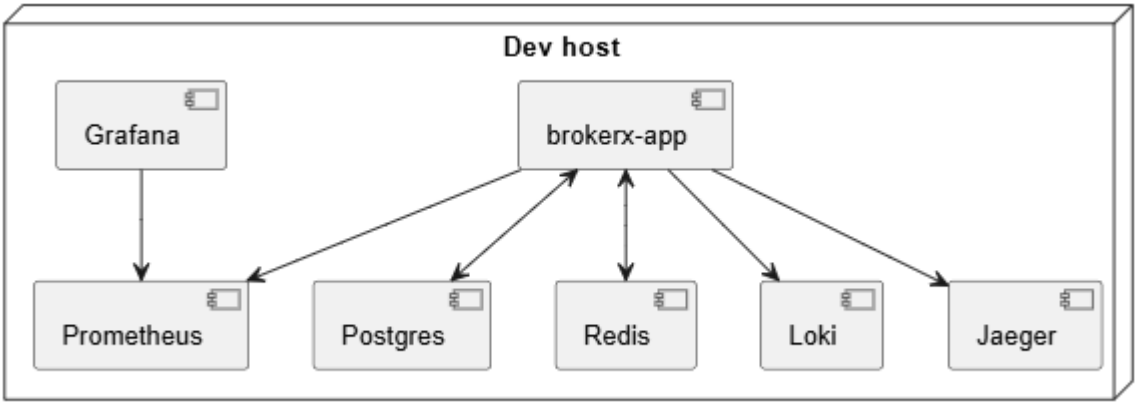


Vue des processus

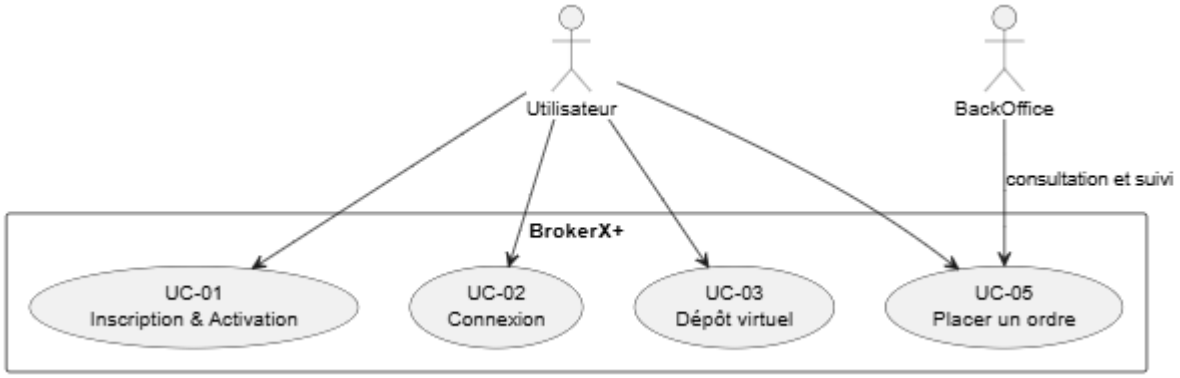
Vue de developpement



Vue de déploiement



Vue Scénario



ADR

ADR-001: Choix d'architecture — Hexagonale (Ports & Adaptateurs) vs MVC

Statut

Accepté

Contexte

BrokerX démarre par un prototype monolithique mais doit pouvoir évoluer vers une architecture microservices et event-driven. Le domaine financier exige testabilité, audit immuable, idempotence et faible couplage aux frameworks.

Décision

Adopter l'architecture Hexagonale (Ports & Adapters / Clean Architecture) comme style de référence pour la structure du code du monolithe. Les adapters (controllers, DB, external clients) implémentent des ports définis dans le domaine. Le domaine ne dépend d'aucun framework.

Justification / Rationale

- Facilite l'isolation du domaine pour tests unitaires.
- Simplifie l'extraction de composants en microservices ultérieurement.

- Permet d'éviter le "framework-lock" des entités métier.
- Le context demande une architecture suivant le DDD, Une architecture hexagonal s'aligne parfaitement avec le Domaine au centre et le ports/adaptateurs autours.
- Cette architecture permet d'extraire plus facilement certains modules (bounded contexts) vers des microservices par la suite.

Conséquences

- Code métier indépendant, meilleure longévité.
- Meilleure testabilité.
- Nécessite discipline (enforcer, arch-check).
- Un peu plus de boilerplate initial.

Plan de migration

- POC initial peut utiliser Spring Boot mais respecter la séparation des modules.
- Ajouter tests ArchUnit & Maven Enforcer pour garantir la règle.

ADR-002 — Choix de la base de données et de la persistance

Statut

Accepté

Contexte

Le système doit stocker de manière fiable des comptes, ordres et transactions. Les données doivent être cohérentes (ACID), et on veut utiliser des outils connus et stables.

Décision

Utiliser PostgreSQL comme base de données relationnelle et Spring Data JPA (Hibernate) pour l’accès aux données.

Justification / Rationale

- PostgreSQL est un SGBD open-source, robuste et mature, largement utilisé en production.
- Il respecte les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité), ce qui garantit la fiabilité des transactions.
- Spring Data JPA simplifie l'accès aux données : moins de code boilerplate, intégration native avec Spring Boot.
- La solution est bien documentée et adaptée à un projet pédagogique comme à un système évolutif.

Conséquences

- Avantages : cohérence forte, transactions fiables, intégration simple avec Spring Boot, outils de migration disponibles (Flyway).

- Inconvénients : nécessite une base relationnelle en arrière-plan (plus lourd qu'une base embarquée), tuning nécessaire pour de fortes charges.

ADR-003 — Gestion des erreurs et uniformisation

Statut

Accepté

Contexte

Les utilisateurs et développeurs doivent comprendre facilement les erreurs, et on doit éviter de renvoyer des messages techniques ou des exceptions brutes.

Décision

Utiliser le format standard **Problem JSON (RFC 7807)** pour toutes les réponses d'erreurs. Les entrées seront validées avec **Bean Validation (JSR-380)**.

Justification / Rationale

- Le format Problem JSON est un standard reconnu pour les API REST, facile à traiter côté client.
- Cela garantit une uniformité des erreurs : même structure quel que soit le type de problème.
- Cela améliore la sécurité, car on ne divulgue pas d'informations techniques sensibles.
- L'usage de Bean Validation est une bonne pratique Spring Boot, qui réduit le code manuel et centralise les règles de validation.

Conséquences

- Avantages : format uniforme, facile à traiter côté client, meilleure lisibilité et sécurité (pas de stacktrace brute exposée).
- Inconvénients : nécessite un peu plus de code pour gérer les erreurs correctement.

References / Annexes

- PostgreSQL: https://www.postgresql.org/
- Spring Boot: https://spring.io/projects/spring-boot
- DDD Evans (2003) - Domain Driven Design: https://fabiofumarola.github.io/nosql/readingMaterial/Evans03.pdf
- ADR Template: https://github.com/pmerson/ADR-template