

Table des matières

Table des matières

- Début de la documentation arc42
- 1. Introduction & Objectifs
 - 1.1 Objectifs métier
 - 1.2 Fonctionnalités essentielles
 - 1.3 Objectifs de qualité pour l'architecture
 - 1.4 Parties prenantes
 - 1.5 Vue d'ensemble des exigences fonctionnelles
 - 1.6 Priorisation MoSCoW des cas d'utilisation (DDD)
- Priorisation MoSCoW des cas d'utilisation
 - 1.7 Description détaillée des cas d'utilisation (DDD)
 - UC-01 — Inscription & Vérification d'identité
 - UC-02 — Authentification & MFA
 - UC-03 — Approvisionnement du portefeuille (dépôt virtuel)
 - UC-04 — Abonnement aux données de marché
 - UC-05 — Placement d'un ordre (marché/limite) avec contrôles pré-trade
 - UC-06 — Modification / Annulation d'un ordre
 - UC-07 — Appariement interne & Exécution (matching)
 - UC-08 — Confirmation d'exécution & Notifications
 - 2. Contraintes d'architecture
 - 2.1 Contraintes techniques
 - 2.2 Contraintes réglementaires et de conformité
 - 2.3 Contraintes organisationnelles
 - 2.4 Contraintes de gestion et maintenance
 - 3. Portée du système et contexte
 - 3.1 Contexte métier
 - 3.2 Diagramme de contexte DDD
 - 3.3 Diagramme des bounded contexts (DDD)
 - 3.4 Contexte technique
 - 4. Stratégie de solution
 - 5. Vue des blocs de construction
 - 5.1 Introduction
 - 5.2 Vue d'ensemble high level (Niveau 1)
 - 5.3 Vue interne des composants (Niveau 2)
 - 5.4 Organisation du code et conventions
 - 6. Vue d'ensemble des scénarios
 - Diagramme
 - Contexte
 - Éléments
 - Relations
 - Rationnel

- 7. Vue de déploiement
 - Contexte
 - Éléments
 - Relations
 - Rationnel
- 8. Vue Logique
 - Diagrammes de classes par microservice
 - Contexte
 - Éléments
 - Relations
 - Rationnel
- 9. Vue Processus (C&C)
 - Diagrammes
 - UC01 — Séquence
 - UC01 — Activité
 - UC02 — Séquence
 - UC02 — Activité
 - UC03 — Séquence
 - UC03 — Activité
 - UC04 — Séquence
 - UC04 — Activité
 - UC05 — Séquence
 - UC05 — Activité
 - UC06 — Séquence
 - UC06 — Activité
 - UC07 — Séquence
 - UC07 — Activité
 - UC08 — Séquence
 - UC08 — Activité
 - Contexte
 - Éléments
 - Relations
 - Rationnel
- 10. Vue Développement
 - Diagrammes de composants
 - Diagrammes de packages
 - Contexte
 - Éléments
 - Relations
 - Rationnel
- 11. Concepts transversaux
 - 11.1 Modèle de persistance par microservice
 - Auth-Service
 - Wallet-Service
 - Order-Service
 - Matching-Service
 - Market-Data-Service

- Notification-Service
 - 11.2 Diagrammes de classes par microservice
 - Auth-Service
 - Wallet-Service
 - Order-Service
 - Matching-Service
 - Market-Data-Service
 - Notification-Service
 - Modèle de domaine
 - 11.2 Persistance
 - 11.2.1 Choix ORM ou DAO
 - 11.2.2 Transactions
 - 11.2.3 Contraintes d'intégrité
 - 11.2.4 Migrations reproductibles
 - 11.2.5 Données seed
 - 11.3 Interface Utilisateur
 - 11.4 Optimisation JavaScript et CSS
 - 11.5 Traitement des transactions
 - 11.6 Gestion de session
 - 11.7 Sécurité
 - 11.8 Sûreté
 - 11.9 Communication inter-microservices
 - 11.10 Vérifications de plausibilité et de validité
 - 11.11 Gestion des exceptions/erreurs
 - 11.12 Journalisation et traçabilité
 - 11.13 Configurabilité
 - 11.14 Internationalisation
 - 11.15 Migration
 - 11.16 Testabilité
 - 11.17 Gestion du build
- 12. Décisions d'architecture
 - 12.1 Architecture microservices et découpage
- ADR 001 : Style architectural & découpage microservices
 - Contexte
 - Décision
 - Conséquences
 - 12.2 Persistance des données
- ADR 002 : Stratégie de persistance & transactions
 - Contexte
 - Décision
 - Conséquences
 - 12.3 Stratégie d'erreurs, versionnage & conformité
- ADR 003 : Stratégie d'erreurs, versionnage & conformité
 - Contexte
 - Décision
 - Conséquences
 - 12.4 Justification du choix de cache

- ADR 004 : Choix du cache
 - Contexte
 - Décision
 - Conséquences
 - 12.5 Justification du choix de load balancer
- ADR 005 : Choix du Load Balancer
 - Contexte
 - Décision
 - Conséquences
 - 12.6 Justification de l'api gateway
- ADR 006 : Choix de l'API Gateway
 - Contexte
 - Décision
 - Conséquences
 - 12.7 Justification du choix de message broker
- ADR 007 : Choix du message broker
 - Contexte
 - Décision
 - Conséquences
 - 12.8 Justification du choix de base de données
- ADR 008 : Choix de la base de données
 - Contexte
 - Décision
 - Conséquences
 - 12.9 Justification du choix de pattern Saga
- ADR 009 : Pattern Saga chorégraphiée vs orchestrée
 - Contexte
 - Décision
 - Conséquences
 - 12.10 Justification de l'architecture interne des microservices
- ADR 010 : Architecture interne des microservices - Hexagonale vs MVC
 - Contexte
 - Décision
 - Conséquences
 - Architecture microservices et évolutivité
 - Description des couches et dépendances
 - Organisation des dépendances
 - Contrôle du couplage aux frameworks
 - Illustration de l'architecture hexagonale de chaque microservice
 - Justification globale
 - 13. Scénarios de Qualité
 - 13.1 Arbre de qualité
 - 13.2 Scénarios d'évaluation
 - 14. Risques Techniques
 - 15. Glossaire
- Fin de la documentation arc42
 - Analyse de performance — Impact de la cache sur le service wallet-service

- Contexte
- Résultats
 - 1. Sans cache
 - 2. Avec cache
- Comparaison
- Conclusion
- Test de charge — Service des ordres (architecture microservices avec gateway)
 - Contexte
 - Résultats observés
 - Grafana — Monitoring en temps réel
 - Résumé du test K6
 - Analyse et limites du test
 - Conclusion
- Test de charge — Appels direct avec architecture monolithique (A/B direct)
 - Contexte
 - Résultats observés
 - Grafana — Monitoring en temps réel
 - Résumé du test K6
- Test de charge — Architecture event-driven avec message broker (RabbitMQ)
 - Contexte
 - Résultats observés
 - Grafana — Monitoring en temps réel
 - Résumé du test K6
 - Analyse
 - Conclusion
- Comparaison — Microservices (Gateway) vs Monolithique (A/B direct) vs Event-driven (RabbitMQ)
 - Analyse
- Test de charge — Évaluation du load balancer (NGINX)
 - Contexte
 - Résultats — 1 instance
 - Résultats — 2 instances
 - Résultats — 3 instances
 - Résultats — 4 instances
- Comparaison et analyse
 - Interprétation
 - Conclusion
- Explication des travaux CI/CD accomplis
 - Détail du pipeline CI (`.github/workflows/ci.yml`)
 - Détail du pipeline CD (`.github/workflows/cd.yml`)
 - Visualisation des pipelines
 - Stratégie de tests automatisés
- Guide d'exploitation (Runbook)
 - Prérequis
 - 1. Récupération du projet
 - 2. Structure attendue du projet
 - 3. Configuration (optionnelle)
 - 4. Construction et lancement de l'application

- 5. Vérification du bon fonctionnement
- 6. Utilisation de tous les services
- 7. Arrêt de l'application
- 8. Nettoyage (optionnel)
- 9. Problèmes fréquents
- 10. Pour rouler les tests en local
- 11. Pour exécuter les tests de charge sur les ordres avec K6
- 12. Pour exécuter les tests de charge sur le portefeuille avec K6
- 13. Pour utiliser le script de déploiement (Optionnel)
 - Exécution du script sur Windows
- Guide de démonstration BrokerX
 - 1. Inscription & Vérification d'identité (UC-01)
 - 2. Authentification & MFA (UC-02)
 - 3. Dépôt dans le portefeuille (UC-03)
 - 4. Abonnement aux données de marché (UC-04)
 - 5. Placement d'un ordre (UC-05)
 - 6. Modification / Annulation d'un ordre (UC-06)
 - 7. Appariement d'un ordre (UC-07)
 - 8. Confirmation d'exécution & Notifications (UC-08)

Début de la documentation arc42

1. Introduction & Objectifs

1.1 Objectifs métier

BrokerX est une plateforme de courtage en ligne, désormais migrée vers une architecture microservices (phase 2) et événementielle (phase 3). Les utilisateurs peuvent s'inscrire, vérifier leur identité, s'authentifier avec MFA, déposer des fonds, s'abonner aux données du marché, placer des ordres, modifier ou annuler des ordres, recevoir des notifications et consulter leur portefeuille. L'objectif principal reste d'offrir une expérience sécurisée, fluide et conforme aux exigences réglementaires du secteur financier, tout en améliorant la performance, la disponibilité et la scalabilité grâce au découpage en services indépendants et à l'API Gateway.

1.2 Fonctionnalités essentielles

- Inscription et vérification d'identité (KYC)
- Authentification forte avec MFA
- Gestion du portefeuille virtuel (dépôt, solde)
- Données de marché en temps réel via WebSockets
- Placement d'ordres d'achat et de vente
- Modification et annulation d'ordres existants

- Notifications push en temps réel
- Gestion de l'idempotence et de la traçabilité des opérations
- Journalisation des actions et audit de sécurité
- Observabilité avancée (logs structurés, métriques Golden Signals)

1.3 Objectifs de qualité pour l'architecture

Objectif de qualité	Scénario concret	Motivation
Sécurité	Un utilisateur ne peut accéder qu'à ses propres données et toutes les transactions sont chiffrées	Protection des données sensibles et conformité réglementaire
Performance	Le système doit répondre à une requête d'ordre en moins de 100 ms (P95) et traiter au moins 1200 ordres/s	Expérience utilisateur optimale, compétitivité accrue avec l'architecture événementielle
Disponibilité	Le service doit être disponible 99,9% du temps, même en cas de panne d'un composant	Continuité de service critique, haute fiabilité grâce aux microservices et WebSockets
Résilience	En cas d'incident, le système doit pouvoir récupérer et restaurer les opérations sans perte de données	Robustesse et gestion proactive des erreurs avec patterns Saga et Outbox
Maintenabilité	Les évolutions fonctionnelles doivent pouvoir être réalisées rapidement et sans régression	Réduction des coûts de maintenance et adaptation aux besoins métier
Scalabilité	Supporter la croissance du nombre d'utilisateurs et la montée en charge	Pérennité et adaptation à la demande avec architecture événementielle
Traçabilité & auditabilité	Toutes les opérations sont journalisées et traçables	Conformité et sécurité
Observabilité	Logs structurés et métriques (Golden Signals) accessibles pour le monitoring et l'optimisation	Pilotage de la performance et détection proactive des incidents
Conformité	Respect des standards du secteur (tokens, MFA, KYC)	Obligations réglementaires

1.4 Parties prenantes

Partie prenante	Rôle	Attente principale
Clients	Utilisateurs via interface web/mobile	Expérience fluide, sécurité des transactions, données de marché en temps réel, notifications instantanées, modification/annulation

Partie prenante	Rôle	Attente principale
		d'ordres
Opérations Back-Office	Gestion des règlements, supervision	Outils de gestion efficaces, visibilité sur les opérations, fiabilité des processus, observabilité avancée avec métriques Golden Signals
Conformité / Risque	Surveillance pré- et post-trade	Accès aux journaux d'audit, alertes en temps réel, conformité réglementaire garantie, traçabilité complète des opérations
Équipe DevOps	Déploiement et monitoring	Architecture microservices scalable, métriques de performance, logs structurés, déploiement automatisé
Développeurs	Maintenance et évolution	Architecture événementielle, patterns Saga/Outbox, tests automatisés, documentation API complète

Cette section synthétise les besoins métier, techniques et réglementaires qui orientent toutes les décisions architecturales.

1.5 Vue d'ensemble des exigences fonctionnelles

Cas d'utilisation	Description	Référence
Inscription & vérification d'identité	Permet à un utilisateur de créer un compte et de valider son identité via un processus KYC	UC01.md
Authentification & MFA	Permet à un utilisateur de s'authentifier avec mot de passe et code MFA	UC02.md
Dépôt dans le portefeuille	Permet à un utilisateur de déposer des fonds dans son portefeuille virtuel	UC03.md
Abonnement aux données de marché	Permet à un utilisateur de recevoir des données de marché en temps réel via WebSockets	UC04.md
Placement d'un ordre	Permet à un utilisateur de placer un ordre d'achat ou de vente sur un actif	UC05.md
Modification / Annulation d'un ordre	Permet à un utilisateur de modifier ou annuler ses ordres existants	UC06.md
Appariement interne & Exécution (matching)	Assure l'exécution automatique des ordres selon la priorité prix/temps avec architecture événementielle (patterns Saga et Outbox)	UC07.md
Notifications push	Permet à un utilisateur de recevoir des notifications en temps réel sur les événements critiques	UC08.md

1.6 Priorisation MoSCoW des cas d'utilisation (DDD)

Priorisation MoSCoW des cas d'utilisation

Cas d'utilisation	Priorité MoSCoW	Justification
UC-01 — Inscription & Vérification d'identité	Must	Sans inscription et vérification, aucun utilisateur ne peut accéder à la plateforme ni respecter les exigences réglementaires (KYC/AML). C'est la base de toute relation de confiance et de conformité légale.
UC-03 — Approvisionnement du portefeuille (dépôt virtuel)	Must	Les utilisateurs doivent pouvoir disposer de liquidités pour effectuer des opérations : sans dépôt, aucune transaction n'est possible, ce qui bloque toute activité sur la plateforme.
UC-05 — Placement d'un ordre (marché/limite) avec contrôles pré-trade	Must	Le placement d'ordre est le cœur du métier : sans cette fonctionnalité, la plateforme ne répond à aucun besoin de courtage et perd toute valeur pour les clients.
UC-02 — Authentification & MFA	Must	La sécurité des accès est indispensable pour la conformité et la confiance : une authentification forte est requise pour garantir la sécurité des comptes et la conformité réglementaire.
UC-07 — Appariement interne & Exécution (matching)	Must	L'appariement automatique est essentiel pour assurer l'exécution des ordres selon les règles de priorité prix/temps : sans cette mécanique, la plateforme ne peut traiter les transactions de manière fiable et conforme aux standards du secteur.
UC-04 — Abonnement aux données de marché	Must	Les données de marché en temps réel sont essentielles pour permettre aux utilisateurs de prendre des décisions éclairées et pour assurer la compétitivité de la plateforme face aux autres courtiers.
UC-06 — Modification / Annulation d'un ordre	Must	La capacité de modifier ou annuler des ordres est indispensable pour la gestion des risques et l'expérience utilisateur : elle évite les pertes dues aux erreurs et répond aux standards du secteur.
UC-08 — Notifications push	Must	Les notifications en temps réel sont critiques pour informer les utilisateurs des événements importants (exécutions, rejets) et garantir la transparence et la réactivité requises dans le trading.

Cette priorisation MoSCoW garantit que les fonctionnalités critiques (Must) sont livrées en priorité pour assurer la valeur métier, la conformité et la sécurité, tandis que les autres (Should/Could) enrichissent l'expérience ou optimisent le service. Les éléments en Won't Have sont explicitement exclus pour permettre une livraison rapide et maîtrisée du périmètre minimal.

1.7 Description détaillée des cas d'utilisation (DDD)

UC-01 — Inscription & Vérification d'identité

Objectif: Faciliter l'enregistrement d'un nouvel utilisateur sur la plateforme BrokerX en recueillant ses informations personnelles, en procédant à la vérification réglementaire de son identité (KYC/AML) et en activant son accès. Ce processus initie la relation de confiance entre l'utilisateur et BrokerX.

Acteur principal: Client

Déclencheur: Le Client souhaite créer un compte pour s'inscrire à la plateforme.

Pré-conditions: Aucune.

Postconditions (succès):

- Un compte utilisateur est créé avec le statut "PENDING".
- Après la vérification d'identité, le compte passe au statut "ACTIVE".

Postconditions (échec):

- Le compte n'est pas créé ou est marqué "REJECTED" avec une raison précisée.

Flux principal

1. Le Client fournit son email, un mot de passe et les données personnelles requises (nom, adresse, date de naissance).
2. Le Système vérifie la validité des informations et crée un compte avec le statut "PENDING".
3. Le Système envoie un lien de vérification d'identité par email.
4. Le Client reçoit un lien OTP (one-time passwords) et confirme son identité en cliquant sur le lien.
5. Le Système change le statut du compte à "ACTIVE" et journalise l'opération (horodatage, adresse IP, identifiant).

Alternatifs / Exceptions

- A1. Vérification d'identité non complétée : Le compte reste avec le statut "PENDING". Le lien de vérification expire après 1 jour.
- E1. Email déjà utilisé : L'opération d'inscription de l'utilisateur est rejetée. On lui propose d'aller faire un login à la place.
- E2. Informations invalides (email du mauvais format) : Le Système rejette l'inscription et demande au Client de corriger les informations.

Critère d'acceptation: Un utilisateur fournit des informations valides, reçoit le lien de vérification, confirme son identité, et son compte passe au statut "ACTIVE".

UC-02 — Authentification & MFA

Objectif: Assurer la sécurité d'accès à la plateforme BrokerX en permettant aux clients de s'authentifier via identifiant/mot de passe et un code multi-facteurs (OTP), afin de protéger les comptes contre toute tentative d'accès non autorisée.

Acteur principal: Client

Déclencheur: Le Client souhaite se connecter à la plateforme.

Pré-conditions: Le compte du Client doit être au statut "ACTIVE".

Postconditions (succès):

- Une session valide est établie pour le client (token de session).
- Le rôle de "Client" est associé à la session.

Postconditions (échec):

- Aucune session n'est créée.
- Le Client ne peut pas accéder à la plateforme.

Flux principal

1. Le Client saisit son identifiant et son mot de passe.
2. Le Système vérifie l'état du compte ainsi que les informations entrées par le client.
3. Le Système envoie un code temporaire à l'utilisation par email.
4. Le Client saisit le code MFA reçu.
5. Le Système valide le code entré, génère le token de session et journalise l'audit (IP, device, horodatage).

Alternatifs / Exceptions

- E1. Challenge MFA expiré : Si le code MFA n'est pas saisi dans le délai imparti, l'authentification échoue et le Client doit recommencer.
- E2. Challenge MFA déjà utilisé : Si le code MFA a déjà été utilisé, l'authentification échoue et un nouveau challenge doit être généré.
- E3. Échec MFA (3 tentatives) : Après 3 échecs de saisie du code MFA, l'utilisateur est verrouillé pendant 30 secondes et il doit attendre avant de réessayer.
- E4. Compte suspendu : Si le Client rate une 4e fois, son compte est suspendu et il doit contacter le support.
- E5. Compte non actif : Si le compte n'est pas au statut "ACTIVE", l'authentification est rejetée avec une raison précisée.

Critère d'acceptation: Un client saisit ses identifiants valides, reçoit le code MFA, le saisit correctement, et accède à la plateforme avec une session active.

UC-03 — Approvisionnement du portefeuille (dépôt virtuel)

Objectif: Permettre aux utilisateurs d'augmenter le solde de leur portefeuille virtuel en réalisant des dépôts simulés, afin de garantir la disponibilité des fonds nécessaires pour placer des ordres d'achat sur la plateforme BrokerX.

Acteur principal: Client

Acteurs secondaires: Service Paiement Simulé

Déclencheur: Le Client crédite son solde en monnaie fiduciaire simulée.

Pré-conditions: Le compte du Client doit être au statut "ACTIVE".

Postconditions (succès):

- Le solde du portefeuille est augmenté.
- Une écriture précise de la transaction effectuée est ajoutée au journal.

Postconditions (échec):

- Le solde du portefeuille reste inchangé.
- Une écriture d'erreur est ajoutée au journal avec le motif de l'échec.

Flux principal

1. Le Client saisit le montant à déposer.
2. Le Système vérifie les limites (montant minimum/maximum).
3. Le Système crée une transaction avec le statut "PENDING".
4. Le Service Paiement Simulé traite la demande et répond "SETTLED".
5. Le Système crédite le portefeuille, journalise la transaction et notifie le Client du résultat de l'opération.

Alternatifs / Exceptions:

- E1. Paiement rejeté : La transaction passe au statut "FAILED" et le Client reçoit une notification avec le motif du rejet.
- E2. Idempotence : Si une demande de dépôt avec la même idempotency-key, le Système renvoie le résultat précédent.
- E3. Montant hors limites : Si le montant est inférieur au minimum ou supérieur au maximum autorisé, le dépôt est refusé et le Client est informé.
- E4. Compte non trouvé ou non actif : Si le compte n'existe pas ou n'est pas au statut "ACTIVE", le dépôt est refusé.

Critère d'acceptation Un client saisit un montant valide, la transaction est acceptée, le portefeuille est crédité, et la transaction est journalisée avec succès.

UC-04 — Abonnement aux données de marché

Objectif: Offrir aux clients un accès en temps réel aux cotations et carnets d'ordres pour les instruments suivis. Ce cas permet aux investisseurs de prendre des décisions éclairées grâce à des données actualisées.

Acteur principal : Client

Secondaires : Fournisseur de données de marché simulé

Déclencheur : Le Client ouvre la vue Marché ou s'abonne à des symboles.

Préconditions : Session valide.

Postconditions (succès) :

- Flux temps réel établi (WebSocket/Server-Sent Events (SSE)), latence < 200 ms.

Postconditions (échec) :

- Aucun flux établi.

Flux principal

1. Le Client demande l'abonnement à une liste de symboles.
2. Le Système autorise (quotas, rate-limit).
3. Le Système ouvre un canal (WS/SSE) et pousse cotations/ordre book snapshots + diff.
4. Le Client reçoit updates (top of book, OHLC, trades).

Critère d'acceptation : Un utilisateur connecté s'abonne au symbole boursier AAPL. Résultat attendu : le système transmet en temps réel les cotations et le carnet d'ordres du titre AAPL.

UC-05 — Placement d'un ordre (marché/limite) avec contrôles pré-trade

Objectif: Offrir aux clients la possibilité de soumettre des ordres d'achat ou de vente (marché ou limite), soumis à des contrôles pré-trade automatisés, afin d'assurer la conformité et la sécurité des opérations sur BrokerX.

Acteur principal: Client

Acteurs secondaires: Moteur de Règles Pré-trade, Comptes/Portefeuilles

Déclencheur: Le Client soumet un ordre.

Pré-conditions: Session valide, portefeuille existant.

Postconditions (succès):

- Ordre accepté et placé dans le carnet interne.

Postconditions (échec):

- Ordre rejeté avec raison.

Flux principal

1. Le Client entre le symbole, le sens (ACHAT/VENTE), le type (MARCHE/LIMITE), la quantité, le prix (si limite) et la durée (DAY/IOC/FOK).
2. Le Système normalise les données et horodate l'opération (timestamp système en UTC avec millisecondes).
3. Le Système effectue les contrôles pré-trade :
 - Pouvoir d'achat et marge disponible
 - Règles de prix (bandes, tick size)

- Interdictions (short-sell n'est pas autorisé)
 - Limites par utilisateur (taille maximales d'ordre)
 - Vérifications de cohérence (quantité > 0)
4. Si tous les contrôles sont validés, le Système attribue un clientOrderId et persiste l'ordre.

Alternatifs / Exceptions:

- A1. Type Marché : Le prix n'est pas requis, routage immédiat.
- E1. Pouvoir d'achat insuffisant : L'ordre est rejeté avec le motif correspondant.
- E2. Violation bande de prix : L'ordre est rejeté avec le motif correspondant.
- E3. Idempotence : Si un ordre avec le même clientOrderId est reçu, le Système renvoie le résultat précédent.
- E4. Prix limite absent pour ordre limite : L'ordre n'est pas effectué et le Client est informé.
- E5. Quantité non positive : L'ordre n'est pas effectué et le Client est informé.
- E6. Short-sell non autorisé : L'ordre est rejeté avec le motif correspondant.
- E7. Tick size non respecté : L'ordre est rejeté si le prix ne respecte pas l'incrément minimal autorisé.
- E8. Taille maximale d'ordre dépassée : L'ordre est rejeté si la quantité dépasse la limite autorisée pour l'utilisateur.

Critère d'acceptation: Un client soumet un ordre valide pour le stock "TEST", tous les contrôles pré-trade sont passés, l'ordre est accepté et enregistré dans le carnet interne.

UC-06 — Modification / Annulation d'un ordre

Objectif: Offrir la possibilité de modifier ou d'annuler un ordre actif dans le carnet tant qu'il n'est pas totalement exécuté. Ce cas donne de la flexibilité aux clients pour gérer leurs stratégies de trading.

Acteur principal : Client

Secondaires : Moteur d'appariement, Portefeuilles

Déclencheur : Le Client modifie ou annule un ordre Working.

Préconditions : Ordre existant, non entièrement exécuté, modifiable.

Postconditions (succès) : Ordre mis à jour ou annulé; journal d'audit.

Postconditions (échec) : Aucune modification.

Flux principal

1. Le Client envoie Cancel ou Replace (nouvelles valeurs : quantité, prix).
2. Le Système verrouille l'ordre (optimisme/pessimisme), vérifie état.
3. Si Replace, repasse contrôles pré-trade; si Cancel, retire du carnet.
4. Retourne Cancel/Replace ACK.

Alternatifs / Exceptions :

- A1. Exécution concurrente partielle : seules quantités restantes modifiables.
- E1. Ordre déjà exécuté/annulé : Reject (avec dernier état).

Critère d'acceptation : L'utilisateur modifie un ordre limite d'achat de 10 actions AAPL à 100 \$ en ajustant le prix à 101. Résultat attendu : l'ordre est mis à jour dans le carnet et le système émet un accusé de modification (Replace ACK).

UC-07 — Appariement interne & Exécution (matching)

Objectif: Assurer l'exécution automatique des ordres en interne selon les règles de priorité (prix/temps) en rapprochant acheteurs et vendeurs. Ce cas fournit la mécanique centrale de traitement des transactions sur la plateforme.

Acteur principal: Moteur d'appariement interne

Acteurs secondaires: Données de Marché, Portefeuilles

Déclencheur: Nouvel ordre arrive dans le carnet.

Pré-conditions: Carnet maintenu (prix/temps), règles de priorité définies.

Postconditions (succès):

- Transactions générées (partielles possibles), état d'ordre mis à jour.

Postconditions (échec):

- Ordre reste Working (pas de contrepartie).

Flux principal

1. Le Moteur insère l'ordre dans le carnet (Buy/Sell).
2. Il recherche la meilleure contrepartie (price-time priority).
3. Si match, crée une ou plusieurs exécutions (fills), met à jour quantités.
4. Émet événements ExecutionReport (Fill/Partial Fill).
5. Met à jour top-of-book, publie update marché.

Alternatifs / Exceptions:

- A1. IOC/FOK : IOC exécute le possible puis annule le reste; FOK exécute tout sinon annule.

Critère d'acceptation: Un ordre d'achat de 10 actions AAPL à 100 \$ rencontre un ordre de vente identique. Résultat attendu : une transaction est générée, les quantités sont ajustées et un rapport d'exécution (Execution Report) est publié.

UC-08 — Confirmation d'exécution & Notifications

Objectif: Notifier les clients de l'état final de leurs ordres (exécuté partiellement ou totalement, rejeté), en fournissant des informations précises et traçables. Ce cas garantit la transparence et la confiance dans les transactions.

Acteur principal : Système

Secondaires : Client, Back-Office

Déclencheur : Réception d'un ExecutionReport.

Préconditions : Ordre existant.

Postconditions (succès) : Client notifié (UI push/email), état ordre mis à jour.

Postconditions (échec) : Notification peut échouer mais l'exécution reste valide.

Flux principal

1. Le Système met à jour l'état de l'ordre (Partial/Filled).
2. Crée un enregistrement d'exécution (prix, qty, frais).
3. Notifie le Client (temps réel).
4. Journalise l'audit (horodatage, source).

Alternatifs / Exceptions :

- E1. Échec de push : retry, puis fallback email.

Critère d'acceptation : Une fois l'ordre exécuté, le système envoie une notification à l'utilisateur. Résultat attendu : l'utilisateur reçoit une confirmation indiquant que l'ordre a été entièrement exécuté dans l'application.

2. Contraintes d'architecture

2.1 Contraintes techniques

Contrainte	Explication
Architecture microservices Java/Spring Boot	L'application est développée en Java avec le framework Spring Boot, organisée en microservices indépendants avec API Gateway.
Base de données PostgreSQL	Toutes les données persistantes doivent être stockées dans une base PostgreSQL. Chaque microservice dispose de sa propre base de données pour respecter l'isolation des données.
Architecture événementielle	Le système utilise une architecture événementielle avec les patterns Saga et Outbox pour gérer la cohérence des données entre microservices lors des transactions distribuées.
Authentification MFA	L'authentification multi-facteurs est obligatoire pour tous les accès utilisateurs.
Docker & Docker Compose	Le déploiement doit se faire via des conteneurs Docker, orchestrés avec Docker Compose. Chaque microservice est conteneurisé indépendamment.
API REST & Communication inter-services	Les interfaces externes doivent être exposées sous forme d'API REST. La communication entre microservices utilise des appels HTTP synchrones et des événements asynchrones.
Load Balancing	NGINX est utilisé comme load balancer pour distribuer le trafic entre les instances des microservices et assurer la haute disponibilité.

Contrainte	Explication
Journalisation et audit	Toutes les opérations critiques, y compris les événements d'exécution (fills, partial fills, ExecutionReport), doivent être journalisées pour audit et traçabilité.
Caching	Un mécanisme de cache (local ou distribué) doit être utilisé pour optimiser la consultation du carnet d'ordres, du top-of-book et la diffusion des données de marché, afin d'améliorer la performance et limiter la charge sur la base de données.
Observabilité	Le système doit fournir des logs structurés, des métriques (Golden Signals) et des traces pour le moteur d'appariement et l'exécution des ordres, afin d'assurer le suivi, la détection des anomalies et la conformité. Prometheus et Grafana sont utilisés pour le monitoring.

2.2 Contraintes réglementaires et de conformité

Contrainte	Explication
KYC (Know Your Customer)	La vérification d'identité est obligatoire pour chaque utilisateur avant toute opération.
Sécurité des données	Chiffrement des données sensibles et respect des standards du secteur financier.
Traçabilité	Toutes les actions doivent être traçables et accessibles pour audit.

2.3 Contraintes organisationnelles

Contrainte	Explication
CI/CD	Les livraisons doivent passer par un pipeline d'intégration et de déploiement continu.
Documentation	Toute nouvelle fonctionnalité doit être documentée selon les standards internes.
Tests automatisés	Les fonctionnalités critiques, dont le moteur d'appariement interne (matching), doivent être couvertes par des tests automatisés (unitaires, d'intégration et E2E).

2.4 Contraintes de gestion et maintenance

Contrainte	Explication
Maintenabilité	Le code doit être structuré pour faciliter les évolutions et la correction des bugs.
Monitoring	Des outils de monitoring doivent être mis en place pour suivre la santé du système, en particulier le moteur d'appariement : détection des incohérences du carnet, alertes et gestion des rollbacks segmentaires. L'observabilité doit permettre un suivi en temps réel des transactions et du matching.

Ces contraintes doivent être respectées tout au long du cycle de vie du projet et orientent toutes les décisions architecturales et techniques.

3. Portée du système et contexte

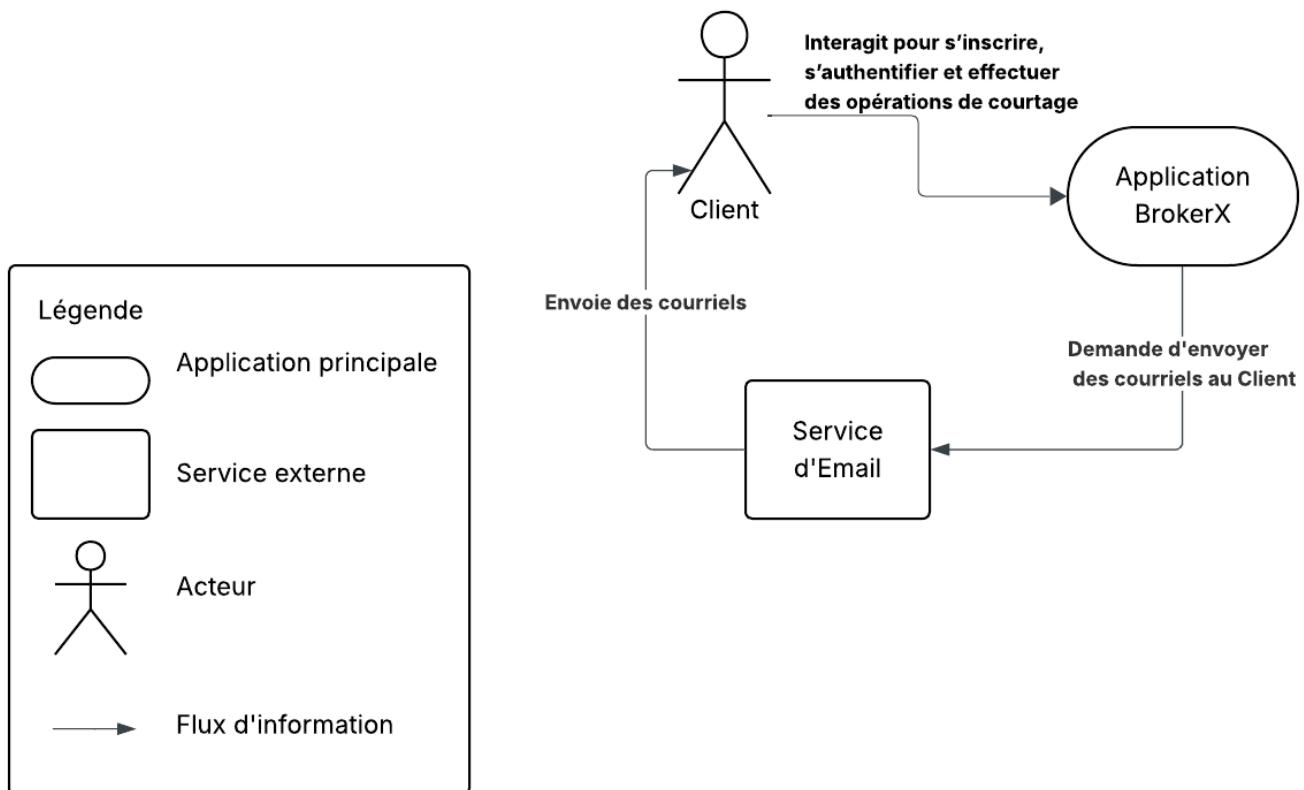
3.1 Contexte métier

Le tableau ci-dessous présente les principaux acteurs et leurs interactions, en cohérence avec le diagramme de contexte DDD (voir section 3.2).

Acteur / Système	Interagit avec	Flux / Description
Client	Application BrokerX	S'inscrit, s'authentifie, effectue des opérations de courtage
Application BrokerX	Service d'Email	Demande l'envoi de courriels (vérification, MFA, notifications) au Service d'Email
Service d'Email	Client	Envoie les courriels (liens de vérification, codes MFA, notifications) au Client

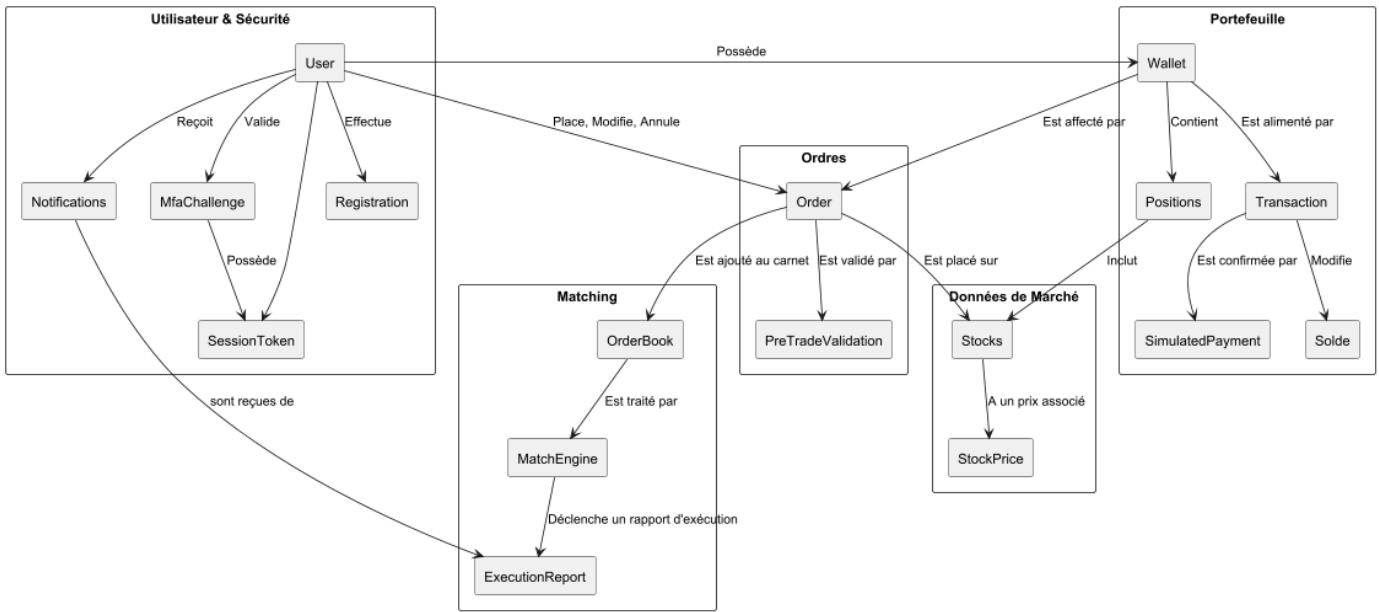
3.2 Diagramme de contexte DDD

Le diagramme de contexte ci-dessous illustre les frontières du système BrokerX, ses principaux partenaires externes et les interactions majeures du point de vue métier et DDD.



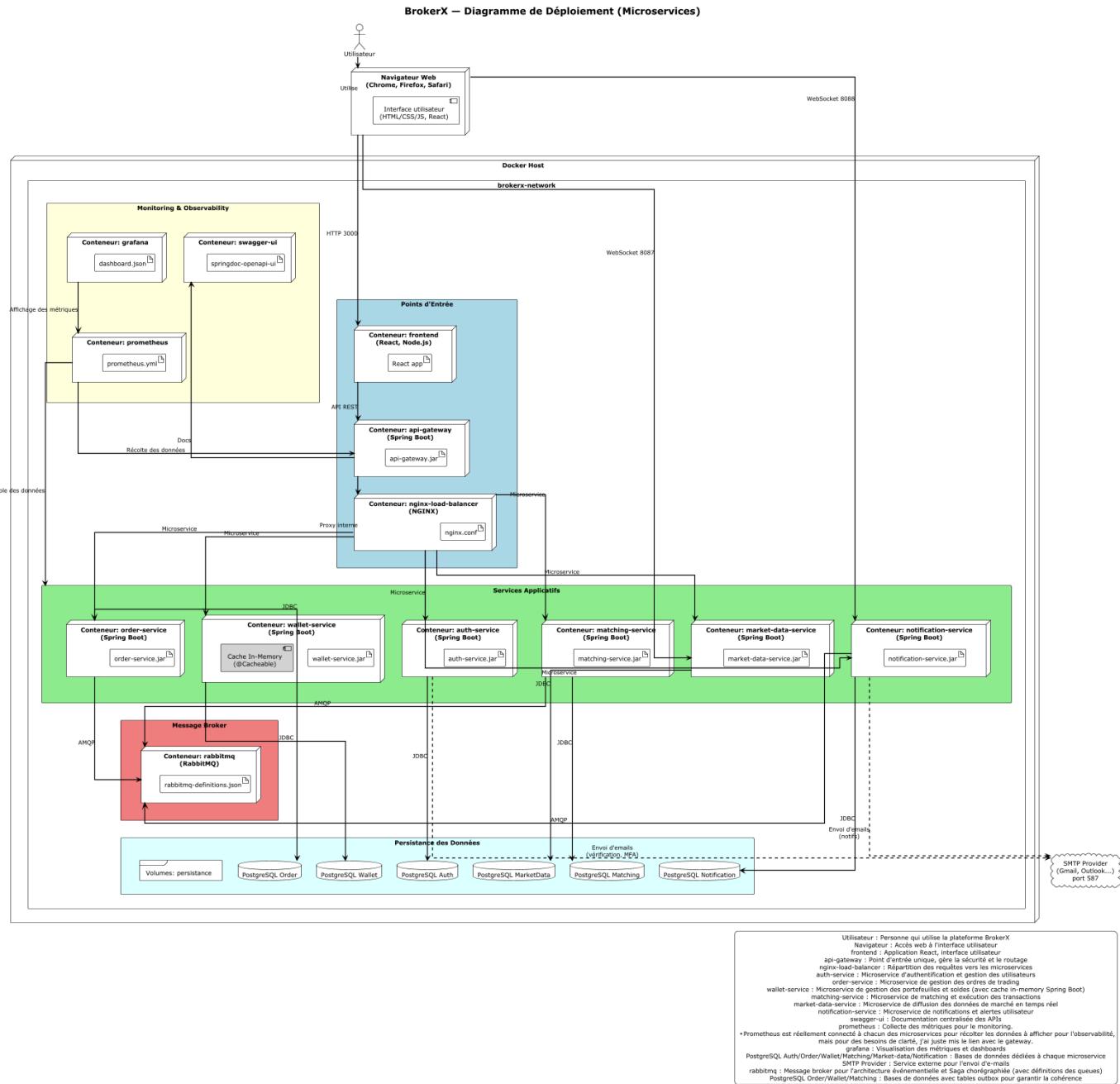
3.3 Diagramme des bounded contexts (DDD)

Le diagramme des bounded contexts présente la découpe du domaine BrokerX en sous-domaines fonctionnels cohérents (bounded contexts), chacun représentant une zone de responsabilité métier distincte. Cette vue permet de visualiser les interactions et les frontières entre les différents contextes métier.



3.4 Contexte technique

Le diagramme de déploiement ci-dessous illustre l'infrastructure technique et les canaux présentement utilisés par BrokerX.



Système technique externe	Canal / Protocole	Format des données	Sécurité	Explication interface
Clients	API REST (HTTPS)	JSON	MFA, chiffrement TLS	Accès utilisateur, ordres, notifications, cotations temps réel
SMTP Provider (GMail)	SMTP (TLS)	Texte, HTML	Chiffrement TLS, authentification SMTP	Envoyer d'e-mails pour notifications, vérification d'adresse, récupération de mot de passe

Chaque interface technique est sécurisée par chiffrement TLS et, selon le cas, par authentification forte (MFA, tokens, SMTP login). Les explications précisent le rôle et les exigences de chaque interface.

4. Stratégie de solution

L'architecture de BrokerX repose désormais sur une approche microservices événementielle, chaque domaine métier étant isolé dans un service indépendant (auth-service, api-gateway, matching-service, order-service, wallet-service, market-data-service, notification-service). Cette stratégie permet une évolutivité, une résilience et une flexibilité accrues, tout en facilitant la maintenance et le déploiement.

Architecture événementielle et messaging : Le système utilise désormais RabbitMQ comme message broker pour implémenter une architecture événementielle avec les patterns Saga chorégraphiée et Outbox. Cette approche garantit la cohérence des données entre microservices lors des transactions distribuées (ex: matching d'ordres, envoi de notifications) tout en maintenant leur indépendance.

Communication hybride : Les communications inter-services combinent des appels HTTP synchrones pour les opérations simples et des événements asynchrones via RabbitMQ pour les workflows complexes nécessitant une coordination entre plusieurs services. Le pattern Outbox garantit la fiabilité de la publication d'événements.

Chaque microservice est développé principalement en Java avec Spring Boot, et expose ses fonctionnalités via des API REST sécurisées (HTTPS/JSON). L'api-gateway centralise l'accès aux services, gère le routage, l'authentification et la sécurité des échanges. Un load balancer avec NGINX distribue le trafic entre les instances des microservices.

La persistance des données est gérée de façon indépendante : chaque microservice possède sa propre base de données (PostgreSQL), avec des tables outbox pour les services participants aux workflows événementiels. Cette séparation permet d'éviter les dépendances fortes et facilite l'évolution de chaque service.

Observabilité renforcée : Le système fournit des logs structurés, des métriques (Golden Signals) et des traces spécifiquement adaptées aux workflows événementiels. Prometheus et Grafana permettent de montrer les performances du message broker, les temps de traitement des événements et la santé globale des Sagas.

La sécurité est centralisée via l'auth-service, qui gère l'authentification forte (MFA), la gestion des rôles et la journalisation des actions critiques. Les exigences réglementaires (KYC, RGPD) sont respectées grâce à une gestion fine des accès et à la traçabilité des opérations sensibles.

Le déploiement et l'orchestration des services sont assurés par Docker et Docker Compose, permettant de gérer facilement les environnements, la scalabilité et la portabilité. Un pipeline CI/CD adapté aux microservices automatise les tests et les livraisons, garantissant la qualité et la rapidité des mises en production.

Un pipeline CI/CD adapté aux microservices automatise les tests (unitaires, d'intégration, E2E) et les livraisons pour chaque service, garantissant la qualité et la rapidité des mises en production. Les tests sont systématisés pour détecter rapidement les régressions et assurer la fiabilité du système global.

La documentation des API et des services est produite en continu, facilitant l'intégration de nouveaux membres et l'évolution du système. L'utilisation de standards ouverts (Swagger) permet de garder une trace claire des interfaces et des contrats entre services.

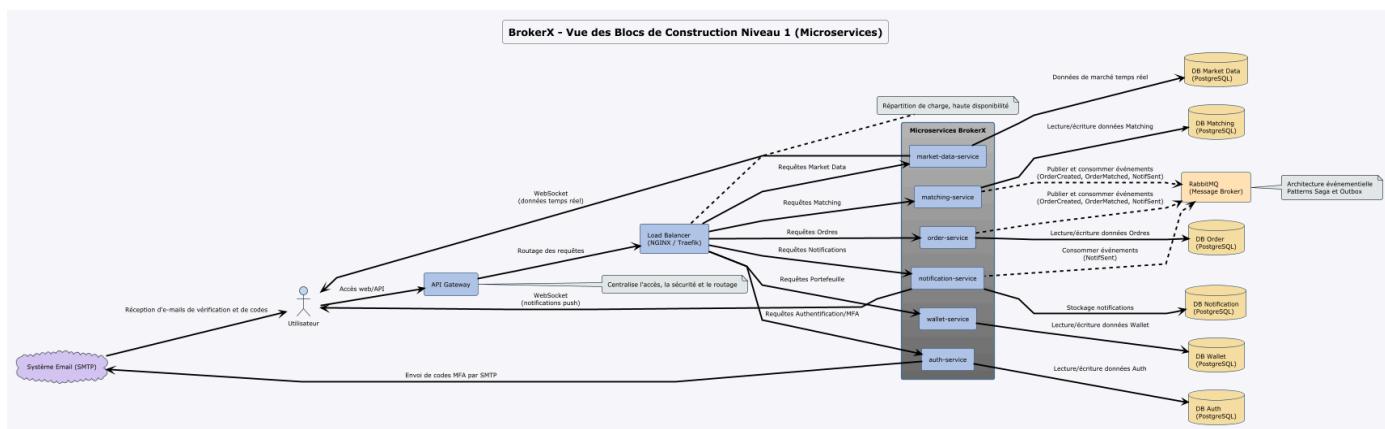
Ce choix d'architecture microservices événementielle, associé à des technologies éprouvées et à des processus automatisés, permet de répondre efficacement aux objectifs de sécurité, performance, disponibilité et conformité, tout en gardant la solution évolutive pour de futurs besoins.

5. Vue des blocs de construction

5.1 Introduction

Cette section présente la structure statique de BrokerX selon deux niveaux de précision : un premier niveau très high level (système vu comme un bloc, interfaces externes), puis un second niveau qui détaille les principaux composants internes.

5.2 Vue d'ensemble high level (Niveau 1)



Le diagramme ci-dessus illustre les principaux éléments et interactions du système BrokerX :

Élément	Type / Technologie	Rôle / Description
Utilisateur	Acteur externe	Interagit avec BrokerX via l'interface web : navigation, formulaires, requêtes REST, réception d'e-mails, WebSocket pour données temps réel
API Gateway	Service dédié (Spring Cloud Gateway, etc.)	Point d'entrée unique : centralise l'accès, la sécurité et le routage des requêtes vers le load balancer
Load Balancer	NGINX	Répartit la charge entre les instances des microservices, assure la haute disponibilité et la résilience du système
RabbitMQ	Message Broker	Hub central événementiel, implémente les patterns Saga et Outbox pour la cohérence des transactions distribuées
auth-service	Microservice Java/Spring Boot	Gère l'authentification, MFA, gestion des rôles, journalisation des actions critiques
matching-service	Microservice Java/Spring Boot	Appariement des ordres selon les règles métier, gestion du carnet d'ordres, publication d'événements de matching

Élément	Type / Technologie	Rôle / Description
order-service	Microservice Java/Spring Boot	Gestion des ordres de trading, validation pré-trade, traçabilité des opérations, publication d'événements d'ordres
wallet-service	Microservice Java/Spring Boot	Gestion du portefeuille virtuel, dépôts, solde, transactions, consommation d'événements pour mises à jour
notification-service	Microservice Java/Spring Boot	Envoi de notifications temps réel, consommation d'événements des autres services
market-data-service	Microservice Java/Spring Boot	Diffusion des cotations et carnets d'ordres via WebSocket, données de marché temps réel
DB Auth	PostgreSQL	Stocke les données d'authentification, MFA, rôles
DB Matching	PostgreSQL	Stocke les données d'appariement, carnet d'ordres
DB Order	PostgreSQL	Stocke les ordres, statuts, logs d'exécution
DB Wallet	PostgreSQL	Stocke les portefeuilles, transactions, historiques
DB Notification	PostgreSQL	Stocke les notifications envoyées, historique des alertes
DB Market Data	PostgreSQL	Stocke les données de marché temps réel, cotations, historiques des prix
Système Email (SMTP)	Service externe	Reçoit les demandes d'envoi d'e-mails (liens de vérification, codes MFA) et transmet les courriels à l'utilisateur

Principales interactions :

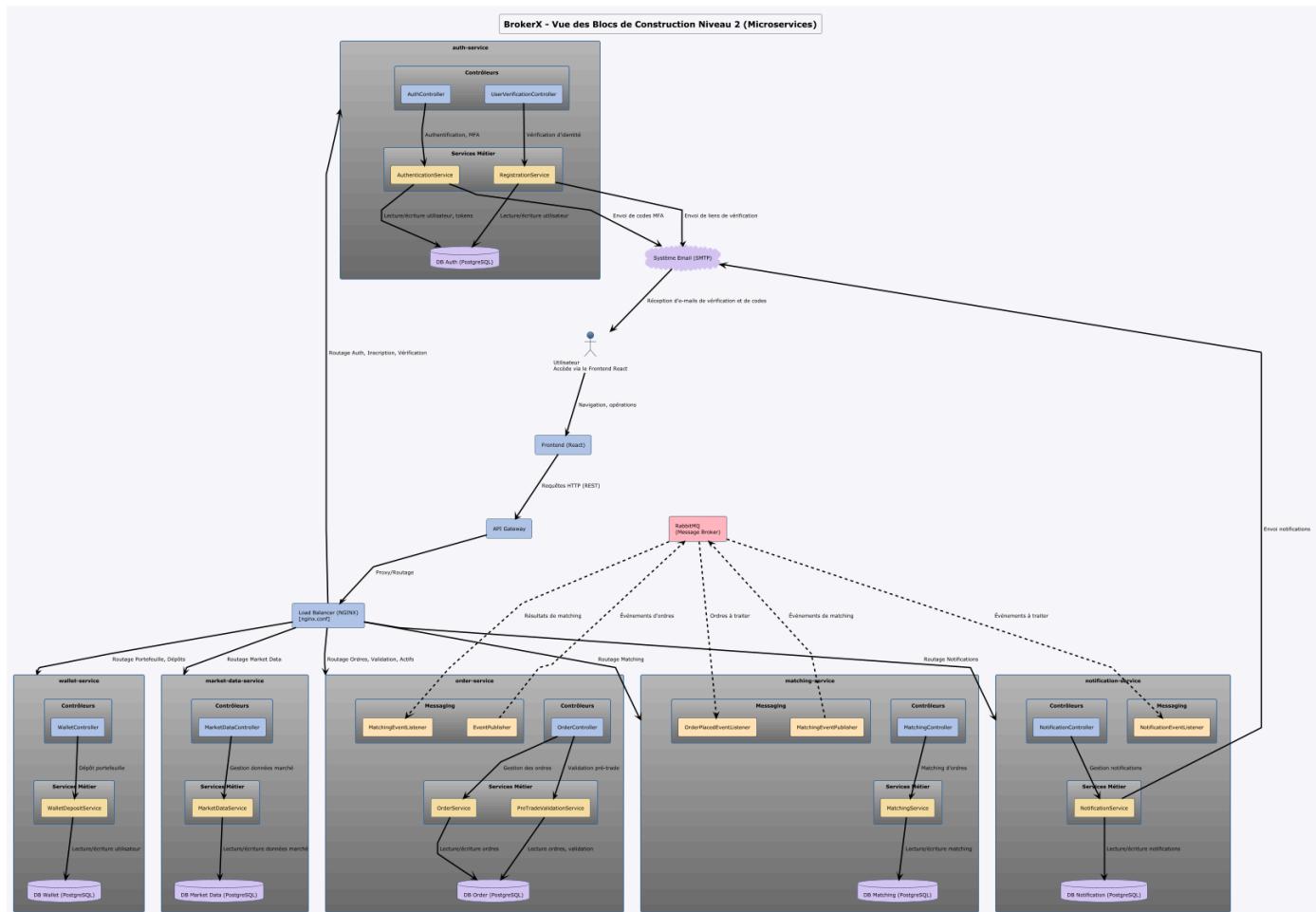
- L'utilisateur accède à BrokerX via l'API Gateway, qui centralise l'accès, la sécurité et le routage des requêtes
- L'API Gateway route les requêtes vers le load balancer, qui répartit la charge vers les microservices
- RabbitMQ orchestre les transactions distribuées avec les patterns Saga et Outbox (matching d'ordres, notifications, mises à jour de portefeuille)
- Chaque microservice gère un domaine métier spécifique et possède sa propre base PostgreSQL
- Les microservices communiquent avec le système email (SMTP) pour l'envoi de codes MFA et de notifications
- Le système email transmet les e-mails de vérification et de codes à l'utilisateur

Données stockées dans les bases :

- Auth-service : utilisateurs, MFA, rôles
- Matching-service : carnet d'ordres, appariements
- Order-service : ordres, statuts, logs d'exécution
- Wallet-service : portefeuilles, transactions
- Notification-service : notifications envoyées, historique des alertes, préférences utilisateur
- Market-data-service : cotations temps réel, carnets d'ordres, historiques des prix, snapshots

Cette vue permet de comprendre la structure globale du système, les principaux composants et leurs interactions, ainsi que la répartition des responsabilités entre les microservices.

5.3 Vue interne des composants (Niveau 2)



Le diagramme ci-dessus détaille l'architecture interne du système BrokerX, organisée en microservices :

Composant	Type / Rôle	Description
Utilisateur (Frontend React)	Acteur externe	Accède à l'application web pour toutes les opérations de courtage
API Gateway	Proxy/Routage	Point d'entrée unique, route les requêtes vers les microservices
Load Balancer (NGINX)	Répartition de charge / proxy	Distribue les requêtes vers les instances des microservices, config nginx.conf
RabbitMQ	Message Broker	Hub central événementiel, orchestre les patterns Saga et Outbox
AuthController	Contrôleur (auth-service)	Gère l'authentification et la vérification MFA
UserVerificationController	Contrôleur (auth-service)	Gère la vérification d'identité (KYC)
OrderController	Contrôleur (order-service)	Permet de passer et consulter des ordres de trading

Composant	Type / Rôle	Description
WalletController	Contrôleur (wallet-service)	Gère le portefeuille et les dépôts
MatchingController	Contrôleur (matching-service)	Gère le matching des ordres
NotificationController	Contrôleur (notification-service)	Gère l'envoi de notifications
MarketDataController	Contrôleur (market-data-service)	Gère les données de marché
WebSocketController	Contrôleur (market-data-service)	Diffuse les données de marché en temps réel via WebSocket
AuthenticationService	Service métier (auth-service)	Logique d'authentification et MFA
RegistrationService	Service métier (auth-service)	Gère l'inscription et la vérification utilisateur
OrderService	Service métier (order-service)	Logique métier pour la gestion des ordres
PreTradeValidationService	Service métier (order-service)	Valide les ordres avant exécution
StockService	Service métier (order-service)	Gestion des actifs financiers et cotations
WalletDepositService	Service métier (wallet-service)	Gère les dépôts dans le portefeuille
MatchingService	Service métier (matching-service)	Logique de matching d'ordres
NotificationService	Service métier (notification-service)	Logique d'envoi de notifications
MarketDataService	Service métier (market-data-service)	Logique des données de marché
EventPublisher	Composant messaging (order-service)	Publie les événements d'ordres vers RabbitMQ
MatchingEventListener	Composant messaging (order-service)	Écoute les événements de matching depuis RabbitMQ
MatchingEventPublisher	Composant messaging (matching-service)	Publie les événements de matching vers RabbitMQ
OrderPlacedEventListener	Composant messaging (matching-service)	Écoute les événements d'ordres placés depuis RabbitMQ

Composant	Type / Rôle	Description
NotificationEventListener	Composant messaging (notification-service)	Écoute tous types d'événements pour générer des notifications
DB Auth (PostgreSQL)	Base de données dédiée	Stocke les données d'authentification et utilisateurs
DB Order (PostgreSQL)	Base de données dédiée	Stocke les ordres et historiques de trading
DB Wallet (PostgreSQL)	Base de données dédiée	Stocke les informations de portefeuille
DB Matching (PostgreSQL)	Base de données dédiée	Stocke les données de matching
DB Notification (PostgreSQL)	Base de données dédiée	Stocke les notifications envoyées et historiques
DB Market Data (PostgreSQL)	Base de données dédiée	Stocke les données de marché temps réel et historiques
Système Email (SMTP)	Service externe	Envoie les notifications et codes MFA aux utilisateurs

Principales interactions :

- L'utilisateur interagit avec le Frontend React, qui transmet les requêtes à l'API Gateway
- L'API Gateway centralise et sécurise l'accès, puis transmet les requêtes au Load Balancer (NGINX)
- Le Load Balancer distribue les requêtes vers les microservices appropriés selon la configuration de routage
- **Architecture événementielle :** RabbitMQ orchestre les workflows complexes via les patterns Saga et Outbox :
 - Order-service publie des événements d'ordres placés via EventPublisher
 - Matching-service écoute ces événements via OrderPlacedEventListener et traite les appariements
 - MatchingEventPublisher diffuse les résultats de matching vers RabbitMQ
 - Order-service consomme les événements de matching via MatchingEventListener pour mettre à jour les statuts
 - Notification-service écoute tous types d'événements via NotificationEventListener pour générer des notifications
- Chaque microservice possède ses propres contrôleurs et services métier, ainsi qu'une base de données dédiée
- Les services d'inscription et d'authentification envoient des e-mails (liens de vérification, codes MFA) au système SMTP, qui transmet ces messages à l'utilisateur
- Le market-data-service diffuse les données temps réel via WebSocket pour les cotations et carnets d'ordres

Cette vue permet de comprendre la répartition des responsabilités, les flux d'information synchrones et asynchrones, ainsi que la collaboration entre les composants dans une architecture microservices événementielle. L'utilisation de RabbitMQ avec les patterns Saga et Outbox garantit la cohérence des données distribuées tout en maintenant l'indépendance des services, permettant une meilleure résilience et scalabilité du système global.

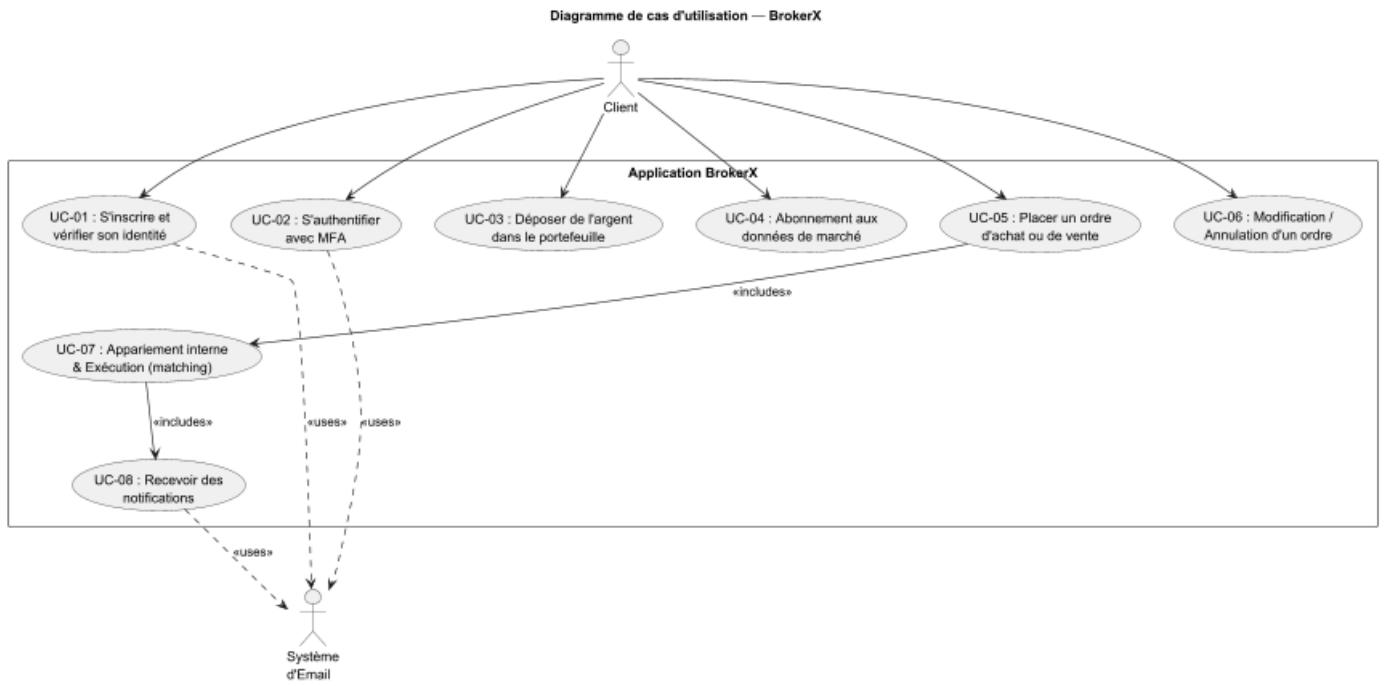
5.4 Organisation du code et conventions

Le code de chacun des microservices est structuré selon une approche hexagonale : les "adapters" gèrent les interactions externes (web, persistence, messaging), le "domain" regroupe la logique métier et les modèles, et

"infrastructure" contient la configuration technique. Les conventions de nommage (camelCase) et de structure facilitent l'extension et la maintenance du projet. Les tests automatisés couvrent les fonctionnalités critiques, et la documentation est maintenue à jour pour chaque évolution majeure.

6. Vue d'ensemble des scénarios

Diagramme



Contexte

La vue scénarios expose les principaux cas d'utilisation du système BrokerX, tels qu'ils sont vécus par les utilisateurs et les systèmes externes. Elle permet de visualiser les interactions entre le client et l'application, ainsi que les dépendances fonctionnelles entre les différents UC.

Éléments

- Acteurs externes : Client (utilisateur principal), Service d'Email (SMTP)
- Cas d'utilisation : Incription & vérification d'identité, Authentification & MFA, Dépôt dans le portefeuille, Abonnement aux données de marché, Placement d'un ordre, Modification / Annulation d'un ordre, Appariement interne & Exécution (matching), Réception de notifications

Relations

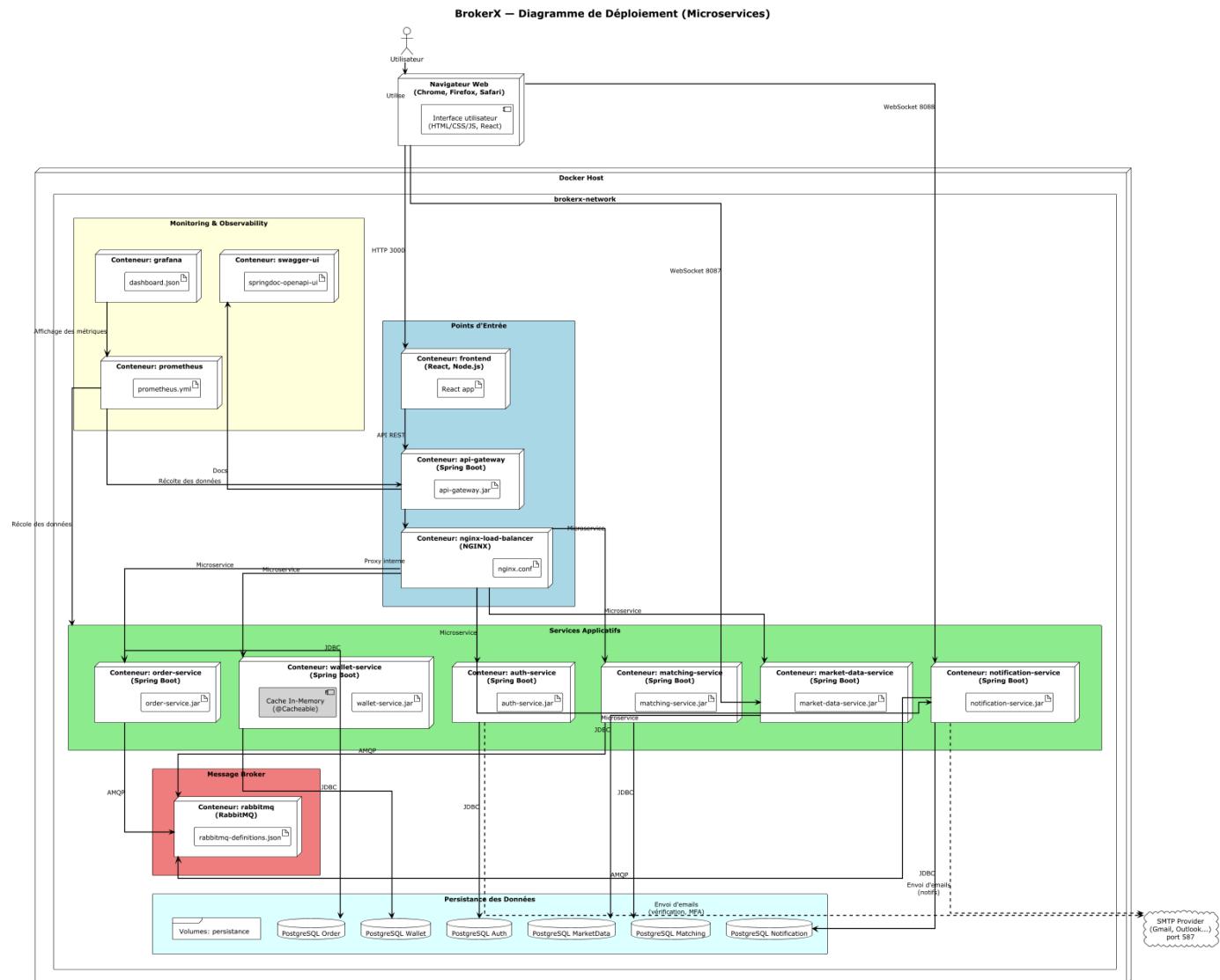
- Le client peut initier chacun des cas d'utilisation principaux, sauf l'appariement interne qui est déclenché automatiquement lorsqu'un placement d'ordre est effectué et les notifications qui sont générées par l'appariement interne
- Le placement d'un ordre déclenche le cas d'utilisation d'appariement interne & exécution
- L'appariement interne génère des notifications pour informer le client

Rationnel

Cette vue permet de relier les besoins métier aux fonctionnalités du système, de valider la couverture fonctionnelle et d'illustrer les interactions principales, y compris le processus complet d'appariement et d'exécution des ordres avec notifications. L'architecture événementielle garantit que tous les événements critiques (exécutions, rejets, mises à jour de portefeuille) génèrent des notifications appropriées vers les clients, assurant transparence et traçabilité des opérations de trading.

7. Vue de déploiement

Le diagramme ci-dessous illustre l'infrastructure technique et la distribution des principaux artefacts du système BrokerX dans une architecture microservices :



Utilisateur : Personne qui utilise la plateforme BrokerX
 Navigateur : Accès web à l'interface utilisateur
 frontend : Application React, interface utilisateur
 api-gateway : Point d'entrée unique, gère la sécurité et le routage
 auth-service : Microservice d'autentification et gestion des utilisateurs
 order-service : Microservice de gestion des ordres de trading
 wallet-service : Microservice de gestion du portefeuille (utilisant Spring Boot)
 matching-service : Microservice de matching et exécution des transactions
 market-data-service : Microservice de diffusion des données de marché en temps réel
 notification-service : Microservice de notification pour les événements critiques
 swagger-ui : Documentation centralisée des APIs
 prometheus : Collecte des métriques pour le monitoring et l'observabilité
 rabbitmq : Message broker pour l'architecture événementielle et Saga chorégraphiée (avec définitions des queues)
 PostgreSQL Auth/Order/Matching/Market-data/Notification : Bases de données dédiées à chaque microservice
 SMTP Provider : Service externe pour l'envoi d'e-mails
 PostgreSQL Order/Wallet/Matching : Bases de données avec tables outil pour garantir la cohérence

Nœud / Composant	Type / Technologie	Canal / Protocole	Sécurité / Persistance	Explication technique
Utilisateur	Acteur externe	-	-	Interagit via le navigateur web
Frontend (React)	Application web	HTTPS/HTTP 8090	TLS	Interface utilisateur, communique avec l'API Gateway
API Gateway	Microservice Java/Spring	HTTPS/HTTP 8080	TLS, routage	Point d'entrée unique, centralise la sécurité et le routage vers les microservices
Load Balancer (NGINX)	Proxy/Routage	Interne Docker	TLS, config nginx.conf	Répartit la charge entre les microservices, assure la haute disponibilité
auth-service	Microservice Java/Spring	Interne Docker	TLS, MFA, DB dédiée	Gère l'authentification, MFA, gestion des rôles, journalisation
order-service	Microservice Java/Spring	Interne Docker	TLS, DB dédiée	Gestion des ordres de trading, validation pré-trade, traçabilité
wallet-service	Microservice Java/Spring	Interne Docker	TLS, DB dédiée	Gestion du portefeuille virtuel, dépôts, solde, transactions
matching-service	Microservice Java/Spring	Interne Docker	TLS, DB dédiée	Appariement des ordres, gestion du carnet d'ordres
market-data-service	Microservice Java/Spring	Interne Docker	TLS, DB dédiée	Données du marché en temps réel, cotations et orderbooks de chaque stock.
notification-service	Microservice Java/Spring	Interne Docker	TLS, DB dédiée	Notifications envoyées à l'utilisateur en temps réel quand un ordre est placé
DB Auth	PostgreSQL	JDBC 5432	Authentification, persistance	Stocke les données d'authentification, MFA, rôles
DB Order	PostgreSQL	JDBC 5432	Persistance	Stocke les ordres, statuts, logs d'exécution

Nœud / Composant	Type / Technologie	Canal / Protocole	Sécurité / Persistance	Explication technique
DB Wallet	PostgreSQL	JDBC 5432	Persistance	Stocke les portefeuilles, transactions, historiques
DB Matching	PostgreSQL	JDBC 5432	Persistance	Stocke les données d'appariement, carnet d'ordres
DB Notification	PostgreSQL	JDBC 5432	Persistance	Stocke les notifications envoyées, historique des alertes
DB Market Data	PostgreSQL	JDBC 5432	Persistance	Stocke les données de marché temps réel, cotations, historiques
Volume: postgres_data	Volume Docker	Interne Docker	Persistance	Persistance des données PostgreSQL
RabbitMQ	Message Broker	AMQP 5672	Authentification	Hub événementiel, patterns Saga et Outbox pour cohérence distribuée
Swagger UI	Documentation	HTTP 8081	-	Documentation centralisée des APIs des microservices
Prometheus	Monitoring	HTTP 9090	-	Collecte des métriques pour observabilité et monitoring
Grafana	Visualisation	HTTP 3001	-	Dashboards et visualisation des métriques Prometheus
SMTP Provider (Gmail, Outlook...)	Service externe (SMTP)	SMTP 587	TLS, authentification	Envoi d'e-mails (notifications, vérification, récupération)

Contexte

La vue déploiement décrit l'architecture physique du système : chaque microservice est déployé dans son propre conteneur Docker, tous connectés au même réseau Docker interne (brokerx-network). Le frontend (React) communique avec l'API Gateway, qui centralise la sécurité et le routage. L'API Gateway transmet les requêtes au load balancer (NGINX), qui répartit la charge vers les microservices métier (auth, order, wallet, matching, market-data, notification). Chaque microservice possède sa propre base PostgreSQL pour garantir l'isolation des données et la robustesse des transactions. Les volumes Docker assurent la persistance des données, même lors des mises à jour ou redémarrages. Le message broker RabbitMQ est lié à certains microservices afin de gérer le placement d'ordre de façon événementielle. Le service SMTP externe gère l'envoi des notifications et des codes MFA. Cette architecture permet

une scalabilité horizontale, une haute disponibilité et une maintenance facilitée, chaque service pouvant être mis à jour ou redémarré indépendamment.

Éléments

- Conteneurs Docker pour chaque microservice (auth-service, order-service, wallet-service, matching-service, market-data-service, notification-service, api-gateway, frontend)
- Load Balancer (NGINX) pour la répartition de charge
- Message Broker (RabbitMQ) pour l'architecture événementielle
- Prometheus, Grafana et Swagger pour le monitoring et l'observabilité
- Réseau Docker interne (brokerx-network) pour la communication sécurisée
- Bases PostgreSQL dédiées pour chaque microservice
- Volume Docker pour la persistance des données
- Service SMTP externe pour l'envoi d'e-mails

Relations

- Le frontend communique avec l'API Gateway via HTTPS
- L'API Gateway transmet les requêtes au load balancer NGINX
- NGINX répartit la charge vers les microservices métier
- Chaque microservice accède à sa propre base PostgreSQL
- RabbitMQ est utilisé pour la communication asynchrone entre certains microservices
- Les volumes Docker assurent la persistance des données
- Les microservices communiquent avec le service SMTP pour l'envoi d'e-mails

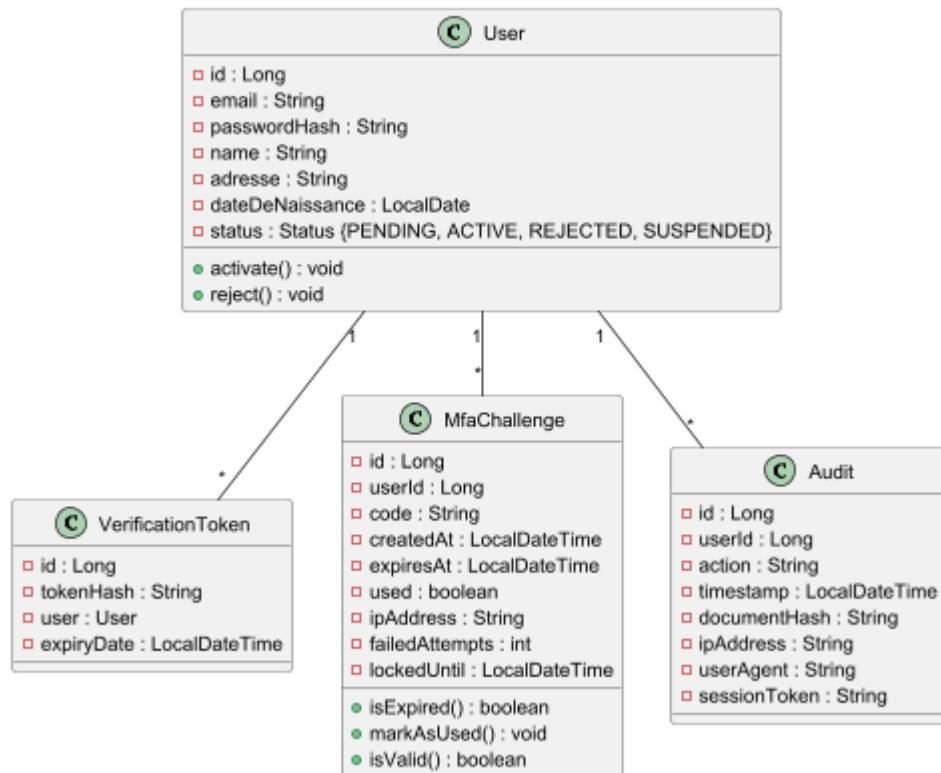
Rationnel

Cette vue permet de comprendre la topologie du système BrokerX dans une architecture microservices et événementielle : chaque service est isolé, scalable et maintenable indépendamment. Le réseau Docker interne garantit la sécurité et la rapidité des communications. La séparation des bases de données assure la conformité, la robustesse et la traçabilité des opérations. Le load balancer et l'API Gateway centralisent le routage et la sécurité, tandis que le service SMTP gère les notifications critiques. Cette organisation facilite la supervision, la gestion des incidents et l'évolution de l'architecture technique.

8. Vue Logique

Diagrammes de classes par microservice

Diagramme de classes - Auth-Service



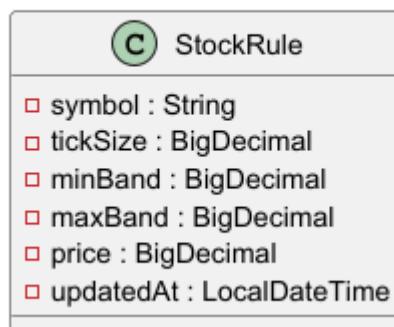
- Auth Service :

Diagramme de classes - Matching-Service



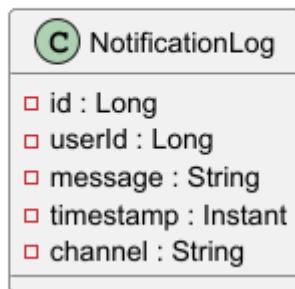
- Matching Service :

Diagramme de classes - Market-Data-Service



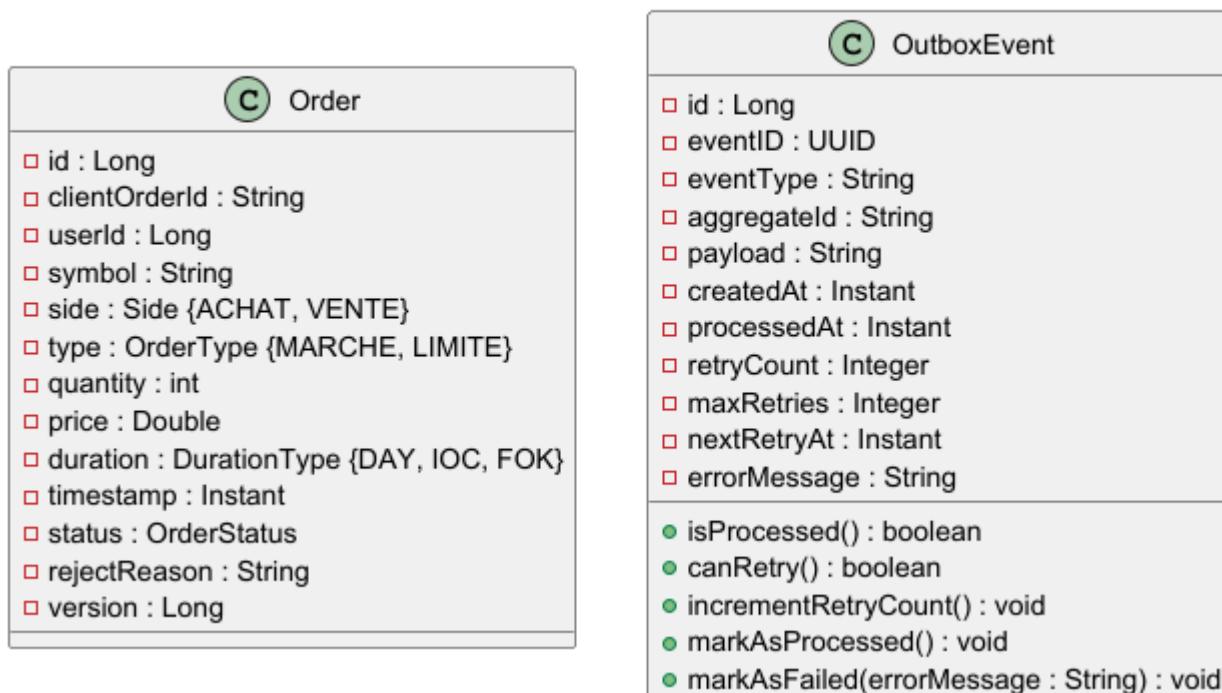
- Market-Data Service :

Diagramme de classes - Notification-Service

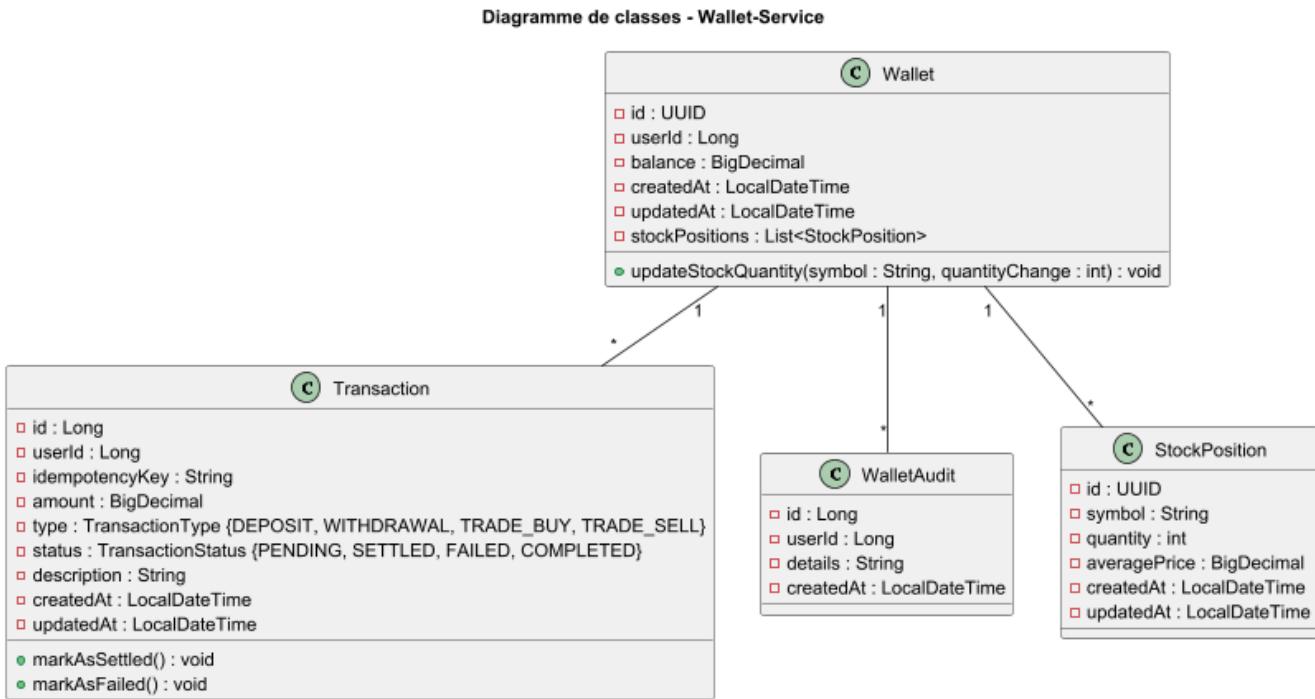


- Notification Service :
 - Order Service :

Diagramme de classes - Order-Service



- Wallet Service :



Contexte

La vue logique présente la structure interne du système, organisée autour de 6 microservices principaux : Auth-Service, Matching-Service, Market-Data-Service, Notification-Service, Order-Service et Wallet-Service. Chaque service possède son propre modèle métier, adapté à ses responsabilités fonctionnelles : gestion des utilisateurs et de l'authentification, appariement des ordres, gestion des données de marché, gestion des notifications envoyées, gestion des transactions et des ordres, gestion des portefeuilles et des positions boursières.

Éléments

- **Auth-Service** : User, MfaChallenge, VerificationToken, UserAudit
- **Matching-Service** : OrderBook, ExecutionReport, OutboxEvent
- **Market-Data-Service** : StockRule
- **Notification-Service** : NotificationLog
- **Order-Service** : Order, OutboxEvent
- **Wallet-Service** : Wallet, StockPosition, Transaction, WalletAudit
- Enums et value objects spécifiques à chaque domaine

Relations

- Les entités de chaque microservice sont liées par des relations métier propres à leur domaine (ex : un Wallet possède des StockPositions et des Transactions ; un User possède des MfaChallenges et des UserAudits ; un Order est audité et typé ; le MatchingEngine gère des Reports et des OrderBooks).
- Les enums et value objects enrichissent la sémantique métier et garantissent la cohérence des règles de gestion.
- Les diagrammes illustrent la séparation stricte des responsabilités entre les microservices, tout en assurant l'intégrité des processus métier globaux.

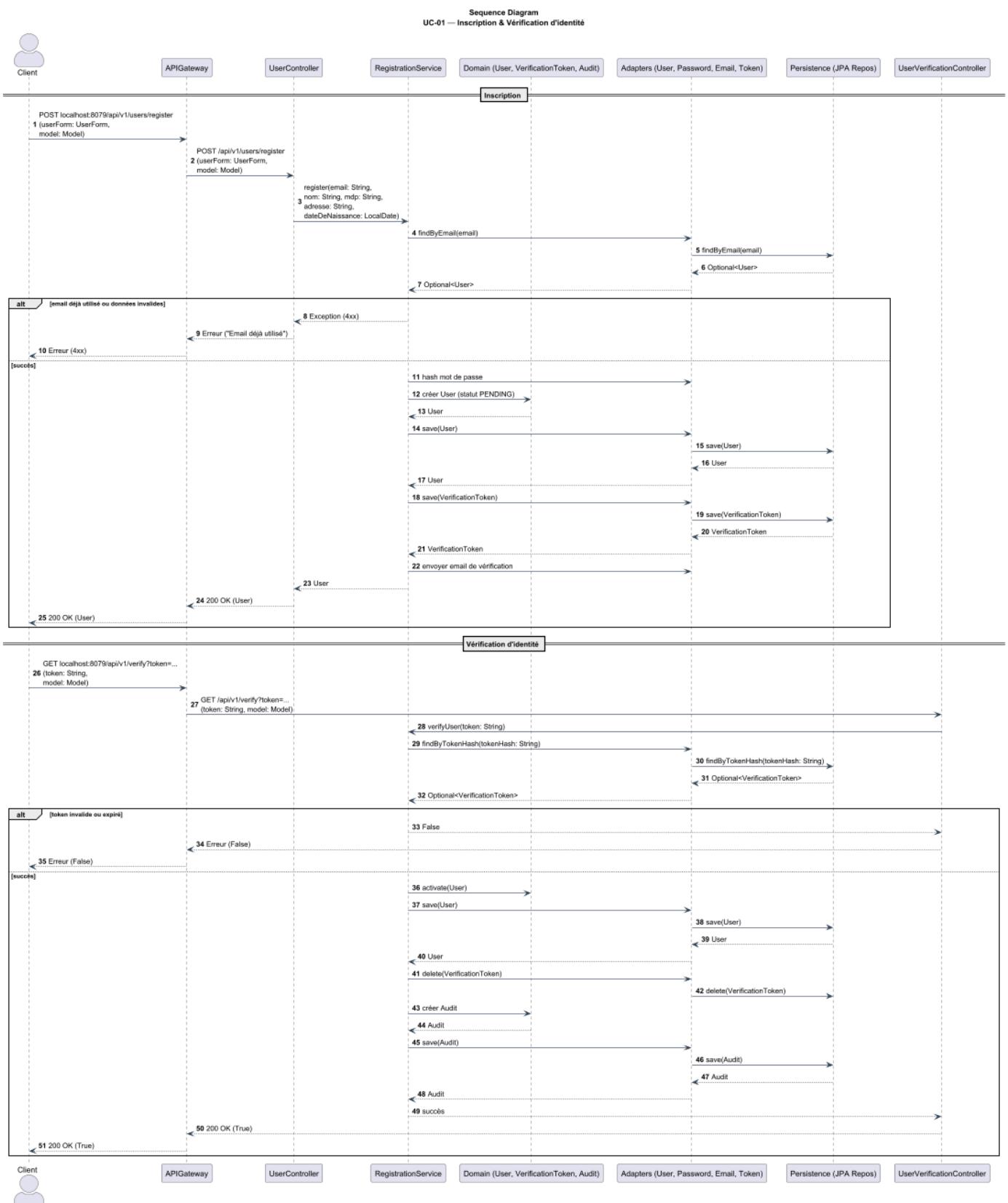
Rationnel

Cette vue permet de comprendre la logique métier profonde du système, d'assurer la cohérence du modèle et de faciliter la maintenance et l'évolution du code. La séparation en microservices, chacun doté de son propre modèle, favorise la robustesse, la scalabilité et l'évolutivité. Les diagrammes de classes détaillent les dépendances et les relations entre les entités, ce qui aide à anticiper les impacts des évolutions fonctionnelles et à garantir la solidité du système. Cette documentation est essentielle pour la formation des nouveaux développeurs, la validation des règles métier et la communication avec les parties prenantes. Elle pourra également servir dans le futur quand il faudra ajouter de nouveaux microservices, car on connaîtra déjà le contexte des autres microservices.

9. Vue Processus (C&C)

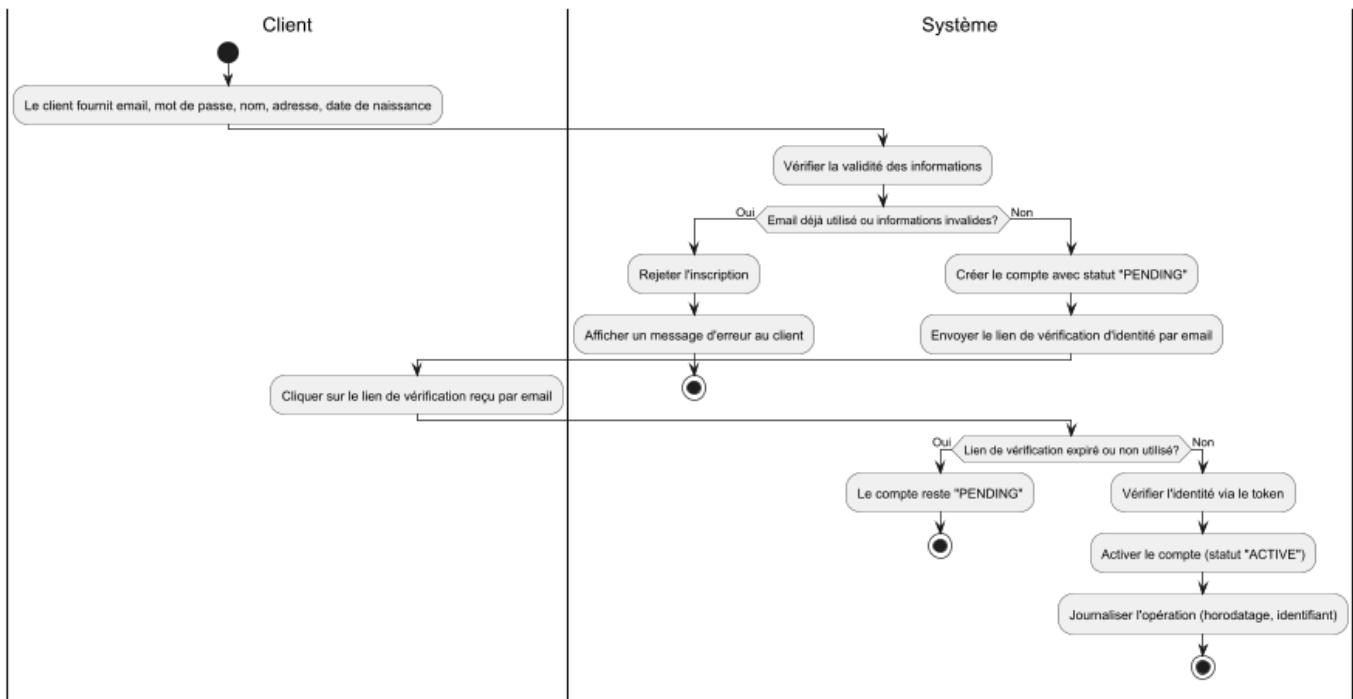
Diagrammes

UC01 — Séquence

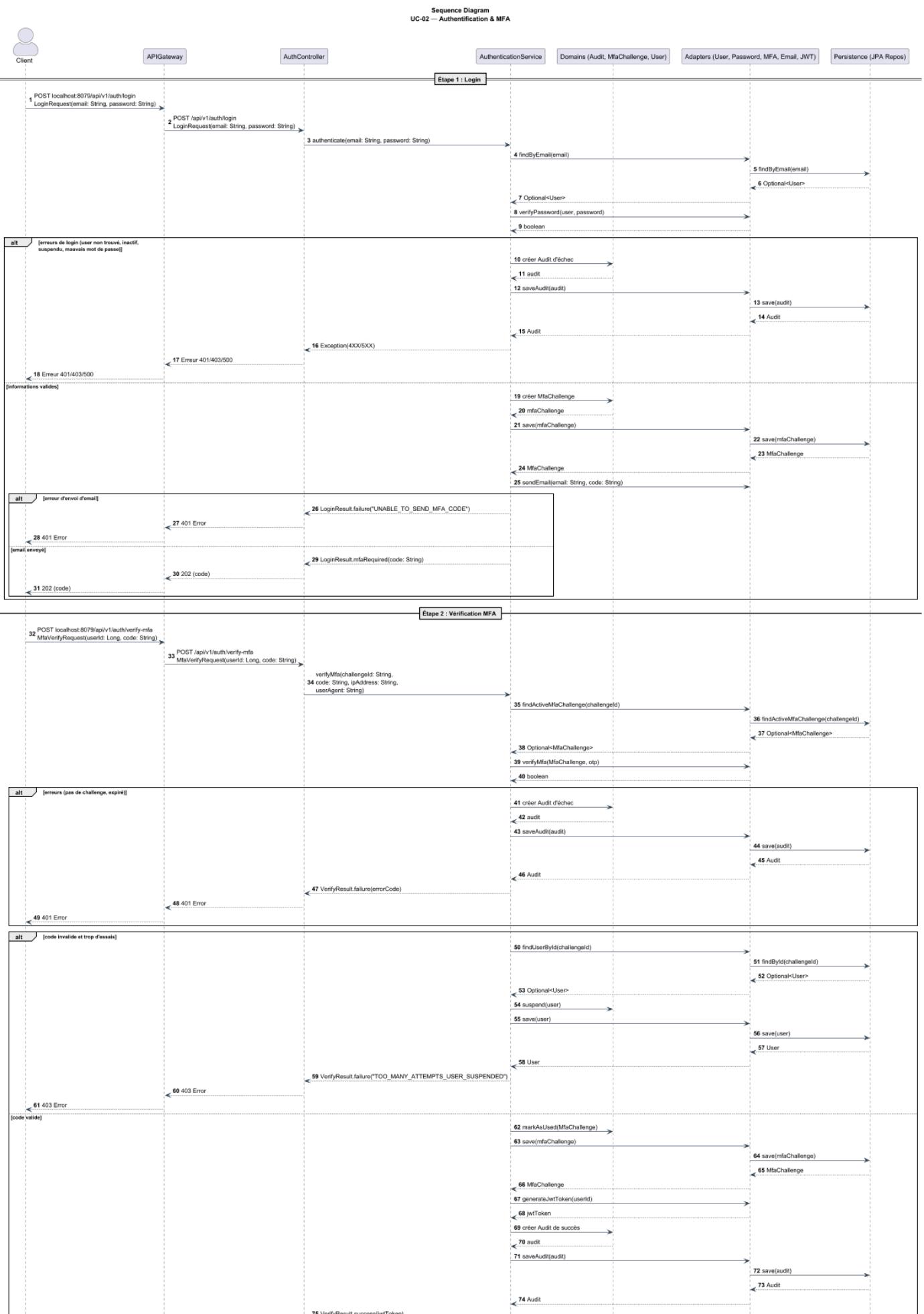


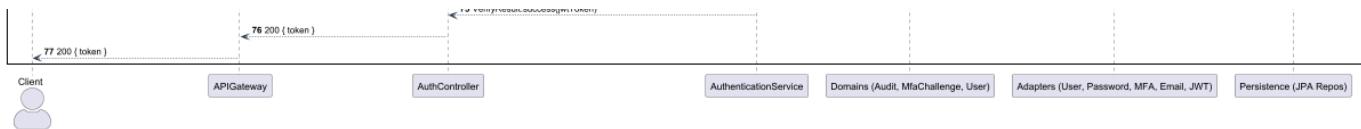
UC01 — Activité

Diagramme d'activité
UC-01 — Inscription & Vérification d'identité

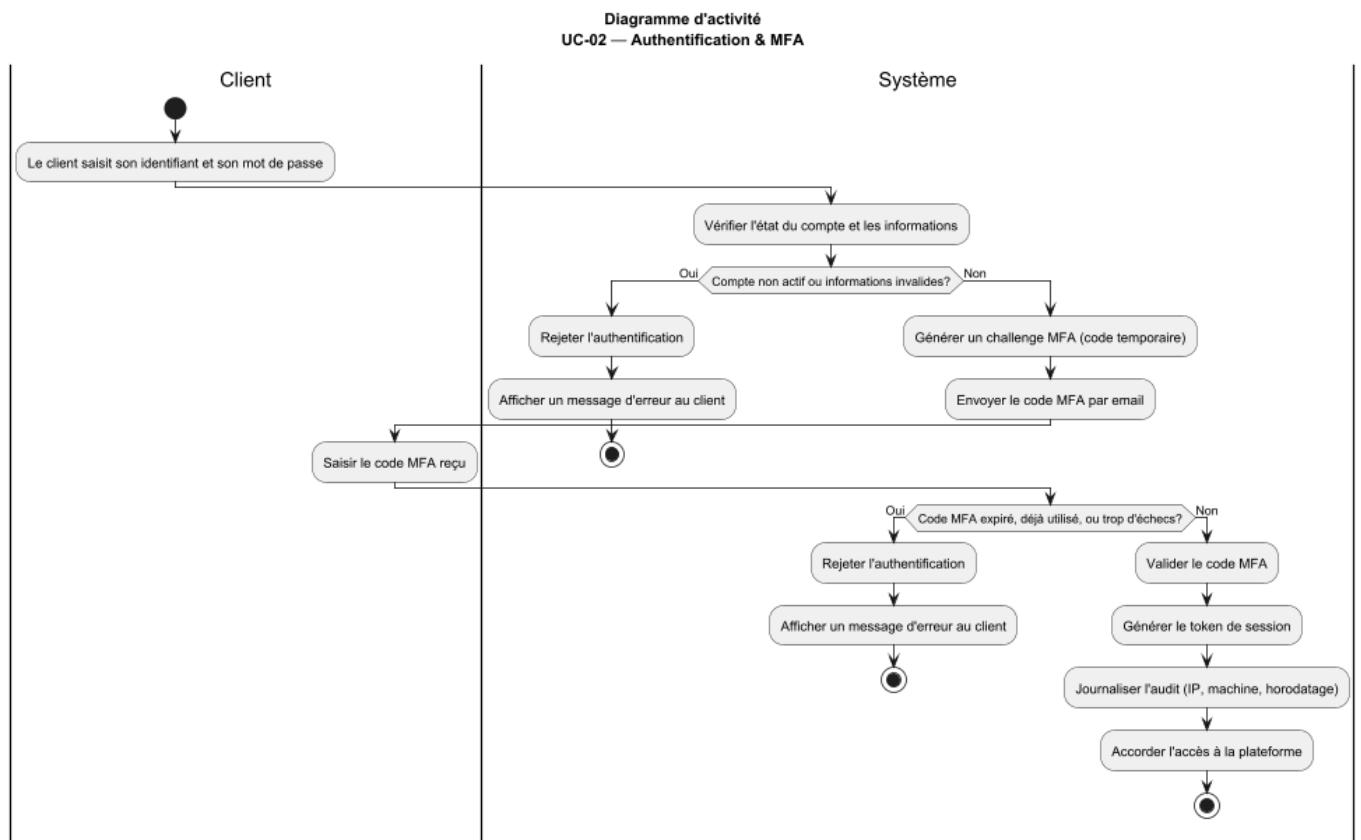


UC02 — Séquence





UC02 — Activité

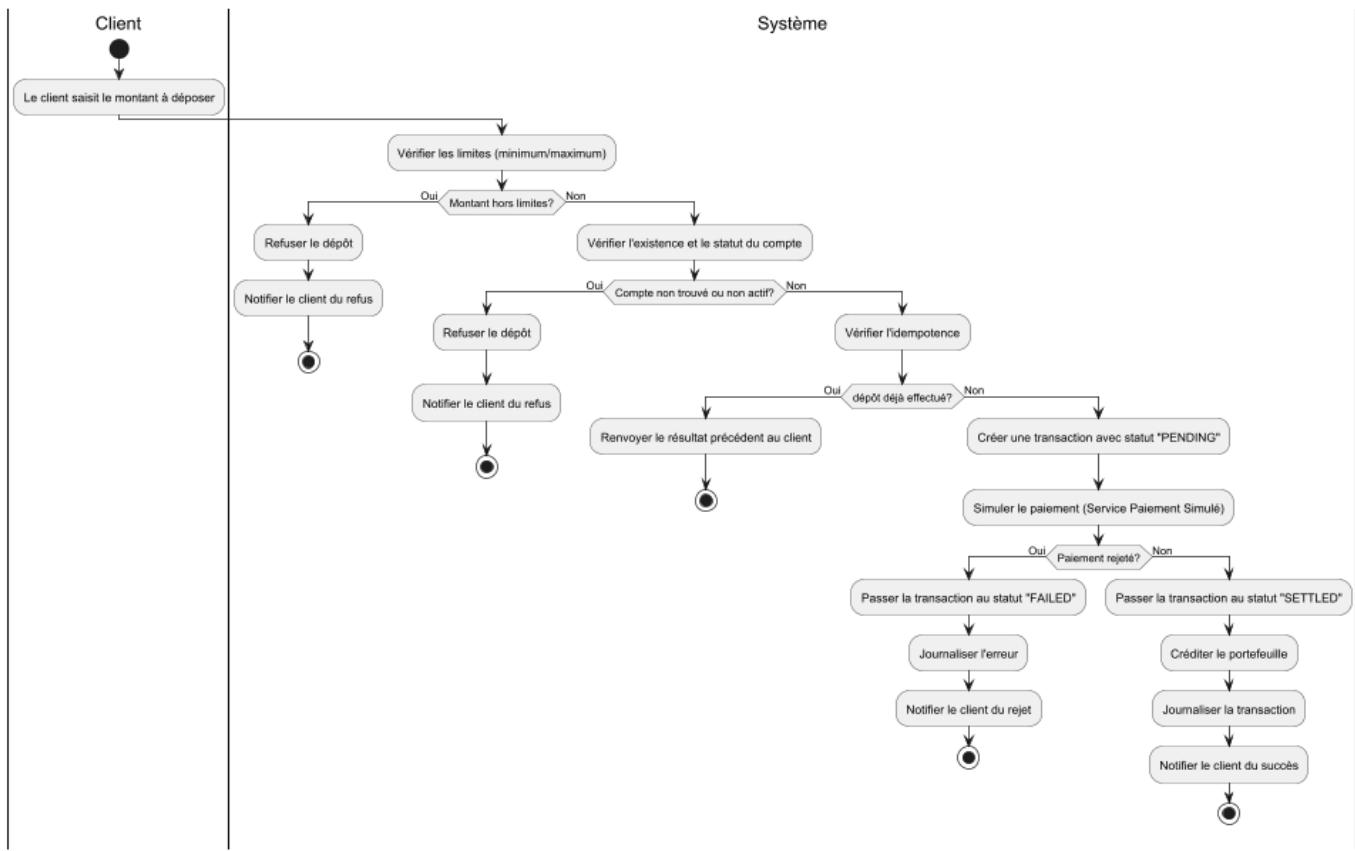


UC03 — Séquence

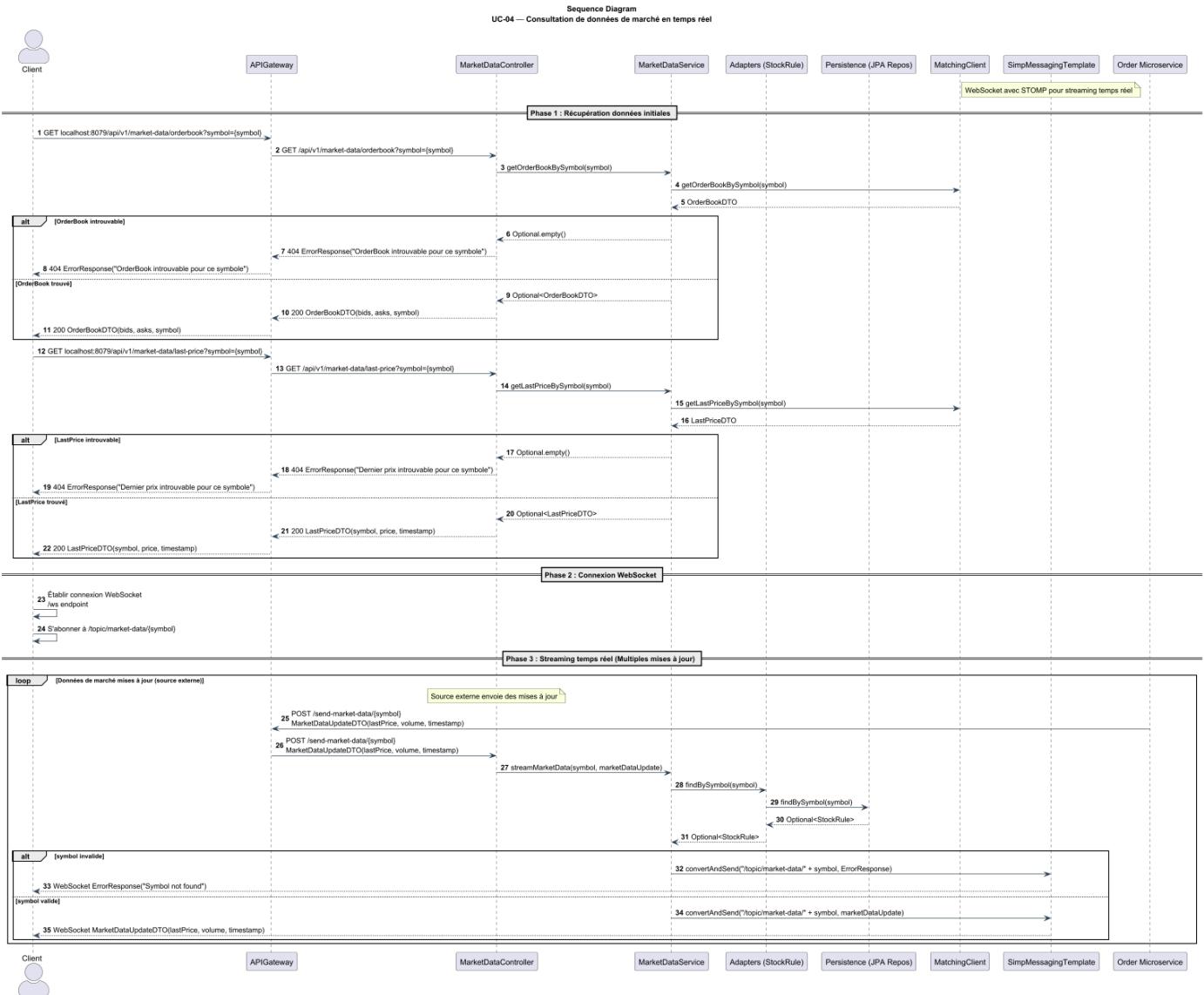


UC03 — Activité

Diagramme d'activité
UC-03 — Approvisionnement du portefeuille

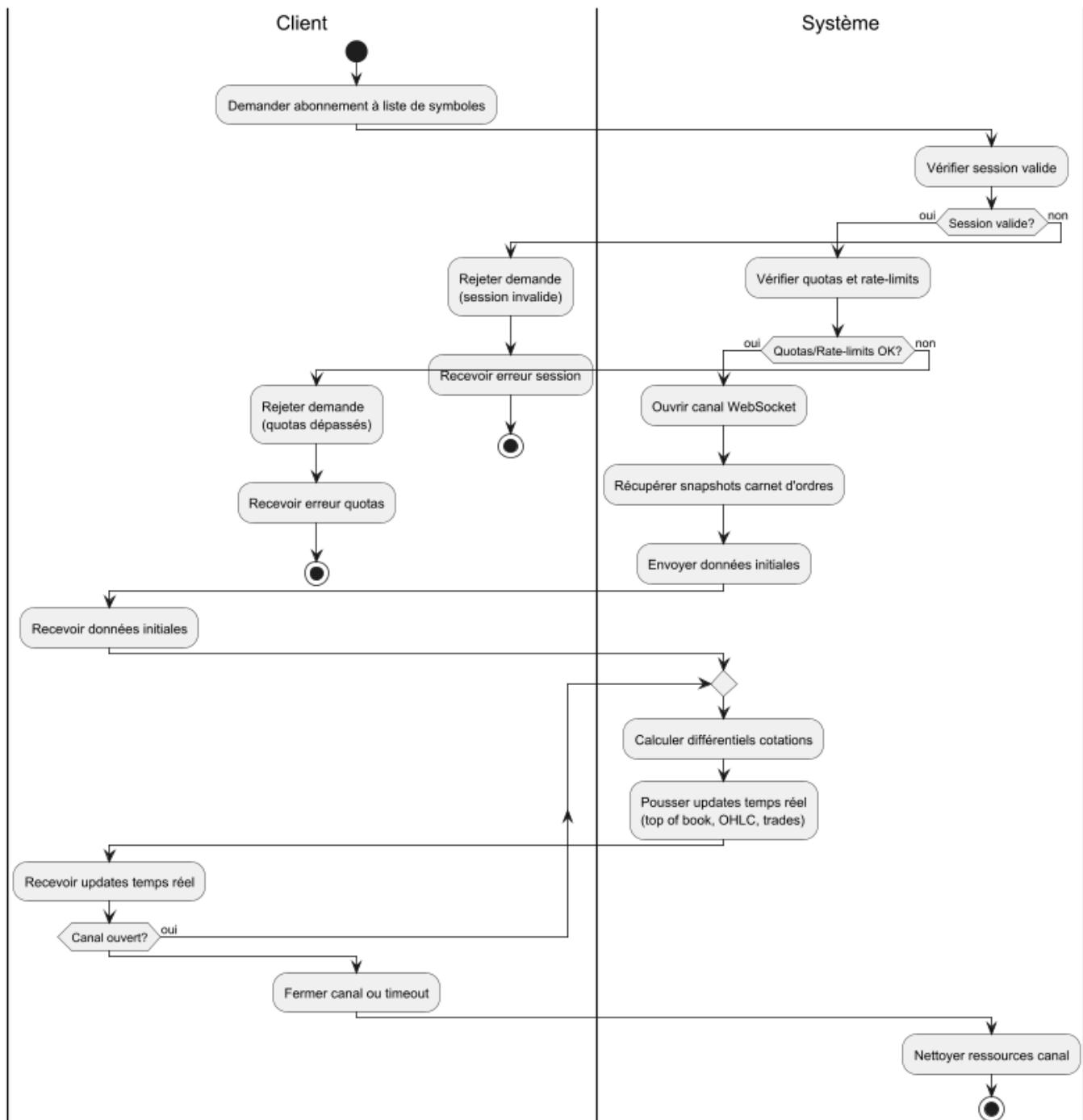


UC04 — Séquence

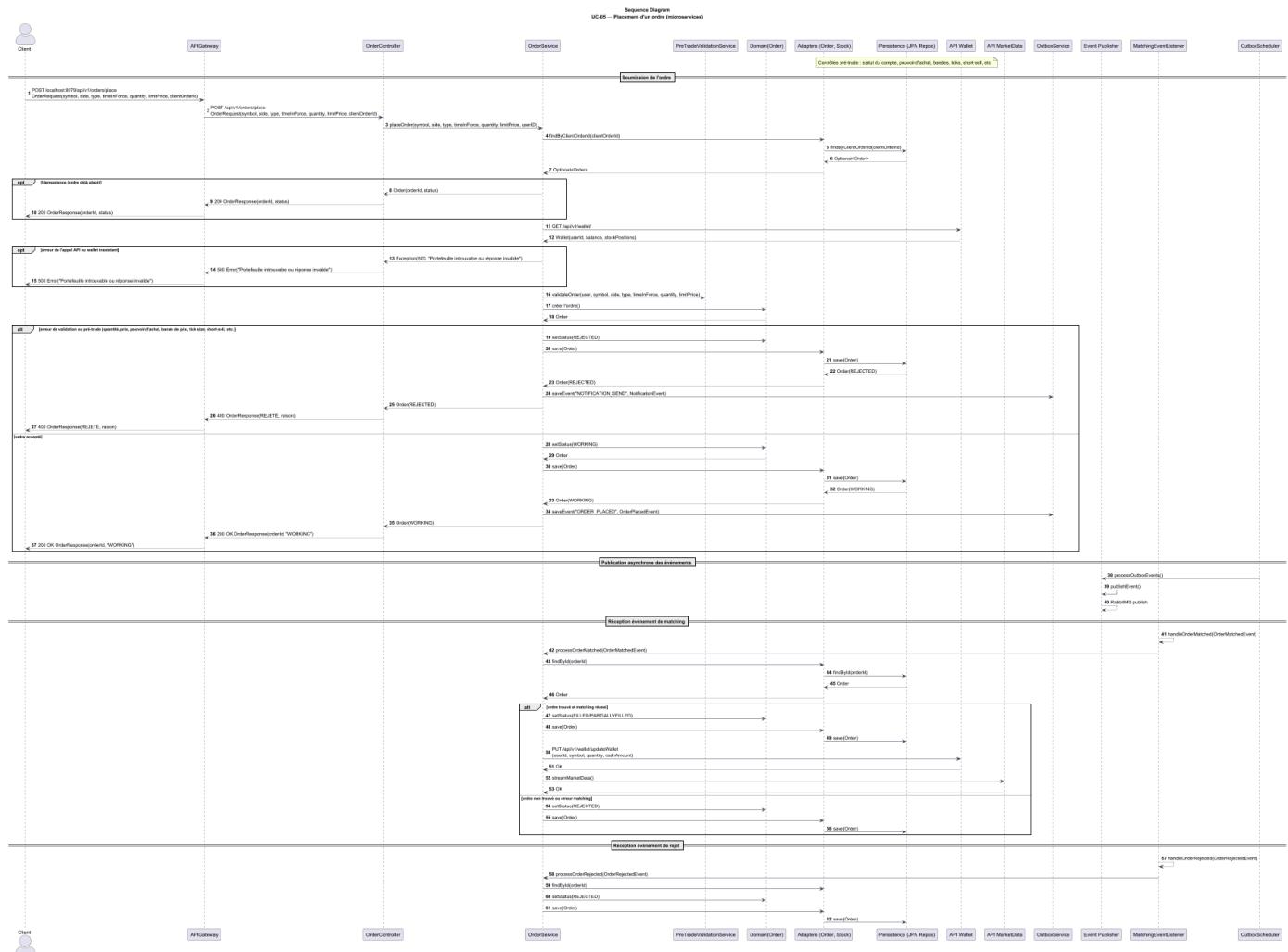


UC04 — Activité

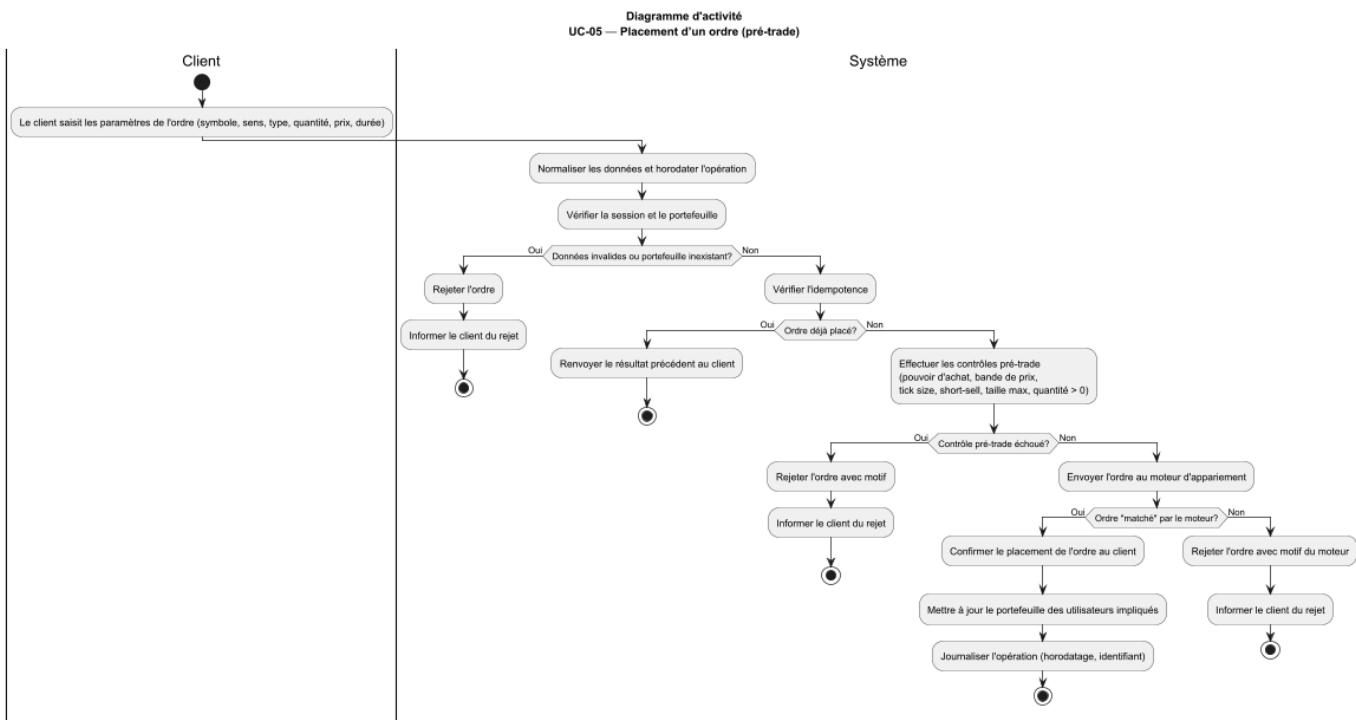
Diagramme d'activité UC-04
Abonnement aux données de marché



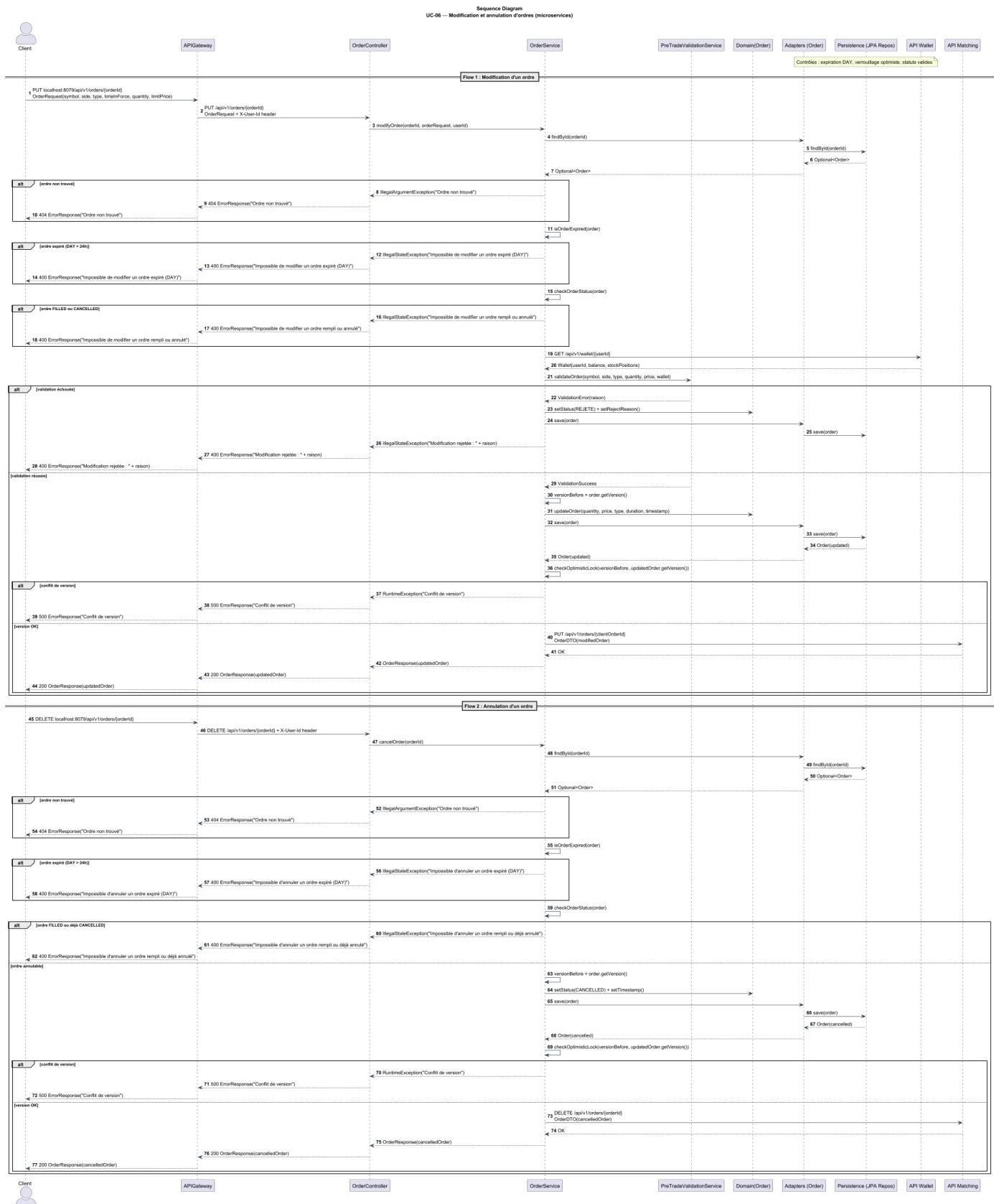
UC05 — Séquence



UC05 — Activité

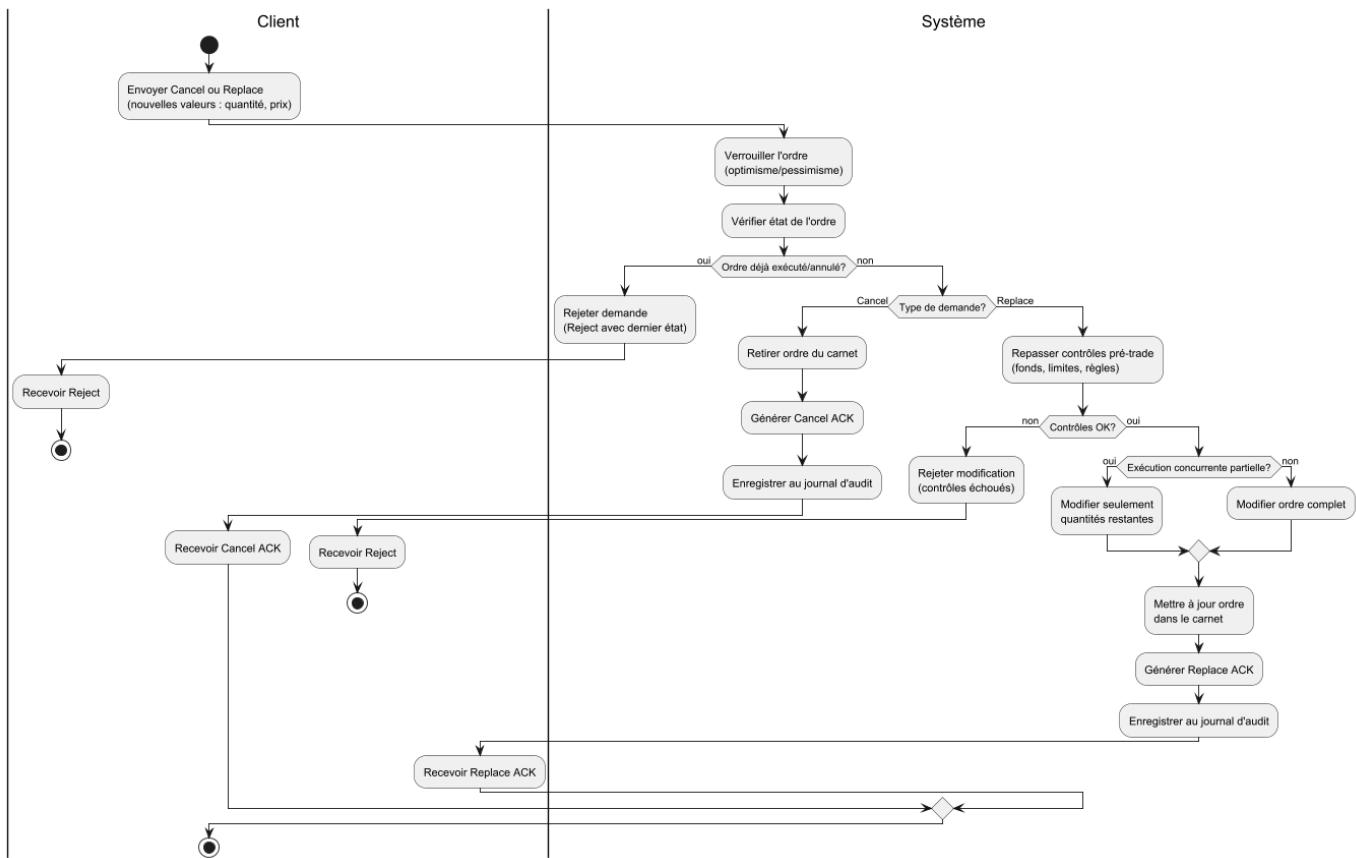


UC06 — Séquence

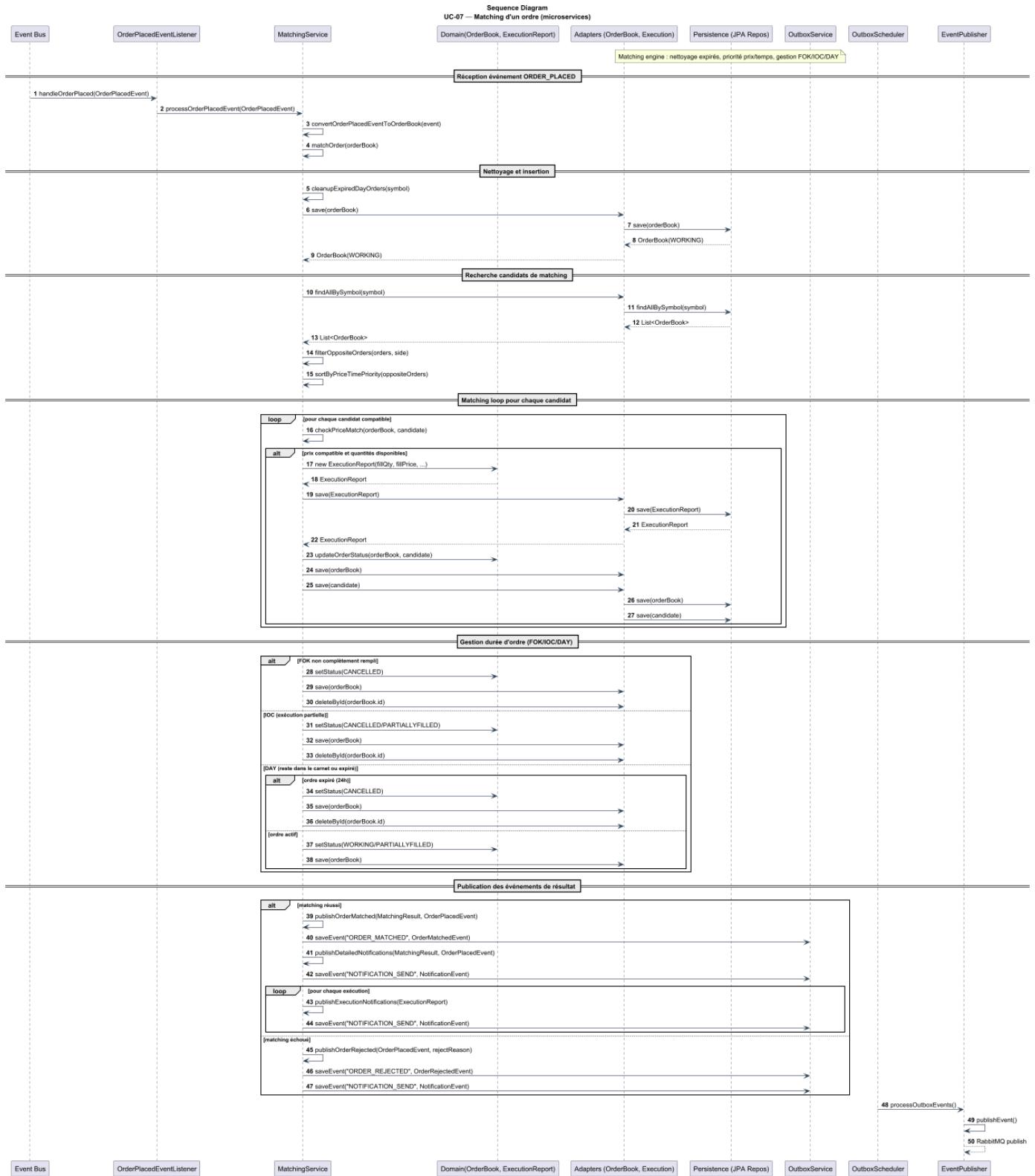


UC06 — Activité

Diagramme d'activité UC-06
Modification / Annulation d'un ordre

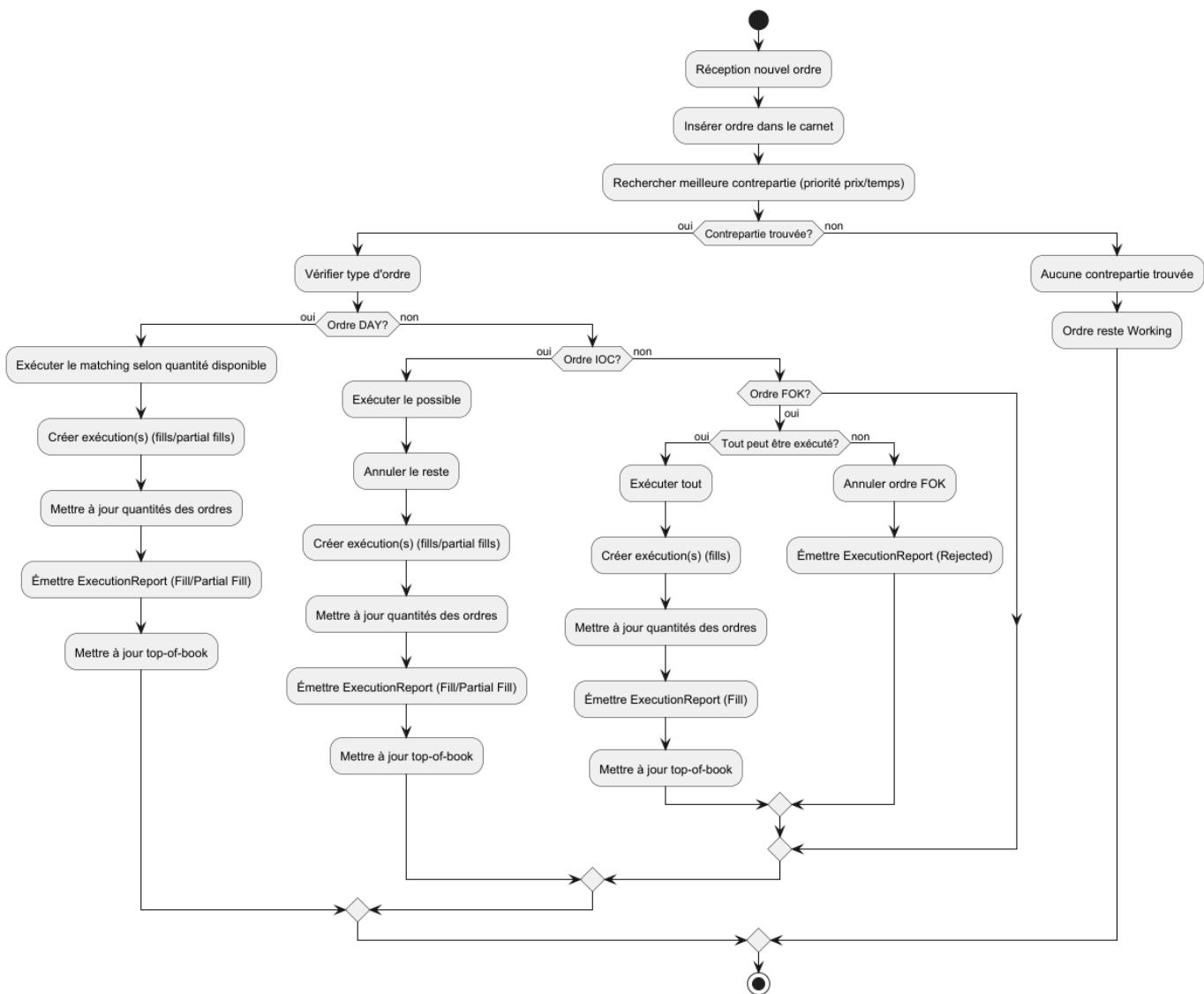


UC07 — Séquence

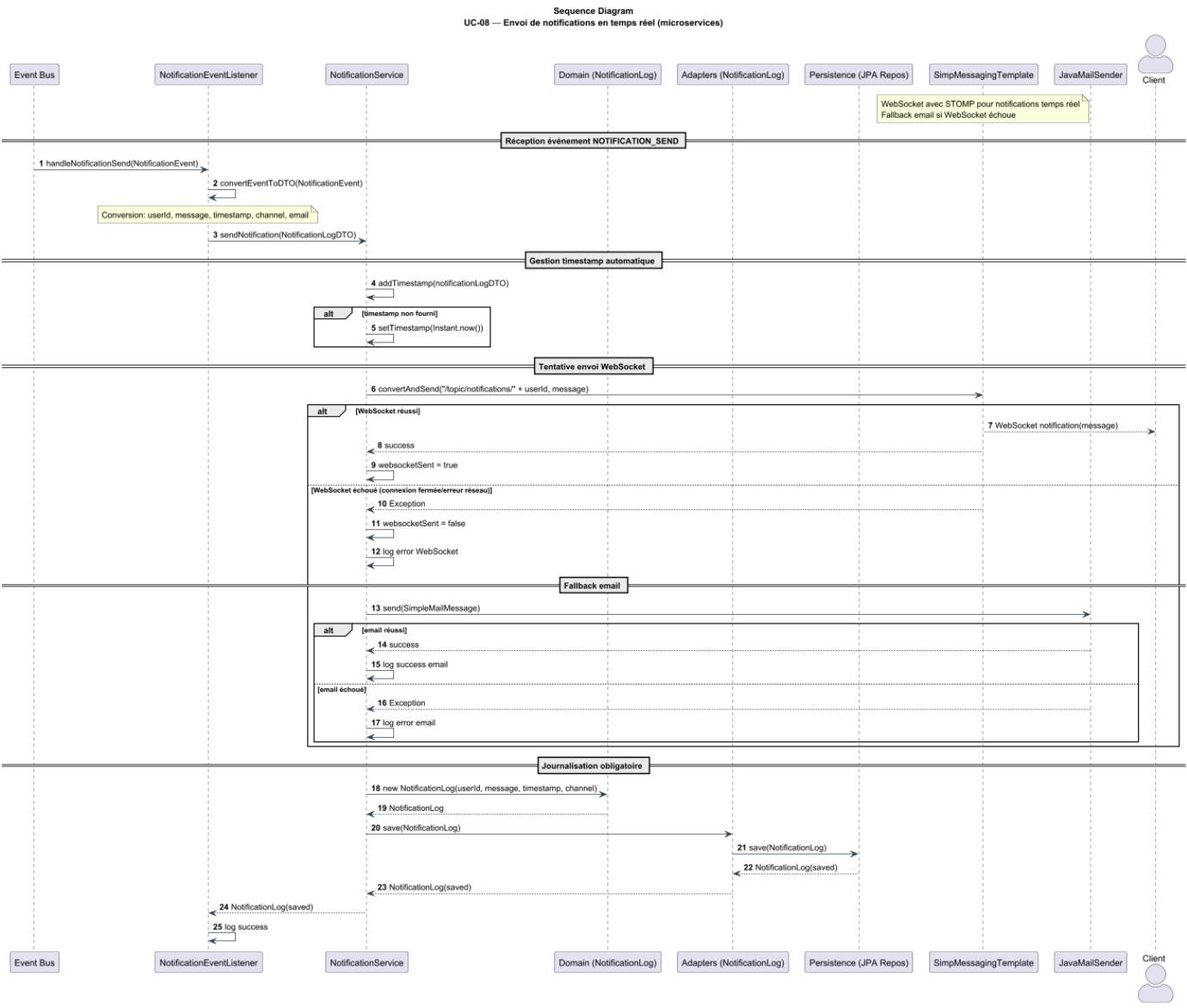


UC07 — Activité

Diagramme d'activité
UC-07 — Appariement interne & Exécution (matching)



UC08 — Séquence



UC08 — Activité

Diagramme d'activité UC-08
Confirmation d'exécution & Notifications



Contexte

La vue processus détaille le comportement dynamique du système lors de l'exécution des cas d'utilisation. Elle montre comment les composants collaborent pour réaliser les opérations métier, gérer les erreurs et orchestrer les interactions.

Éléments

- Services applicatifs (RegistrationService, AuthenticationService, WalletDepositService, OrderService, MatchingService, MarketDataService, NotificationService)
- Contrôleurs web (UserVerificationController, AuthController, WalletController, OrderController, MatchingController, MarketDataController, NotificationController, WebSocketController)
- Moteur d'appariement interne (MatchingService avec architecture événementielle)
- Listeners événementiels (NotificationEventListener, MatchingEventListener, OrderPlacedEventListener)
- Publishers événementiels (EventPublisher, MatchingEventPublisher)
- Adapters (UserAdapter, TransactionAdapter, OrderAdapter, NotificationAdapter, etc.)
- Persistance (JPA Repos avec tables outbox pour patterns Saga/Outbox)
- Message Broker (RabbitMQ pour orchestration événementielle)
- Acteurs externes (Client, Service Email SMTP)

Relations

- Les contrôleurs reçoivent les requêtes des clients et délèguent aux services
- Les services orchestrent la logique métier et interagissent avec les adapters et publishers événementiels
- Les listeners événementiels consomment les messages RabbitMQ et déclenchent les traitements appropriés
- Les publishers événementiels publient les événements vers RabbitMQ via les patterns Saga et Outbox
- Les adapters font le lien avec la persistance, les systèmes externes et les tables outbox
- RabbitMQ orchestre les workflows distribués (matching d'ordres, notifications, mises à jour de portefeuille)
- Les diagrammes d'activité synthétisent les étapes clés, les alternatives et les flux événementiels

Rationnel

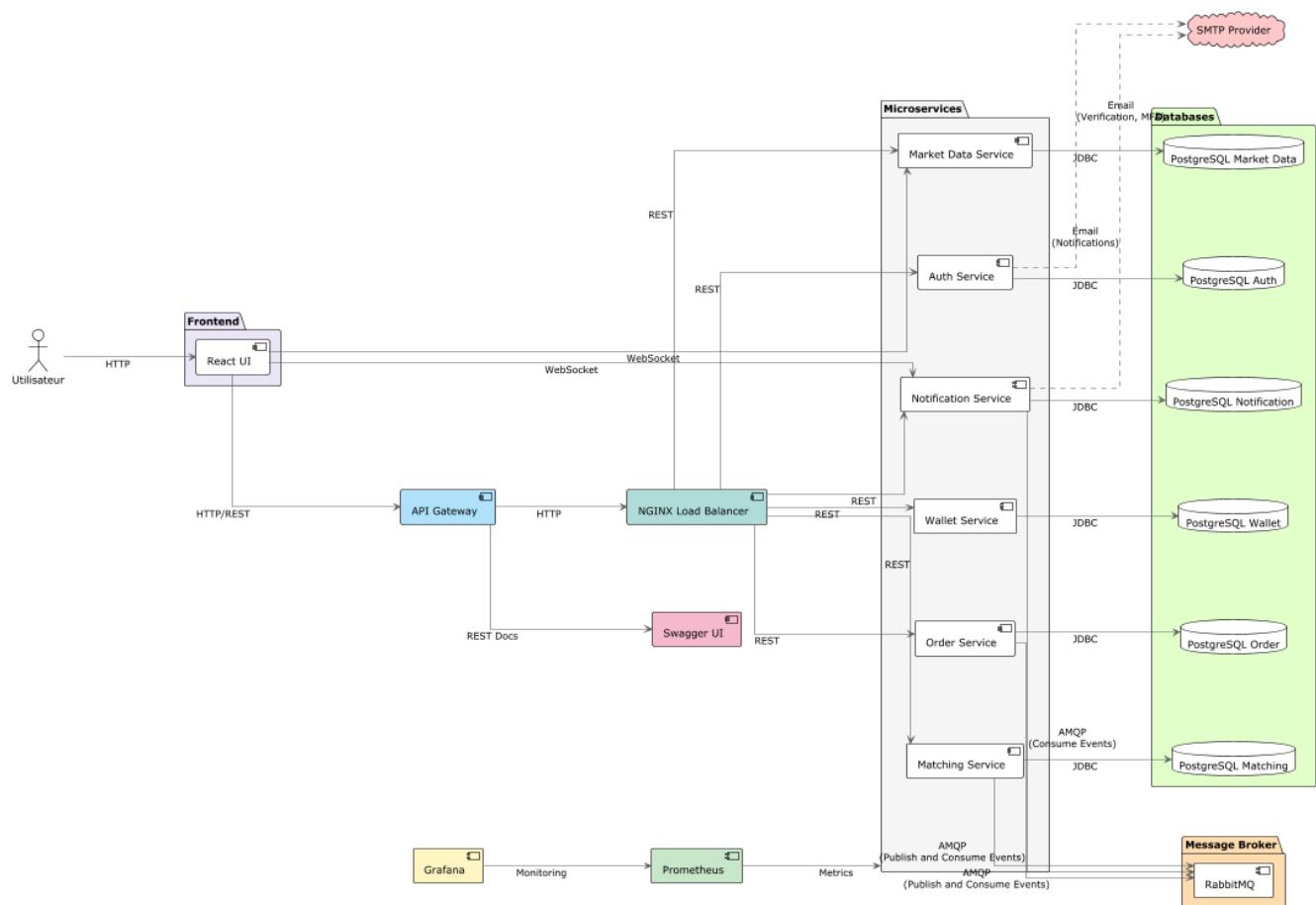
Cette vue permet de visualiser le flow des opérations synchrones et asynchrones, la gestion des exceptions, l'idempotence et la coordination entre les modules dans une architecture événementielle. Elle est essentielle pour valider la robustesse, la sécurité et la performance du système lors des opérations critiques distribuées. Elle aide à identifier les points de synchronisation événementielle, les risques de concurrence dans les workflows Saga, et à optimiser la répartition des responsabilités entre microservices. Elle est aussi précieuse pour l'analyse des scénarios d'erreur dans les transactions distribuées, la traçabilité des événements et des actions, et la préparation des tests d'intégration événementielle et de non-régression des patterns Outbox.

10. Vue Développement

Diagrammes de composants

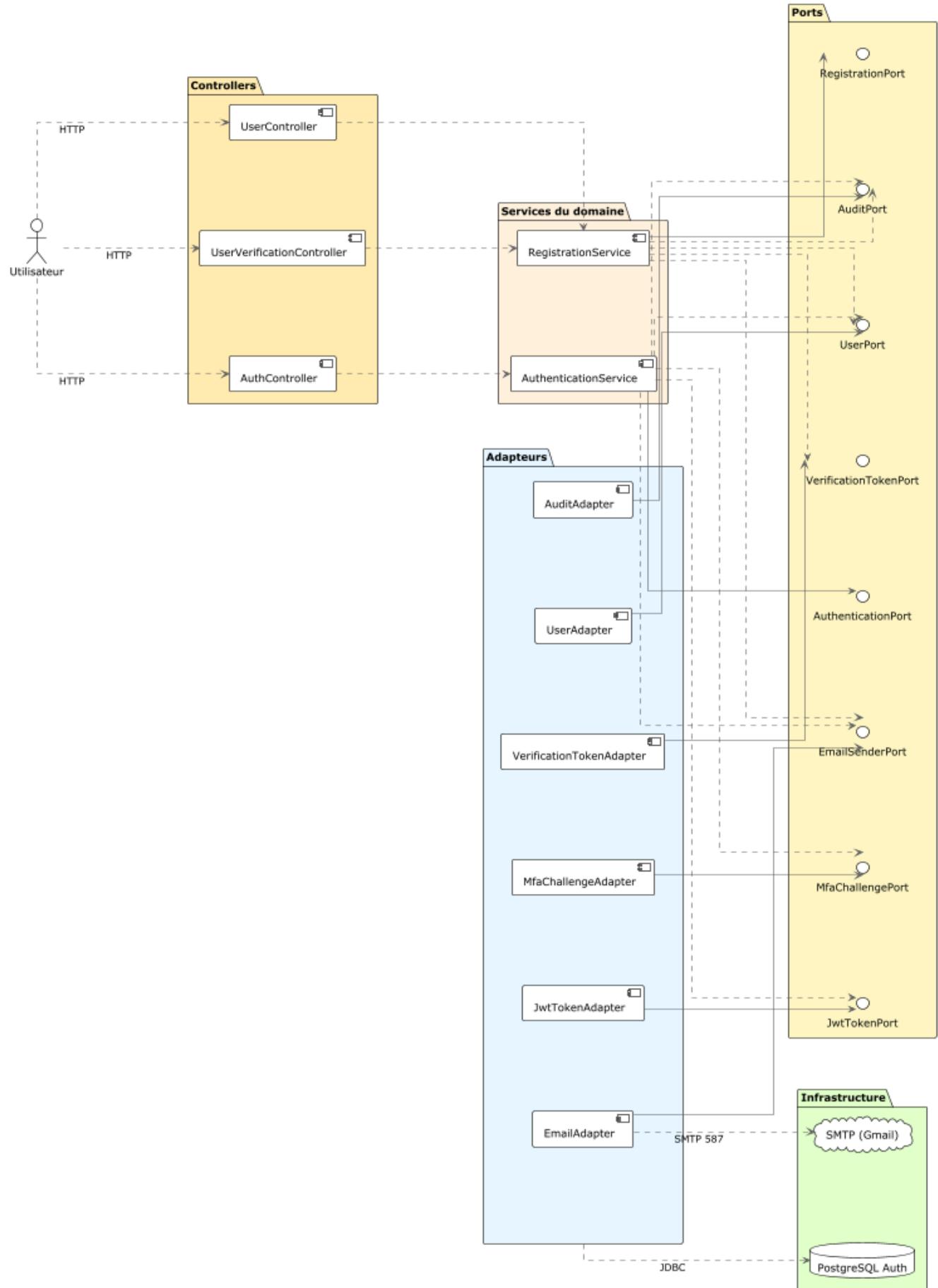
- Diagramme de composants global :

BrokerX - Global Component Diagram (Phase 3 - Architecture Événementielle)

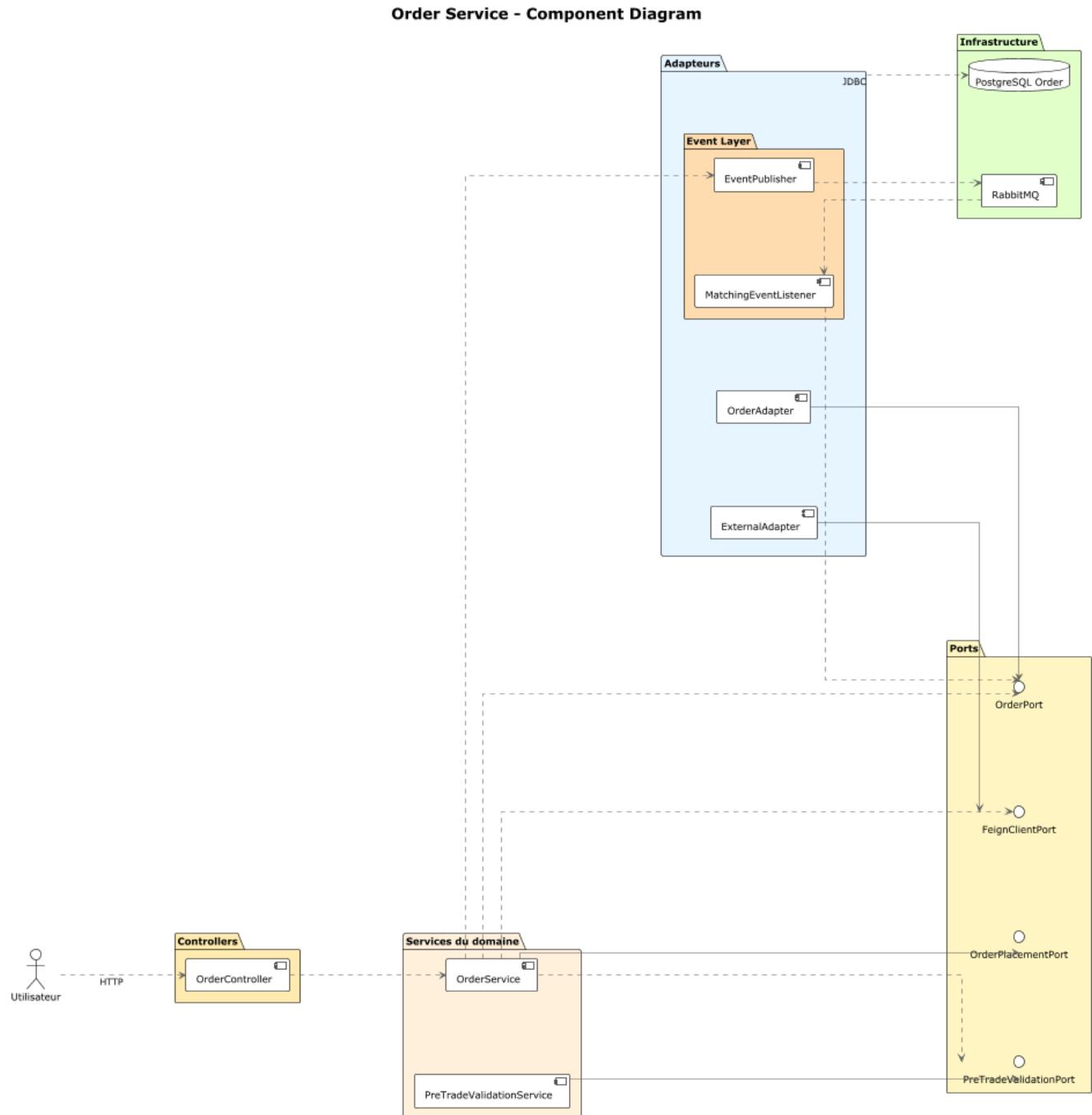


- Diagramme de composants auth-service :

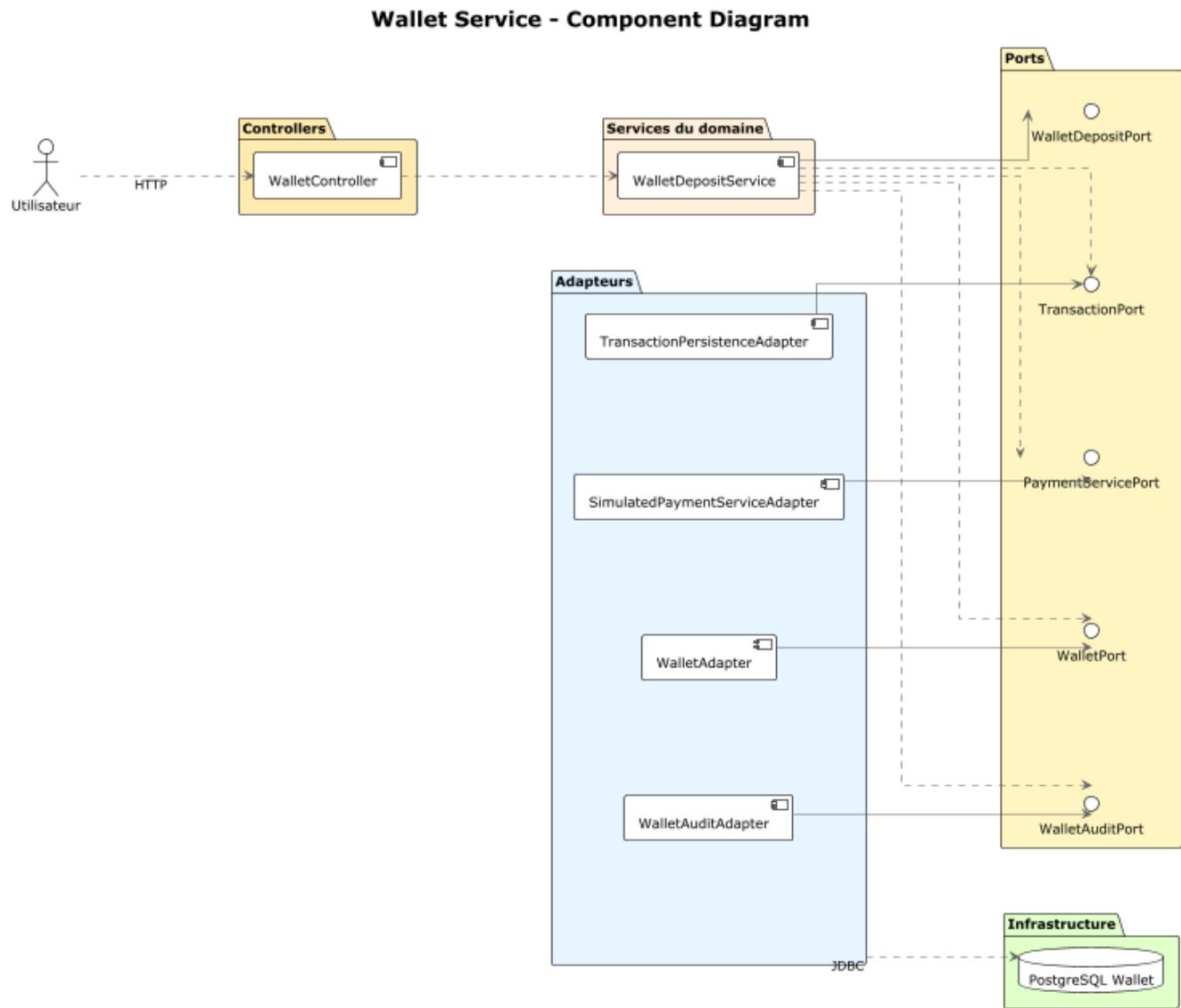
Auth Service - Component Diagram



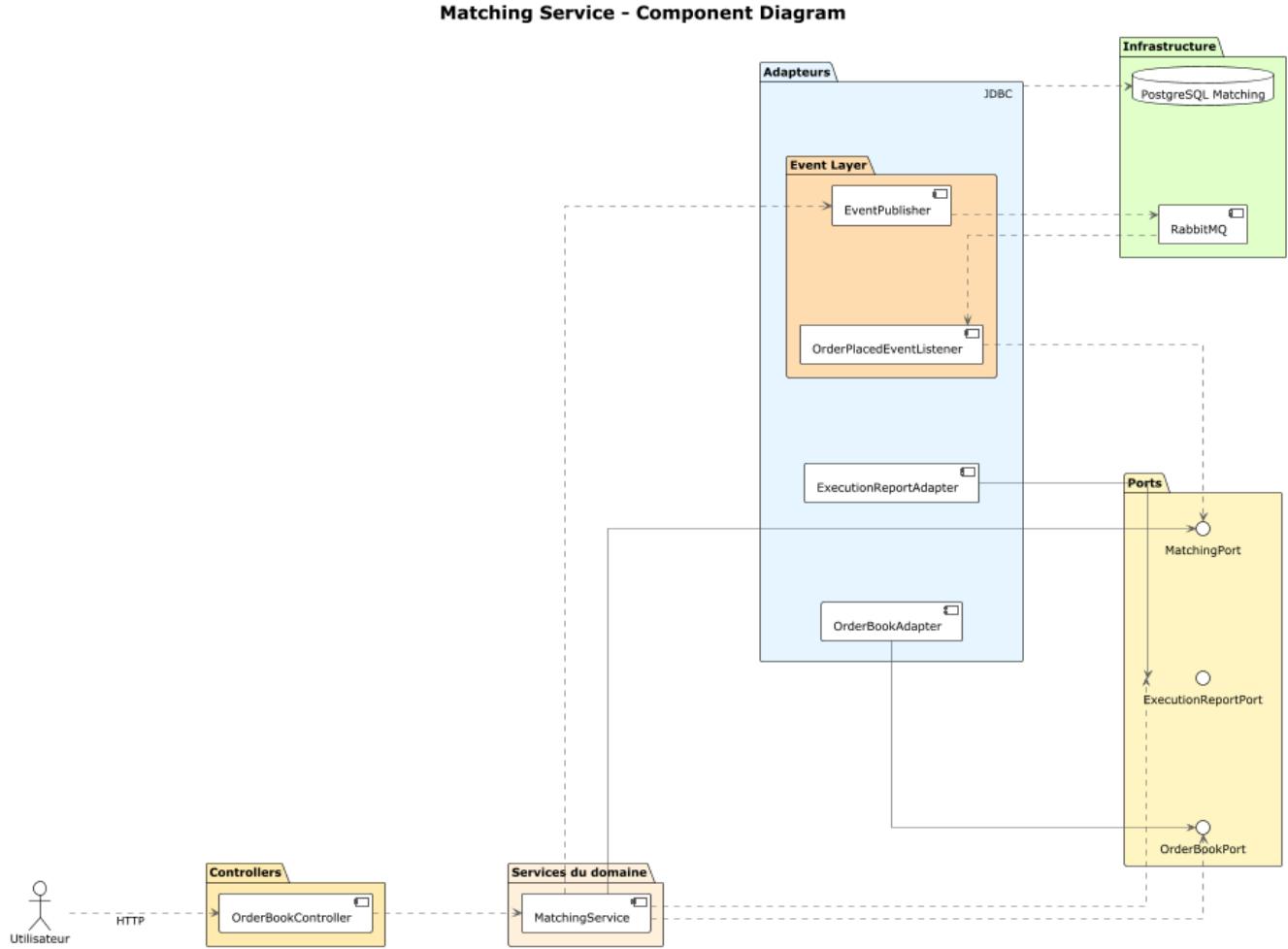
- Diagramme de composants order-service :



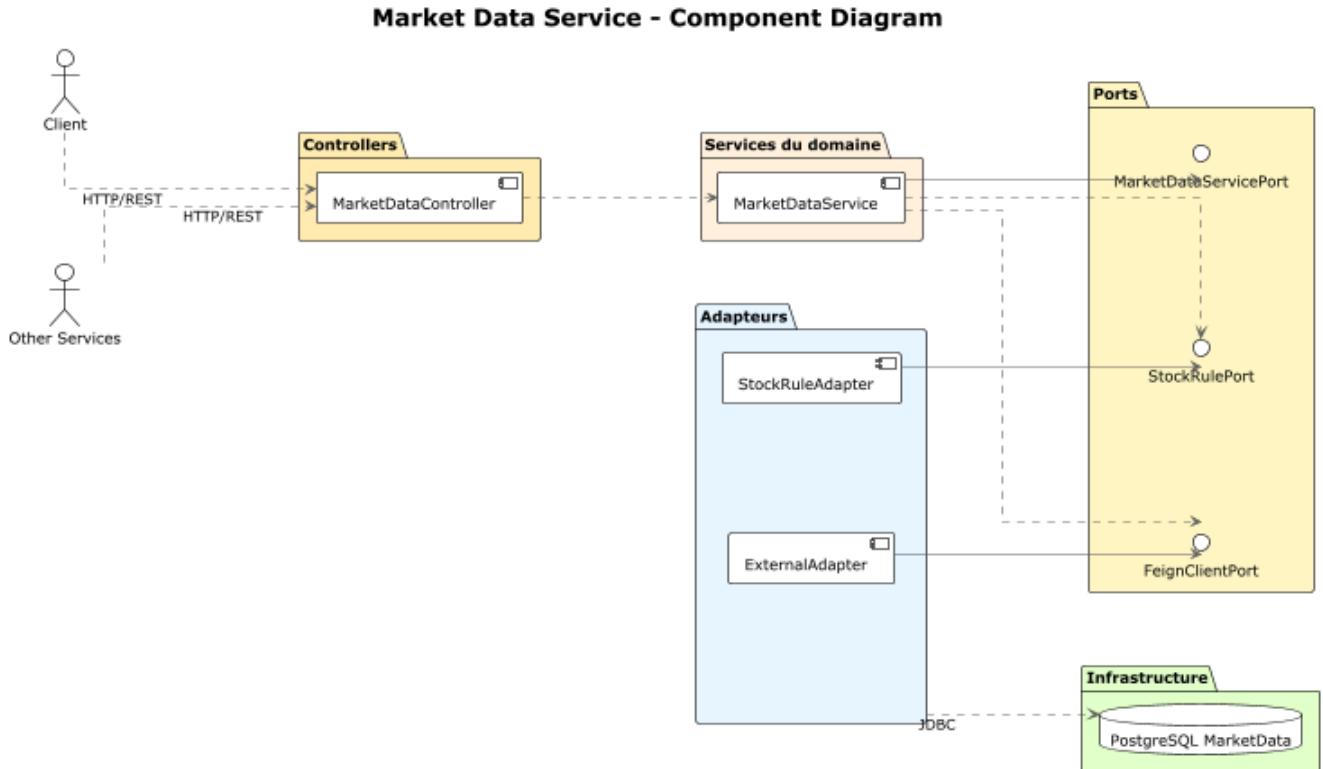
- Diagramme de composants wallet-service :



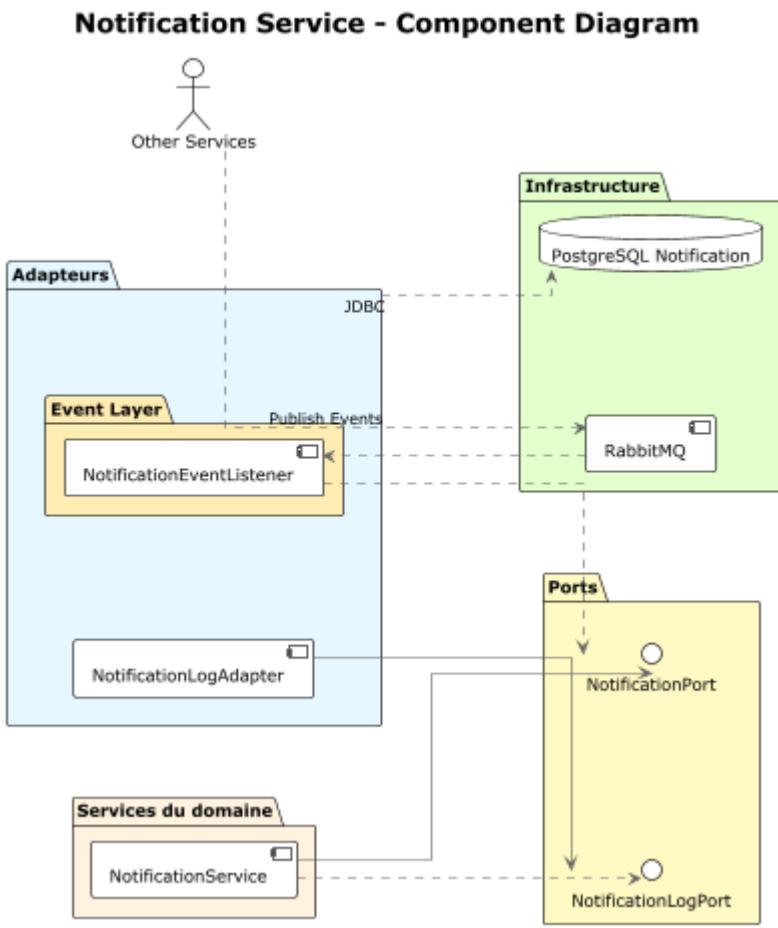
- Diagramme de composants matching-service :



- Diagramme de composants market-data-service :

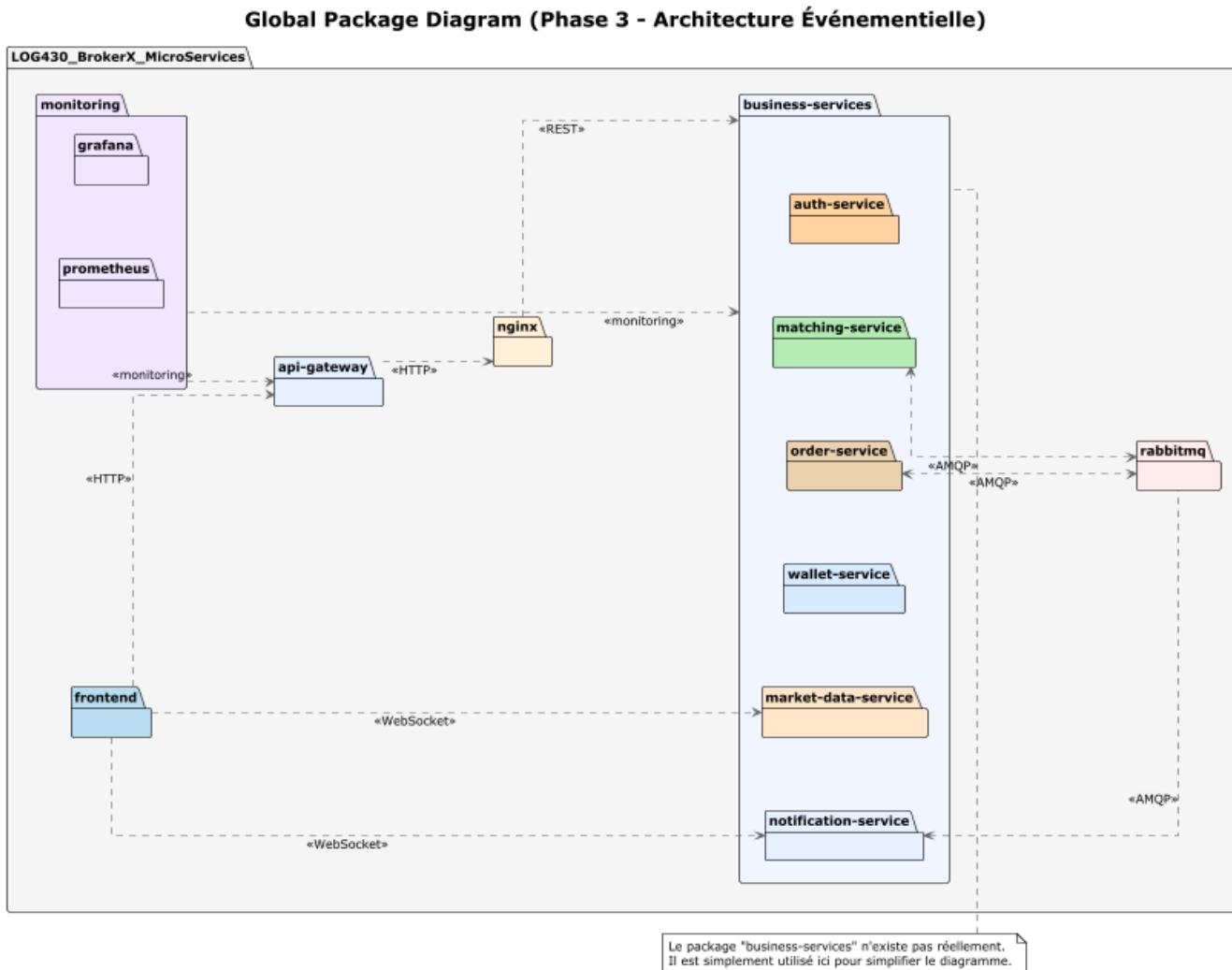


- Diagramme de composants notification-service :



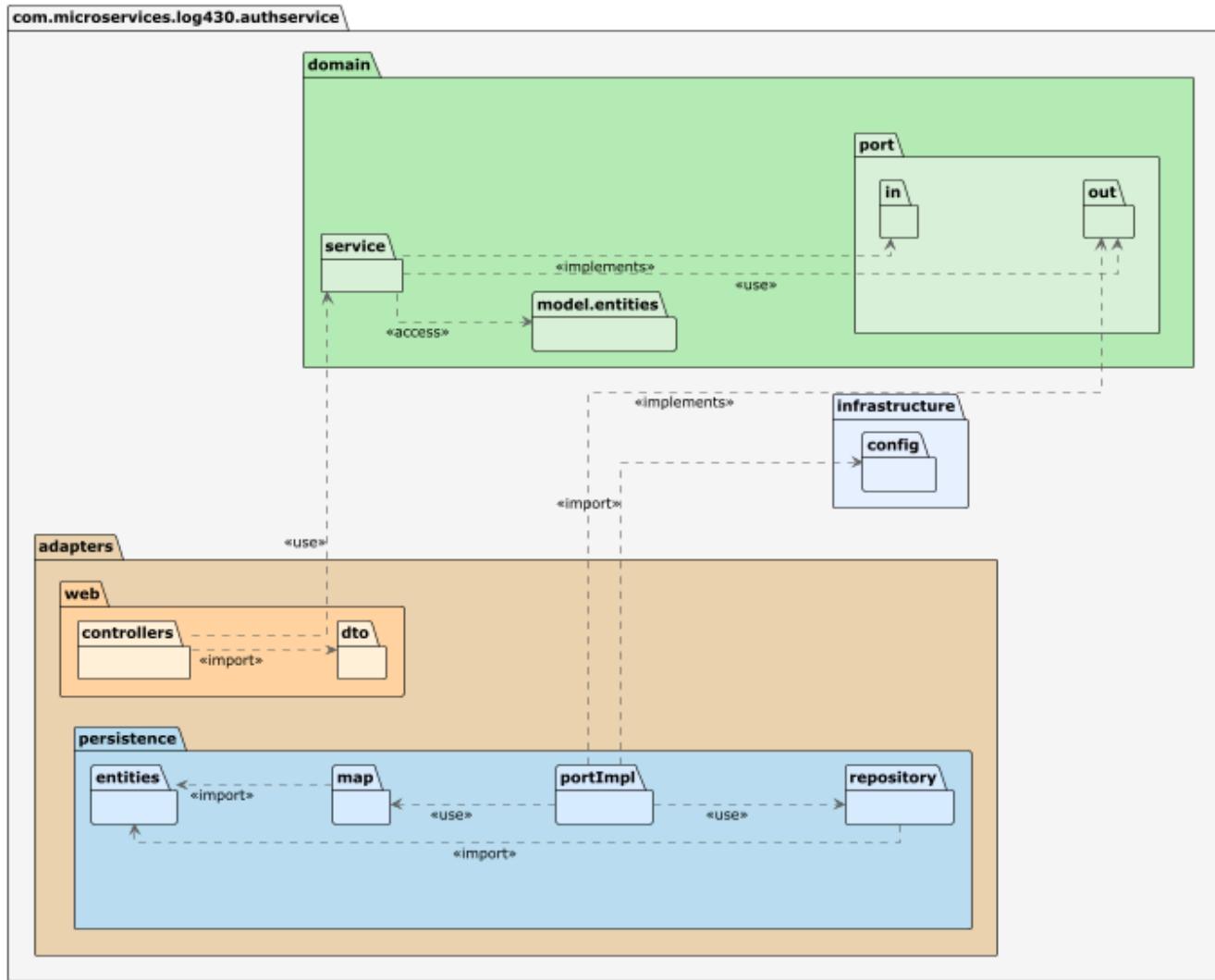
Diagrammes de packages

- Diagramme de packages global :



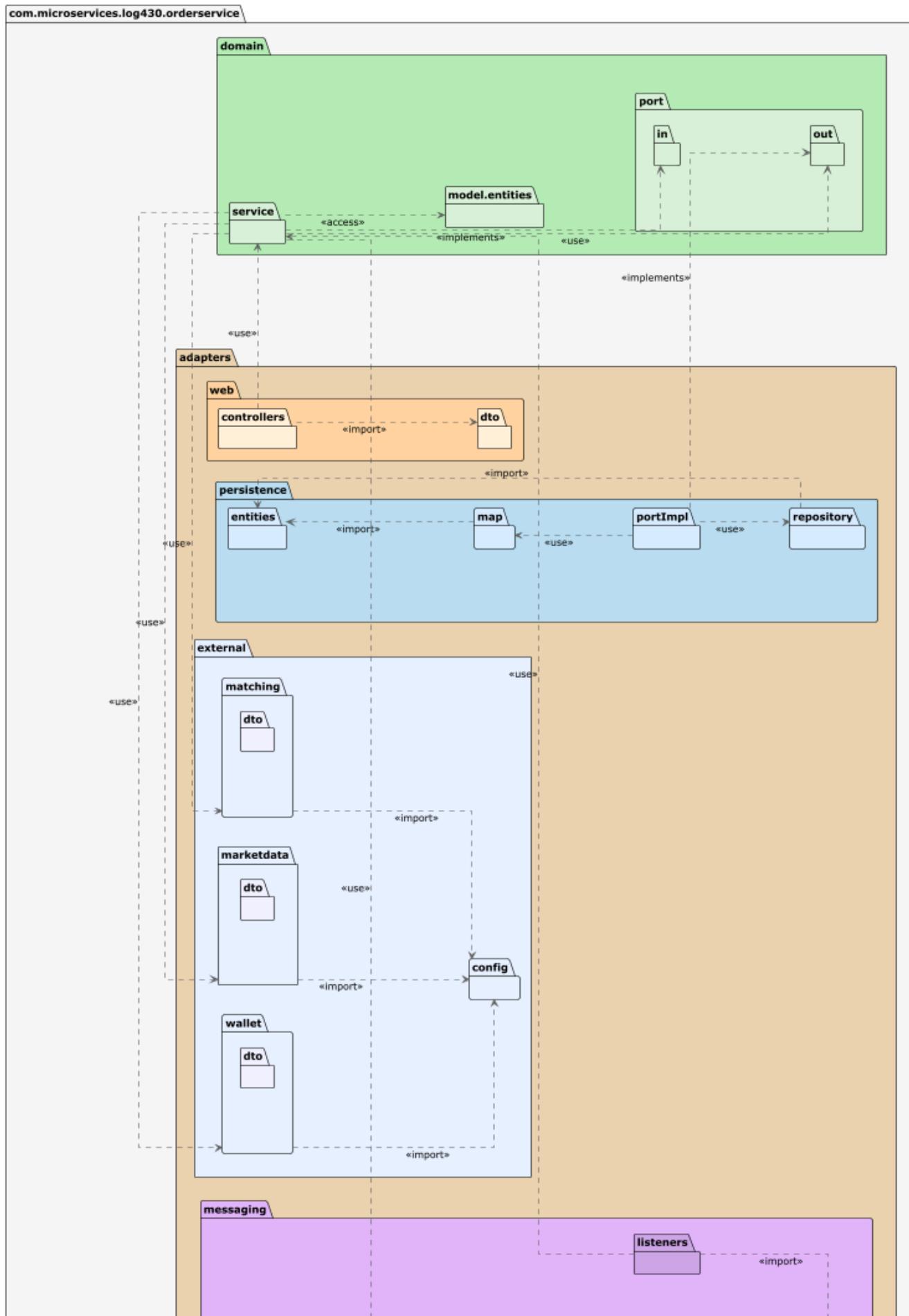
- Diagramme de packages auth-service :

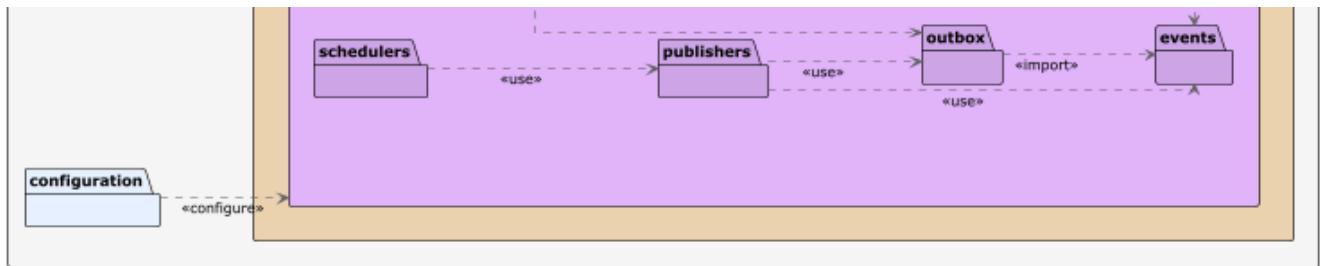
Auth Service - Package Diagram



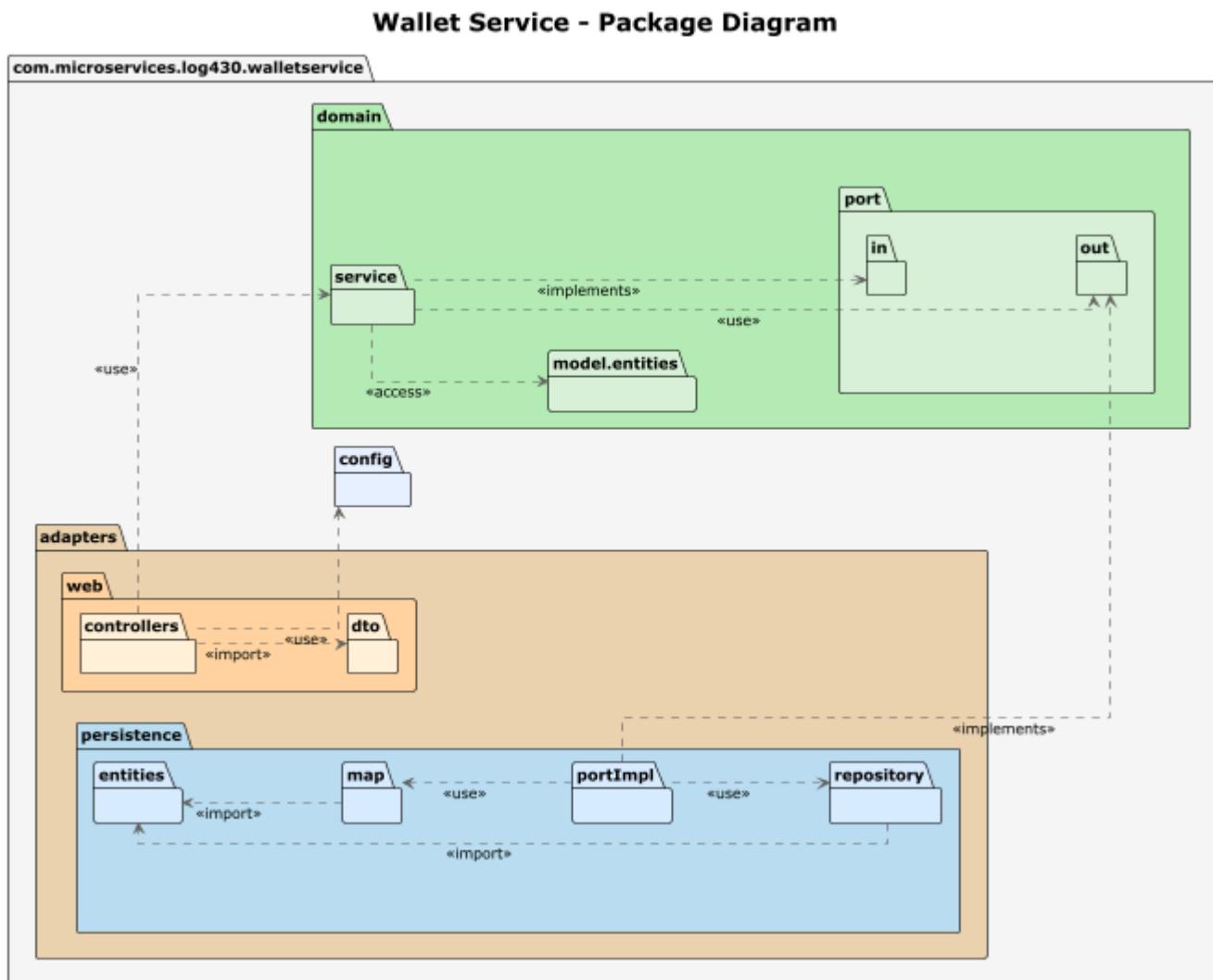
- Diagramme de packages order-service :

Order Service - Package Diagram



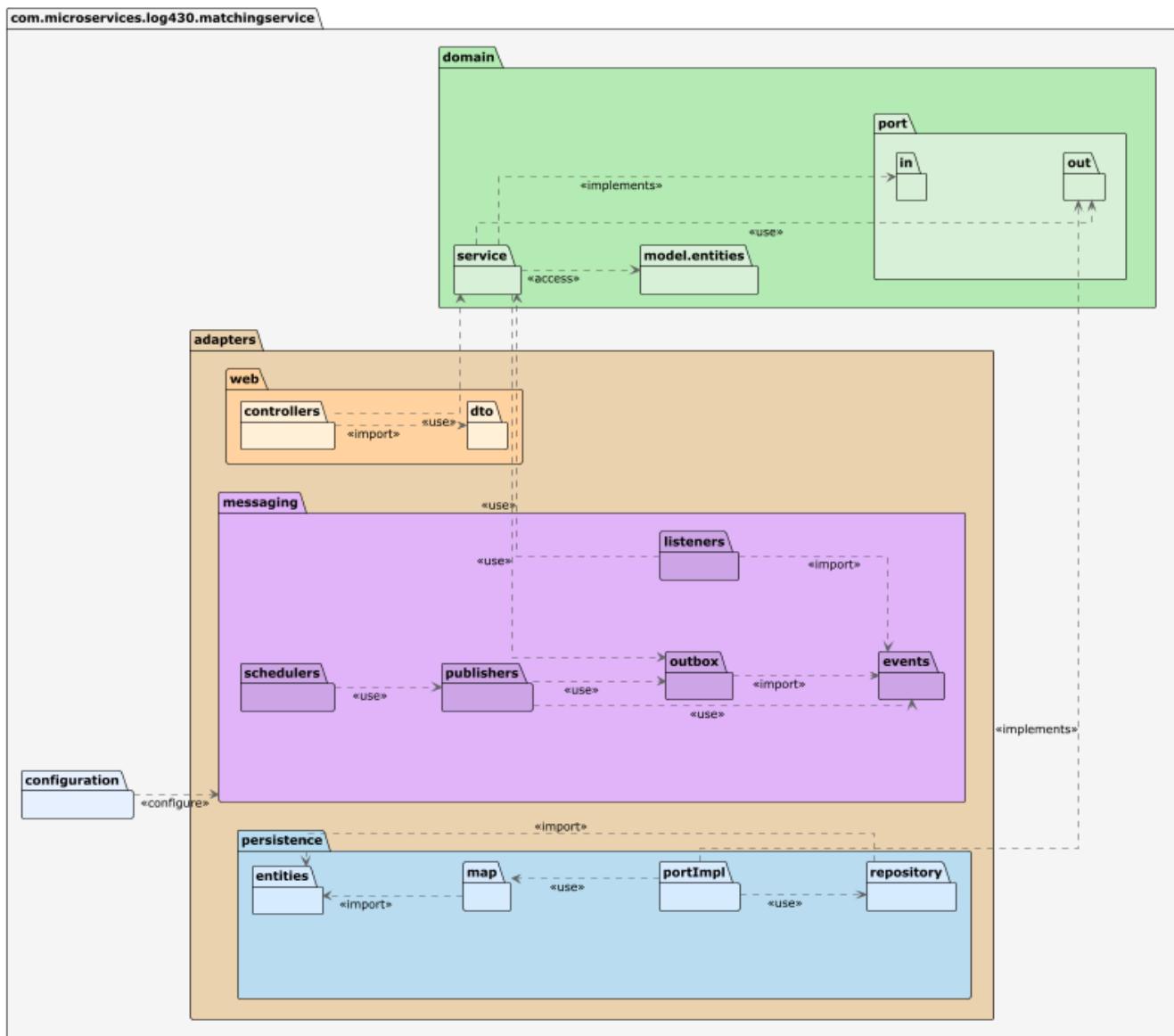


- Diagramme de packages wallet-service :



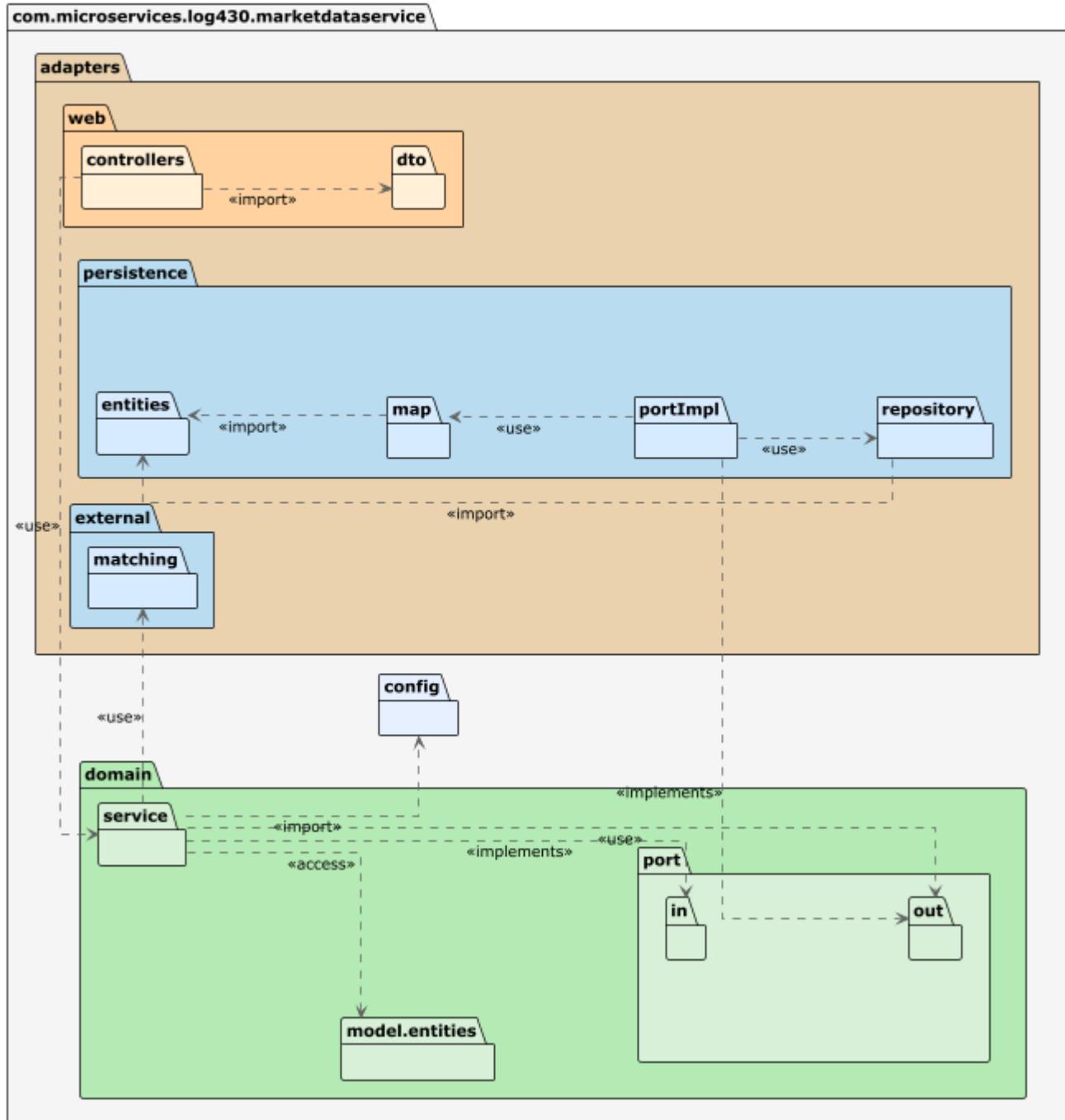
- Diagramme de packages matching-service :

Matching Service - Package Diagram

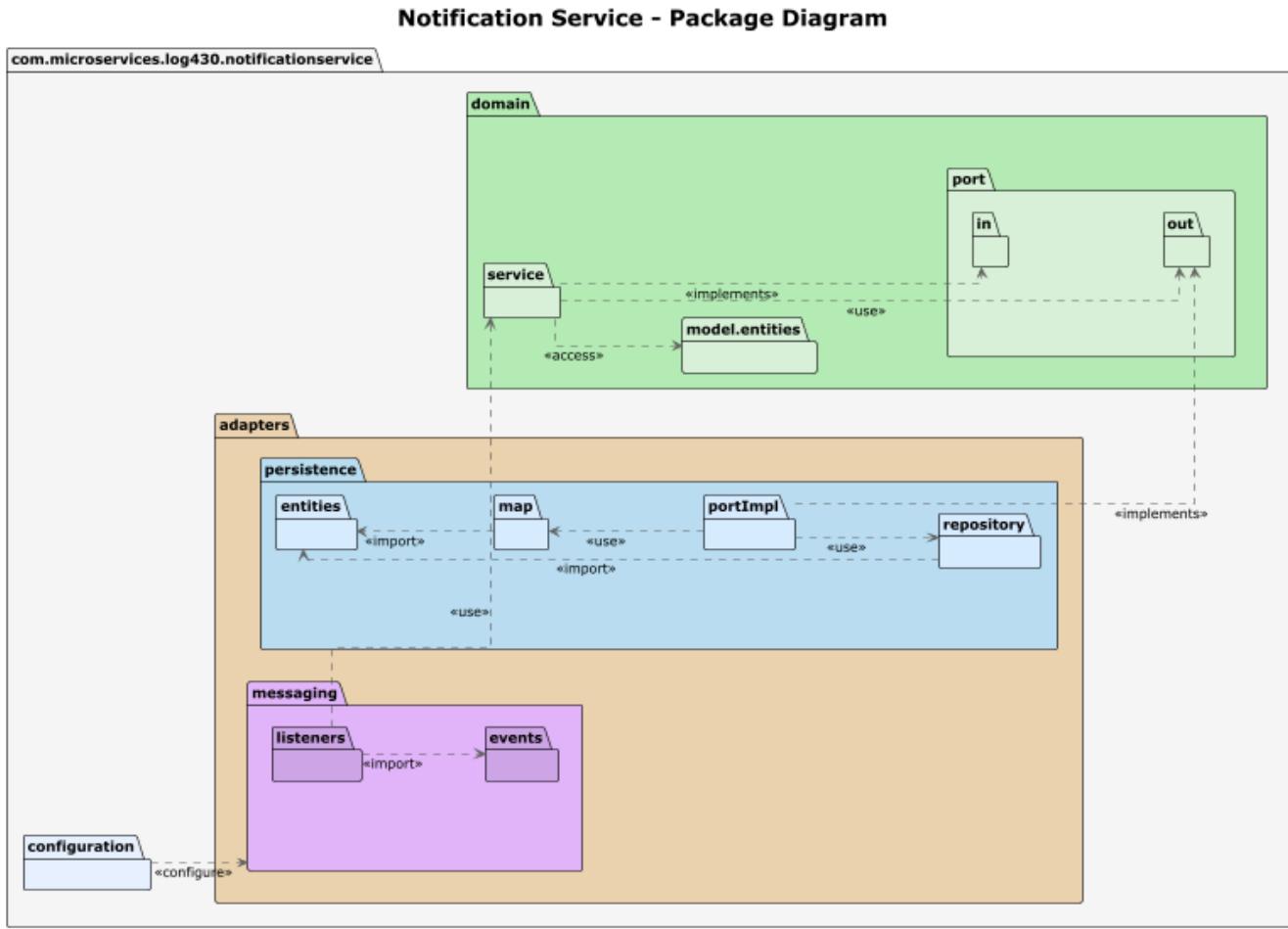


- Diagramme de packages market-data-service :

Market Data Service - Package Diagram



- Diagramme de packages notification-service :



Contexte

La vue développement présente l'organisation du code source, la structure des dossiers, la modularité et les dépendances internes. Elle met en avant la façon dont le projet est découpé pour faciliter le travail des développeurs. Elle propose une double vision : une vue globale de l'architecture (tous les microservices et l'infrastructure) et une vue détaillée de la structure interne de chaque microservice métier.

Le packageDiagram global permet de visualiser la séparation entre les microservices, rabbitMQ, le frontend, le monitoring et la configuration nginx. Les diagrammes détaillés montrent la structure interne de chaque microservice, avec les dépendances typiques entre contrôleurs, services, ports, entités et repositories.

Les microservices comme l'api-gateway n'ont pas de diagramme de package ou de composant détaillé, car ils n'embarquent pas de logique métier propre : ils se contentent de router les requêtes via Spring Cloud Gateway.

Éléments

- Packages principaux : adapters, domain, infrastructure
- Composants techniques : contrôleurs, services, adapters, ports
- Diagrammes de composants et de packages pour chaque microservice (auth, order, wallet, matching, market-data, notification)
- Structure des dossiers et conventions de nommage

Relations

- Les packages sont organisés selon l'architecture hexagonale
- Les dépendances entre modules sont explicites et maîtrisées
- La séparation entre microservices et infrastructure est clairement illustrée
- Les conventions facilitent la maintenance et l'évolution

Rationnel

Cette vue facilite la compréhension du projet pour les développeurs, la maintenance et l'évolution du code. La double vision (globale + microservices) favorise la modularité, la réutilisabilité et la robustesse de l'application. Elle permet d'anticiper les impacts des changements, d'améliorer la qualité du code et de réduire les risques de dette technique. Elle est aussi utile pour l'onboarding, la gestion des versions et la collaboration entre équipes.

11. Concepts transversaux

11.1 Modèle de persistance par microservice

Chaque microservice possède sa propre base de données, avec un schéma ERD dédié :

Auth-Service

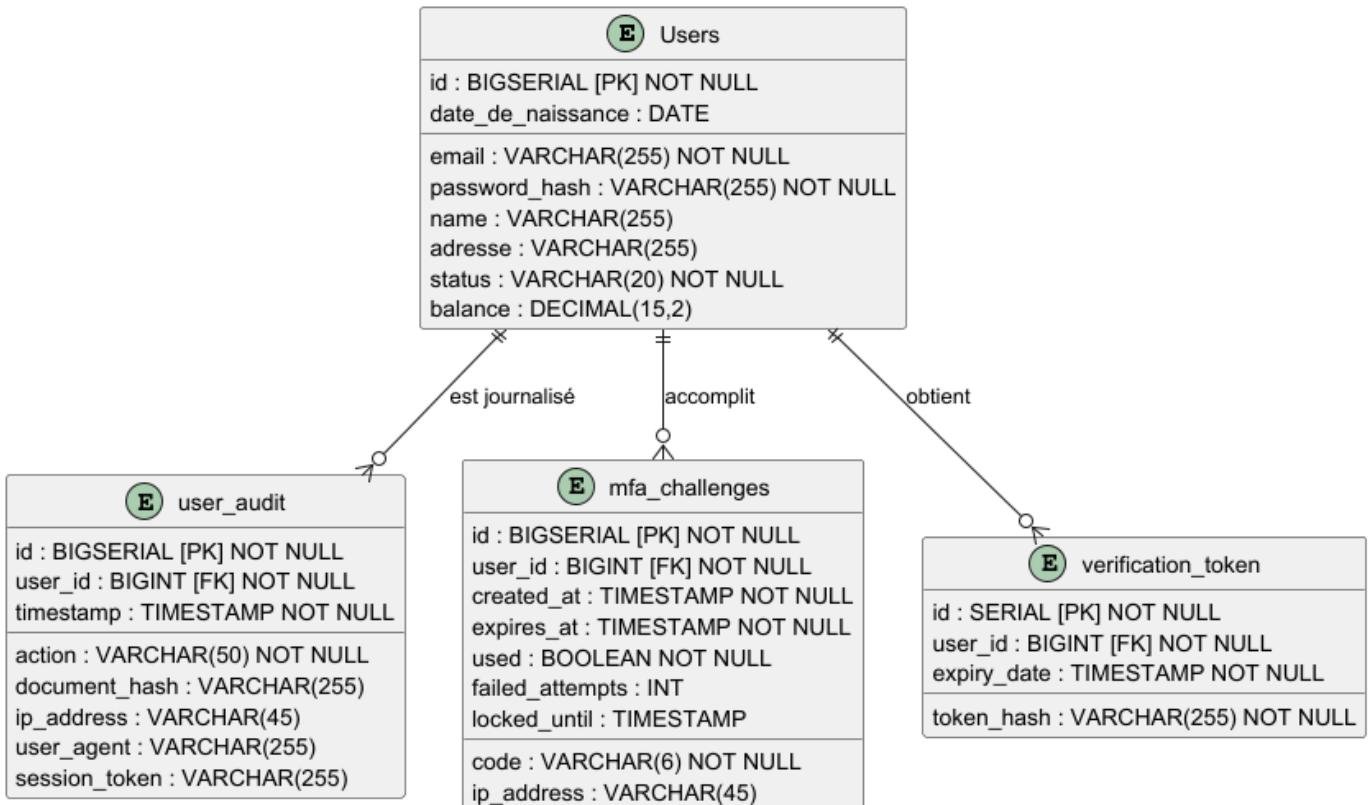


Table	Description
Users	Stocke les utilisateurs : email, mot de passe, nom, adresse, date de naissance, statut, solde, etc.
user_audit	Journalise les actions utilisateur : action, timestamp, document hash, IP, user agent, session token.
mfa_challenges	Stocke les défis MFA : code, date de création/expiration, état d'utilisation, IP, tentatives, verrouillage.
verification_token	Stocke les tokens de vérification : hash du token, date d'expiration.

Wallet-Service

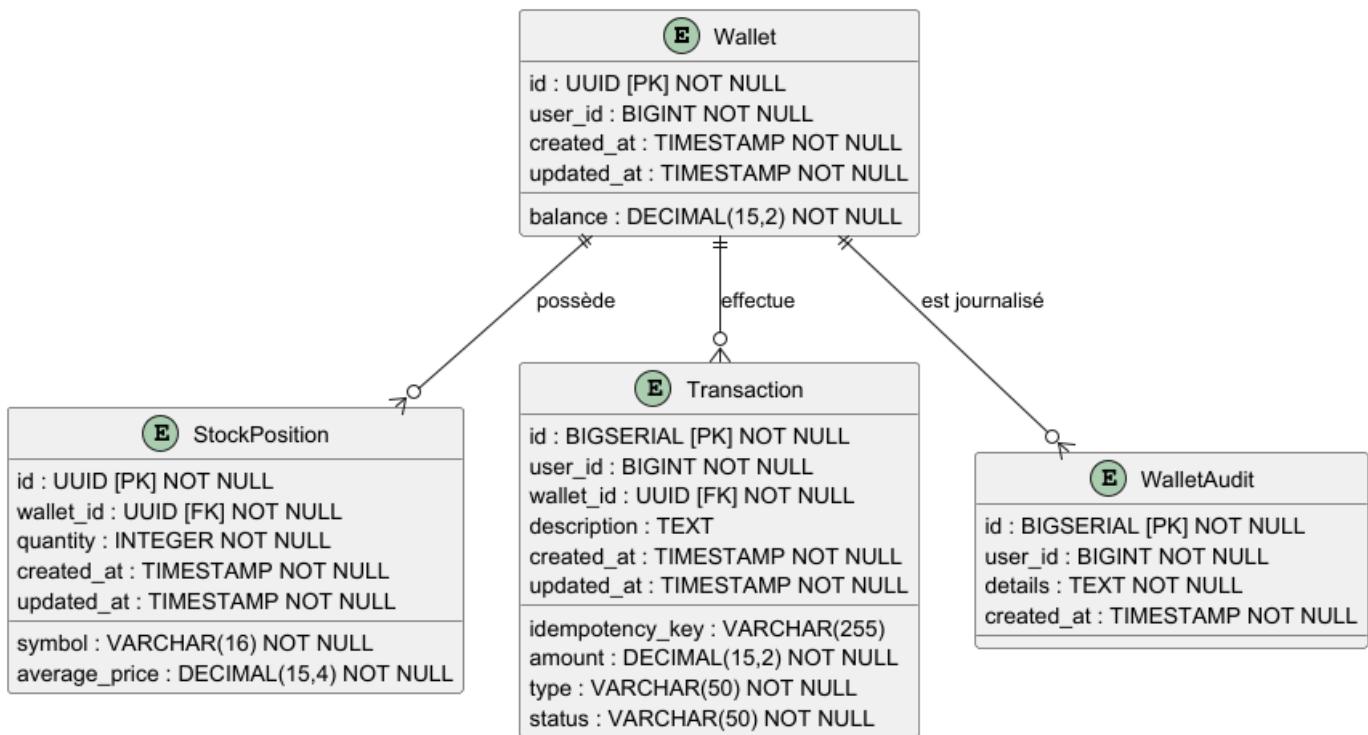


Table	Description
Wallet	Portefeuille virtuel associé à un utilisateur, solde, statut.
StockPosition	Position sur un actif financier détenue dans le portefeuille.
Transaction	Mouvement de fonds : dépôt, retrait, achat, vente, avec idempotence et statut.
WalletAudit	Journalisation des actions sur le portefeuille : dépôt, retrait, modification, etc.

Order-Service

 Order
id : BIGSERIAL [PK] NOT NULL
user_id : BIGINT NOT NULL
quantity : INTEGER NOT NULL
price : DOUBLE PRECISION
timestamp : TIMESTAMP NOT NULL
version : BIGINT NOT NULL
client_order_id : VARCHAR(64)
symbol : VARCHAR(16) NOT NULL
side : VARCHAR(16) NOT NULL
type : VARCHAR(16) NOT NULL
duration : VARCHAR(16) NOT NULL
status : VARCHAR(16) NOT NULL
reject_reason : VARCHAR(255)

 OutboxEvent
id : BIGSERIAL [PK] NOT NULL
event_id : UUID NOT NULL
payload : TEXT NOT NULL
created_at : TIMESTAMP NOT NULL
processed_at : TIMESTAMP
retry_count : INTEGER NOT NULL DEFAULT 0
max_retries : INTEGER NOT NULL DEFAULT 3
next_retry_at : TIMESTAMP
error_message : TEXT
event_type : VARCHAR(64) NOT NULL
aggregate_id : VARCHAR(64) NOT NULL

Table	Description
Order	Ordre de trading : symbole, côté (achat/vente), type, quantité, prix, durée, statut, raison de rejet, timestamp.
OutboxEvent	Événements à publier : type d'événement, payload JSON, statut de traitement, timestamp pour pattern Outbox.

Matching-Service

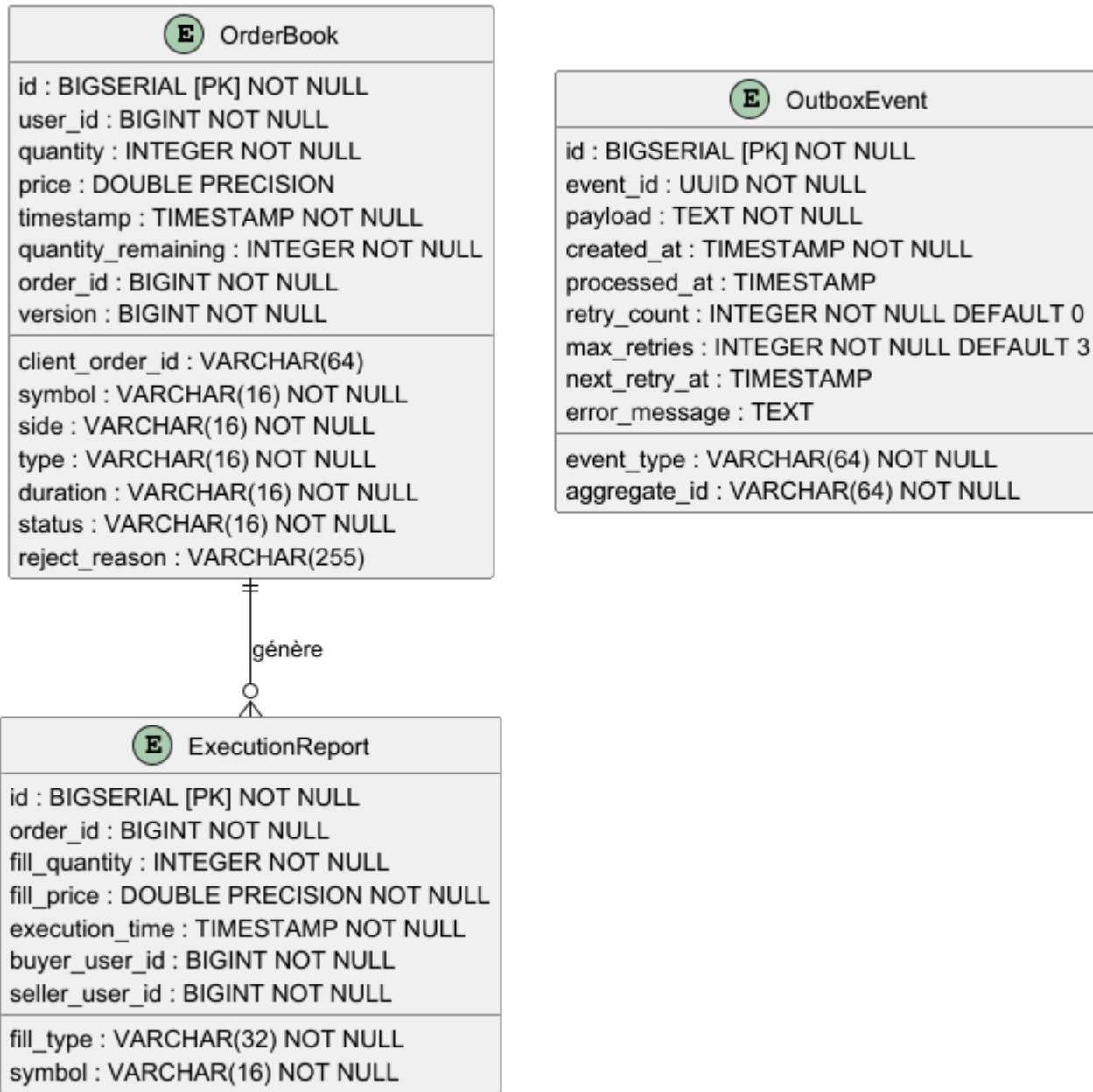


Table	Description
OrderBook	Carnet d'ordres pour un symbole, date de création.
ExecutionReport	Rapport d'exécution : type, quantité, prix, timestamp, lié à un carnet d'ordres et à un ordre.
OutboxEvent	Événements à publier : type d'événement, payload JSON, statut de traitement, timestamp pour pattern Outbox.

Market-Data-Service

 StockRule
id : BIGSERIAL [PK] NOT NULL
updated_at : TIMESTAMP NOT NULL
symbol : VARCHAR(16) NOT NULL
tick_size : DECIMAL(15,8) NOT NULL
min_band : DECIMAL(15,8) NOT NULL
max_band : DECIMAL(15,8) NOT NULL
price : DECIMAL(15,4) NOT NULL

Table	Description
StockRule	Règle métier associée à une position (ex : limites, alertes, restrictions).

Notification-Service

 NotificationLog
id : BIGSERIAL [PK] NOT NULL
user_id : BIGINT NOT NULL
message : TEXT NOT NULL
timestamp : TIMESTAMP NOT NULL
channel : VARCHAR(50) NOT NULL

Table	Description
NotificationLog	Journal des notifications : utilisateur, message, timestamp, canal d'envoi (email, push, WebSocket).

Chaque schéma ERD est adapté à la logique métier et à la séparation stricte des responsabilités de chaque microservice.

11.2 Diagrammes de classes par microservice

Chaque microservice possède son propre diagramme de classes, illustrant les entités métier et leurs relations :

Auth-Service

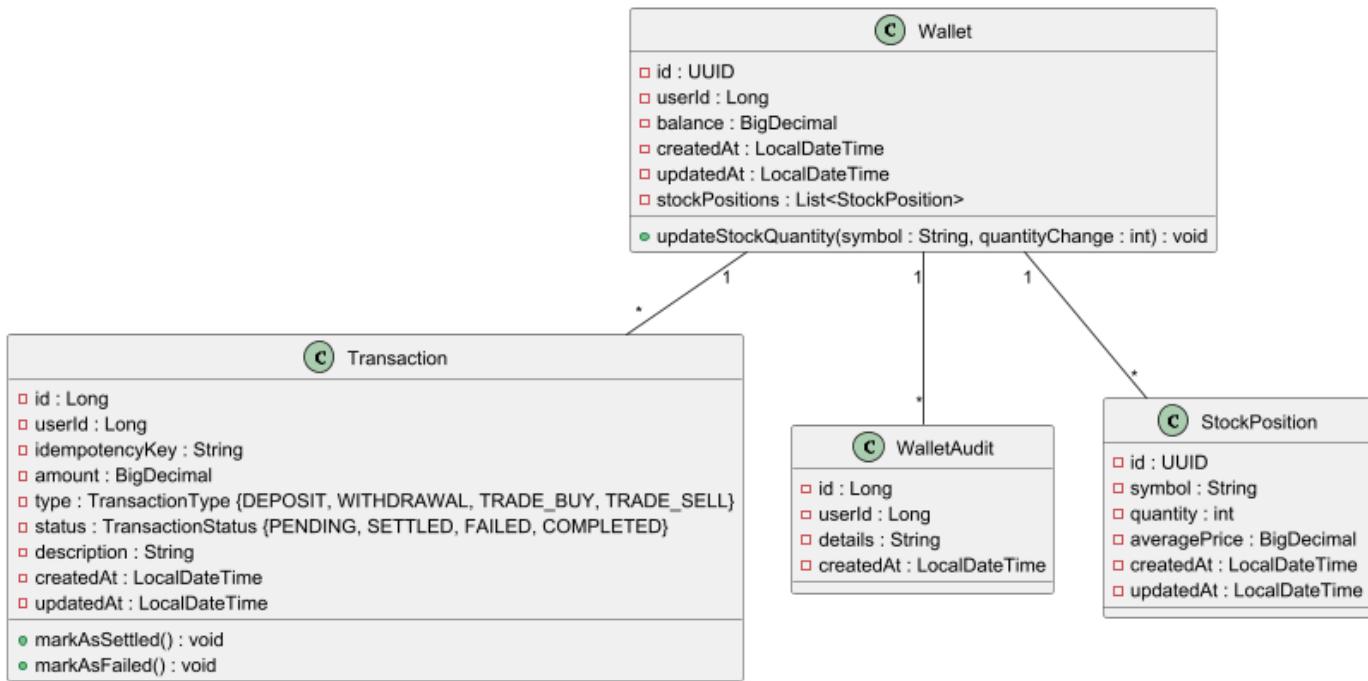
Diagramme de classes - Auth-Service



Classe	Rôle métier
User	Représente un utilisateur, gère l'activation, le rejet, le crédit du solde, et les statuts de compte.
MfaChallenge	Gère les défis MFA, leur validité, expiration, utilisation et verrouillage en cas d'échecs.
VerificationToken	Gère les tokens de vérification pour l'activation de compte ou la récupération d'accès.
UserAudit	Permet de tracer toutes les actions importantes réalisées par un utilisateur.

Wallet-Service

Diagramme de classes - Wallet-Service



Classe	Rôle métier
Wallet	Portefeuille virtuel, gère le solde, les positions, les transactions et l'audit.
StockPosition	Position sur un actif financier détenue dans le portefeuille.
Transaction	Mouvement de fonds : dépôt, retrait, achat, vente, avec gestion d'idempotence et de statut.
WalletAudit	Journalisation des actions sur le portefeuille : dépôt, retrait, modification, etc.

Order-Service

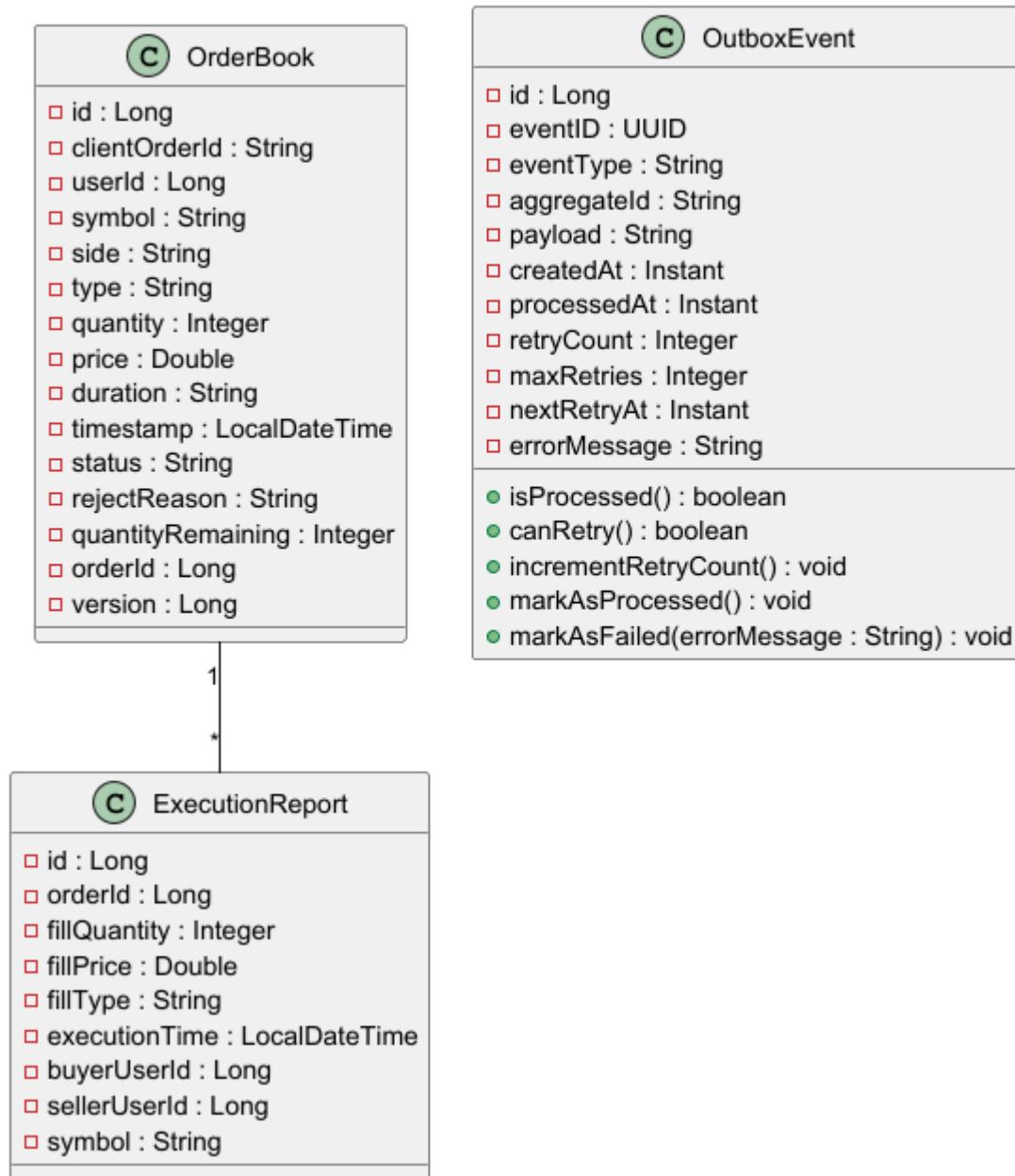
Diagramme de classes - Order-Service

<p>C Order</p> <ul style="list-style-type: none"> □ id : Long □ clientOrderId : String □ userId : Long □ symbol : String □ side : Side {ACHAT, VENTE} □ type : OrderType {MARCHE, LIMITE} □ quantity : int □ price : Double □ duration : DurationType {DAY, IOC, FOK} □ timestamp : Instant □ status : OrderStatus □ rejectReason : String □ version : Long 	<p>C OutboxEvent</p> <ul style="list-style-type: none"> □ id : Long □ eventID : UUID □ eventType : String □ aggregateId : String □ payload : String □ createdAt : Instant □ processedAt : Instant □ retryCount : Integer □ maxRetries : Integer □ nextRetryAt : Instant □ errorMessage : String ● isProcessed() : boolean ● canRetry() : boolean ● incrementRetryCount() : void ● markAsProcessed() : void ● markAsFailed(errorMessage : String) : void
---	--

Classe	Rôle métier
Order	Représente un ordre de trading, avec gestion du type, quantité, prix, durée, statut et raison de rejet.
OutboxEvent	Événement à publier vers RabbitMQ, implémente le pattern Outbox pour garantir la cohérence événementielle.

Matching-Service

Diagramme de classes - Matching-Service



Classe	Rôle métier
OrderBook	Carnet d'ordres, gère la priorité prix/temps et la gestion des ordres.
ExecutionReport	Rapport d'exécution : type, quantité, prix, timestamp, lié à un carnet d'ordres et à un ordre.
OutboxEvent	Événement à publier vers RabbitMQ, implémente le pattern Outbox pour garantir la cohérence événementielle.

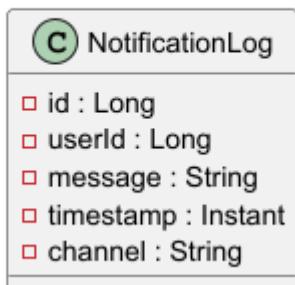
Diagramme de classes - Market-Data-Service



Classe	Rôle métier
StockRule	Règles de cotation des actions : symbole, tick size, bandes min/max, prix actuel, timestamp de mise à jour.

Notification-Service

Diagramme de classes - Notification-Service

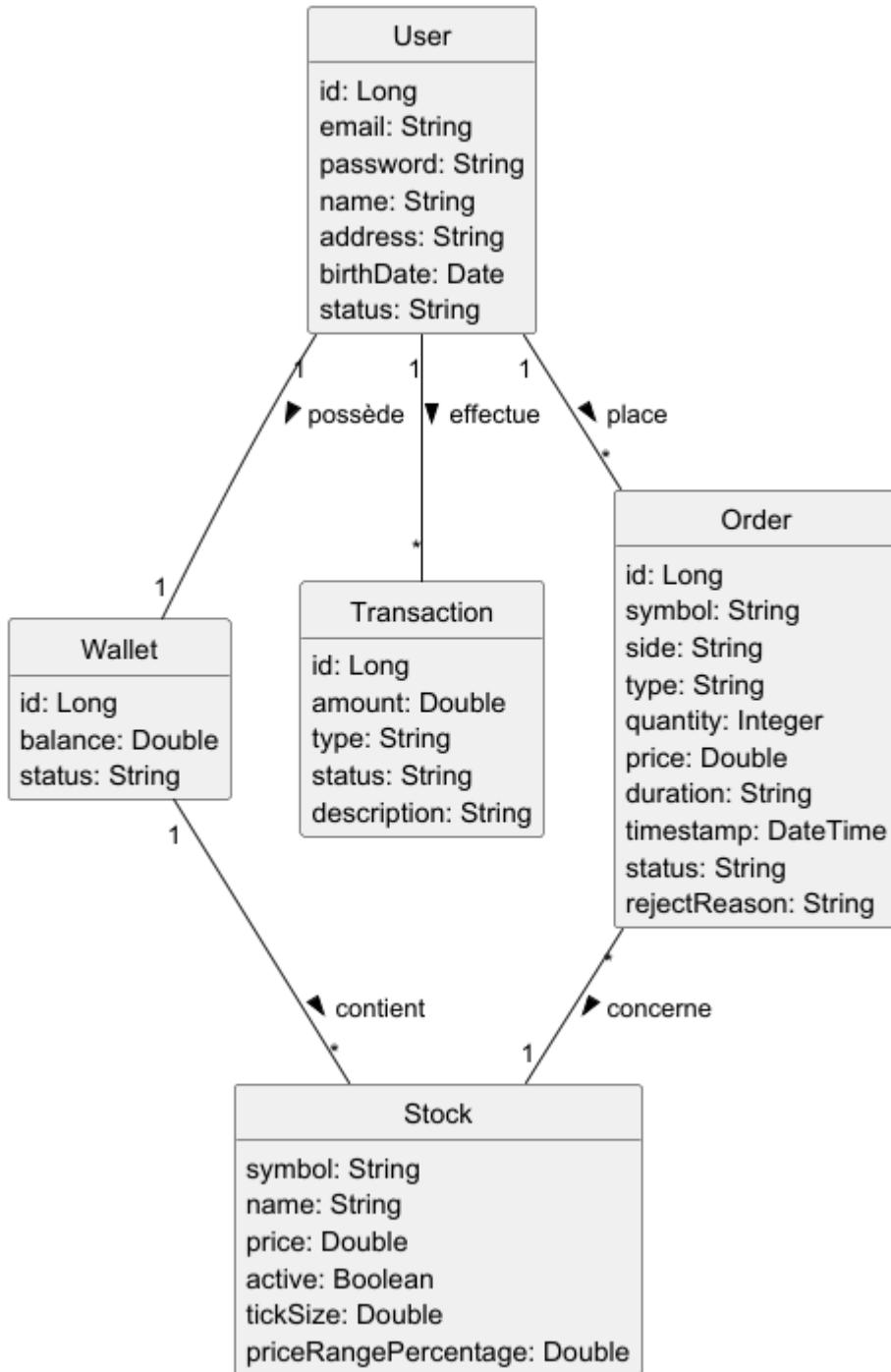


Classe	Rôle métier
NotificationLog	Journal des notifications : utilisateur, message, timestamp, canal d'envoi (email, push, WebSocket).

Chaque diagramme de classes permet de visualiser la structure métier propre à chaque microservice, facilitant la compréhension, la maintenance et l'évolution du code.

Modèle de domaine

Le diagramme de modèle de domaine ci-dessous synthétise les entités principales, leurs relations et les règles métier fondamentales de BrokerX. Il offre une vue conceptuelle du cœur métier, indépendante des choix techniques de persistance ou d'implémentation.



Ce modèle permet de visualiser rapidement les objets métier, leurs interactions et les invariants structurants du domaine de courtage en ligne.

11.2 Persistance

BrokerX adopte une architecture microservices événementielle : chaque microservice possède sa propre base PostgreSQL, avec un schéma métier dédié. Les données critiques (utilisateurs, ordres, transactions, MFA, audit, etc.) sont isolées par domaine, ce qui garantit la robustesse, la conformité et la scalabilité.

La persistance de chaque microservice s'appuie sur :

- **PostgreSQL** : chaque microservice dispose d'une base indépendante, gérée via Docker volumes pour la durabilité.

- **Spring Data JPA/Hibernate** : les entités métier sont annotées `@Entity`, les accès se font via des repositories Spring, assurant l'abstraction et la sécurité.
- **Scripts de migration Flyway** : chaque service maintient ses migrations SQL dans son propre dossier, garantissant la reproductibilité et la traçabilité des évolutions de schéma.
- **Données seed** : chaque microservice peut insérer ses propres données de démonstration ou de test via des scripts dédiés.
- **Pattern Outbox** : les microservices participants aux workflows événementiels (order-service, matching-service) utilisent des tables `outbox_event` pour garantir la cohérence entre les opérations métier et la publication d'événements vers RabbitMQ. Cette approche assure la fiabilité de la publication événementielle via des transactions locales.

Architecture événementielle et persistance : Les services order-service et matching-service implémentent le pattern Outbox avec des tables dédiées (`outbox_event`) qui stockent les événements à publier. Un processus de publication asynchrone lit ces tables et transmet les événements vers RabbitMQ, garantissant qu'aucun événement n'est perdu même en cas de défaillance du message broker. Cette approche maintient la cohérence des données distribuées tout en respectant l'isolation des bases de données par microservice.

Aucune donnée métier n'est stockée en mémoire ou dans des fichiers plats : tout est centralisé dans PostgreSQL. Les fichiers statiques (images, documents) sont référencés par leur hash ou URL, mais non stockés dans la base.

La configuration des connexions (URL, credentials) est externalisée dans les variables d'environnement Docker et les fichiers de configuration de chaque microservice.

11.2.1 Choix ORM ou DAO

Chaque microservice utilise Spring Data JPA/Hibernate pour la gestion de la persistance. Les entités sont mappées via annotations, les relations sont explicites (`OneToMany`, `ManyToOne`, etc.), et les repositories héritent des interfaces Spring Data. Cette approche réduit le code technique, facilite la maintenance et la testabilité, et assure la cohérence entre le modèle objet et la base.

11.2.2 Transactions

La gestion des transactions est assurée par l'annotation `@Transactional` sur les services et les repositories critiques. Spring orchestre automatiquement le commit et le rollback, garantissant l'atomicité et la cohérence des opérations sensibles (dépôts, ordres, MFA, audit). Les transactions sont isolées par microservice : il n'y a pas de transaction distribuée entre bases.

11.2.3 Contraintes d'intégrité

Les contraintes d'intégrité (clés primaires, étrangères, unicité, NOT NULL, index) sont définies dans les scripts Flyway et les entités JPA de chaque microservice. Les validations métier sont doublées côté code et côté base, assurant la fiabilité et la conformité des données.

11.2.4 Migrations reproductibles

Chaque microservice gère ses propres migrations Flyway, versionnées et appliquées indépendamment. Les scripts sont idempotents et traçables, garantissant la synchronisation des schémas entre les environnements. Les migrations sont testées et validées à chaque livraison.

11.2.5 Données seed

Les données de seed sont insérées par chaque microservice via des scripts dédiés, permettant de peupler les bases avec des utilisateurs, portefeuilles, ordres, transactions et MFA pour les tests et les démonstrations. Les seeds couvrent les principaux cas d'utilisation métier et assurent un démarrage cohérent du système.

Cette organisation garantit la robustesse, la conformité et la portabilité du système BrokerX, tout en facilitant la maintenance, l'évolution et la scalabilité de chaque domaine métier.

11.3 Interface Utilisateur

L'interface utilisateur de BrokerX est une application web moderne développée avec React. Elle offre une expérience fluide, responsive et sécurisée pour toutes les opérations de courtage : inscription, vérification d'identité (KYC), authentification MFA, gestion du portefeuille, passage d'ordres et consultation des historiques. Les principales caractéristiques :

- Frontend React : SPA (Single Page Application) avec gestion des états, navigation dynamique et appels API REST sécurisés (HTTPS/JWT).
- Séparation frontend/backend : le frontend communique exclusivement avec l'API Gateway, qui centralise la sécurité et le routage vers les microservices.
- Gestion des formulaires : toutes les interactions (inscription, dépôt, ordre, MFA) passent par des formulaires React, avec validation côté client et serveur.
- Sécurité : authentification forte via MFA, gestion des rôles, tokens JWT, et chiffrement TLS pour toutes les communications.
- Design : interface responsive, adaptée aux usages financiers, avec composants réutilisables et gestion des erreurs utilisateur.
- Dépendances JS/CSS : utilisation de npm pour la gestion des librairies installées comme react-router-dom.

Toutes les opérations critiques (inscription, authentification, ordres, dépôts) sont journalisées côté backend : l'UI affiche les statuts, les notifications et les historiques en temps réel.

Cette organisation garantit une expérience utilisateur moderne, sécurisée et conforme aux exigences du secteur financier, tout en facilitant l'évolution et la maintenance du frontend.

11.4 Optimisation JavaScript et CSS

Le frontend BrokerX est développé avec React et utilise des fichiers .css classiques ou du CSS inline, sans framework CSS externe (type Bootstrap, Material UI) ni librairie de gestion avancée des styles. L'organisation des styles repose sur des fichiers CSS modulaires importés dans les composants React, ou sur du style inline directement dans le JSX.

La gestion des dépendances JavaScript se fait via npm, principalement pour React et quelques librairies utilitaires (ex : react-router-dom pour la navigation). Aucune librairie de gestion ou d'optimisation CSS n'est utilisée : pas de SASS, LESS, Tailwind, Styled Components, etc.

L'optimisation des ressources JS/CSS est assurée par le build React (Webpack/Babel), qui minifie et bundle automatiquement le code lors de la compilation. Aucun outil externe de minification ou de post-traitement n'est requis.

Cette approche garantit :

- Une structure simple et maintenable des styles
- Une portabilité maximale du frontend

- Un déploiement sans dépendance à des frameworks CSS ou outils de build additionnels

Le code CSS et JS est donc soit inline dans les composants, soit dans des fichiers .css importés, et le build React assure l'optimisation standard pour la production.

11.5 Traitement des transactions

BrokerX combine la gestion des transactions locales via Spring Boot et JPA avec une architecture événementielle pour les workflows distribués.

Transactions locales : Toutes les opérations critiques au sein d'un microservice (ordres, dépôts, MFA) sont traitées de façon transactionnelle via l'annotation `@Transactional` pour garantir la cohérence des données locales.

Cohérence distribuée : Pour le workflow du placement d'ordre, BrokerX utilise le pattern Outbox couplé à RabbitMQ. Les événements métier sont stockés dans des tables `outbox_event` au sein de la même transaction que les opérations métier, puis publiés de manière asynchrone vers RabbitMQ. Cette approche garantit qu'aucun événement n'est perdu même en cas de défaillance du message broker.

Pattern Saga chorégraphiée : Les transactions distribuées sont orchestrées via des Sagas chorégraphiées, où chaque microservice publie des événements et réagit aux événements des autres services. Cette approche maintient l'indépendance des services tout en assurant la cohérence finale des données distribuées.

BrokerX ne supporte pas les transactions distribuées synchrones (2PC), privilégiant la résilience et la scalabilité des microservices via l'architecture événementielle.

11.6 Gestion de session

BrokerX expose uniquement une API publique stateless (à travers le gateway) : aucune gestion de session côté serveur. L'authentification et l'autorisation sont gérées par des tokens JWT et MFA, sans stockage de session. Ces tokens JWT sont transmis lors de chaque appel d'API.

11.7 Sécurité

La sécurité des endpoints API BrokerX repose sur l'authentification forte (MFA, JWT) et le chiffrement TLS. Les accès sont contrôlés par des rôles et toutes les opérations sensibles sont journalisées. Toutes les étapes critiques nécessitent un token valide pour pouvoir être effectuées. Pour renforcer la sécurité, l'application peut être déployée derrière un proxy SSL ou avec la configuration TLS du conteneur Tomcat embarqué.

Ce niveau de sécurité est adapté au type de données gérées et aux exigences réglementaires du secteur financier.

11.8 Sûreté

Aucune partie du système BrokerX ne présente de risque vital ou d'impact sur la sécurité physique des utilisateurs.

11.9 Communication inter-microservices

BrokerX utilise une architecture de communication hybride combinant appels synchrones et événements asynchrones :

Communication synchrone : Les opérations simples utilisent des appels HTTP REST entre microservices pour les consultations directes et les validations immédiates (ex : vérification de solde, authentification).

Communication asynchrone : Les workflows complexes impliquant plusieurs microservices utilisent une architecture événementielle via RabbitMQ. Le pattern Outbox garantit la fiabilité de la publication d'événements : les événements sont d'abord stockés dans des tables `outbox_event` lors de la transaction métier, puis publiés de manière asynchrone vers RabbitMQ.

Patterns événementiels :

- **Pattern Saga chorégraphiée** : Chaque microservice publie des événements et réagit aux événements des autres services, maintenant l'indépendance tout en assurant la cohérence finale
- **Pattern Outbox** : Garantit qu'aucun événement n'est perdu, même en cas de défaillance du message broker

Message Broker : RabbitMQ orchestre les échanges asynchrones avec des queues dédiées par type d'événement (`order.placed`, `order.matched`, `notification.sent`). Cette approche assure la résilience, la scalabilité et le découplage des microservices.

L'API Gateway centralise l'accès externe et le routage, tandis que le Load Balancer (NGINX) distribue les requêtes vers les instances des microservices, garantissant haute disponibilité et performance.

11.10 Vérifications de plausibilité et de validité

La validation des types et des plages de données est assurée par des annotations JSR-303 sur les entités du domaine (ex : `@NotNull`). Les contrôleurs REST et services métier vérifient systématiquement la conformité des données reçues.

Principales règles métier :

- Un utilisateur désactivé ou rejeté ne peut pas passer d'ordre ni déposer de fonds (contrôlé dans UserService et OrderService).
- Un ordre ne peut être placé que si le solde du portefeuille est suffisant (contrôlé dans OrderService).
- Un dépôt doit être strictement positif (contrôlé dans WalletDepositService).

Les contrôles sont réalisés à la fois au niveau des entités (annotations) et dans les services métier pour garantir la cohérence métier et la sécurité des opérations.

11.11 Gestion des exceptions/erreurs

La gestion des erreurs dans BrokerX repose sur un format JSON normalisé pour toutes les réponses d'erreur, mis en place au niveau de l'API Gateway. Chaque microservice retourne ses erreurs métier ou techniques avec le code HTTP approprié (400, 401, 403, 404, 409, 422, 500, etc.), et le gateway centralise la transformation : toutes les erreurs sont renvoyées au client dans un format uniforme, facilitant l'intégration et le traitement côté frontend

Le format d'erreur JSON inclut systématiquement : le code HTTP, un identifiant d'erreur, un message explicite, et éventuellement des détails ou des champs invalides. Cela garantit une expérience cohérente pour les consommateurs d'API et simplifie le monitoring et le debug.

Les contrôleurs REST de chaque microservice gèrent la conversion des exceptions en réponses HTTP standardisées, et le gateway applique la normalisation du format. Les erreurs critiques sont journalisées pour analyse et audit,

assurant la traçabilité et la conformité réglementaire.

Gestion des erreurs dans les workflows asynchrones : Pour les opérations événementielles (matching d'ordres, notifications), les erreurs métier sont capturées et transformées en événements d'erreur publiés vers RabbitMQ. Le notification-service écoute ces événements d'erreur et envoie automatiquement des notifications d'erreur aux utilisateurs concernés via WebSocket, assurant un retour d'information en temps réel même pour les opérations asynchrones.

11.12 Journalisation et traçabilité

BrokerX journalise toutes les actions critiques des utilisateurs (connexion, ordres, dépôts, MFA, vérification) dans la table UserAudit de la base PostgreSQL. Les opérations sur les portefeuilles sont spécifiquement tracées dans la table WalletAudit, et tous les événements d'exécution d'ordres sont persistés dans la table ExecutionReport. Chaque entrée d'audit contient l'action, le timestamp, l'IP, le user agent et le token de session.

La journalisation technique est assurée via SLF4J : tout le code métier contient des logs structurés, facilitant la traçabilité et l'analyse des opérations. Les logs applicatifs sont envoyés en stdout et collectés par l'infrastructure.

Pour l'observabilité, BrokerX expose des métriques via Prometheus, permettant le monitoring en temps réel. Grafana est utilisé pour la visualisation des logs, métriques et alertes, ce qui renforce la supervision, la détection d'anomalies et la conformité réglementaire.

Les noms des loggers correspondent aux packages des classes pour faciliter l'identification des modules dans les logs. La traçabilité métier est garantie par la persistance des logs d'audit en base et la collecte centralisée des métriques et logs techniques.

Journalisation événementielle : Les workflows asynchrones bénéficient d'une traçabilité renforcée via la journalisation des événements RabbitMQ. Chaque événement publié (order.placed, order.matched, notification.sent) est tracé avec un identifiant unique, permettant de suivre les Sagas distribuées de bout en bout. Les tables `outbox_event` conservent l'historique de tous les événements publiés, garantissant l'auditabilité des transactions distribuées.

11.13 Configurabilité

BrokerX adopte une approche de configuration externalisée pour faciliter le déploiement et la gestion des environnements (développement, test, production).

Configuration des microservices : Chaque microservice utilise les fichiers `application-properties.yml` et les variables d'environnement Docker pour sa configuration. Les paramètres sensibles (mots de passe de base de données, clés API, secrets JWT) sont externalisés via les variables d'environnement définies dans le `docker-compose.yml`, évitant leur exposition dans le code source.

Configuration de l'infrastructure :

- **PostgreSQL** : Les paramètres de connexion (URL, utilisateur, mot de passe) sont configurés via les variables d'environnement Docker pour chaque base de données dédiée par microservice
- **RabbitMQ** : La configuration du message broker (URL, credentials, queues, exchanges) est externalisée dans les variables d'environnement et les fichiers de configuration Spring AMQP
- **NGINX** : La configuration du load balancer est définie dans le fichier `nginx.conf`, permettant d'ajuster le routage, les timeouts et la répartition de charge

Configuration événementielle : Les paramètres des patterns Saga et Outbox (intervalles de publication, retry policies, timeouts des événements) sont configurables via les fichiers `application.yml` de chaque microservice participant aux workflows asynchrones.

Observabilité configurable : Les métriques Prometheus, les dashboards Grafana et les niveaux de logging sont configurables sans recompilation, permettant d'adapter le monitoring aux besoins spécifiques de chaque environnement.

Configuration de sécurité : Les paramètres MFA (durée de validité des codes, nombre de tentatives autorisées), les secrets JWT (clés, durées d'expiration) et les configurations SMTP sont externalisés et modifiables selon l'environnement de déploiement.

Cette approche garantit la portabilité du système entre environnements, facilite les déploiements automatisés et permet une adaptation fine des paramètres opérationnels sans modification du code source.

11.14 Internationalisation

L'unique langue supportée par BrokerX est le français. Il n'existe aucun mécanisme d'internationalisation dans l'interface utilisateur ou l'API, et aucune évolution n'est prévue à ce sujet.

11.15 Migration

Le projet BrokerX a initialement été développé sous forme d'application monolithique Java/Spring Boot. Une migration vers une architecture microservices a été réalisée : le code, les modèles métier et les données ont été découpés et répartis dans des microservices indépendants (auth-service, order-service, wallet-service, matching-service), chacun avec sa propre base PostgreSQL.

Première migration vers les microservices : La migration a impliqué :

- La séparation du code source en plusieurs projets Maven distincts, un par microservice
- La refonte des schémas de base de données : chaque microservice possède désormais son propre schéma et ses scripts de migration Flyway
- La migration des données existantes du monolithe vers les bases dédiées des microservices, avec adaptation des formats et des relations
- La mise en place d'une API Gateway et d'un load balancer pour centraliser l'accès et le routage
- L'adaptation des scripts de déploiement Docker et Docker Compose pour orchestrer les nouveaux services

Migration vers l'architecture événementielle : Une seconde évolution majeure a consisté à introduire progressivement l'architecture événementielle pour gérer les workflows complexes impliquant plusieurs microservices :

- **Introduction de RabbitMQ** : Déploiement du message broker pour orchestrer les communications asynchrones entre microservices
- **Implémentation du pattern Outbox** : Ajout des tables `outbox_event` dans order-service et matching-service pour garantir la cohérence entre les opérations métier et la publication d'événements
- **Migration des workflows critiques** : Le processus de placement d'ordre a été transformé d'un workflow synchrone vers un workflow événementiel orchestré via des Sagas chorégraphiées
- **Ajout des composants événementiels** : Développement des EventPublishers, EventListeners et des services de traitement des événements dans chaque microservice participant

- **Configuration des échanges RabbitMQ** : Mise en place des queues, exchanges et routage pour les différents types d'événements (order.placed, order.matched, notification.sent)

Approche hybride de migration : La migration vers l'architecture événementielle a été réalisée de manière progressive, conservant les appels synchrones pour les opérations simples (consultations, authentification) tout en introduisant les événements asynchrones pour les workflows distribués complexes. Cette approche a permis de minimiser les risques et de valider progressivement la fiabilité des patterns Saga et Outbox.

Aucune donnée n'a été perdue lors des migrations : toutes les opérations critiques (utilisateurs, ordres, portefeuilles, MFA, audit) ont été migrées et vérifiées. Le système fonctionne désormais en architecture microservices événementielle hybride, avec une isolation stricte des domaines métier, une scalabilité accrue et une résilience renforcée grâce aux patterns de cohérence distribuée.

11.16 Testabilité

Chaque microservice BrokerX est couvert par des tests automatisés : les services métier, les contrôleurs REST et les entités principales sont testés via JUnit dans le dossier standard `src/test/java` de chaque projet Maven. Les tests d'intégration vérifient les interactions avec la base PostgreSQL dédiée, et des mocks sont utilisés pour les dépendances externes (SMTP, API Gateway).

Le pipeline CI/CD exécute systématiquement tous les tests à chaque build Maven : aucune livraison n'est acceptée si les tests échouent. Les cas d'utilisation critiques (inscription, authentification MFA, dépôt, passage d'ordre, matching, etc.) sont systématiquement couverts par des scénarios de test, garantissant la robustesse et la non-régression du système.

La couverture de test est mesurée et documentée : chaque microservice doit atteindre un seuil minimal de couverture sur les classes métier et les contrôleurs. Les tests facilitent la maintenance, l'évolution et la fiabilité de l'architecture microservices.

11.17 Gestion du build

Le build de BrokerX est entièrement automatisé via Maven et le pipeline CI/CD défini dans les fichiers `ci.yml` et `cd.yml`. À chaque push ou pull request, le pipeline exécute les étapes suivantes :

- Compilation des microservices Java/Spring Boot avec Maven (➔ `mvn clean install`)
- Exécution des tests unitaires et d'intégration : le build échoue si un test ne passe pas
- Packaging des artefacts JAR pour chaque microservice
- Construction des images Docker via `docker build`
- Publication des images sur le registre Docker si les tests sont validés
- Déploiement automatisé sur l'environnement cible (dev, prod) via Docker Compose

Les logs de build et de test sont archivés pour audit et traçabilité. Les dépendances sont gérées exclusivement via Maven et npm (pour le frontend React). Aucun artefact n'est livré si la qualité ou la couverture de test minimale n'est atteinte. Si tout passe, un artefact pour le build du projet ainsi qu'un artefact pour le rapport de tests sont générés et stockés.

Cette organisation garantit la reproductibilité, la fiabilité et la rapidité des livraisons, tout en assurant la conformité aux exigences de qualité et de sécurité du projet BrokerX.

12. Décisions d'architecture

12.1 Architecture microservices et découpage

ADR 001 : Style architectural & découpage microservices

Statut : Acceptée

Date : 2025-10-24

Contexte

Le projet BrokerX, initialement monolithique, doit évoluer vers une architecture microservices pour répondre aux exigences de scalabilité, modularité, résilience et déploiement indépendant. Le découpage logique s'appuie sur les domaines métier : Authentification, Ordres, Matching, Wallet. L'API Gateway devient le point d'entrée unique, assurant le routage, la sécurité et la cohérence des appels.

Décision

Nous adoptons une architecture microservices REST, chaque service étant conteneurisé et indépendant, avec une API Gateway (Spring Cloud Gateway) pour le routage et la gestion des accès. Ce choix permet de centraliser la sécurité, le monitoring, la documentation (Swagger) et le versionnage des APIs, tout en évitant la duplication de logique dans chaque microservice. Le découpage par domaine métier assure une meilleure évolutivité : chaque équipe peut travailler sur un service sans impacter les autres, et chaque service peut être déployé ou mis à l'échelle indépendamment. Les communications inter-services se font via HTTP REST, avec des routes versionnées et des codes d'erreur normalisés.

Conséquences

- Scalabilité horizontale : chaque service peut être répliqué indépendamment.
- Déploiement et maintenance facilités : isolation des pannes, évolutivité.
- Complexité accrue : gestion des dépendances, monitoring, orchestration.
- API Gateway centralise la sécurité, le routage et la documentation (Swagger).
- Migration facilitée : chaque domaine peut évoluer ou être remplacé sans impacter les autres.

12.2 Persistance des données

ADR 002 : Stratégie de persistance & transactions

Statut : Acceptée

Date : 2025-10-24

Contexte

Dans BrokerX, chaque microservice gère sa propre base de données PostgreSQL pour garantir l'isolation, la scalabilité et la conformité. Les migrations sont gérées par Flyway, l'accès aux données par JPA/Hibernate, et l'intégrité par des transactions ACID. L'audit, l'idempotence et la traçabilité sont des exigences fortes du métier (ordres, comptes, positions).

Décision

Chaque microservice dispose d'une base PostgreSQL dédiée, avec schéma et migrations indépendants. Ce choix permet d'éviter les effets de bord et les blocages entre domaines : une erreur ou une surcharge sur un service n'impacte pas les autres. PostgreSQL a été choisi pour sa robustesse, sa gestion avancée des transactions et sa compatibilité avec les outils de migration (Flyway) et d'ORM (JPA/Hibernate). Les migrations indépendantes facilitent l'évolution du schéma sans coordination complexe entre équipes. Les transactions sont gérées au niveau du service, avec rollback sur erreur. Les clés **d'idempotence** sont utilisées pour les opérations critiques (dépôts, ordres). Un journal d'audit append-only est mis en place dans chaque base de données pour la traçabilité et la conformité. Cette approche évite la complexité des transactions distribuées, tout en assurant la cohérence locale et la traçabilité.

Conséquences

- Isolation des données : chaque service est responsable de son modèle et de son intégrité.
- Scalabilité et résilience accrues : pas de dépendance croisée sur la persistance.
- Migrations reproductibles et auditables (Flyway).
- Gestion robuste des erreurs et des transactions.
- Complexité accrue pour la cohérence globale (pas de transactions distribuées).

12.3 Stratégie d'erreurs, versionnage & conformité

ADR 003 : Stratégie d'erreurs, versionnage & conformité

Statut : Acceptée

Date : 2025-10-24

Contexte

En architecture microservices, la gestion des erreurs, du versionnage d'API et de la conformité (audit, sécurité, traçabilité) est essentielle pour garantir la robustesse, l'évolutivité et la conformité réglementaire. Les erreurs doivent être normalisées (JSON, codes HTTP), le versionnage doit permettre l'évolution sans rupture, et la traçabilité doit répondre aux exigences KYC/AML et audit métier.

Décision

- **Versionnage** : Chaque endpoint de chaque microservice commence systématiquement par `/api/v1`, ce qui permet de gérer l'évolution des contrats d'API sans impacter les clients existants.
- **Stratégie d'erreur** : Chaque microservice retourne ses erreurs dans un format JSON standardisé (code HTTP, identifiant d'erreur, message, détails éventuels). L'API Gateway collecte ces erreurs et les reformate selon le format attendu par le frontend, garantissant ainsi une normalisation et une cohérence parfaite des réponses d'erreur pour l'interface utilisateur.
- **Audit** : Un audit append-only est mis en place pour toutes les opérations sensibles, et la conformité est assurée par la journalisation et la validation des entrées (authentification, sécurité JWT, validation des payloads). La conformité réglementaire et l'audit sont garantis par la journalisation append-only de toutes les opérations sensibles, permettant une traçabilité complète et un accès aux historiques pour contrôle externe.

Ce mécanisme assure que chaque action critique est enregistrée de façon immuable, répondant aux exigences d'audit métier et réglementaire (KYC/AML, contrôle interne, audit externe).

Conséquences

- Robustesse accrue : erreurs claires, versionnage maîtrisé, audit complet.
- Facilité d'évolution des APIs et des clients.
- Conformité réglementaire et traçabilité assurées.
- Complexité technique : gestion des versions, des audits et de la sécurité sur chaque service.

12.4 Justification du choix de cache

ADR 004 : Choix du cache

Statut : Acceptée

Date : 2025-10-24

Contexte

Pour améliorer la performance et la scalabilité, BrokerX doit mettre en place un mécanisme de cache pour les endpoints coûteux. Plusieurs solutions sont envisageables : cache mémoire local, Redis, ou cache distribué. Le cache doit permettre de réduire la charge sur la base de données et d'accélérer les réponses, tout en garantissant la cohérence et la gestion des expirations/invalidation.

Décision

Nous avons choisi d'utiliser un cache mémoire in-memory (Caffeine) pour les endpoints critiques, plutôt que Redis. Ce choix est motivé par la simplicité d'intégration, la rapidité d'accès, et le fait que les données à cacher sont temporaires et propres à chaque instance. Redis aurait été pertinent pour un cache partagé entre plusieurs instances, mais la complexité d'administration et la latence réseau ne sont pas justifiées pour ces usages. Le cache in-memory offre une latence ultra-faible et une intégration native avec Spring Boot. Les endpoints ciblés sont très sollicités mais ne nécessitent pas de cohérence globale entre instances : chaque instance peut gérer son propre cache. Cette approche simplifie le déploiement et la maintenance, tout en maximisant la performance. Les endpoints critiques qui ont été sélectionnés sont : GET api/v1/wallet et GET api/v1/wallet/stock/{stockId}. Ces endpoints ont été sélectionnés parce qu'ils sont énormément utilisés dans toutes les opérations critiques de l'application. Il semblait donc logique de mettre la cache à cette endroit, car le gain semble être le plus important. Des règles d'expiration et de taille maximale du cache sont définies pour éviter la surconsommation de mémoire et garantir la fraîcheur des données.

Conséquences

- Amélioration significative de la latence et du throughput sur les endpoints critiques.
- Réduction de la charge sur la base de données.
- Complexité minimale : pas d'administration Redis, pas de gestion réseau.
- Risque de données obsolètes (stale) limité à l'instance : nécessite des règles d'expiration adaptées.

12.5 Justification du choix de load balancer

ADR 005 : Choix du Load Balancer

Statut : Acceptée

Date : 2025-10-24

Contexte

La montée en charge et la tolérance aux pannes nécessitent un mécanisme de répartition du trafic entre plusieurs instances de microservices. Plusieurs solutions sont envisageables : NGINX, HAProxy, Traefik, ou des services cloud. Le load balancer doit permettre le routage dynamique, la gestion des sessions, le monitoring et la facilité de configuration.

Décision

Nous avons choisi NGINX comme load balancer principal pour BrokerX. Ce choix est motivé par sa robustesse, sa performance, et son intégration simple avec Docker et les microservices. NGINX permet le routage HTTP, la gestion des headers, le monitoring basique, et la configuration de règles avancées (sticky sessions, healthchecks). Il offre une

répartition efficace du trafic, une tolérance aux pannes et une scalabilité horizontale. Les alternatives comme HAProxy ou Traefik ont été écartées pour privilégier la simplicité et la maturité de NGINX dans l'écosystème Docker. Les tests de charge sont réalisés avec NGINX pour comparer la performance selon le nombre d'instances.

Conséquences

- Répartition efficace du trafic et tolérance aux pannes.
- Scalabilité horizontale facilitée.
- Configuration flexible et intégration simple avec Docker Compose.
- Complexité supplémentaire pour le monitoring avancé et la gestion des sessions.

12.6 Justification de l'api gateway

ADR 006 : Choix de l'API Gateway

Statut : Acceptée

Date : 2025-10-24

Contexte

L'API Gateway est un composant clé pour centraliser le routage, la sécurité, la documentation et la gestion des accès aux microservices. Plusieurs solutions sont envisageables : Kong, KrakenD, Spring Cloud Gateway, etc. Le choix doit permettre une intégration native avec l'écosystème Spring, la gestion des routes versionnées, des headers, du CORS, et la documentation Swagger.

Décision

Nous avons choisi Spring Cloud Gateway comme API Gateway pour BrokerX. Ce choix s'explique par l'intégration native avec Spring Boot, la facilité de configuration des routes, la gestion des filtres, du CORS, des headers, et la compatibilité avec Docker. Spring Cloud Gateway permet d'unifier la sécurité, le monitoring, la documentation Swagger et le versionnage des APIs, tout en évitant la duplication de logique dans chaque microservice. La gestion des routes, des filtres et du CORS est flexible et adaptée à l'écosystème Spring déjà utilisé dans tous les microservices. La documentation Swagger peut être centralisée et exposée via la Gateway, facilitant l'accès pour les développeurs et les clients. La compatibilité avec Docker et la facilité de déploiement sont des atouts majeurs pour la CI/CD et la maintenance.

Conséquences

- Centralisation du routage, de la sécurité et de la documentation.
- Intégration native avec l'écosystème Spring et Docker.

- Facilité d'évolution et de maintenance des routes et des règles d'accès.
- Complexité supplémentaire pour la gestion des filtres avancés et du monitoring.

12.7 Justification du choix de message broker

ADR 007 : Choix du message broker

Statut : Acceptée

Date : 2025-11-14

Contexte

La migration vers l'architecture événementielle de BrokerX nécessite un message broker robuste pour orchestrer les communications asynchrones entre microservices. Le workflow de placement d'ordre implique plusieurs étapes : validation, matching, notification et mise à jour du portefeuille. Ces opérations doivent être fiables et coordonnées. Deux solutions principales ont été évaluées : RabbitMQ et Apache Kafka.

Décision

Nous avons choisi RabbitMQ comme message broker pour BrokerX. Trois raisons principales motivent ce choix :

1. **Simplicité du routage** : RabbitMQ excelle dans les patterns de routage complexes grâce à ses exchanges et routing keys. Pour BrokerX, nous devons router différents types d'événements (ordres, matching, notifications) vers les bons services selon des critères précis. RabbitMQ rend cette configuration simple et intuitive.
2. **Garanties de livraison** : RabbitMQ offre des acknowledgments robustes et des dead letter queues pour gérer les erreurs. Dans le trading, chaque événement (ordre placé, exécuté, rejeté) doit être traité de façon fiable. Si un service tombe en panne, RabbitMQ garantit qu'aucun message n'est perdu et les retraite automatiquement.
3. **Intégration Spring native** : RabbitMQ s'intègre parfaitement avec Spring AMQP, déjà utilisé dans tous nos microservices. Cela simplifie énormément le développement des EventPublishers et EventListeners, réduisant la complexité technique et les risques d'erreur.

Kafka aurait été plus adapté pour du streaming de gros volumes de données, mais BrokerX privilégie la fiabilité et la simplicité de routing sur le débit pur.

Conséquences

- Routage flexible et intuitif pour les différents types d'événements
- Fiabilité maximale avec gestion automatique des erreurs et retraitement
- Développement simplifié grâce à l'intégration Spring native
- Débit moindre que Kafka (non critique pour les volumes de BrokerX)

12.8 Justification du choix de base de données

ADR 008 : Choix de la base de données

Statut : Acceptée

Date : 2025-09-14

Contexte

Chaque microservice de BrokerX nécessite sa propre base de données pour garantir l'isolation des domaines métier. Le système doit gérer des données critiques (utilisateurs, ordres, transactions financières, audit) avec des exigences fortes de cohérence, intégrité et conformité réglementaire. Plusieurs options ont été évaluées : PostgreSQL, MySQL, MongoDB et des solutions NoSQL spécialisées.

Décision

Nous avons choisi PostgreSQL comme système de gestion de base de données pour tous les microservices de BrokerX. Trois raisons principales motivent ce choix :

- 1. Conformité ACID et intégrité des données** : PostgreSQL offre des garanties transactionnelles strictes essentielles pour les opérations financières. Les ordres, transactions et positions doivent être cohérents et durables. Les contraintes d'intégrité référentielle et les index uniques préviennent les incohérences critiques dans un système de trading.
- 2. Fonctionnalités avancées pour l'audit** : PostgreSQL supporte nativement les triggers, les fonctions stockées et les types de données JSON, facilitant l'implémentation des tables d'audit append-only. Les extensions comme `pgcrypto` permettent le hachage sécurisé des mots de passe directement en base.
- 3. Écosystème Java mature** : L'intégration avec Spring Data JPA/Hibernate est native et robuste. Les migrations Flyway sont parfaitement supportées, et les drivers JDBC sont optimisés. Cette compatibilité réduit la complexité technique et accélère le développement.

MySQL aurait pu convenir mais offre moins de fonctionnalités avancées. MongoDB aurait été inadapté pour les transactions ACID critiques du trading.

Conséquences

- Garanties ACID strictes pour toutes les opérations financières critiques
- Fonctionnalités d'audit et de sécurité natives (triggers, fonctions, chiffrement)
- Intégration parfaite avec l'écosystème Spring/Java existant
- Coût de performance légèrement supérieur aux solutions NoSQL (non critique pour nos volumes)

12.9 Justification du choix de pattern Saga

ADR 009 : Pattern Saga chorégraphiée vs orchestrée

Statut : Acceptée

Date : 2025-11-14

Contexte

L'architecture événementielle de BrokerX nécessite une gestion de la cohérence des transactions distribuées. Le workflow de placement d'ordre implique plusieurs microservices : validation de l'ordre, vérification du solde, matching, mise à jour du portefeuille et envoi de notifications. Deux approches Saga sont possibles : orchestrée (avec un coordinateur central) ou chorégraphiée (avec événements décentralisés).

Décision

Nous avons choisi le pattern Saga chorégraphiée pour BrokerX. Trois raisons principales motivent ce choix :

- 1. Autonomie des microservices :** Chaque service réagit aux événements RabbitMQ selon sa logique métier sans dépendre d'un orchestrateur central. Le order-service publie `order.placed`, le matching-service écoute et publie `order.matched`, etc. Cette approche respecte l'indépendance des domaines métier.
- 2. Résilience et absence de point de défaillance unique :** Aucun orchestrateur central ne peut faire échouer l'ensemble du workflow. Si un service est temporairement indisponible, RabbitMQ conserve les événements en queue jusqu'à son retour. Cette approche améliore la disponibilité globale du système.
- 3. Évolutivité simple :** Ajouter un nouveau participant (ex: service de risk-management) ne nécessite que de configurer ses listeners d'événements, sans modifier l'orchestrateur. Les workflows peuvent évoluer de façon décentralisée selon les besoins métier.

L'orchestration aurait été plus adaptée pour des workflows très complexes avec de nombreuses conditions, mais BrokerX privilégie la simplicité et l'autonomie des services.

Conséquences

- Indépendance totale des microservices dans les workflows distribués
- Résilience accrue sans point de défaillance central
- Évolutivité simplifiée pour ajouter de nouveaux participants
- Complexité de debugging des workflows distribués (compensée par la traçabilité événementielle)

12.10 Justification de l'architecture interne des microservices

ADR 010 : Architecture interne des microservices - Hexagonale vs MVC

Statut : Acceptée

Date : 2025-11-14

Contexte

Chaque microservice de BrokerX nécessite une architecture interne pour organiser la logique métier, les accès aux données et les interfaces externes. Deux approches principales ont été évaluées : l'architecture MVC traditionnelle (Model-View-Controller) largement utilisée avec Spring Boot, et l'architecture hexagonale (ports et adapters) qui sépare strictement la logique métier des préoccupations techniques.

Décision

Nous avons choisi l'architecture hexagonale pour structurer l'intérieur de chaque microservice BrokerX. Trois raisons principales motivent ce choix :

- 1. Isolation de la logique métier :** L'architecture hexagonale place le domaine métier au centre, complètement isolé des frameworks et des technologies. Les entités (Order, Wallet, User) et les services métier ne dépendent ni de Spring, ni de JPA, ni de RabbitMQ. Cette séparation garantit que les règles de trading, de validation des ordres et de gestion des portefeuilles restent stables même lors des évolutions technologiques.
- 2. Testabilité maximale :** La logique métier peut être testée unitairement sans base de données, sans message broker, sans serveur web. Les ports définissent des contrats clairs (OrderRepository, NotificationPort, EventPublisher) qui peuvent être mockés facilement. Cette approche améliore drastiquement la couverture de test et la rapidité d'exécution des tests unitaires.
- 3. Évolutivité technique :** Changer de base de données (PostgreSQL vers MongoDB), de message broker (RabbitMQ vers Kafka) ou d'API (REST vers GraphQL) ne nécessite que de modifier les adapters externes, sans impacter la logique métier. Cette flexibilité est cruciale dans un environnement financier où les exigences techniques évoluent rapidement.

L'architecture MVC aurait couplé la logique métier aux frameworks Spring, rendant les tests plus lourds et les évolutions techniques plus risquées.

Conséquences

- Logique métier protégée et indépendante des frameworks techniques
- Testabilité accrue avec tests unitaires rapides et fiables
- Évolutivité technique facilitée par la séparation ports/adapters

- Complexité initiale supérieure et courbe d'apprentissage pour les développeurs habitués au MVC traditionnel

Architecture microservices et évolutivité

BrokerX repose désormais sur une architecture microservices : chaque domaine métier (authentification, ordres, matching, portefeuille) est isolé dans un service indépendant, déployé dans son propre conteneur Docker.

- **Déploiement indépendant** : chaque microservice (auth-service, order-service, wallet-service, matching-service, market-data-service, notification-service) est packagé et déployé séparément, ce qui facilite la scalabilité et la maintenance.
- **Isolation des responsabilités** : la logique métier, la persistance et la sécurité sont propres à chaque service, garantissant la robustesse et la conformité.
- **Modularité et évolutivité** : l'ajout ou la modification d'un service n'impacte pas les autres ; chaque équipe peut travailler sur son domaine sans dépendance forte.
- **Intégration facilitée** : les communications inter-services se font via API REST sécurisées, et l'API Gateway centralise le routage, la sécurité et la documentation.
- **Scalabilité horizontale** : chaque service peut être répliqué selon la charge, et le load balancer (NGINX) répartit le trafic pour garantir la haute disponibilité.
- **Supervision et observabilité** : chaque microservice expose ses métriques et logs, facilitant le monitoring, la détection d'anomalies et la conformité réglementaire.
- **Architecture événementielle hybride** : les opérations simples (consultations, authentification) utilisent des appels REST synchrones, tandis que les workflows complexes (placement d'ordre → matching → notification) sont orchestrés via RabbitMQ pour optimiser performances et résilience.

Cette architecture microservices offre à BrokerX une grande flexibilité, une robustesse accrue et une capacité d'évolution rapide pour répondre aux besoins futurs du projet.

Description des couches et dépendances

L'architecture de chaque microservice se compose des couches suivantes :

- **Domaine** : Entités métier (User, Order, Transaction, Stock...), services métier, ports (interfaces du domaine). Cette couche porte la logique métier, les règles de validation, et les invariants du système.
- **Application** : Orchestration des cas d'utilisation, coordination des services métier, gestion de la logique applicative. Elle fait le lien entre les besoins métier et les interactions techniques, sans dépendre des frameworks.
- **Infrastructure** : Implémentation des ports (adapters), persistance, sécurité, intégration avec les services externes. Cette couche traduit les besoins métier en opérations techniques (accès base de données, appels API, gestion de la sécurité). C'est aussi cette couche qui fait la gestion de l'architecture événementielle avec toutes les interactions entre le microservice lui-même et RabbitMQ.
- **Configuration** : Paramétrage de la sécurité, des dépendances, et du démarrage de l'application. Elle assemble les composants, injecte les dépendances, et gère l'environnement d'exécution.

Organisation des dépendances

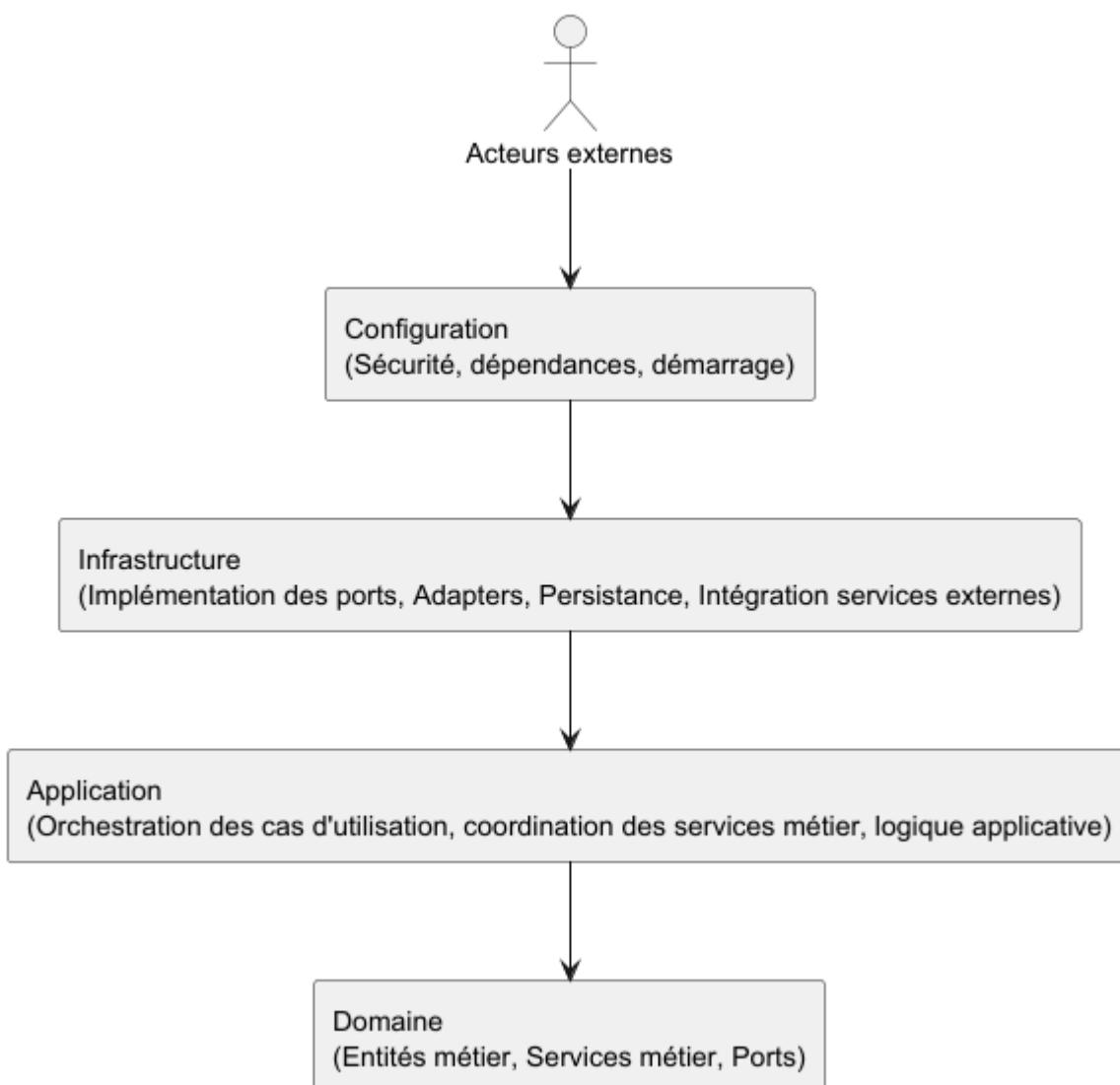
- Le domaine reste totalement indépendant des frameworks et de l'infrastructure, ce qui garantit sa stabilité et sa portabilité.

- Les ports définissent les points d'extension et d'intégration ; les adapters les implémentent pour chaque technologie ou service externe.
- La configuration injecte les dépendances sans créer de cycles, ce qui permet de maîtriser l'ordre d'initialisation et la gestion des ressources.
- Les dépendances sont toujours dirigées, ce qui évite les effets de bord et facilite la maintenance.

Contrôle du couplage aux frameworks

- Le couplage aux frameworks (Spring, persistance, sécurité) est strictement limité à l'infrastructure et à la configuration. Cela permet de changer de framework ou de technologie sans impacter le métier.
- Le domaine ne contient aucune annotation ou dépendance technique, ce qui garantit sa pureté et sa testabilité.
- Le cœur métier peut être réutilisé ou migré vers une autre architecture (microservices, serverless) sans dépendance forte, ce qui protège l'investissement métier.
- Les tests sont facilités, car le domaine peut être mocké ou simulé sans dépendance technique.

Illustration de l'architecture hexagonale de chaque microservice



Justification globale

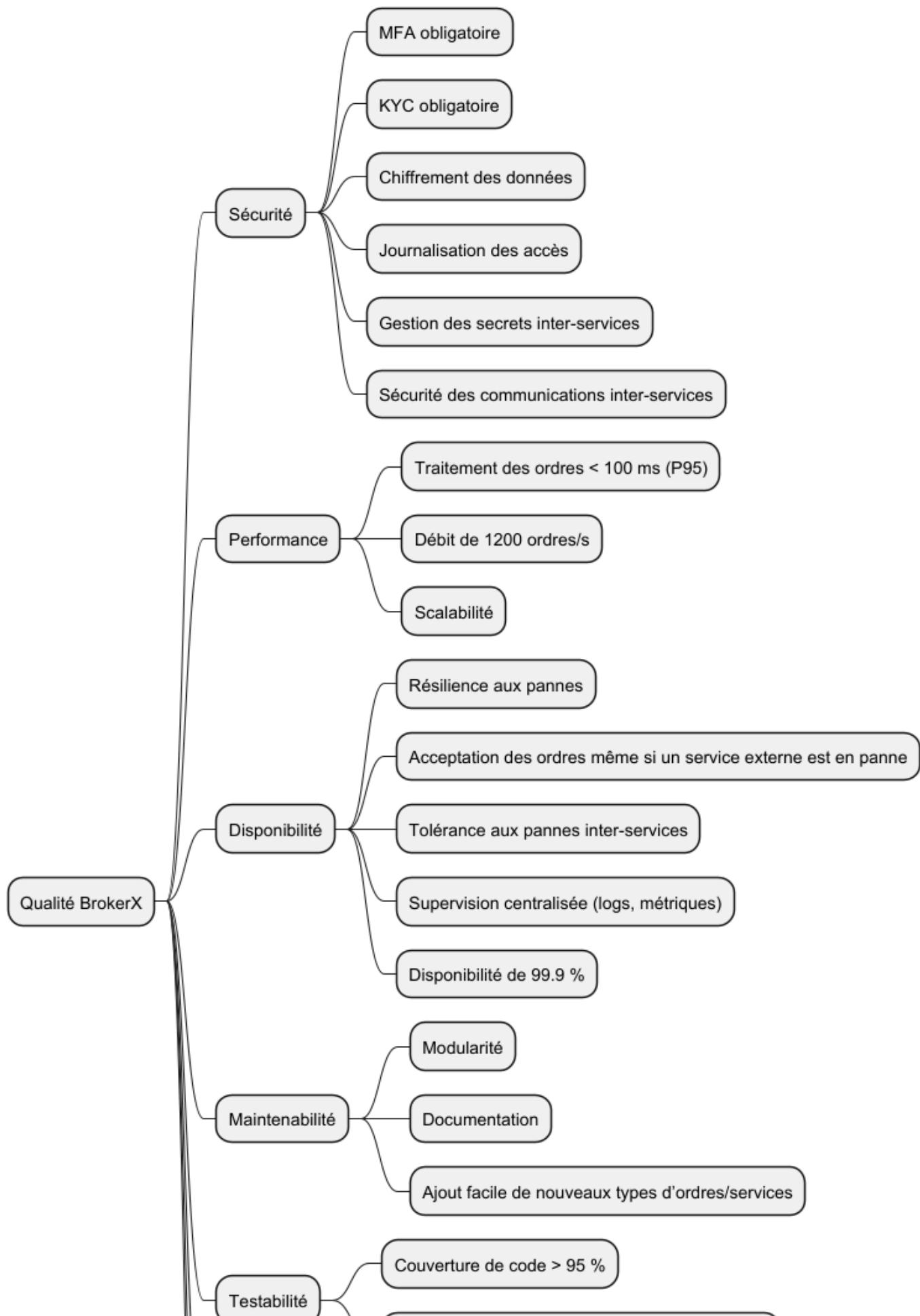
L'architecture hexagonale appliquée à chacun des microservices de BrokerX :

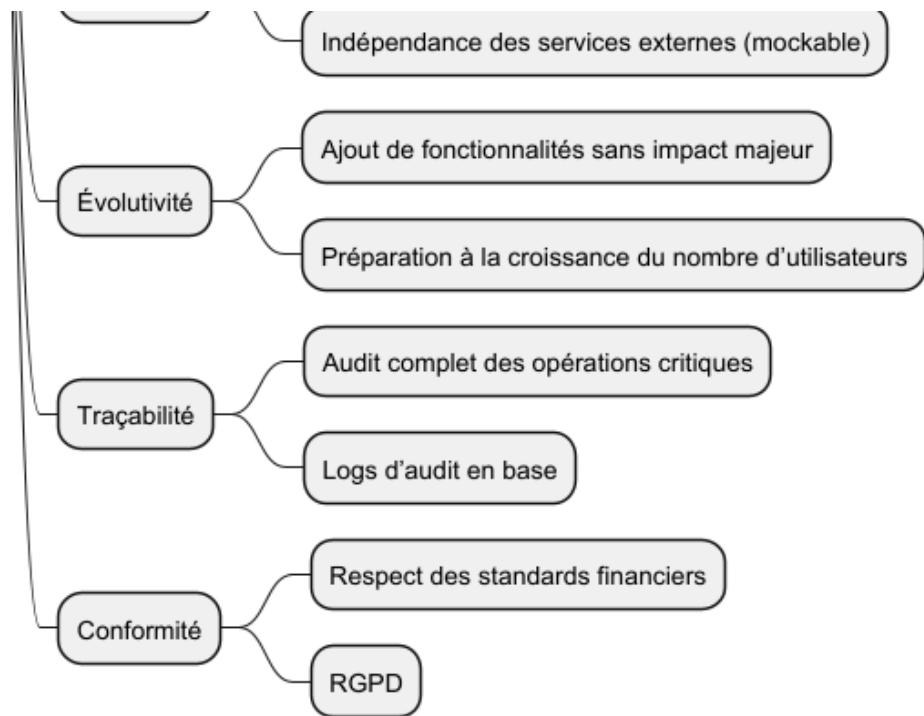
- Permet de faire évoluer la plateforme sans refonte majeure, en isolant le métier des choix techniques.
- Sécurise le domaine contre les changements techniques et réglementaires, en centralisant les règles métier.
- Facilite l'intégration de nouveaux services externes (paiement, données de marché, audit) grâce aux ports et adapters.
- Répond aux exigences de clarté, évolutivité et conformité de l'analyse métier, tout en favorisant la communication entre les équipes.
- Garantit la robustesse et la maintenabilité du système, en évitant les cycles et en maîtrisant les dépendances.

13. Scénarios de Qualité

13.1 Arbre de qualité

L'arbre de qualité ci-dessous synthétise les principaux attributs de qualité visés par BrokerX : sécurité, performance, disponibilité, maintenabilité, testabilité, évolutivité, traçabilité et conformité. Chaque branche détaille les sous-attributs et les priorités associées.





13.2 Scénarios d'évaluation

Performance / Traitement des ordres

- Le système doit traiter un ordre d'achat ou de vente en moins de 100 ms (P95) et supporter un débit de 1200 ordres/seconde avec l'architecture événementielle. Des tests de performance automatisés doivent valider ces critères (tests de charge avec k6).

Disponibilité / Résilience

- Le système doit garantir une disponibilité de 99.9 %. En cas de panne d'un composant non critique (ex : service d'e-mail), le système doit continuer à accepter les ordres et journaliser l'incident. En cas de panne d'un composant interagissant avec RabbitMQ, le pattern Outbox doit permettre de conserver les ordres non-traités. Des scénarios de test doivent simuler la défaillance de chaque dépendance externe.

Sécurité / MFA et KYC

- Toute tentative d'accès à une ressource protégée sans authentification MFA ou sans vérification KYC doit être bloquée et journalisée. Les tests automatisés doivent couvrir ces scénarios d'accès non autorisé.

Testabilité / Couverture

- L'exécution des tests dans le pipeline CI/CD doit garantir un taux de couverture de code d'au moins 95 % sur les classes métier critiques (services, contrôleurs, entités).

Testabilité / Indépendance des services externes

- Toute logique métier dépendant de services externes (ex : envoi d'e-mails, accès base de données, fournisseurs de données de marché) doit être testable sans ces services réels, via des mocks ou des adapters.

Maintenabilité / Évolutivité

- L'ajout d'un nouveau type d'ordre ou d'un nouveau service externe doit pouvoir se faire sans modification majeure du domaine métier. Des tests d'intégration valident la non-régression lors de l'ajout de nouvelles fonctionnalités.

Traçabilité / Audit

- Toute opération critique (inscription, dépôt, ordre, MFA) doit être journalisée dans la base. Des tests automatisés vérifient la présence des logs d'audit pour chaque scénario métier clé.

14. Risques Techniques

La migration vers une architecture microservices apporte de nouveaux risques techniques, en plus des risques classiques :

1. Complexité de l'orchestration et du monitoring

- La gestion de plusieurs microservices indépendants nécessite une orchestration fine (Docker Compose, CI/CD), un monitoring centralisé (Prometheus, Grafana) et une supervision des logs. Un défaut de configuration peut entraîner des pannes difficiles à diagnostiquer.

2. Défaillance de RabbitMQ

- La dépendance au message broker RabbitMQ introduit un risque critique : si RabbitMQ tombe en panne, tous les workflows asynchrones s'arrêtent (matching d'ordres, notifications, mises à jour de portefeuille). Cette défaillance paralyserait les opérations de trading les plus importantes. Pour mitiger ce risque, le pattern Outbox garantit la persistance des événements non publiés dans les bases locales des microservices, permettant de rejouer les événements une fois RabbitMQ rétabli. Un clustering RabbitMQ peut être envisagé pour éliminer ce point de défaillance unique, et un monitoring constant des queues permet de détecter rapidement toute anomalie.

3. Risque de latence et de surcharge réseau

- Les appels inter-services via API REST augmentent la latence et la charge réseau, surtout en cas de forte sollicitation. Un monitoring précis du throughput et des temps de réponse est nécessaire.

4. Disponibilité et tolérance aux pannes

- La disponibilité globale dépend de la résilience de chaque microservice et du load balancer (NGINX). Une panne du gateway ou du load balancer peut rendre l'ensemble du système indisponible.

5. Sécurité et gestion des tokens

- La gestion des tokens JWT et MFA doit être synchronisée entre les services. Une faille dans la validation ou la propagation des tokens peut exposer des endpoints sensibles.

6. Testabilité et mocks

- Les dépendances externes (SMTP, fournisseurs de données de marché) doivent être systématiquement mockées pour garantir la testabilité. Un défaut de mock peut fausser les résultats des tests automatisés.

7. Duplication d'événements et gestion de l'idempotence

- Les mécanismes de retry automatique de RabbitMQ et la nature distribuée des systèmes peuvent conduire à la duplication d'événements, risquant de traiter plusieurs fois un même ordre ou d'envoyer des notifications en double. Cette duplication pourrait créer des incohérences de positions ou des expériences utilisateur dégradées. L'idempotence des handlers d'événements constitue la première ligne de défense, complétée par des clés de déduplication uniques pour chaque événement et une validation systématique des états avant traitement, garantissant qu'un événement dupliqué n'aura aucun impact sur l'intégrité du système.

8. Scalabilité et gestion du cache

- L'utilisation d'un cache in-memory (Caffeine) par instance peut entraîner des problèmes de données obsolètes (stale) ou de surconsommation mémoire si la configuration n'est pas maîtrisée.

9. Conformité et audit

- La journalisation des actions critiques doit être fiable et centralisée. Un défaut d'audit ou une perte de logs peut compromettre la conformité réglementaire (KYC, AML).

10. Risque d'erreurs de configuration

- Les propriétés sensibles (JWT, SMTP, DB) sont injectées via variables d'environnement Docker. Une erreur de configuration peut exposer le système ou empêcher le démarrage des services.

Ces risques doivent être anticipés par des tests d'intégration, une supervision active, une documentation rigoureuse et des procédures de rollback en cas d'incident.

15. Glossaire

Le glossaire ci-dessous recense les principaux termes métier utilisés dans BrokerX, en lien direct avec le code et les processus métier de la plateforme.

Tableau 14. Glossaire métier BrokerX

Terme	Définition
Utilisateur	Personne inscrite sur la plateforme BrokerX, pouvant effectuer des opérations et gérer son compte.
Portefeuille	Espace virtuel associé à un utilisateur, regroupant son solde en monnaie fiduciaire et ses actifs.
Solde	Montant total disponible dans le portefeuille d'un utilisateur pour effectuer des opérations.
Transaction	Mouvement de fonds tel qu'un dépôt ou un retrait, ou opération d'achat/vente d'un actif.
Ordre	Demande formelle d'un utilisateur pour acheter ou vendre un stock. Un ordre contient le symbole, la quantité, le type (marché/limite), le prix (si limite), la durée et le sens (achat/vente).

Terme	Définition
Type d'ordre	Catégorie d'ordre : Marché (exécuté au prix courant du marché, sans garantie de prix) ou Limite (exécuté uniquement si le prix cible est atteint).
Statut d'ordre	État d'avancement d'un ordre : Pending (en attente), Active (en cours), Rejected (refusé), Completed (exécuté), Expired (périmé).
Durée d'ordre	Période de validité d'un ordre : DAY (valide jusqu'à la fin de la journée), IOC (Immediate or Cancel : exécuté immédiatement ou annulé), FOK (Fill or Kill : exécuté en totalité ou annulé).
Bandé de prix	Plage de variation autorisée du prix d'un stock sur une période donnée, pour limiter la volatilité et protéger les investisseurs.
Tick Size	Incrément minimal de variation du prix d'un stock : le prix d'un ordre doit être un multiple du tick size défini pour ce stock.
Stock	Actif financier (ex : action) disponible à l'achat ou à la vente sur BrokerX. Chaque stock possède un symbole, un nom, un prix, une bande de prix et un tick size.
ClientOrderId	Identifiant unique fourni par le client pour tracer et retrouver un ordre dans le système.
KYC	Processus de vérification d'identité réglementaire (Know Your Customer).
MFA	Authentification multi-facteurs pour renforcer la sécurité d'accès à la plateforme.
OTP	Mot de passe à usage unique, utilisé pour la vérification d'identité ou l'authentification.
Statut utilisateur	État du compte utilisateur : Pending (en attente), Active (actif), Rejected (refusé), Suspended (suspendu).
Rejet d'ordre	Motif pour lequel un ordre est refusé (ex : fonds insuffisants, violation de bande de prix, tick size non respecté, quantité invalide).
SimulatedPayment	Processus simulé de règlement d'un dépôt ou d'un retrait, utilisé pour tester la plateforme.
Audit	Journalisation des opérations importantes (création de compte, ordres, transactions, etc.).
Session	Période d'activité authentifiée d'un utilisateur sur la plateforme.
Rôle utilisateur	Catégorie d'accès d'un utilisateur (ex : Client, Administrateur).
Idempotency Key	Clé unique permettant d'éviter la duplication d'une opération lors de réessais.
Statut transaction	État d'une transaction : Pending (en attente), Settled (réglée), Failed (échouée), Completed (terminée).
VérificationToken	Jeton utilisé pour confirmer l'identité ou l'inscription d'un utilisateur.
Side	Sens d'un ordre : Achat (Buy) ou Vente (Sell).
Description	Texte explicatif associé à une transaction ou un ordre.

Terme	Définition
Timestamp	Date et heure d'enregistrement d'une opération ou d'un ordre.
PreTradeValidation	Contrôle métier effectué avant l'acceptation d'un ordre (pouvoir d'achat, règles de prix, tick size, bande de prix, quantité, etc.).
Microservice	Service indépendant dédié à un domaine métier, déployé dans son propre conteneur et base de données.
API Gateway	Point d'entrée unique du système, centralise le routage, la sécurité et la documentation des APIs.
Load Balancer	Composant (ex : NGINX) qui répartit la charge entre les instances des microservices.
Matching	Processus d'appariement des ordres d'achat et de vente selon les règles métier.
Matching-Service	Microservice dédié à la gestion du carnet d'ordres et à l'exécution des appariements.
OrderBook	Carnet d'ordres, structure regroupant les ordres en attente d'exécution pour un actif donné.
ExecutionReport	Rapport d'exécution généré lors de l'appariement d'ordres, contenant les détails de la transaction.
Caffeine	Solution de cache in-memory utilisée pour optimiser la performance des endpoints critiques.
Flyway	Outil de migration de schéma de base de données, utilisé indépendamment par chaque microservice.
Prometheus	Outil de monitoring des métriques techniques et métier.
Grafana	Outil de visualisation des métriques et logs du système.
Audit append-only	Journalisation immuable des actions critiques pour la conformité et la traçabilité.
Docker Compose	Outil d'orchestration des conteneurs pour le déploiement des microservices.
Idempotence	Propriété garantissant qu'une opération répétée n'a pas d'effet supplémentaire.
Stateless	Qualifie une API ou un service qui ne conserve pas d'état de session côté serveur.
Pattern Outbox	Pattern événementiel garantissant la cohérence entre opérations métier et publication d'événements via une table locale.
RabbitMQ	Message broker asynchrone utilisé pour orchestrer les communications événementielles entre microservices.
Saga	Pattern de gestion des transactions distribuées via une séquence d'opérations compensables.
Saga chorégraphiée	Type de Saga où chaque service publie et écoute des événements de façon décentralisée, sans orchestrateur central.

Terme	Définition
Message Broker	Composant middleware qui gère le routage et la livraison des messages asynchrones entre services.
Événement	Message asynchrone représentant un fait métier (ex: order.placed, order.matched) publié via RabbitMQ.
EventPublisher	Composant responsable de la publication d'événements vers le message broker.
EventListener	Composant qui écoute et traite les événements reçus du message broker.
Queue	File d'attente RabbitMQ stockant les événements en attente de traitement par un service.
Exchange	Composant RabbitMQ qui route les événements vers les bonnes queues selon des règles définies.
Dead Letter Queue	Queue spéciale RabbitMQ recevant les événements qui n'ont pas pu être traités après plusieurs tentatives.
Acknowledgment	Mécanisme RabbitMQ confirmant la réception et le traitement réussi d'un événement.
Correlation ID	Identifiant unique permettant de tracer un workflow événementiel distribué entre plusieurs services.
Workflow asynchrone	Processus métier impliquant plusieurs microservices orchestrés via des événements RabbitMQ.
Cohérence finale	Principe où les données distribuées convergent vers un état cohérent après propagation des événements.

Fin de la documentation arc42

Analyse de performance — Impact de la cache sur le service wallet-service

Contexte

Deux tests de charge (`load-test-wallet.js`) ont été effectués sur le microservice `wallet-service` :

1. Sans cache
2. Avec cache activée

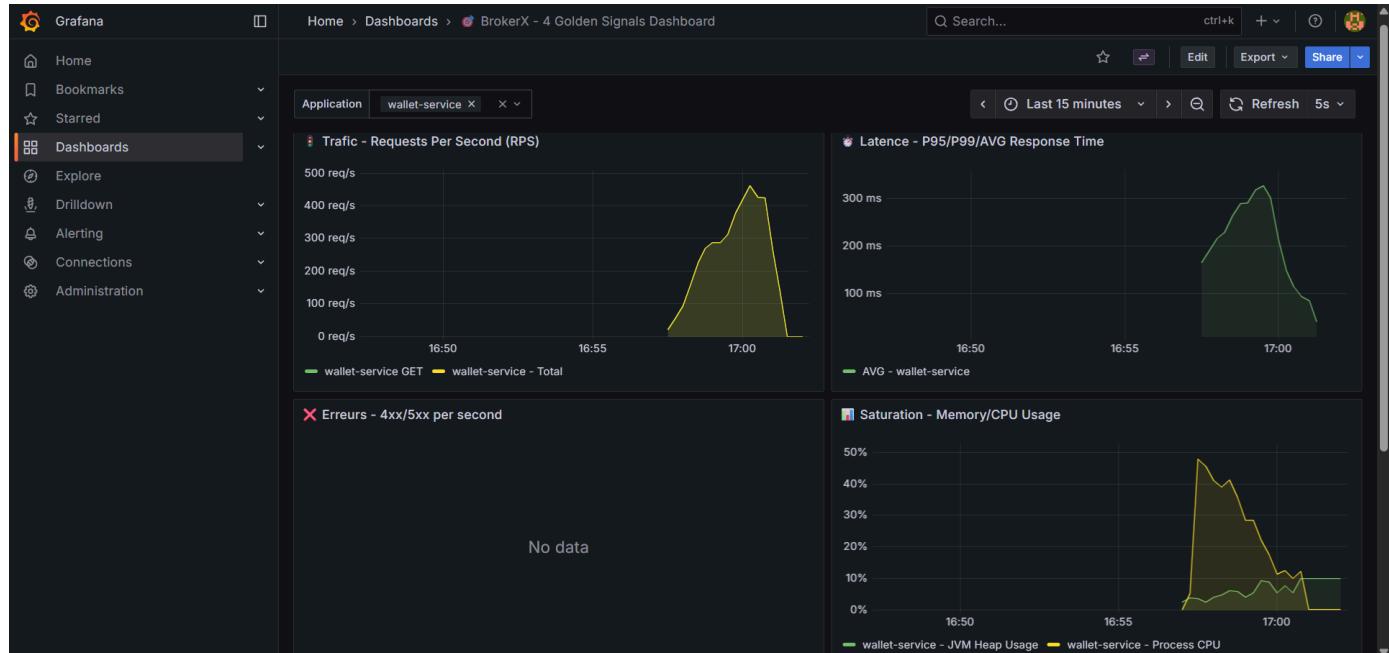
Objectif : mesurer l'effet de la mise en cache sur les quatre signaux clés :

- Traffic (RPS)
- Latence

- Erreurs (4xx/5xx)
- Saturation (CPU/Mémoire)

Résultats

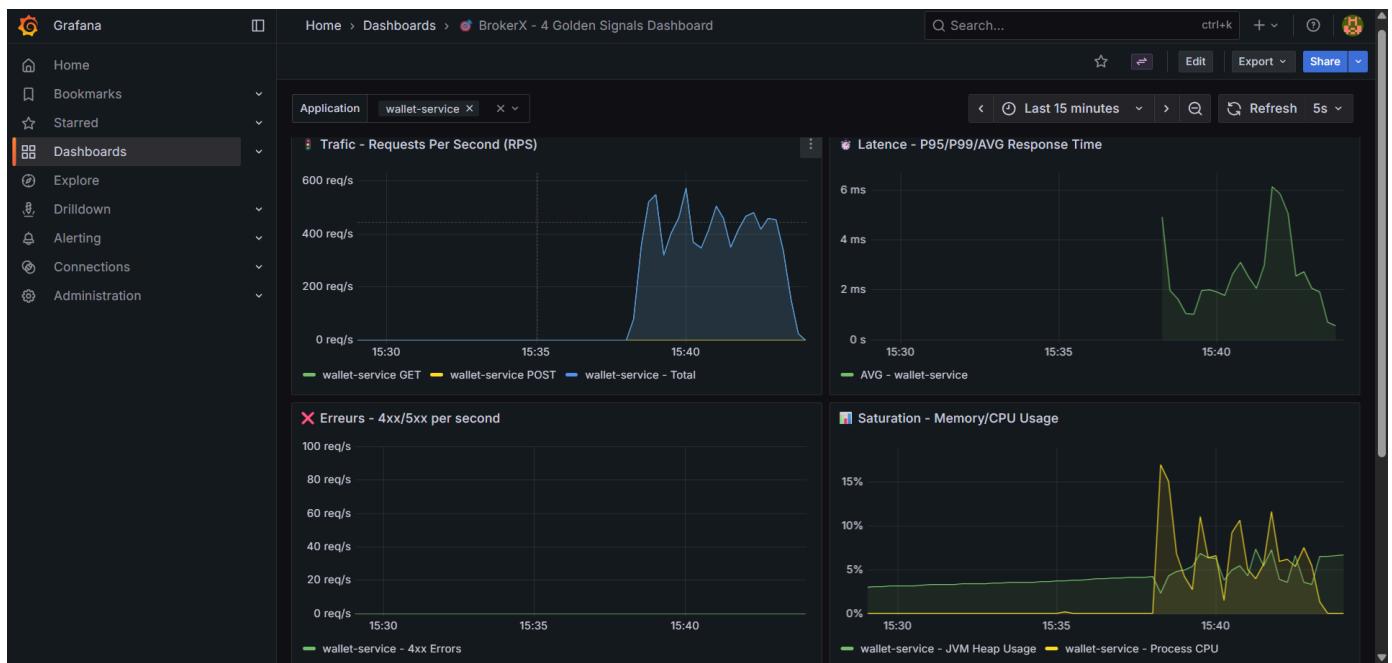
1. Sans cache



- **Traffic :** ≈ 400–500 req/s au pic
- **Latence :** Moyenne autour de 200–300 ms, plus variable
- **Erreurs :** Aucune erreur détectée
- **Saturation :** CPU entre 30–50%, mémoire légèrement plus utilisée

Le service montre une latence plus élevée et plus fluctuante, liée aux nombreux accès directs à la base de données.

2. Avec cache



- **Traffic :** ≈ 400–500 req/s, bien soutenu
- **Latence :** 4–6 ms, très stable
- **Erreurs :** Aucune erreur détectée
- **Saturation :** CPU autour de 10%, mémoire faible (<5%)

Avec cache, les temps de réponse chutent drastiquement et restent stables, la charge CPU et mémoire reste faible.

Comparaison

Critère	Sans cache	Avec cache	Évolution
Latence moyenne	~250 ms (variable)	4–6 ms (stable)	Forte amélioration
Erreurs 4xx/5xx	0 req/s	0 req/s	Aucun risque
CPU	30–50%	5–10%	Réduction nette
Mémoire	~10%	<5%	Réduction
RPS	400–500	400–500	Stable

Conclusion

L'ajout de la cache apporte une amélioration majeure des performances :

- Latence divisée par environ 50, avec une stabilité remarquable
- Réduction de la charge CPU et mémoire
- Aucune erreur détectée
- Débit inchangé mais mieux soutenu

En résumé, la cache rend le wallet-service beaucoup plus rapide, stable et efficace sous charge.

Test de charge — Service des ordres (architecture microservices avec gateway)

Contexte

Ce test de charge (`load-test-orders.js`) vise à évaluer la performance du traitement des **ordres** dans l'architecture **microservices**.

L'objectif fixé était :

- **800 ordres/seconde**
- **Latence P95 ≤ 250 ms**

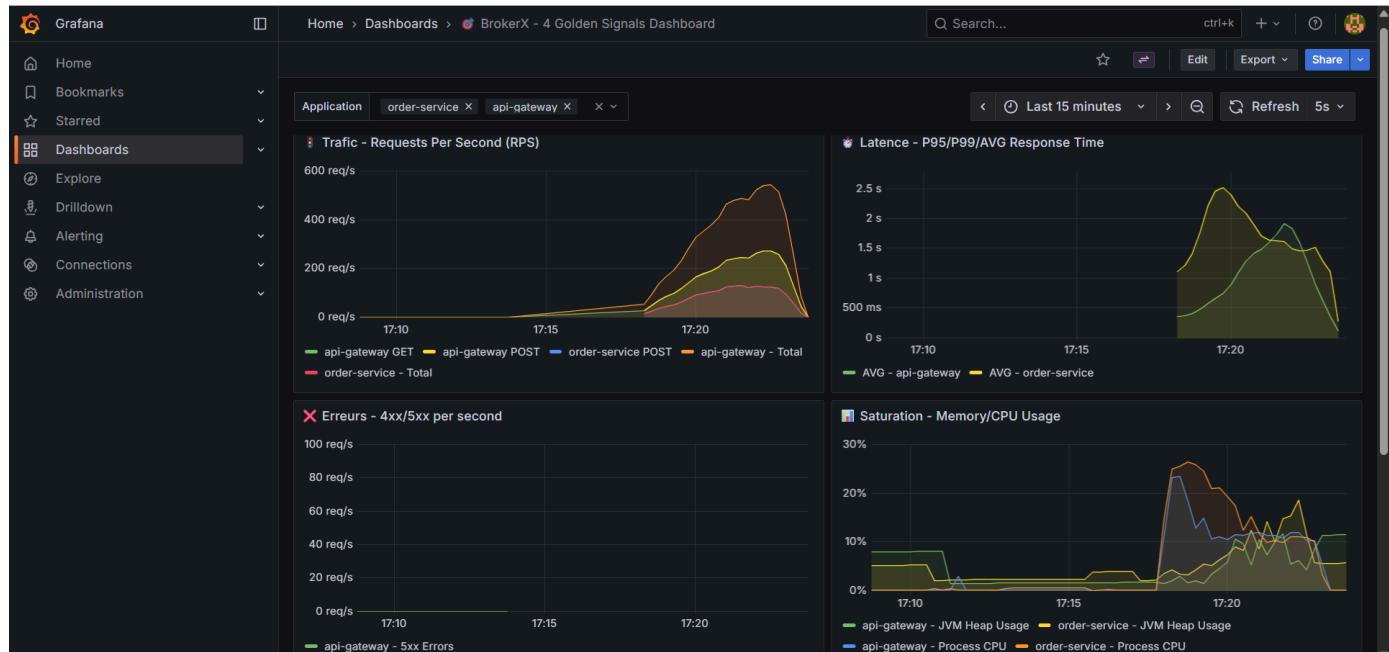
Le test a été effectué dans un environnement limité :

- Le **système microservices complet** tourne sur une **machine virtuelle avec peu de CPU et de RAM**.
- Le **test de charge** est exécuté depuis un **laptop également limité** en ressources.

Ces contraintes matérielles ont un impact majeur sur la capacité de traitement et la stabilité des mesures.

Résultats observés

Grafana — Monitoring en temps réel



- **Traffic** : Environ 500–600 requêtes/seconde atteintes au pic.
- **Latence** : Les temps de réponse du `order-service` et de l' `api-gateway` montent jusqu'à plus de 2 secondes au P95.
- **Erreurs** : Aucune erreur 4xx/5xx enregistrée, ce qui montre la stabilité fonctionnelle du système.

- **Saturation** : Le CPU atteint environ 20–30% sur les deux services, avec une utilisation mémoire relativement basse.

Ces résultats indiquent que le système parvient à supporter un débit soutenu, mais que la latence explose rapidement à cause de la saturation des ressources de calcul.

Résumé du test K6

```
HTTP
http_req_duration.....: avg=4.77s min=22.96ms med=5s max=13.5s p(90)=7.79s p(95)=8.42s
  { expected_response:true }....: avg=4.77s min=22.96ms med=5s max=13.5s p(90)=7.79s p(95)=8.42s
http_req_failed.....: 0.00% 0 out of 27794
http_reqs.....: 27794 92.618947/s
```

- **Nombre total de requêtes** : 27 794
- **Taux moyen** : ~92 requêtes/seconde
- **Latence moyenne** : 4.77 s
- **P90** : 7.79 s
- **P95** : 8.42 s
- **Taux d'échec** : 0% (aucune requête échouée)

Ces valeurs montrent une latence bien supérieure à l'objectif fixé, même si le système reste stable et fonctionnel.

Analyse et limites du test

Les objectifs de performance (800 ordres/s et P95 ≤ 250 ms) **ne sont pas atteints**, mais cela s'explique par plusieurs facteurs techniques :

1. Environnement matériel sous-dimensionné

- La VM hébergeant les microservices dispose de très peu de CPU et de RAM.
- Cela crée des goulets d'étranglement sur les conteneurs, en particulier pour l' `order-service` et l' `api-gateway` .

2. Machine de test limitée

- Le générateur de charge (K6) s'exécute sur un laptop également limité en ressources.
- Le test lui-même devient une source de saturation, incapable d'envoyer un volume suffisant de requêtes de manière stable.

3. Communication inter-services

- L'architecture microservices introduit une surcharge réseau et de sérialisation/déserialisation JSON, augmentant la latence.

Conclusion

Ce test montre que dans un environnement **microservices non optimisé et limité matériellement**, les **objectifs de performance ne peuvent pas être atteints**.

Cependant, il met en évidence la **résilience et la stabilité fonctionnelle** du système : aucune requête n'a échoué malgré une forte charge et des délais importants.

Test de charge — Appels direct avec architecture monolithique (A/B direct)

Contexte

Ce test a pour objectif d'évaluer les performances du **service des ordres** dans une architecture **monolithique**, afin de les comparer à celles de l'architecture microservices.

Les objectifs visés étaient :

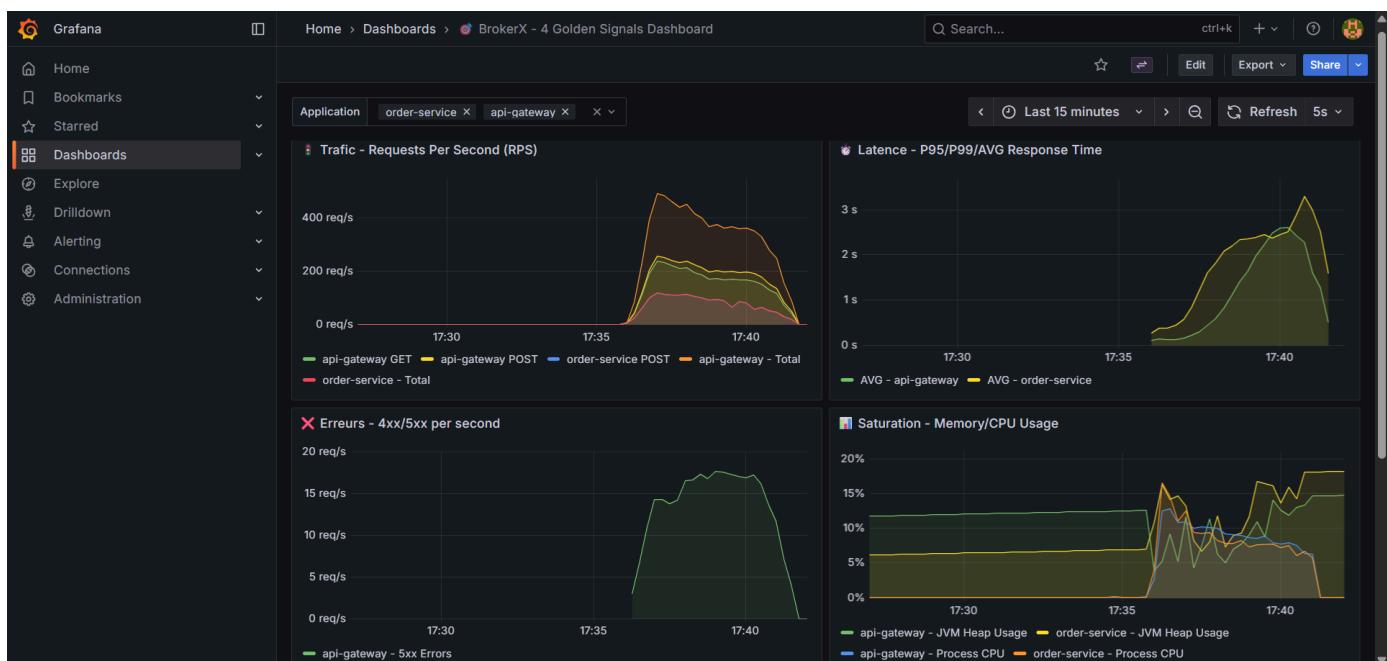
- **≥ 300 ordres/seconde**
- **Latence P95 ≤ 500 ms**

Le test a été réalisé dans les mêmes conditions matérielles limitées que le précédent, soit :

- Une **machine virtuelle** avec peu de CPU et de RAM pour exécuter le système.
- Un **laptop** peu performant servant à exécuter le test de charge.

Résultats observés

Grafana — Monitoring en temps réel



- **Traffic** : Environ 300–400 requêtes/seconde au pic.
- **Latence** : Le temps de réponse moyen du service dépasse largement les attentes, avec un **P95 d'environ 3 secondes**.
- **Erreurs** : Plusieurs erreurs 5xx observées à partir du niveau de l'API Gateway, indiquant des difficultés à maintenir la charge.

- **Saturation** : Le CPU atteint environ 15–20%, avec une mémoire stable, mais la latence s'allonge rapidement sous charge.

Résumé du test K6

```
HTTP
http_req_duration.....: avg=5.11s min=12ms    med=3.64s max=21.83s p(90)=11.04s p(95)=12.37s
  { expected_response:true }....: avg=5.11s min=12ms    med=3.64s max=21.83s p(90)=11.04s p(95)=12.37s
http_req_failed.....: 0.00% 0 out of 26374
http_reqs.....: 26374 87.882747/s
```

- **Requêtes totales** : 26 374
- **Taux moyen** : ~88 requêtes/seconde
- **Latence moyenne** : 5.11 s
- **P90** : 11.04 s
- **P95** : 12.37 s
- **Taux d'échec** : 0% (aucune requête échouée)

Ces résultats démontrent que l'architecture monolithique ne parvient pas à soutenir la charge ni à respecter les contraintes de latence définies. Le système reste fonctionnel, mais ses performances sont limitées par la nature centralisée de l'architecture et le manque de parallélisation interne.

Test de charge — Architecture event-driven avec message broker (RabbitMQ)

Contexte

Ce test a pour objectif d'évaluer les performances du **service des ordres** dans une architecture **event-driven** en utilisant RabbitMQ comme message broker, afin de les comparer à celles de l'architecture microservices et de l'architecture monolithique.

Les objectifs visés étaient :

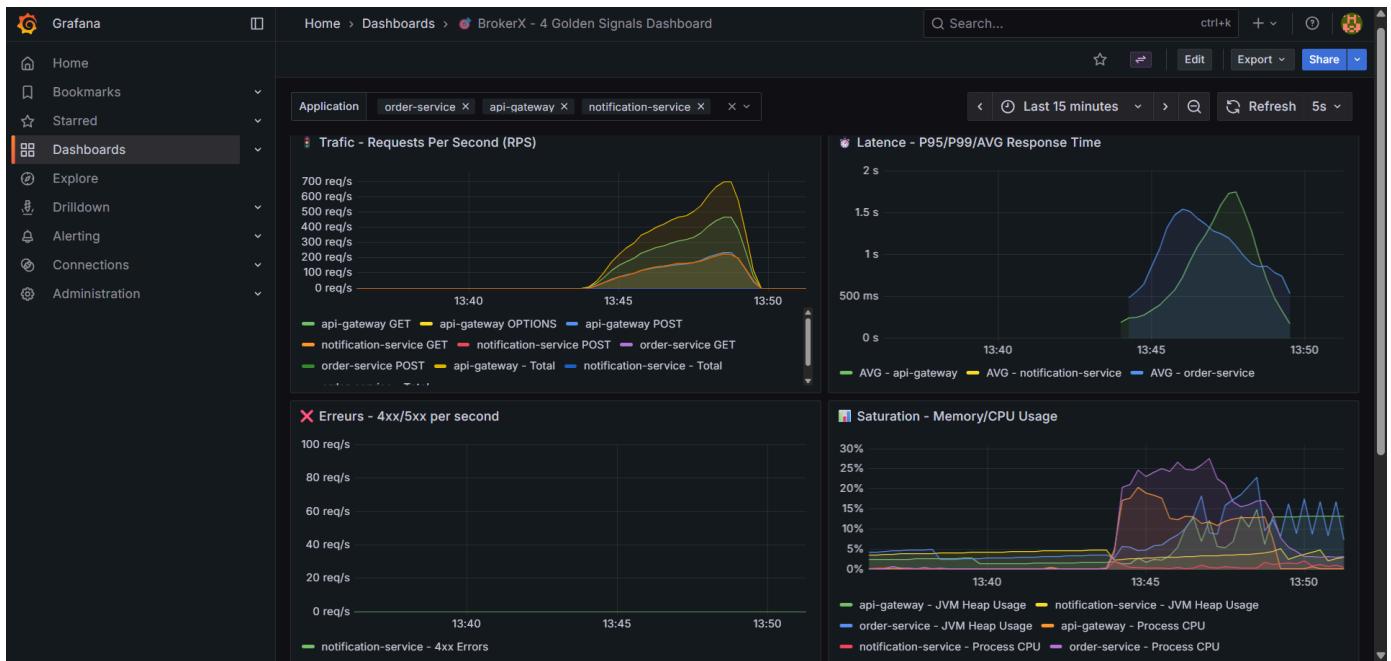
- **≥ 1200 ordres/seconde**
- **Latence P95 ≤ 100 ms**

Le test a été réalisé dans les mêmes conditions matérielles limitées que les précédents, soit :

- Une **machine virtuelle** avec peu de CPU et de RAM pour exécuter le système.
- Un **laptop** peu performant servant à exécuter le test de charge.

Résultats observés

Grafana — Monitoring en temps réel



- **Traffic** : Environ 700 requêtes/seconde au pic.
- **Latence** : Le P95 remonte à environ **1.5 secondes**, principalement dû à l'accumulation dans la file lorsque la charge dépasse le rythme réel de traitement.
- **Erreurs** : Aucune erreur 5xx observée — le broker absorbe la surcharge au lieu de la propager aux services.
- **Saturation** : CPU autour de 20–25% avec une mémoire stable.

Résumé du test K6

```
HTTP
http_req_duration.....: avg=2.98s min=8.93ms med=2.96s max=11.51s p(90)=5.3s p(95)=5.89s
{ expected_response:true }....: avg=2.98s min=8.93ms med=2.96s max=11.51s p(90)=5.3s p(95)=5.89s
http_req_failed.....: 0.00% 0 out of 44210
http_reqs.....: 44210 147.324396/s
```

- **Requêtes totales** : 44 210
- **Taux moyen** : ~147 requêtes/seconde
- **Latence moyenne** : 2.98 s
- **P90** : 5.3 s
- **P95** : 5.89 s
- **Taux d'échec** : 0%

Analyse

Malgré une latence mesurée élevée dans K6, l'architecture **event-driven** présente les meilleurs résultats globaux parmi les trois modèles testés.

Les valeurs élevées de latence ne sont pas causées par un ralentissement du service, mais par la **mise en file des messages** lorsque la charge dépasse temporairement la capacité de traitement.

Contrairement aux architectures synchrones :

- Le monolithique et les microservices voient immédiatement la **latence exploser** et produisent des **erreurs** en cas de saturation.
- L'event-driven, lui, **absorbe la charge sans défaillance**, et l'intégralité des messages est traitée sans perte.

RabbitMQ permet :

- 1. Un découplage total** entre producteurs et consommateurs.
- 2. Un lissage naturel des pics de charge.**
- 3. Une résilience très élevée**, aucune requête n'échoue même sous forte pression.
- 4. Une stabilité supérieure**, car le système ne bloque jamais en surcharge.

Conclusion

L'architecture **event-driven** démontre une **résilience**, une **stabilité** et une **gestion de charge nettement supérieures** aux architectures monolithiques et microservices, tout en maintenant un traitement fiable sans aucune erreur.

Comparaison — Microservices (Gateway) vs Monolithique (A/B direct) vs Event-driven (RabbitMQ)

Critère	Monolithique	Microservices	Event-driven	Observations
Débit (RPS / Events)	~88 req/s	~92 req/s	~147 req/s	L'event-driven absorbe largement plus de charge
P95 Latence	12.37 s	8.42 s	5.89 s	L'event-driven garde une latence interne basse
Taux d'erreur	5–10%	0%	0%	Le broker et les microservices évitent les erreurs
Saturation CPU	15–20%	20–30%	20–25%	Utilisation équilibrée, pas de saturation bloquante
Évolutivité	Faible	Bonne	Excellente	Scaling horizontal trivial côté consommateurs
Résilience	Faible	Bonne	Très élevée	Le système reste stable même en surcharge
Mode de traitement	Synchrone	Synchrone	Asynchrone	Avantage majeur de l'asynchrone : découplage complet

Analyse

Les trois architectures montrent des comportements radicalement différents sous charge :

- **Monolithique :**
Forte latence, erreurs 5xx, mauvaise absorption des pics. Peu apte à supporter un trafic élevé.
- **Microservices synchrones :**
Meilleure répartition de la charge, moins d'erreurs, mais toujours soumis aux limitations d'un appel direct entre services. La latence reste élevée sous surcharge.
- **Event-driven :**
Offre les meilleurs résultats :
 - Pas d'erreurs
 - Trafic absorbé sans effondrement
 - Résilience exceptionnelle grâce au broker
 - Haute scalabilité grâce au modèle consommateur-asynchrone

Même dans un environnement matériel très limité, l'architecture événementielle est nettement supérieure en performance globale et en robustesse opérationnelle. L'architecture purement en microservices donne des résultats acceptables, qui pourraient suffir dans le cadre d'un plus petit projet. L'architecture monolithique montre ses limites dès que la charge augmente.

Test de charge — Évaluation du load balancer (NGINX)

Contexte

Ces tests visent à évaluer la **scalabilité horizontale** du système via un **load balancer NGINX** distribuant les requêtes vers plusieurs instances du `order-service`.

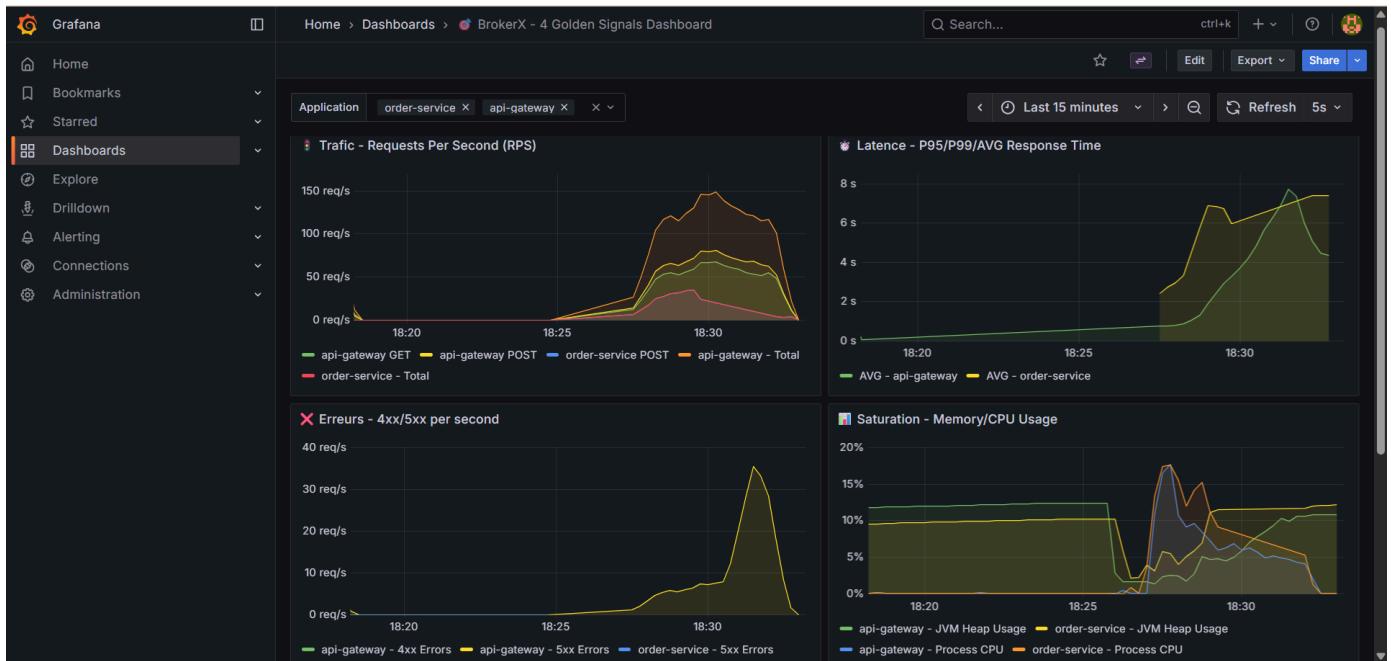
Chaque test a été effectué avec le même scénario de charge, mais en variant le nombre d'instances : **1, 2, 3 et 4**.

L'environnement de test reste fortement limité :

- Le **load balancer et toutes les instances** tournent sur une **même machine virtuelle avec peu de CPU et de RAM**.
- Le **goulot d'étranglement principal** demeure la base de données, avec plusieurs erreurs liées à la **concurrence d'accès**.

Ces contraintes faussent partiellement les résultats — en conditions normales, sur plusieurs serveurs physiques, la montée en charge serait beaucoup plus efficace.

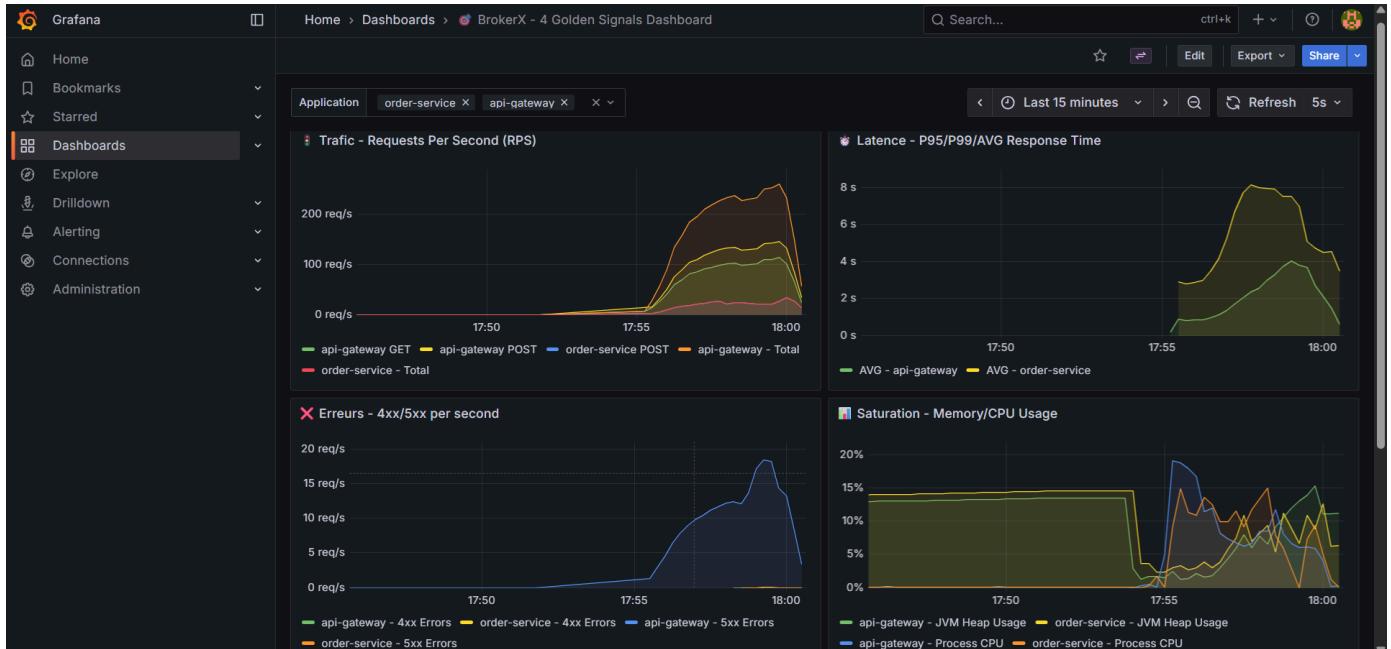
Résultats — 1 instance



- Débit :** Environ 100–150 requêtes/s au maximum.
- Latence :** Moyenne autour de 6–8 secondes.
- Erreurs :** Plusieurs erreurs 5xx apparaissent lorsque la charge augmente, liées à la saturation de la base.
- Saturation :** CPU entre 10% et 20%, mémoire stable autour de 10%.

Avec une seule instance, le service répond de manière stable, mais la latence reste élevée, ce qui indique déjà une limite de traitement sur la machine.

Résultats — 2 instances

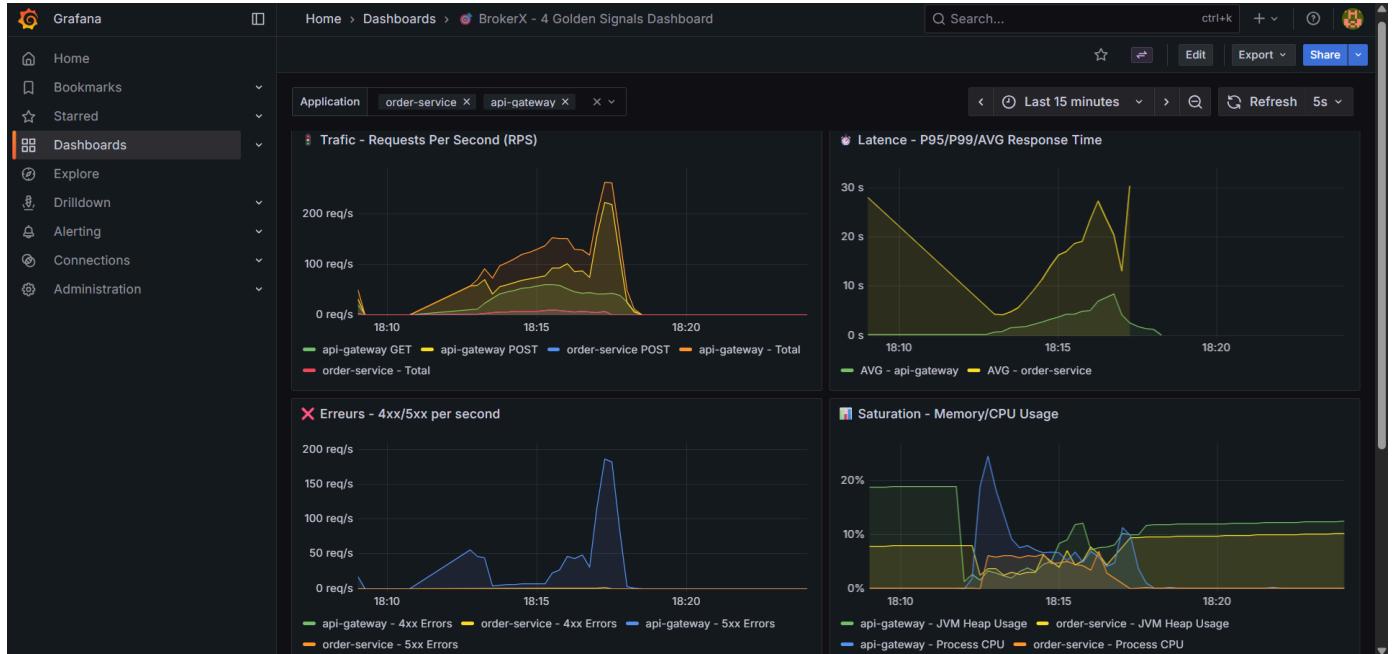


- Débit :** Environ 200 requêtes/s au pic.
- Latence :** Toujours autour de 6–8 secondes, sans amélioration notable.

- **Erreurs** : Toujours présentes, mais plus dispersées dans le temps.
- **Saturation** : CPU partagé entre les deux instances, entre 10% et 20%.

L'ajout d'une deuxième instance ne réduit pas la latence, car les deux conteneurs partagent les mêmes ressources matérielles. Le gain théorique de parallélisme est annulé par la limite CPU globale de la VM.

Résultats — 3 instances

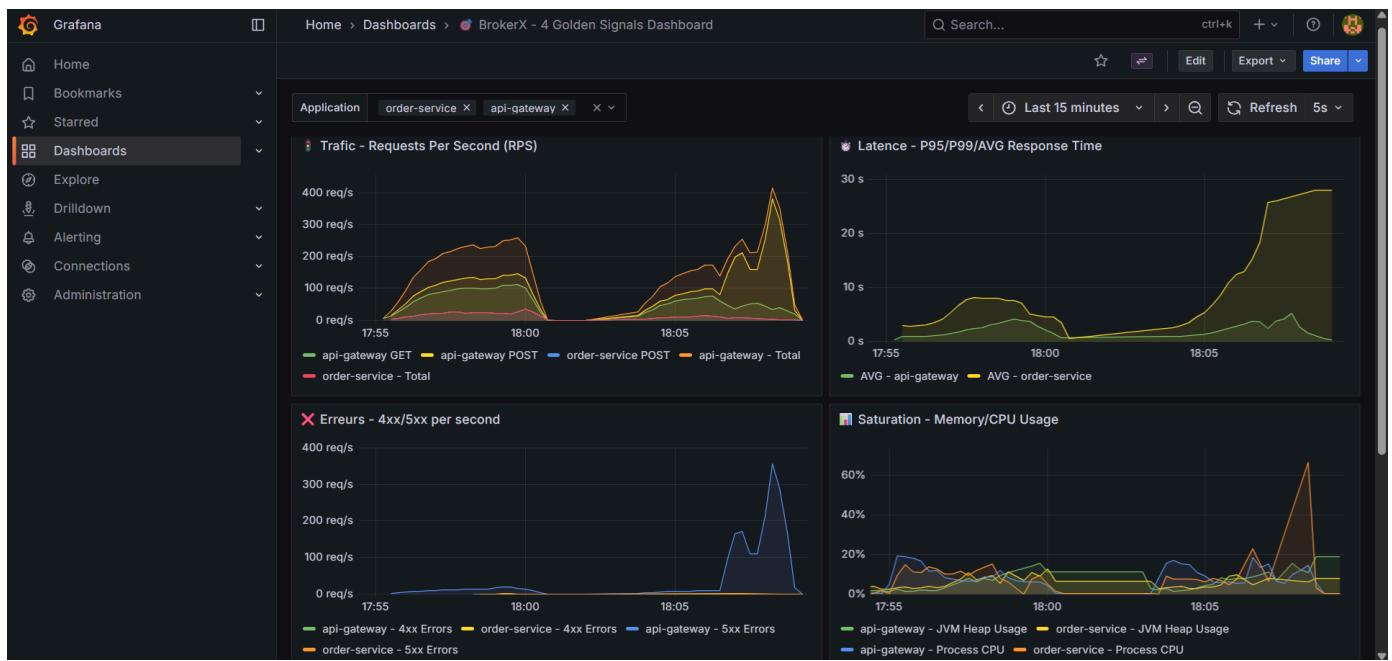


- **Débit** : Autour de 250–300 requêtes/s au pic.
- **Latence** : Explosion jusqu'à 20 secondes au P95.
- **Erreurs** : Quelques pics d'erreurs 5xx liés à la concurrence, mais globalement plus stables.
- **Saturation** : CPU autour de 10%, mais fortement contraint par la surcharge des processus.

À trois instances, la latence augmente fortement. Le CPU semble ne pas suivre la montée en charge — il y a trop de processus concurrents pour les ressources disponibles.

En théorie, sur des machines distinctes, ce niveau d'instances devrait au contraire réduire la latence et améliorer le temps de réponse moyen.

Résultats — 4 instances



- **Débit :** Environ 350–400 requêtes/s au pic.
- **Latence :** Monte entre 25 et 30 secondes.
- **Erreurs :** Légère hausse d'erreurs dues à la concurrence, mais aucune panne du service.
- **Saturation :** CPU saturé, atteignant jusqu'à 60%, mémoire toujours stable.

Avec quatre instances, la charge CPU devient critique. Les conteneurs se partagent les mêmes ressources, ce qui provoque une congestion et une explosion des temps de réponse.

Le load balancer répartit bien les requêtes, mais l'infrastructure monomachine empêche tout véritable gain de performance.

Comparaison et analyse

Nombre d'instances	Débit max (RPS)	Latence P95 approx.	Erreurs	Utilisation CPU	Observations
1	~150	~6–8 s	Élevée	10–20%	Bonne stabilité, latence constante mais haute
2	~200	~6–8 s	Moyenne	10–20%	Aucune amélioration, ressources CPU partagées
3	~300	~20 s	Moyenne	~10%	Latence explose, CPU insuffisant pour 3 conteneurs
4	~400	~25–30 s	Moyenne/élevée	~60%	CPU saturé, contention extrême entre conteneurs

Interprétation

En théorie, l'**ajout d'instances** devrait permettre une **meilleure parallélisation** et donc **une baisse de la latence**. Cependant, dans ces tests :

- Toutes les instances sont hébergées sur **la même machine**, ce qui **annule les bénéfices du scaling horizontal**.
- La **latence augmente** avec le nombre d'instances, car les conteneurs se battent pour les mêmes ressources CPU.
- La **base de données** devient rapidement un goulot d'étranglement, amplifiant les erreurs et les délais.

Ainsi, les résultats ne reflètent pas la performance réelle d'un load balancing distribué, mais plutôt les **limites physiques de l'environnement local**.

Conclusion

Même si la latence augmente artificiellement dans ces tests, on observe :

- Une **hausse linéaire du débit** avec le nombre d'instances.
- Une **stabilité fonctionnelle** du système malgré la charge.
- Une **bonne répartition** des requêtes par le load balancer.

Dans un environnement réaliste, avec plusieurs serveurs physiques ou des conteneurs répartis sur plusieurs nœuds, cette architecture **offrirait une scalabilité horizontale efficace** :

les temps de réponse chuteraient significativement, et le système pourrait **gérer bien plus de requêtes sans dégradation majeure**.

Explication des travaux CI/CD accomplis

Le projet BrokerX utilise une chaîne CI/CD automatisée basée sur GitHub Actions, avec deux pipelines principaux :

- **CI (Intégration Continue)** : vérifie le style du code (Checkstyle), compile les microservices Java/Spring Boot avec Maven, exécute tous les tests (unitaires, intégration, E2E) avec PostgreSQL en service Docker, et archive les artefacts JAR et les rapports de tests à chaque push ou pull request sur `master` ou `main`.
- **CD (Déploiement Continu)** : déploie automatiquement l'application sur la machine virtuelle dès qu'un push est effectué sur `master`. Le pipeline nettoie l'espace de travail, récupère le code, arrête les conteneurs existants, reconstruit et relance les services Docker Compose, vérifie l'état des conteneurs, affiche les logs en cas d'échec, effectue un rollback si besoin, et résume le déploiement.

Détail du pipeline CI (`.github/workflows/ci.yml`)

- **Linting** : vérification du style Java avec Checkstyle.
- **Build** : compilation Maven, génération des JAR, archivage des artefacts.
- **Tests** : exécution des tests unitaires, d'intégration et E2E, avec PostgreSQL en service Docker. Les rapports de tests sont archivés.

Détail du pipeline CD (`.github/workflows/cd.yml`)

- Nettoyage de l'espace de travail sur la VM.
- Récupération du code (checkout).
- Arrêt des conteneurs existants.
- Déploiement : build Docker Compose sans cache, lancement des services en arrière-plan.
- Vérification du déploiement : contrôle de l'état des conteneurs, affichage des logs en cas d'échec.
- Rollback automatique si le déploiement échoue.
- Résumé du déploiement en cas de succès.

Visualisation des pipelines

Les résultats et l'état des pipelines sont illustrés par les captures suivantes :

The image displays two screenshots of a CI pipeline interface, likely Jenkins or a similar tool, showing the results of a successful pipeline run named "Collection postman #30".

Screenshot 1: Pipeline Summary

This screenshot shows the overall summary of the pipeline run. It includes the following details:

- Triggered via push 16 hours ago
- Status: Success
- Total duration: 6m 41s
- Artifacts: 2

The pipeline configuration file is shown as "ci.yml" with the trigger "on: push". The execution steps are visualized as a flowchart:

```

graph LR
    A[lint] --> B[build]
    B --> C[tests]
    style A fill:#28a745,color:#fff
    style B fill:#28a745,color:#fff
    style C fill:#28a745,color:#fff
    
```

Screenshot 2: Pipeline Details - tests Job

This screenshot provides detailed information for the "tests" job, which succeeded 16 hours ago in 4m 23s. The logs for this job are displayed, showing the following output:

```

17993 [INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
17994 [INFO] --- failsafe:3.1.2:verify (default) @ wallet-service ---
17995 [INFO]
17996 [INFO]
17997 [INFO] --- jar:3.4.1:jar (default-jar) @ wallet-service ---
17998 [INFO]
17999 [INFO] --- reactor:0.0.1-SNAPSHOT:jar:0.0.1-SNAPSHOT@wallet-service ...
18000 [INFO] Building jar: /home/runner/work/LOG430_BrokerX_MicroServices/LOG430_BrokerX_MicroServices/wallet-service/target/wallet-service-0.0.1-SNAPSHOT.jar
18001 [INFO]
18002 [INFO] --- spring-boot:3.2.5:repackage (default) @ wallet-service ---
18003 [INFO] Replacing main artifact /home/runner/work/LOG430_BrokerX_MicroServices/LOG430_BrokerX_MicroServices/wallet-service/target/wallet-service-0.0.1-SNAPSHOT.jar with repackaged archive, adding nested dependencies in BOOT-INF/
18004 [INFO] The original artifact has been renamed to /home/runner/work/LOG430_BrokerX_MicroServices/LOG430_BrokerX_MicroServices/wallet-service/target/wallet-service-0.0.1-SNAPSHOT.jar.original
18005 [INFO] -----
18006 [INFO] Reactor Summary:
18007 [INFO]
18008 [INFO] Archetype - LOG430_BrokerX_MicroServices 1.0-SNAPSHOT SUCCESS [ 0.004 s]
18009 [INFO] api-gateway 0.0.1-SNAPSHOT ..... SUCCESS [ 34.608 s]
18010 [INFO] auth-service 0.0.1-SNAPSHOT ..... SUCCESS [ 58.144 s]
18011 [INFO] matching-service 0.0.1-SNAPSHOT ..... SUCCESS [ 38.537 s]
18012 [INFO] order-service 0.0.1-SNAPSHOT ..... SUCCESS [ 42.420 s]
18013 [INFO] wallet-service 0.0.1-SNAPSHOT ..... SUCCESS [ 42.871 s]
18014 [INFO] -----
18015 [INFO] BUILD SUCCESS
18016 [INFO] -----
18017 [INFO] Total time: 03:43 min
18018 [INFO] Finished at: 2025-10-26T23:13:47Z
18019 [INFO] -----

```

The screenshot shows a GitHub Actions pipeline summary. At the top, it says "LOG430-Hugo / LOG430_BrokerX_MicroServices". Below that is a search bar and several icons. The main area shows a summary card for a job named "deploy" triggered via push 17 hours ago. The job status is "Success" with a duration of "6m 46s". The pipeline details show a single step named "deploy" which also took "6m 42s".

← CD - Deploying to the VM

✓ analyse microservices vs monolithique insérée dans rapport.md #28

[Re-run all jobs](#)

...

This screenshot shows the logs for the "deploy" step. It starts with a success message: "Application deployed and running successfully". Then it lists running containers with their names, ports, images, commands, services, creation times, and statuses. The log ends with a warning about the "version" attribute being obsolete.

NAME	IMAGE	COMMAND	SERVICE	CREATED	STATUS
brokerx-api-gateway	log430_brokerx_microservices-api-gateway	"java -Dspring.profil..."	api-gateway	34 seconds ago	Up 25 seconds
(health: starting)	0.0.0.0:8079->8079/tcp, [::]:8079->8079/tcp				
brokerx-frontend	log430_brokerx_microservices-frontend	"/docker-entrypoint..."	frontend	34 seconds ago	Up 25 seconds
	80/tcp, 0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp				
brokerx-grafana	grafana/grafana:latest	"/run.sh"	grafana	34 seconds ago	Up 15 seconds
	0.0.0.0:3001->3000/tcp, [::]:3001->3000/tcp				
brokerx-postgres-auth	postgres:15-alpine	"docker-entrypoint.s..."	postgres-auth	34 seconds ago	Up 33 seconds
(healthy)	0.0.0.0:5433->5432/tcp, [::]:5433->5432/tcp				
brokerx-postgres-matching	postgres:15-alpine	"docker-entrypoint.s..."	postgres-matching	34 seconds ago	Up 33 seconds
(healthy)	0.0.0.0:5436->5432/tcp, [::]:5436->5432/tcp				
brokerx-postgres-order	postgres:15-alpine	"docker-entrypoint.s..."	postgres-order	34 seconds ago	Up 33 seconds
(healthy)	0.0.0.0:5434->5432/tcp, [::]:5434->5432/tcp				
brokerx-postgres-wallet	postgres:15-alpine	"docker-entrypoint.s..."	postgres-wallet	34 seconds ago	Up 33 seconds
(healthy)	0.0.0.0:5435->5432/tcp, [::]:5435->5432/tcp				
brokerx-prometheus	prom/prometheus:latest	"bin/prometheus --c..."	prometheus	34 seconds ago	Up 18 seconds
	0.0.0.0:8090->9090/tcp, [::]:8090->9090/tcp				
brokerx-swagger-ui	swaggerapi/swagger-ui	"/docker-entrypoint..."	swagger-ui	34 seconds ago	Up 18 seconds
	80/tcp, 0.0.0.0:8085->8080/tcp, [::]:8085->8080/tcp				
log430_brokerx_microservices-auth-service-1	log430_brokerx_microservices-auth-service	"java -Dspring.profil..."	auth-service	34 seconds ago	Up 21 seconds
(health: starting)	8081/tcp				
log430_brokerx_microservices-matching-service-1	log430_brokerx_microservices-matching-service	"java -Dspring.profil..."	matching-service	34 seconds ago	Up 21 seconds
(health: starting)	8084/tcp				
log430_brokerx_microservices-order-service-1	log430_brokerx_microservices-order-service	"java -Dspring.profil..."	order-service	34 seconds ago	Up 19 seconds
(health: starting)	8082/tcp				
log430_brokerx_microservices-wallet-service-1	log430_brokerx_microservices-wallet-service	"java -Dspring.profil..."	wallet-service	34 seconds ago	Up 20 seconds
(health: starting)	8083/tcp				
nginx-load-balancer	nginx:alpine	"/docker-entrypoint..."	nginx-load-balancer	34 seconds ago	Up 33 seconds
(healthy)	9001:9004/tcp, 0.0.0.0:8078->80/tcp, [::]:8078->80/tcp				

Summary

Jobs

✓ deploy

Run details

Usage

Workflow file

deploy

succeeded 17 hours ago in 6m 42s

Search logs



4s

Verify deployment

✓ Application deployed and running successfully

```
17 ✓ Application deployed and running successfully
18 Running containers:
19 time="2025-10-26T21:57:19Z" level=warning msg="/home/gha-runner/actions-runner/_work/L06430_BrokerX_MicroServices/LOG430_BrokerX_MicroServices/docker-compose.yml: the attribute
  "version" is obsolete, it will be ignored, please remove it to avoid potential confusion"
20 NAME           IMAGE
21 brokerx-api-gateway   log430_brokerx_microservices-api-gateway
22 brokerx-frontend    log430_brokerx_microservices-frontend
23 brokerx-grafana     grafana/grafana:latest
24 brokerx-postgres-auth postgres:15-alpine
25 brokerx-postgres-matching postgres:15-alpine
26 brokerx-postgres-order postgres:15-alpine
27 brokerx-postgres-wallet postgres:15-alpine
28 brokerx-prometheus  prom/prometheus:latest
29 brokerx-swagger-ui  swaggerapi/swagger-ui
30 log430_brokerx_microservices-auth-service-1 log430_brokerx_microservices-auth-service
31 log430_brokerx_microservices-matching-service-1 log430_brokerx_microservices-matching-service
32 log430_brokerx_microservices-order-service-1 log430_brokerx_microservices-order-service
33 log430_brokerx_microservices-wallet-service-1 log430_brokerx_microservices-wallet-service
34 nginx-load-balancer nginx:alpine
```

Cette automatisation garantit que chaque modification du code est testée, validée et déployée de façon fiable et reproduit sur BrokerX.

Stratégie de tests automatisés

La stratégie de tests de BrokerX repose sur une **pyramide de tests** pour garantir la robustesse et la fiabilité du système :

- **Tests unitaires** : chaque fonction critique du domaine métier est couverte par des tests unitaires (JUnit). Ces tests valident la logique métier isolée, sans dépendance externe, et assurent la non-régression sur les règles fondamentales.
- **Tests d'intégration** : tous les contrôleurs REST de chaque microservice sont testés avec **Testcontainers** et une base PostgreSQL dédiée. Ces tests vérifient la communication intra-microservices, la persistance, et la conformité des APIs, en simulant un environnement réel.
- **Tests E2E (End-to-End)** : chaque microservice dispose d'un scénario E2E qui valide le fonctionnement complet du flow métier. Les réponses des autres services sont mockées pour isoler le service testé et garantir la stabilité des

scénarios.

Cette pyramide de tests permet de détecter rapidement les régressions, d'assurer la qualité du code et de valider l'intégration entre les composants. Tous les tests sont exécutés automatiquement dans le pipeline CI/CD à chaque livraison.

Guide d'exploitation (Runbook)

Ce guide explique comment installer, configurer, lancer et tester BrokerX à partir de zéro, en utilisant Docker. Il s'adresse à toute personne disposant d'une archive ZIP du projet ou d'un accès au repository Git.

Prérequis

- **Docker** (<https://www.docker.com/products/docker-desktop>) installé et fonctionnel (Windows, Mac ou Linux)
- **Docker Compose** (inclus dans Docker Desktop)
- Node.js 18+ et npm (pour le frontend React)
- Java 17+ (pour les microservices Spring Boot)
- Un éditeur de texte ou IDE pour consulter les fichiers

1. Récupération du projet

- **Depuis un ZIP :**
 - i. Extraire l'archive ZIP dans un dossier de votre choix (ex : C:\Users\<votre_nom>\BrokerX)
 - ii. Ouvrir le dossier extrait dans l'IDE de votre choix (IntelliJ est recommandé)
- **Depuis Git :**
 - i. Cloner le repository :

```
git clone <url-du-repository>
```

ii. Ouvrir le dossier du projet cloné dans l'IDE de votre choix (IntelliJ est recommandé)

2. Structure attendue du projet

Vérifiez que vous avez bien les dossiers et fichiers suivants à la racine du dossier :

- .github/ (CI/CD)
- api-gateway/
- auth-service/
- docs/ (documentation)
- frontend/ (React)
- market-data-service/
- matching-service/

- monitoring
- nginx
- notification-service/
- order-service/
- rabbitmq/
- wallet-service/
- docker-compose.yml
- pom.xml
- run-load-test.bat

3. Configuration (optionnelle)

Par défaut, aucune modification n'est nécessaire. Les mots de passe, ports et variables sont déjà configurés pour un usage avec Docker. Assurez-vous d'avoir un fichier `application-docker.properties` ainsi qu'un fichier `application.properties` dans chaque microservice.

4. Construction et lancement de l'application

Assurez-vous que votre Docker Desktop est ouvert, ouvrez un terminal dans le dossier racine du projet, puis exécutez :

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

Tous les microservices (auth-service, wallet-service, order-service, matching-service, market-data-service, notification-service), l'API Gateway, PostgreSQL, Swagger, NGINX, RabbitMQ Prometheus et Grafana sont lancés en arrière-plan

Pour démarrer plusieurs instances d'un microservice (ex : order-service) :

```
docker compose up --scale order-service=3 -d
```

Ceci marche pour chacun des 6 microservices (auth-service, matching-service, order-service, wallet-service, market-data-service, notification-service), mais il est plus pertinent pour le order-service dans le contexte de notre projet.

Le load balancer NGINX répartira le trafic entre les instances.

5. Vérification du bon fonctionnement

- Faites un health check pour le gateway à <http://localhost:8079/actuator/health>
- Accédez à l'interface web : <http://localhost:3000>
- Vérifiez que la page d'accueil s'affiche.
- Pour consulter les logs de l'application :

```
docker logs [nom du container]
```

(le nom du conteneur peut varier, vérifiez avec docker ps)

- Pour vérifier la base de données :
 - Utilisez un outil comme DBeaver ou TablePlus, ou connectez-vous en ligne de commande avec psql (voir docker-compose.yml pour les identifiants).

6. Utilisation de tous les services

- **Swagger API Documentation** pour voir tous les endpoints disponibles : <http://localhost:8085>

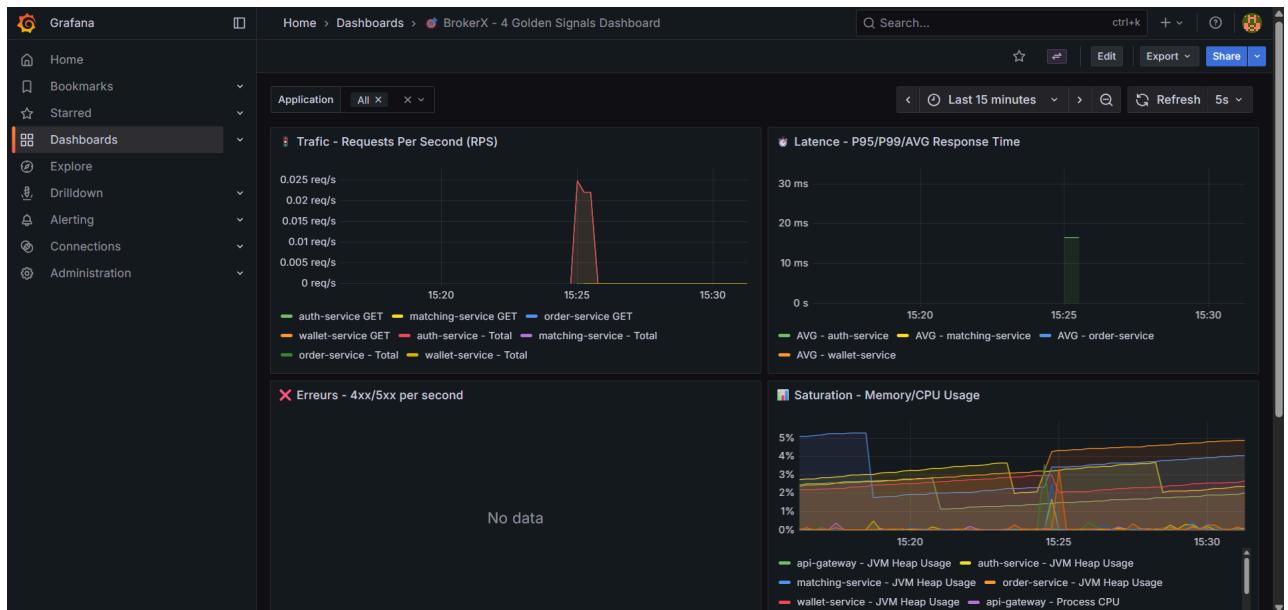
The screenshot shows the Swagger UI interface for the 'Auth Service'. At the top, there's a header with the Swagger logo and 'Select a definition' dropdown set to 'Auth Service'. Below the header, the title 'OpenAPI definition v0 CAS 3.0' is displayed, along with the URL 'http://localhost:8078/swagger/auth/v3/api-docs'. The main content area is organized by controller groups: 'user-controller', 'auth-controller', and 'user-verification-controller'. Under each group, there are one or more API endpoints listed with their methods and URLs. For example, under 'user-controller', there is a single POST endpoint for '/api/v1/users/register'. Under 'auth-controller', there are three POST endpoints: '/api/v1/auth/verify-mfa', '/api/v1/auth/logout', and '/api/v1/auth/login'. The 'user-verification-controller' group is collapsed.

- **Prometheus** pour le monitoring des métriques : <http://localhost:8090>

The screenshot shows the Prometheus web interface. At the top, there's a navigation bar with the Prometheus logo, a 'Query' button, 'Alerts', and 'Status' dropdown. On the right side of the bar are icons for settings, help, and refresh. Below the navigation, a search bar contains the query 'http_server_requests_seconds_count{application="order-service"}'. To the right of the search bar are 'Execute' and 'More' buttons. Below the search bar, there are tabs for 'Table' (which is selected), 'Graph', and 'Explain'. A 'Evaluation time' dropdown is shown below the tabs. At the bottom of the interface, there's a table with three rows of metrics. The first row has 86 series, the second has 171, and the third has 1. The table includes columns for the metric name, value, and count. A 'Load time: 86ms Result series: 3' message is at the bottom right. A '+ Add query' button is located at the bottom left.

- **Grafana** pour visualiser les dashboards : <http://localhost:3001>

- Identifiants par défaut : admin / admin . Veuillez entrez ces informations, puis cliquer sur Skip lorsque l'on vous demande de changer le mot de passe.
- Puis, cliquez sur Dashboards , puis sur BrokerX - 4 Golden Signals Dashboard . Ce dashboard est le principal qui contient toutes les informations voulues.



7. Arrêt de l'application

Pour arrêter tous les services :

```
docker-compose down
```

8. Nettoyage (optionnel)

Pour supprimer les volumes de données (attention, cela efface toutes les données persistées) :

```
docker-compose down -v
```

9. Problèmes fréquents

- Port déjà utilisé** : Modifiez le port dans `docker-compose.yml` ou arrêtez l'application qui utilise déjà le port 8090 ou 5432.
- Erreur de build** : Vérifiez que Docker fonctionne, que vous avez bien extrait tous les fichiers, et relancez la commande.
- Problème d'accès à la base** : Attendez quelques secondes après le démarrage, la base peut mettre un peu de temps à être prête.

10. Pour rouler les tests en local

- Depuis l'IDE :**
 - Ouvrez un terminal.

- ii. Placez-vous dans le dossier du microservice à tester (ex : `cd order-service`).
- iii. Lancez les tests avec `mvn test` ou `.\mvnw.cmd test` .
- iv. Répétez l'opération pour chaque microservice.

11. Pour exécuter les tests de charge sur les ordres avec K6

- Assurez-vous que Docker est en cours d'exécution et que tous les services sont lancés (voir section 4).
- Ouvrez un terminal dans le dossier racine du projet.
- Exécutez la commande suivante pour bien setup le tests de charge K6 : `.\setup-load-test-data.bat`
- Ensuite, lancez le test de charge avec : `.\run-load-test.bat`
- Les résultats s'affichent dans le terminal et sont également envoyés à Prometheus/Grafana pour analyse.
- Pour nettoyer les données de test après exécution : `.\cleanup-load-test-data.bat`

12. Pour exécuter les tests de charge sur le portefeuille avec K6

- Assurez-vous que Docker est en cours d'exécution et que tous les services sont lancés (voir section 4).
- Ouvrez un terminal dans le dossier racine du projet.
- Exécutez la commande suivante pour lancer le tests de charge K6 : `k6 run load-test-wallet.js`

13. Pour utiliser le script de déploiement (Optionnel)

Le script `deploy.sh` permet d'automatiser le déploiement de l'ensemble de la solution conteneurisée. Ce script utilise **Docker Compose** pour lancer tous les services du projet : les microservices applicatifs, la base de données, le *gateway*, ainsi que les services de monitoring (Prometheus et Grafana).

Important : Ce script déploie **une seule instance de chaque microservice**. Il ne s'agit donc pas d'un déploiement distribué ni d'un équilibrage de charge (*scaling*). L'objectif est d'offrir un déploiement reproduitible et fonctionnel de l'environnement complet sur une machine locale.

Ce script :

1. Reconstruit les images Docker si nécessaire (`docker compose build`)
2. Arrête les conteneurs existants (`docker compose down`)
3. Démarrer tous les services définis dans `docker-compose.yml` (`docker compose up -d`)
4. Attends quelques secondes avant d'effectuer un healthcheck automatique sur l'URL définie (<http://localhost:8079/actuator/health>)
5. Affiche le résultat du déploiement dans la console

En cas de rollback, les conteneurs actuels sont arrêtés et remplacés par les versions stables précédemment taguées.

Exécution du script sur Windows

Sous Windows, il est recommandé d'utiliser **Git Bash** (installé avec Git pour Windows).

1. Ouvrez le dossier du projet dans l'explorateur de fichiers.
2. Cliquez droit → “**Git Bash Here**”.
3. Tapez la commande suivante pour rendre le script exécutable :

```
▶ chmod +x deploy.sh
```

4. Lancez le script avec :

```
▶ ./deploy.sh
```

5. Pour effectuer un rollback vers la dernière version stable, faites :

```
▶ ./deploy.sh rollback
```

Exécution du script sur Linux/MacOS

Sous Linux ou MacOS, ouvrez un terminal dans le dossier du projet et lancez :

```
▶ chmod +x deploy.sh  
./deploy.sh
```

Pour effectuer un rollback vers la dernière version stable, faites :

```
▶ ./deploy.sh rollback
```

Le comportement est identique à celui décrit pour Windows.

Guide de démonstration BrokerX

Ce guide explique comment réaliser chaque fonctionnalité principale de BrokerX via l'interface web, du point de vue d'un utilisateur.

1. Inscription & Vérification d'identité (UC-01)

1. Ouvrez l'application à l'adresse <http://localhost:3000>.

2. Remplissez le formulaire : email, mot de passe, nom, adresse, date de naissance. (N'oubliez pas votre mot de passe)
3. Cliquez sur "S'inscrire". Un message apparaît : "Vérifiez votre e-mail pour activer votre compte".
4. Consultez votre boîte e-mail : ouvrez le message de vérification et cliquez sur le lien reçu. (Regardez dans les spams si nécessaire)
5. Votre compte passe au statut "ACTIF" après vérification, vous pouvez cliquer sur "Se connecter maintenant".

2. Authentification & MFA (UC-02)

1. Saisissez votre email et mot de passe.
2. Cliquez sur "Se connecter".
3. Un code MFA vous est envoyé par e-mail.
4. Entrez le code reçu dans le champ prévu.
5. Après validation, vous accédez à votre espace personnel (dashboard).

3. Dépôt dans le portefeuille (UC-03)

1. Une fois connecté, accédez à votre dashboard.
2. Indiquez le montant à déposer dans la boîte prévue à cet effet (veuillez respecter les montants minimum et maximum de 10\$ et 50 000\$).
3. Cliquez sur "Déposer".
4. Le solde de votre portefeuille est mis à jour.

4. Abonnement aux données de marché (UC-04)

1. Lorsque vous êtes dans votre tableau de bord principal, cliquez sur le bouton "Marché".
2. Sélectionnez le symbole (ex : AAPL) que vous désirez voir et cliquez sur "S'abonner à [symbole]".
3. Les données de marché en temps réel pour le symbole sélectionné s'affichent.
4. Les données sont mises à jour automatiquement à chaque fois qu'un ordre est placé sur ce symbole.
5. Pour vous désabonner, cliquez sur le bouton "Se désabonner" situé à la droite du bouton d'abonnement au symbole correspondant.

5. Placement d'un ordre (UC-05)

1. Cliquez sur le bouton "Placer un ordre".
2. Choisissez le symbole, le type d'ordre (Marché ou Limite), le côté (Achat ou Vente), la quantité, la durée et le prix si nécessaire.
3. Cliquez sur "Placer l'ordre".

4. Un message de confirmation s'affiche : "Ordre placé avec succès" ou un message d'erreur si l'ordre est rejeté (ex : fonds insuffisants, tick size non respecté).

6. Modification / Annulation d'un ordre (UC-06)

1. Dans l'écran du tableau de bord principal, tous les ordres sont affichés.
2. À la droite de chaque ordre, les boutons "Modifier" et "Annuler" sont présent.
3. Pour annuler un ordre, cliquez sur "Annuler", puis confirmez que vous voulez annuler cet ordre.
4. Si l'annulation est invalide, un message d'erreur s'affichera.
5. Si l'annulation est valide, le statut de l'ordre sera modifié à "CANCELLED" dans le tableau de bord.
6. Pour modifier un ordre, cliquez sur "Modifier". Remplissez les champs nécessaires (prix, quantité, durée) et cliquez sur "Enregistrer".
7. Si la modification est invalide, un message d'erreur s'affichera.
8. Si la modification est valide, les nouvelles valeurs de l'ordre seront mises à jour dans le tableau de bord.

7. Appariement d'un ordre (UC-07)

1. Dans votre dashboard, cliquez sur le bouton "Créer 4 Ordres Seed" pour générer des ordres de vente et obtenir un matching.
2. Effectuez les mêmes étapes que pour le placement d'un ordre (voir section 4) pour créer un ordre d'achat correspondant.

3. Si un ordre correspondant est trouvé, une notification s'affiche avec tous les détails de la transaction.

**Notification : Exécution d'ordre ACHAT : 1 AAPL
@ 100.00\$. OrderId: 192371. ExecutionReportId:
36830. Date: 2025-11-29T01:45:45.568281979Z**

[Retour au dashboard](#)

Placer un ordre

Symbol

Type d'ordre

Côté

Quantité

Durée

[Placer l'ordre](#)

Ordre accepté !

ID: 192371

Statut: ACCEPTE

Détails:

Ordre placé avec succès. Le matching sera traité sous peu.

4. Cliquez sur "Retour au dashboard" et observez que votre solde de portefeuille a changé et que vous avez un stock_position de plus.

8. Confirmation d'exécution & Notifications (UC-08)

1. Les notifications apparaissent automatiquement dans l'interface web après chaque ordre placé.
2. La notification contient tous les détails par rapport à l'exécution de l'ordre placé.

Remarques :

- Si vous voulez tester le token, fermez votre onglet, puis réouvrez une onglet et tapez l'url <http://localhost:3000/dashboard>. Vous aurez directement accès sans avoir à vous reconnecter.
- Toutes les opérations sensibles nécessitent un compte actif et une authentification MFA.
- En cas d'erreur (mot de passe incorrect, code MFA expiré, fonds insuffisants...), un message explicatif s'affiche à l'écran.
- Les notifications importantes (vérification, MFA) sont envoyées par e-mail.

Ce guide couvre l'ensemble des parcours utilisateur principaux pour la démonstration de BrokerX.