Rapport final BrokerX

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
Université du Québec

Hugo Vaillant
Rapport de laboratoire
LOG430 — Architecture logicielle
28 septembre 2025
École de technologie supérieure

Ce document constitue le rapport final du projet BrokerX. Il regroupe l'ensemble de la documentation d'architecture (arc42), ainsi que les guides pratiques nécessaires à l'exploitation et à la démonstration du système. Vous y trouverez :

- La documentation complète de l'architecture (arc42)
- Une explication des travaux CI/CD accomplis
- Un guide d'exploitation (runbook) pour l'administration et la maintenance
- Un guide de démonstration pour présenter les fonctionnalités clés

note : Le guide d'exploitation ainsi que le guide de démonstration se trouvent à la toute fin du document.

Début de la documentation arc42

1. Introduction & Objectifs

1.1 Objectifs métier

BrokerX est une plateforme monolithique de courtage en ligne permettant aux utilisateurs de s'inscrire, de vérifier leur identité, de s'authentifier avec MFA, de déposer des fonds et de placer des ordres sur des actifs financiers. L'objectif principal est d'offrir une expérience sécurisée, fluide et conforme aux exigences réglementaires du secteur financier.

1.2 Fonctionnalités essentielles

- Inscription et vérification d'identité (KYC)

- Authentification forte avec MFA
- Gestion du portefeuille virtuel (dépôt, solde)
- Placement d'ordres d'achat et de vente
- Gestion de l'idempotence et de la traçabilité des opérations
- Journalisation des actions et audit de sécurité

1.3 Objectifs de qualité pour l'architecture

Tableau 1. Objectifs de qualité

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 500 ms (P95) et traiter au moins 300 ordres/s	Expérience utilisateur et compétitivité du service
Disponibilité	Le service doit être disponible 90% du temps, même en cas de panne d'un composant	Continuité de service et fiabilité pour les utilisateurs
Résilience	En cas d'incident, le système doit pouvoir récupérer et restaurer les opérations sans perte de données	Robustesse et gestion proactive des erreurs
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	Pérennité et adaptation à la demande
Traçabilité & auditabilité	Toutes les opérations sont journalisées et traçables	Conformité et sécurité
Conformité	Respect des standards du secteur (tokens, MFA, KYC)	Obligations réglementaires

1.4 Parties prenantes

Tableau 2. Parties prenantes

Partie prenante	Rôle	Attente principale
Clients	Utilisateurs via interface web/mobile	Expérience fluide, sécurité des transactions, accès rapide aux cotations et exécution des ordres

Partie prenante	Rôle	Attente principale
Opérations Back-Office	Gestion des règlements, supervision	Outils de gestion efficaces, visibilité sur les opérations, fiabilité des processus
Conformité / Risque	Surveillance pré- et post-trade	Accès aux journaux d’audit, alertes en cas d’anomalie, conformité réglementaire garantie

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

1.5 Vue d’ensemble des exigences fonctionnelles

Tableau 3. 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
Placement d’un ordre	Permet à un utilisateur de placer un ordre d’achat ou de vente sur un actif	UC05.md

1.6 Priorisation MoSCoW des cas d’utilisation

Priorisation MoSCoW des cas d'utilisation

Tableau 4. Priorisation MoSCoW

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

Cas d'utilisation	Priorité MoSCoW	Justification
		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	Should	La sécurité des accès est importante pour la conformité et la confiance, mais une authentification forte peut être ajoutée après la mise en place des fonctionnalités principales. Une version initiale peut fonctionner avec des contrôles simplifiés.
UC-07 — Appariement interne & Exécution (matching)	Should	L'appariement automatique améliore la rapidité et la précision des exécutions : il est recommandé pour optimiser l'expérience utilisateur, mais peut être simplifié ou simulé dans une première version si nécessaire.
UC-04 — Abonnement aux données de marché	Could	Permet aux utilisateurs de prendre des décisions informées, mais la plateforme peut fonctionner sans cette fonctionnalité, en mode minimal ou pour des tests.
UC-06 — Modification / Annulation d’un ordre	Could	Offre de la flexibilité et réduit les erreurs, mais un MVP peut fonctionner sans cette capacité, en imposant plus de rigueur à l'utilisateur.
UC-08 — Confirmation d’exécution & Notifications	Won't Have	Utile pour la transparence et l'information client, mais sera exclu de la première version pour se concentrer sur les fonctionnalités essentielles et réduire la complexité technique.

> 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

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 envoit 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-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.

2. Contraintes d'architecture

2.1 Contraintes techniques

Tableau 5. Contraintes techniques

Contrainte	Explication
Monolithe Java/Spring Boot	L'application doit être développée en Java avec le framework Spring Boot, sous forme de monolithe.
Base de données PostgreSQL	Toutes les données persistantes doivent être stockées dans une base PostgreSQL.
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.
API REST	Les interfaces externes doivent être exposées sous forme d'API REST.
Journalisation et audit	Toutes les opérations critiques doivent être journalisées pour audit et traçabilité.

2.2 Contraintes réglementaires et de conformité

Tableau 6. Contraintes règlementaires

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

Tableau 7. 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 doivent être couvertes par des tests automatisés (unitaires, d’intégration et E2E).

2.4 Contraintes de gestion et maintenance

Tableau 8. Contraintes de gestion

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.

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

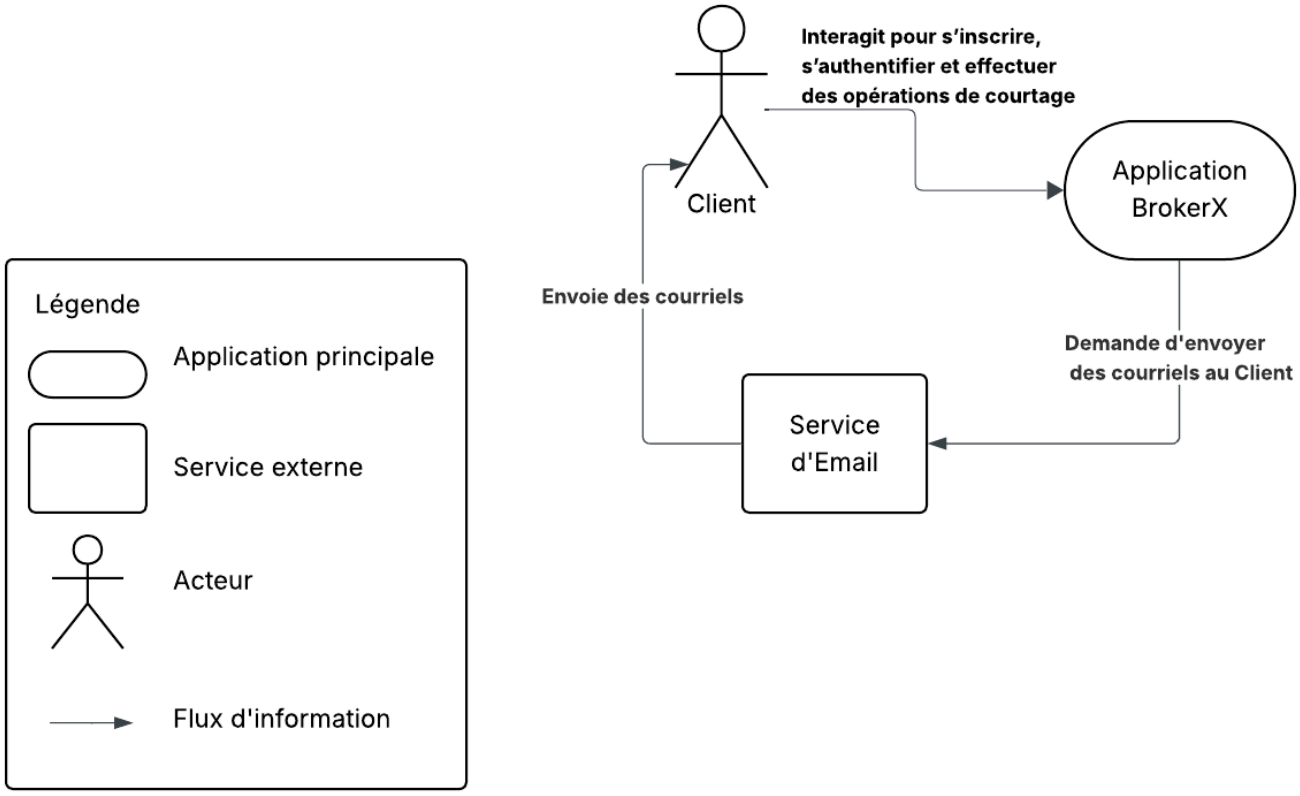
Le tableau ci-dessous présente les principaux acteurs et leurs interactions, en cohérence avec le diagramme de contexte DDD (voir section 3.2).

Tableau 9. Contexte métier

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

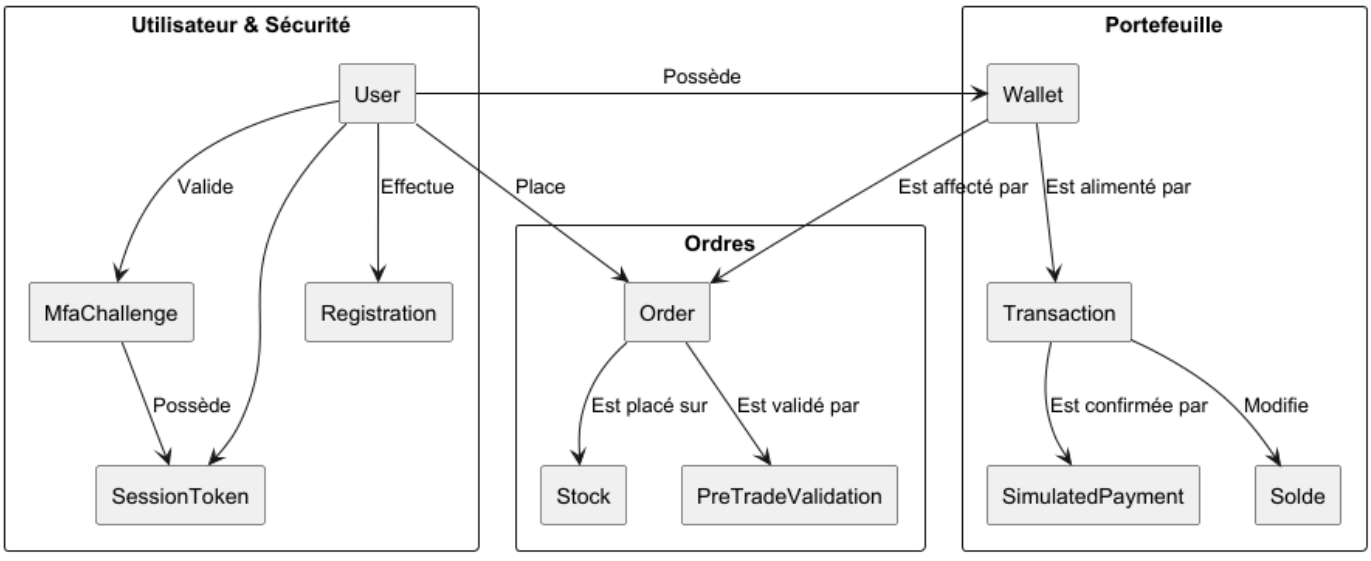
3.2 Diagramme de contexte DDD

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

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.

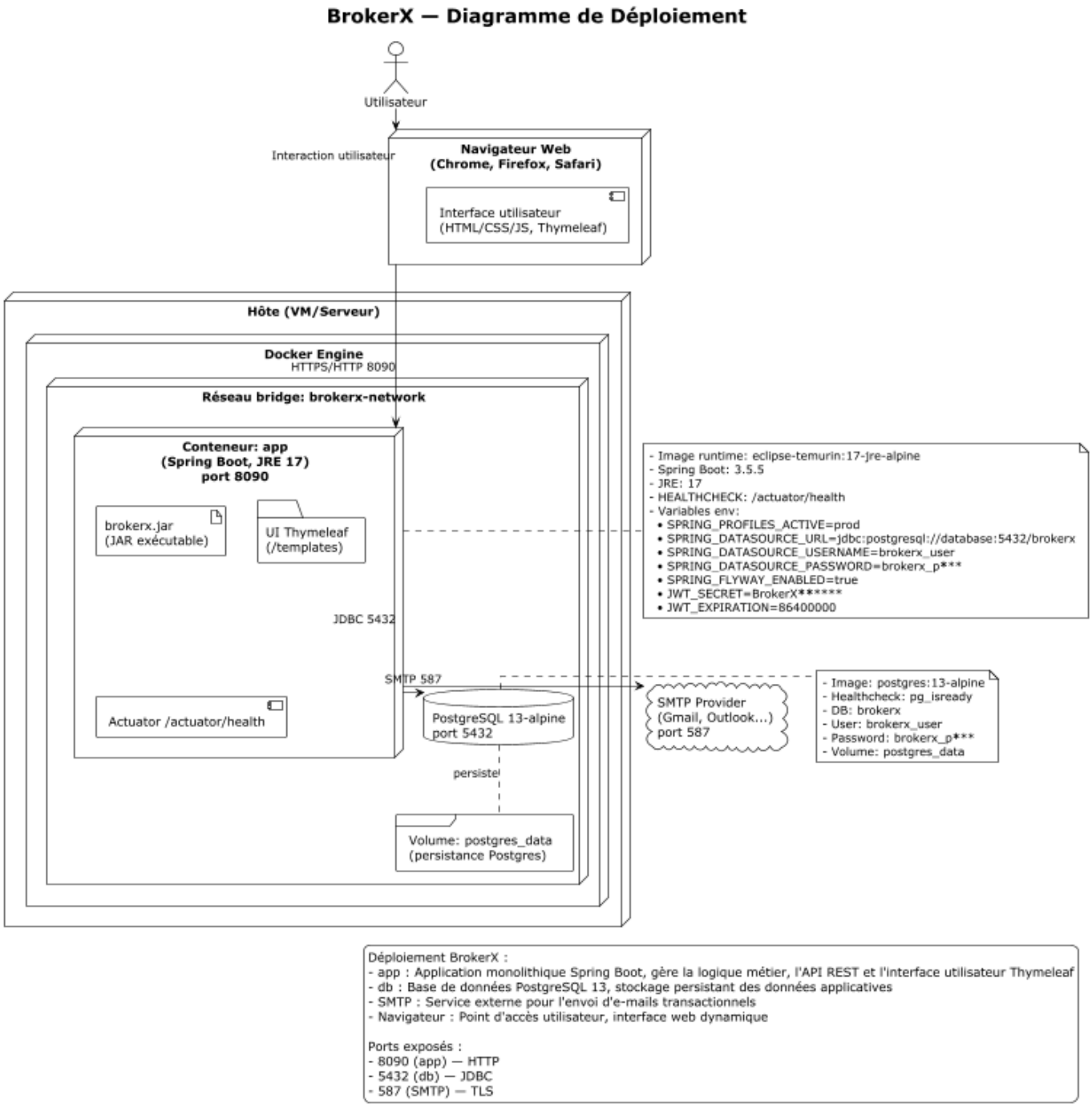


Tableau 10. Contexte technique

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

Système technique externe	Canal / Protocole	Format des données	Sécurité	Explication interface
SMTP Provider (GMail)	SMTP (TLS)	Texte, HTML	Chiffrement TLS, authentification SMTP	Envoi 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 sur des choix technologiques et organisationnels simples, robustes et adaptés aux besoins métier et réglementaires du courtage en ligne.

Le projet est développé en Java avec le framework Spring Boot, sous forme de monolithe. Ce choix permet une gestion centralisée des fonctionnalités, une maintenance facilitée et une intégration rapide des évolutions. La structure en couches (contrôleur, service, repository) favorise la séparation des responsabilités et la testabilité du code, ce qui simplifie la correction des bugs et l'ajout de nouvelles fonctionnalités.

La persistance des données est assurée par PostgreSQL, garantissant fiabilité, performance et conformité avec les standards du secteur. Ce SGBD est reconnu pour sa robustesse et sa capacité à gérer des transactions critiques, ce qui est essentiel pour un système financier.

Les échanges avec les clients se font via des API REST sécurisées (HTTPS/JSON), assurant une expérience utilisateur fluide et réactive.

La sécurité est une priorité : authentification forte (MFA) gestion des rôles et journalisation systématique des actions critiques. Ces mesures répondent aux exigences réglementaires (KYC, RGPD) et protègent les données sensibles contre les accès non autorisés et les fraudes.

L'envoi de code et de vérifications par e-mail est externalisé via un fournisseur SMTP (GMail), simplifiant la gestion des communications et garantissant la fiabilité du service. Cela évite de gérer un serveur mail interne et assure une meilleure délivrabilité des messages.

Le déploiement s'appuie sur Docker et Docker Compose, facilitant la portabilité, la reproductibilité et la gestion des environnements. Cette approche permet de déployer le système rapidement sur différents serveurs ou dans le cloud, tout en gardant une configuration cohérente.

Un pipeline CI/CD automatise les tests et les livraisons, assurant la qualité et la rapidité des mises en production. Cela réduit les risques d'erreur humaine et accélère le cycle de développement.

Les tests automatisés (unitaires, d'intégration et E2E) sont systématisés pour garantir la stabilité et la conformité du système. Ils permettent de détecter rapidement les régressions et d'assurer la fiabilité du code à chaque évolution.

La documentation est produite en continu pour accompagner chaque évolution et faciliter la prise en main par l'équipe. Elle permet de garder une trace des choix et des architectures, et d'accélérer l'intégration de nouveaux membres.

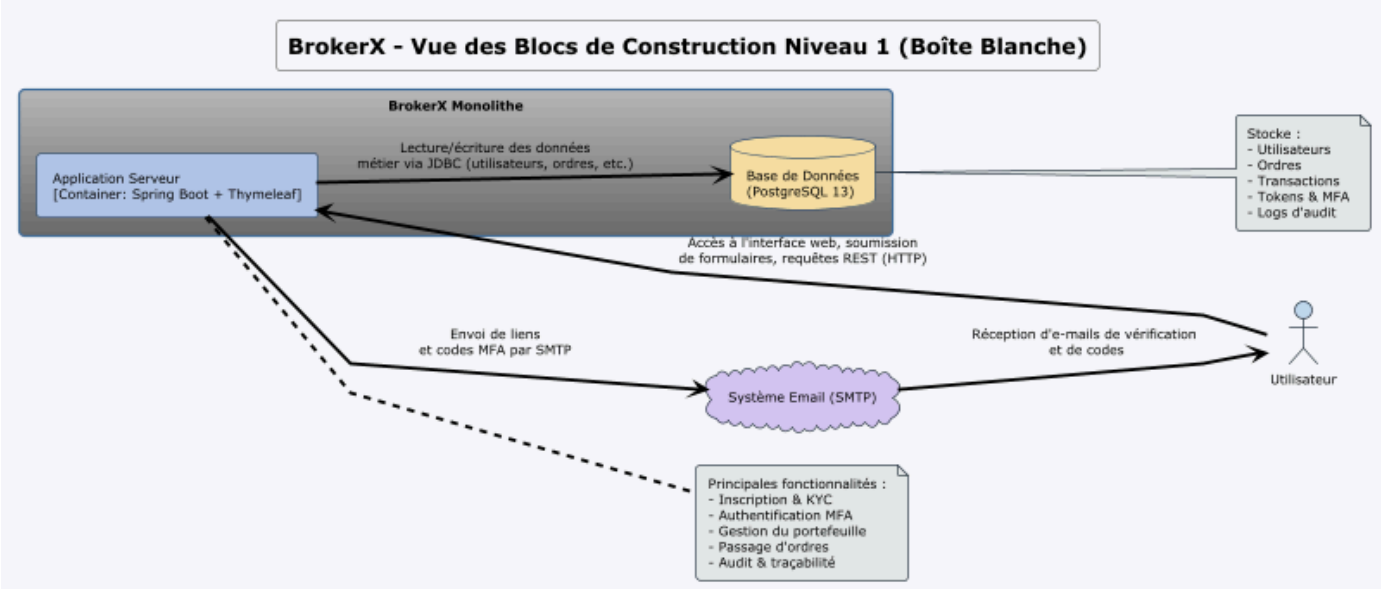
Le choix du monolithe, des technologies éprouvées et des processus automatisés permet de répondre efficacement aux objectifs de sécurité, performance, disponibilité et conformité, tout en gardant la solution évolutive pour de futurs besoins. Cette stratégie assure un socle solide pour le projet et prépare l'ouverture vers des intégrations ou des évolutions plus complexes.

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 :

Tableau 11. Aperçu niveau 1

Élément	Type / Technologie	Rôle / Description
Utilisateur	Acteur externe	Interagit avec BrokerX via l'interface web : navigation, formulaires, requêtes REST, réception d'e-mails
Application Serveur	Container Spring Boot + Thymeleaf	Fournit l'interface utilisateur web et l'API REST : inscription, KYC, MFA, gestion du portefeuille, passage d'ordres, audit & traçabilité
Base de Données	PostgreSQL 13	Stocke : utilisateurs, ordres, transactions, tokens MFA, logs d'audit. Accès via JDBC depuis l'application serveur

Élément	Type / Technologie	Rôle / Description
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 :

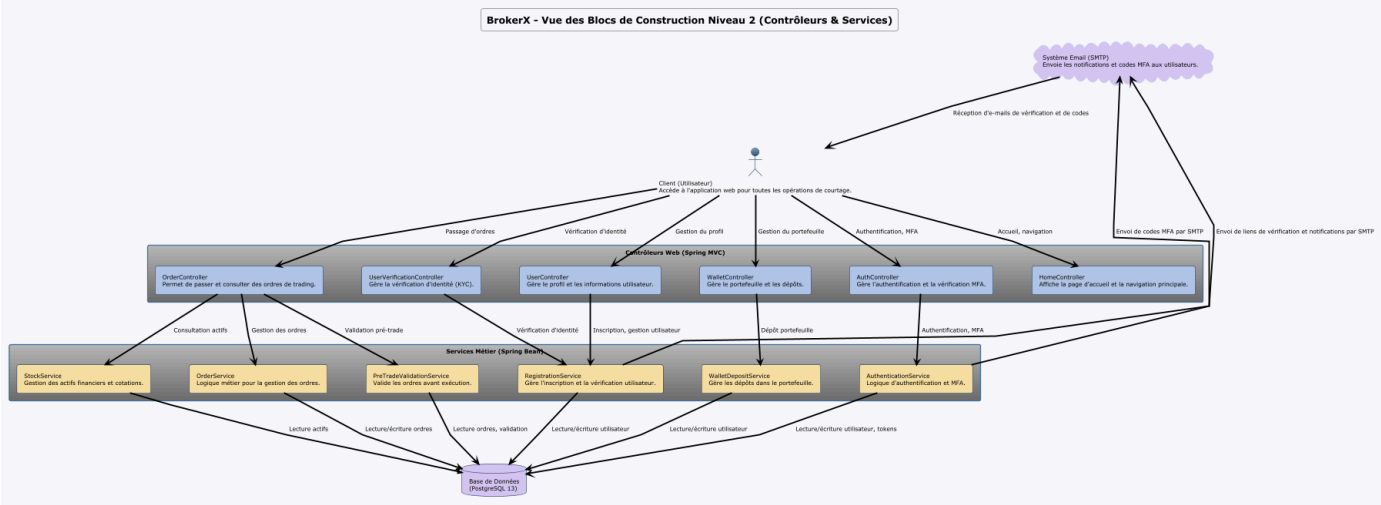
- L'utilisateur accède à l'application serveur pour toutes les opérations (web, formulaires, API REST)
- L'application serveur lit/écrit les données métier dans la base PostgreSQL via JDBC
- L'application serveur envoie des liens et des codes MFA au système email (SMTP)
- Le système email transmet les e-mails de vérification et de codes à l'utilisateur

Données stockées dans la base :

- Utilisateurs
- Ordres
- Transactions
- Tokens & MFA
- Logs d'audit

Cette vue permet de comprendre les principaux composants, leurs rôles et les flux d'information entre les acteurs, le système et les services externes.

5.3 Vue interne des composants (Niveau 2)



Le diagramme ci-dessus détaille les principaux composants internes du système :

Tableau 12. Aperçu niveau 2

Composant	Type / Rôle	Description
Client (Utilisateur)	Acteur externe	Accède à l'application web pour toutes les opérations de courtage
AuthController	Contrôleur web	Gère l'authentification et la vérification MFA

Composant	Type / Rôle	Description
HomeController	Contrôleur web	Affiche la page d'accueil et la navigation principale
OrderController	Contrôleur web	Permet de passer et consulter des ordres de trading
UserController	Contrôleur web	Gère le profil et les informations utilisateur
UserVerificationController	Contrôleur web	Gère la vérification d'identité (KYC)
WalletController	Contrôleur web	Gère le portefeuille et les dépôts
AuthenticationService	Service métier	Logique d'authentification et MFA
OrderService	Service métier	Logique métier pour la gestion des ordres
PreTradeValidationService	Service métier	Valide les ordres avant exécution
RegistrationService	Service métier	Gère l'inscription et la vérification utilisateur
StockService	Service métier	Gestion des actifs financiers et cotations
WalletDepositService	Service métier	Gère les dépôts dans le portefeuille
Base de Données (PostgreSQL)	Base de données	Stocke toutes les données métier
Système Email (SMTP)	Service externe	Envoie les notifications et codes MFA aux utilisateurs

Principales interactions :

- Le client interagit avec les contrôleurs web pour toutes les opérations (authentification, ordres, profil, portefeuille, etc.)
- Les contrôleurs délèguent la logique métier aux services Spring correspondants
- Les services accèdent à la base de données pour lire/écrire les données métier
- Les services d'inscription et d'authentification envoient des e-mails (liens de vérification, codes MFA) au système SMTP, qui transmet ces messages au client

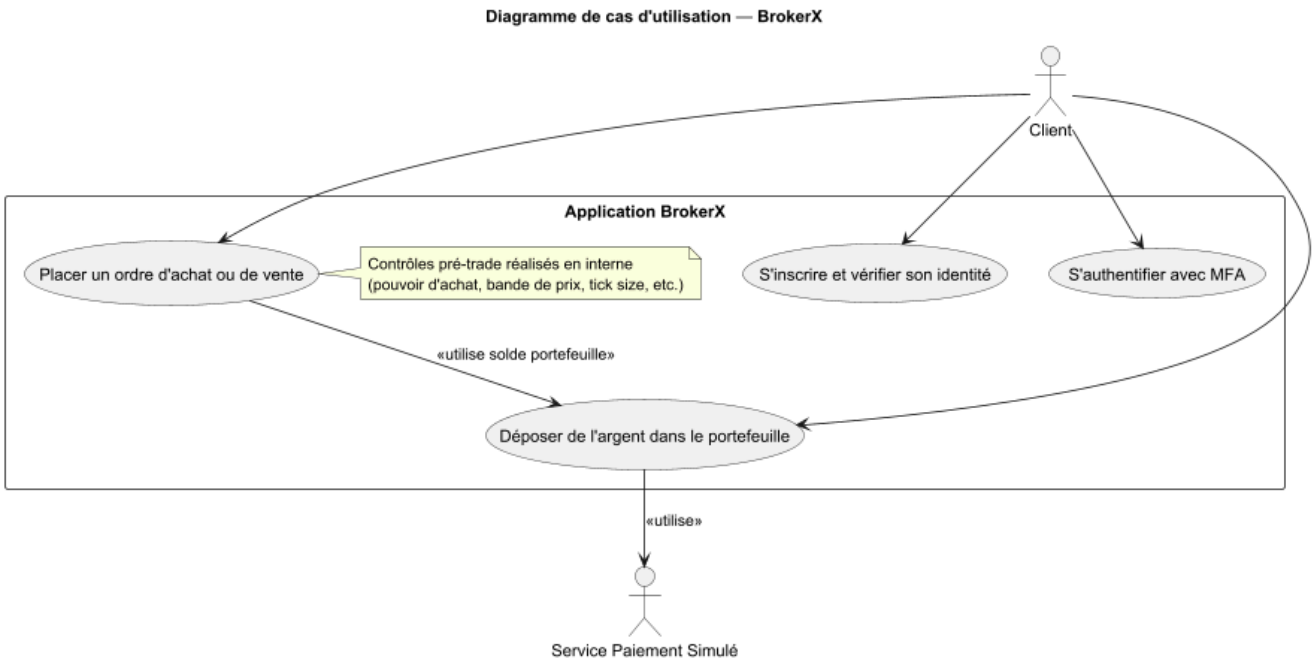
Cette vue permet de comprendre la répartition des responsabilités, les flux d'information et la collaboration entre les contrôleurs, les services, la base de données et les systèmes externes.

5.4 Organisation du code et conventions

Le code est structuré selon une approche hexagonale : les "adapters" gèrent les interactions externes (web, persistence), 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 Paiement Simulé (pour le dépôt)
- Cas d’utilisation : Inscription & vérification d’identité, Authentification & MFA, Dépôt dans le portefeuille, Placement d’un ordre

Relations

- Le client peut initier chacun des cas d’utilisation
- Le dépôt utilise le service de paiement simulé
- Le placement d’un ordre dépend du solde du portefeuille

Rationnel

Cette vue permet de relier les besoins métier aux fonctionnalités du système, de valider la couverture fonctionnelle et d’illustrer les interactions principales. Elle sert de point de départ pour la modélisation des autres vues et garantit que l’architecture répond bien aux attentes des utilisateurs et des parties prenantes. Elle facilite aussi la communication entre les équipes métier et technique, en offrant une vision synthétique des parcours utilisateurs et des dépendances fonctionnelles. Enfin, elle permet d’identifier rapidement les points d’intégration et les scénarios critiques à tester.

7. Vue de déploiement

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

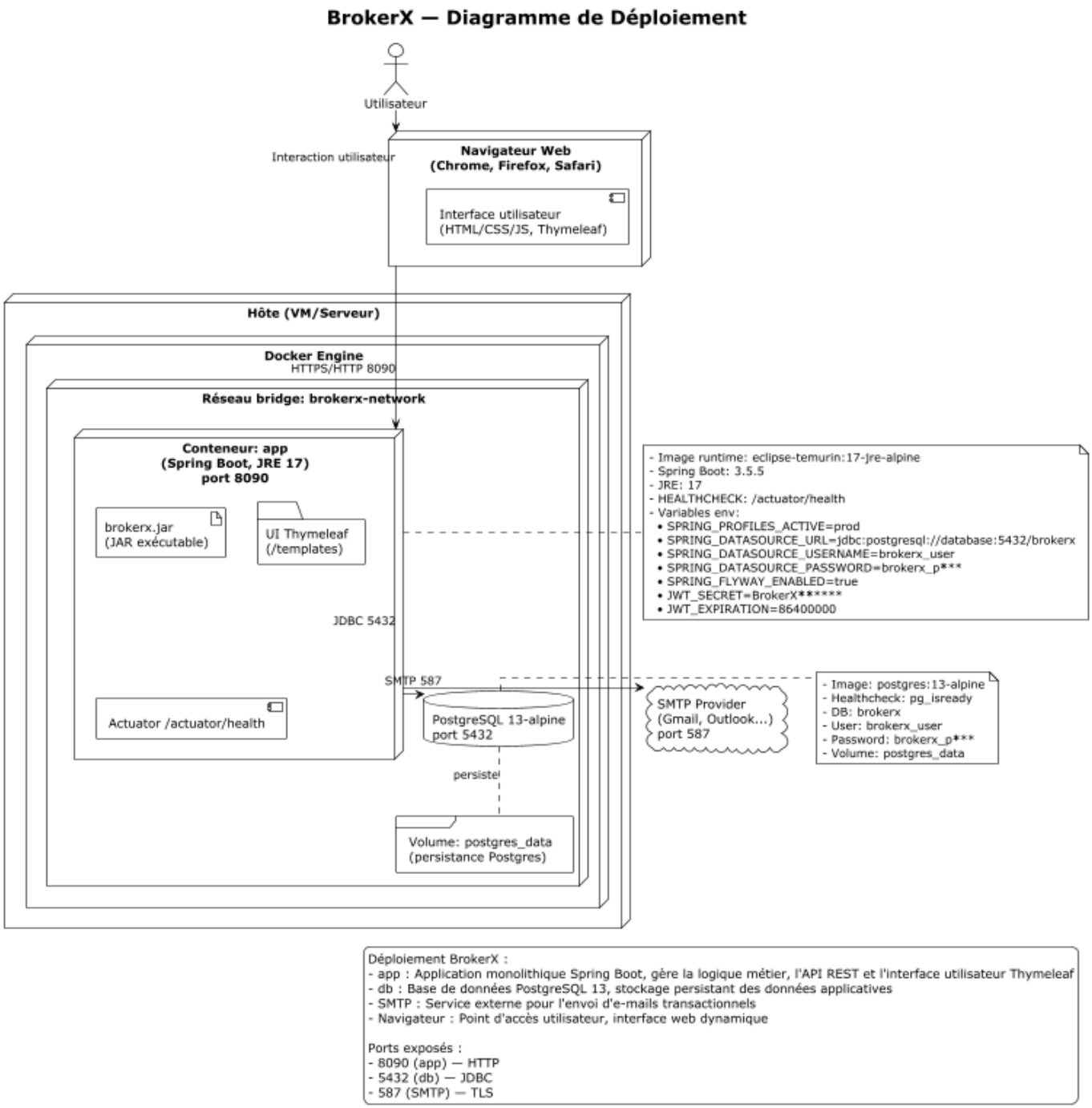


Tableau 13. Composants de déploiement

Nœud / Composant	Type / Technologie	Canal / Protocole	Sécurité / Persistance	Explication technique
Utilisateur	Acteur externe	-	-	Interagit via le navigateur web
Navigateur Web (Chrome, Firefox...)	Poste utilisateur	HTTPS/HTTP 8090	TLS	Interface utilisateur, rendu HTML/CSS/JS

Nœud / Composant	Type / Technologie	Canal / Protocole	Sécurité / Persistance	Explication technique
				(Thymeleaf)
Hôte (VM/Serveur)	VM/Serveur physique	-	-	Héberge le moteur Docker
Docker Engine	Plateforme conteneur	Interne Docker	Isolation conteneurs	Orchestration et isolation des composants applicatifs
Réseau bridge: brokerx-network	Réseau virtuel Docker	Interne Docker	Isolation réseau	Communication sécurisée entre conteneurs
Conteneur: app (Spring Boot, JRE 17)	Conteneur applicatif	HTTPS/HTTP 8090	TLS, MFA	Application principale, expose l'API REST et l'interface utilisateur
brokerx.jar (JAR exécutable)	Artefact Java	Interne au conteneur	-	Déployé dans le conteneur app
UI Thymeleaf (/templates)	Dossier de templates	Interne au conteneur	-	Rendu côté serveur
Actuator /actuator/health	Endpoint de monitoring	HTTP interne	-	Monitoring de l'état de l'application
PostgreSQL 13-alpine	Base de données	JDBC 5432	Authentification, persistance	Stocke toutes les données métier
Volume: postgres_data	Volume Docker	Interne Docker	Persistance	Persistance des données PostgreSQL
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 : comment les composants sont déployés sur l'infrastructure réelle (VM, Docker, réseau, base de données, services externes).

Éléments

- Conteneur applicatif Spring Boot
- Base de données PostgreSQL
- Service SMTP externe
- Volumes et réseaux Docker

Relations

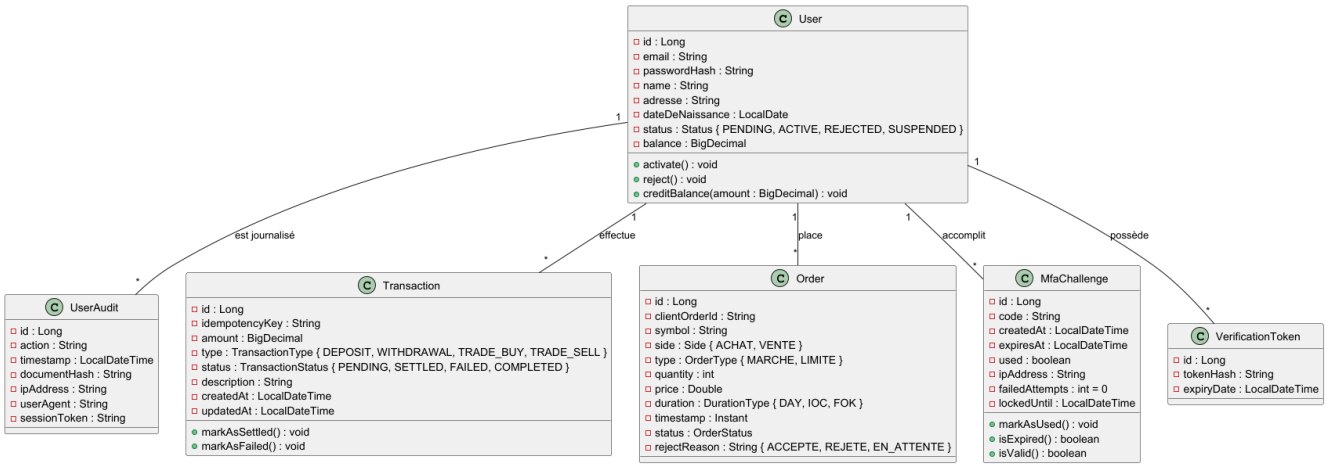
- L’application communique avec la base de données via JDBC
- Les emails sont envoyés via le service SMTP externe
- Les volumes assurent la persistance des données
- Le réseau Docker isole et sécurise les communications

Rationnel

Cette vue permet de comprendre la topologie du système, les points d’intégration, la sécurité et la résilience de l’application BrokerX en production. Elle est essentielle pour la gestion des déploiements, la scalabilité et la supervision. Elle aide à anticiper les besoins d’infrastructure, à optimiser la disponibilité et à garantir la conformité aux exigences de sécurité et de performance. Elle facilite aussi la gestion des incidents, la reprise après sinistre et l’évolution de l’architecture technique.

8. Vue Logique

Diagramme



Contexte

La vue logique présente la structure interne du système, centrée sur le modèle métier et les entités du domaine. Elle met en avant la modélisation des concepts clés, leur organisation et leurs relations.

Éléments

- Classes métier : User, Transaction, Order, MfaChallenge, VerificationToken, UserAudit
- Enums et value objects utilisés dans le domaine
- Relations entre les entités (composition, agrégation, associations)

Relations

- Les entités sont liées par des relations métier (ex : un User effectue des Transactions, place des Orders, etc.)
- Les enums et value objects enrichissent la sémantique métier

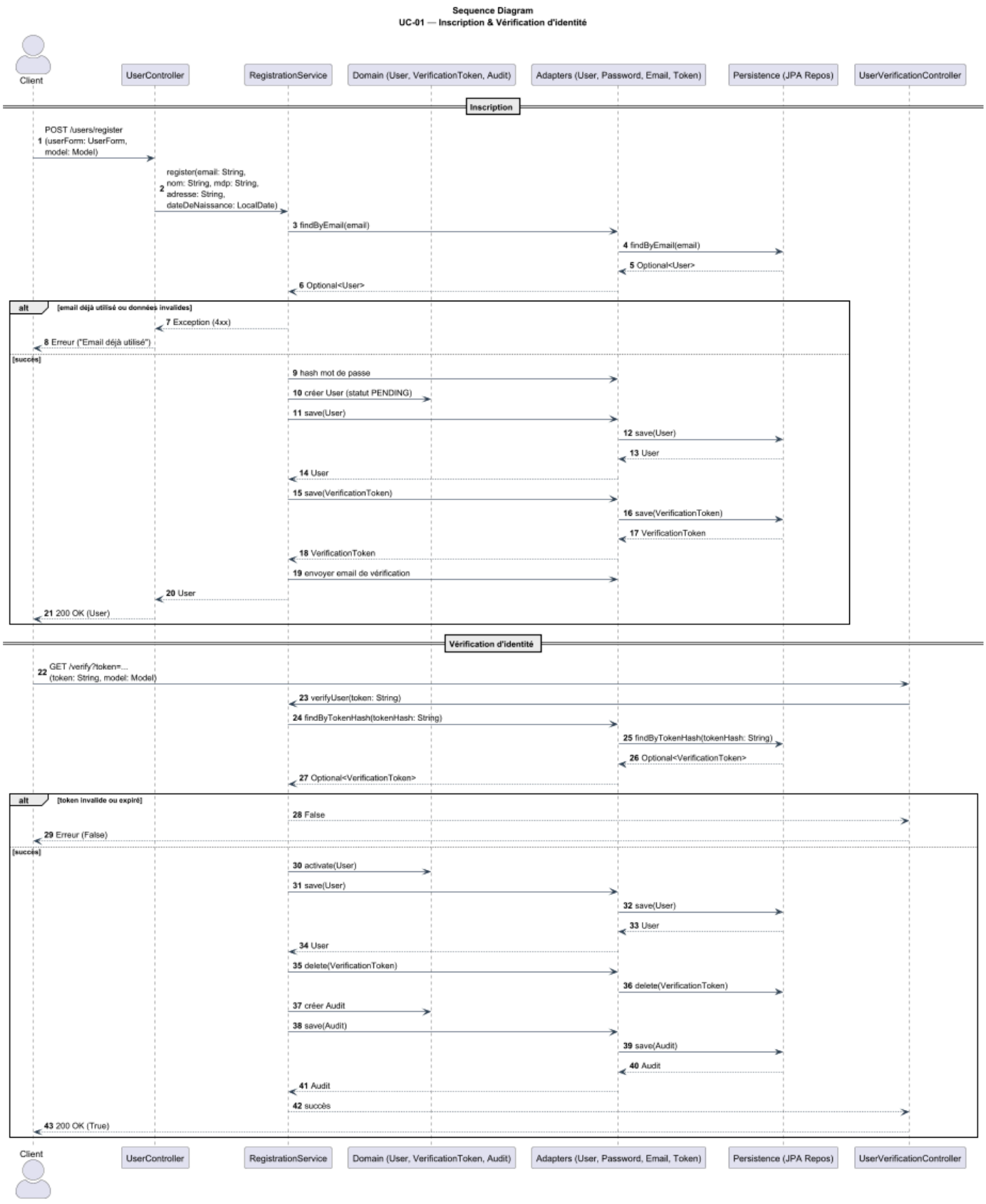
Rationnel

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. Elle sert de base à la validation des règles métier et à la conception des services applicatifs. En explicitant les dépendances et les relations entre les entités, elle aide à anticiper les impacts des évolutions fonctionnelles et à garantir la robustesse du modèle. Elle est aussi essentielle pour la documentation, la formation des nouveaux développeurs et la communication avec les parties prenantes métier.

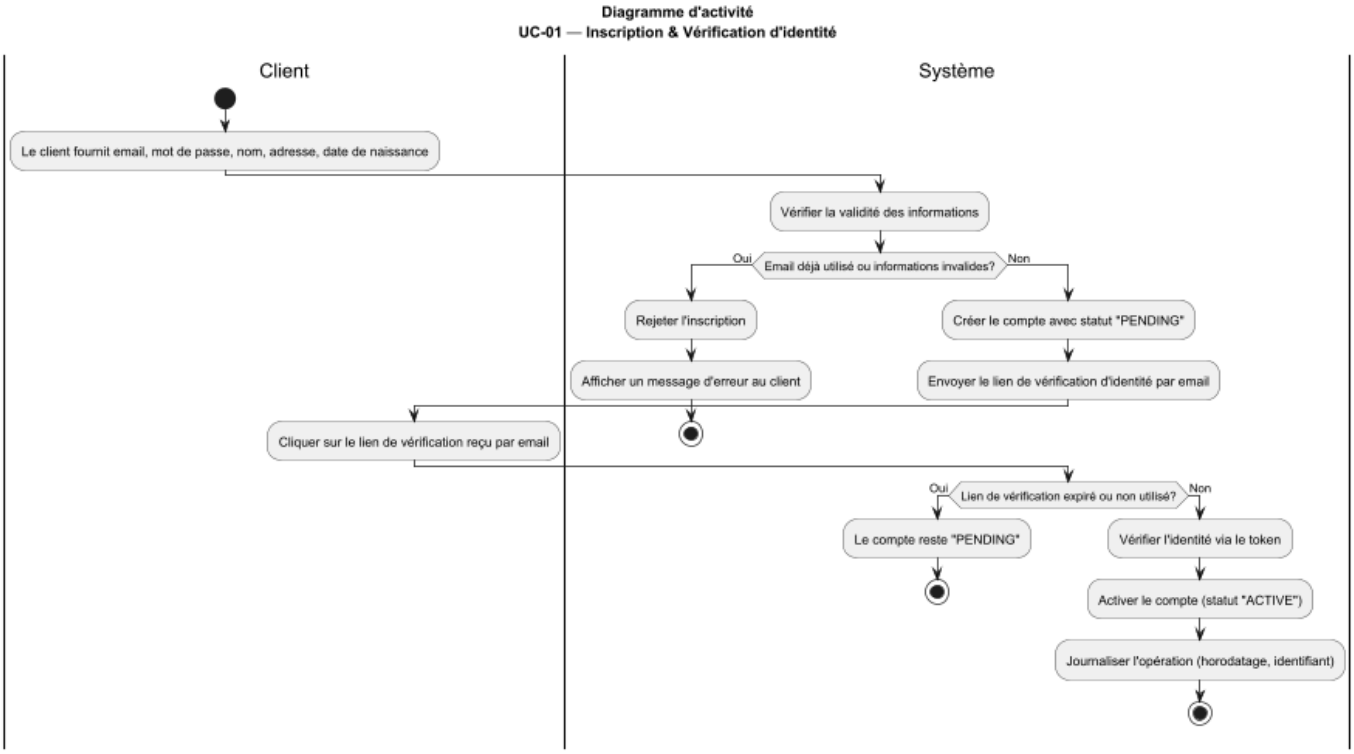
9. Vue Processus (C&C)

Diagrammes

UC01 — Séquence

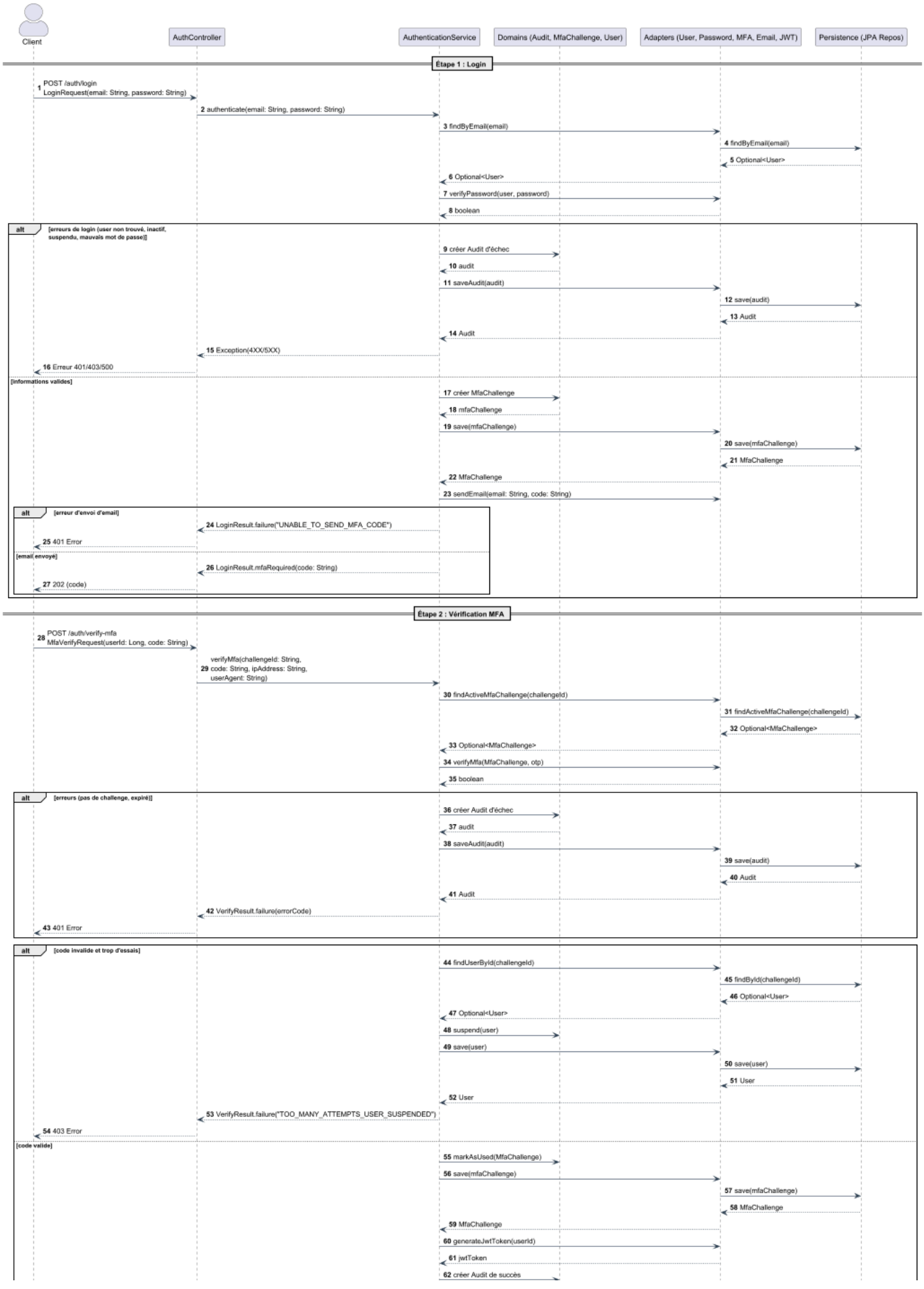


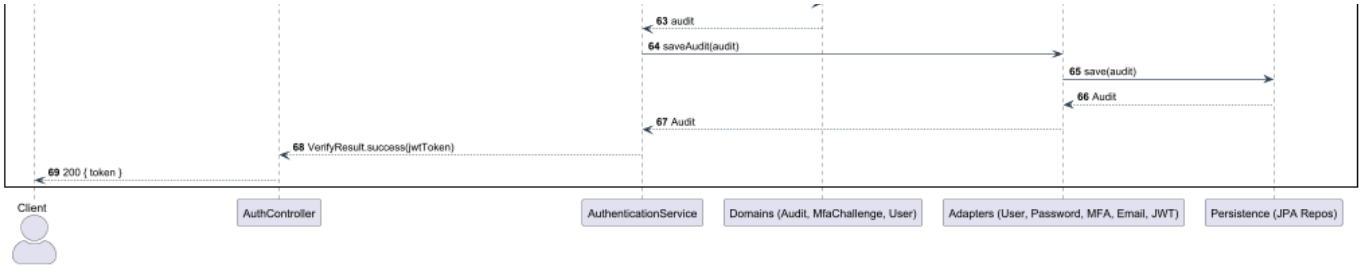
UC01 — Activité



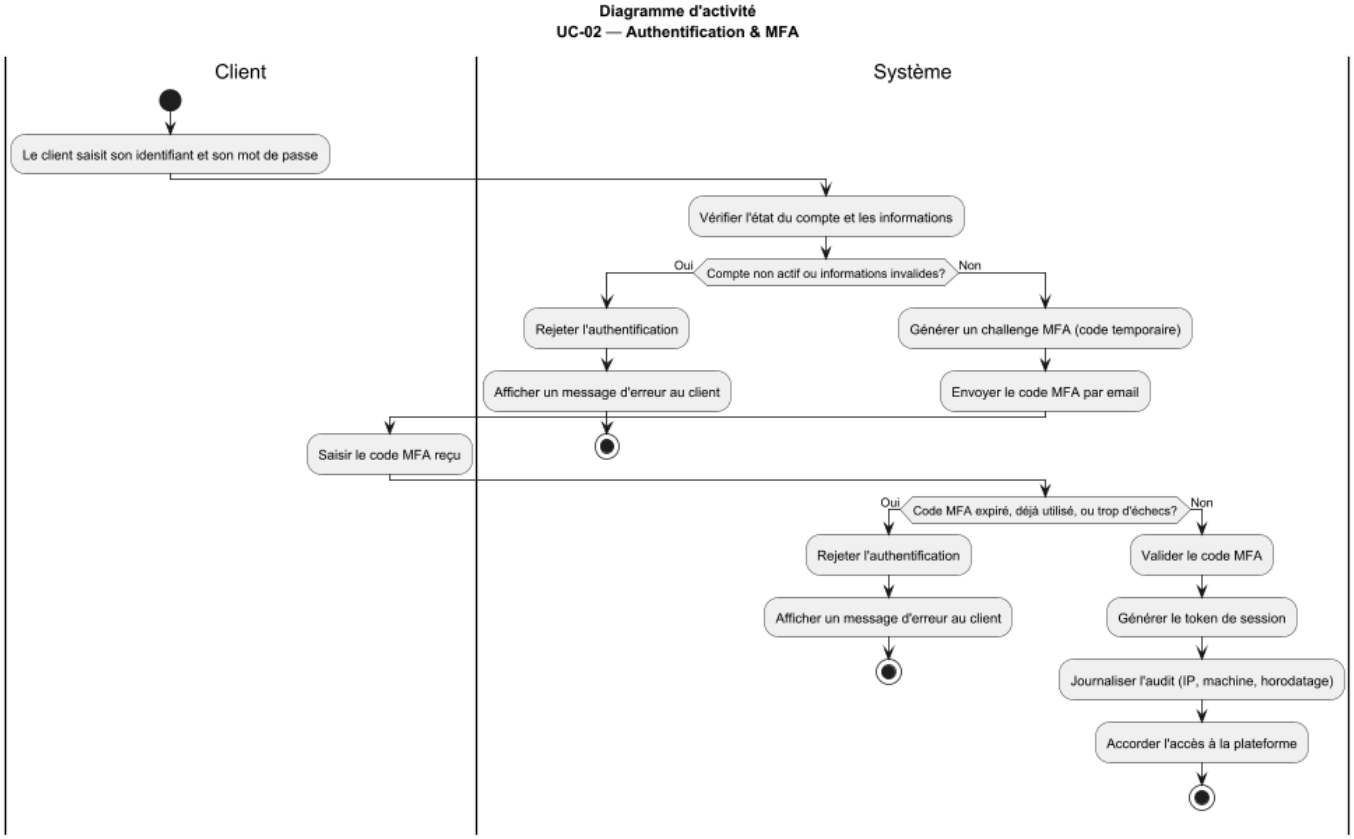
UC02 — Séquence

Sequence Diagram
UC-02 — Authentification & MFA

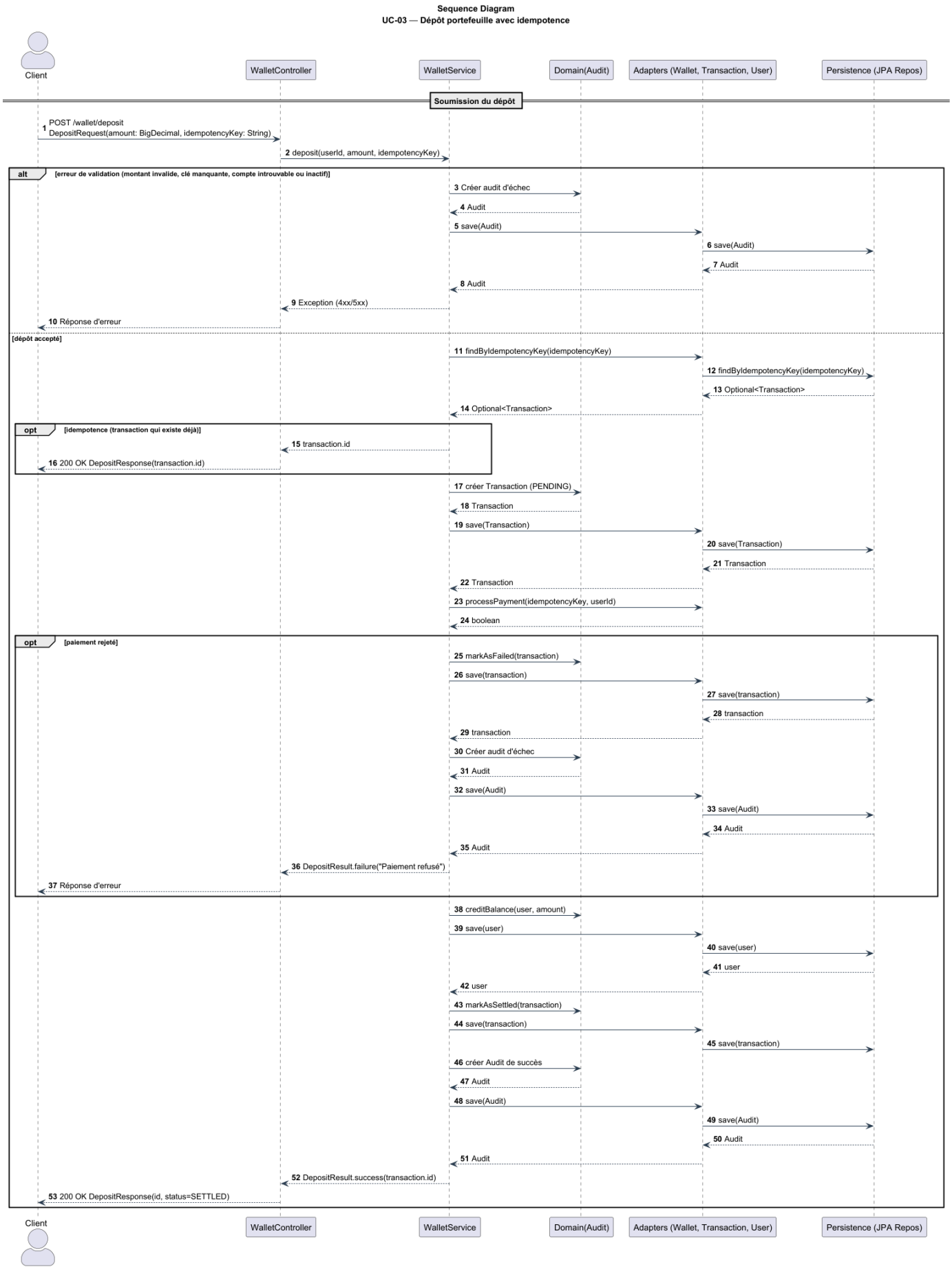




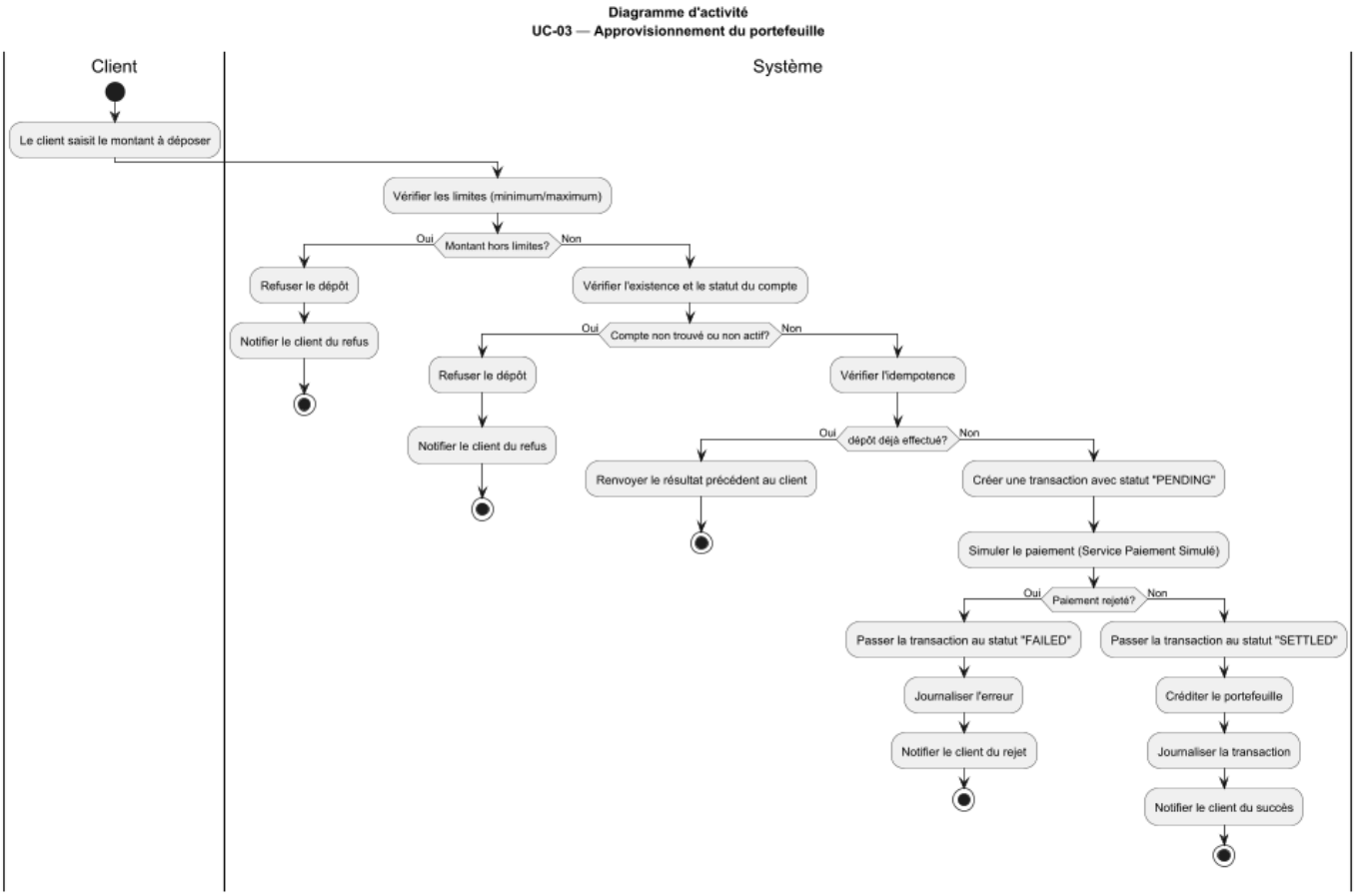
UC02 — Activité



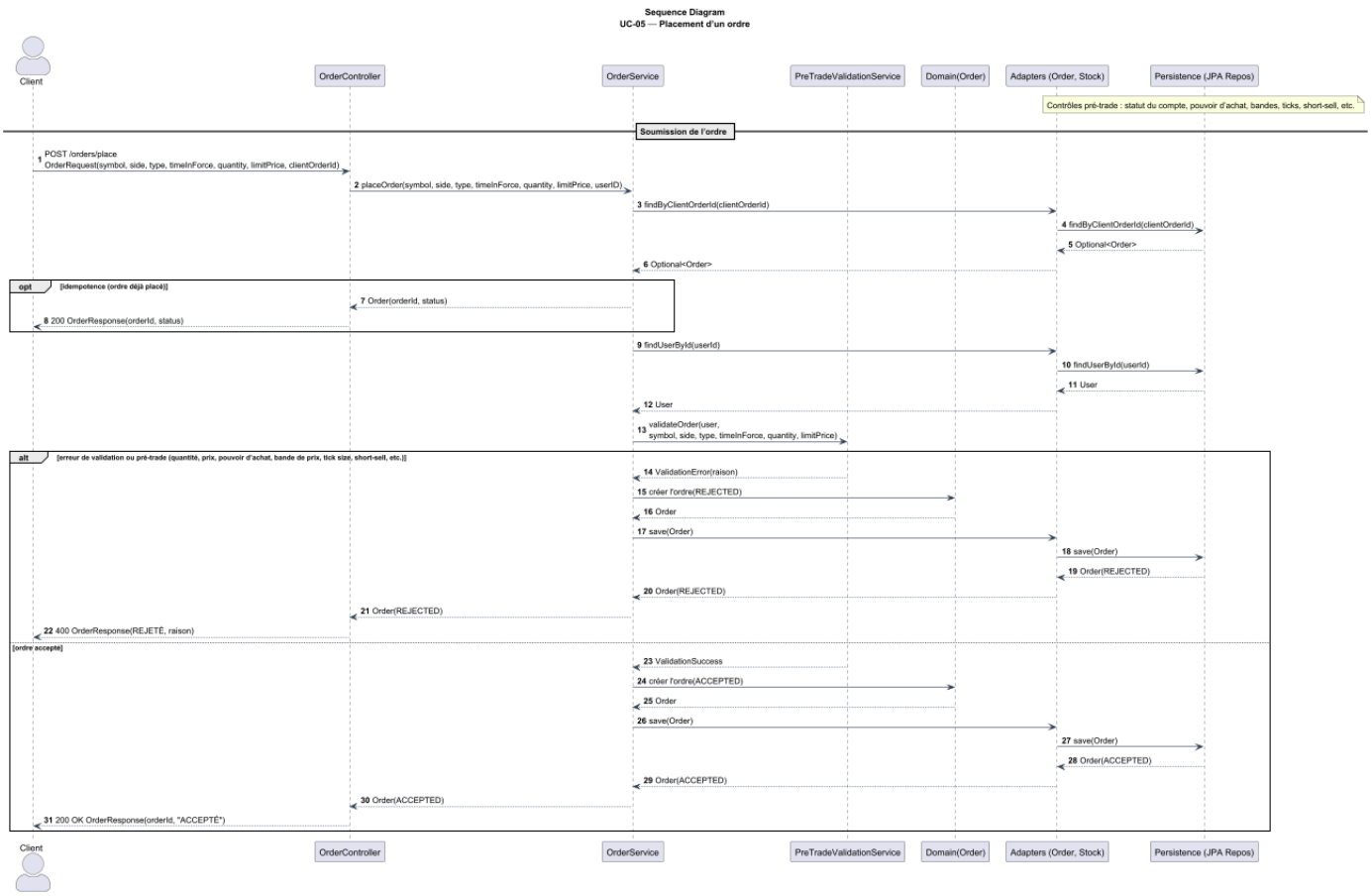
UC03 — Séquence



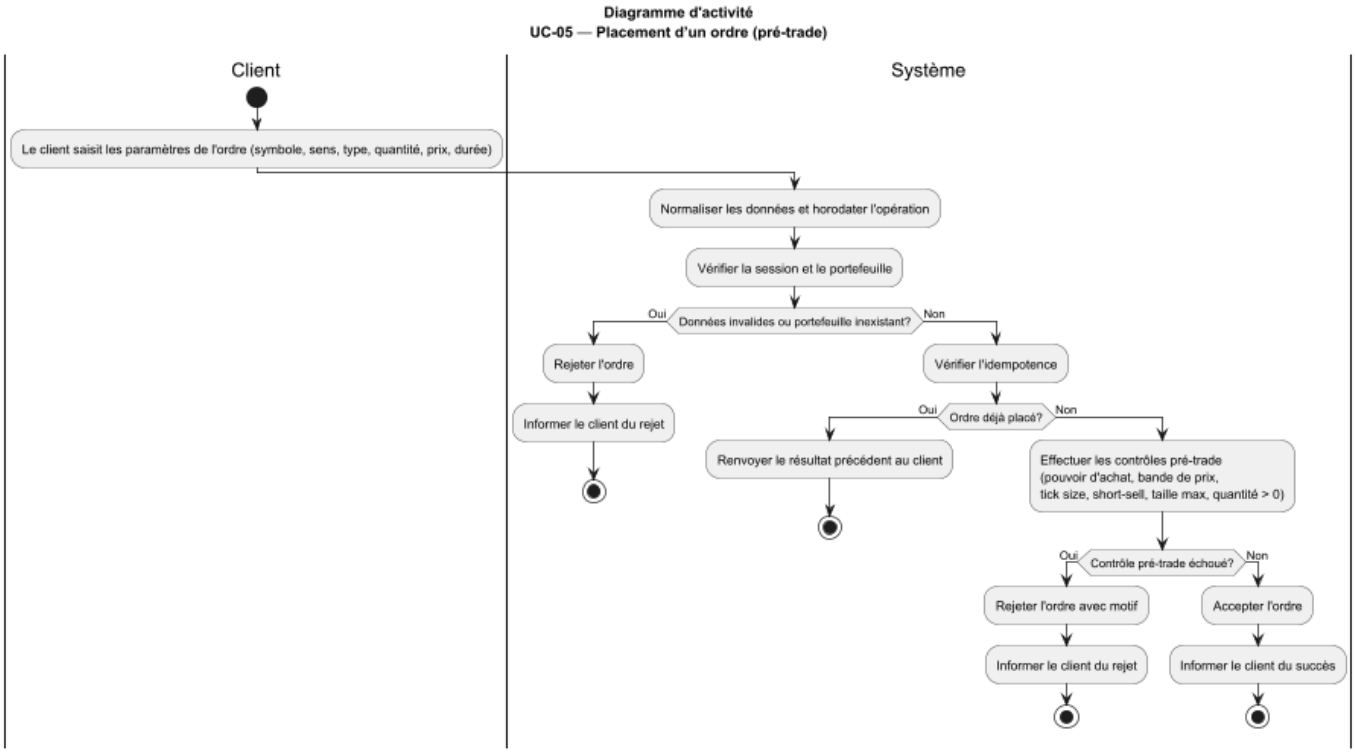
UC03 — Activité



UC05 — Séquence



UC05 — Activité



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, AuthService, WalletService, OrderService)
- Contrôleurs web (UserController, AuthController, WalletController, OrderController)
- Adapters (UserAdapter, TransactionAdapter, etc.)
- Persistance (JPA Repos)
- Acteurs externes (Client)

Relations

- Les contrôleurs reçoivent les requêtes des clients et délèguent aux services
- Les services orchestrent la logique métier et interagissent avec les adapters
- Les adapters font le lien avec la persistance et les systèmes externes
- Les diagrammes d'activité synthétisent les étapes clés et les alternatives

Rationnel

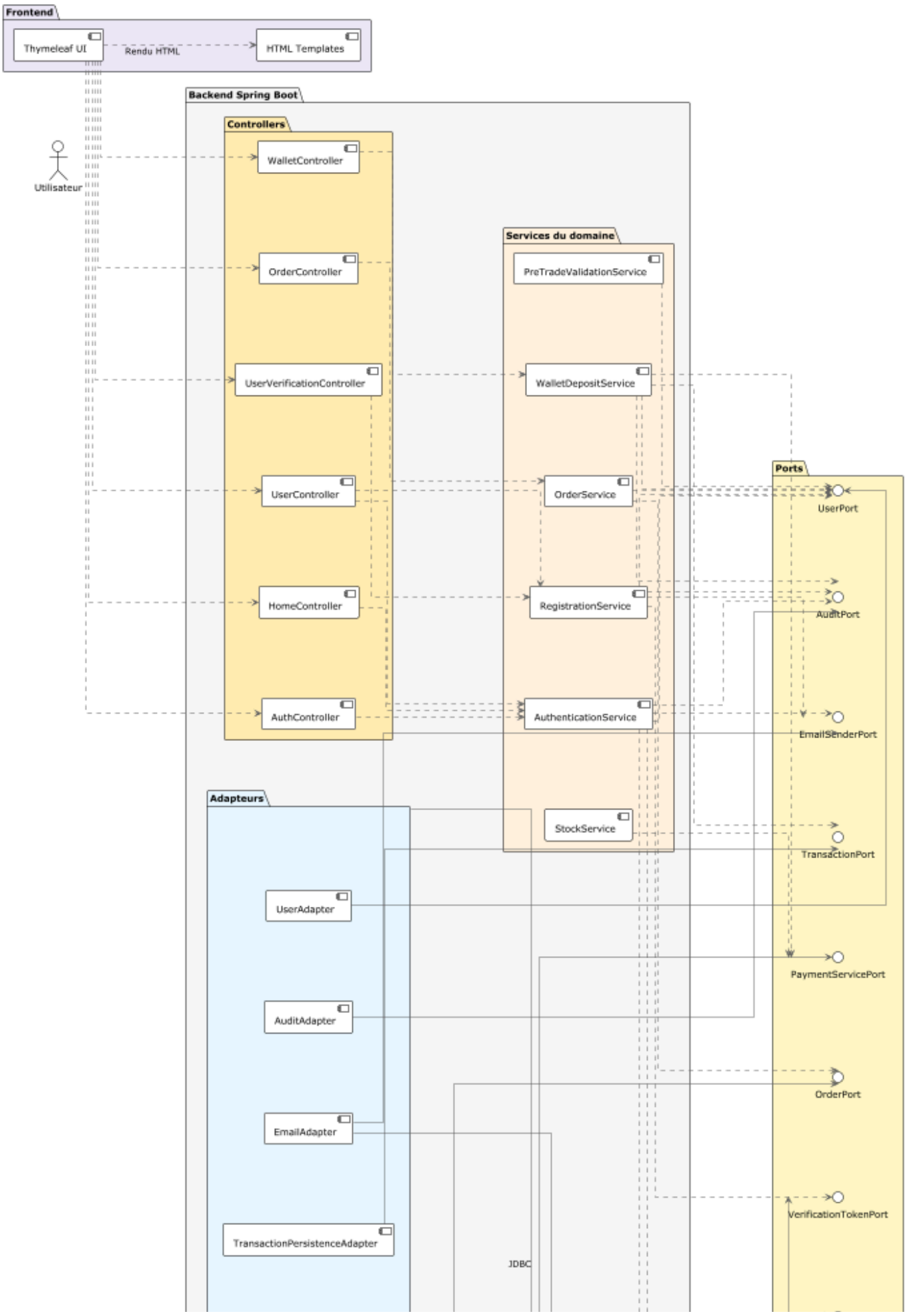
Cette vue permet de visualiser le flow des opérations, la gestion des exceptions, l'idempotence et la coordination entre les modules. Elle est essentielle pour valider la robustesse, la sécurité et la performance du système lors des opérations critiques. Elle aide à identifier les points de synchronisation, les risques de concurrence, et à optimiser la

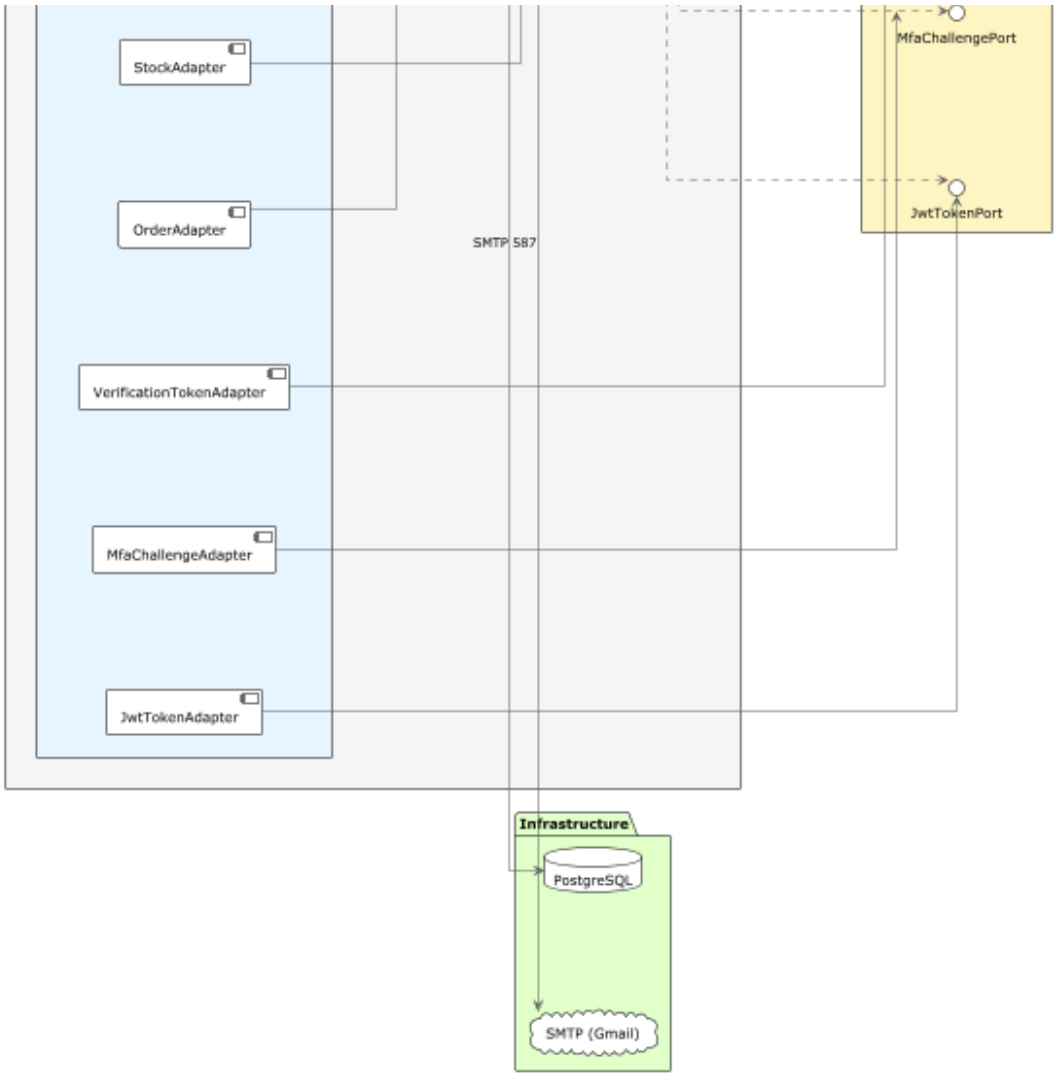
répartition des responsabilités. Elle est aussi précieuse pour l’analyse des scénarios d’erreur, la traçabilité des actions et la préparation des tests d’intégration et de non-régression.

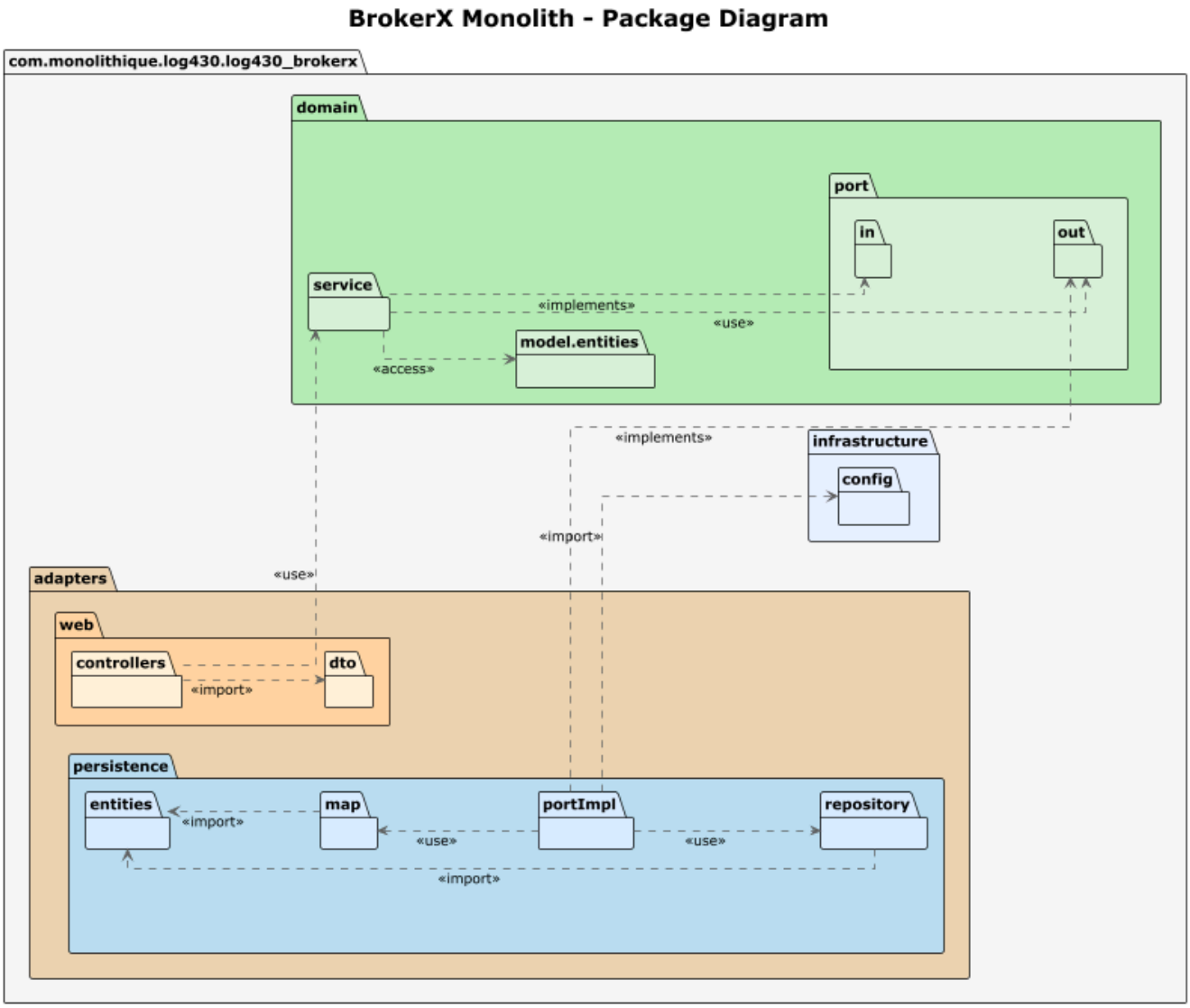
10. Vue Développement

Diagrammes

BrokerX - Component Diagram




StockAdapter
OrderAdapter
VerificationTokenAdapter
MfaChallengeAdapter
JwtTokenAdapter
SMTP 587
MfaChallengePort
JwtTokenPort
Infrastructure
PostgreSQL
SMTP (Gmail)



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.

Éléments

- Packages principaux : adapters, domain, infrastructure
- Composants techniques : contrôleurs, services, adapters, ports
- Structure des dossiers et conventions de nommage

Relations

- Les packages sont organisés selon l'architecture hexagonale
- Les dépendances entre modules sont explicites et maîtrisées
- Les conventions facilitent la maintenance et l'évolution

Rationnel

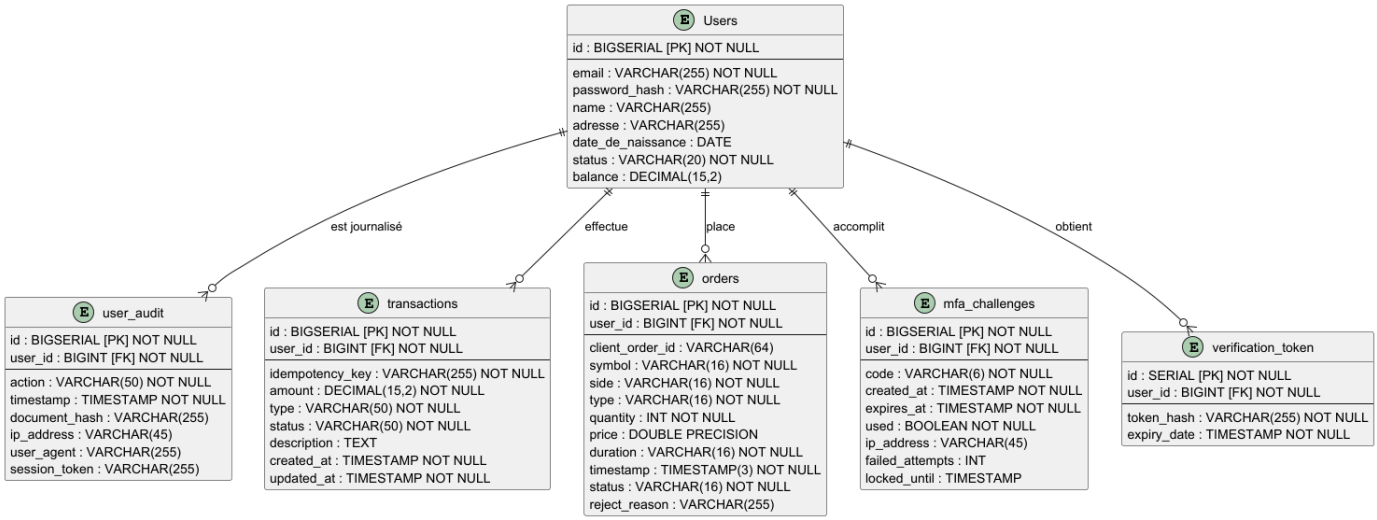
Cette vue facilite la compréhension du projet pour les développeurs, la maintenance et l'évolution du code. Elle favorise la modularité, la réutilisabilité et la robustesse de l'application. Elle permet d'anticiper les impacts des changements, d'améliorer la qualité du code et de réduire les risques 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 et de domaine

Avant d'aborder la structure des données, voici le diagramme ERD qui présente les principales entités persistées et leurs relations dans BrokerX. Ce schéma permet de visualiser la base du modèle de données et les liens entre les tables.

Diagramme ERD



Le tableau suivant détaille chaque table du modèle de persistance, avec une brève description de son rôle dans le système.

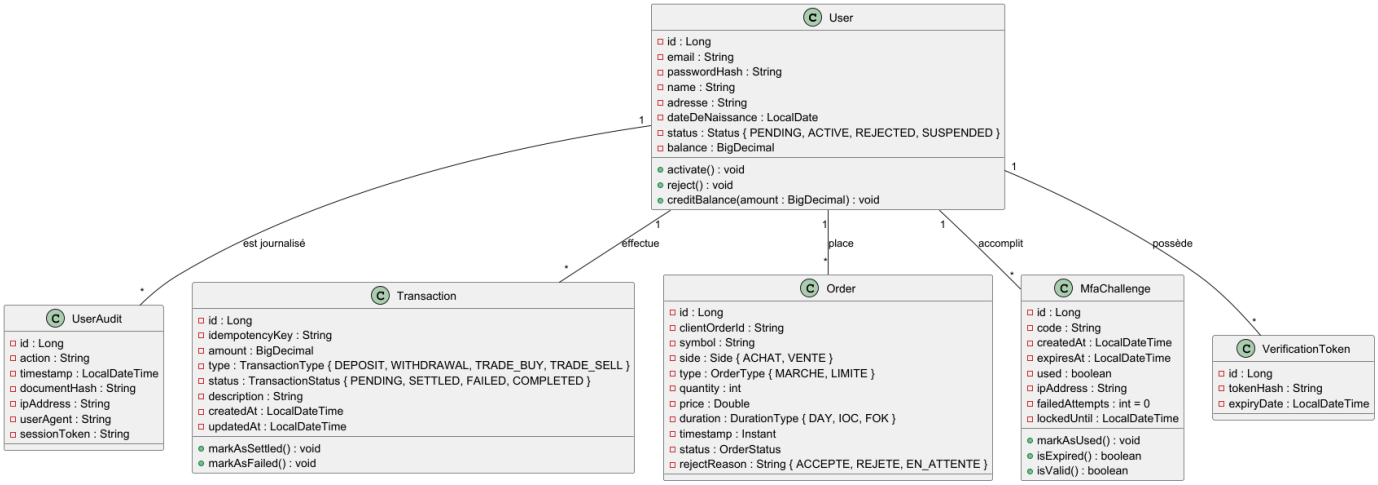
Tableau 14. Tables de persistance

Nom	Description
User	Stocke les utilisateurs : email, mot de passe, nom, adresse, date de naissance, statut, solde, etc.
UserAudit	Journalise les actions utilisateur : action, timestamp, document hash, IP, user agent, session token.
Transaction	Stocke les transactions : montant, type (dépôt, retrait, achat, vente), statut, description, dates.
Order	Stocke les ordres de trading : symbole, côté (achat/vente), type, quantité, prix, durée, statut, raison rejet.

Nom	Description
MfaChallenge	Stocke les défis MFA : code, date de création/expiration, état d'utilisation, IP, tentatives, verrouillage.
VerificationToken	Stocke les tokens de vérification : hash du token, date d'expiration.

Pour mieux comprendre la logique métier et les interactions entre les objets, le diagramme de classes ci-dessous illustre les entités principales et leurs relations dans le code.

Diagramme de classes



Le tableau suivant présente chaque entité du modèle de domaine, avec une explication de son rôle métier.

Tableau 15. Modèle de domaine

Nom d'entité	Description métier
User	Représente un utilisateur, gère l'activation, le rejet, le crédit du solde, et les statuts de compte.
UserAudit	Permet de tracer toutes les actions importantes réalisées par un utilisateur.
Transaction	Représente une opération financière : dépôt, retrait, achat ou vente, avec gestion d'idempotence et statut.
Order	Représente un ordre de trading, avec gestion du type, quantité, prix, durée, statut et raison de rejet.
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.

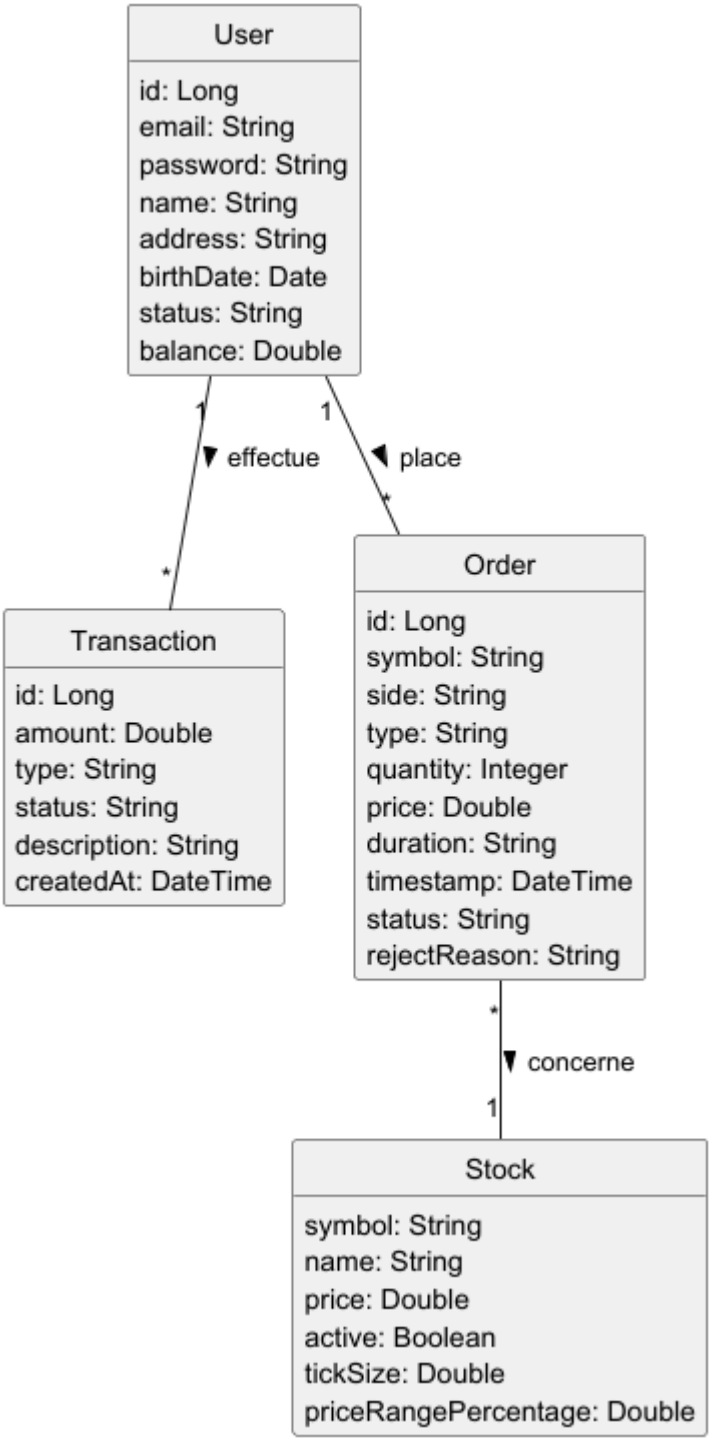
Puis, voici les principales méthodes métier de l'entité User, qui illustrent les opérations clés sur les comptes utilisateurs.

Tableau 16. Méthodes métier importantes sur User

Méthode	Description
activate()	Active le compte utilisateur.
reject()	Rejette le compte utilisateur.
creditBalance(amount)	Ajoute un montant au solde de l’utilisateur.

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 utilise une base de données PostgreSQL pour stocker toutes les données métier : utilisateurs, ordres, transactions, défis MFA, tokens de vérification et journaux d'audit. La base est persistée sur disque, et la localisation du fichier de données est gérée par la configuration Docker : le volume nommé `postgres_data` assure la persistance des données, même lors du redémarrage ou de la mise à jour des conteneurs.

Toutes les opérations sur la base de données passent par JPA, avec Hibernate comme fournisseur. Les entités persistées sont décrites dans la section Domain Models. L'accès aux entités se fait exclusivement via les repositories Spring Data JPA, garantissant une abstraction et une sécurité des accès.

Aucune donnée métier n'est stockée en mémoire ou dans des fichiers plats : toutes les informations critiques (identité, ordres, transactions, MFA, audit) sont centralisées dans PostgreSQL pour garantir la fiabilité, la traçabilité et la conformité réglementaire.

Les fichiers statiques (templates Thymeleaf, ressources front-end) sont embarqués dans le JAR de l'application et servis par Spring Boot. Aucun fichier utilisateur (image, document, etc.) n'est stocké en dehors de la base : toutes les pièces justificatives ou documents d'audit sont référencés par leur hash dans la base, assurant l'intégrité et la non-répudiation.

La configuration de la connexion à la base de données (URL, utilisateur, mot de passe) est externalisée dans les fichiers de propriétés et les variables d'environnement Docker, permettant une gestion flexible des environnements de développement, test et production.

En résumé, la persistance des données dans BrokerX repose sur :

- PostgreSQL pour toutes les données métier et d'audit
- Docker volumes pour la durabilité des données
- Spring Data JPA pour l'accès aux entités
- Aucun stockage de fichiers binaires ou documents en dehors de la base

Cette approche garantit la robustesse, la conformité et la portabilité du système, tout en facilitant la maintenance et la migration des données.

11.2.1 Choix ORM ou DAO

Le projet BrokerX utilise l'ORM JPA/Hibernate pour la gestion de la persistance, car il offre une abstraction puissante entre le modèle objet Java et la base de données relationnelle. L'ORM permet de définir les entités métier avec l'annotation `@Entity`, de gérer les relations (OneToMany, ManyToOne, etc.), et de bénéficier de la génération automatique des requêtes SQL. Les accès aux données sont réalisés via des interfaces annotées `@Repository`, qui héritent des interfaces Spring Data (JpaRepository, CrudRepository). Cette approche réduit le code boilerplate, facilite la maintenance et la testabilité, et permet de profiter de la gestion transactionnelle native de Spring. Contrairement à une approche DAO classique, qui nécessite d'écrire manuellement chaque requête et chaque mapping, l'ORM centralise la logique de persistance et garantit la cohérence entre le code et la base. Ce choix est particulièrement pertinent pour BrokerX, qui doit gérer des entités complexes, des relations multiples et des évolutions fréquentes du modèle.

11.2.2 Transactions

La gestion des transactions est assurée par l'annotation `@Transactional` sur les services et les repositories critiques (ex : WalletDepositService, OrderRepository, TransactionRepository, MfaChallengeAdapter). Spring gère automatiquement le début, la validation et le rollback des transactions, ce qui garantit l'atomicité des opérations et la cohérence des données. Les transactions couvrent les opérations sensibles comme les dépôts, les placements d'ordres, la gestion des MFA et la création de tokens. En cas d'erreur ou d'exception, toutes les modifications sont annulées pour éviter les incohérences.

11.2.3 Contraintes d'intégrité

Les contraintes d'intégrité sont définies dans les scripts de migration SQL (Flyway) et dans les entités JPA. On retrouve des clés primaires, des clés étrangères, des contraintes d'unicité (ex : client_order_id), des contraintes NOT NULL, et des index pour optimiser les accès. Par exemple, la table `orders` possède une contrainte d'unicité sur `client_order_id` pour éviter les doublons, et toutes les relations sont sécurisées par des clés étrangères avec gestion des cascades. Les entités JPA reflètent ces contraintes via les annotations (@Id, @Column(nullable = false), @UniqueConstraint, @ManyToOne, etc.), ce qui assure une double validation côté code et côté base de données.

11.2.4 Migrations reproductibles

Les migrations de schéma sont gérées par Flyway, avec des scripts SQL versionnés dans le dossier `db/migration`. Chaque modification de la structure de la base (création de table, ajout de colonne, contrainte, index) est tracée et appliquée de façon déterministe sur tous les environnements. Flyway garantit que chaque migration est idempotente, traçable et réversible. Cela permet de synchroniser la base entre les développeurs, les environnements de test et de production, et de revenir à un état antérieur en cas de problème. Les scripts sont testés et validés à chaque livraison, et la stratégie de versionnement évite les conflits et les pertes de données.

11.2.5 Données seed

Des données de seed sont insérées via le script `V12__Insert_seed_data.sql`. Ce script permet de peupler la base avec des utilisateurs, des ordres, des transactions et des MFA pour les démonstrations et les tests. Les données seed facilitent la validation fonctionnelle, la démo du produit et la reproductibilité des scénarios métier. Elles sont conçues pour couvrir les principaux cas d'utilisation et garantir que le système démarre toujours dans un état cohérent et exploitable.

11.3 Interface Utilisateur

L'interface utilisateur de BrokerX est une application web classique, rendue côté serveur avec Thymeleaf et servie par Spring Boot. Elle propose une navigation fluide et sécurisée pour toutes les opérations de courtage : inscription, authentification MFA, gestion du portefeuille, passage d'ordres et consultation des historiques.

L'UI utilise HTML, CSS et JavaScript, avec un thème sobre et responsive adapté aux besoins financiers. Toutes les interactions passent par des formulaires web et des appels REST sécurisés.

Aucune interface mobile native ou SPA n'est fournie par défaut : l'accès se fait exclusivement via le navigateur web.

11.4 Optimisation JavaScript et CSS

L'optimisation des ressources JavaScript et CSS dans BrokerX est volontairement simple : aucun framework externe, gestionnaire de dépendances ou outil de minification n'est utilisé. Tout le code JavaScript et CSS est écrit en inline directement dans les templates Thymeleaf, ce qui évite toute dépendance supplémentaire et simplifie le déploiement.

Aucune bibliothèque tierce (npm, bower, webjars, wro4j, etc.) n'est requise : l'ensemble des scripts et styles nécessaires à l'interface sont embarqués dans le JAR et servis par Spring Boot. Cette approche garantit la portabilité et la maintenance, tout en limitant la surface d'attaque et les risques liés aux mises à jour de dépendances externes.

En résumé : tout le CSS et le JavaScript de BrokerX est inline, intégré dans les templates, sans optimisation ou minification avancée.

11.5 Traitement des transactions

BrokerX s’appuie sur Spring Boot pour la gestion des transactions locales au sein du JPA EntityManager. Toutes les opérations critiques (ordres, dépôts, retraits, MFA) sont traitées de façon transactionnelle pour garantir la cohérence des données. BrokerX ne supporte pas les transactions distribuées.

11.6 Gestion de session

BrokerX expose uniquement une API publique stateless : aucune gestion de session côté serveur. L’authentification et l’autorisation sont gérées par des tokens JWT et MFA, sans stockage de session.

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. 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 Communications et intégration

BrokerX communique principalement via des API REST sécurisées (HTTPS) et SMTP pour l’envoi d’e-mails. Aucune file de messages ou broker interne n’est utilisé : toutes les intégrations externes (KYC, notifications, audit) passent par des appels HTTP ou SMTP. Les messages ne sont pas persistés en dehors de la base de données métier.

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

Les erreurs de validation (données invalides, contraintes métier) sont mappées sur des codes HTTP appropriés (400, 403, 422) et retournées par les contrôleurs Spring. Les erreurs techniques (base de données, réseau, SMTP) sont loguées et peuvent entraîner l’échec de la requête ou une réponse 500.

Les contrôleurs REST gèrent la conversion des exceptions en réponses HTTP standardisées. Les erreurs critiques sont journalisées pour analyse et audit.

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. Chaque entrée contient l'action, le timestamp, l'IP, le user agent et le token de session. Les transactions et les placements d'ordre sont également journalisés dans les tables Transaction et Order.

La journalisation technique (logs applicatifs) est assurée via SLF4J et la configuration par défaut de Spring Boot : les logs sont envoyés en stdout et peuvent être collectés par l'infrastructure d'hébergement.

La traçabilité métier est garantie par la persistance des logs d'audit en base, permettant l'analyse, la conformité et la détection d'anomalies. Les noms des loggers correspondent aux packages des classes pour faciliter l'identification des modules dans les logs.

11.13 Configurabilité

BrokerX utilise Spring Boot pour la gestion de la configuration. Les propriétés principales sont définies dans `src/main/resources/application.properties` pour le développement, et surchargées par `application-prod.properties` en production. Il est possible d'ajouter ou de modifier des propriétés via des variables d'environnement ou des fichiers spécifiques à l'environnement (ex : application-test.properties).

Tableau 17. Principales propriétés de configuration BrokerX

Propriété	Valeur par défaut (dev)	Description
spring.application.name	LOG430_BrokerX	Nom de l'application
spring.datasource.url	jdbc:postgresql://localhost:5432/brokerxdb	URL de la base de données PostgreSQL
spring.mail.host	smtp.gmail.com	Hôte SMTP pour l'envoi d'e-mails
server.port	8081 (dev) 8090 (prod)	Port HTTP de l'application (dev)
jwt.secret	your-256-bit-secret-key-for-development-change-in-production-please	Clé secrète JWT (dev)
jwt.expiration	86400000	Durée de validité des

Propriété	Valeur par défaut (dev)	Description
		tokens JWT (ms)
spring.thymeleaf.cache	false	Cache des templates Thymeleaf
management.endpoints.web.exposure.include	health,info,metrics	Endpoints exposés pour le monitoring
app.base-url	http://localhost:8090 (prod)	URL de base de l'application (prod)
logging.level.com.monolithique.log430	INFO (prod)	Niveau de log pour le code métier

La configuration est centralisée et facilement modifiable selon l'environnement (développement, production). Les propriétés sensibles (mots de passe, clés) doivent être gérées via des variables d'environnement en production.

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

BrokerX est une application développée from scratch en Java/Spring Boot. Aucune migration de données ou d'application antérieure n'a été réalisée : toutes les données ont été créées directement dans la base PostgreSQL du projet.

11.16 Testabilité

Le projet contient des tests automatisés (JUnit) dans le dossier standard `src/test/java` d'un projet Maven. Les tests couvrent les services métier, les contrôleurs web et les entités principales. Les tests sont exécutés à chaque build Maven et ne doivent pas être ignorés.

11.17 Gestion du build

L'application se construit avec Maven sans dépendances externes hors Maven. Toutes les étapes (compilation, tests, packaging) sont automatisées via le pipeline CI/CD. Le build produit un JAR exécutable prêt à être déployé dans Docker.

12. Décisions d'architecture

12.1 Architecture hexagonale vs MVC

ADR 001 : Architecture hexagonale vs MVC

Statut : Acceptée
Date : 2025-09-23

Contexte

Le projet BrokerX doit adopter une structure claire, évolutive et adaptée à la complexité métier (gestion d'ordres, portefeuille, sécurité, intégration de services externes). Deux styles sont envisagés : MVC (Model-View-Controller) et Hexagonal (Ports & Adapters).

Décision

Nous choisissons l'architecture hexagonale pour BrokerX. Ce style permet une séparation stricte du domaine métier des dépendances techniques, facilite l'intégration de nouveaux services, la testabilité, et prépare le projet à une éventuelle modularisation ou migration vers des microservices.

Conséquences

- Le domaine métier est indépendant des frameworks et de l'infrastructure.
- Les ports et adapters facilitent l'intégration de services externes.
- La structure est plus modulaire et évolutive qu'un MVC classique.
- La courbe d'apprentissage peut être plus élevée pour les nouveaux développeurs.
- La documentation et la communication sont facilitées.

12.2 Persistance des données

ADR 002 : Persistance des données

Statut : Acceptée
Date : 2025-09-23

Contexte

BrokerX doit stocker de façon fiable les utilisateurs, ordres, transactions, stocks, et assurer la traçabilité (audit). Plusieurs options sont possibles : base de données relationnelle, NoSQL, fichiers, etc.

Décision

Nous retenons une base de données relationnelle (ex : PostgreSQL) pour la persistance principale. Les ports du domaine définissent les interfaces, et les adapters d'infrastructure implémentent l'accès aux données via JPA/Hibernate.

Conséquences

- Les modèles métier sont mappés sur des tables relationnelles.
- La cohérence transactionnelle est assurée.
- La migration vers d'autres solutions (NoSQL, cloud) reste possible via de nouveaux adapters.
- Les requêtes complexes et la traçabilité sont facilitées.
- La gestion des migrations de schéma doit être planifiée.

12.3 Journalisation des opérations utilisateur

Journalisation des opérations utilisateur

Statut : Acceptée
Date : 2025-09-23

Contexte

La traçabilité des actions des utilisateurs est essentielle pour la conformité réglementaire (KYC/AML), la sécurité et l'analyse métier. Plusieurs approches sont possibles : logs applicatifs, solutions externes (ELK, SIEM), ou persistance dédiée en base de données. Il faut garantir la robustesse, la facilité d'accès et la pérennité des données d'audit.

Décision

Nous avons choisi de créer une table dédiée `UserAudit` dans la base de données. Toutes les opérations importantes (inscription, authentification, ordres, transactions, vérifications) sont journalisées avec les informations pertinentes (horodatage, identifiant utilisateur, type d'action, détails). Cette solution permet une requêtabilité directe, une intégration simple avec le domaine, et une conformité aux exigences internes et externes.

Conséquences

- La traçabilité est centralisée et facilement accessible pour les audits et analyses.

- La solution est évolutive : de nouveaux types d'événements peuvent être ajoutés facilement.
- Les performances sont maîtrisées, car la journalisation est intégrée au modèle de données.
- L'intégration avec des outils externes reste possible via export ou synchronisation.
- La gestion de la volumétrie et de la rétention des données doit être planifiée pour éviter l'engorgement.

12.4 Justification de l'architecture

Architecture de BrokerX

BrokerX adopte une **architecture hexagonale (Ports & Adapters)**.

Justification du choix

- **Séparation stricte du domaine métier** : Le cœur métier (modèles, services, logique métier) est isolé des dépendances techniques (frameworks, bases de données, API externes). Les règles métier restent stables et compréhensibles, même si la technologie évolue.
- **Évolutivité et adaptabilité** : L'architecture hexagonale facilite l'ajout ou le remplacement de services externes (paiement, données de marché, audit) sans impacter le domaine. Par exemple, un changement de fournisseur de paiement n'affecte que l'adapter correspondant.
- **Facilité de test et de validation métier** : Le découplage permet de tester le domaine indépendamment des frameworks, ce qui accélère la validation métier et la détection des régressions. Les ports facilitent la création de mocks pour les tests unitaires et d'intégration.
- **Dépendances dirigées et absence de cycles** : Les dépendances sont orientées du domaine vers les ports, puis vers les adapters, ce qui évite les cycles et rend l'architecture plus robuste et maintenable.
- **Réduction du couplage** : Le domaine ne dépend jamais de l'infrastructure ou des frameworks, ce qui permet de migrer vers une autre technologie (ex : changement de framework, passage à une architecture microservices) sans refonte du métier.
- **Conformité réglementaire et sécurité** : En isolant le métier, on facilite la traçabilité et la conformité (audit, KYC, MFA), car les règles métier sont centralisées et documentées.
- **Clarté et communication** : Ce style favorise une documentation claire et une compréhension partagée entre les équipes métier et technique, car chaque couche a une responsabilité bien définie.

Ce style répond aux besoins de BrokerX : évolutivité, conformité réglementaire, sécurité, intégration de nouveaux services externes, et robustesse face aux changements technologiques.

Architecture monolithique et évolutivité

BrokerX est une application monolithique : toutes les fonctionnalités métier, la logique applicative et l'intégration technique sont regroupées dans un même déploiement.

- **Un seul artefact déployé** : toutes les couches (domaine, application, infrastructure, configuration) sont assemblées et exécutées ensemble, ce qui simplifie la gestion et la supervision.

- **Centralisation des règles métier** : la logique métier et les processus sont gérés dans un même espace, ce qui facilite la cohérence et la traçabilité.
- **Gestion transactionnelle fiable** : les opérations complexes (dépôt, placement d'ordre, audit) bénéficient d'une gestion transactionnelle robuste, sans complexité distribuée.
- **Modularité interne** : chaque couche et chaque port/adaptateur est clairement séparé, ce qui permet d'ajouter, modifier ou remplacer des modules sans impacter le reste du système.
- **Préparation à la modularisation** : les ports et adapters peuvent être extraits vers des microservices ou modules indépendants si le besoin de scalabilité ou d'évolution se présente.
- **Facilité d'intégration de nouveaux services** : l'ajout de services externes (paiement, données de marché, audit) se fait via de nouveaux adapters, sans toucher au domaine métier.
- **Clarté et robustesse** : la séparation stricte des responsabilités rend le code plus lisible, plus maintenable et facilite l'intégration de nouveaux développeurs.

Ainsi, BrokerX combine la simplicité et la robustesse du monolithe avec la modularité et l'évolutivité de l'architecture hexagonale.

Description des couches et dépendances

L'architecture 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).
- **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

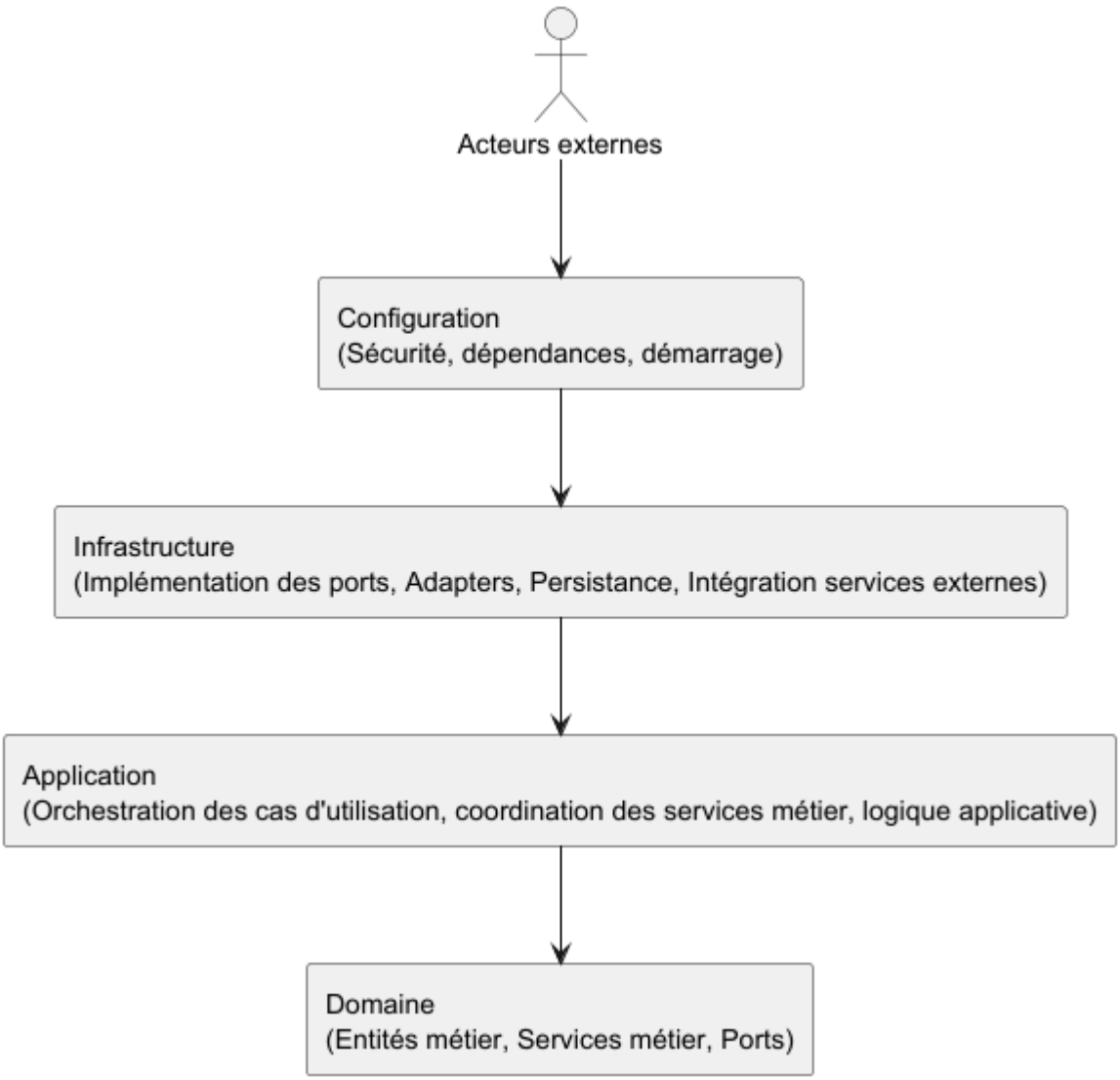
- Le domaine reste totalement indépendant des frameworks et de l'infrastructure, ce qui garantit sa stabilité et sa portabilité.
- Les ports définissent les points d'extension et d'intégration ; les adapters les implémentent pour chaque technologie ou service externe.
- La configuration injecte les dépendances 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



Justification globale

L'architecture hexagonale appliquée au monolithe BrokerX :

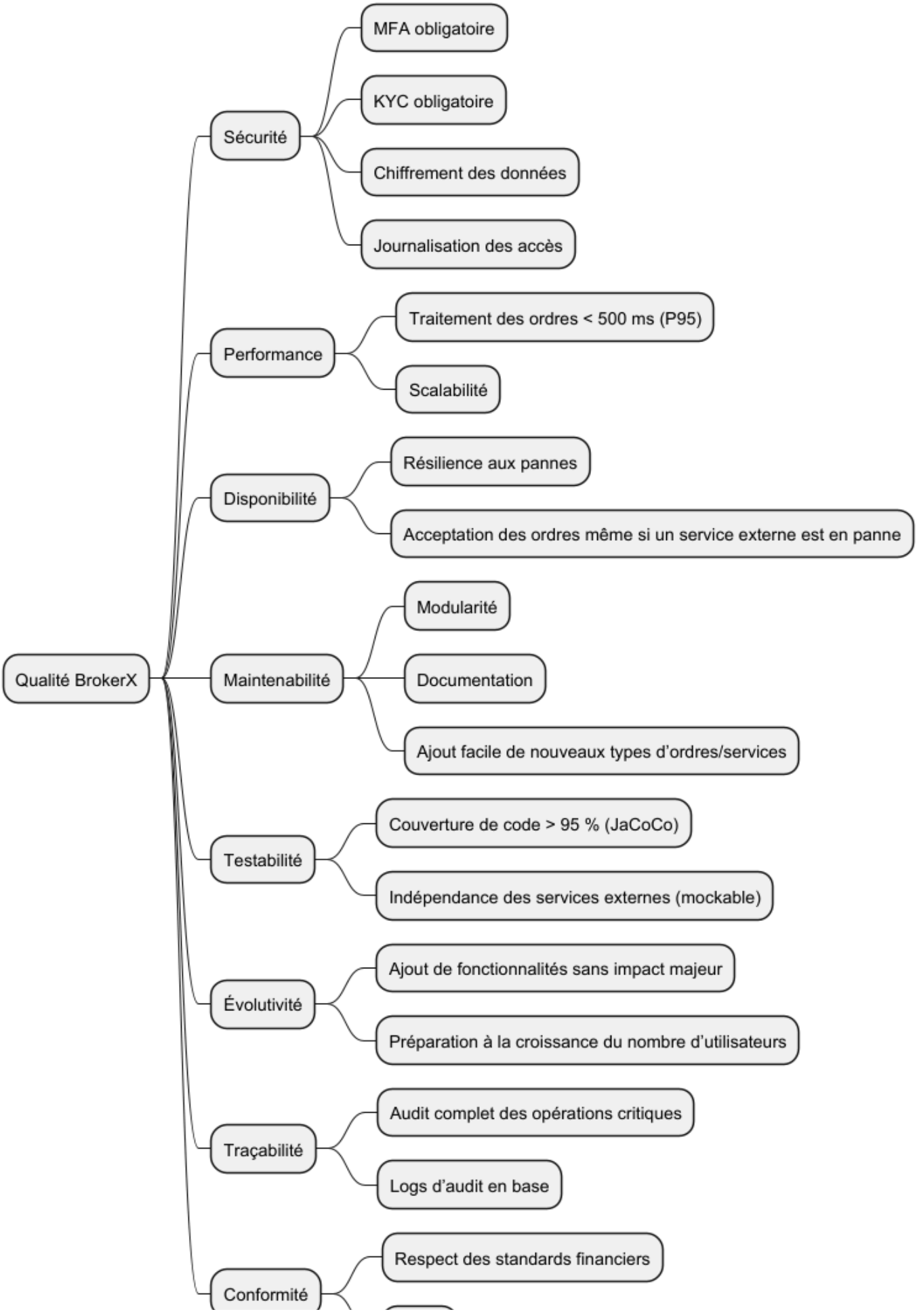
- Permet de faire évoluer la plateforme sans refonte majeure, en isolant le métier des choix techniques.
- Sécurise le domaine contre les changements techniques et réglementaires, en centralisant les règles métier.
- Facilite l'intégration de nouveaux services externes (paiement, données de marché, audit) 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.

- Prépare la plateforme à une éventuelle migration vers une architecture distribuée ou microservices, sans perte de logique métier.

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.


Qualité BrokerX
Sécurité
MFA obligatoire
KYC obligatoire
Chiffrement des données
Journalisation des accès
Performance
Traitement des ordres < 500 ms (P95)
Scalabilité
Disponibilité
Résilience aux pannes
Acceptation des ordres même si un service externe est en panne
Maintenabilité
Modularité
Documentation
Ajout facile de nouveaux types d'ordres/services
Testabilité
Couverture de code > 95 % (JaCoCo)
Indépendance des services externes (mockable)
Évolutivité
Ajout de fonctionnalités sans impact majeur
Préparation à la croissance du nombre d'utilisateurs
Traçabilité
Audit complet des opérations critiques
Logs d'audit en base
Conformité
Respect des standards financiers

13.2 Scénarios d’évaluation

Testabilité / Couverture

- L’utilisation de JaCoCo pendant le développement et 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

- L’architecture doit permettre de tester toute logique métier dépendant de services externes (ex : envoi d’e-mails, accès base de données, fournisseurs de données de marché) sans avoir besoin de ces services réels. Toutes les dépendances externes doivent être mockables via des interfaces ou des adapters.

Exemple :

- Le service d’envoi de codes MFA utilise une interface EmailProvider. Lors des tests, un mock de cette interface permet de valider la logique métier sans connexion SMTP réelle.

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.

Performance / Traitement des ordres

- Le système doit traiter un ordre d’achat ou de vente en moins de 500 ms (P95) même sous charge. Des tests de performance automatisés doivent valider ce critère.

Disponibilité / Résilience

- 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. Des scénarios de test doivent simuler la défaillance de chaque dépendance externe.

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

BrokerX a été conçu selon des standards éprouvés et bénéficie d'une architecture robuste, éprouvée en environnement de développement et de test. À ce jour, aucune faille technique majeure n'a été identifiée pour les scénarios d'utilisation prévus.

Cependant, certains risques techniques subsistent :

- **Risque de corruption de la base PostgreSQL** en cas d'arrêt brutal du serveur (panne matérielle, crash OS, coupure électrique). Ce risque est atténué par l'utilisation de volumes Docker persistants et la mise en place de sauvegardes régulières de la base de données.
- **Dépendance à des services externes (SMTP, fournisseurs de données de marché)** : une indisponibilité temporaire de ces services peut impacter certaines fonctionnalités (ex : envoi de codes MFA, notifications). L'architecture prévoit une gestion de la résilience : les ordres et opérations critiques restent acceptés et journalisés même en cas de panne d'un service externe.
- **Montée en charge** : si le nombre d'utilisateurs ou le volume d'ordres croît très rapidement, des ajustements d'infrastructure (scaling, tuning PostgreSQL, optimisation du code) pourraient être nécessaires. Des tests de charge réguliers et une surveillance proactive permettent d'anticiper ce risque.
- **Sécurité** : bien que l'authentification MFA, le KYC et le chiffrement soient en place, le risque d'attaque (phishing, brute force, faille 0-day) ne peut jamais être totalement éliminé.

En résumé, les principaux risques techniques sont identifiés, documentés et font l'objet de mesures de mitigation adaptées à la criticité de BrokerX.

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 18. 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).
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).

Terme	Définition
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).
Bande 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.
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

Terme	Définition
	size, bande de prix, quantité, etc.).

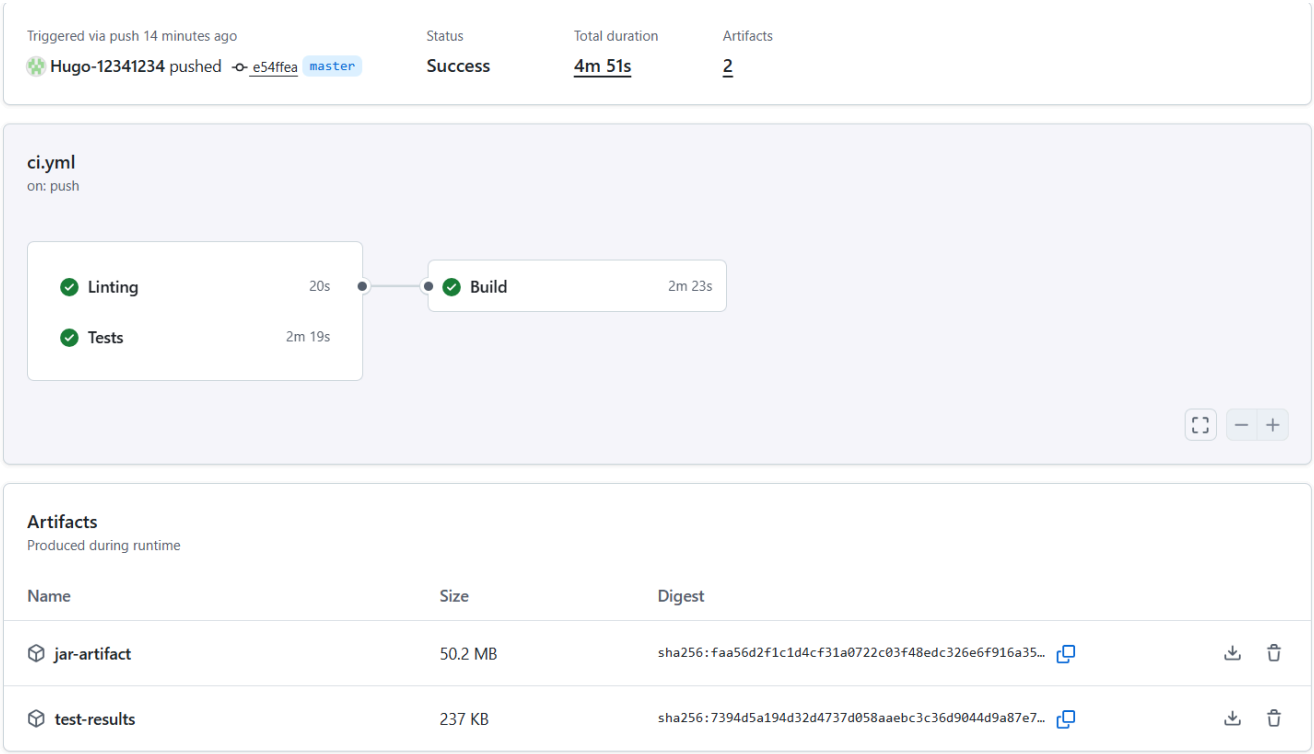
Fin de la documentation arc42

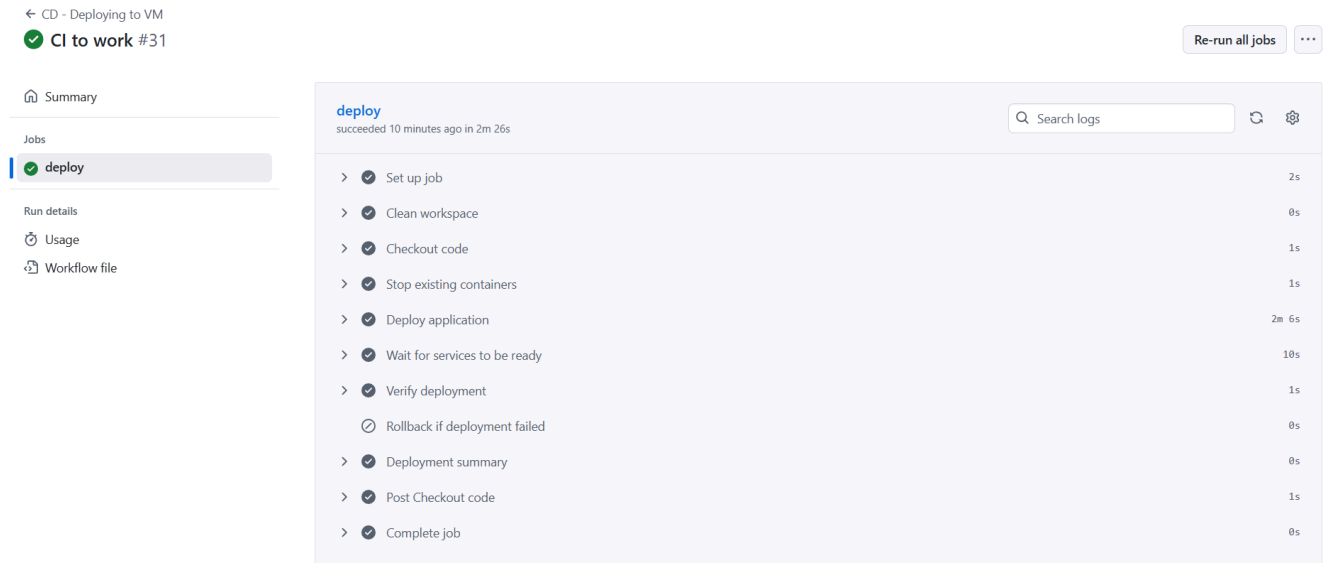
Explication des travaux CI/CD accomplis

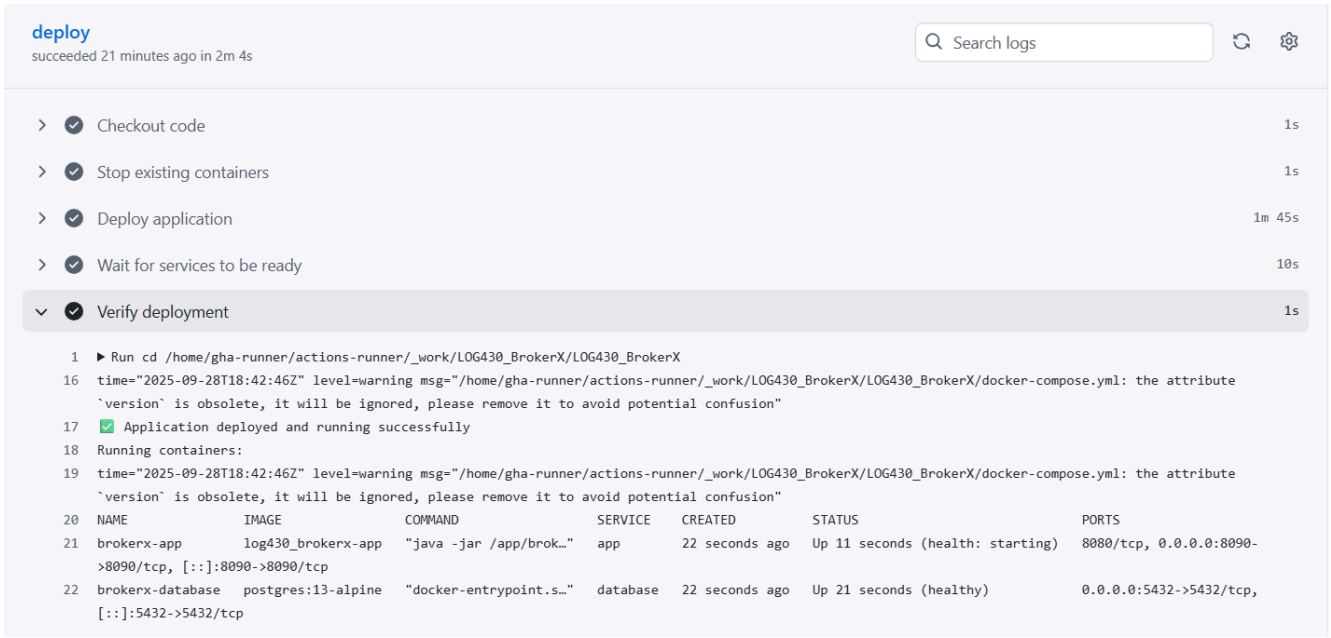
Le projet BrokerX intègre un pipeline CI/CD automatisé pour garantir la qualité et la fiabilité des livraisons. À chaque push ou pull request, le pipeline effectue les étapes suivantes :

- **Compilation du code** avec Maven
- **Exécution de la pyramide de tests**
- **Vérification du style et de la qualité du code**
- **Construction de l'image Docker** de l'application
- **Déploiement automatique** sur la machine virtuelle

Chaque étape est validée avant de passer à la suivante, ce qui permet de détecter rapidement les erreurs et d'assurer la stabilité du projet. Les résultats des pipelines sont consultables dans GitHub, mais comme vous n'y avez pas accès directement, voici quelques captures d'écran illustrant les exécutions :







Ces automatisations assurent que chaque modification du code est testée, validée et déployée, ce qui renforce la fiabilité et la rapidité des livraisons sur BrokerX.

note : Le déploiement automatique est fait par la pipeline CD sur GitHub Actions grâce à un runner "self-hosted" que j'ai créé sur la VM. C'est à dire que dès qu'un push est effectué sur cette branche, l'application est redéployée automatiquement sur la VM. Si vous n'avez pas accès à la pipeline GitHub, veuillez regardez les captures d'écran comme preuves de ce processus.

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)
- 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 fichiers suivants à la racine du dossier :

- `docker-compose.yml`
- `Dockerfile`
- `pom.xml`
- `src/` (code source)
- `docs/` (documentation)

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-prod.properties` ainsi qu'un fichier `application.properties` .

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
```

- Cette commande construit l'image Docker de l'application (si nécessaire) et démarre tous les services (application Java, base PostgreSQL).
- L'option `-d` lance les conteneurs en arrière-plan.

5. Vérification du bon fonctionnement

- Faites un health check à http://localhost:8090/actuator/health
- Accédez à l'interface web : http://localhost:8090
- Vérifiez que la page d'accueil s'affiche.
- Pour consulter les logs de l'application :

```
docker logs brokerx-app
```

  (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 sur le port 5432 (voir `docker-compose.yml` pour les identifiants).

6. Arrêt de l'application

Pour arrêter tous les services :

```
docker-compose down
```

7. Nettoyage (optionnel)

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

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

8. 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.

9. Pour rouler les tests en local

- **Depuis l'IDE** :
  i. Assurez-vous que les conteneurs Docker sont en marche.

ii. Ouvrez votre fichier `application-prod.properties` et remplacez la ligne `spring.datasource.url=jdbc:postgresql://database:5432/brokerx` par `spring.datasource.url=jdbc:postgresql://localhost:5432/brokerx` .
iii. Allez dans le fichier BrokerXTradingWorkflowE2E.java et assurez vous que la ligne @ActiveProfile("prod") n'est pas commentée.
iv. Ouvrez votre terminal au dossier root du projet.
v. Assurez-vous d'avoir maven installé, puis faites ▶ `mvn test` ou `.\mvnw.cmd test` .
vi. Si vous avez des problèmes de dépendances, faites ▶ `mvn clean install` ou `.\mvnw.cmd clean install` avant de relancer les tests.

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:8090.
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 la section "Portefeuille".
2. Indiquez le montant à déposer (veuillez respecter les montants minimum et maximum inscrits).
3. Cliquez sur "Effectuer le dépôt".
4. Le solde de votre portefeuille est mis à jour.

4. Placement d’un ordre (UC-05)

1. Cliquez sur le bouton "Placer un ordre".
2. Lisez bien les instructions vous informant des règles de placement d’ordre (ex. : Seulement TEST est permis comme symbole).
3. Choisissez le type d’ordre (Marché ou Limite), la quantité, et le prix si nécessaire.
4. Cliquez sur "Placer l'ordre".
5. 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).

5. Déconnexion

- Cliquez sur “Déconnexion” pour quitter votre session en toute sécurité.

Remarques :

- Si vous voulez tester le token, fermez votre onglet, puis réouvrez une onglet et tapez l'url http://localhost:8090/auth/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, confirmation) sont envoyées par e-mail.

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