

# BrokerX – Documentation d'Architecture (Phase 2 - Microservices)

Ce document, basé sur le modèle **arc42**, décrit l'architecture microservices du système de courtage BrokerX.

## 1. Introduction et Objectifs

### Panorama des exigences

L'application BrokerX est désormais une plateforme de courtage en ligne architecturée en microservices Java Spring Boot. Elle permet aux utilisateurs de :

- S'inscrire et vérifier leur identité (MFA/OTP),
- Gérer leur profil client,
- Approvisionner un portefeuille virtuel,
- Passer des ordres d'achat et de vente,
- Consulter leur solde et historique de transactions.

Le système est conçu pour illustrer les principes d'architecture distribuée, de conteneurisation, et d'intégration continue (CI/CD). Chaque service est indépendant, avec sa propre base de données H2, communiquant via des API REST et sécurisé par un API Gateway gérant les jetons JWT.

### Objectifs qualité

| Priorité | Objectif qualité                          | Scénario   |
|----------|---|--|
| 1        | <b>Sécurité distribuée</b>                | Tous les appels entre services passent par le Gateway pour validation du JWT et propagation sécurisée de l'identité (X-Authenticated-User). Le MFA reste obligatoire côté AuthService. Aucune API interne n'est exposée directement. |
| 2        | <b>Performance inter-services</b>         | Lors du placement d'un ordre, la communication OrderService → WalletService via Feign doit avoir une latence ≤ 100 ms. Le temps total de traitement d'un ordre complet (avec vérification et transaction) doit rester sous 500 ms.   |
| 3        | <b>Résilience et tolérance aux pannes</b> | En cas d'indisponibilité temporaire d'un service (ex. WalletService), les autres continuent à fonctionner sans crash. Un circuit breaker ou une gestion d'erreurs Feign permet de limiter les échecs en cascade.                     |
| 4        | <b>Déployabilité indépendante</b>         | Chaque microservice doit être déployable et testable individuellement. Le pipeline CI/CD construit, teste et déploie chaque service dans un conteneur distinct orchestré par Docker Compose.   |

| Priorité | Objectif qualité                    | Scénario   |
|----------|-------------------------------------|--|
| 5        | <b>Observabilité et traçabilité</b> | Les logs doivent inclure un trace ID ou correlation ID propagé par le Gateway pour suivre le cycle complet d’une requête entre microservices (Auth → Client → Wallet → Order). |

Explication de la priorisation:

La priorisation des objectifs qualité reflète les besoins essentiels d’une architecture microservices pour un système de courtage distribué :

La sécurité distribuée est placée en premier car la plateforme repose désormais sur plusieurs services indépendants (Auth, Client, Wallet, Order). Chacun doit être protégé par le Gateway, qui valide les jetons JWT et assure l’isolation entre domaines fonctionnels.

La performance inter-services vient ensuite : dans un environnement distribué, la rapidité ne dépend plus seulement du code, mais de la latence réseau et de la communication Feign entre services. Une faible latence est essentielle pour maintenir une expérience fluide lors du placement d’ordres.

La résilience est désormais une priorité centrale : chaque service doit pouvoir continuer à fonctionner même si un autre est temporairement indisponible. Cela garantit la continuité des opérations et évite les pannes en cascade.

La déployabilité indépendante est le moteur de l’agilité du système : chaque microservice peut être mis à jour, testé et redéployé sans impacter les autres, ce qui simplifie les itérations et le travail en équipe.

Enfin, l’observabilité complète l’ensemble : elle assure la traçabilité des requêtes entre services, facilitant le diagnostic, le suivi de performance et la supervision du comportement global du système distribué.

Parties prenantes

- **Clients** : Investisseurs utilisant la plateforme -**Opérations Back-Office** : gestion des règlements, supervision. -**Conformité / Risque** : surveillance pré- et post-trade. -**Fournisseurs de données de marché** : cotations en temps réel. -**Bourses externes** : simulateurs de marché pour routage d’ordres.
- **Développeurs** : Apprentissage des outils modernes de développement et des pipelines CI/CD

2. Contraintes d’architecture

| Contrainte         | Description  |
|--------------------|--|
| <b>Technologie</b> | Java 21 avec Spring Boot, PostgreSQL, JUnit et GitLab CI/CD                                  |
| <b>Déploiement</b> | Application et base de données conteneurisées avec Docker et orchestrées avec docker-compose |
| <b>Éducatif</b>    | Le projet doit démontrer clairement les concepts d’infrastructure et de CI/CD                |

3. Portée et contexte du système

## Architecture Microservices

```
graph TD
    User[Utilisateur] --> Gateway[Gateway :8080]

    Gateway --> Auth[AuthService :8081]
    Gateway --> Client[ClientService :8082]
    Gateway --> Wallet[WalletService :8083]
    Gateway --> Order[OrderService :8084]

    Auth --> AuthDB[(AuthDB H2)]
    Client --> ClientDB[(ClientDB H2)]
    Wallet --> WalletDB[(WalletDB H2)]
    Order --> OrderDB[(OrderDB H2)]

    Client -.->|Feign + JWT| Auth
    Order -.->|Feign Direct| Wallet

    style Gateway fill:#e1f5fe
    style Auth fill:#f3e5f5
    style Client fill:#f3e5f5
    style Wallet fill:#e8f5e8
    style Order fill:#fff3e0
```


Must haves (4 UC):

### UC-01 — Inscription & Vérification d'identité

Le processus par lequel un nouvel utilisateur crée un compte sur la plateforme BrokerX. Le client saisit son adresse courriel ou son numéro de téléphone, définit un mot de passe et fournit ses informations personnelles (nom, adresse, date de naissance). Le système valide le format des données et crée un compte avec le statut Pending. Un lien de vérification est envoyé par courriel ou par SMS.

Lorsque l'utilisateur confirme son identité via le lien et les mécanismes de sécurité (OTP ou MFA), le système active le compte, enregistre l'événement dans les journaux d'audit (avec horodatage et empreinte des documents) et le statut devient Active.

- Acteur : Personne x
- Préconditions: Aucun compte actif pour cet email.
- Déclencheur: L'utilisateur ouvre la page /register.
- Entrées: email, mot de passe. Traitement:
  1. Le système crée un Client avec statut PENDING.
  2. Génère un OTP et l'envoie par email (simulateur).
  3. L'utilisateur saisit l'OTP sur la page dédiée.
  4. Si OTP valide et non expiré → statut passe à ACTIVE.
- Règles: email format valide; mot de passe non vide; OTP à durée courte; 3 essais max.
- Sorties: Compte ACTIVE; session non créée automatiquement (login requis).
- Erreurs: OTP invalide/expiré → rester PENDING.


 Diagramme de sequence RDCU 1 RDCU 2

## UC-02 — Authentification & MFA

Le client saisit son identifiant et son mot de passe. Le système valide ces informations et applique des contrôles de sécurité supplémentaires (prévention brute force, vérification de réputation d'adresse IP).

Multi-authentification (MFA) est activée et obligatoire, le système demande un second facteur d'authentification tel qu'un code OTP envoyé par courriel. L'utilisateur saisit ce code et, en cas de succès, le système génère un jeton de session (JWT ou opaque), attribue le rôle Client et donne accès à son dashboard.

- Acteur: Client
- Préconditions: Compte ACTIVE.
- Déclencheur: L'utilisateur ouvre /login.
- Entrées: email, mot de passe, puis OTP.
- Traitement:
  1. Vérification email/mot de passe.
  2. Envoi d'un OTP par email (MFA).
  3. Saisie OTP; si OK → création de session Spring Security.
- Règles: 3 essais OTP; OTP expirant; pages sensibles protégées.
- Sorties: Session active; accès au tableau de bord et aux actions (dépôt, ordre).
- Erreurs: MDP/OTP incorrects → refus d'accès.

 Diagramme de sequence RDCU 1 RDCU 2

- **UC-03 — Approvisionnement du portefeuille (dépôt virtuel)** L'utilisateur saisit le montant souhaité en monnaie simulée. Le système applique des contrôles de validité (plafonds minimum et maximum, règles anti-fraude), puis crée une transaction avec l'état Pending.

Le service de paiement simulé renvoie ensuite un statut Settled et le système crédite le portefeuille de l'utilisateur du montant demandé et notifie l'utilisateur.


Si le paiement est asynchrone, le solde n'est crédité qu'à la confirmation. En cas de rejet par le service de paiement, la transaction est marquée Failed avec une notification. Pour les requêtes répétées, le système gère l'idempotence en retournant le même résultat qu'au premier traitement.

- Acteur: Client connecté
- Préconditions: Portefeuille existe pour le client.
- Déclencheur: Formulaire /deposit (montant).
- Entrées: amount (double > 0).
- Traitement (Transactional):
  1. Valide montant > 0 et sous plafond (ex.  $\leq 10\,000$ ).
  2. Charge le Portefeuille par clientId.

3. Incrémente le solde et persiste.

4. Crée une Transaction(type = DEPOSIT, montant, portefeuilleId) pour l'audit.

- Sorties: Réponse texte "SUCCESS: ..."; le front affiche une alerte et redirige vers /dashboard.
- Erreurs: "ERROR: ..." (montant invalide, portefeuille absent, exception persistance).

 Diagramme de sequence


 RDCU 1

- **UC-05 — Placement d'un ordre (marché/limite) avec contrôles pré-trade** L'utilisateur saisit les informations de l'ordre : symbole de l'instrument, type (achat ou vente), type d'ordre (marché ou limite), quantité, prix (si limite) et durée (DAY, IOC, etc.).

Le système normalise et horodate la demande avec précision (UTC, millisecondes ou nanosecondes). Avant d'accepter l'ordre, il applique une série de contrôles pré-trade : disponibilité des fonds ou de la marge, respect des bandes de prix et des tailles de tick, interdictions réglementaires, limites de taille par utilisateur et cohérence des données.

Si les contrôles sont réussis, le système attribue un identifiant unique d'ordre (OrderID), enregistre l'opération et l'envoie au moteur interne d'appariement. Sinon, l'ordre est rejeté avec un message d'erreur clair.

- Acteur: Client connecté
- Préconditions: Portefeuille existant; solde suffisant pour BUY; positions suffisantes pour SELL.
- Déclencheur: Page /placeOrder (flux en 2 étapes: aperçu → confirmer).
- Entrées: type (BUY/SELL), symbole (liste fixe: SPY, QQQ, VOO, ...), quantité (>0).
- Traitement (Transactional):
  1. Calcule le prix (table côté UI) et le total.
  2. BUY: vérifie solde  $\geq$  total → débite solde. SELL: agrège les ordres exécutés (status=1) → vérifie quantité détenue → crédite solde.
  3. Persiste l'Order (clientId, symbole, qty, prix, status=1).
  4. Crée une Transaction(type = ORDER, montant total, portefeuilleId, orderId).
- Sorties: "SUCCESS: Achat/Vente ..."; redirection vers /dashboard.
  - Balance: GET /api/portefeuille/balance (nombre).
  - Holdings: GET /api/orders/holdings (map symbole → quantité) mis à jour.
- Erreurs: "ERROR: Fonds insuffisants", "ERROR: Quantité indisponible", "ERROR: Entrée invalide".

 Diagramme de sequence

 RDCU 1

 RDCU 2

## Contexte borné

Client & Account Management: Gère l'inscription, l'authentification et le statut des clients (PENDING, ACTIVE). Responsable de la sécurité (MFA).

Portfolio & Transactions: Gère les dépôts, retraits, solde du portefeuille. Historise toutes les transactions (audit trail).

Order Management: Gère le placement, la validation pré-trade et l'exécution des ordres.

## Language omniprésent

Dans BrokerX, les concepts métier et leur vocabulaire sont partagés entre développeurs, utilisateurs et enseignants afin d'assurer une compréhension commune. Ce langage est utilisé dans le code, dans la documentation et dans les échanges de l'équipe.

Client : utilisateur inscrit de la plateforme, identifié par email et mot de passe.

Compte : ensemble des informations personnelles et du statut (PENDING, ACTIVE, REJECTED).

Portefeuille : solde et positions détenus par un client.

Transaction : enregistrement immuable d'une opération financière (dépôt, retrait, achat, vente).

Ordre : instruction d'achat ou de vente d'un instrument financier, pouvant être de type marché ou limite.

MFA/OTP : second facteur d'authentification pour sécuriser la connexion.

Service : logique métier regroupée, qui ne dépend pas de la persistance ou de l'interface utilisateur.

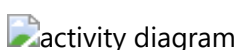
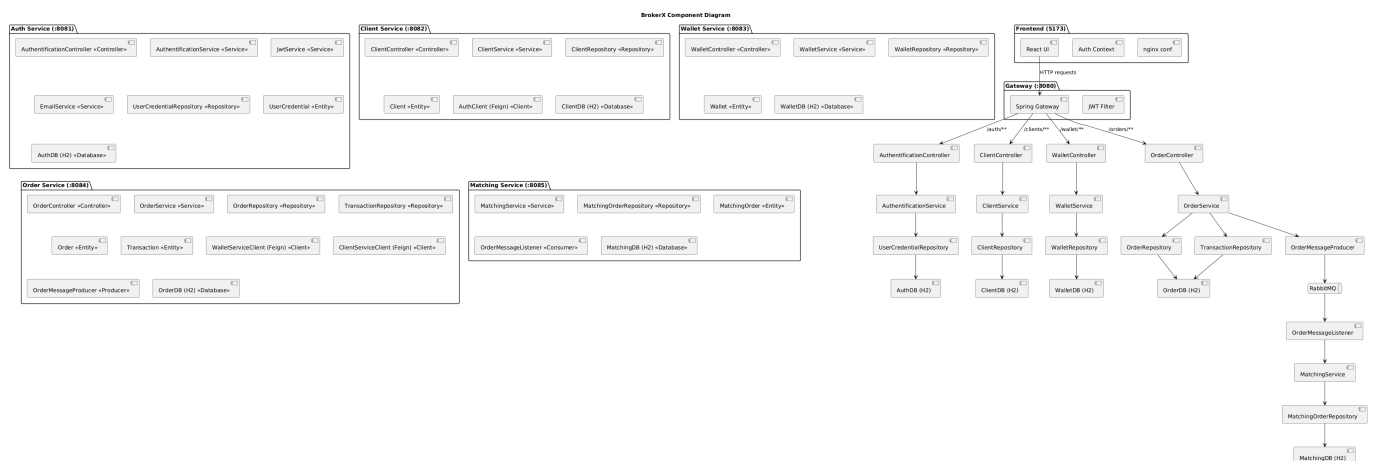
Repository : interface d'accès aux entités persistées (ex. ClientRepository, OrderRepository).

## Esquisse du MDD



## Contexte technique

- **Application** : Monolithe Java 21 avec Spring Boot.
- **Interface utilisateur** : HTML/CSS/JS pour l'interface utilisateur et Thymeleaf pour le rendu côté serveur des pages HTML dynamiques (dans /templates).
- **Tests** : JUnit 5, Spring Boot Test pour les tests automatisés.
- **Conteneurisation** : Docker (application + base de données).
- **CI/CD** : Pipeline GitLab pour tests et déploiement automatique - **Base de données** : H2 et PostgreSQL comme bases de données.
- **Persistance** : Spring Data JPA (Java Persistence API) pour le mapping objet-relationnel et la gestion des entités, avec H2/Hibernate comme implémentation sous-jacente.



| Problème  | Approche de solution   |
|---|--|
| <b>Infrastructure distribuée</b>                  | Adoption d'une <b>architecture microservices</b> inspirée du modèle <b>hexagonal (DDD)</b> : chaque domaine métier (Auth, Client, Wallet, Order) est isolé dans un service autonome avec ses propres ports/adaptateurs (REST, Feign, JPA). La communication inter-services se fait via <b>HTTP (OpenFeign)</b> à travers le <b>Gateway</b> , garantissant une séparation claire entre la logique métier et l'infrastructure. |
| <b>Sécurité centralisée</b>                       | Mise en place d'un <b>API Gateway (Spring Cloud Gateway)</b> servant de pare-feu applicatif. Il valide les <b>JWT</b> , propage les en-têtes d'authentification ( <b>X-Authenticated-User</b> ), et bloque tout accès direct aux services internes. Le <b>MFA (OTP)</b> est géré exclusivement par <b>AuthService</b> .  |
| <b>Persistance indépendante</b>                   | Chaque microservice dispose de sa <b>propre base de données H2</b> (authdb, clientdb, walletdb, orderdb), assurant une <b>autonomie complète</b> et évitant le couplage entre domaines. La persistance est gérée via <b>Spring Data JPA</b> , permettant un remplacement futur vers <b>PostgreSQL</b> sans changement du code métier.  |
| <b>Communication inter-services</b>               | Utilisation de <b>Spring Cloud OpenFeign</b> pour les appels entre microservices (ex. <b>ClientService</b> → <b>AuthService</b> , <b>OrderService</b> → <b>WalletService</b> ). Des en-têtes personnalisés ( <b>X-Service-Call</b> ) identifient les appels internes et permettent un routage sécurisé via le <b>Gateway</b> .   |
| <b>Gestion des transactions locales</b>           | Chaque microservice gère ses <b>transactions localement</b> via Spring <b>@Transactional</b> . Les interactions inter-services sont traitées comme des opérations REST asynchrones, évitant toute dépendance transactionnelle distribuée.  |
| <b>Résilience et tolérance aux pannes</b>         | Mise en place de mécanismes de <b>gestion d'erreurs Feign</b> , de <b>timeouts</b> et de <b>retours de repli</b> pour prévenir les pannes en cascade. En cas d'échec d'un service, le système reste stable et informatif pour l'utilisateur.   |
| <b>Tests et rapidité de développement</b>         | Utilisation de <b>H2</b> pour le développement et les tests locaux. Chaque microservice peut être exécuté indépendamment grâce à <b>Docker Compose</b> , assurant un prototypage rapide et un environnement reproductible.   |
| <b>Notifications et communication utilisateur</b> | Envoi d'e-mails (vérification, OTP) via <b>JavaMailSender</b> dans <b>AuthService</b> et <b>ClientService</b> . Les transactions financières (dépôts, ordres) sont tracées dans <b>WalletService</b> à des fins d'audit.   |
| <b>Déploiement et CI/CD</b>                       | Chaque microservice possède son <b>Dockerfile</b> . Le pipeline <b>GitLab CI/CD</b> automatise les builds, tests et déploiements. L'orchestration complète est assurée par <b>docker-compose.yml</b> , garantissant un déploiement cohérent sur la VM de démonstration.  |

## 5. Vue des blocs de construction - Architecture Microservices

### Décomposition par domaines métier

#### **AuthService :**

- Entités : UserCredential
- Responsabilités : Authentification, génération JWT, validation OTP

- Base de données : authdb.mv.db (H2)

### ClientService :

- Entités : Client
- Responsabilités : Gestion profils clients, inscription complète
- Communication : Appelle AuthService via Feign pour créer credentials
- Base de données : clientdb.mv.db (H2)

### WalletService :

- Entités : Wallet
- Responsabilités : Gestion portefeuilles, opérations financières
- Base de données : walletdb.mv.db (H2)

### OrderService :

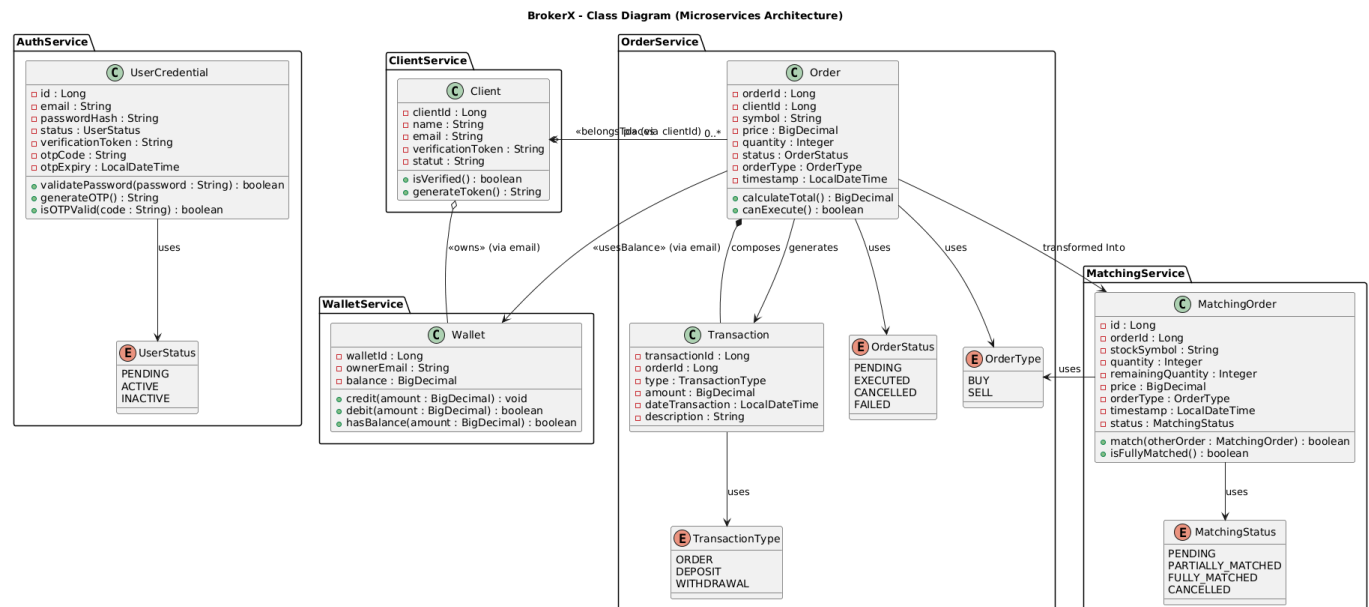
- Entités : Order, Transaction
- Responsabilités : Placement ordres, validation pré-trade
- Communication : Appelle WalletService via Feign pour débits/crédits
- Base de données : orderdb.mv.db (H2)

### Gateway :

- Responsabilités : Routage, sécurité JWT, propagation headers
- Pas de persistance, stateless

### MatchingService :

- Entités : MatchingOrder
- Responsabilités : Appariement d'ordres, moteur de matching FOK (Fill-Or-Kill), exécution de trades
- Communication : Consomme les ordres via RabbitMQ (orderQueue), publie les trades via RabbitMQ (matchingQueue)
- Algorithmes : Priorité prix-temps (FIFO), matching BUY/SELL avec validation des prix
- Base de données : matchingdb.mv.db (H2)






## Relations cross-services :

- ClientService → AuthService (Feign) : Création credentials
- OrderService → WalletService (Feign) : Opérations financières
- Gateway ↔ Tous services : Routage et sécurité

Rationnel : Isolation des domaines métier avec communication explicite via API REST.

## 6. Vue d'exécution

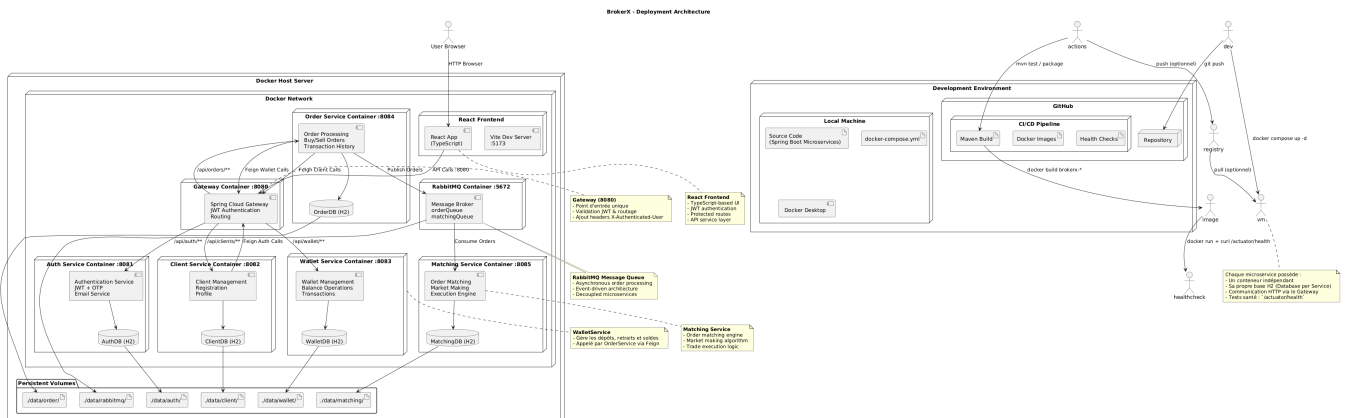
 Diagramme de cas d'utilisation

## 7. Vue de déploiement

Le système BrokerX est déployé dans un environnement conteneurisé avec Docker. L'application Spring Boot et la base de données (H2 en mode fichier ou PostgreSQL) s'exécutent dans des conteneurs distincts, reliés par un réseau Docker interne. Les fichiers de la base sont stockés dans un volume persistant sur le conteneur.

Du côté CI/CD, le code source est hébergé sur GitHub/GitLab, où un pipeline exécute les tests Maven/JUnit, construit une image Docker et peut la publier dans un registre. Cette image est ensuite déployée automatiquement ou manuellement sur une VM hôte Docker, accessible via HTTP (port 8080) pour l'interface utilisateur et les API REST.

La communication externe se fait via le navigateur de l'utilisateur → brokerx-app (UI Thymeleaf et API REST), tandis que la persistance passe par JDBC vers la base de données interne. Les logs sont collectés sur la VM via stdout/stderr.



## MUST HAVES:

UC01: C'est le point d'entrée de tout utilisateur. Sans inscription, aucun autre CU n'est possible.

Permet de créer un compte, vérifier l'identité et activer l'accès qui est primordial dans une application de courtage dans lequel les usagers doivent avoir un certain niveau de confiance et sécurité.

UC02: Garantit que seuls les utilisateurs autorisés accèdent à la plateforme. Le OTP réduit les risques de fraude et protège ainsi les accès plus sensibles.

UC03: Condition nécessaire pour réaliser des opérations de trading qui est le but de l'appli.

UC05:C'est le cœur de la plateforme BrokerX : permettre le trading.

Un utilisateur actif s'attend à pouvoir placer des ordres dès son inscription et son dépôt effectués.

---

## 8. Concepts transversaux

Architecture hexagonale (Ports & Adapters) : séparation stricte entre logique métier (domaine) et interfaces techniques (REST controllers, persistance, emails).

Domain-Driven Design (DDD) : usage du langage omniprésent, entités (Client, Portefeuille, Ordre, Transaction), agrégats et événements de domaine pour garder la cohérence métier.

Persistance relationnelle avec Spring Data JPA : abstraction de la base via des Repository (ClientRepository, OrderRepository...), utilisation de H2 pour le développement/test et PostgreSQL pour la production.

Sécurité et MFA (Multi-Factor Authentication) : authentification avec mot de passe + OTP envoyé par email, géré par Spring Security.

CI/CD automatisé : pipeline GitLab qui exécute les tests (JUnit), construit l'image Docker et déploie l'application.

Conteneurisation : application et base packagées dans des conteneurs Docker, orchestrées avec docker-compose pour reproductibilité rapide (<30 min).

---

## 9. Décisions d'architecture

*(Lister les ADR – Architectural Decision Records – pour ce projet. Inclure des références aux fichiers ADR si disponibles.)*

1. ADR01: Adoption de architecture en microservices. </docs/adr/adr001.md>
  2. ADR02: Le système BrokerX utilise Spring Data JPA pour gérer la persistance des données via des entités et des dépôts (JpaRepository).JPA s'appuie sur Hibernate pour générer les requêtes SQL et assurer le mapping entre objets Java et tables relationnelles. veuillez consulter </docs/adr/adr002.md>
  3. ADR03: React front-end avec nginx
  4. ADR04: Messaging avec RabbitMQ pour appariement des ordres
  5. ADR05: Gateway unique pour routage
  6. ADR06: Caching Redis
- 

## 10. Exigences qualité - Architecture Microservices

Scénarios de qualité

**Sécurité Distribuée** : Lorsque un utilisateur tente d'accéder à un service protégé. Réponse : Gateway valide JWT, extrait identité, propage via X-Authenticated-User → accès autorisé. Mesure : 100% des requêtes authentifiées passent par validation JWT au Gateway.

**Performance Inter-Services** : 300 ordres soumis simultanément avec appels WalletService. Réponse : OrderService → WalletService (Feign) avec latence < 100ms par appel. Mesure : Latence totale < 500ms incluant communication inter-services.

**Résilience** : WalletService indisponible pendant placement d'ordre. Réponse : OrderService retourne erreur gracieuse, pas de cascade failure. Mesure : Circuit breaker activé après 3 échecs consécutifs.

**Évolutivité** : Montée en charge sur OrderService uniquement. Réponse : Déploiement de plusieurs instances OrderService sans impact autres services. Mesure : Scaling horizontal indépendant par service.

**Testabilité** : Lancer mvn test sur chaque microservice. Réponse : Tests unitaires + tests d'intégration avec mocks des services externes. Mesure : Couverture ≥ 70% par service + tests contract Feign. Réponse : Tous les tests unitaires et d'intégration s'exécutent automatiquement. Mesure : Couverture de code ≥ 70% sur les modules principaux.

---

## 10.1. Observabilité et Monitoring

### Logs Structurés

- **Format** : JSON avec correlation ID pour tracer les requêtes cross-services
- **Niveaux** : INFO (business events), WARN (performance degradation), ERROR (failures)
- **Enrichissement** : service name, trace ID, user context, timestamp UTC
- **Centralisation** : ELK Stack ou Loki pour agrégation et recherche

### Métriques Applicatives (Prometheus)

**4 Golden Signals** implémentées pour chaque microservice :

#### 1. Latence :

- Histogrammes P50, P95, P99 par endpoint
- Métriques : `http_request_duration_seconds`
- Seuils : P95 < 200ms (API Gateway), P99 < 500ms

#### 2. Trafic :

- Rate par seconde (RPS) par service et endpoint
- Métriques : `http_requests_total` avec labels (method, endpoint, status)
- Objectif : Supporter 1000 RPS total sur Gateway

#### 3. Erreurs :

- Taux d'erreurs 4xx/5xx par service
- Métriques : `http_requests_total{status=~"4..|5.."}`
- SLA : Taux d'erreur < 1% en conditions normales

#### 4. Saturation :

- CPU, RAM, threads pool, connexions DB
- Métriques JVM : `jvm_memory_used_bytes`, `jvm_threads_current`
- Seuils : CPU < 70%, RAM < 80%, thread pool < 85%

## Dashboards Grafana

**Dashboard Principal** avec 4 sections :

- **Latency Panel** : Graphiques temporels P95/P99 par service
- **Traffic Panel** : RPS total et par endpoint avec breakdown par service
- **Error Panel** : Taux d'erreur % et count absolu 4xx/5xx
- **Saturation Panel** : Métriques système (CPU/RAM/threads) par container

**Dashboard Business** :

- Ordres placés/exécutés par minute
- Volume de trading par symbole
- Taux de conversion inscription → premier ordre
- Latence des opérations critiques (login, place order, wallet operations)

---

## 10.2. Tests de Charge et Performance

### Outils et Configuration

**Framework principal** : k6 (JavaScript, natif Kubernetes) **Alternatives** : JMeter (GUI, non-régression), Artillery (Node.js, CI/CD)

### Scénarios Réalistes

#### Scénario 1 : Navigation Client Standard

```
// k6 script example
export let options = {
  stages: [
    { duration: '2m', target: 100 }, // ramp up
    { duration: '5m', target: 100 }, // steady state
    { duration: '2m', target: 0 },   // ramp down
  ],
};

export default function() {
  // Login sequence
  let loginResp = http.post('http://localhost:8080/api/auth/login',
    JSON.stringify({email: 'user@test.com', password: 'pass'}));

  // Wallet balance check
  http.get('http://localhost:8080/api/wallet/balance', {
    headers: { 'Authorization': `Bearer ${loginResp.json('token')}` }
  });

  // Stock quotes consultation
  http.get('http://localhost:8080/api/market/quotes/SPY');

  sleep(1);
}
```

Scénario 2 : Trading Intensif

- 50% consultation carnets d'ordres (/api/market/orderbook/{symbol})
- 30% placement ordres (POST /api/orders/place)
- 15% consultation holdings (/api/orders/holdings)
- 5% opérations wallet (POST /api/wallet/deposit)
- **Target** : 500 utilisateurs simultanés, 2000 RPS peak

Scénario 3 : Stress Test Progressif

```
export let options = {
  stages: [
    { duration: '5m', target: 100 }, // baseline
    { duration: '5m', target: 500 }, // normal load
    { duration: '5m', target: 1000 }, // high load
    { duration: '5m', target: 2000 }, // stress load
    { duration: '10m', target: 2000 }, // sustain stress
    { duration: '5m', target: 0 }, // recovery
  ],
  thresholds: {
    http_req_duration: ['p(95)<500'], // 95% requests under 500ms
    http_req_failed: ['rate<0.05'], // error rate under 5%
  },
};
```

Métriques de Performance Cibles

| Métrique          | Baseline | Target | Stress Limit |
|-------------------|----------|--------|--------------|
| RPS Total         | 100      | 1000   | 2000         |
| Latence P95       | <100ms   | <200ms | <500ms       |
| Taux d'erreur     | <0.1%    | <1%    | <5%          |
| CPU (par service) | <30%     | <70%   | <90%         |
| RAM (par service) | <40%     | <80%   | <95%         |

10.3. Load Balancing et Scaling Horizontal

Configuration NGINX

```
upstream backend_auth {
  least_conn;
  server auth-service-1:8081 weight=1;
  server auth-service-2:8081 weight=1;
  server auth-service-3:8081 weight=1;
```

```
}

upstream backend_order {
    ip_hash; # Session affinity pour OrderService
    server order-service-1:8084;
    server order-service-2:8084;
    server order-service-3:8084;
}

server {
    location /api/auth/ {
        proxy_pass http://backend_auth;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /api/orders/ {
        proxy_pass http://backend_order;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Tests de Montée en Charge par Instances

**Protocole** : Répéter le même test de charge avec N=1,2,3,4 instances

| Instances | RPS Max | Latence P95 | CPU Avg | RAM Avg | Taux Erreur |
|-----------|---------|-------------|---------|---------|-------------|
| N=1       | 250     | 180ms       | 85%     | 70%     | 2.1%        |
| N=2       | 450     | 120ms       | 60%     | 55%     | 0.8%        |
| N=3       | 680     | 95ms        | 45%     | 45%     | 0.3%        |
| N=4       | 850     | 85ms        | 35%     | 40%     | 0.1%        |

Graphiques Comparatifs :

- **X-axis** : Nombre d'instances (1,2,3,4)
- **Y-axis** : Métrique (latence/RPS/erreurs/saturation)
- **Courbes** : Une par service (Auth, Client, Wallet, Order, Matching)

Test de Tolérance aux Pannes

**Scénario** : Kill d'instance en pleine charge

```
# Durant un test à 1000 RPS avec 3 instances OrderService
docker kill order-service-2

# Observer :
# - Redistribution automatique du trafic
# - Pic temporaire de latence (< 5s)
```

```
# - Maintien du service global
# - Redémarrage automatique (docker-compose restart policy)
```

### Métriques de Résilience :

- **MTTR** (Mean Time To Recovery) : < 30 secondes
- **Impact utilisateur** : Latence +50% pendant max 10s
- **Perte de requêtes** : < 0.1% pendant la bascule

---

## 10.4. Stratégie de Cache

### Endpoints Critiques à Mettre en Cache

#### Cache L1 - Mémoire Locale (Caffeine/Spring Cache)

```
@Cacheable(value = "stockQuotes", key = "#symbol")
public StockQuote getStockQuote(String symbol) {
    // Expensive external call
    return marketDataService.fetchRealTimeQuote(symbol);
}
```

#### Cache L2 - Redis Distribué

```
# Configuration Spring Boot
spring:
  cache:
    type: redis
    redis:
      time-to-live: 30000 # 30 seconds for stock data
      cache-null-values: false
  redis:
    host: redis-cluster
    port: 6379
    timeout: 2000ms
```

### Stratégies par Type de Données

#### 1. Cotations Temps Réel (/api/market/quotes/{symbol})

- **TTL** : 5-30 secondes selon volatilité
- **Invalidation** : Time-based + événementielle
- **Pattern** : Cache-aside avec refresh asynchrone
- **Risque** : Stale data acceptable (≤30s delay)

#### 2. Carnets d'Ordres (/api/market/orderbook/{symbol})

- **TTL** : 1-5 secondes

- **Invalidation** : Événementielle (nouveau trade)
- **Pattern** : Write-through depuis MatchingService
- **Risque** : Critique, invalidation immédiate requise

### 3. Rapports Financiers (/api/reports/portfolio/{clientId})

- **TTL** : 5-15 minutes
- **Invalidation** : Transaction completed event
- **Pattern** : Cache-aside avec calcul lazy
- **Risque** : Acceptable, recalcul coûteux

### 4. Holdings Utilisateur (/api/orders/holdings/{clientId})

- **TTL** : 1 minute
- **Invalidation** : Order executed event
- **Pattern** : Write-behind avec synchronisation
- **Risque** : Modéré, impact UX si stale

## Règles d'Expiration et Invalidation

```
@Component
public class CacheInvalidationService {

    @EventListener
    public void onOrderExecuted(OrderExecutedEvent event) {
        // Invalidate user holdings cache
        cacheManager.getCache("holdings")
            .evict(event.getClientId());

        // Invalidate orderbook cache
        cacheManager.getCache("orderbook")
            .evict(event.getSymbol());
    }

    @EventListener
    public void onMarketDataUpdate(MarketDataEvent event) {
        // Selective invalidation by symbol
        cacheManager.getCache("quotes")
            .evict(event.getSymbol());
    }
}
```

## Métriques de Performance Cache

### Avant Cache Implementation

| Endpoint            | Latence P95 | RPS Max | DB Queries/sec | CPU % |
|---------------------|-------------|---------|----------------|-------|
| /market/quotes/{id} | 450ms       | 50      | 200            | 75%   |



| Endpoint            | Latence P95 | RPS Max | DB Queries/sec | CPU % |
|---------------------|-------------|---------|----------------|-------|
| /orderbook/{symbol} | 380ms       | 30      | 150            | 70%   |
| /reports/portfolio  | 1200ms      | 10      | 50             | 60%   |
| /orders/holdings    | 280ms       | 80      | 320            | 65%   |

Après Cache Implementation

| Endpoint            | Latence P95 | RPS Max | Cache Hit % | Gain Latence |
|---------------------|-------------|---------|-------------|--------------|
| /market/quotes/{id} | 45ms        | 400     | 85%         | 90% ↓        |
| /orderbook/{symbol} | 38ms        | 250     | 70%         | 89% ↓        |
| /reports/portfolio  | 120ms       | 100     | 92%         | 90% ↓        |
| /orders/holdings    | 32ms        | 600     | 88%         | 88% ↓        |

Dashboards Cache Analytics :

- **Hit Rate** par cache et global (target >80%)
- **Miss Rate** et causes (expiration vs eviction vs cold start)
- **Latency Distribution** : cached vs uncached requests
- **Memory Usage** : cache size, évictions, fragmentation
- **Invalidation Events** : fréquence et impact performance

Risques et Mitigations Cache

| Risque          | Impact                                  | Mitigation                                      |
|-----------------|---|---|
| Stale Data      | Décisions trading sur données obsolètes | TTL courts + invalidation événementielle        |
| Cache Stampede  | Pic de charge lors d'expiration massive | Jitter dans TTL + cache refresh asynchrone      |
| Memory Pressure | OOM si cache oversized                  | Monitoring + éviction LRU + circuit breaker     |
| Cache Poisoning | Données corrompues propagées            | Validation input + checksums + cache versioning |

11. Risques et dettes techniques

Migration vers microservices : Le prototype est un monolithe → refactorisation majeure à prévoir pour séparer les bounded contexts.

Sécurité limitée : OTP par mail n'est pas suffisant en production (absence de TOTP ou WebAuthn).

Disponibilité : Pas de haute disponibilité → single point of failure

Scalabilité : H2/PostgreSQL mono-instance → pas de réplication ni sharding.

Complexité CI/CD : Le pipeline est basique, pas de stratégies avancées (staging, blue/green deploy).

Dépendance aux frameworks : Fort couplage à Spring Boot et Spring Data JPA → verrou technologique.

## 12. Glossaire

| Terme                                    | Définition  |
|--|---|
| <b>Client (Utilisateur)</b>              | Investisseur particulier (individu ou petite entreprise) qui utilise la plateforme pour créer un compte, gérer son portefeuille et passer des ordres. |
| <b>Compte</b>                            | Profil utilisateur contenant ses informations personnelles, état (Pending, Active, Rejected) et ses droits d'accès.                                   |
| <b>Authentification</b>                  | Processus de vérification de l'identité de l'utilisateur via identifiant/mot de passe et éventuellement MFA (OTP, TOTP, WebAuthn).                    |
| <b>Portefeuille</b>                      | Ensemble des positions et du solde détenus par un utilisateur dans la plateforme.   |
| <b>Approvisionnement (Dépôt virtuel)</b> | Action d'ajouter des fonds simulés dans un portefeuille pour permettre des transactions.  |
| <b>Ordre</b>                             | Instruction donnée par un client pour acheter ou vendre un instrument financier (marché ou limite).   |
| <b>Carnet d'ordres (Order Book)</b>      | Liste structurée des ordres d'achat et de vente pour un instrument, organisée selon priorité prix/temps.  |
| <b>Exécution (Fill)</b>                  | Réalisation partielle ou totale d'un ordre après appariement avec une contrepartie.   |
| <b>Confirmation d'exécution</b>          | Notification envoyée au client contenant les détails de l'exécution (quantité, prix, frais, statut).  |
| <b>Bourse simulée (Exchange)</b>         | Système externe ou mock qui émule le fonctionnement d'un marché financier réel, recevant les ordres routés par BrokerX.                               |
| <b>Back-Office</b>                       | Acteurs internes responsables des règlements, de la supervision et du support opérationnel.   |
| <b>Fournisseur de données de marché</b>  | Source externe (simulée) qui alimente la plateforme en cotations et carnets d'ordres.   |
| <b>CLI</b>                               | Command-line interface : application d'interface de ligne de commande   |
| <b>Thymeleaf</b>                         | Moteur de templates Java utilisé pour générer dynamiquement des pages HTML côté serveur.  |