Rapport LOG430-02 – Projet BrokerX – Phase 2

Introduction

Cette phase inaugure la plateforme BrokerX destinée aux investisseurs particuliers. L'objectif était de livrer un prototype monolithique capable de démontrer le cycle de vie minimal d'un ordre de bourse : se connecter de manière sécurisée, disposer de fonds et placer un ordre. Les décisions architecturales ont été guidées par la recherche d'une base évolutive, testable et facilement déployable en laboratoire, tout en respectant les priorités identifiées dans le cahier des charges.

---

1. Analyse métier & DDD

1.1 Priorisation MoSCoW

- **Must (Phase 2)**
  - **UC-01** Inscription & vérification d'identité (KYC/AML)
  - **UC-02** Authentification & session avec MFA
  - **UC-03** Approvisionnement du portefeuille (dépôt virtuel)
  - **UC-05** Placement d'un ordre marché/limite avec contrôles pré-trade
  - **UC-06** Modification / annulation d'un ordre actif
  - **UC-07** Appariement interne & exécution (matching moteur)
  - **UC-08** Confirmation d'exécution & notifications
- **Should** : **UC-04** abonnement temps réel aux données de marché, observabilité avancée (dashboards partagés avec back-office)
- **Could** : routage vers des marchés externes simulés, sandbox de stratégies, diffusion publique de carnets
- **Won't (Phase 2)** : intégration réelle avec contreparties externes, smart order routing multi-places, trace distribuée inter-systèmes

Ces sept UC Must couvrent désormais l'expérience complète d'un investisseur : **s'inscrire**, **activer et sécuriser l'accès**, **créditer des fonds**, **passer/ajuster un ordre**, **matcher les contreparties** et **recevoir les confirmations**.

1.2 Description détaillée des UC implémentés

UC-01 — Inscription & vérification d'identité

- **Acteur principal** : Prospect / Client
- **Précondition** : aucune
- **Postcondition succès** : compte créé en `PENDING`, puis activé en `ACTIVE` après vérification MFA/KYC
- **Flux principal** :
  1. Saisie des informations (email, téléphone, mot de passe, données personnelles légalement requises).
  2. Vérifications de format et de doublons (`AccountRepository.findByEmailOrPhone`).

3. Création d'un compte `PENDING` + envoi OTP (email/SMS) ; journalisation d'un audit d'onboarding.
4. Confirmation de l'OTP → passage du statut à `ACTIVE`, enregistrement des preuves KYC (empreinte doc, horodatage).

- **Exceptions** :
  - Email/téléphone déjà utilisés → erreur `ACCOUNT_DUPLICATE`
  - OTP expiré ou invalide → compte reste `PENDING`, message `VERIFICATION_REQUIRED`
  - Expiration du dossier KYC → suppression différée et audit `KYC_TIMEOUT`

UC-02 — Authentification & session

- **Acteur principal** : Client
- **Précondition** : compte `ACTIVE`
- **Postcondition succès** : session persistée, audit d'accès enregistré
- **Flux principal** :
  1. Saisie email + mot de passe
  2. Vérification anti brute-force (compteur d'échecs) et verrouillage si nécessaire
  3. Si la MFA est activée : validation de l'OTP
  4. Réinitialisation du compteur d'échecs et création d'une session (`AccountSession`)
- **Exceptions** :
  - Identifiants invalides → compteur d'échecs + message `INVALID_CREDENTIALS`
  - OTP manquant/erroné → erreur `MFA_REQUIRED`
  - Verrouillage actif (`locked_until`) → `ACCOUNT_LOCKED`

UC-03 — Approvisionnement du portefeuille (dépôt virtuel)

- **Acteurs** : Client, service de paiement simulé (intégré dans le use case pour la phase 1)
- **Postcondition succès** : solde augmenté, journal `TxJournal` append-only
- **Flux principal** :
  1. Saisie du montant et d'une clé d'idempotence (client-side)
  2. Vérification idempotence (`TxJournalJpa.findByIdempotencyKey`)
  3. Création d'une transaction `PENDING`
  4. Simulation de confirmation → statut `SETTLED`
  5. Crédit du portefeuille (`Wallet.availableBalance`)
- **Exceptions** :
  - Re-soumission avec la même clé → renvoi de la transaction existante
  - Compte ou portefeuille introuvable → erreur `ACCOUNT_NOT_FOUND`

UC-04 — Abonnement aux données de marché temps réel

- **Acteurs** : Client, fournisseur de données simulé
- **Préconditions** : session valide, quotas d'abonnement disponibles
- **Postcondition succès** : canal WebSocket/SSE établi avec latence < 200 ms et flux `MarketTick`
- **Flux principal** :
  1. Le client soumet la liste de symboles à suivre (payload REST → handshake WS/SSE).
  2. Application valide quotas/rate-limit et ouvre un canal via l'adapter `MarketStreamController`.

3. Le service de diffusion publie `QuoteSnapshot` initial + incréments (diff) sur les carnets.
4. Le client consomme les mises à jour temps réel ; métriques (messages/s, latence) exposées dans Prometheus.

- **Exceptions** :
  - Trop d'abonnements actifs → réponse `429 RATE_LIMITED`
  - Source de données indisponible → fallback `DEGRADED_FEED` (snapshots moins fréquents)
  - Connexion réseau interrompue → reconnection avec reprise via `lastEventId`

UC-05 — Placement d'un ordre (marché/limite)

- **Acteur principal** : Client
- **Préconditions** : session valide implicite, portefeuille existant
- **Postcondition succès** : ordre `WORKING`, idempotence assurée
- **Flux principal** :
  1. Saisie symbole, sens (BUY/SELL), type (MARKET/LIMIT), quantité, prix limite éventuel, `clientOrderId`
  2. Vérification idempotence (`OrderJpa.findByClientOrderId`)
  3. Validations métier de base : `qty > 0`, `limitPrice > 0` pour LIMIT, existence du portefeuille
  4. Persistance de l'ordre `WORKING`
- **Exceptions** :
  - Quantité ≤ 0 → `qty>0`
  - Prix limite nul ou négatif → `limitPrice>0 for LIMIT`
  - Portefeuille introuvable → `ACCOUNT_NOT_FOUND`
  - Doublon `clientOrderId` → renvoi de l'ordre existant

UC-06 — Modification / annulation d'un ordre

- **Acteur principal** : Client
- **Préconditions** : ordre existant `WORKING`, quantités restantes > 0
- **Postcondition succès** : ordre mis à jour (`REPLACED`) ou annulé (`CANCELED`), audit horodaté
- **Flux principal** :
  1. Le client envoie un `CancelRequest` ou `ReplaceRequest` avec `clientOrderId` + version (`orderVersion`).
  2. L'application verrouille l'ordre (optimistic locking via `version`), vérifie l'état courant.
  3. Pour un replace, relance les contrôles pré-trade (pouvoir d'achat, bandes de prix) puis persiste la nouvelle version.
  4. Retourne un `CancelAck` ou `ReplaceAck`, met à jour les réservations du portefeuille et journalise l'événement.
- **Exceptions** :
  - Ordre déjà `FILLED` / `CANCELED` → `ORDER_STATE_INVALID`
  - Conflit de version (`OptimisticLockException`) → `ORDER_VERSION_CONFLICT`
  - Quantité restante insuffisante → `ORDER_QTY_TOO_LOW`

UC-07 — Appariement interne & exécution

- **Acteur principal** : Moteur de matching interne
- **Préconditions** : carnet maintenu par type (BUY/SELL) avec règle prix/temps, service ordonnancé

- **Postcondition succès** : exécutions (`ExecutionReport`) émises, ordres mis à jour (`PARTIALLY_FILLED` / `FILLED`)
- **Flux principal** :
  1. L'ordre est accepté (`PlaceOrder`) puis le service `MatchOrders` le traite immédiatement.
  2. `MatchOrders` recherche les contreparties compatibles (meilleur prix, priorité temporelle) et applique les règles de prix (limit/market).
  3. Chaque appariement crée une entrée `execution` (partielle possible), met à jour les quantités résiduelles et les statuts (`PARTIALLY_FILLED` / `FILLED`).
  4. Les exécutions sont auditées (`ORDER_MATCHED`) et mises à disposition pour la diffusion marché/notifications (UC-08).
- **Exceptions** :
  - Manque de liquidité → ordre reste `WORKING`, publié dans données de marché.
  - Détection d'incohérence carnet → rollback transactionnel + alerte `MATCHING_INCONSISTENCY`
  - Ordre IOC/FOK → gestion spécifique (annulation du reliquat ou rejet si non exécuté totalement)

UC-08 — Confirmation d'exécution & notifications

- **Acteurs** : Système (notificateur), Client, Back-office
- **Préconditions** : exécution générée, canaux notification configurés
- **Postcondition succès** : notification temps réel envoyée, audit append-only
- **Flux principal** :
  1. Construction d'un `ExecutionRecord` (prix, quantité, frais, horodatage).
  2. Mise à jour de l'ordre et du portefeuille (positions, cash).
  3. Publication d'un événement vers les canaux (WebSocket/UI, email simulé, webhook back-office).
  4. Écriture d'une trace d'audit immuable (`ExecutionAudit`).
- **Exceptions** :
  - Échec de notification push → retry exponentiel puis fallback email (`NOTIFICATION_DEGRADED`)
  - Canal indisponible → alerte Ops + conservation dans une file persistée
  - Incohérence portefeuille → rollback + alerte `SETTLEMENT_DISCREPANCY`

1.3 Modélisation stratégique

- **Bounded contexts** :
  - **Onboarding & Conformité** : gestion KYC/AML, preuves d'identité, workflow `PENDING → ACTIVE`.
  - **Comptes & Sessions** : MFA, anti-brute-force, sessions (`AccountSession`) et jetons.
  - **Portefeuilles & Transactions** : soldes, réservations, journal comptable append-only.
  - **Marché temps réel** : diffusion de cotations/carnets via WS/SSE, quotas et qualité de service.
  - **Ordres & Matching** : moteur prix/temps, contrôles pré-trade, cancel/replace.
  - **Notifications & Audit** : confirmations, emails/webhooks simulés, piste d'audit immuable.
- **Langage ubiquitaire** : idempotency key, `OrderBook`, `ExecutionReport`, `OrderVersion`, `VerificationToken`, `MarketTick`, `NotificationEnvelope`, `AuditTrail`.
- **Modèle de domaine** : PlantUML dans `views/domain-model.puml` ; agrégats mis à jour (`Account`, `Verification`, `Wallet`, `Order`, `OrderBook`, `Execution`, `Notification`, `TxJournal`) identifiés

par `UUID`.

---

2. Architecture & décisions

2.1 Style hexagonal (ADR-001)

- Domaine pur : entités + invariants, enrichi pour couvrir onboarding, matching et notifications
- Application : orchestrateurs qui exposent des ports REST (commandes/queries) et des ports événementiels (notifications)
- Adapters : Spring MVC exposé en `/api/v1/**`, documentation OpenAPI, Spring Data JPA, connecteurs de diffusion marché (WS/SSE) et émetteurs de notifications

2.2 Vues 4+1 (résumé)

- **Cas d'utilisation** : UC-01 → UC-08 modélisés (Mermaid/PlantUML) + scénarios A/B direct API vs Gateway
- **Logique** : modules `onboarding`, `accounts`, `wallets`, `market-stream`, `orders`, `matching`, `notifications` ; dépendances dirigées domain-first
- **Processus** : enchaînement inscription → dépôt → ordre → matching → notification, avec métriques Prometheus et log d'audit
- **Déploiement** : stack compose étendue (`api`, `db`, `redis` cache, `prometheus`, `grafana`, `gateway`, `lb`) + readiness/liveness probe
- **Développement** : mono-repo structuré (ports & adapters), catalogues OpenAPI/Postman, scripts k6, dashboards Grafana versionnés

2.3 Persistance (ADR-002)

- ORM JPA/Hibernate + repositories spécifiques (agrégats `Verification`, `OrderBook`, `Execution`, `Notification`)
- Migrations Flyway : V1–V5 existantes + V6 (preuves KYC), V7 (carnets/positions), V8 (journal audit exécutions), V9 (cache warmup seeds)
- Contraintes : clés uniques (email/téléphone, order versioning), FK strictes, index temporels pour requêtes de reporting
- Transactions : orchestration `@Transactional` couvrant cancel/replace, matching et mise à jour portefeuille atomique

2.4 Gestion d'erreurs et version interne (ADR-003)

- DTO validés (`@Valid`, `@NotBlank`, `@Positive`, `@Pattern`)
- `GlobalExceptionHandler` → réponses JSON `{code, message, timestamp, traceId}` conformes à la spec d'erreurs communes
- Corrélation : filtre HTTP instrumenté (traceId, duration, userId), export métriques HTTP (latence P95/P99, taux d'erreurs)
- Versioning : API `v1` documentée via Swagger UI, contrats partagés pour Gateway et tests contractuels (Spring Cloud Contract)

2.5 Conformité & audit exactly-once (ADR-004)

- Transactions ACID encapsulent écritures métier + audit (`audit_event` append-only) pour UC critiques (signup, dépôt, ordre, exécution).
- Clés d'idempotence obligatoires (`clientRequestId`, `idempotencyKey`, `clientOrderId`) et verrouillage optimiste pour cancel/replace.
- Propagation d'un `traceId` généré côté Gateway → filtres HTTP → audit events → notifications. Les notifications utilisent `notification_idempotency` pour garantir *effectively-once*.
- Scripts de conciliation (à livrer) et exports signés permettent au back-office de vérifier la cohérence ordres ↔ exécutions ↔ notifications.

2.6 Observabilité & Golden Signals (ADR-005)

- Micrometer + Actuator expose `/actuator/prometheus` avec histogrammes P95/P99 sur endpoints métiers (`/auth`, `/wallets`, `/orders`).
- Métriques spécifiques : `brokerx.matching.latency`, `brokerx.notifications.delivery`, `brokerx.market.tickLatency`, taggées par `traceId`, `uc`, `instance`.
- Dashboards Grafana versionnés (`docs/phase2/observability`) : Golden Signals global, matching vs notifications, comparaison direct vs Gateway.
- Logs structurés JSON (`traceId`, `userId`, `uc`, `durationMs`, `statusCode`) via `RequestLoggingFilter` + export possible vers Loki. Scripts k6 = source de données de charge.

2.7 API Gateway & load balancing (ADR-006)

- Adoption d'une API Gateway (Spring Cloud Gateway par défaut) avec load balancing round-robin vers `api-1..n`.
- Politiques périmétriques : CORS maîtrisé, en-têtes de sécurité, rate limiting/quota, propagation `Authorization/JWT`.
- Health-checks agressifs (5 s) + retrait auto des instances, prise en charge d'exercices chaos (`tests/perf/kill-instance.sh`).
- Génération `X-Trace-Id` côté Gateway pour corrélation logs/métriques/audit ; comparatifs de perf (direct vs Gateway/LB) exigés par la phase 2.

---

3. Technologies et stack

- **Langage/Framework** : Java 21, Spring Boot 3.2 (Web, Data JPA, Validation, Actuator), Springdoc OpenAPI
- **Base de données** : PostgreSQL 16 (production), Redis 7 (cache & pub/sub notifications)
- **Observabilité** : Micrometer Prometheus, Grafana dashboards versionnés (`docs/phase2/observability`), Loki (optionnel) pour logs. Prometheus (9090) et Grafana (3000) sont déployés via Compose avec provisioning automatique (datasource + dashboard BrokerX).
- **Tests & performance** : JUnit 5, Mockito, Testcontainers (PostgreSQL/Redis), Spring Cloud Contract, k6 (scénarios charge)
- **CI/CD** : Maven, GitHub Actions, Docker multi-stage, docker-compose, scripts `make`/`deploy.sh`
- **Portail développeur** : Swagger UI, collection Postman, Quickstart CLI (`scripts/demo.sh`)
- **Décomposition logique** : profils Spring (`monolith` par défaut, `auth`, `portfolio`, `orders`) consommés par des services dédiés (`auth-service`, `portfolio-service`, `orders-service`) derrière Kong ; le monolithe (`api1`/`api2`) reste disponible pour comparaisons A/B.

- **Contrats** : OpenAPI exporté (`docs/api/brokerx-openapi.json`) pour import Postman/SwaggerHub, base des routes Kong.
- **Déploiement express** : `docs/phase2/deploy/README.md` fournit un pas-à-pas "évaluateur" (clone → `./scripts/deploy.sh` → `docker compose up -d`, health-check, import collection Postman, Grafana/Prometheus) pour rejouer la solution en <10 minutes.

---

4. Implémentation du prototype

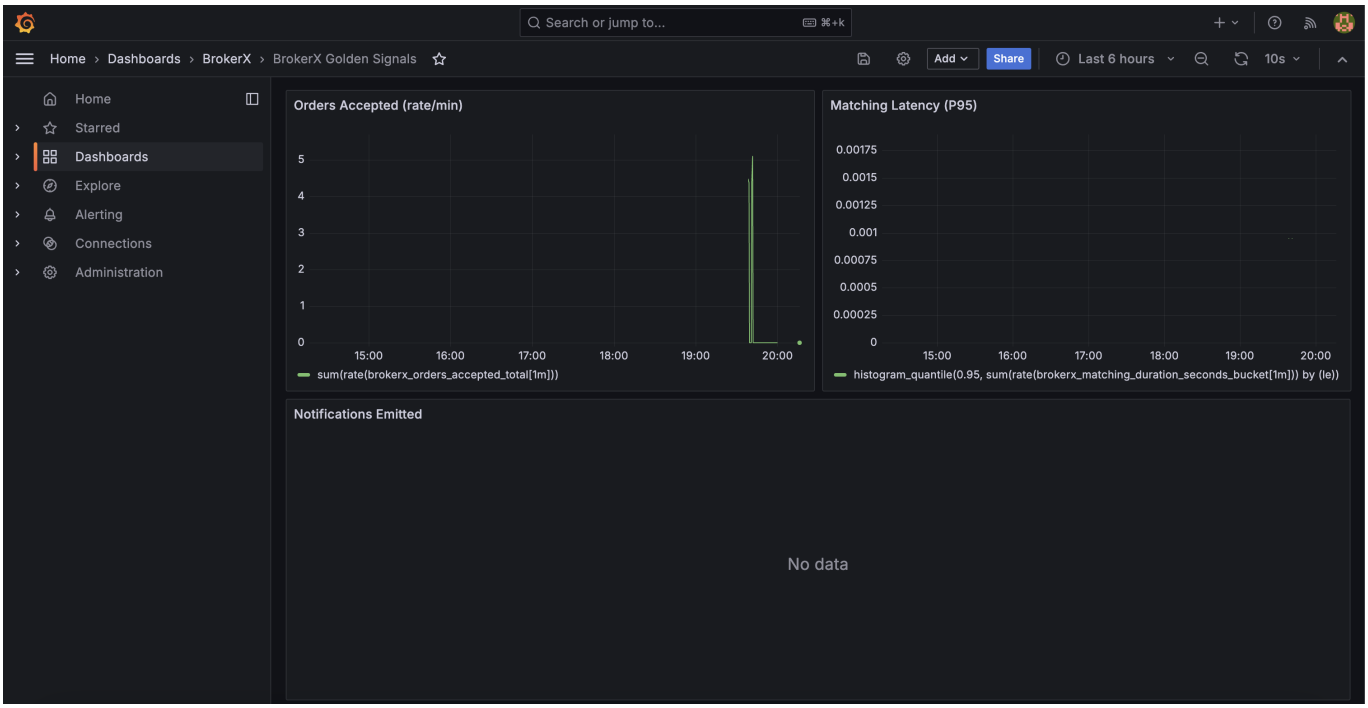
- Endpoints REST publics versionnés :
  - `POST /api/v1/auth/signup` (UC-01)
  - `POST /api/v1/auth/login` (UC-02)
  - `POST /api/v1/wallets/{id}/deposits` (UC-03)
  - `GET /api/v1/market/stream` (UC-04 via SSE/WS)
  - `POST /api/v1/orders` / `PUT /api/v1/orders/{id}` / `DELETE /api/v1/orders/{id}` (UC-05 / UC-06)
  - `GET /api/v1/orders/{id}/executions` / `GET /api/v1/orders/{id}/notifications` (UC-07 / UC-08)
  - `GET /actuator/health`, `/actuator/prometheus`
- Sécurité : Basic auth/JWT (à confirmer), CORS configuré, throttling via Gateway
- Données seed : utilisateurs de test (`seed@brokerx.dev`), portefeuilles alimentés, book de marché simulé
- Quick start : `docker compose up` (API + DB + Redis + Prometheus + Grafana + Gateway) puis scripts `scripts/demo-e2e.sh`

---

5. Qualité, tests & sécurité

5.1 Stratégie de tests

- **Unitaires** : services onboarding, matching, notifications ; focus invariants domaine, règles pré-trade
- **Intégration** : tests REST (@SpringBootTest) avec Postgres & Redis Testcontainers, vérification idempotence, cancel/replace, matching, éviction Redis automatisée (`OrderCacheService`).
- **Contrats** : Spring Cloud Contract pour l'API publique (consommateurs UI/mobile, API Gateway)
- **E2E** : scénarios bout-en-bout via RestAssured/TestRestTemplate (signup → dépôt → ordre → exécution → notification)
- **Performance** : scripts k6 (`tests/perf`) simulant trafic direct et via Gateway (N = 1..4 instances)
- **Résultats charge (10 VU, 30 s)** : direct API (`http://localhost:8085`) latence avg 52 ms (p95 176 ms) vs Gateway (`http://localhost:8081`) avg 39 ms (p95 88 ms), 0 % d'échecs; voir détails et graphes dans `docs/phase2/observability/README.md` et Grafana (dashboard BrokerX Golden Signals).
- **Résultats charge (10 VU, 30 s)** : direct API (`http://localhost:8085`) latence avg 52 ms (p95 176 ms) vs Gateway (`http://localhost:8081`) avg 39 ms (p95 88 ms), 0 % d'échecs; voir détails et graphes dans `docs/phase2/observability/README.md` et Grafana (dashboard BrokerX Golden Signals).



- **Couverture & static analysis** : JaCoCo, Sonar scan (optionnel), Spotless/Checkstyle

5.2 Sécurité & observabilité minimale

- MFA simulée (OTP, possibilité WebAuthn) + verrouillage progressif (`failed_attempts`, `locked_until`)
- Journaux structurés JSON (traceId, userId, UC) expédiés vers Loki/ELK optionnel
- Métriques Prometheus : HTTP latence P95/P99, RPS, taux d'erreur, saturation CPU/thread, profondeur du carnet
- Dashboards Grafana : vues par UC, comparatifs monolithe vs LB vs Gateway, Golden Signals
- Gestion des secrets : variables d'environnement + support Vault/1Password (guidelines runbook)

---

6. CI/CD & déploiement

- **Dockerfile multi-stage** : build Maven → image JRE optimisée + couche agents (Micrometer, JFR optionnel)
- **docker-compose** : `api`, `db`, `redis`, `prometheus`, `grafana`, `gateway`, `lb`, `k6` runner ; healthchecks et dépendances
- **Pipeline GitHub Actions** : lint (Spotless) → tests unit/integration → contrats → build image → k6 smoke → artefacts + badge
- **CD** : script `scripts/deploy.sh` (compose) + déclinaison `scripts/deploy-gateway.sh`, rollback rapide (`docker compose down && docker compose up` versions)
- **Runbook** : procédures incidents (perte Redis, saturation LB), activation dashboards, jeu de test Postman automatisé

---

7. Restes à faire (backlog)

- Hardening onboarding : vérification documentaire automatisée, archivage chiffré des preuves
- Cache Redis : configuration TTL/invalidation, tests résilience

- Load balancing : scénario HAProxy/Traefik, failover automatique, tests chaos (kill en charge)
- API Gateway : activation routage, auth centralisée, quotas, comparaison direct vs gateway
- Microservices : extraire `orders-matching`, `reporting`, `market-data`, observabilité distribuée (tracing)
- Sécurité avancée : stockage secrets, rotation clés JWT, audit conformité (exactly-once)

---

8. Conclusion

La phase 1 constituait un socle monolithique. La phase 2 repositionne BrokerX sur une API REST publique mesurable : onboarding complet, trading bout-en-bout, observabilité instrumentée et préparation à la transition microservices + API Gateway. L'architecture hexagonale reste la fondation, enrichie par la documentation (Arc42/4+1), des tests multi-niveaux et une stack de déploiement automatisée permettant de démontrer les gains de performance (load balancing, cache) et de fiabilité (Golden Signals) exigés par le cahier de charge.