Rapport LOG430-02 – Projet BrokerX – Phase 1

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 1)**
  - **UC-02** Authentification & session (MFA optionnelle)
  - **UC-03** Approvisionnement du portefeuille (dépôt virtuel)
  - **UC-05** Placement d'un ordre marché/limite avec validations pré-trade de base
- **Should** : modification/annulation d'ordre, notifications d'exécution
- **Could** : abonnement aux données de marché en temps réel
- **Won't (Phase 1)** : routage vers des bourses externes simulées, observabilité avancée (traces distribuées)

Ces trois UC Must constituent le noyau minimal pour offrir une expérience de courtage : **authentifier**, **créditer le portefeuille** et **placer un ordre**.

1.2 Description détaillée des UC implémentés

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-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

1.3 Modélisation stratégique

- **Bounded contexts** : Comptes (identité, MFA, sessions), Portefeuilles (soldes, transactions), Ordres (soumissions et statuts). Marché et conformité avancée demeurent en backlog.
- **Langage ubiquitaire** : idempotency key, journal append-only, statut `WORKING`, session, OTP.
- **Modèle de domaine** : PlantUML dans `views/domain-model.puml` ; agrégats `Account`, `Wallet`, `Order`, `TxJournal` avec identifiants `UUID`.

---

2. Architecture & décisions

2.1 Style hexagonal (ADR-001)

- Domain pur : entités + invariants simples
- Application : use cases orchestrant validations, idempotence, transactions
- Adapters : Spring MVC pour HTTP, Spring Data JPA pour la persistance, Bean Validation + handlers pour la sécurité applicative

2.2 Vues 4+1

- **Cas d'utilisation** : login, dépôt, ordre (diagramme Mermaid intégré à Arc42)
- **Logique** : découplage controllers → services → repositories ; dépendances dirigées
- **Processus** : séquence login + dépôt (idempotence et mise à jour du wallet)
- **Déploiement** : deux conteneurs docker-compose (`app`, `db`) avec healthchecks `/health` et `pg_isready`
- **Développement** : organisation du code (`adapters/web`, `adapters/persistence`, `application`, `config`, `domain` implicite)

2.3 Persistance (ADR-002)

- ORM JPA/Hibernate + Modules Spring Data
- Migrations Flyway : V1 schéma, V2 seed initial, V3 contraintes supplémentaires, V4 tables credentials/sessions, V5 seed auth+MFA
- Contrôles BD : FK, CHECK (`amount > 0`, `qty > 0`), contraintes uniques (idempotence, email, client order)
- Transactions : dépôt/ordre encapsulés dans des `@Transactional`

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

- DTO validés (`@Valid`, `@NotBlank`, `@Positive`, etc.)
- `GlobalExceptionHandler` → réponses JSON `{code, message, timestamp}` avec codes HTTP cohérents (400/409/500)
- `RequestLoggingFilter` → logs structurés `method/path/status/duration`

---

3. Technologies et stack

- **Langage/Framework** : Java 21, Spring Boot 3.2 (Web, Data JPA, Validation)
- **Base de données** : PostgreSQL 16
- **Gestion schéma** : Flyway
- **Tests** : JUnit 5, Mockito, Testcontainers (PostgreSQL), JaCoCo
- **CI/CD** : Maven, GitHub Actions, Docker, docker-compose
- **Front de démo** : page HTML/JS simple (`static/index.html`) articulant les 3 UC Must

---

4. Implémentation du prototype

- Endpoints REST internes :
  - `POST /internal/auth/login`
  - `POST /internal/deposits`
  - `POST /internal/orders`
  - `GET /health`
- Interface HTML : formulaires pour chaque UC, journal des réponses
- Données seed (Flyway V2 + V5) : compte `seed@brokerx.dev`, wallet 1000 $, OTP `123456`
- Quick start (README) : docker-compose ou `mvn spring-boot:run`, commandes `curl`

---

5. Qualité, tests & sécurité

5.1 Stratégie de tests

- **Unitaires** : `AuthenticateServiceTest`, `PlaceOrderServiceTest` (Mockito)
- **Intégration** : `AuthIntegrationTest`, `DepositIntegrationTest`, `OrderIntegrationTest` avec PostgreSQL Testcontainers (validations Flyway, idempotence)
- **E2E** : `ScenarioE2ETest` (login → dépôt → ordre via `TestRestTemplate`)
- **Couverture** : `mvn clean test` (+ `mvn verify` pour rapport JaCoCo `target/site/jacoco/index.html`)
- **Validation** : Bean Validation sur tous les DTO et sur `DepositFunds`, `PlaceOrder`

5.2 Sécurité & observabilité minimale

- MFA simulée (OTP stocké pour la phase 1)
- Comptage des tentatives échouées + verrouillage (`failed_attempts`, `locked_until`)
- Logs applicatifs centralisés (filtre HTTP et gestionnaire d'erreurs)
- Secrets gérés via variables d'environnement (pas de credentials en clair dans le code)

---

6. CI/CD & déploiement

- **Dockerfile multi-stage** : build Maven → image JRE légère avec `curl`
- **docker-compose** : services `app` et `db`, healthchecks, réseau dédié
- **Script `scripts/deploy.sh`** : build, `docker compose up -d`, instructions rollback
- **GitHub Actions (`ci.yml`)** : checkout → JDK 21 → `mvn clean verify` → publication artefact → fail rapide en cas de tests KO
- **Runbook (`deploy/README.md`)** : prérequis, commandes, smoke test, gestion des variables d'environnement

---

7. Conclusion

La phase 1 livre un prototype robuste et testable qui couvre les UC essentiels du courtage. L'architecture hexagonale, combinée aux migrations Flyway, aux tests (unit, intégration, E2E) et aux pipelines CI/CD, fournit une base solide pour les phases suivantes. Le Quick Start (<30 min), la documentation consolidée (`Documentation.md`) et l'interface HTML de démonstration permettent à tout nouvel arrivant de cloner, lancer, tester et déployer BrokerX en toute autonomie, en attendant l'implémentation des UC Should/Could et l'évolution vers des architectures distribuées.