Documentation sur le développement

YANG Chen

30/11/2023

Table des matières

1 Besoins du projet

1.1 Introduction

Ce document vise à définir et décrire les besoins de développement de l'application de gestion de portefeuilles financiers. L'application offrira des outils pour créer, gérer et analyser des portefeuilles financiers personnels ou multiples, couvrant les investissements en actions et cryptomonnaies.

1.2 Besoins Fonctionnels

1.2.1 Gestion de Portefeuille

— Création et gestion de portefeuille : Permettre aux utilisateurs de créer et de gérer plusieurs portefeuilles.
— Ajout/Suppression d'actifs : Soutenir l'ajout ou la suppression d'actions et de cryptomonnaies dans le portefeuille.

1.2.2 Suivi des Données en Temps Réel

— Suivi de la valeur en temps réel : Afficher en temps réel la valeur de chaque actif dans le portefeuille.
— Consultation des données historiques : Fournir une fonction pour consulter la valeur et la performance du portefeuille à des moments passés.

1.2.3 Intégration et Stockage des Données

— Intégration API : Obtenir des données en temps réel des marchés financiers via des API publiques.
— Importation de données : Permettre aux utilisateurs d'importer des données de plateformes de transactions externes (comme Coinbase ou Binance).
— Cache de données local : Pour améliorer la vitesse et l'efficacité d'accès, stocker les données localement.

1.2.4 Interface Utilisateur

— Interface graphique : Offrir une interface intuitive et facile à utiliser.
— Visualisation des données : Présenter les actifs et la valeur du portefeuille sous forme de graphiques.

1.3 Besoins Non Fonctionnels

1.3.1 Performance

— Réactivité rapide : Le temps de réponse de l'interface ne doit pas dépasser quelques secondes.
— Fréquence de mise à jour des données : La mise à jour des données financières ne devrait pas avoir plus d'une minute de retard.

1.3.2 Sécurité

— Cryptage des données : Les données sensibles doivent être cryptées lors du stockage local.
— Sécurité d'accès : Nécessiter un mot de passe utilisateur pour accéder à l'application.

1.4 Fonctionnalités Avancées

1.4.1 Fonctions d'Analyse

— Fournir des outils d'analyse de la performance des actifs et des portefeuilles.
— Encourager l'innovation et les fonctionnalités d'analyse personnalisées.

1.4.2 Suivi des Transactions

— Suivre les transactions importantes sur la blockchain, en particulier les transactions de grande valeur (connues sous le nom de "traque des baleines").

1.4.3 Choix de la Devise

— Permettre aux utilisateurs de choisir la devise de référence pour afficher la valeur des actifs (par exemple, EUR, USD, etc.).

1.5 Besoins de l'Interface Utilisateur

— Intuitivité : L'interface doit être intuitive et facile à comprendre, simplifiant l'opération pour l'utilisateur.
— Personnalisation : Offrir une certaine mesure de personnalisation de l'interface et des options fonctionnelles.

1.6 Contraintes Système

— Limitations techniques : L'application doit être compatible avec les systèmes d'exploitation et dispositifs principaux.
— Conformité légale : Respecter les lois et réglementations relatives aux données financières et à la vie privée.

1.7 Spécification Du Code

— Langage de programmation : Java
— Style de codage
  Suivez les conventions de code Java d'Oracle
  Utilisez une indentation appropriée (généralement quatre espaces)
— Structure du projet
  Suivez la structure standard des projets Maven
  Le code source est placé dans src/main/java et le code de test dans src/test/java
— Conventions de dénomination
  Les noms de classes et d'interfaces sont nommés en CamelCase
  Les noms de méthodes et de variables sont nommés en camelCase
  Les constantes sont séparées par des lettres majuscules et des traits de soulignement (par exemple, MAX_VALUE)
— Spécification des annotations
  Utilisez Javadoc pour annoter les classes, les méthodes et les champs publics
  Utilisez des commentaires de ligne pour les segments de code logiques complexes
— Gestion des erreurs
  Préférez la gestion des exceptions au renvoi de codes d'erreur
  Les exceptions personnalisées doivent être claires et significatives
  Évitez les blocs catch vides
— Meilleures pratiques en matière de performances
  Évitez de créer des objets inutiles dans les boucles
  Utilisez des structures de données appropriées
  Optimisez les interactions avec les bases de données et les appels au réseau
— Spécification des tests
  Écrivez des tests unitaires qui couvrent les principales fonctionnalités et les conditions limites
  Utilisez JUnit ou d'autres cadres de test
— Contrôle de version
  Gérez le code à l'aide d'un système de contrôle de version tel que Github
  Suivez un processus clair de gestion des branches et de demande de fusion

2 Méthodologie de développement et gestion de projet

2.1 Choix du modèle en cascade

Ce projet utilise le modèle en cascade comme méthodologie de développement logiciel. Le modèle en cascade est une méthode de développement séquentielle où chaque phase doit être complètement terminée avant de passer à la suivante. Ce modèle convient aux projets avec des objectifs clairs et des exigences stables.

Bien que notre équipe soit petite, composée de seulement deux membres, le choix du modèle en cascade est le plus approprié pour plusieurs raisons :

— **Objectifs clairs du projet** : Les objectifs et les besoins du logiciel sont bien définis au début du projet.
— **Expérience des membres** : C'est la première participation de notre membre Remi à un projet Java, donc une approche structurée et phasée aidera à une meilleure compréhension et participation.

2.2 Planification du projet et chronologie

Voici le plan détaillé du projet et la chronologie prévue :

— **26 novembre 2023** - Lancement du projet
  — Définition des objectifs et des exigences du projet.
  — Discussion et détermination de la conception et de l'architecture de base du logiciel.
— **1er décembre 2023** - Préparatifs initiaux
  — Finalisation du document de besoins.
  — Conception de l'architecture logicielle.
  — Création de la structure du projet.
  — Réalisation des diagrammes UML et des flux de processus métier.
— **3 au 7 décembre 2023** - Création de classes et itération de la documentation
  — Création de toutes les classes nécessaires selon le diagramme UML.
  — Première itération de l'ensemble du document de développement pour assurer l'exactitude et l'exhaustivité de toutes les informations.
— **10 au 14 décembre 2023** - Intégration et tests de l'API
  — Finalisation de l'intégration avec les API externes.
  — Réalisation de tests fonctionnels pour assurer la correcte intégration et la stabilité de l'API.
— **17 au 22 décembre 2023** - Développement de la base de données et du frontend
  — Construction de la base de données.
  — Début du développement du frontend en JavaFx.
  — Commencement du développement du backend.
  — Deuxième itération du document de développement.
— **25 décembre 2023 au 6 janvier 2024** - Développement du backend et tests
  — Finalisation du développement du backend.
  — Début des tests sur l'ensemble du système, incluant les tests unitaires et d'intégration.
— **8 au 12 janvier 2024** - Finalisation des tests et préparation du lancement
  — Achèvement de tous les travaux de test pour assurer la stabilité et la performance du logiciel.
  — Préparation du lancement du logiciel, incluant la documentation finale pour les utilisateurs et le guide de déploiement.

2.3 Remarques

— Comme nous utilisons le modèle en cascade, la transition entre les phases est cruciale. À la fin de chaque phase, une revue complète et une mise à jour de la documentation sont nécessaires.
— Il est essentiel de surveiller attentivement la chronologie du projet pour s'assurer que chaque phase est complétée à temps. Tout problème susceptible d'affecter le calendrier doit être signalé et résolu immédiatement.

3 Environnement et outils de développement

— Système d'exploitation : MacOS Ventura 13.4 & Windows 11
— IDE : Visual Studio Code & IntelliJ IDEA
— Langage de programmation : Java 19.0.2
— Cadre Back-End : Spring boot 2.7.0
— Cadre Front-End : JavaFx 17.0.1
— Base de données : Simuler le stockage d'une base de données à l'aide de fichiers json
— Autre : Maven & GitHub & Draw.io & LaTex

4 Technologie d'architecture et de conception

4.1 Architecture logicielle de haut niveau

Le projet est structuré selon le modèle MVC (Modèle-Vue-Contrôleur), complété par des couches supplémentaires pour la gestion des services et l'accès aux données. Cette organisation vise à séparer les préoccupations, facilitant ainsi la maintenance et l'évolutivité du système.

4.1.1 Modèle (Model)

La couche modèle constitue le cœur de l'architecture logicielle. Elle encapsule la logique métier et les structures de données, gérant les interactions avec la base de données et l'exécution des règles métier. Les entités au sein du dossier `model` représentent les objets du domaine tels que les comptes financiers, les transactions et les portefeuilles d'investissement.

4.1.2 Vue (View)

La couche vue est chargée de l'affichage des données à l'utilisateur et de la gestion des interactions utilisateur. Elle est construite en utilisant JavaFX pour offrir une expérience utilisateur graphique riche et interactive. Les éléments dans le dossier `view` sont des composants graphiques qui présentent les données et réagissent aux actions de l'utilisateur.

4.1.3 Contrôleur (Controller)

Les contrôleurs servent de médiateurs entre la vue et le modèle. Ils interprètent les entrées de l'utilisateur, invoquent des changements sur le modèle et sélectionnent la vue appropriée pour la réponse. Le dossier `controller` contient la logique de navigation et de coordination des flux d'utilisation.

4.1.4 Couche de services (Service)

La couche de services offre une abstraction de la logique métier, exposant un ensemble de services de haut niveau utilisés par les contrôleurs. Elle facilite la réutilisation du code métier et décore le modèle avec des opérations complexes. Le dossier `service` encapsule les processus métier tels que l'analyse financière et la synchronisation des données.

4.1.5 Accès aux données (Repository)

La couche d'accès aux données abstrait la persistance et la récupération des données. Elle fournit une interface pour interagir avec les sources de données, garantissant l'isolation entre la logique métier et les détails de la base de données. Le dossier `repository` implémente les interactions avec la base de données.

4.1.6 Configuration (Config)

La configuration de l'application est gérée dans la couche de configuration. Elle inclut les paramètres de démarrage, les configurations de la base de données, les clés API pour les services externes et les détails de l'injection de dépendance. Le dossier `config` contient des classes de configuration et du code d'initialisation.

Chaque couche est conçue pour être indépendante mais interopérable, permettant une division claire des responsabilités et une évolution simplifiée du système.

4.2 Interaction du système

4.3 Considérations de sécurité et de performance

4.4 Diagramme de processus métier

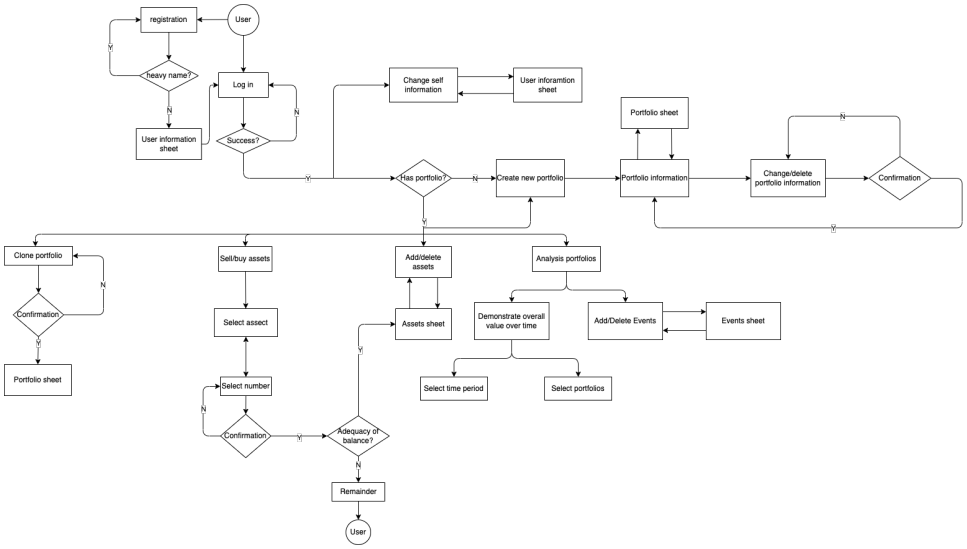

FIGURE 1 – Diagramme de processus métier

4.5 Diagramme de UML

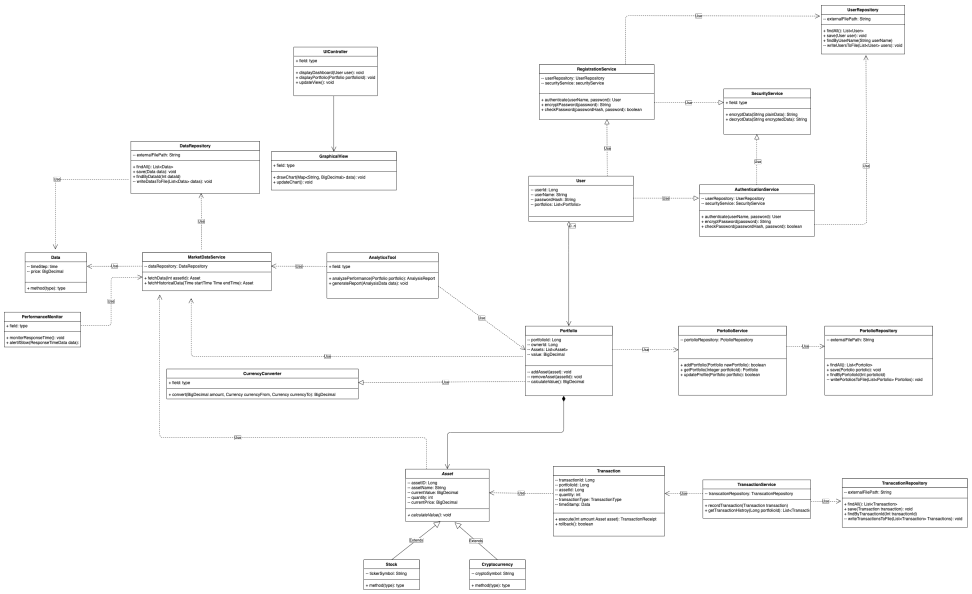

FIGURE 2 – Diagramme de UML

4.6 Changement de Modèle de Conception

4.6.1 Introduction

5 Conception détaillée

5.1 Service

5.1.1 Registration Service (Service d'Enregistrement)

Responsabilités : Traite la logique d'enregistrement des nouveaux utilisateurs.

Méthodes :

— **register**(userName : String, password : String, passwordEnsurance : String) : RegistrationResult

Exceptions :
— **IOException** : Si le fichier de stockage des utilisateurs n'est pas accessible.

5.1.2 Authentication Service (Service d'Authentification)

Responsabilités : Traite la logique d'authentification des utilisateurs.

Méthodes :
— **authenticate**(userName : String, password : String) : AuthenticationResult
— **checkPassword**(passwordHash : String, password : String) : Boolean

5.1.3 Security Service (Service de Sécurité)

Responsabilités : Gère les opérations de sécurité et d'autorisation.

Méthodes :
— **encryptData**(plainData : String) : String
— **decryptData**(encryptedData : String) : String

5.1.4 Portfolio Service (Service de Portefeuille)

Responsabilités : Gère les services de portefeuille d'investissement.

Méthodes :
— **getPortfolio**(portfolioId : int) : Portfolio
— **updatePortfolio**(portfolio : Portfolio) : Boolean
— **addPortfolio**(portfolio : Portfolio) : Boolean
— **createPortfolio**(portfolioName : String, ownerId : int) : Portfolio

5.1.5 Asset Service (Service d'Actif)

Responsabilités : Gère les services d'actif.

Méthodes :
— **getAsset**(assetId : int) : Asset
— **updateAsset**(asset : Asset) : Boolean
— **addAsset**(asset : Asset) : Boolean
— **createAsset**(assetName : String, portfolioId : int, quantity : int, price : BigDecimal, assetType : ASSET_TYPE, interestRate : BigDecimal) : Asset

5.1.6 Transaction Service (Service de Transaction)

Responsabilités : Gère la logique des transactions.

Méthodes :
— **recordTransaction**(transaction : Transaction) : void
— **getTransactionHistory**(portfolioId : int) : List<Transaction>

5.1.7 Market Data Service (Service des Données du Marché)

Responsabilités : Fournit des services de données de marché.

Méthodes :
— **fetchData**(assetId : int) : List<Asset>
— **fetchHistoricalData**(startTime : Time, endTime : Time) : List<Asset>

Exceptions :
— **IOException** : Si l'API externe est inaccessible.

5.2 Repository

5.2.1 User Repository (Dépôt Utilisateur)

Responsabilités : Gère le stockage et la récupération des données utilisateur.

Méthodes :
- **findAll**() : List<User>
- **save**(user : User) : Boolean
- **findByUserName**(userName : String) : User

5.2.2 Data Repository (Dépôt de Données)

Responsabilités : Gère le stockage et la récupération des données d'application.

Méthodes :
- **findAll**() : List<Asset>
- **save**(Data : data) : Boolean
- **findByDataTimeStamp**(dataTimeStamp : Time) : Data
- **findByDataTimeStamp**(startTime : Time, endTime : Time) : List<Data>

5.2.3 Transaction Repository (Dépôt de Transaction)

Responsabilités : Gère le stockage et la récupération des données de transaction.

Méthodes :
- **findAll**() : List<Transaction>
- **save**(transaction : Transaction) : Boolean
- **findByPortfolioId**(portfolioId : int) : List<Transaction>

5.2.4 Portfolio Repository (Dépôt de Portefeuille)

Responsabilités : Gère le stockage et la récupération des données de portefeuille d'investissement.

Méthodes :
- **findAll**() : List<Portfolio>
- **save**(portfolio : Portfolio) : Boolean
- **findByPortfolioId**(portfolioId : int) : Portfolio

5.2.5 Asset Repository (Dépôt d'Actif)

Responsabilités : Gère le stockage et la récupération des données d'actif.

Méthodes :
- **findAll**() : List<Asset>
- **save**(asset : Asset) : Boolean
- **findByAssetId**(assetId : int) : Asset
- **findByOwnerId**(ownerId : int) : List<Asset>

5.3 Model

5.3.1 Analytics Tool (Outil d'Analyse)

Responsabilités : Fournit des outils d'analyse de données.

5.3.2 Performance Monitor (Surveillance de la Performance)

Responsabilités : Surveille la performance du Service des Données du Marché pour déterminer si l'API externe est trop lente.

5.3.3 User (Utilisateur)

Responsabilités : Représente un utilisateur.

Propriétés :
- **userName** : String
- **userId** : int
- **passwordHash** : String
- **portfolios** : List<Portfolio>

5.3.4 Data (Donnée)

Responsabilités : Représente une donnée collecté.

Propriétés :
- **price** : BigDecimal
- **timeStamp** : Time

5.3.5 Authentication Result (Résultat d'Authentification)

Responsabilités : Représente le résultat d'une tentative d'authentification.

Propriétés :
- **sucess** : Boolean
- **failureType** : AUTHENTICATION_FAILIURE_TYPE
- **user** : User

5.3.6 Registration Result (Résultat d'Inscription)

Responsabilités : Représente le résultat d'une tentative d'inscription.

Propriétés :
- **sucess** : Boolean
- **failureType** : REGISTRATION_FAILIURE_TYPE
- **user** : User

5.3.7 Portfolio (Portefeuille)

Responsabilités : Représente un portefeuille d'investissement.

Propriétés :
- **portfolioId** : int
- **ownerId** : int
- **assets** : List<Asset>

5.3.8 Asset (Actif)

Responsabilités : Représente un actif.

Propriétés :
- **assetId** : int
- **assetName** : String
- **assetType** : ASSET_TYPE
- **quantity** : int
- **price** : BigDecimal
- **value** : BigDecimal

5.3.9 Transaction (Transaction)

Responsabilités : Représente une transaction.

Propriétés :
- **transactionId** : int
- **portfolioId** : int
- **assetId** : int
- **quantity** : int
- **transactionType** : TRANSACTION_TYPE
- **timeStamp** : Time

6 Guide de développement et de déploiement

7 Test Plan et Test Cases

8 Manuel de l'utilisateur

9 Évaluation des performances

10 Exemple de code

11 Journal des modifications

12 Annexe