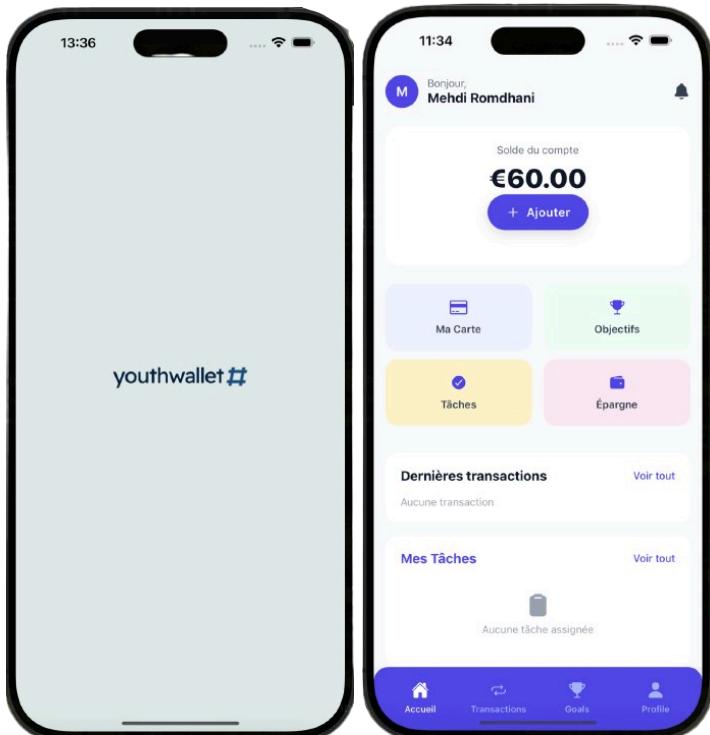


ROMDHANI MEHDI

33 rue borde 13008 Marseille

DOSSIER DE PROJET

youthwallet #



Dossier YouthWallet (conforme RNCP 37873) /

Collaborateur : ROMAIN CHAZAUT - AUGUSTIN YVON

Tables des matières

1. Présentation du projet – YouthWallet.....	5
1.1. Contexte de la formation.....	5
1.2. Présentation de l'équipe projet.....	6
1.3. Présentation de l'application YouthWallet.....	7
1.3. Objectifs pédagogiques et professionnels.....	8
1.4. Inspirations du projet.....	9
2. Expression des besoins du projet – YouthWallet.....	10
2.1. Problématique initiale.....	10
2.2. Objectifs du projet.....	11
2.3. Cibles utilisateurs.....	12
2.4. Fonctionnalités principales attendues.....	13
2.5 – Justification des choix technologiques.....	15
3. Cahier des charges fonctionnel et technique.....	18
3.1 Objectifs fonctionnels.....	18
3.2 Contraintes techniques, de sécurité et d'accessibilité.....	18
4. Spécifications fonctionnelles.....	20
4.1 Diagrammes de cas d'usage (parents/ados).....	20
Diagramme Parent :.....	20
Lecture du diagramme de cas d'usage – Parent.....	21
Diagramme Adolescent.....	23
Lecture du diagramme de cas d'usage – Adolescent.....	24
4.2. Diagramme de classes UML.....	25
Entités principales :.....	25
4.3. Diagramme de séquence UML – Crédit d'une tâche.....	28
4.4 Maquettes UI (Figma).....	31
1.UI Kit : Typographie, couleurs et composants.....	31
2. Composants interactifs et navigation.....	32
Wireframes basse fidélité.....	33
Wireframe de haute fidélité – ex : Dashboard utilisateur.....	35
Conclusion – Spécifications fonctionnelles.....	36
5. Réalisation technique de l'application.....	37
5.1 Conception des interfaces et navigation mobile (Expo Router, UX, RGAA, RGPD).....	37

5.1.2 Design system et maquettes (Figma).....	38
5.1.3 Expérience utilisateur adaptée selon le rôle.....	39
5.1.4 Respect des normes d'accessibilité – RGAA.....	40
5.1.5 RGPD – Protection des données simulée.....	41
5.1.6 Organisation technique du frontend.....	42
Conclusion 5.1.....	42
5.2 – Composants métier, structure backend et logique applicative.....	43
5.2.1 Organisation d'un module métier.....	43
5.2.2 Prisma ORM & schema.prisma.....	50
5.2.3 Les DTO (Data Transfer Objects).....	52
5.2.4 Injection de dépendances (Dependency Injection).....	54
5.2.5 Principe SOLID.....	55
5.2.6 Tests Jest & Postman.....	58
Conclusion – 5.2 Composants métier et logique backend.....	60
5.3 – Base de données relationnelle (PostgreSQL + Prisma).....	61
Choix techniques.....	61
5.3 – Base de données relationnelle (PostgreSQL + Prisma).....	62
MCD – Modèle conceptuel de données.....	62
MLD – Modèle logique de données.....	62
MPD – Modèle physique de données.....	63
Prisma ORM – Mapping de la base.....	63
Outils complémentaires.....	64
5.4 – Développement des composants d'accès aux données SQL et NoSQL.....	65
Accès aux données SQL avec Prisma (YouthWallet).....	65
6. Gestion de projet.....	67
Méthodologie projet – Kanban agile.....	67
Outils utilisés.....	68
Répartition de l'équipe.....	68
7. Tests et validation de la logique métier.....	70
8. Déploiement & DevOps.....	72
Conteneurisation avec Docker.....	72
Intégration continue avec GitHub Actions.....	73
Lancement local et documentation.....	74
9. Veille technologique et UX.....	75
Sécurité backend (NestJS, JWT, Prisma).....	75

ORM Prisma & modélisation.....	75
Gamification & UX mobile.....	76
CI/CD & Docker.....	76
10. Annexes.....	77
10.1 Diagrammes et Modèles de données.....	77
10.2 Références techniques externes.....	77
Conclusion.....	78
Remerciements.....	78

1. Présentation du projet – *YouthWallet*

1.1. Contexte de la formation

Ce projet a été mené dans le cadre du Titre Professionnel de Concepteur Développeur d'Applications (RNCP 37873), diplôme de niveau 6 délivré par le Ministère du Travail. L'objectif de ce projet est de démontrer la maîtrise des compétences attendues dans les trois blocs suivants :

- Bloc 1 : Développer une application sécurisée ;
- Bloc 2 : Concevoir et développer une application organisée en couches ;
- Bloc 3 : Préparer le déploiement d'une application sécurisée dans une démarche DevOps.

YouthWallet a été conçu comme un projet complet, intégrant à la fois les aspects de conception, développement, sécurité, accessibilité, architecture, tests, documentation et déploiement. Ce projet constitue un support de validation technique, pédagogique et professionnel.

1.2. Présentation de l'équipe projet

Le projet **YouthWallet** a été réalisé en trinôme, avec une répartition initiale des rôles claire, tout en favorisant la polyvalence et l'entraide au fil de l'avancement :

- **Augustin Yvon** : chargé de la **conception backend**, il a travaillé sur la structure de l'API REST, les modèles de données avec Prisma, et la sécurisation des accès.
- **Romain Chazaut** : également responsable de la conception backend, il a contribué à la mise en place des routes NestJS, des contrôleurs et des logiques métier complexes ainsi que l'intégration des test.
- **Mehdi Romdhani** : chargé de la **conception frontend**, notamment avec **Expo / React Native**, en assurant la création des interfaces, l'intégration API, l'expérience utilisateur (UX), et la navigation.

Évolution de la collaboration

Tout au long du projet, une méthodologie **Agile Scrum** a été adoptée avec :

- Des **daily meetings** réguliers pour faire le point sur l'avancement et répartir les tâches.
- Des **sprints** courts avec objectifs clairs.
- Un suivi via un **board Trello** pour organiser les tâches et maintenir une bonne visibilité.

L'équipe a fait preuve de souplesse et de complémentarité : chacun a contribué à la fois au **frontend** et au **backend** lorsque nécessaire, pour répondre aux priorités du projet et respecter les délais. Cette dynamique a renforcé la cohésion et permis une montée en compétence mutuelle.

1.3. Présentation de l'application YouthWallet

YouthWallet est une application mobile de gestion budgétaire fictive à visée pédagogique, destinée à un public adolescent et supervisée par leurs parents. Elle permet d'initier les jeunes à la gestion de l'argent de manière ludique, encadrée et progressive.

L'application propose deux profils d'utilisateur distincts :

- **Le parent, qui peut verser de l'argent de poche fictif, attribuer des tâches rémunérées, suivre les dépenses et valider les actions effectuées ;**
- **L'adolescent, qui peut consulter son solde, effectuer des dépenses, atteindre des objectifs d'épargne, et réaliser des missions pour gagner de l'argent.**

L'interface utilisateur est adaptée à chaque profil avec des droits, des vues et des actions différenciés. L'objectif est d'apporter une expérience utilisateur éducative, tout en intégrant des notions concrètes de gestion financière (budget, économies, transaction, récompense).

1.3. Objectifs pédagogiques et professionnels

Le projet YouthWallet permet de valider les compétences techniques attendues dans le référentiel CDA :

- **Installation et configuration d'un environnement de développement :** mise en place d'un backend NestJS avec Prisma, d'une base PostgreSQL dans un conteneur Docker, et d'un frontend mobile via React Native Expo ;
Développement d'interfaces utilisateur mobiles : interfaces adaptatives pour les deux profils utilisateur, en respectant une charte graphique, la logique UX/UI mobile, ainsi que les normes d'accessibilité RGAA ;
Architecture logicielle modulaire : découpage en modules (authentification, utilisateurs, transactions, tâches, objectifs), séparation claire des couches (contrôleur, service, DTO, modèle) ;
- **Gestion sécurisée des données :** rôles, guards, validation des entrées, chiffrement, token JWT, RGPD ;
- **Conception de base de données relationnelle :** conception du MCD, MLD, MPD, mise en œuvre avec Prisma ORM et PostgreSQL ;
- **CI/CD et conteneurisation :** intégration continue via GitHub Actions, conteneurisation via Docker Compose.

Le projet mobilise également des compétences transversales comme :

- **La communication écrite et technique en français et en anglais (résumé, documentation, README) ;**
- **L'organisation en mode projet (Agile/Kanban) ;**
- **La veille technologique (sécurité, UX mobile, benchmarks).**

1.4. Inspirations du projet

Le projet YouthWallet a été inspiré de solutions existantes sur le marché telles que :

- Pixpay : carte bancaire pour adolescents avec suivi parental ;
- GoHenry : système de tâches rémunérées et éducation budgétaire ;
- Mydoh : gamification de la gestion financière familiale ;
- Money Walkie : outils ludiques de suivi des dépenses.

Toutefois, YouthWallet se distingue par son orientation 100 % fictive et pédagogique, sans interaction réelle avec un système bancaire. Cela permet une plus grande liberté de conception tout en répondant à un besoin réel de sensibilisation des jeunes à la gestion de l'argent.

2. Expression des besoins du projet – *YouthWallet*

2.1. Problématique initiale

Dans un contexte où l'éducation financière est rarement abordée dans les programmes scolaires, les adolescents manquent souvent d'outils pour comprendre et gérer un budget. Les parents, quant à eux, expriment souvent le besoin d'encadrer l'apprentissage de leurs enfants de manière ludique et pédagogique.

L'enjeu est donc double :

- Proposer un outil pédagogique, sécurisé et adapté à un jeune public ;
- Permettre aux parents de suivre et encadrer l'évolution de cet apprentissage budgétaire.

YouthWallet vient répondre à ce besoin éducatif en proposant une application mobile ludique et intuitive qui simule une gestion de portefeuille fictif supervisée.

2.2. Objectifs du projet

L'objectif principal est de développer une application complète qui :

- Permet aux adolescents de gérer un budget fictif en autonomie ;
- Donne aux parents les moyens de contrôler, alimenter, et orienter cet apprentissage ;
- Propose une expérience gamifiée favorisant l'engagement et la compréhension des mécanismes financiers de base.

L'application doit respecter les standards en matière :

- d'ergonomie et d'accessibilité (RGAA),
- de sécurité (authentification, validation, séparation des rôles),
- de qualité de code (POO, SOLID),
- et de maintenabilité (architecture modulaire, séparation des responsabilités).

2.3. Cibles utilisateurs

- **Adolescents (13-17 ans) :**
 - **Gèrent leur propre solde fictif ;**
 - **Effectuent des transactions ;**
 - **Créent et suivent des objectifs d'épargne ;**
 - **Réalisent des tâches pour obtenir des récompenses.**
- **Parents :**
 - **Son tuteur des comptes pour leurs enfants ;**
 - **Versent de l'argent fictif ;**
 - **Attribuent et valident des tâches rémunérées ;**
 - **Suivent les dépenses et les objectifs atteints.**

2.4. Fonctionnalités principales attendues

- **Inscription / Connexion sécurisée**
 - **Authentification via JWT, mot de passe chiffré (bcrypt)**
 - **Rôles utilisateur (parent / ado)**
- **Gestion des utilisateurs**
 - **Création de comptes parent / ado**
 - **Attribution des rôles**
 - **Interface différenciée selon le type d'utilisateur**
- **Suivi des transactions**
 - **Affichage temps réel du solde et des dépenses**
 - **Liste filtrable des opérations (classé par catégories)**
 - **Différents types de transaction : dépense, recharge, remboursement**
- **Objectifs financiers (épargne)**
 - **Création et suivi d'objectifs**
 - **Mise à jour automatique de la progression**
 - **Visualisation graphique des avancements**
- **Tâches rémunérées**
 - **Création de tâches par les parents**
 - **Réalisation et validation par les adolescents**
 - **Versement automatique ou manuel d'un montant fictif**

- **Système de notifications**
 - **Notifications push en cas de transaction ou de validation de tâche**
Alertes en cas d'atteinte d'un objectif ou d'événement important
- **Gamification**
 - **Attribution de badges ou trophées pour certaines actions**
- **Dashboard personnalisé**
 - **Vue centralisée avec solde, objectifs, tâches à faire / terminées**
Interface simple et accessible, avec affichage adapté à l'utilisateur

2.5 – Justification des choix technologiques

Le projet YouthWallet a été développé avec des technologies modernes, sélectionnées pour leur robustesse, leur maintenabilité et leur adéquation aux exigences fonctionnelles et techniques.



Frontend – Expo (React Native)

Expo permet de créer une application mobile multiplateforme rapidement, avec une gestion simplifiée des permissions, des notifications et des animations. Ce choix a facilité l'implémentation de deux interfaces distinctes (parent / adolescent) tout en assurant une base de code unique.



Nest JS

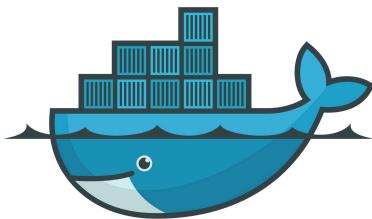
Backend – NestJS avec Fastify

NestJS a été retenu pour sa structure modulaire, son support natif de TypeScript et son alignement avec les principes d'architecture logicielle (séparation des responsabilités, injection de dépendances). Le moteur HTTP **Fastify** a été préféré à Express pour ses meilleures performances.



Base de données – PostgreSQL + Prisma ORM

PostgreSQL a été choisi pour sa fiabilité, sa conformité SQL et sa gestion avancée des types. Couplé à Prisma, il permet une modélisation claire du schéma, une génération automatique des types TypeScript et une gestion simplifiée des migrations.



Conteneurisation – Docker

Docker isole les services (API, base de données, interface d'administration) et garantit un environnement de développement identique sur toutes les machines. Docker Compose facilite l'orchestration de l'ensemble.



GitHub Actions

Intégration continue – GitHub Actions

GitHub Actions assure l'automatisation des tests, des builds et du déploiement à chaque push. Ce choix garantit un suivi constant de la qualité et évite les régressions.

Chaque choix a été dicté par la volonté d'obtenir une base de code solide, performante, facile à maintenir dans le temps, et respectant les standards de sécurité attendus dans une application éducative.

Cette partie mobilise la compétence suivante du référentiel :

Installer et configurer son environnement de travail

Elle a été appliquée dans le projet à travers :

- La création d'un environnement local complet via Docker Compose ;
- L'installation, la configuration et l'interconnexion des outils : NestJS, Prisma, PostgreSQL, Expo ;
- La gestion des variables d'environnement et des configurations multi-profil (parent / ado) ;
- La mise en place de pipelines CI/CD avec GitHub Actions pour automatiser les tests et les déploiements à chaque push.

3. Cahier des charges fonctionnel et technique

3.1 Objectifs fonctionnels

L'application YouthWallet a pour but d'offrir un environnement pédagogique permettant aux adolescents de s'initier à la gestion de l'argent, tout en garantissant aux parents un rôle de supervision et d'accompagnement. Le système repose sur la distinction claire entre deux profils : **le parent et l'adolescent**, chacun disposant de droits et de vues adaptées.

Le parent peut verser de l'argent de poche fictif, créer des tâches, les valider, suivre les objectifs de son enfant, et avoir une vision globale de l'évolution du portefeuille. De son côté, l'adolescent peut gérer ses transactions, réaliser des missions pour gagner de l'argent, fixer des objectifs d'épargne et suivre sa progression.

L'un des piliers fonctionnels repose sur la **gamification**, utilisée ici comme levier d'engagement. L'utilisateur adolescent est encouragé à accomplir des actions (réaliser une tâche, atteindre un objectif, économiser) qui déclenchent des **badges ou trophées symboliques**. Cette dynamique renforce l'implication tout en intégrant des concepts financiers clés (budget, solde, dépenses, épargne).

Enfin, l'application centralise les informations essentielles dans un **dashboard clair**, adapté à chaque profil. Celui de l'adolescent met en avant son solde, ses objectifs en cours, et ses tâches. Celui du parent synthétise les versements effectués, les tâches créées et les progrès observés.

3.2 Contraintes techniques, de sécurité et d'accessibilité

La conception de YouthWallet s'est faite dans un cadre contraint, avec des exigences à la fois techniques, sécuritaires et ergonomiques.

Sur le plan technique, l'application repose sur une architecture en couches modulaires, organisée autour de **NestJS pour le backend et Expo (React Native) pour le frontend**. Chaque domaine fonctionnel (authentification, utilisateurs, transactions, tâches, objectifs) est isolé en modules pour garantir une maintenabilité et une extensibilité du code. **Le backend expose une API REST sécurisée via Fastify**, tandis que le frontend exploite Expo Router pour naviguer entre les vues.

La sécurité a été traitée de manière transversale, dès les premières phases de développement. Les échanges sont protégés par des jetons JWT, les mots de passe sont chiffrés avec bcrypt, et des guards contrôlent les rôles et les droits d'accès. Tous les points d'entrée API utilisent des DTO validés pour empêcher les injections et autres attaques classiques (XSS, CSRF, etc.). L'ensemble du projet respecte également les principes de confidentialité et de minimisation des données recommandés par le RGPD.

Du côté de l'accessibilité, les interfaces ont été pensées pour être simples, lisibles et intuitives, **dans le respect du référentiel RGAA**. Les éléments visuels ont été hiérarchisés, les couleurs contrastées, les textes clairs, et l'ergonomie mobile optimisée. L'objectif est de s'adresser à un jeune public sans sacrifier l'efficacité ou la lisibilité.

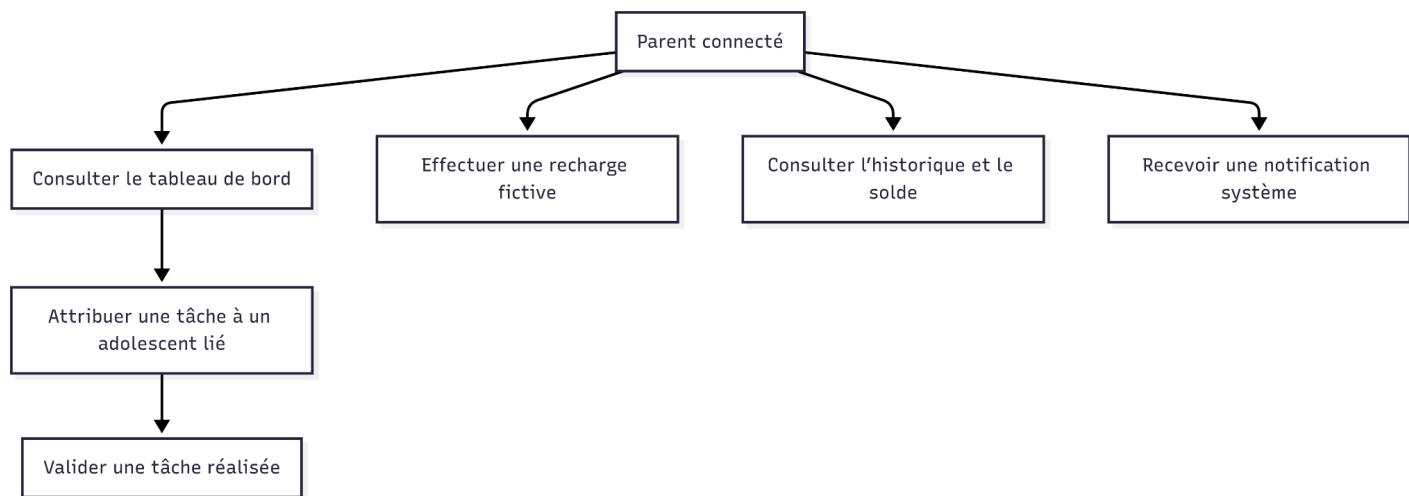
4. Spécifications fonctionnelles

4.1 Diagrammes de cas d'usage (parents/ados)

Les cas d'usage suivants modélisent les actions principales accessibles aux deux types d'utilisateurs : parent et adolescent.

Ils servent de base à la conception des écrans, à la définition des routes API, et à la logique métier.

Diagramme Parent :



Lecture du diagramme de cas d'usage – Parent

1. Connexion

- Le parent se connecte à son propre compte via une authentification sécurisée.
- Il accède à un dashboard personnalisé qui regroupe les informations principales des adolescents liés.

2. Gestion des tâches

- Le parent peut :
 - Créer une tâche (ex. : ranger sa chambre, lire un chapitre) ;
 - Définir une récompense en argent fictif ; - à enlever
 - Attribuer la tâche à un adolescent spécifique.

3. Suivi et validation

- Une fois la tâche réalisée par l'ado, elle passe en statut "à valider" ;
- Le parent peut vérifier la tâche et cliquer sur "valider" ;
- Selon la configuration, la récompense est versée automatiquement ou manuellement.

4. Rechargement de solde

- Le parent a la possibilité de verser une somme fictive sur le compte d'un adolescent à tout moment ;
- Il choisit le montant et le destinataire via une interface simple.

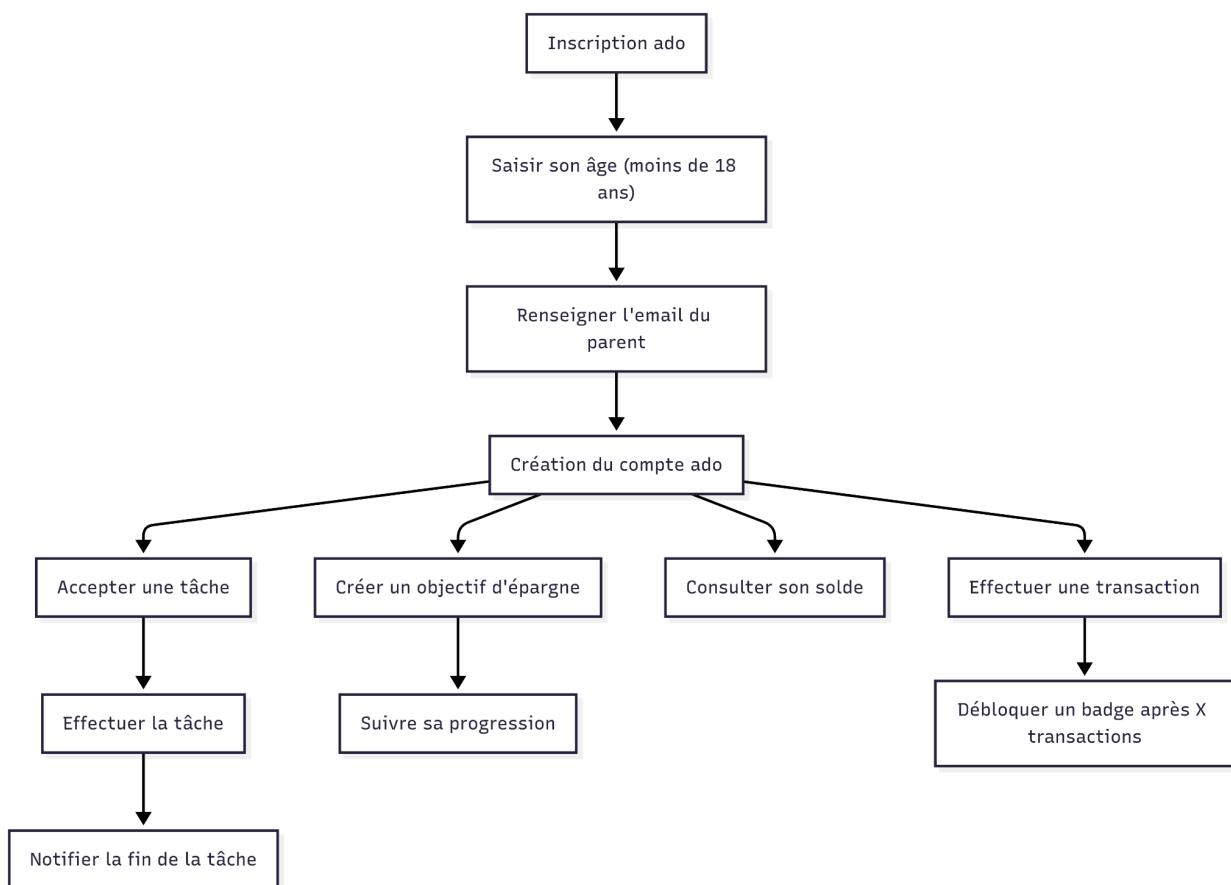
5. Suivi des comptes

- Il peut consulter l'historique complet des transactions de chaque ado ;
- Il a aussi accès à la progression des objectifs d'épargne, à l'état des tâches, et aux badges obtenus.

6. Notifications

- **Le parent reçoit des notifications push ou internes :**
 - **Lorsqu'un ado termine une tâche ;**
 - **Lorsqu'un objectif est atteint ;**
- En cas de comportement anormal (ex. : trop de dépenses d'un coup).**

Diagramme Adolescent



Lecture du diagramme de cas d'usage – Adolescent

1. Inscription

- L'utilisateur adolescent accède à l'application et initie une inscription.
- Il doit obligatoirement :
 - Indiquer qu'il a moins de 18 ans ;
 - Renseigner une adresse e-mail valide appartenant à un parent.
- Ces deux champs conditionnent l'accès à la suite de l'inscription.

2. Création du compte

- Une fois les informations renseignées, le compte est créé immédiatement.
- Le lien parent/ado est enregistré via l'e-mail fourni (aucune validation manuelle par le parent n'est nécessaire).

3. Utilisation post-inscription

- L'ado peut :
 - Accepter une tâche assignée par le parent ;
 - Réaliser la tâche selon les consignes ;
 - Notifier la fin de la tâche via l'interface.

4. Gestion de l'argent fictif

- L'adolescent peut consulter son solde en temps réel ;
- Il peut effectuer des transactions fictives (achats, transferts) ;
- Il a la possibilité de créer un objectif d'épargne, puis de suivre sa progression graphique.

5. Gamification

- Lorsqu'un certain nombre d'actions est atteint (par exemple, 5 transactions réussies), un badge est automatiquement attribué ;
- Le système de badges renforce l'engagement et valorise l'activité régulière.

4.2. Diagramme de classes UML

Le diagramme de classes permet de représenter de manière statique la structure des entités principales de l'application YouthWallet, ainsi que les relations entre elles. Il sert de base à la modélisation de la base de données et à l'organisation des modules backend. Il reflète également la séparation des responsabilités, essentielle à la maintenabilité de l'architecture logicielle.

Entités principales :

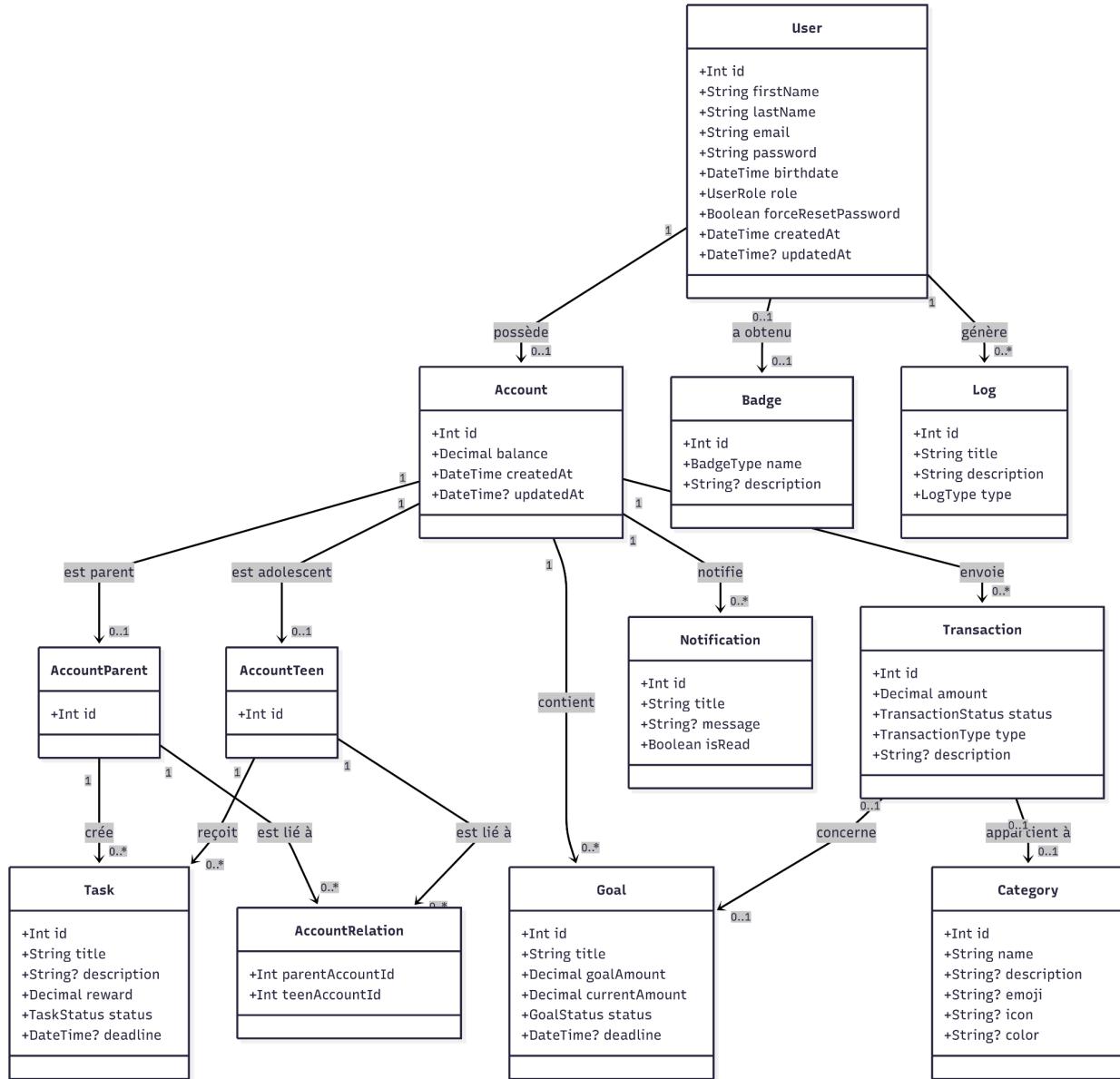
- **User** : représente tout utilisateur de l'application (parent ou adolescent). Il possède un compte (**Account**) unique. Il peut aussi être associé à un badge (**Badge**) valorisant ses actions, et génère des logs (**Log**) liés à son activité.
- **Account** : entité centrale représentant le portefeuille fictif de l'utilisateur. Il est lié à un **User**, et peut être spécialisé en **AccountParent** ou **AccountTeen** selon le rôle. Il détient un solde, un historique de transactions, des objectifs d'épargne (**Goal**) et des notifications (**Notification**).
- **AccountParent** : spécialisation d'un compte parent. Il peut être lié à plusieurs comptes adolescents via **AccountRelation**. Il est également responsable de la création de tâches (**Task**) qui seront assignées à un ado.
- **AccountTeen** : spécialisation d'un compte adolescent. Il peut être lié à plusieurs comptes parents via **AccountRelation**. Il reçoit et exécute des tâches, et gère ses transactions, objectifs et badges.
- **AccountRelation** : table de liaison N-N entre parents et adolescents, permettant à plusieurs parents de superviser plusieurs adolescents et inversement.

- **Transaction** : représente tout mouvement d'argent fictif entre comptes (recharge, dépense, récompense, remboursement, etc.). Chaque transaction peut être liée à un **Goal** et/ou une **Category**.
- **Task** : mission attribuée par un parent à un adolescent, associée à une récompense fictive. La tâche a un statut évolutif (en cours, terminée, expirée...) et une échéance éventuelle.
- **Goal** : objectif d'épargne défini par un adolescent. Il possède un montant cible et un suivi de progression automatique. Il peut être lié à des transactions.
- **Notification** : système d'alerte permettant d'informer l'utilisateur d'une action (ex : tâche validée, objectif atteint, transaction effectuée).
- **Category** : permet de catégoriser les transactions (alimentaire, loisirs, etc.) avec des couleurs, emojis, et icônes personnalisables.
- **Badge** : élément de gamification. Il est attribué à un utilisateur selon certains critères (première transaction, objectif atteint, etc.).
- **Log** : enregistre les événements techniques ou fonctionnels liés à un utilisateur (ex. : erreur, validation, opération).

Ce diagramme permet :

- de formaliser les structures de données essentielles ;
- de garantir l'intégrité référentielle via les relations entre entités ;
- de faciliter le développement backend avec Prisma ;
- et de fournir une documentation claire pour l'équipe de développement et les évaluateurs

(Cf voir-ci dessous)



4.3. Diagramme de séquence UML – Création d'une tâche

Le diagramme de séquence permet de représenter le déroulement dynamique d'un scénario fonctionnel en illustrant les interactions entre les différents composants du backend lors de la création d'une tâche (Task). C'est un outil essentiel pour visualiser le flux des requêtes et des réponses au sein de l'architecture modulaire de l'application.

Scénario choisi : création d'une tâche par un parent

Ce scénario commence par une action de l'utilisateur (parent) sur l'application mobile. L'objectif est de créer une tâche à attribuer à un compte adolescent lié, avec une récompense fictive.

Déroulement du processus :

1. Parent connecté (Frontend – Expo React Native)
 - Appuie sur le bouton "Créer une tâche" et remplit le formulaire : titre, description, montant, ado cible, date limite.
2. Requête HTTP POST
 - L'application envoie la requête vers le backend via une API REST `/task/new`.
3. Contrôleur NestJS (TaskController)
 - Reçoit la requête, extrait les données (DTO), déclenche le service associé.
4. Service Métier (TaskService)
 - Effectue les vérifications nécessaires (droits du parent, ado lié, format des données).
 - Crée une nouvelle entrée Task avec le statut initial PENDING.
5. ORM Prisma
 - Enregistre la tâche dans la base PostgreSQL via Prisma.
 - Lie la tâche au AccountParent et au AccountTeen concernés.

6. Réponse HTTP 201 Created

→ Retourne la tâche créée au frontend avec toutes ses informations.

7. Confirmation UI

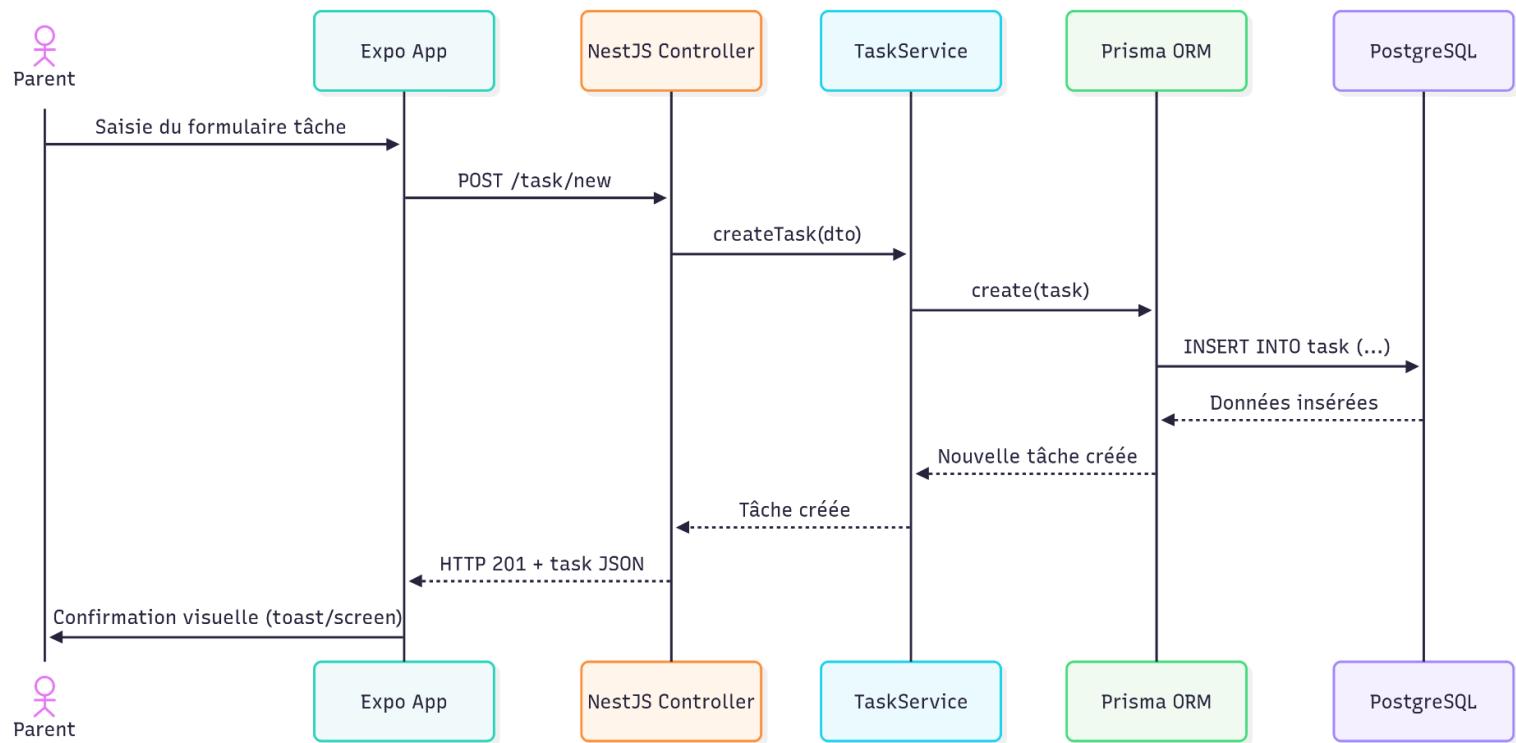
→ Le parent reçoit une confirmation visuelle que la tâche a bien été enregistrée.

Objectifs pédagogiques de ce diagramme :

- Montrer la circulation des responsabilités entre les différentes couches : contrôleur, service, modèle.
- Souligner l'usage des DTO, des services métiers et des relations Prisma dans un flux standardisé.
- Illustrer un exemple concret de scénario métier, utile lors de l'oral pour justifier la structure de l'application.

(CF voir ci-dessous)

Diagramme de séquence UML – Crédit d'une tâche



4.4 Maquettes UI (Figma)

1. UI Kit : Typographie, couleurs et composants

Le design repose sur une charte claire :

- **Typographie** : Inter (headers, body, labels), avec déclinaisons par poids et taille pour chaque niveau hiérarchique.
- **Couleurs** :
 - Primaires (CTA) : violet / bleu électrique
 - Secondaires : rouge, vert, violet secondaire
 - Textes : nuances de gris (du #1E1E1E au #A4A4A4)
- **Composants** :
 - Boutons (arrondis, contrastés)
 - Barres de navigation (icônes claires)
 - Sliders, modales, champs texte stylisés

(C.F Voir capture ci-dessous)

Typographie :

Frame 60842

Typography
361 x 66

Typography

Inter

Header 1 Semi-Bold	Family: Inter Weight: Semi-bold Size: 16px Letter Spacing: -1%	The face of the moon was in shadow.
Header 2 Semi-Bold	Family: Inter Weight: Semi-bold Size: 14px Letter Spacing: -1%	The face of the moon was in shadow.
Body 1 Regular	Family: Inter Weight: Regular Size: 14px Letter Spacing: -1%	The face of the moon was in shadow.
Body 2 Regular	Family: Inter Weight: Regular Size: 12px Letter Spacing: -1%	The face of the moon was in shadow.
Body 3 Semi-Bold	Family: Inter Weight: Semi-bold Size: 10px Letter Spacing: -1%	The face of the moon was in shadow.
Body Small - Regular	Family: Inter Weight: Regular Size: 9px Letter Spacing: -1%	The face of the moon was in shadow.
Body Small - Medium	Family: Inter Weight: Medium Size: 8px Letter Spacing: -1%	The face of the moon was in shadow.
Body Small - Semi-Bold	Family: Inter Weight: Semi-bold Size: 7px Letter Spacing: -1%	The face of the moon was in shadow.
Body Extra Small - regular	Family: Inter Weight: Regular Size: 6px Letter Spacing: -1%	The face of the moon was in shadow.

Frame 60861

Colours

Colours

Primary Colours
Primary colours for CTAs & active states

Accent Colours
Primary colours for CTAs & active states

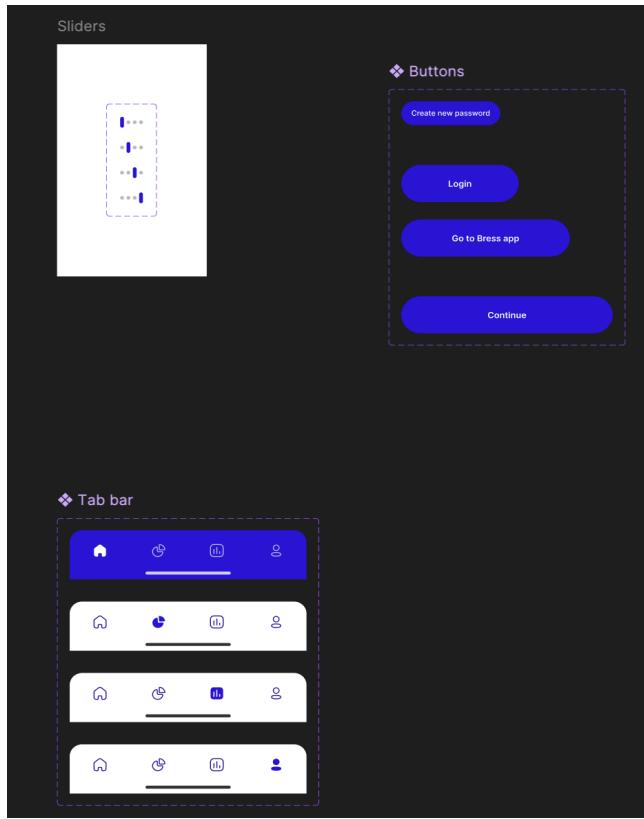
Text Colours
Text colours cover greys used for this project

2. Composants interactifs et navigation

Plusieurs composants réutilisables ont été conçus :

- **Boutons (CTA, secondaire)**
- **Sliders / switch**
- **Tab bar personnalisée avec 4 sections : Accueil, Objectifs, Tâches, Profil**

(Voir capture ci-dessous)



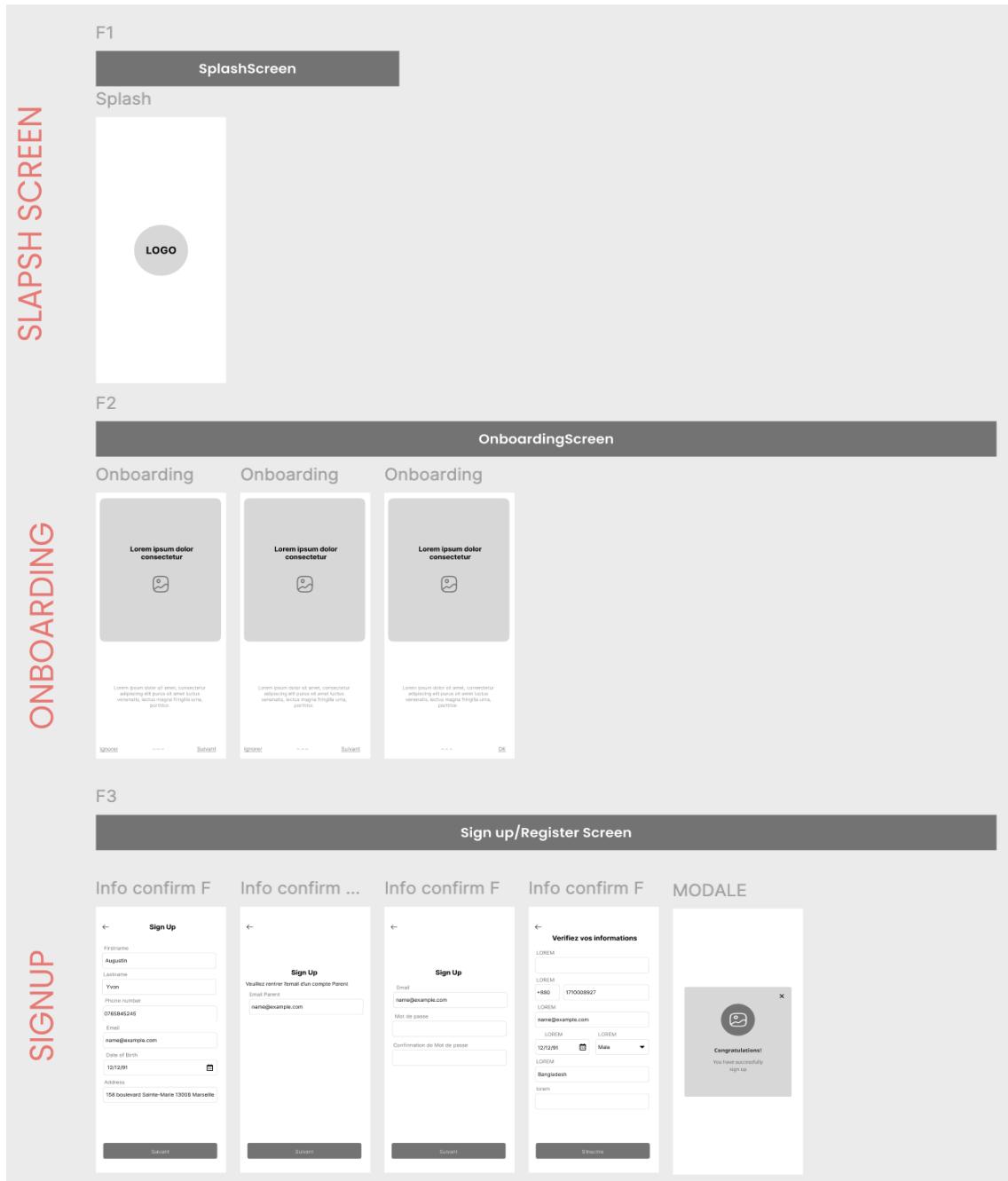
Wireframes basse fidélité

Les wireframes ont permis de structurer l'expérience utilisateur dès les premières étapes :

- **Splash screen**
- **Onboarding (3 écrans d'introduction)**
- **Inscription ado avec saisie de l'email parent**
- **Confirmation et modale de succès**

(Voir capture ci-dessous)

Wireframe basse-fidélité.



Wireframe de haute fidélité – ex : Dashboard utilisateur

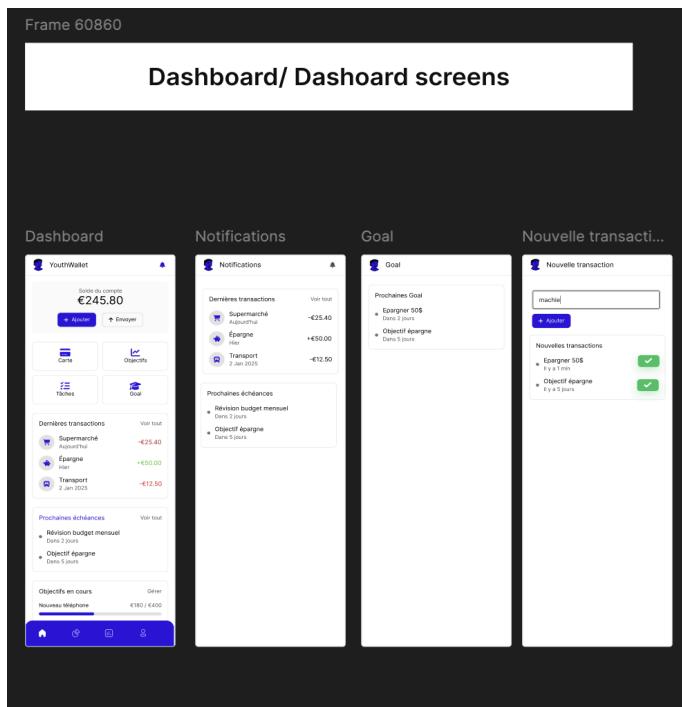
Les écrans fonctionnels incluent :

- Dashboard principal (solde, raccourcis, transactions)
- Écran de notifications
- Vue des objectifs
- Création d'une nouvelle transaction

Ces écrans sont pensés pour une navigation fluide, avec un affichage clair des soldes, échéances et objectifs.

(Voir capture ci-dessous)

WireFrame Haute Fidélité



Conclusion – Spécifications fonctionnelles

La phase de conception UI/UX a permis de structurer l'interface et l'expérience utilisateur de façon claire, intuitive et conforme aux standards mobiles.

Les maquettes ont guidé le développement des composants visuels et fonctionnels, en s'appuyant sur une charte cohérente et un système de navigation unifié.

Cette base graphique, combinée aux cas d'usage identifiés, garantit une implémentation fidèle aux besoins des utilisateurs parents et adolescents.

5. Réalisation technique de l'application

5.1 Conception des interfaces et navigation mobile (Expo Router, UX, RGAA, RGPD)

L'interface utilisateur de YouthWallet a été conçue pour répondre aux contraintes d'un usage mobile pédagogique, en combinant ergonomie, accessibilité et modularité. Le développement repose sur React Native via Expo, avec une architecture de navigation assurée par Expo Router.

Structure de navigation – Expo Router

- app/
- └── (auth)/ → Onboarding, Inscription (3 étapes), Login
- └── (cgu)/ → Conditions Générales d'Utilisation
- └── (tabs)/ → Espace connecté avec :
 - └── dashboard/ → solde, raccourcis, tâches, notifications
 - └── transactions/ → liste, création
 - └── goal/ → objectifs d'épargne
 - └── profile/ → badges, suppression de compte

Chaque onglet est intégré dans une **tab bar claire et accessible**.

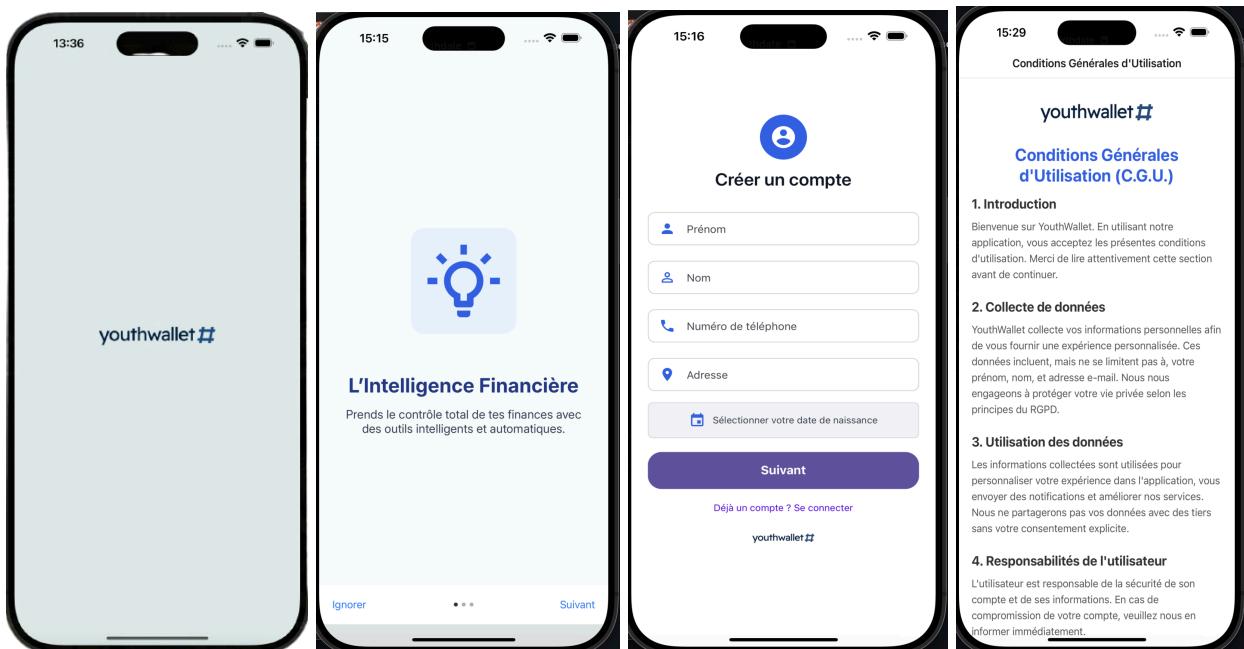
Des modales sont utilisées pour les écrans secondaires comme la recharge du compte ou la confirmation d'action

Ce découpage respecte la logique métier et facilite le développement de workflows séparés pour chaque rôle utilisateur.

5.1.2 Design system et maquettes (Figma)

Le design de YouthWallet a été entièrement réalisé dans Figma, avec un UI Kit complet (couleurs, typographie, composants, boutons, tab bar, etc.), des wireframes basse et haute fidélité, ainsi que des maquettes finales.

- UI Kit (typographie, couleurs, composants - **CF à voir ci-dessus section 4**)
- Splash screen, onboarding, dashboard ado, écran CGU, transaction
- L'approche Figma a permis de travailler en itération avec l'équipe sur une base cohérente entre le développement et l'expérience utilisateur.



CF : Exemples de certaines screen de l'applications.

5.1.3 Expérience utilisateur adaptée selon le rôle

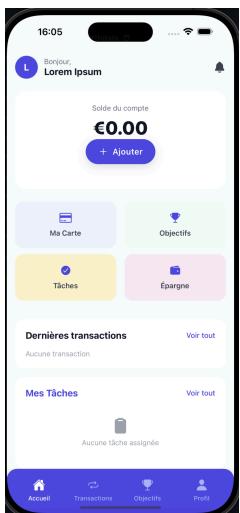
Les parcours utilisateurs sont clairement différenciés selon qu'il s'agisse d'un ado ou d'un parent.

Compte adolescent

- Interface simple, accessible et visuelle
- Solde en grand, navigation tabulaire épurée
- Accès rapide aux tâches et objectifs
- Icônes + libellés, peu de texte technique

Compte parent

- Vue synthétique de tous les comptes ados
- Possibilité de recharger un compte ou suivre les tâches
- Écrans adaptés à une gestion rapide



CF : ceci a été mis en place mais plus loin dans la conception

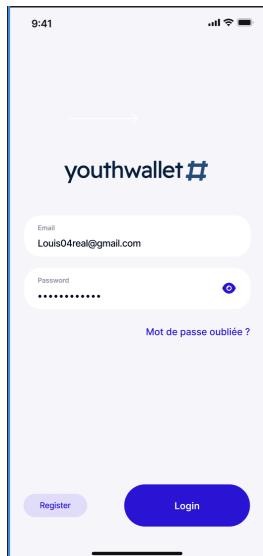
Cette logique de différenciation par rôle a guidé la création des routes et composants spécifiques dans app / (tabs) /

5.1.4 Respect des normes d'accessibilité – RGAA

Conformément aux recommandations du RGAA, plusieurs bonnes pratiques ont été appliquées dès la phase de maquette et confirmées lors du développement :

- Contraste suffisant entre fond et texte
- Hiérarchie visuelle claire (titres, textes secondaires)
- Navigation fluide et logique
- Icônes systématiquement accompagnées d'un label textuel
- Taille de police minimum respectée

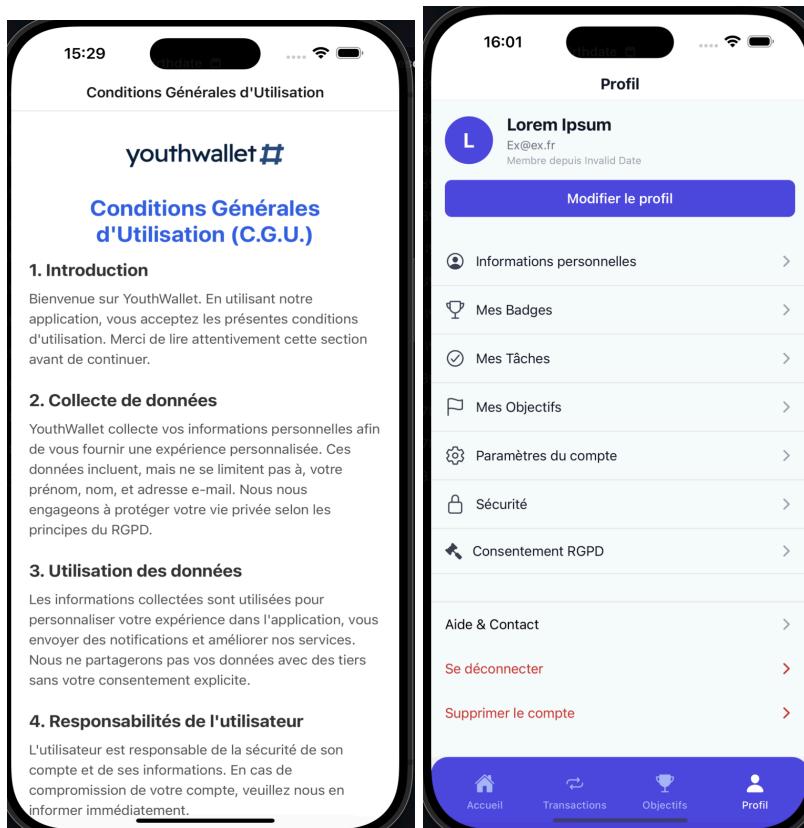
CF : maquette Figma haute fidélité (exemple : de connexion).



5.1.5 RGPD – Protection des données simulée

Même dans un contexte pédagogique fictif, YouthWallet simule un respect du Règlement Général sur la Protection des Données :

- Affichage des Conditions Générales d'Utilisation dès l'onboarding
- Collecte limitée à des données non sensibles (nom, e-mail, téléphone, âge)
- Aucune validation manuelle par le parent, mais enregistrement d'un e-mail parent obligatoire
- Authentification sécurisée par JWT + mot de passe chiffré (bcrypt)
- Présence d'un bouton "Supprimer mon compte" dans l'écran **profile**/



- Écran CGU / Écran de suppression de compte

5.1.6 Organisation technique du frontend

Le code est organisé selon une architecture modulaire claire, respectant les bonnes pratiques de séparation des responsabilités :

frontend/

```

├── app/           → toutes les routes
|   ├── (auth)/    → écrans publics
|   ├── (cgu)/     → CGU
|   └── (tabs)/    → espace utilisateur connecté
└── api/          → fichiers d'appel backend typés avec fetch
├── components/ui/ → composants réutilisables (Button, Modal...)
├── assets/        → images, icônes, polices
└── src/           → hooks, services, types, utils

```

Conclusion 5.1

La conception des interfaces et de la navigation dans YouthWallet répond à un double objectif:

- Offrir une expérience fluide et ludique pour les adolescents, tout en garantissant aux parents un outil de supervision efficace.
- Le découpage **Expo Router**, l'organisation du frontend, les maquettes Figma et les bonnes pratiques **RGAA/RGPD** forment un ensemble cohérent, propice à une utilisation pédagogique dans un contexte mobile moderne.

5.2 – Composants métier, structure backend et logique applicative

L'architecture backend de YouthWallet repose sur **le framework NestJS**, conçu pour faciliter le développement de serveurs Node.js robustes et maintenables grâce à une structure modulaire, fortement inspirée d'Angular.

Chaque domaine fonctionnel (utilisateurs, transactions, tâches, objectifs, etc.) est encapsulé dans un module NestJS dédié, **respectant les principes SOLID**, avec une séparation claire entre contrôleur, service, entités et objets de transfert de données (DTO).

5.2.1 Organisation d'un module métier

Un module dans NestJS est une unité logique indépendante, qui regroupe tous les éléments nécessaires à une fonctionnalité :

- ***.controller.ts** → gère les routes entrantes, les requêtes HTTP.
- ***.service.ts** → contient toute la logique métier.
- **dto/** → répertoire des Data Transfer Objects, objets typés servant à valider les données en entrée et sortie.
- ***.spec.ts** → fichiers de tests unitaires.
- **entities/ ou Prisma** : définition des modèles de données.

Un exemple avec Transactions : Arborescence VS Code du dossier

- **src/transaction**
- **dto/**,
- **controller.ts**,
- **service.ts**,
- **transaction.module.ts**

Exemple complet : le module Transaction

Le module Transaction incarne parfaitement les principes architecturaux de l'application. Il gère :

- la création de transactions (dépense, recharge, récompense de tâche, refund),
- l'association à une catégorie ou un objectif,
- le suivi de l'historique financier,
- et l'impact sur le solde utilisateur.

Structure de ce module : [Arborescence VS Code du dossier](#)

- **transaction.controller.ts**
Expose les routes REST : [/transaction/new](#), [/transaction/link/category](#), etc.
- **transaction.service.ts**
Vérifie les règles métier : solde disponible, statut de transaction, refund automatique...
- **transaction.module.ts**
Déclare les dépendances (PrismaService, controllers, services).
- **dto/*.ts**
Gère les règles de validation avec class-validator.

CF: voir ci-dessous capture de la méthode `create()` dans `transaction.service.ts`

Responsabilité principale

Créer une transaction dans l'application YouthWallet, avec gestion :

- de la validation métier (montants, présence de goal ou target),
- des effets de bord (mise à jour des soldes, logs, notifications),
- des badges.

- Bloc de validation métier

```
if (!targetAccountId && !goalId) throw new BadRequestException("Target or goal required");
if (amount <= 0) throw new BadRequestException("Amount must be > 0");

const sourceAccount = sourceAccountId ? await
this.accountService.findOneById(sourceAccountId) : null;
if (!sourceAccount && type !== TransactionType.DEPOSIT) throw new
NotFoundException("Source not found");

const targetAccount = targetAccountId ? await
this.accountService.findOneById(targetAccountId) : null;
if (targetAccountId && !targetAccount) throw new NotFoundException("Target not
found");

const goal = goalId ? await this.goalService.findOneById(goalId) : null;
if (goalId && !goal) throw new NotFoundException("Goal not found");
if (goal && goal accountId !== sourceAccountId) throw new BadRequestException("Goal
ownership mismatch");
```

- Création de la transaction

```
const transaction = await this.prisma.transaction.create({
  data: {
    amount,
    description,
    type: type || TransactionType.SPENDING,
    sourceAccount: sourceAccountId ? { connect: { id: sourceAccountId } } : undefined,
    targetAccount: targetAccountId ? { connect: { id: targetAccountId } } : undefined,
    goal: goalId ? { connect: { id: goalId } } : undefined,
    category: categoryId || goalId ? { connect: { id: categoryId || 10 } } : undefined,
  },
});
```

- LOG D'activité

```
await this.prisma.log.create({
  data: {
    title: "Nouvelle transaction",
    description: goalId
      ? `Virement de ${amount}€ vers "${goal.title}"`
      : `Transaction de ${amount}€ vers compte ${targetAccountId}`,
    type: LogType.INFO,
    user: sourceAccount ? { connect: { id: sourceAccount.userId } } : undefined,
  },
});
```

- Application de la transaction au solde

```
...  
  
if (type !== TransactionType.DEPOSIT) {  
    await this.accountService.applyTransactionToBalance(  
        sourceAccountId, targetAccountId, goalId, amount, transaction.id  
    );  
}
```

- Attribution de badge

```
...  
  
const count = await this.prisma.transaction.count({ where: { sourceAccountId } });  
if ([1, 10].includes(count)) {  
    const badge = count === 1 ? BadgeType.PREMIERE TRANSACTION :  
    BadgeType.GESTIONNAIRE_CONFIRME;  
    await this.badgeService.assignBadge({ id: sourceAccount.userId, name: badge });  
}
```

- **Notification (si reçu)**

```
...  
  
if (transaction.status === "COMPLETED" && transaction.targetAccountId) {  
  await this.notificationService.create({  
    accountId: transaction.targetAccountId,  
    title: "Transaction reçue",  
    message: `Vous avez reçu ${transaction.amount}€ sur votre compte.`,  
  });  
}
```

- **Gestion des erreurs**

```
...  
  
catch (error) {  
  console.error("✖ Error creating transaction:", error.message);  
  throw new InternalServerErrorException("Erreur pendant la création");  
}
```

- Capture de la route POST /transaction/new dans transaction.controller.ts

```
...  
  
catch (error) {  
    console.error("✖ Error creating transaction:", error.message);  
    throw new InternalServerErrorException("Erreur pendant la création");  
}
```

5.2.2 Prisma ORM & schema.prisma

Le backend de YouthWallet s'appuie sur Prisma ORM pour modéliser et interagir avec la base PostgreSQL. Tous les modèles (**User**, **Account**, **Transaction**, etc.) sont centralisés dans le fichier **schema.prisma**. Ce schéma définit les types, relations, contraintes (@relation, @default, @map, etc.), et génère un client TypeScript sécurisé. Grâce à cela, chaque requête (ex. **prisma.transaction.create**) est typée, fiable et maintenable.

Les relations entre entités sont explicites : un compte peut être lié à un ado ou à un parent, une transaction peut viser un objectif, être catégorisée ou transférée d'un compte à un autre. Prisma gère également les migrations de base et la visualisation via Prisma Studio.

Le backend s'appuie sur Prisma, un ORM moderne générant automatiquement :

- des entités typées (TypeScript),
- les relations entre modèles (@relation),
- un client sécurisé (prisma.<model>.create() / update() / findMany()...).

Le fichier **schema.prisma** contient l'ensemble des modèles :

User, **Account**, **Transaction**, **Goal**, **Task**, **Category**, etc.

Les relations sont explicites (1-1, 1-N, N-N) avec des stratégies de suppression (onDelete: Cascade ou SetNull).

- Schéma Prisma : schema.prisma

```

model Transaction {
    id      Int          @id @default(autoincrement())
    amount  Decimal
    status  TransactionStatus @default(PENDING)
    type    TransactionType
    ...
    sourceAccountId Int?
    sourceAccount   Account? @relation("SourceTransactions", fields:
        [sourceAccountId], references: [id], onDelete: SetNull)
}

```

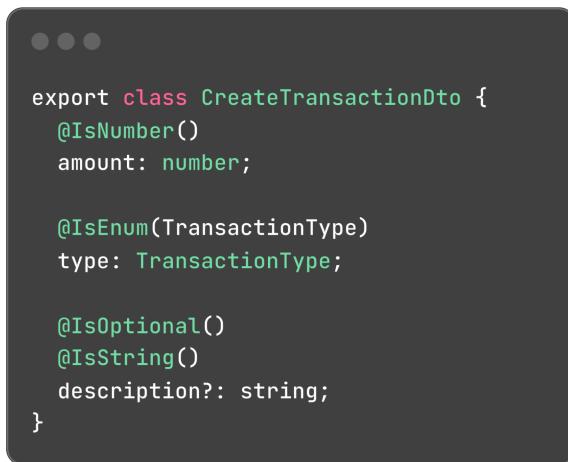
- Une vue de Prisma Studio (relations visibles entre les entités)

C	Filters	None	Fields	All	Showing	1 of 1	Add record									
								id #	amount #	status	createdAt	updatedAt	description A?	sourceAccountId #?	sourceAccount {}?	targetAc
								1	100	COMPLETED	2025-07-31T13:24:52.3...	2025-07-31T13:24:52.3...	manuel	null	Account	1
								id	balance	createdAt	updatedAt	userId	user	parentAccount	teenAccount	
								search id...	search balance...	search createdAt...	search updatedAt...	search userId...	User	AccountParent	AccountTeen	
								1	100	2025-07-31T13:23:43.6...	2025-07-31T13:24:52.3...	1	User	AccountParent	AccountTeen	
								2	0	2025-07-31T13:31:13.0...	2025-07-31T13:31:13.0...	2	User	AccountParent	AccountTeen	

[Open in new tab](#)

5.2.3 Les DTO (Data Transfer Objects)

Les DTO (Data Transfer Objects) sont des classes utilisées pour valider les données entrantes dans les requêtes HTTP. Dans YouthWallet, chaque opération critique (création de transaction, liaison à une catégorie, etc.) dispose de son DTO, décoré avec des validateurs **class-validator** (`@IsNumber`, `@IsString`, `@IsEnum`, etc.). Cela permet d'intercepter les erreurs avant même d'exécuter la logique métier.



```
export class CreateTransactionDto {
  @IsNumber()
  amount: number;

  @IsEnum(TransactionType)
  type: TransactionType;

  @IsOptional()
  @IsString()
  description?: string;
}
```

Utilisés dans les contrôleurs, ils garantissent une validation automatique et sécurisée des données. En cas de non-respect, une erreur 400 est renvoyée sans même exécuter la logique métier.

Par exemple, `CreateTransactionDto` valide que le montant est bien un nombre, que le type est conforme à l'enum `TransactionType`, et que la description est facultative mais bien une chaîne.

- Capture de **create-transaction.dto.ts** (**snippet**)

```
● ● ●

@IsNumber()
@Transform(({ value }) => Number(value))
@Min(0.01, { message: 'Minimum amount must be 0.01' })
amount: number;

@IsString()
@IsOptional()
description?: string;

@IsNumber()
@IsOptional()
@Transform(({ value }) => Number(value))
sourceAccountId?: number;
```

- Capture de **link-category.dto.ts**

```
● ● ●

export class LinkCategoryDto {
    @Type(() => Number)
    @IsInt()
    transactionId: number;

    @Type(() => Number)
    @IsInt()
    categoryId: number;
}
```

5.2.4 Injection de dépendances (Dependency Injection)

NestJS repose sur un système d'injection de dépendances intégré. Cela permet :

L'injection de dépendances (DI) est assurée par NestJS via les décorateurs `@Injectable()` et les constructeurs. Cela permet d'injecter le `PrismaService` dans n'importe quel service métier, comme ici dans `TransactionService`, sans avoir à instancier manuellement quoi que ce soit. Cela respecte le principe d'inversion des dépendances (D du SOLID) et rend le code plus modulaire et testable.

- de ne pas instancier manuellement les services,
- de tester facilement chaque module en isolant ses dépendances,
- de respecter le principe D (Dependency Inversion) de SOLID.

Exemple d'injection de dépendance dans le cadre de `TransactionService`

```
...  
@Injectable()  
export class TransactionService {  
  constructor(private readonly prisma: PrismaService) {}  
}
```

5.2.5 Principe SOLID

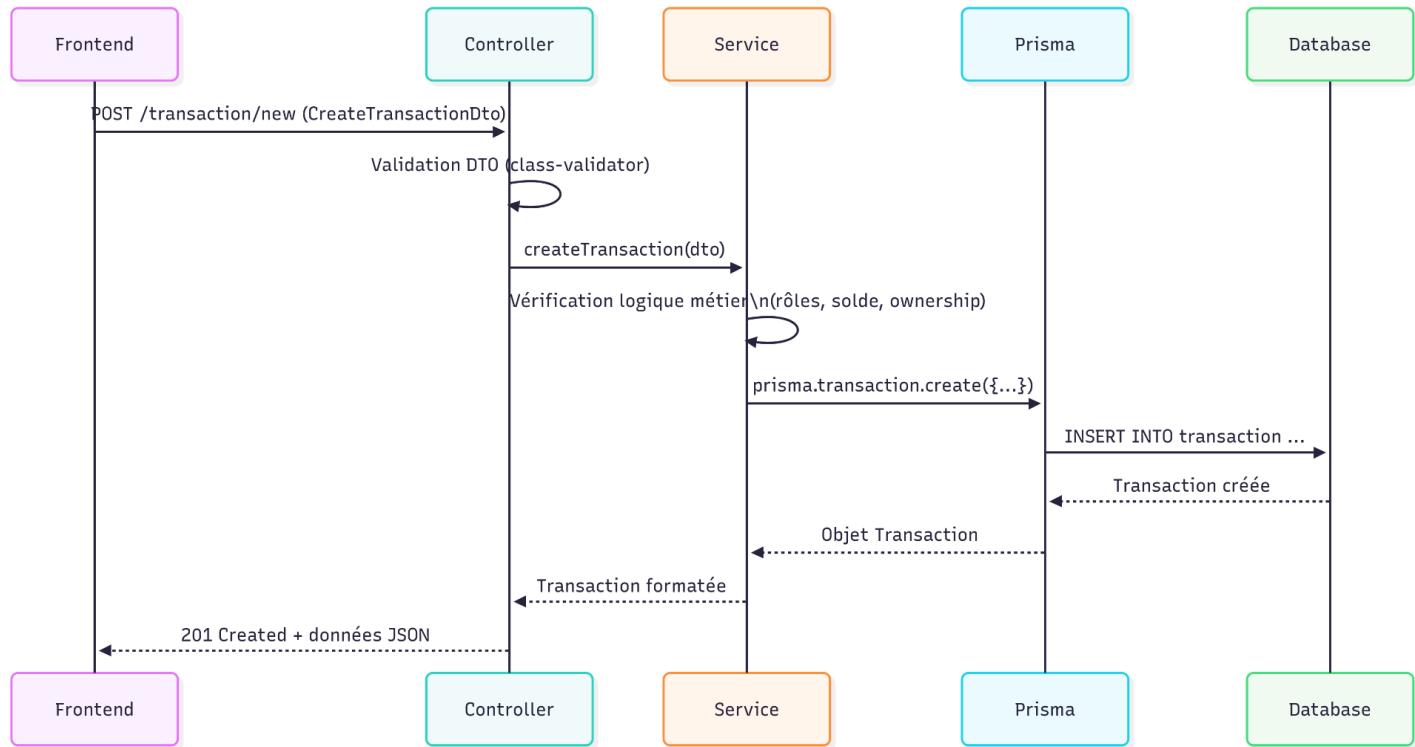
Le projet YouthWallet applique rigoureusement les principes SOLID dans la conception de ses modules backend.

Chaque classe a un rôle précis (Single Responsibility), par exemple le **TransactionService** s'occupe exclusivement de la logique liée aux transactions. Les fonctionnalités sont extensibles sans modifier l'existant (Open/Closed), comme l'ajout d'un nouveau type de transaction. Les services ou DTO peuvent être remplacés sans altérer les appels (Liskov Substitution). Chaque action dispose de ses propres DTO, spécifiques à leur besoin (Interface Segregation). Enfin, les dépendances comme **PrismaService** ou **AuthService** sont injectées dans les services pour respecter l'inversion des dépendances (Dependency Inversion).

La logique métier suit un enchaînement clair lors d'une requête typique, comme la création d'une transaction :

1. Le contrôleur (`transaction.controller.ts`) reçoit une requête POST `/transaction/new` contenant un objet `CreateTransactionDto`.
2. NestJS valide les données entrantes via `class-validator`. Si les contraintes ne sont pas respectées, une erreur 400 est renvoyée.
3. Le service vérifie les rôles, la validité du solde, et applique la logique métier (débit/crédit).
4. Prisma est ensuite utilisé pour créer l'entrée en base, de manière typée et sécurisée (`prisma.transaction.create()`).
5. La réponse formatée est renvoyée au frontend, avec les détails de la transaction enregistrée.

- Diagramme de séquence UML en Mermaid : controller → service → prisma → base



- Capture d'un test Postman montrant une transaction créée avec succès

The screenshot shows a POST request to `{baseUrl}/transaction/create`. The request body is a JSON object:

```
1 {  
2   "amount": 100,  
3   "description": "Faut les sousou",  
4   "sourceAccountId": 1,  
5   "targetAccountId": 2  
6 }
```

The response status is 201 Created, with a response time of 109 ms and a response size of 513 B. The response body is:

```
1 {  
2   "id": 3,  
3   "amount": "100",  
4   "status": "COMPLETED",  
5   "createdAt": "2025-07-31T17:14:58.344Z",  
6   "updatedAt": "2025-07-31T17:14:58.384Z",  
7   "description": "Faut les sousou",  
8   "sourceAccountId": 1,  
9   "targetAccountId": 2,  
10  "goalId": null,  
11  "categoryId": null,  
12  "type": "SPENDING"  
13 }
```

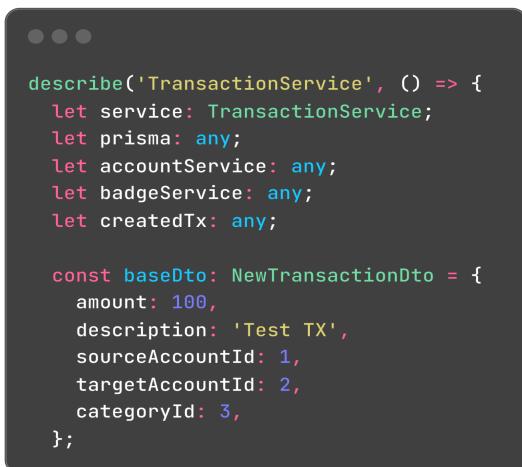
5.2.6 Tests Jest & Postman

Les tests jouent un rôle fondamental pour fiabiliser l'application.

Chaque couche métier est couverte par des tests automatisés via Jest. Le fichier **transaction.service.spec.ts** simule plusieurs cas : création simple, solde insuffisant, remboursement d'objectif, etc. Le fichier **dto.spec.ts** teste les règles de validation (IsNumber, IsEnum, etc.). Enfin, **transaction.controller.spec.ts** simule des appels réels à l'API avec Supertest et vérifie les réponses.

Des tests fonctionnels ont également été réalisés via Postman : rechargement de solde, création de transaction, refund automatique d'un objectif abandonné, création d'objectif, etc. Une collection Postman JSON a été utilisée en local avec authentification JWT (bearer token dans les headers).

- Un test **describe()** dans **transaction.service.spec.ts** (snippet)



```
describe('TransactionService', () => {
  let service: TransactionService;
  let prisma: any;
  let accountService: any;
  let badgeService: any;
  let createdTx: any;

  const baseDto: NewTransactionDto = {
    amount: 100,
    description: 'Test TX',
    sourceAccountId: 1,
    targetAccountId: 2,
    categoryId: 3,
  };
}
```

- Une assertion expect(...) vérifiant un comportement métier

```
describe('TransactionController', () => {
  let controller: TransactionController;
  let service: { newTransaction: jest.Mock; findAll: jest.Mock };

  beforeEach(async () => {
    service = {
      newTransaction: jest.fn(),
      findAll: jest.fn(),
    } as any;

    const module: TestingModule = await Test.createTestingModule({
      controllers: [TransactionController],
      providers: [{ provide: TransactionService, useValue: service }],
    }).compile();

    controller = module.get<TransactionController>(TransactionController);
  });

  it('should be defined', () => {
    expect(controller).toBeDefined();
  });
}
```

Conclusion – 5.2 Composants métier et logique backend

L'architecture backend de YouthWallet repose sur une base solide, conçue pour assurer la robustesse, la maintenabilité et la scalabilité du projet. En structurant chaque domaine métier dans un module dédié selon les principes SOLID, l'équipe a pu garantir une séparation claire des responsabilités entre contrôleurs, services, DTO, entités et tests.

L'utilisation de NestJS couplée à Prisma ORM permet un accès sécurisé, typé et performant aux données, tout en respectant les bonnes pratiques de validation et de gestion des erreurs. Le recours à l'injection de dépendances favorise un code modulaire, facilement testable et prêt à évoluer.

Cette organisation, alliée à une politique de tests rigoureuse (Jest, Postman), pose les fondations d'une application fiable, avec une logique métier claire et centralisée. Elle facilite également la collaboration entre les développeurs frontend et backend grâce à une API documentée, typée et prévisible.

5.3 – Base de données relationnelle (PostgreSQL + Prisma)

La base de données du projet YouthWallet repose sur PostgreSQL, un SGBD robuste et open source, parfaitement adapté aux applications nécessitant des relations complexes, des contraintes d'intégrité fortes et des opérations financières fiables.

Choix techniques

Le choix de PostgreSQL s'explique par plusieurs avantages :

- Fiabilité et cohérence transactionnelle, essentielles pour une app budgétaire simulant des mouvements d'argent.
- Support des types avancés : **Decimal**, **Enum**, **Array**, très utiles pour gérer les statuts, rôles, montants précis, etc.
- Facilité de conteneurisation avec Docker, assurant un environnement reproductible.

Le modèle de données est géré via Prisma ORM, qui offre un typage fort en TypeScript, une génération automatique des requêtes, des migrations structurées, et une interface visuelle (Prisma Studio) pour manipuler les données

5.3 – Base de données relationnelle (PostgreSQL + Prisma)

L'architecture de données de YouthWallet repose sur une base relationnelle PostgreSQL. Elle a été modélisée selon une démarche classique en trois étapes : MCD, MLD puis MPD. Le mapping final a ensuite été implémenté via l'ORM Prisma, qui génère un client TypeScript typé et maintenable.

MCD – Modèle conceptuel de données

Le MCD représente les grandes entités fonctionnelles du système, leurs attributs et leurs relations :

- **User, Account, Transaction, Task, Goal, Badge, Category, Notification, etc.**
- Les rôles (**PARENT, TEEN**) sont différenciés à travers deux entités spécialisées : **AccountParent** et **AccountTeen**.
- Une relation N:N entre parents et adolescents est gérée via la table associative **AccountRelation**.

 [Voir MCD.drawio en annexe.](#)

MLD – Modèle logique de données

Le MLD traduit le MCD en un schéma conforme aux contraintes relationnelles :

- Ajout des clés primaires et étrangères explicites.
- Définition des cardinalités (1:N, N:N).
- Précision des types de données.
- Crédation de tables techniques pour les relations complexes (**AccountRelation, Transaction, Task, etc.**).

 [Voir MLD.drawio en annexe.](#)

MPD – Modèle physique de données

Le MPD adapte le MLD à PostgreSQL en tenant compte des optimisations techniques :

- Types spécifiques PostgreSQL (ex. `Decimal`, `DateTime`)
- Stratégies de suppression (`onDelete: CASCADE` ou `SET NULL`)
- Index sur les clés étrangères et champs uniques (email, userId...)

 [Voir MPD.drawio en annexe.](#)

Prisma ORM – Mapping de la base

La structure finale est codée dans le fichier `schema.prisma`, qui contient :

- Les modèles (`model`) : chaque entité métier y est définie avec ses relations, contraintes et types.
- Les énumérations (`enum`) : pour les rôles (`UserRole`), types de transaction (`TransactionType`), statuts, etc.
- Les relations : un `Goal` appartient à un `Account`, une `Transaction` est liée à une `Category`, etc.
- Les stratégies de suppression : certaines relations utilisent `onDelete: Cascade` pour supprimer automatiquement les dépendances.

Extrait du modèle Transaction :

```
model Transaction {  
    id      Int      @id @default(autoincrement())  
    amount  Decimal  
    status  TransactionStatus @default(PENDING)  
    type    TransactionType  
    goalId  Int?  
    goal    Goal?     @relation("GoalTransactions", fields: [goalId], references: [id])  
    ...  
}
```

Prisma génère ensuite un client typé (**PrismaClient**) utilisé dans les services pour interagir avec la base de manière sécurisée et performante.

Outils complémentaires

- **Prisma Migrate** : gère les migrations (**npx prisma migrate dev**) et la synchronisation avec PostgreSQL.
- **Prisma Studio** : visualisation graphique de la base (comparable à phpMyAdmin).
- **Prisma Seed** : permet d'injecter des données de test (parents, ados, badges...) via **npx prisma db seed**.

5.4 – Développement des composants d'accès aux données SQL et NoSQL

Accès aux données SQL avec Prisma (YouthWallet)

L'application YouthWallet utilise PostgreSQL comme base de données relationnelle et Prisma ORM pour l'accès aux données. Prisma agit comme une couche d'abstraction typée entre le code TypeScript (NestJS) et la base de données. L'interface `prisma.transaction.create()` permet de créer une transaction sans écrire une seule ligne de SQL brut.

Le fichier `schema.prisma` est central : il définit tous les modèles (User, Account, Transaction, etc.), leurs champs, types, relations, contraintes (unicité, `@default`, etc.), ainsi que les énumérations comme `TransactionType` ou `GoalStatus`.

Extrait typique :

```
model Account {
    id      Int      @id @default(autoincrement())
    balance Decimal  @default(0.00)
    userId  Int      @unique
    user    User     @relation(fields: [userId], references: [id])
}
```

Grâce à Prisma Client, les requêtes deviennent typées, sûres, et vérifiées à la compilation :

```
...  
await this.prisma.account.findUnique({  
  where: { userId: user.id },  
});
```

Les composants d'accès aux données sont centralisés dans des services NestJS
(`transaction.service.ts`, `goal.service.ts`, etc.) où Prisma est injecté comme dépendance :

```
...  
@Injectable()  
export class TransactionService {  
  constructor(private readonly prisma: PrismaService) {}  
}
```

Prisma gère également :

- la migration de schéma (`npx prisma migrate dev`)
- le seed de données
- Prisma Studio pour la visualisation

6. Gestion de projet

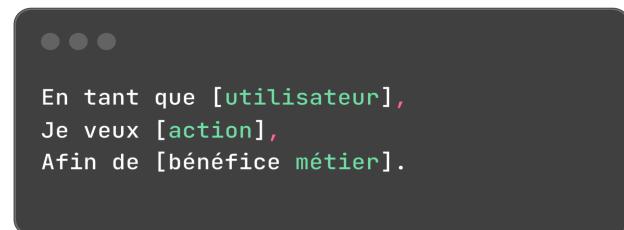
Cette section montre comment YouthWallet a été conduit de façon professionnelle, en respectant les bonnes pratiques de gestion agile, les outils de suivi, et l'organisation de l'équipe

Méthodologie projet – Kanban agile

Le projet a suivi une méthode Agile Kanban adaptée à la durée et à la charge de travail. Chaque membre de l'équipe (backend, frontend, fullstack) avait des tâches claires assignées sur Trello. Le Kanban comportait les colonnes : **À faire, En cours, En attente de test, Fait.**

Le découpage s'est fait par fonctionnalité métier (transaction, objectifs, tâches, etc.) avec des livraisons régulières et des points de synchronisation quotidiens (daily stand-up simulé).

Les user stories étaient structurées de cette manière :



Exemple :

En tant que parent, je veux attribuer une tâche à mon enfant pour l'aider à apprendre à gérer son argent

Outils utilisés

- Trello : suivi des tâches (capture écran à insérer) / GOOGLE MEET
- GitHub + GitFlow : branche **main**, **dev**, branches fonctionnelles (**feature/**, **fix/**, etc.)

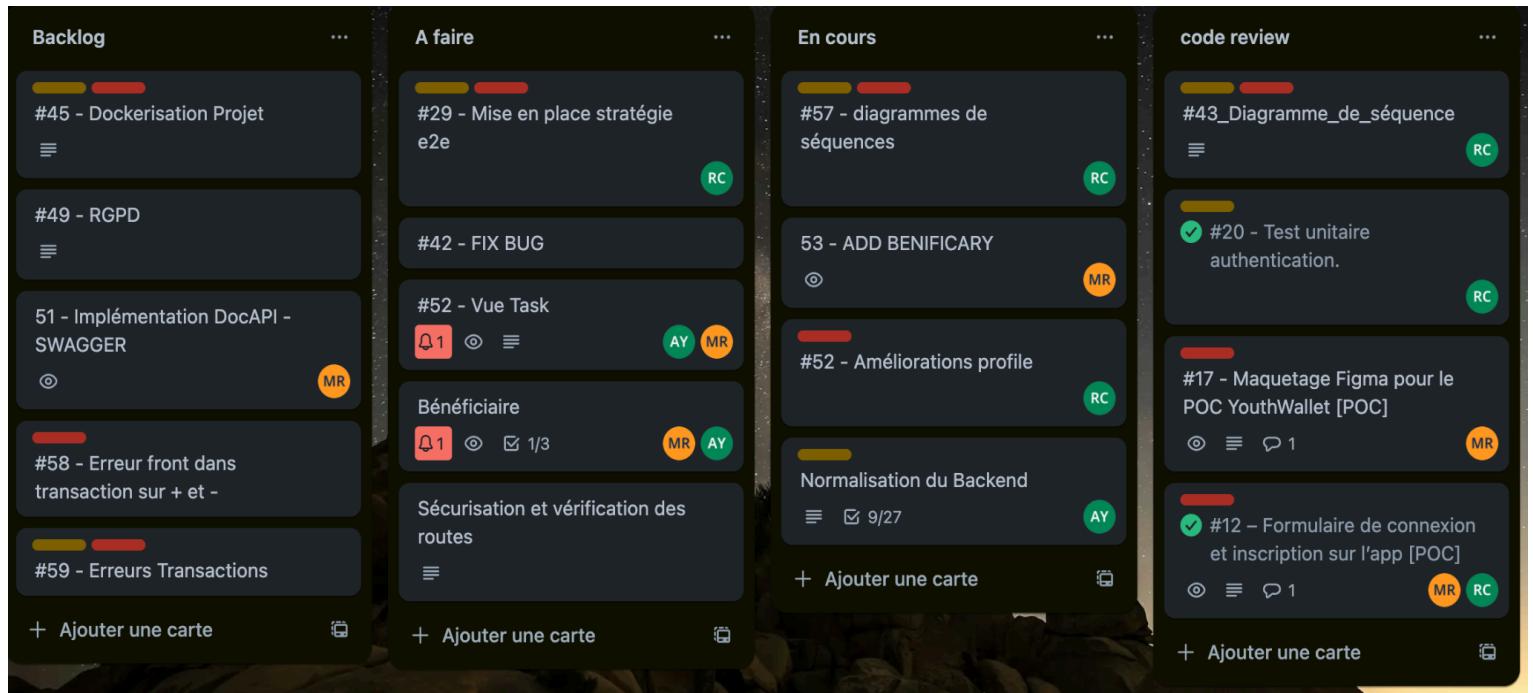
Répartition de l'équipe

Le projet a été réalisé en équipe avec répartition progressive des responsabilités :

- Augustin Yvon : Backend, structure NestJS, base PostgreSQL, logique métier
- Romain Chazaut : Backend et Frontend (intégration, Expo Router, tests, CI/CD)
- Mehdi Romdhani (auteur) : Frontend principal (Expo, navigation, maquettes Figma, logique RGAA), gestion des badges, contribution backend.

Malgré cette séparation, chacun a été amené à travailler en fullstack pour comprendre les flux de bout en bout. (CF: voir ci-dessous les screenshots)

- Tableau Trello - Tickets



- Extract FORK (Client GIT)

The screenshot shows a Git commit history for the 'YouthWallet' repository. The main branch is 'dev'. A merge branch, 'feat/#59_SUITE', has been merged into 'dev'. The commit history includes:

- Merge remote-tracking branch 'origin/feat/#56' into dev
- origin/feat/#56 feat: test CI
- Merge branch 'feat/backend-standardization' into dev
- feat/#58 feat/backend-standardization all routes secured
- feat(task): secure task routes and restrict create, update, complete to non-teen roles
- feat(auth): add global AuthMiddleware to validate JWT and inject req.user
- propagate multi-role support across app
- allow multiple roles per user by changing role to UserRole[]
- add TEEN and PARENT roles to UserRole enum and run migration
- remove duplicate guards and update imports in AdminController
- origin/feat/#57_diagram feat: ajout diagram séquence
- Merge branch 'feat/#54' into dev
- origin/feat/#54 fix amount in deposit
- fix deposit and resolve migration conflict
- feat/#56 Fix DOCKER .env
- Add README_CI
- ADD worflow CI for PR for test
- ADD CI & DockerCompose, DockerFile
- origin/feat/#52-Améliorations-profile feat: wip
- Merge branch 'feat/backend-standardization' into dev
- fix
- feat/#54 Add feature for branch 54
- fix deposit
- Create Api function for index/create transactions for frontend

7. Tests et validation de la logique métier

L'application YouthWallet a été testée de manière rigoureuse grâce à Jest pour les tests unitaires et à Postman pour les tests fonctionnels de l'API.

Côté unitaires, chaque module métier (transactions, objectifs, tâches) possède ses fichiers **.spec.ts**, permettant de valider :

- la logique métier (ex. : vérification du solde, refund d'objectif),
- les DTO avec **class-validator**,
- les réponses attendues des contrôleurs avec Supertest.

Les services sont testés indépendamment grâce au mock de PrismaService, garantissant l'isolation et la stabilité de chaque composant.

Côté fonctionnel, une collection Postman a permis de tester manuellement toutes les routes :

- authentification JWT,
- création et liaison de transaction à une catégorie,
- refund automatique,
- rechargement de compte,
- tests des rôles (Parent / Teen).

Ces tests ont facilité l'intégration avec le frontend et mis en évidence les erreurs éventuelles.

Un jeu d'essai réaliste a été appliqué avec :

- création de comptes,
- plusieurs transactions,
- abandon d'objectif,
- attribution de badge automatique après 5 transactions.

- **Test Jest (extrait `transaction.service.spec.ts`)**

```
describe('newTransaction - flux standard (avec targetAccountId)', () => {
  it('crée la transaction, met à jour les soldes, sans badge', async () => {
    const result = await service.newTransaction(baseDto);
    expect(prisma.transaction.create).toHaveBeenCalledWith({
      data: {
        amount: baseDto.amount,
        description: baseDto.description,
        sourceAccount: { connect: { id: baseDto.sourceAccountId } },
        targetAccount: { connect: { id: baseDto.targetAccountId } },
        goal: undefined,
        category: { connect: { id: baseDto.categoryId } },
      },
    });
    expect(accountService.applyTransactionToBalance).toHaveBeenCalledWith(
      baseDto.sourceAccountId,
      baseDto.targetAccountId,
      null,
      baseDto.amount,
      createdTx.id,
    );
    expect(badgeService.assignBadge).not.toHaveBeenCalled();
    expect(result).toEqual(createdTx);
  });
});
```

8. Déploiement & DevOps

Le projet YouthWallet intègre une démarche DevOps simplifiée mais complète, permettant un environnement de développement stable, reproductible et automatisé pour toute l'équipe.

Conteneurisation avec Docker

Tous les services (backend NestJS, base de données PostgreSQL, pgAdmin) sont conteneurisés à l'aide de Docker. Le fichier **docker-compose.yml** orchestre les services, les variables d'environnement et les volumes de données.

Chaque composant dispose de son propre **Dockerfile**, permettant un build indépendant :

- **backend/Dockerfile** → installation de NestJS, Prisma, et exécution du serveur HTTP
- **frontend/Dockerfile** → build du projet Expo en mode dev ou prod

Cette conteneurisation garantit un fonctionnement identique sur tous les postes et facilite le passage en production.

- Extrait du
docker-compose.yml

```
...
postgres:
  image: postgres:17
  container_name: postgres_dev
  environment:
    POSTGRES_USER: ${POSTGRES_USER}
    POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    POSTGRES_DB: ${POSTGRES_DB}
  ports:
    - "5432:5432"
  volumes:
    - postgres_data:/var/lib/postgresql/data
  healthcheck:
```

- Extrait DockerFILE /backend

```
FROM node:20-alpine
WORKDIR /app
# Installer git
RUN apk add --no-cache git
# Copier les fichiers package
COPY package*.json ./
# Installer les dépendances
RUN npm install
# Copier le schéma Prisma
COPY prisma ./prisma/
# Générer le client Prisma
RUN npx prisma generate
EXPOSE 3000
CMD ["npm", "run", "start:dev"]
```

Intégration continue avec GitHub Actions

Un pipeline CI a été mis en place pour automatiser les étapes clés :

- exécution des tests Jest à chaque push,
- vérification du linting,
- message de notification sur Google Chat (optionnel).

Des workflows spécifiques sont déclenchés à chaque push, pull request, ou fusion vers dev.

- Extrait workflow github action CI

```
name: Vérification PR vers dev

on:
  pull_request:
    branches:
      - dev
```

Lancement local et documentation

Un **Makefile** simplifie les commandes : **make start**, **make stop**, **make db-reset**, etc.

L'environnement complet peut être lancé en une seule commande

- CMD docker-compose

```
... ...
```

```
docker-compose up --build
```

9. Veille technologique et UX

Tout au long du projet, une veille active a été menée sur plusieurs thématiques essentielles pour garantir un haut niveau de qualité, de sécurité et d'expérience utilisateur. Cela a permis de faire des choix techniques éclairés et d'adopter les bonnes pratiques recommandées par la communauté.

Sécurité backend (NestJS, JWT, Prisma)

Des recherches ont été effectuées sur les meilleures pratiques en matière d'authentification sécurisée avec JWT, la gestion des rôles, et la protection des routes sensibles via les Guards dans NestJS. L'équipe a également exploré les stratégies de validation des données via les DTO et les décorateurs **class-validator**, ainsi que les protections contre les injections SQL via Prisma ORM.

ORM Prisma & modélisation

Une veille spécifique a été menée sur Prisma, afin de maîtriser :

- la modélisation relationnelle complexe (1-1, 1-N, N-N),
- la génération des types TypeScript,
- les migrations et le versioning de la base de données,
- l'utilisation avancée de Prisma Studio pour tester les requêtes.

Gamification & UX mobile

L'ajout de badges et d'objectifs à atteindre s'est inspiré de solutions comme Pixipay ou GoHenry. La gamification a été utilisée comme levier pédagogique pour l'apprentissage budgétaire. Une veille UX a permis de :

- construire des interfaces ludiques et simples pour les ados,
- garantir une hiérarchie visuelle claire,
- et adapter le design au mobile, tout en respectant les principes d'accessibilité RGAA.

CI/CD & Docker

La mise en place de workflows GitHub Actions et de conteneurs Docker a été appuyée par une veille technique sur :

- les bonnes pratiques de Dockerisation multi-service (**base, backend, frontend**),
- la configuration d'environnements cohérents (**.env, docker-compose**),
- l'automatisation des builds, tests et initialisation via Makefile.

10. Annexes

Cette section regroupe les éléments techniques, visuels et documentaires qui complètent et illustrent les parties précédentes du dossier. Ces annexes permettent de visualiser concrètement les choix techniques, la modélisation, la structure du code et les pratiques DevOps ou UX mises en œuvre dans YouthWallet.

10.1 Diagrammes et Modèles de données

- MCD, MLD, MPD générés avec Draw.io – formats PNG exportés.
- Diagrammes UML : cas d'usage, séquence (transaction), classe (entités principales).
- Arborescence du frontend (Expo Router) et du backend (NestJS).

10.2 Références techniques externes

- Prisma ORM (modélisation, migrations, client typé)
 - <https://www.prisma.io/docs>
- NestJS – Structure modulaire, services, DTO, guards
 - 👉 <https://docs.nestjs.com>
- Authentification JWT dans NestJS
 - 👉 <https://docs.nestjs.com/security/authentication>

10.3 Captures de code et outils DevOps

- **schema.prisma** : définition complète des entités (User, Account, Transaction, etc.)
- **create-transaction.dto.ts** : validation des données côté backend
- **transaction.service.ts** : logique métier (refund, create, link category...)
- **docker-compose.yml** : orchestration base de données, backend, frontend
- **.github/workflows/ci.yml** : workflow GitHub Actions

Conclusion

Le projet [YouthWallet](#) a permis de développer une application mobile complète, pédagogique et sécurisée, répondant à une problématique actuelle : l'éducation financière des adolescents, encadrée par leurs parents. L'application combine une interface intuitive, un backend modulaire, une gestion sécurisée des données et une architecture technique robuste.

Grâce à l'utilisation de technologies modernes comme NestJS, Prisma, Docker, Expo React Native et GitHub Actions, ce projet a été conçu et développé en respectant les meilleures pratiques de l'ingénierie logicielle (principes SOLID, tests, CI/CD, sécurité, RGAA).

Ce projet illustre concrètement les compétences attendues d'un Concepteur Développeur d'Applications : concevoir, développer, documenter, sécuriser, déployer et maintenir une application complexe, au sein d'une équipe en méthode Agile.

Remerciements

Je tiens à remercier :

- Mes collaborateurs [Romain Chazaut](#) et [Augustin Yvon](#), pour leur implication tout au long du développement, entre backend, frontend et coordination technique
- L'équipe pédagogique de La Plateforme, pour l'accompagnement, les conseils techniques et les exigences formatrices ;
- Et toutes les personnes qui ont apporté leur regard extérieur, leurs retours utilisateurs ou leur soutien tout au long de cette aventure.

Ce projet a été autant un défi technique qu'une expérience humaine enrichissante.