RAPPORT DE PROJET

Semestre 2 - Développement d'application

Durée : 4 mois

HealthyCore

(Application mobile de sport et de nutrition)

Réalisé par :

- EL MOURABIT Badre
- ROBLES Nicolas
- CATTAROSSI DARTIGUELONGUE Thomas

Master 1 Informatique Parcours Full Stack Web Mobile
2024-2025

Engagement de non plagiat

Nous, soussignés, Badre El Mourabit, Nicolas Robles, et Thomas Cattarossi Dartiguelongue, déclarons être pleinement conscients que le plagiat de documents ou d'une partie d'un document publiés sur toutes formes de support, y compris l'internet, constitue une violation des droits d'auteur ainsi qu'une fraude caractérisée. En conséquence, nous nous engageons à citer toutes les sources que nous avons utilisées pour écrire ce rapport.

Signalement et description de l'usage fait des IA génératives

Dans le cadre de la rédaction de ce rapport, nous avons utilisé des outils d'intelligence artificielle générative de manière ponctuelle, comme suit :

- Nous avons demandé à ChatGPT des reformulations de certaines phrases pour améliorer la clarté du texte.
- Claude nous a aidé à structurer le plan initial du rapport.
- Nous avons vérifié et corrigé l'orthographe et la grammaire à l'aide de ChatGPT.

Sommaire

Introduction

Dans le cadre du premier semestre de Master 1, nous devions concevoir un projet de développement d'application. Nous avons choisi de nous diriger vers le domaine du sport. Le marché des applications mobiles de nutrition et de sport est un énorme pan de l'industrie mobile. En effet, il existe une grande quantité d'applications dans ce secteur. Malgré cette abondance, seules quelques-unes parviennent à intégrer efficacement sport et nutrition dans une seule et même application. Cela conduit souvent les débutants à jongler entre plusieurs applications pour reprendre leur santé en main, ce qui peut rapidement entraîner une perte de motivation.

C'est de cette observation qu'est née l'idée de **HealthyCore** : créer une application qui rassemble sport et nutrition de manière cohérente et pratique, offrant ainsi une solution complète pour accompagner les utilisateurs dans leurs objectifs de santé.

Lors du premier semestre, notre travail s'est concentré sur la phase de conception du projet. Nous avons défini les besoins fonctionnels et non fonctionnels de notre application, établi les cas d'utilisation principaux et conçu un schéma relationnel solide pour notre base de données. Cette phase s'est conclue par la réalisation d'une maquette fonctionnelle UI/UX, ainsi que par des choix technologiques pour le développement.

Le second semestre a marqué le début de la concrétisation de l'application. Après avoir posé les bases conceptuelles, nous nous sommes attelés au développement technique de l'application, en mettant en œuvre les choix architecturaux définis précédemment.

Nous avons structuré ce rapport en trois grandes parties distinctes. Dans un premier temps, nous parlerons du développement du backend, de l'implémentation de la base de données et de la logique métier. Ensuite, nous verrons le développement du frontend mobile à travers l'implémentation des services et des différents composants. Nous finirons par aborder l'implémentation des tests frontend et backend dans le cadre l'intégration continue.

I. Développement du backend

I.1) Mise en place de la base de données PostgreSQL

La mise en place de notre base de données PostgreSQL pour notre application backend s'est faite en plusieurs étapes essentielles, allant de la containerisation avec Docker de Postgres à l'initialisation des données à l'aide de Prisma.

A) Containerisation avec Docker et Docker Compose

Afin de gérer efficacement la base de données et son interface d'administration, nous avons fait le choix d'utiliser Docker Compose avec les images officielles de PostgreSQL et de pgAdmin (cf. Schéma 1). Voici comment nous avons procédé. Nous avons créé un fichier Docker Compose pour mettre en place notre infrastructure de base de données. Pour le service PostgreSQL, nous utilisons l'image par défaut de postgres, avec des variables d'environnement que nous avons configurées dans un fichier d'environnement [1]. Ce fichier contient toutes sortes d'informations importantes, comme les identifiants de l'administrateur, le nom de la base de données, et d'autres informations cruciales. Nous avons également rendu cette base de données accessible sur le port 5432:5432. Le service pgAdmin, quant à lui, est basé sur l'image de pgAdmin, les autres informations nécessaires à la configuration de cette image sont dans le .env. Le service pgAdmin est exposé sur le port 5050:80.

```
1.  postgres:
2.    image: postgres
3.    container_name: postgres_healthycore
4.    restart: always
5.    environment:
6.      POSTGRES_USER: ${POSTGRES_USER}
7.      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
8.      POSTGRES_DB: ${POSTGRES_DB}
```

Code 1 : Extrait de code du fichier docker-compose.yml

Un Makefile est mis en place pour faciliter la gestion des services Docker Compose avec des cibles telles que up, down et down-volumes. Une cible work combine les commandes up et start-back pour lancer rapidement l'ensemble sans installation ni migration.

```
1. FRONT_DIR = ./frontend
2. BACK_DIR = ./backend
3. ADMIN_DIR = ./back-office
4. DOCKER_COMPOSE = docker-compose -f
   $(BACK_DIR)/docker-compose.yml
5. up:
```

```
6.      $(DOCKER_COMPOSE) up -d
7. down:
8.      $(DOCKER_COMPOSE) down
```

Code 2 : Extrait de code du fichier Makefile

B) Configuration de Prisma

Dans le cadre de ce projet, Prisma a été choisi comme ORM pour gérer l'interaction entre l'application et la base de données PostgreSQL. Nous avons choisi Prisma car pour la gestion des évolutions du schéma de la base, Prisma propose un système de migrations.

Afin de pouvoir faire les migrations, nous avons dû créer un fichier schema.prisma contenant le modèle de données de notre application ce fichier est en conformité avec les spécifications issues des DDL SQL de Prisma, afin d'obtenir une migration correcte en adéquation avec nos attendus.

```
1. model activites {
2.   id_activite         Int      @id@default(autoincrement())
3.   nom                 String   @unique @db.VarChar(50)
4.   description         String
5.   preferences_activites preferences_activites[]
6. }
```

Code 3 : Extrait de code du fichier schema.prisma

Cette commande a pour effet de générer un dossier prisma/migrations/ contenant le SQL correspondant et d'appliquer directement cette migration sur la base de développement. Afin de faciliter le déclenchement de ces migrations lors du développement, une cible migrate a été ajoutée dans un Makefile, automatisant ainsi le processus [2].

C) Seeding Initial

Le seeding de la base de données consiste à la pré-remplir avec des données initiales indispensables au bon fonctionnement de l'application. Cela inclut notamment la création d'un compte administrateur, l'ajout de badges, d'objectifs utilisateur, de niveaux de sédentarité, ainsi que, dans un second temps, des tags et des exercices. Pour cela, un script a été développé. Il s'appuie sur les bibliothèques de prisma pour interagir avec la base, bcrypt pour le hachage sécurisé du mot de passe de l'administrateur, et dotenv pour le chargement des variables d'environnement définies dans le fichier d'environnement .

Au sein du script, plusieurs opérations sont enchaînées : création d'un utilisateur administrateur, insertion en masse des badges prédéfinis, des objectifs utilisateur et

des niveaux de sédentarité. Le script prévoit également l'ajout des tags et des exercices dans des étapes ultérieures, ainsi que la création des relations entre ces entités via des tables pivot (comme exercices_tags). Ce dispositif de seeding permet de disposer systématiquement d'une base de développement cohérente et fonctionnelle dès l'initialisation de l'environnement [3].

D) Flux de déploiement pour le développement

Afin de garantir un environnement de développement homogène et reproductible, un flux de mise en route structuré a été mis en place. Celui-ci débute par la configuration du fichier d'environnement du backend, dans lequel sont définies les variables nécessaires au fonctionnement de l'application. Une fois cette configuration réalisée, les services Docker sont lancés via la commande make up, assurant le démarrage des conteneurs nécessaires, notamment PostgreSQL et PgAdmin. Les dépendances du projet sont ensuite installées à la fois pour le backend et le frontend en utilisant make install-back et make install-front.

Le schéma de la base de données est appliqué à travers les migrations Prisma avec make migrate, puis les données initiales sont injectées à l'aide du seeding via make seed. Enfin, le backend peut être démarré avec la commande make start-back.

Ce dispositif, articulant Docker, Prisma Migrate et un script de seed, permet d'obtenir une base PostgreSQL containerisée, versionnée et alimentée de manière automatisée, assurant ainsi la cohérence et la reproductibilité de l'environnement de développement pour l'API backend.

I.2) Mise en place de l'API et développement

Le développement de notre API s'appuie sur Node.js et le framework Express, permettant de monter rapidement un serveur HTTP léger et performant. L'architecture choisie est modulaire : chaque domaine fonctionnel (authentification, gestion des utilisateurs, nutrition, etc.) est isolé dans son propre « mini-service », composé d'un service métier, d'un contrôleur, d'un routeur et, si nécessaire, de middlewares spécifiques. Cette organisation facilite la maintenance, la lisibilité du code et l'évolution indépendante de chaque composant [4].

La configuration initiale réserve une place centrale aux middlewares indispensables au bon fonctionnement de l'API. Nous utilisons dotenv pour charger, au démarrage, les variables d'environnement (port d'écoute, URL de la base, secret JWT, etc.) depuis un fichier d'environnement, cors pour autoriser les appels cross-origin depuis l'interface cliente, express.json() pour parser automatiquement les requêtes entrantes au format JSON et un routeur global monté sur /api qui délègue chaque appel aux modules fonctionnels adéquats.

```
1. app.use(
```

```
  cors({
    origin: config.CORS_ORIGIN,
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"],
  })
);
```

Code 4 : Extrait de code du fichier app.js

A) Sécurisation des données et authentification

Sur le plan de la sécurité et de l'authentification, nous combinons plusieurs outils. Tout d'abord, bcrypt pour le hachage des mots de passe avant leur stockage en base ensuite, jsonwebtoken pour la génération, la signature et la vérification des tokens JWT, utilisés dans le header Authorization des requêtes privées. Enfin, nous utilisons express-validator pour valider les schémas de données des requêtes entrantes (taille minimale d'un mot de passe, format d'email, etc.)

```
  } catch (err) {
    // Les erreurs spécifiques à JWT seront capturées par le middleware d'erreur
    if (err.name === "TokenExpiredError") {
      return next(new AppError("Votre session a expiré", 401, "TOKEN_EXPIRED"));
    }

    if (err.name === "JsonWebTokenError") {
      return next(
        new AppError("Token d'authentification invalide", 401, "INVALID_TOKEN")
      );
    }
```

Code 5 : Extrait de code d'un fichier de middleware

B) Organisation des micro-services

Enfin, chaque fonctionnalité est organisée en mini-service. Chaque module possède son service contenant la logique métier pure (accès aux données, règles de gestion), son contrôleur recevant la requête Express, appelant le service et renvoyant la réponse HTTP, ses routes liant les URL aux contrôleurs, en appliquant les middlewares de validation ou d'autorisation et, si nécessaire, des middlewares dédiés (authentification, vérification de rôle, contrôle d'accès).

Cette structure claire et répétitive permet d'ajouter ou de modifier des fonctionnalités sans impacter les autres modules, tout en maintenant un haut niveau de cohérence dans l'ensemble du projet.

Voici ci-dessous un exemple avec le module recouvrant la partie CRUD des programmes pour le Back-Office de l'application (cf. Schéma 2).

Le fichier **admin.programs.controller.js** sert de pont entre les requêtes HTTP et la logique métier. Il orchestre le flux des données : il réceptionne les paramètres de requête (comme les critères de pagination ou de recherche), délègue le traitement au service métier, puis structure la réponse HTTP adaptée (succès avec les données formatées ou gestion centralisée des erreurs). Par exemple, la fonction *getAllPrograms* extrait les paramètres de pagination de la requête, interroge le service, et renvoie une réponse JSON standardisée avec les programmes récupérés et les métadonnées de pagination.

```
1. exports.getAllPrograms = catchAsync(async (req, res, next)
   => {
2.   const data = await
   adminProgramService.getAllPrograms(...);
3. // Renvoi formaté au client
4. res.status(200).json({ status: "success", data });
5. });
```

Code 6 : Extrait de code de admin.programs.controller.js

Le fichier **admin.programs.service.js** encapsule la logique métier et les interactions complexes avec la base de données. À l'aide de Prisma, il construit des requêtes optimisées, gère les relations entre entités (comme les tags associés aux programmes), et transforme les résultats bruts en formats adaptés au frontend. Un extrait illustre cette dualité : une requête combine *findMany* pour récupérer les programmes paginés et `count` pour calculer le total d'enregistrements, permettant une pagination précise tout en minimisant les appels à la base de données. Les données sont ensuite restructurées pour fournir une réponse claire, comme la transformation des tags bruts en objets simplifiés.

```
1. const [programs, total] = await Promise.all([
2.   prisma.programmes.findMany({...}), // Récupération
   données
3.   prisma.programmes.count({ where }), // Comptage total
   pour pagination
4. ]);
```

Code 7 : Extrait de code de admin.programs.service.js

Le fichier **admin.programs.routes.js** définit l'architecture publique de l'API. Chaque route associe une URL à un contrôleur spécifique, tout en intégrant des *middlewares* pour sécuriser l'accès. Par exemple, la route *GET /:programId* applique successivement deux contrôles d'accès (*checkAuth* pour l'authentification et *isAdmin* pour le rôle) avant de rediriger vers le contrôleur. Ce fichier agit comme

une carte routière qui garantit que seuls les utilisateurs autorisés peuvent déclencher les opérations sensibles, comme la suppression d'un programme.

```
1. // Extrait : Sécurisation d'une route avec 2 middlewares
2. router.get("/:programId", checkAuth, isAdmin,
   adminProgramsController.getProgramById);
```

Code 8 : Extrait de code de admin.programs.routes.js

Enfin, bien que non inclus dans les extraits, le fichier **auth.middleware.js** joue un rôle critique de vigile. Les fonctions *checkAuth* et *isAdmin* vérifient respectivement l'identité de l'utilisateur et ses privilèges avant toute interaction avec les contrôleurs. Elles forment un rempart contre les accès non autorisés, s'intégrant discrètement dans les chaînes de middlewares des routes pour sécuriser l'ensemble du module sans polluer la logique métier.

C) Intégration d'une API tierce: OpenFoodFacts

L'intégration d'OpenFoodFacts repose sur **Axios** pour interroger l'API tierce de manière ciblée. Lors d'une recherche par nom (méthode searchProducts), la requête HTTP est paramétrée pour limiter les résultats aux produits disponibles en France. Le code utilise le filtre tag_0: "france" dans les paramètres de l'appel API, combiné à des contraintes sur la structure de réponse (json: 1). Cela garantit que seuls les produits pertinents pour le marché français sont synchronisés avec l'application [5].

Pour un produit spécifique (méthode getProductByBarcode), Axios récupère les données via l'endpoint world.openfoodfacts.org/api/v2/product/. Les informations brutes (nutriments, ingrédients) sont normalisées : valeurs numériques converties, noms tronqués si nécessaire pour respecter les limites de la base de données, et catégories OpenFoodFacts (categories_tags) transformées en tags applicatifs. Un système de vérification évite les doublons. Les erreurs d'API (comme un produit non trouvé) sont gérées discrètement – une réponse { status: "fail" } est retournée sans interrompre le flux. Cette intégration maintient un équilibre entre données externes et cohérence interne, tout en respectant les spécificités régionales via les paramètres d'appel Axios.

```
1.          const response = await axios.get(
2.            "https://world.openfoodfacts.org/cgi/search.pl",
3.            {
4.              params: {
5.                search_terms: searchTerm,
6.                search_simple: 1,
7.                action: "process",
8.                json: 1,
9.                page_size: limit - formattedLocalProducts.length,
10.                  tagtype_0: "countries",
11.                  tag_contains_0: "contains",
12.                  tag_0: "france",
```

```
13.                 },});
```

Code 9 : Extrait de code de openfoodfacts.service.js

II. Développement du frontend mobile

II.1) Architecture et structure du frontend

A) Choix technologiques

L'architecture frontend de HealthyCore a été conçue en privilégiant la performance, la maintenabilité et l'expérience utilisateur (cf. Schéma 3). Les choix technologiques suivants ont guidé le développement de l'application.

Framework principal : React Native avec Expo

React Native a été sélectionné lors de la conception de l'application au 1er semestre (cf. Rapport de conception, IV. B. Choix des technologies). Ce choix présente plusieurs avantages [6] :

- Code unique pour iOS et Android, réduisant de manière significative le temps de développement ;
- Performances natives grâce à la conversion des composants React en composants natifs ;
- Large écosystème de bibliothèques et de composants réutilisables.

L'utilisation d'Expo comme plateforme de développement a considérablement simplifié le processus grâce à des outils intégrés lors de la phase de développement (hot reloading, outils de développeur), à un accès simplifié aux fonctionnalités natives (permissions, caméra) [7] ainsi qu'une configuration minimale pour la prise en main de l'outil (Expo CLI) qui ne nécessite pas l'installation complexe d'environnements de développement natifs.

Routing : Expo Router

Le système de navigation de l'application repose sur Expo Router, une solution qui est basée sur les fichiers. Cette solution repose sur une structure intuitive où l'arborescence des fichiers [8] reflète directement celle des routes. Elle utilise une navigation déclarative qui simplifie la gestion des transitions entre écrans.

```
1. import { Link } from 'expo-router';
2. import { View } from 'react-native';
3. export default function Route() {
4.  return (
5.   <View>
6.    <Link href="/about">About</Link>
```

```
7.    <Link href={{pathname: '/user/[id]',params: { id:
   'bacon' }}}>View user</Link>
8.   </View>);}
```

Code 10 : Exemple d'une fonction de navigation utilisant Expo Router

Ce bout de code (lignes 5 à 9) montre un composant simple qui rend deux liens de navigation : l'un vers une page statique, l'autre vers une page dynamique avec un paramètre (id). Cela illustre la simplicité et la puissance du système de routing basé sur les fichiers.

Gestion d'état : Contexte React et hooks personnalisés

La gestion de l'état global et local s'appuie sur les fonctionnalités natives de React :

- Contexte React pour le partage d'état global dans l'arborescence des composants ;
- Hooks personnalisés encapsulant la logique métier réutilisable (formulaires, validation, etc.).

Cette façon de faire a permis d'éviter l'introduction de bibliothèques externes de gestion d'état, simplifiant ainsi l'architecture tout en maintenant une séparation claire entre la logique et la présentation.

UI/UX : Composants sur mesure avec système de design cohérent

L'interface utilisateur repose sur un système de design développé spécifiquement pour HealthyCore :

- Bibliothèque de composants réutilisables suivant les principes de l'atomic design ;
- Système de style centralisé avec constantes pour les couleurs, espacements et typographie ;
- Composants adaptatifs s'ajustant aux différentes tailles d'écran.

L'atomic design, conceptualisée par Brad Frost [9], structure les éléments d'interface en cinq niveaux distincts et hiérarchiques : atomes (éléments fondamentaux comme les boutons), molécules (groupes d'atomes fonctionnant ensemble), organismes (assemblages complexes), templates (structures de page) et pages (instances spécifiques). Cette approche modulaire favorise la cohérence, la réutilisabilité et la maintenance, tout en facilitant l'évolution progressive de l'interface utilisateur.

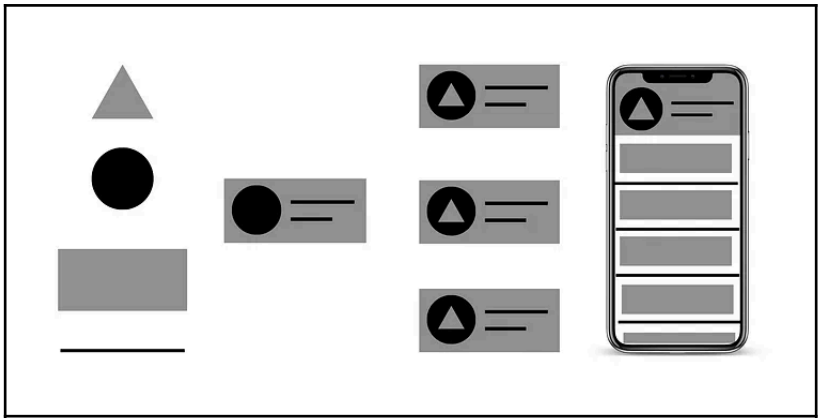

Figure 1 : Exemple de la méthodologie de l'atomic design

Dans la Figure 1, on peut observer cette hiérarchie visuellement représentée : les formes simples (triangle, cercle, rectangle) symbolisent les atomes, les groupes de ces formes deviennent des molécules, leur composition forme des organismes, qui sont ensuite disposés en templates dans l'interface d'un smartphone, représentant une page complète. Cette représentation montre comment de simples composants s'assemblent progressivement pour former une interface cohérente et complète.

B) Structure du projet

La structure du projet HealthyCore a été organisée pour favoriser la maintenabilité, la lisibilité et la séparation des préoccupations. L'architecture des dossiers a été pensée pour refléter la logique métier.
Le projet adopte une organisation modulaire [10] où chaque répertoire répond à une responsabilité spécifique :

- app/ : Contient toutes les routes et écrans de l'application, structurés selon la logique fonctionnelle du produit ;
- components/ : Regroupe les composants réutilisables selon leur niveau d'abstraction et leur rôle ;
- constants/ : Centralise les valeurs constantes comme les couleurs, les dimensions et la typographie ;
- context/ : Implémente les contextes React pour la gestion de l'état global ;
- hooks/ : Rassemble les hooks personnalisés pour encapsuler la logique réutilisable ;
- services/ : Définit les services pour communiquer avec les API externes.

Cette séparation permet une meilleure isolation des préoccupations, facilitant le développement et la maintenance du code à long terme.

II.2) Système d'authentification et inscription

Le système d'authentification et d'inscription constitue un élément fondamental de l'application. Cette partie détaille l'implémentation technique de ces fonctionnalités

essentielles, en se concentrant sur la sécurité, l'expérience utilisateur et la gestion efficace des données.

A) Contexte d'authentification

La gestion de l'authentification dans HealthyCore repose sur un système centralisé qui utilise le Context API de React [11] pour partager l'état d'authentification à travers toute l'application. Cette approche permet de rendre l'état d'authentification facilement accessible depuis n'importe quel composant.

Mise en place du AuthContext pour la gestion globale de l'état d'authentification

Le AuthContext encapsule toute la logique liée à l'authentification, offrant une interface claire pour les autres composants :

```
1. interface AuthContextType {
2.   isAuthenticated: boolean;
3.   user: User | null;
4.   loading: boolean;
5.   error: string | null;
6.   login: (email: string, password: string) =>
   Promise<void>;
7.   logout: () => Promise<void>;
8.   register: (userData: any) => Promise<void>;
9.   clearError: () => void;}
10.  const AuthContext = createContext<AuthContextType |
   undefined>(undefined);
```

Code 11 : Définition de l'interface du contexte d'authentification

Dans le Code 11, on définit l'interface qui spécifie la structure complète du contexte d'authentification, incluant les états (isAuthenticated, user, loading, error) et les méthodes de manipulation (login, logout, register, clearError).
On va pouvoir utiliser le contexte pour répondre à différents besoins :

- Gérer les tokens d'authentification dans les méthodes login et logout grâce à Expo SecureStore [12] ;
- Au démarrage de l'application, vérifier la présence d'un token valide stocké localement pour maintenir la session utilisateur ;
- Protéger les routes qui nécessitent une connexion valide.

Gestion des routes protégées

La protection des routes est assurée par l'utilisation des layouts dans Expo Router. Chaque section de l'application vérifie l'état d'authentification en important le contexte et redirige l'utilisateur si nécessaire.

```
1. import { useAuth } from "../../context/AuthContext";
2. export default function UserLayout() {
3.   const { isAuthenticated } = useAuth();
4.   if (!isAuthenticated)
5.     {return <Redirect href={"/welcome" as any} />;}
```

Code 12 : Extrait du fichier UserLayout.tsx

Ce layout, utilisé par tous les écrans de l'application du côté utilisateur, vérifie si l'utilisateur est authentifié via le AuthContext (ligne 4). S'il ne l'est pas, il est automatiquement redirigé vers l'écran de bienvenue (ligne 5).

B) Flux d'inscription multi-étapes

L'inscription dans HealthyCore est un processus complet qui collecte diverses informations sur l'utilisateur pour personnaliser l'expérience. Cette complexité est gérée à travers un flux d'inscription multi-étapes.

Conception du parcours d'inscription en 8 étapes

Le parcours d'inscription est divisé en 8 étapes distinctes, chacune avec un objectif spécifique :

1. Informations de profil : Collecte des informations de base (nom, prénom, email, mot de passe) ;
2. Attributs physiques : Caractéristiques physiques (genre, date de naissance, poids, taille) ;
3. Niveau d'activité : Degré d'activité quotidienne de l'utilisateur ;
4. Objectif de poids : Définition du poids cible et calcul des besoins caloriques ;
5. Plan nutritionnel : Choix d'un plan adapté aux objectifs et préférences ;
6. Régime alimentaire : Spécification de contraintes alimentaires (végétarien, sans gluten, etc.) ;
7. Activités préférées : Sélection des types d'activités physiques préférées ;
8. Séances hebdomadaires : Définition du nombre de séances d'entraînement par semaine.

L'ensemble du parcours d'inscription est accompagné d'un indicateur de progression qui permet à l'utilisateur de visualiser clairement sa position dans le processus et le nombre d'étapes restantes. Chaque étape est matérialisée par un écran distinct dans l'arborescence des routes.

Développement du RegistrationContext pour la gestion de l'état

Comme pour l'authentification, la gestion des données entre chaque écran de l'inscription doit impérativement fonctionner. C'est donc dans le but de maintenir des données cohérentes entre les différentes étapes qu'un contexte dédié RegistrationContext a été développé.

```
interface RegistrationContextType {
  data: Partial<RegistrationData>;
  setField: <K extends keyof RegistrationData>(field:
K,value: RegistrationData[K]) => void;
  setFields: (fields: Partial<RegistrationData>) =>
void;
  resetForm: () => void;
  goToNextStep: () => void;
  goToPreviousStep: () => void;
  currentStep: number;
  totalSteps: number;
    completeRegistration: () => Promise<void>;
    validateStep: (step: number) => Promise<boolean>;
    loading: boolean;
    error: string | null;}
```

Code 13 : Définition de l'interface du contexte d'inscription

Ce contexte fournit :

- Un objet data qui contient toutes les informations collectées ;
- Des méthodes pour mettre à jour les données (setField, setFields) ;
- Des fonctions de navigation entre les étapes (goToNextStep, goToPreviousStep) ;
- Une fonction de validation pour chaque étape (validateStep) ;
- Une méthode finale pour compléter l'inscription (completeRegistration).

L'utilisation du contexte permet aux composants de chaque étape d'accéder aux données globales d'inscription sans avoir à les passer explicitement entre les écrans.

Validation des données à chaque étape

Chaque étape du processus d'inscription intègre une validation pour garantir l'exactitude et la cohérence des données.

```
const validateStep = async (step: number):
Promise<boolean> => {
  try {
    switch (step) {
      case 1:
        // Validation de l'étape 1: Informations de base
        const profileValidation = await
validationService.validateProfile({
          firstName: data.firstName || "",
          lastName: data.lastName || "",
          email: data.email || "",
            password: data.password || "",
         });
```

```
          // Autres cas pour les différentes étapes...
        case 4:
          // Validation de l'étape 4: Poids cible
          if (data.targetWeight === undefined) {
            setError("Le poids cible est requis");
            return false;
          }
```

Code 14 : Extrait de la fonction de la validation validateStep

La fonction validateStep effectue une validation spécifique en fonction de l'étape actuelle du formulaire. Par exemple ici, l'étape 1 vérifie les champs de profil via un service de validation externe, tandis que l'étape 4 s'assure que le poids cible a bien été renseigné avant de poursuivre.
La validation est effectuée à la fois côté client et côté serveur via des appels API, garantissant la double vérification des données et une meilleure expérience utilisateur.

II. 3) Composants UI réutilisables

La création d'une interface utilisateur cohérente, esthétique et fonctionnelle est essentielle pour offrir une expérience utilisateur de qualité. Notre application s'appuie sur un système de design robuste et une bibliothèque de composants réutilisables qui facilitent le développement tout en garantissant une uniformité visuelle à travers l'écran.

A) Système de design et cohérence visuelle

HealthyCore adopte une approche systématique pour la conception d'interface, avec des règles clairement définies pour les couleurs, la typographie, les espacements et les dimensions.

Système de couleurs

Les couleurs dans HealthyCore sont organisées autour d'une palette principale complétée par des couleurs fonctionnelles et des nuances secondaires. Elles sont définies dans le fichier constants/Colors.ts.

```
const Colors = {
  // Couleurs principales
  brandBlue: ["#92A3FD", "#9DCEFF"], // Dégradé de bleu
  secondary: ["#C58BF2", "#EEA4CE"], // Dégradé secondaire
  // Couleurs de base
  black: "#1D1617",
  white: "#FFFFFF",
  red: "#FF5252",
  green: "#4CAF50",
    // Nuances de gris
    gray: {dark: "#7B6F72", medium: "#ADA4A5", light: "#DDDADA", ultraLight: "#F7F8F8",},
    // Couleurs sémantiques
    success: "#4CAF50",
    warning: "#FFC107",
    error: "#FF5252",
    info: "#2196F3",
    // Opacités pour différents états
    opacity: { disabled: 0.5,pressed: 0.7,},};
```

Code 15 : Extrait du fichier Colors.ts

Ce système de couleurs permet de garantir une utilisation cohérente des teintes [13] à travers à l'application avec :

- Une identité visuelle distincte basée sur les dégradés de bleu et violet (cf. Rapport de conception, III. C. Maquettes UX/UI) ;
- Des couleurs sémantiques spécifiques (succès, erreur, avertissement) ;
- Une gamme de gris pour les éléments secondaires (texte, bordures, etc.).

Système typographique

La typographie est définie dans le fichier constants/Fonts.ts, avec une structure qui organise les polices, tailles et styles.

```
const FontFamilies = {
  primary: "Poppins-Regular",
  primaryBold: "Poppins-Bold",}
const FontSizes = {
  xs: 10,
  sm: 12,
  md: 14,}
const TextStyles = {h1: {
    fontFamily: FontFamilies.primaryBold,
      fontSize: FontSizes["4xl"],
      fontWeight: FontWeights.bold,
      lineHeight: FontSizes["4xl"] * LineHeights.tight}}
```

Code 16 : Extrait du fichier Fonts.ts

Cette structure permet :

- Une hiérarchie claire des textes [14] (h1 à h5, body, caption, etc.) ;
- L'utilisation cohérente de la famille de police Poppins ;
- Des proportions harmonieuses entre les différents niveaux de titres.

Exemple d'utilisation des constantes de thèmes

Grâce à la centralisation des constantes, il est possible de les intégrer dans n'importe quel design de notre application. Il suffit juste d'importer le module au sein de la page pour pouvoir utiliser toutes les constantes

```
1. const styles = StyleSheet.create({
2.   container: {
3.     backgroundColor: Colors.white,
4.     borderRadius: Layout.borderRadius.md,
5.     padding: Layout.spacing.md,  },});
```

Code 17 : Exemple d'utilisation des constantes

La centralisation des constantes de design dans des fichiers dédiés présente plusieurs autres avantages :

- Facilité de maintenance : Modifier une couleur ou un espacement à un seul endroit impacte toute l'application ;
- Cohérence garantie : Utilisation des mêmes valeurs à travers tous les composants ;
- Adaptabilité : Possibilité d'implémenter des thèmes alternatifs (mode sombre, etc.) en modifiant uniquement les constantes.

Cette approche facilite également l'onboarding des nouveaux développeurs, qui peuvent rapidement comprendre et adopter les conventions de design de l'application.

B) Création de composants personnalisés pour l'interface

Afin d'obtenir une interface de design complète et soigné à l'instar de celle définie lors de la conception, nous avons créé une bibliothèque complète de composants personnalisés qui sont utilisés dans toute l'application.

Création du composant Button

Le composant Button constitue un élément fondamental [15] de l'interface utilisateur. Il devait répondre à plusieurs exigences importantes : s'adapter visuellement à différents contextes d'utilisation, offrir une expérience interactive cohérente, et rester extrêmement flexible pour pouvoir le modifier si besoin.

Pour maximiser sa réutilisabilité, nous avons conçu ce composant avec de nombreuses options de personnalisation :

```
1. const Button: React.FC<ButtonProps> = ({
2.   text,
3.   onPress,
4.   variant = "primary",
5.   size = "md",
6.   fullWidth = false,
7.   leftIcon,
8.   rightIcon,
9.   loading = false,
10.    disabled = false,
11.    style,
12.    textStyle,
13.    ...props})
```

Code 18 : Paramètres du composant Button

Les caractéristiques clés de ce composant sont :

- Plusieurs variantes (primary, secondary, outline, ghost) pour s'adapter à différentes hiérarchies d'actions ;
- Différentes tailles (sm, md, lg) pour s'intégrer harmonieusement dans divers contextes d'interface ;
- Support pour les icônes (gauche et droite) afin d'améliorer la compréhension visuelle des actions ;
- État de chargement permettant de fournir un retour visuel lors des opérations asynchrones ;
- Personnalisation via des props de style pour des ajustements contextuels sans compromettre la cohérence globale.

Cette approche modulaire permet d'utiliser le même composant Button à travers toute l'application, tout en adaptant son apparence et son comportement selon les besoins spécifiques de chaque écran. Cela garantit une cohérence visuelle et interactive, tout en réduisant considérablement la duplication de code.

Création du composant Card

Le composant Card offre un conteneur polyvalent [16] pour structurer visuellement l'interface et présenter des informations de manière claire et cohérente. Ce composant a été conçu pour être à la fois flexible et simple d'utilisation, s'adaptant à différents types de contenu.

```
const Card: React.FC<CardProps> = ({
  children,
  style,
  onPress,
  variant = "default",
  elevation = "sm",
  backgroundColor = Colors.white,
  disabled = false,
})
```

Code 19 : Paramètres du composant Button

Les principales caractéristiques du composant Card sont :

- Mobilité de présentation avec plusieurs variantes (default, elevated, outlined) ;
- Niveaux d'élévation configurables (sm, md, lg) pour créer une hiérarchie visuelle ;
- Possibilité de rendre la carte interactive via la propriété onPress ;
- Animations subtiles lors des interactions pour enrichir l'expérience utilisateur ;
- Personnalisation de l'apparence via des propriétés (backgroundColor, style).

Cette approche permet d'utiliser le même composant Card pour des cas d'usage très différents, comme la présentation d'informations statiques ou la création d'éléments cliquables, tout en maintenant une cohérence visuelle à travers l'application.
Le composant Card est notamment utilisé pour afficher les plans nutritionnels, les résumés d'activités sportives, et les cartes de statistiques sur le tableau de bord. Cette réutilisation d'un même composant pour différents contextes contribue à l'unité visuelle de l'application tout en réduisant significativement la quantité de code à maintenir.

II. 4) Gestion de l’état et des données

La gestion efficace de l’état et des données est un partie cruciale de l’application. L’état (ou state) représente l’ensemble des données internes d’un composant qui influencent son affichage et son comportement. Pour garantir la maintenabilité et la performance, une architecture a été mise en place autour de services spécialisés et de hooks personnalisés.

A) Architecture des services

Afin de récupérer toutes les données du backend, notre application repose sur une architecture orientée services qui centralise et organise tous les appels API. Cette approche facilite la maintenance et renforce la séparation des préoccupations.

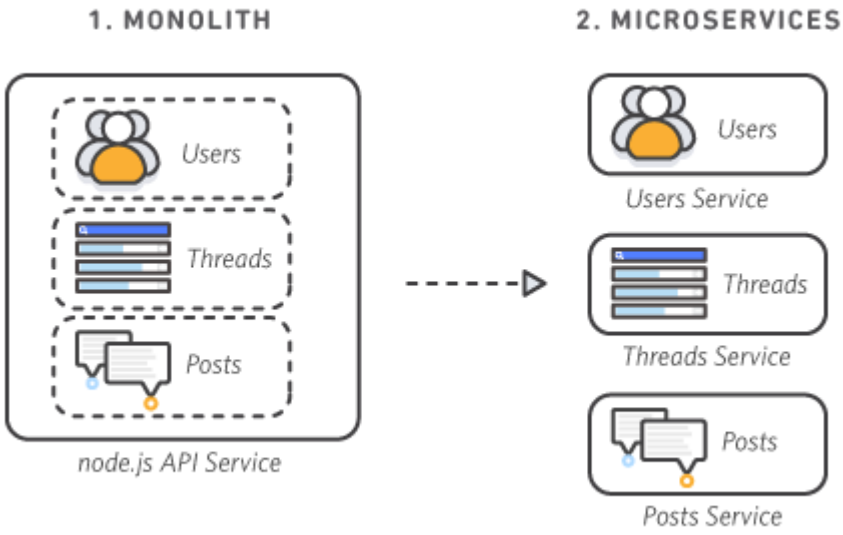

Figure 2 : Architecture orientée services [17]

Organisation des services API

L'architecture des services est structurée autour d'un service de base, apiService, qui gère les interactions fondamentales avec les API, et des services spécifiques construits au-dessus pour répondre aux différentes fonctionnalités de l'application.

```
services/
├── api.service.ts          # Service de base pour les requêtes HTTP
├── auth.service.ts         # Service d'authentification
├── data.service.ts         # Service de données générales
├── nutrition.service.ts    # Service pour la nutrition
├── programs.service.ts     # Service pour les programmes sportifs
├── user.service.ts         # Service pour les données utilisateur
└── validation.service.ts   # Service de validation
```

Code 20 : Organisation des services

Chaque service est responsable d'un domaine fonctionnel précis. Par exemple auth.service.ts gère l'authentification, tandis que nutrition.service.ts s'occupe des opérations liées à la nutrition.

```
const apiService = {
  async request<T = any>(): Promise<T> {// Méthode principale pour effectuer une requête API},
  async get<T = any>(): Promise<T> {// Méthode pour les requêtes GET},
```

```
4.   // Idem pour POST, PUT, et DELETE.};
```

Code 21 : Extrait simplifié du fichier services/api.service.ts

L'utilisation d'un service de base apiService permet de centraliser la logique commune à tous les appels API.

Stratégie de gestion des erreurs

Pour offrir une expérience utilisateur optimale, l'application implémente une stratégie de gestion des erreurs à plusieurs niveaux qui permet de détecter, traiter et présenter les erreurs de manière appropriée.
La première ligne de défense se situe dans le service API de base qui capture et normalise les erreurs provenant du backend :

```
1. if (!response.ok) {
2.   const errorData: ApiErrorData = await response.json();
3.   throw {
4.     status: response.status,
5.     message: errorData.message || "Une erreur est
   survenue",
6.     code: errorData.code || "UNKNOWN_ERROR",
7.     errors: errorData.errors || null,
8.   };
9. }
```

Code 22 : Gestion d'erreur du fichier services/api.service.ts

Cette approche garantit que toutes les erreurs HTTP sont transformées en un format cohérent avec :

- Un code de statut HTTP pour identifier le type d'erreur ;
- Un message utilisateur explicite ;
- Un code d'erreur technique pour le débogage ;
- Un objet d'erreurs détaillées par champ (utile pour les validations de formulaire).

Ensuite les services personnalisés (sport, nutrition, etc.) se chargent de traiter les codes d'erreur et d'envoyer les messages adéquats aux composants d'interface. Ces composants peuvent ensuite manipuler les erreurs en les affichant sur des alertes, sur l'écran directement ou via des modales.

B) Hooks personnalisés

Pour améliorer la réutilisabilité du code et faciliter le développement, nous avons créé plusieurs hooks personnalisés [18] qui encapsulent la logique métier complexe et récurrente.

Hook de gestion de formulaires

Le hook useForm simplifie considérablement la gestion des formulaires dans l'application et permet d'éviter une grande quantité de duplication de code et de logique.

```
1. export function useForm<T>({initialValues, validate,
   onSubmit}) {
2.   const [values, setValues] =
   useState<T>(initialValues);
3.   const [errors, setErrors] = useState({});
4.   const [touched, setTouched] = useState({});
5.   // Autres fonctionnalités du hook...
6.   return {
7.     values, errors, touched, handleChange, handleSubmit,
8.     // Autres propriétés et méthodes retournées
9.   };
10.  }
```

Code 23 : Structure du hook useForm

Ce hook centralise toute la logique liée aux formulaires, notamment :

- La gestion de l'état des champs ;
- La validation des données en temps réel ;
- Le suivi des champs modifiés ;
- La soumission sécurisée avec gestion des erreurs.

L'utilisation de ce hook dans les écrans d'inscription et de connexion a permis de réduire considérablement la complexité du code et d'assurer une expérience utilisateur cohérente.

Hook pour la sélection de dates

La sélection de dates présente des défis particuliers, notamment en raison d'une implémentation différente entre iOS et Android.

```
1. export function useDatePicker({initialDate, minDate,
   maxDate}) {
2.   const [date, setDate] = useState(initialDate);
3.   const [show, setShow] = useState(false);
4.   const showDatePicker = useCallback(() => {
5.     // Logique adaptée selon la plateforme
6.     if (Platform.OS === "android") {
7.       DateTimePickerAndroid.open({/* config */});
8.     } else {
9.       setShow(true);
10.      }
11.    }, [/*dépendances*/]);
12.    return { date, show, showDatePicker, hideDatePicker
   };
```

```
13.  }
```

Code 24 : Hook useDatePicker avec adaptations multi-plateformes

Ce hook gère la sélection de date de façon multiplateforme. Il initialise d'abord deux états : date et show (lignes 2-3). Puis la fonction showDatePicker adapte le comportement selon la plateforme. L'interface retournée (ligne 14) permet aux composants d'accéder facilement à la date sélectionnée et de contrôler l'affichage du sélecteur sans avoir à se préoccuper des détails d'implémentation à chaque plateforme.
La séparation de cette logique complexe dans un hook dédié permet aux composants d'interface de rester simples et lisibles, tout en garantissant un comportement cohérent à travers l'application.

III. Tests

III. 1) Stratégie de tests mise en œuvre

Dans le cadre de ce projet, notre équipe a accordé une importance à la qualité et à la fiabilité du code. Nous avons élaboré une stratégie de tests couvrant les aspects frontend et backend de l'application HealthyCore.

A) Tests Frontend

Notre architecture de tests est structurée selon trois niveaux distincts :

Tests unitaires :

Ces tests ciblent les hooks, services et fonctions utilitaires de manière isolée. Le hook useForm a fait l'objet d'une attention particulière en raison de son utilisation généralisée dans les formulaires de l'application.

Tests de composants :

Pour les composants d'interface utilisateur réutilisables, nous validons leur rendu et leur comportement lors d'interactions. L'implémentation de ces tests a présenté certaines difficultés spécifiques à React Native, mais nous avons développé une méthodologie efficace utilisant un système de mocks approprié. [19]

Tests d'intégration :

Ces tests évaluent les interactions entre différents composants et services. Nous vérifions notamment que les processus critiques, tels que l'authentification, fonctionnent correctement de bout en bout. [20]

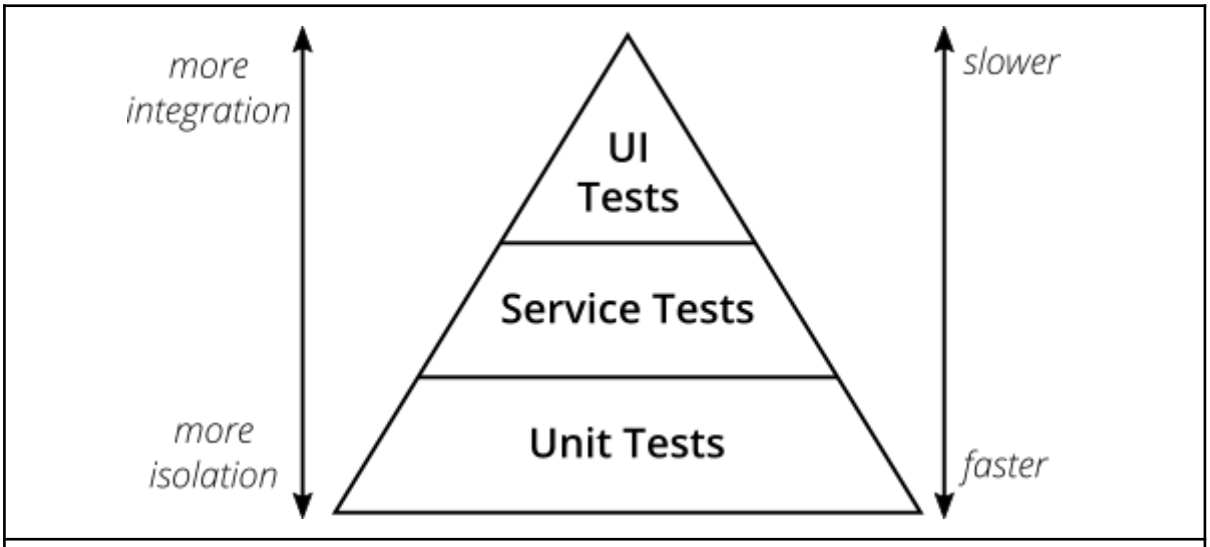

Figure 3 : Diagramme de la pyramide de tests frontend

B) Tests Backend

Pour le backend, notre approche s'est concentrée sur la validation des API et de la logique métier, avec une attention particulière portée à l'intégrité des données.

Tests d'API

Nous vérifions systématiquement que certaines routes API répondent conformément aux spécifications dans différents scénarios d'utilisation. Cette validation est essentielle pour garantir la compatibilité avec le frontend. [22]

```
1. it('procède à l'enregistrement d\'un utilisateur, sa
   connexion et la vérification de son token', async () =>
   {
2.   // Génération d'un email unique pour le test
3.   const userEmail = `test_${Date.now()}@example.com`;
4.   // Étapes d'enregistrement, connexion et vérification
5.   expect(verifyRes.body.data.valid).toBe(true);
6. });
```

Code 25 : Validation du flux d'authentification complet

Base de données de test

Nous avons implémenté une instance PostgreSQL dédiée aux tests. Cette base de données est recréée lors de chaque exécution des tests afin d'assurer leur isolation et leur reproductibilité. [21]

III. 2) Intégration Continue (CI)

Nous avons configuré GitHub Actions pour automatiser l'exécution des tests à chaque commit et pull request.

A) Workflows CI

Deux workflows principaux ont été établis:

Frontend CI

Ce workflow teste l'application React Native, vérifie la validité de la compilation et génère des rapports de couverture détaillés. [25]

Backend CI

Ce workflow exécute les tests d'API, valide les migrations de base de données et procède à une analyse statique du code. [24]
Un aspect notable de notre configuration réside dans l'intégration d'un environnement PostgreSQL directement dans le workflow GitHub Actions : [23]

```yaml
# Configuration PostgreSQL dans GitHub Actions
services:
  postgres:
    image: postgres:latest
    env:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      POSTGRES_DB: db_healthycore_test
```

Code 26 : Configuration de PostgreSQL dans le workflow GitHub Actions

III. 3) Défis rencontrés et solutions

A) Tests React Native

Le défi majeur a consisté en l'implémentation de tests pour l'environnement React Native. Les composants natifs présentent des difficultés spécifiques lorsqu'ils sont testés dans un contexte Node.js.
Notre solution a été d'élaborer un système de mocks :

```javascript
// Approche de mock pour les composants React Native
jest.mock('react-native', () => ({
  View: jest.fn(props => ({ type: 'View', props })),
  Text: jest.fn(props => ({ type: 'Text', props })),
}));
```

Code 27 : Système de mocks pour les composants React Native

Cette méthodologie nous a permis de tester la logique des composants indépendamment de leur rendu natif. [26]

B) Tests d'intégration end-to-end API

L'implémentation de tests end-to-end au niveau des API backend a représenté un défi technique majeur. Ces tests reproduisent des séquences complètes d'interactions utilisateur en enchaînant plusieurs appels API successifs, comme dans le cas du parcours d'authentification (connexion, vérification de token). [27]

La nature asynchrone de ces opérations, combinée à la nécessité de maintenir un état cohérent entre les requêtes (tokens d'authentification, IDs créés, etc.), a initialement posé des problèmes de fiabilité.

Pour garantir la stabilité de ces tests, nous avons implémenté plusieurs solutions :

- **Augmentation ciblée des timeouts** : Comme on peut le voir dans notre configuration jest.setTimeout(30000), nous avons significativement augmenté le timeout global pour accommoder les opérations asynchrones complexes impliquant la base de données.

- **Génération dynamique d'identifiants** : Nous utilisons systématiquement Date.now() pour créer des emails uniques pour chaque exécution de test, évitant ainsi les conflits de données entre les différentes exécutions.
- **Nettoyage de la base de données** : Notre configuration beforeAll inclut une instruction TRUNCATE TABLE users CASCADE qui assure que la base de données de test est dans un état propre avant le début des tests.
- **Isolation de l'environnement** : Nous utilisons une base de données PostgreSQL dédiée aux tests, configurée via les variables d'environnement dans .env.test, garantissant que les tests n'interfèrent jamais avec les données de développement ou de production.

IV. Bilan et perspectives

IV.1) Comparatif des fonctionnalitées prévues et implémentées

La Version 1 a été intégralement finalisée, couvrant les bases d'un écosystème complet de santé connectée. L'authentification robuste (inscription/connexion sécurisée) et la gestion administrative (CRUD des programmes, séances, tags, aliments/recettes) forment le socle technique. Les utilisateurs peuvent suivre leur nutrition via l'intégration d'OpenFoodFacts (synchro des codes-barres), adhérer à des programmes sportifs structurés (avec suivi d'évolution visuel), et valider des objectifs quotidiens. La découverte de contenu (programmes, recettes) et le suivi nutritionnel s'appuient sur une API interne normalisée, combinant pagination et filtrage par tags.

Les Versions 2 et 3, initialement prévues pour étoffer les fonctionnalités sociales et personnalisées (recettes/programmes personnalisés, signalements, évaluations), ont été reportées. Les contraintes de temps et la charge de travail ont priorisé la stabilisation de la V1, notamment l'optimisation des appels API externes et la gestion des transactions critiques (exemple : suppression en cascade des programmes). Ces évolutions restent inscrites dans une roadmap future, nécessitant un cadre technique renforcé (comme un système de queuing pour les signalements) et des ressources dédiées.

IV.2) Perspectives d'évolutions d'HealthyCore

L'évolution prioritaire vise à finaliser les versions V2 et V3, incluant la création de programmes et recettes personnalisés, un système de signalement d'aliments avec modération, ainsi que des notations communautaires et des recettes générées par IA selon les préférences utilisateurs.

L'innovation technologique intégrerait :

- Reconnaissance photo d'étiquettes (OCR) pour ajouter des aliments non répertoriés.
- Application mobile offline avec synchronisation automatique.
- Suggestions intelligentes (programmes/recettes) basées sur les objectifs et l'historique.

L'ouverture à des partenariats professionnels (coachs, nutritionnistes) enrichirait l'offre, tandis qu'une API sécurisée permettrait des intégrations externes. Un système de récompenses ludiques (badges, défis) boosterait l'engagement.

Ces évolutions, déployées progressivement selon les retours utilisateurs, transformerait l'application en plateforme connectée et interactive, bien au-delà d'un simple suivi fitness.

Conclusion

Ce projet a permis de consolider les compétences techniques et méthodologiques abordées durant le Master 1 et les années antérieures. En concevant une application full-stack complète, nous avons approfondi notre maîtrise du développement front-end (interfaces utilisateur réactives) et back-end (API REST structurée avec Node.js et Express), ainsi que la gestion de base de données relationnelle via Prisma et PostgreSQL. Malgré les défis techniques rencontrés, notamment lors de l'intégration des API tierces et des tests cross-platform, nous avons réussi à livrer une Version 1 pleinement fonctionnelle offrant des fonctionnalités essentielles de suivi sportif et nutritionnel.

Webographie

[1] - Vadym Chernykh. “Use Docker to Dockerize Express.js & Prisma with Postgres & PGAdmin.” Medium, 27 Jan. 2024, medium.com/@vadymchernykh/use-docker-to-dockerize-express-js-prisma-with-postgres-pgadmin-f0586becfab0. Accessed 21 May 2025.

[2] - Sharma, Vishal. “Run PostgreSQL and PGAdmin Using Docker Compose - Vishal Sharma - Medium.” Medium, 20 Apr. 2023, medium.com/@vishal.sharma./run-postgresql-and-pgadmin-using-docker-compose-34120618bcf9. Accessed 21 May 2025.

[3] - “Seeding | Prisma Documentation.” Prisma.io, 2025, www.prisma.io/docs/orm/prisma-migrate/workflows/seeding. Accessed 21 May 2025
.
[4] - Srikanth, Akshaya. “Creating a REST API with Node.js and Express.” Postman Blog, 22 May 2023, blog.postman.com/how-to-create-a-rest-api-with-node-js-and-express/.

[5] - “Introduction to Open Food Facts API Documentation - Product Opener (Open Food Facts Server).” Openfoodfacts.github.io, openfoodfacts.github.io/openfoodfacts-server/api/.

[6] - Mangal , Nemi . “15 Advantages of React Native for Mobile App Development.” WDP Technologies Pvt. Ltd., 10 July 2024, www.wdptechnologies.com/advantages-of-react-native/. Accessed 21 May 2025.

[7] - Docs, Expo. “Camera.” Expo Documentation, docs.expo.dev/versions/latest/sdk/camera. Accessed 21 May 2025.

[8] - Docs, Expo. “Core Concepts of File-Based Routing.” Expo Documentation, 7 Apr. 2025, docs.expo.dev/router/basics/core-concepts/. Accessed 21 May 2025.

[9 - Frost, Brad. “Atomic Design Methodology | Atomic Design by Brad Frost.” Atomicdesign.bradfrost.com, 2016, atomicdesign.bradfrost.com/chapter-2/. Accessed 21 May 2025.

[10] - Wieruch, Robin. “React Folder Structure in 5 Steps [2025].” Robinwieruch.de, 8 Jan. 2025, www.robinwieruch.de/react-folder-structure/. Accessed 21 May 202

[11] - Kimtai Developer. “React Typescript Authentication Guide Using Context API.” Medium, 22 July 2024, medium.com/@kimtai.developer/react-typescript-authentication-guide-using-context-api-5c82f2530eb1. Accessed 21 May 2025

[12] - Docs, Expo. “SecureStore.” Expo Documentation, docs.expo.dev/versions/latest/sdk/securestore/. Accessed 21 May 2025

[13] - Curtis, Nathan. “Color in Design Systems.” Medium, 24 Jan. 2020, medium.com/eightshapes-llc/color-in-design-systems-a1c80f65fa3. Accessed 21 May 2025.

[14] - M3. “Typography – Material Design 3.” Material Design, m3.material.io/styles/typography/applying-type. Accessed 21 May 2025.

[15] - Shah, Sanket. “The Ultimate Guide to React Button Styles.” Dhiwise.com, 6 Sept. 2024, www.dhiwise.com/post/the-ultimate-guide-to-customizing-react-button-styles. Accessed 21 May 2025.

[16] - Onur Dayıbaşı. “React Cards Examples - Enterprise React Knowledge Maps - Medium.” Medium, Enterprise React Knowledge Maps, 4 Feb. 2025, medium.com/react-digital-garden/react-cards-examples-9f5d8166e16f. Accessed 21 May 2025.

[17] - AWS. “What Is Service-Oriented Architecture? Service-Oriented Architecture Explained - AWS.” Amazon Web Services, Inc., 2024, aws.amazon.com/what-is/service-oriented-architecture/. Accessed 21 May 2025.

[18] - Fritz. “Getting Started with React Native and Expo Using Hooks 2024 - Fritz Ai.” Fritz Ai, Fritz.ai, 18 Sept. 2023, fritz.ai/getting-started-with-react-native-and-expo-using-hooks/. Accessed 21 May 2025.

[19] - “Testing · React Native.” Reactnative.dev, 14 Apr. 2025, reactnative.dev/docs/testing-overview.

[20] - “Go to GoGuardian App.” Stackoverflow.com, 2025, stackoverflow.com/questions/520064/what-are-the-differences-between-unit-tests-integration-tests-smoke-tests-and. Accessed 21 May 2025.

[21] - “31.1. Running the Tests.” PostgreSQL Documentation, 8 May 2025, www.postgresql.org/docs/current/regress-run.html. Accessed 21 May 2025.

[22] - Zanini, Antonello. “Unit Testing in Node.js with Jest.” Appsignal.com, AppSignal, 27 Nov. 2024, blog.appsignal.com/2024/11/27/unit-testing-in-nodejs-with-jest.html. Accessed 21 May 2025.

[23] - “Creating PostgreSQL Service Containers - GitHub Docs.” GitHub Docs, 2025, docs.github.com/en/actions/use-cases-and-examples/using-containerized-services/creating-postgresql-service-containers. Accessed 21 May 2025.

[24] - Datadog. “Tests Continus et GitHub Actions Pour La CI.” Surveillance d’Applications et d’Infrastructures Avec Datadog, 2025, docs.datadoghq.com/fr/continuous_testing/cicd_integrations/github_actions/. Accessed 21 May 2025.

[25] - Ledezma, Marco . “A Comprehensive Guide to React Native, Jest and GitLab CI/CD.” Evolvingweb.com, 2025,

evolvingweb.com/blog/comprehensive-guide-react-native-jest-and-gitlab-cicd. Accessed 21 May 2025

[26] - Loveall, Edward. “Mocking React Components with Jest.” Thoughtbot, 24 Sept. 2019, thoughtbot.com/blog/mocking-react-components-with-jest.

[27] - “End to End Testing Using SuperTest | Faisal Khatri.” Github.io, 2 Feb. 2022, mfaisalkhatri.github.io/2022/02/02/endtoendtestingusingsupertest/. Accessed 21 May 2025.

Schémas de l’application

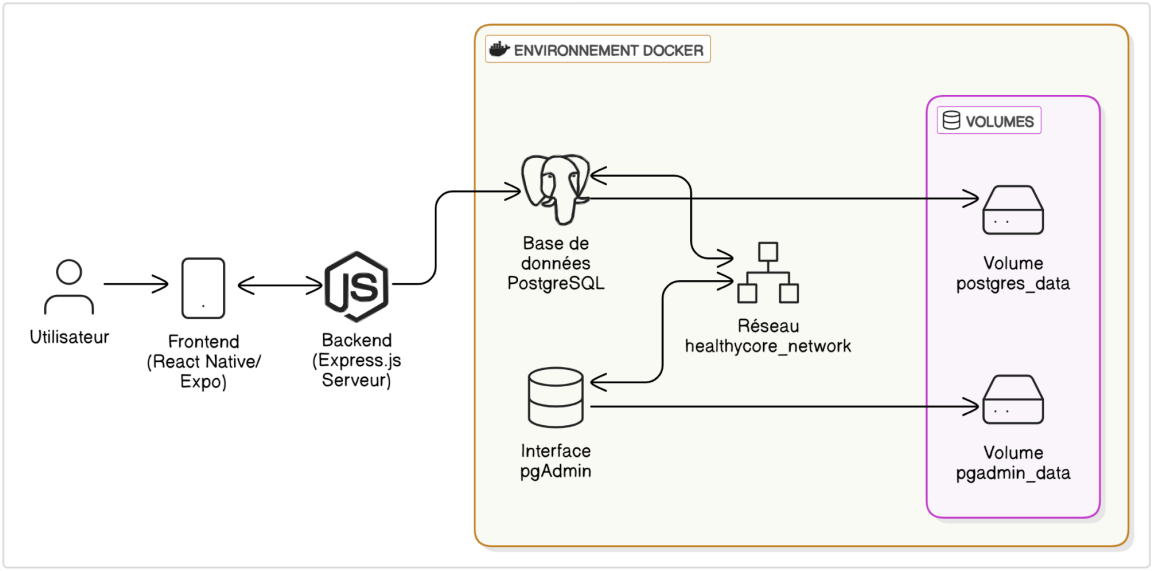

Schéma 1 : Architecture de déploiement

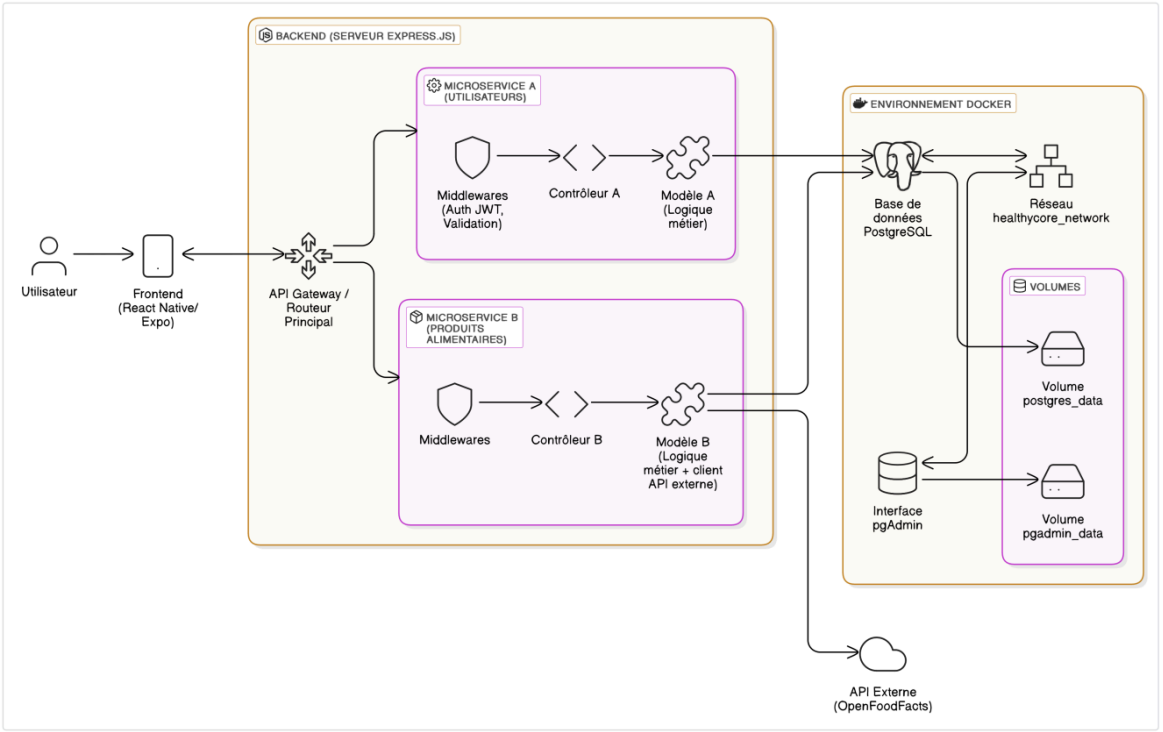

Schéma 2 : Architecture du backend (microservices)

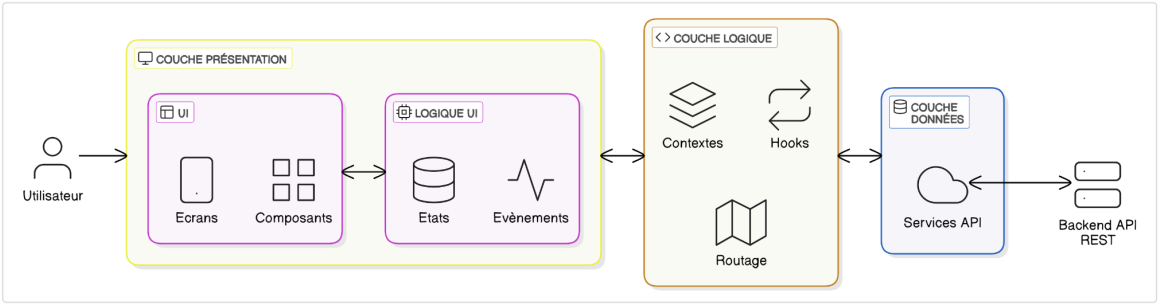

Schéma 3 : Architecture Expo frontend (en couche)