### **PROMPT DE TÂCHE : BE-001.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

### **ID de la Tâche :** BE-001.1 **Titre :** user-service : Définition des Modèles de Données Initiaux

### **IA Spécialiste Assignée :** IA #1 - User & Core

### **--- Contexte et État Actuel ---** C'est le tout début du projet. Le Sprint 1 vient de commencer. Le répertoire apps/backend/user-service/ a été créé mais est actuellement vide, à l'exception d'un Dockerfile de base. Il n'existe aucune structure de code, aucun modèle de données, ni aucune connexion à une base de données.

### **--- Objectif de la Tâche ---** Votre objectif est de créer la toute première couche de notre application : la définition des entités de données qui seront manipulées par le user-service.

### Créer la structure de dossiers de base pour une application FastAPI à l'intérieur de apps/backend/user-service/, incluant au minimum les répertoires app/ et app/models/.

### Dans un nouveau fichier app/models/user\_models.py, définir les modèles de données en utilisant la librairie SQLModel.

### Créer un modèle User qui représente la table users. Ce modèle doit inclure les champs spécifiés dans le CDC Données, tels que id (UUID), email (unique), hashed\_password, role (Enum), is\_active, etc.

### Créer un modèle CompanyProfile qui représente la table company\_profiles. Ce modèle doit inclure les champs spécifiés dans le CDC Données, comme company\_name, website, etc.

### Établir la relation One-to-One entre User et CompanyProfile en utilisant une clé étrangère (foreign key) et l'attribut relationship de SQLModel. Un utilisateur de type "entreprise" doit avoir un profil d'entreprise, mais un profil ne peut appartenir qu'à un seul utilisateur.

### Créer les schémas Pydantic (qui peuvent être dérivés des modèles SQLModel) nécessaires pour les futures opérations API, comme UserCreate, UserRead, et UserUpdate.

### **--- Prérequis et Dépendances ---**

### **Tâche(s) précédente(s) complétée(s) :** Aucune. C'est la première tâche de code du projet.

### **Fichier(s) clé(s) à lire avant de commencer :** Aucun fichier de code n'est nécessaire. La seule source de vérité est la documentation.

### **Information(s) clé(s) nécessaire(s) :** Le projet utilise SQLModel pour combiner les fonctionnalités de SQLAlchemy (ORM) et Pydantic (validation de données).

### **--- Références aux Documents Clés ---**

### **Source de Vérité Absolue :** CDC Données (EDR), Partie 3 : Dictionnaire de Données Détaillé (pour les tables users et company\_profiles).

### **--- Livrables Attendus ---**

### La structure de dossiers de base créée dans apps/backend/user-service/.

### Le contenu complet du nouveau fichier apps/backend/user-service/app/models/user\_models.py. Ce fichier doit contenir la définition des modèles User et CompanyProfile, ainsi que les schémas Pydantic associés.

### Un rapport de fin de tâche complet.

### 

### 

### **PROMPT DE TÂCHE : BE-001.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

ID de la Tâche : BE-001.2 Titre : user-service : Configuration Initiale d'Alembic

IA Spécialiste Assignée : IA #1 - User & Core

--- Contexte et État Actuel --- La sous-tâche BE-001.1 a été complétée avec succès. Le service user-service dispose maintenant d'un fichier app/models/user\_models.py contenant la définition de nos modèles de données User et CompanyProfile.

Cependant, il n'existe aucun mécanisme pour traduire ces modèles Python en tables réelles dans une base de données. Nous avons besoin d'un outil de migration de base de données pour gérer l'évolution de notre schéma.

--- Objectif de la Tâche --- Votre objectif est d'initialiser et de configurer Alembic, l'outil de migration de base de données standard pour SQLAlchemy, afin qu'il soit conscient de nos modèles SQLModel.

Initialiser un environnement Alembic à la racine du service user-service (apps/backend/user-service/). Cela créera un répertoire alembic/ et un fichier de configuration alembic.ini.

Modifier le fichier alembic.ini pour pointer vers l'URL de notre base de données de développement. Cette URL doit être lue à partir des variables d'environnement pour éviter de la coder en dur.

Modifier le fichier alembic/env.py. C'est l'étape la plus critique. Vous devez importer le target\_metadata de nos modèles SQLModel (défini dans user\_models.py) et l'assigner à la variable target\_metadata dans env.py.

Cette connexion garantit qu'Alembic pourra "voir" nos modèles et les comparer à l'état actuel de la base de données pour générer automatiquement les scripts de migration.

S'assurer que le chemin d'accès aux modèles est correctement configuré pour que le script env.py puisse les importer sans erreur.

--- Prérequis et Dépendances ---

Tâche(s) précédente(s) complétée(s) : BE-001.1.

Fichier(s) clé(s) à lire avant de commencer : apps/backend/user-service/app/models/user\_models.py.

Information(s) clé(s) nécessaire(s) : Alembic a besoin de connaître le MetaData de SQLAlchemy pour fonctionner. SQLModel expose cet objet via SQLModel.metadata.

--- Références aux Documents Clés ---

CDC Données (EDR), Partie 7 : Gestion des Versions et des Migrations.

--- Livrables Attendus ---

Le contenu complet du fichier de configuration apps/backend/user-service/alembic.ini mis à jour.

Le contenu complet du script de configuration apps/backend/user-service/alembic/env.py mis à jour, montrant l'importation et l'assignation correcte du target\_metadata.

Un rapport de fin de tâche complet.

### 

### **PROMPT DE TÂCHE : BE-001.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-001.3 **Titre :** user-service : Génération de la Première Migration de Base de Données

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** La sous-tâche BE-001.2 a été complétée avec succès. Le service user-service dispose maintenant de modèles de données définis dans app/models/user\_models.py et d'une configuration Alembic fonctionnelle et correctement connectée à ces modèles.

Cependant, aucune "version" de base de données n'a encore été créée. Il n'existe aucun script pour traduire nos modèles Python en tables SQL. La base de données, bien que conceptuellement définie, est vide.

**--- Objectif de la Tâche ---** Votre objectif est de générer le tout premier script de migration pour notre base de données en utilisant la puissance de détection automatique d'Alembic.

1. Utiliser la fonctionnalité --autogenerate d'Alembic pour comparer l'état actuel de nos modèles SQLModel (le target\_metadata) avec l'état d'une base de données vide.
2. Générer un nouveau fichier de script de migration en Python.
3. Ce script doit contenir les instructions op.create\_table() nécessaires pour créer les tables users et company\_profiles avec toutes leurs colonnes, types, contraintes de clé primaire et de clé étrangère, conformément aux modèles définis dans la tâche BE-001.1.
4. Le script doit également contenir la logique inverse dans la fonction downgrade() pour supprimer ces tables.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-001.1, BE-001.2
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/user-service/alembic/env.py, apps/backend/user-service/app/models/user\_models.py
* **Information(s) clé(s) nécessaire(s) :** La configuration d'Alembic doit être correcte pour que la détection automatique des modèles fonctionne sans erreur.

**--- Références aux Documents Clés ---**

* CDC Données (EDR), Partie 7 : Gestion des Versions et des Migrations

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier de script de migration Python qui sera généré par Alembic. Ce fichier sera situé dans apps/backend/user-service/alembic/versions/ et aura un nom de fichier unique généré automatiquement (ex: 1a2b3c4d5e6f\_create\_initial\_user\_and\_company\_tables.py). Vous devez fournir le contenu de ce fichier.
2. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-001.4**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-001.4 **Titre :** user-service : Création des Fonctions CRUD de Base pour les Utilisateurs

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** Les sous-tâches BE-001.1 à BE-001.3 sont complétées. Nous disposons maintenant de modèles de données SQLModel (User, CompanyProfile) et de schémas Pydantic (UserCreate, UserRead) dans le fichier app/models/user\_models.py. Nous avons également un script de migration Alembic prêt à créer les tables correspondantes.

Cependant, il n'existe actuellement aucune fonction pour interagir avec ces tables de manière structurée et réutilisable. Nous avons besoin de créer une couche d'abstraction (souvent appelée couche CRUD ou Repository) pour séparer la logique de base de données de la logique des routes API que nous créerons plus tard.

**--- Objectif de la Tâche ---** Votre objectif est de créer un module Python contenant les fonctions pour les opérations de base (Create, Read, Update, Delete) sur l'entité User.

1. Créer un nouveau répertoire app/crud/ à l'intérieur du user-service (apps/backend/user-service/app/crud/).
2. Dans ce nouveau répertoire, créer un fichier nommé user\_crud.py.
3. Dans user\_crud.py, implémenter les fonctions de base pour manipuler les utilisateurs dans la base de données.
4. Chaque fonction doit prendre une session de base de données (Session de SQLModel/SQLAlchemy) comme premier argument et utiliser nos modèles Pydantic/SQLModel pour les entrées et les sorties.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-001.1
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/user-service/app/models/user\_models.py
* **Information(s) clé(s) nécessaire(s) :** La structure exacte des modèles User, UserCreate et UserRead est requise pour typer correctement les arguments et les retours des fonctions.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 5 : Logique Métier et Services (qui mentionne le principe de séparation des couches).

**--- Livrables Attendus ---**

1. La création du répertoire apps/backend/user-service/app/crud/.
2. Le contenu complet du nouveau fichier user\_crud.py.
3. Ce fichier doit contenir au minimum les fonctions suivantes, entièrement typées et fonctionnelles :
   * get\_user\_by\_email(db: Session, email: str) -> Optional[User]
   * get\_user(db: Session, user\_id: UUID) -> Optional[User]
   * create\_user(db: Session, \*, user\_in: UserCreate, hashed\_password: str) -> User
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-002.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-002.1 **Titre :** project-service : Définition des Modèles de Données (Project, Skill, et entités liées)

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** Le projet a démarré. En parallèle, l'IA #1 travaille sur le user-service. Nous considérons donc que le modèle User existe et qu'il sera disponible dans la base de données. Votre travail doit être cohérent avec cette entité existante, notamment pour les clés étrangères.

Le répertoire de votre service, apps/backend/project-service/, a été créé mais est actuellement vide. Ce service sera le cœur de la logique métier de la plateforme, gérant les projets, les compétences associées, les livrables soumis par les utilisateurs, et les évaluations générées.

**--- Objectif de la Tâche ---** Votre objectif est de créer le fichier Python unique qui définit tous les modèles de données relatifs à la gestion d'un projet.

1. Créer un nouveau répertoire app/models/ à l'intérieur du project-service.
2. Dans ce répertoire, créer un nouveau fichier nommé project\_models.py.
3. Dans ce fichier, implémenter les classes SQLModel pour les tables suivantes : Project, Skill, ProjectSkill (la table de liaison many-to-many), Milestone, Deliverable, et Evaluation.
4. Définir scrupuleusement toutes les colonnes, leurs types (y compris Vector pour les embeddings et JSONB pour les données de feedback), les contraintes et les relations (ForeignKey, Relationship) en vous basant sur le CDC Données.
5. Implémenter la relation many-to-many entre Project et Skill en utilisant la table de liaison ProjectSkill et l'attribut link\_model de Relationship.
6. Inclure la définition des Enum Python nécessaires (ProjectStatus, DeliverableType, MilestoneStatus).
7. Créer les schémas Pydantic de base pour l'entité Project : ProjectCreate, ProjectRead, ProjectReadWithDetails (incluant les compétences), et ProjectUpdate.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** Aucune dépendance directe, mais le travail se fait en parallèle de BE-001.
* **Fichier(s) clé(s) à lire avant de commencer :** Aucun fichier de code, votre source de vérité est la documentation.
* **Information(s) clé(s) nécessaire(s) :** La structure des tables doit être strictement conforme au CDC Données. Le modèle Project doit avoir une colonne owner\_id qui sera une clé étrangère vers la table users (créée dans le cadre de BE-001).

**--- Références aux Documents Clés ---**

* **Source de Vérité Absolue :** CDC Données (EDR), Partie 3 : Dictionnaire de Données Détaillé.
* CDC Données (EDR), Partie 5.2 : Types Énumérés (ENUM).
* CDC Back-End, Partie 4 : Modélisation des Données (ORM).

**--- Livrables Attendus ---**

1. La création de la structure de répertoires apps/backend/project-service/app/models/.
2. Le contenu complet du nouveau fichier project\_models.py.
3. Le fichier doit contenir toutes les classes SQLModel et Enum listées dans les objectifs.
4. Le fichier doit contenir les schémas Pydantic de base pour l'entité Project.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-002.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-002.2 **Titre :** project-service : Initialisation et Configuration d'Alembic

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** La sous-tâche BE-002.1 est terminée. Le fichier apps/backend/project-service/app/models/project\_models.py a été créé et contient les définitions des modèles SQLModel pour Project, Skill, Deliverable et toutes les entités liées.

Comme pour le user-service, ce service a besoin de son propre mécanisme de migration indépendant pour gérer l'évolution de son schéma de base de données. L'outil Alembic doit être initialisé et configuré spécifiquement pour ce service.

**--- Objectif de la Tâche ---** Votre objectif est d'initialiser et de configurer Alembic pour le project-service afin de permettre la gestion versionnée de ses tables en base de données.

1. Initialiser Alembic à la racine du service project-service (apps/backend/project-service/) pour créer la structure de migration (alembic init alembic).
2. Configurer le fichier alembic.ini pour que l'URL de la base de données soit lue à partir de la variable d'environnement DATABASE\_URL.
3. Modifier le script alembic/env.py pour qu'il reconnaisse et cible les métadonnées de tous les modèles SQLModel définis dans le fichier app/models/project\_models.py.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-002.1
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/project-service/app/models/project\_models.py
* **Information(s) clé(s) nécessaire(s) :** L'existence et l'importabilité des modèles SQLModel dans le fichier mentionné ci-dessus sont indispensables pour la configuration de env.py.

**--- Références aux Documents Clés ---**

* CDC Données (EDR), Partie 7 : Gestion des Versions et des Migrations

**--- Livrables Attendus ---**

1. Le répertoire alembic/ généré à la racine de apps/backend/project-service/.
2. Le fichier alembic.ini de ce service, correctement modifié.
3. Le fichier alembic/env.py de ce service, modifié pour importer les modèles de project\_models.py et assigner SQLModel.metadata à target\_metadata.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-002.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-002.3 **Titre :** project-service : Génération de la Migration Initiale des Tables

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** La sous-tâche BE-002.2 a été complétée avec succès. Le project-service dispose maintenant de modèles de données complets et d'une configuration Alembic fonctionnelle qui est consciente de ces modèles.

Cependant, aucun script de migration n'a encore été généré pour les tables de ce service (projects, skills, deliverables, etc.). La base de données ne contient pas encore la structure nécessaire pour stocker les informations relatives au cœur métier de l'application.

**--- Objectif de la Tâche ---** Votre objectif est de générer le script de migration initial qui créera toutes les tables gérées par le project-service.

1. Exécuter la commande alembic revision --autogenerate pour que Alembic compare les modèles définis dans project\_models.py à l'état vide de la base de données.
2. Produire un nouveau script de migration Python qui en résulte.
3. Vérifier que ce script généré contient toutes les opérations op.create\_table() nécessaires pour les tables : projects, skills, project\_skills, milestones, deliverables, et evaluations.
4. Confirmer que toutes les colonnes, leurs types de données spécifiques (particulièrement Vector et JSONB), les clés primaires, les clés étrangères (notamment vers la table users) et les relations sont correctement représentés dans le script.
5. S'assurer que la fonction downgrade() du script contient les opérations op.drop\_table() correspondantes, dans l'ordre inverse de la création pour respecter les contraintes de clés étrangères.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-002.1, BE-002.2
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/project-service/alembic/env.py, apps/backend/project-service/app/models/project\_models.py
* **Information(s) clé(s) nécessaire(s) :** La configuration d'Alembic doit être capable de détecter la totalité des modèles importés pour générer une migration complète et correcte.

**--- Références aux Documents Clés ---**

* CDC Données (EDR), Partie 7 : Gestion des Versions et des Migrations

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier de script de migration Python généré par Alembic. Ce fichier sera situé dans apps/backend/project-service/alembic/versions/ et aura un titre descriptif comme "Create project related tables".
2. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-002.4**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-002.4 **Titre :** project-service : Création des Fonctions CRUD de Base pour les Projets et Compétences

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** Les sous-tâches BE-002.1 à BE-002.3 sont complétées. Le project-service dispose de modèles de données SQLModel et d'un script de migration prêt à créer les tables projects, skills, et leurs tables liées.

Cependant, il n'existe aucune fonction de haut niveau pour interagir avec ces tables. Nous avons besoin d'une couche d'abstraction (CRUD/Repository) pour encapsuler la logique de base de données, notamment pour la création d'un projet et la gestion de sa relation many-to-many avec les compétences.

**--- Objectif de la Tâche ---** Votre objectif est de créer un module Python contenant les fonctions pour les opérations de base (Create, Read) sur les entités Project et Skill.

1. Créer un nouveau répertoire app/crud/ à l'intérieur du project-service (apps/backend/project-service/app/crud/).
2. Dans ce répertoire, créer un nouveau fichier nommé project\_crud.py.
3. Dans project\_crud.py, implémenter les fonctions CRUD essentielles pour les entités Project et Skill.
4. Les fonctions doivent prendre une session de base de données (Session) comme argument et utiliser les schémas Pydantic/SQLModel (ProjectCreate, etc.) pour les opérations. Une attention particulière doit être portée à la gestion de la relation many-to-many.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-002.1
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/project-service/app/models/project\_models.py
* **Information(s) clé(s) nécessaire(s) :** La structure exacte des modèles Project, Skill, ProjectSkill, ProjectCreate et ProjectRead est requise pour le typage et la logique des fonctions.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 5 : Logique Métier et Services

**--- Livrables Attendus ---**

1. La création du répertoire apps/backend/project-service/app/crud/.
2. Le contenu complet du nouveau fichier project\_crud.py.
3. Ce fichier doit contenir au minimum les fonctions suivantes, entièrement typées et fonctionnelles :
   * get\_or\_create\_skill(db: Session, skill\_in: SkillCreate) -> Skill : Une fonction qui récupère une compétence par son nom ou la crée si elle n'existe pas, pour éviter les doublons.
   * create\_project(db: Session, \*, project\_in: ProjectCreate, owner\_id: UUID) -> Project : Cette fonction doit créer une instance Project, et pour chaque compétence listée dans project\_in, utiliser get\_or\_create\_skill et lier la compétence au projet via la table ProjectSkill.
   * get\_project(db: Session, project\_id: UUID) -> Optional[Project]
   * get\_all\_projects(db: Session, skip: int = 0, limit: int = 100) -> List[Project]
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-001.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-001.1 **Titre :** frontend : Initialisation du projet Vite + React et Nettoyage

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** C'est la toute première tâche pour l'équipe Frontend. Le monorepo a été créé et les équipes Back-End commencent à travailler sur la couche de données. Le répertoire apps/frontend/ est actuellement vide.

L'objectif est de créer un squelette d'application React propre, moderne et performant en utilisant l'outil de build Vite, comme spécifié dans le CDC Front-End.

**--- Objectif de la Tâche ---** Votre objectif est de générer le projet Front-End initial et de le nettoyer pour préparer le terrain pour les développements futurs.

1. Générer un nouveau projet React + TypeScript dans le répertoire apps/frontend/ en utilisant la commande standard npm create vite@latest.
2. Supprimer tous les fichiers de style et les ressources de démonstration générés par défaut par Vite (par exemple index.css, App.css, le logo react.svg, etc.).
3. Simplifier le composant principal App.tsx pour qu'il n'affiche qu'un simple titre de bienvenue, comme <h1>Bienvenue sur SkillForge AI</h1>.
4. S'assurer que le projet se lance correctement via la commande npm run dev après ce nettoyage, sans aucune erreur.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** Aucune. C'est un point de départ.
* **Fichier(s) clé(s) à lire avant de commencer :** Aucun fichier de code n'est nécessaire.
* **Information(s) clé(s) nécessaire(s) :** Le projet doit être initialisé en utilisant le template react-ts de Vite.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 3 : Choix Technologiques (qui mandate l'utilisation de Vite, React et TypeScript).

**--- Livrables Attendus ---**

1. La structure de fichiers initiale générée par Vite dans le répertoire apps/frontend/.
2. Le contenu du fichier package.json résultant de l'initialisation.
3. Le contenu du fichier src/App.tsx après avoir été nettoyé et simplifié.
4. La liste des fichiers qui ont été supprimés par rapport au template Vite par défaut.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-001.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-001.2 **Titre :** frontend : Structure de Dossiers et Intégration de Tailwind CSS

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-001.1 est terminée. Nous disposons d'une application React + Vite de base, nettoyée de son contenu de démonstration, dans le répertoire apps/frontend/.

Actuellement, l'application n'a pas de structure de dossiers organisée pour accueillir les futurs composants, pages, et services. De plus, aucun framework de stylisme n'est installé, ce qui rendrait la création d'interfaces utilisateur incohérente et fastidieuse. Le CDC Front-End mandate l'utilisation de Tailwind CSS.

**--- Objectif de la Tâche ---** Votre objectif est de structurer le projet Front-End de manière professionnelle et d'y intégrer le framework de stylisme Tailwind CSS.

1. Créer une structure de dossiers scalable et maintenable à l'intérieur du répertoire apps/frontend/src/. Cette structure doit inclure, au minimum, les répertoires suivants : assets, components, hooks, layouts, pages, services, store, et utils.
2. Installer les dépendances de développement nécessaires pour Tailwind CSS en utilisant npm (tailwindcss, postcss, autoprefixer).
3. Exécuter la commande d'initialisation de Tailwind pour générer les fichiers tailwind.config.js et postcss.config.js.
4. Configurer le fichier tailwind.config.js pour qu'il analyse tous les fichiers sources de l'application (.js, .ts, .jsx, .tsx) dans le répertoire src afin de détecter les classes utilitaires.
5. Créer un fichier CSS principal, src/index.css, et y ajouter les trois directives @tailwind de base (base, components, utilities).
6. Importer ce fichier src/index.css dans le point d'entrée de l'application (src/main.tsx) pour activer Tailwind sur l'ensemble du projet.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-001.1
* **Fichier(s) clé(s) à lire avant de commencer :** apps/frontend/package.json, apps/frontend/src/main.tsx.
* **Information(s) clé(s) nécessaire(s) :** Le projet est une application Vite + React + TypeScript. La procédure d'installation de Tailwind CSS doit être celle recommandée pour Vite.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 4 : Organisation du Code et Structure des Dossiers
* CDC Front-End, Partie 5 : Stratégie de Stylisme et UI

**--- Livrables Attendus ---**

1. L'arborescence complète de la nouvelle structure de dossiers créée à l'intérieur de apps/frontend/src/.
2. Le contenu des fichiers de configuration générés et modifiés : tailwind.config.js et postcss.config.js.
3. Le contenu du fichier CSS principal src/index.css.
4. Le contenu mis à jour du fichier src/main.tsx montrant l'importation du fichier CSS.
5. Le fichier package.json mis à jour, montrant les nouvelles devDependencies.
6. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-001.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-001.3 **Titre :** frontend : Création du Layout Principal et du système de Routage

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-001.2 est terminée. L'application dispose d'une structure de dossiers propre et le framework de stylisme Tailwind CSS est correctement intégré.

Cependant, l'application est actuellement monolithique et n'affiche qu'un seul composant racine (App.tsx). Il n'existe aucun mécanisme pour naviguer entre différentes "pages" ou vues, ni de structure visuelle persistante (comme une barre de navigation ou un pied de page) qui encadrerait le contenu.

**--- Objectif de la Tâche ---** Votre objectif est de mettre en place le système de routage de l'application et de créer un layout principal pour assurer une structure de page cohérente.

1. Installer la librairie de routage react-router-dom via npm.
2. Créer un composant de layout principal MainLayout.tsx dans le répertoire apps/frontend/src/layouts/. Ce layout devra contenir des placeholders pour une future Navbar et un Footer, et utiliser le composant <Outlet /> de react-router-dom pour afficher le contenu de la page active.
3. Créer des composants de page de base qui serviront de placeholders, comme HomePage.tsx et LoginPage.tsx, dans le répertoire apps/frontend/src/pages/.
4. Configurer le système de routage de l'application. Il est recommandé de le faire dans le fichier App.tsx en utilisant la fonction createBrowserRouter de react-router-dom pour définir les routes (ex: la route / pointe vers HomePage, /login vers LoginPage, etc.), en utilisant MainLayout comme route parente.
5. Intégrer le routeur dans le point d'entrée de l'application (main.tsx) en utilisant le composant <RouterProvider /> pour activer la navigation dans toute l'application.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-001.2
* **Fichier(s) clé(s) à lire avant de commencer :** La structure de dossiers existante dans apps/frontend/src/.
* **Information(s) clé(s) nécessaire(s) :** Le système de routage doit utiliser les conventions modernes de react-router-dom (v6.4+).

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 4 : Organisation du Code et Structure des Dossiers
* CDC Front-End, Partie 6 : Routage (qui mandate react-router-dom).

**--- Livrables Attendus ---**

1. Le fichier package.json mis à jour avec la nouvelle dépendance react-router-dom.
2. Le contenu du nouveau fichier src/layouts/MainLayout.tsx.
3. Le contenu des nouveaux fichiers de page, par exemple src/pages/HomePage.tsx et src/pages/LoginPage.tsx.
4. Le contenu du fichier src/App.tsx (ou un fichier de routage dédié) contenant la configuration createBrowserRouter.
5. Le contenu mis à jour du fichier src/main.tsx qui rend le <RouterProvider />.
6. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-003.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-003.1 **Titre :** user-service : Utilitaires de Sécurité et Dépendance de Base de Données

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** L'Epic BE-001 est terminé. La couche de données du user-service (modèles, CRUD, migrations) est en place. Nous démarrons le Sprint 2, dont l'objectif est d'exposer cette logique via une API REST.

Avant de pouvoir créer les endpoints d'inscription et de connexion, nous devons mettre en place deux briques techniques fondamentales et réutilisables :

1. Un module pour la gestion de la sécurité (hachage de mots de passe, création de jetons JWT).
2. Le système d'injection de dépendances de FastAPI pour fournir les sessions de base de données à nos routes.

**--- Objectif de la Tâche ---** Votre objectif est de créer les modules de support pour la sécurité et l'accès à la base de données.

1. Installer la librairie passlib avec le support de bcrypt et l'ajouter au fichier requirements.txt du user-service.
2. Créer un nouveau module app/core/security.py.
3. Dans security.py, implémenter deux fonctions : get\_password\_hash(password: str) -> str et verify\_password(plain\_password: str, hashed\_password: str) -> bool.
4. Dans ce même fichier, implémenter la fonction create\_access\_token(data: dict) -> str pour générer les jetons JWT en utilisant les variables d'environnement JWT\_SECRET\_KEY et JWT\_ALGORITHM.
5. Créer un module app/db/session.py qui initialise le moteur (engine) SQLModel/SQLAlchemy en utilisant la DATABASE\_URL des variables d'environnement.
6. Créer un module app/api/dependencies.py qui contiendra une fonction de dépendance FastAPI, get\_db(), fournissant une session de base de données par requête, en suivant le pattern officiel de FastAPI.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** Toutes les tâches de l'Epic BE-001.
* **Fichier(s) clé(s) à lire avant de commencer :** Le fichier .env.template pour comprendre comment les variables (DATABASE\_URL, JWT\_SECRET\_KEY) sont nommées.
* **Information(s) clé(s) nécessaire(s) :** Le projet utilise FastAPI et SQLModel. Les librairies à utiliser sont passlib pour le hachage et python-jose pour les JWT (à ajouter aux dépendances si ce n'est pas déjà fait).

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 6 : Sécurité et Authentification
* Documentation officielle de FastAPI sur la sécurité et les dépendances de base de données.

**--- Livrables Attendus ---**

1. Le fichier apps/backend/user-service/requirements.txt mis à jour avec passlib[bcrypt] et python-jose.
2. Le contenu complet du nouveau fichier app/core/security.py.
3. Le contenu complet du nouveau fichier app/db/session.py.
4. Le contenu complet du nouveau fichier app/api/dependencies.py.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-003.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-003.2 **Titre :** user-service : Implémentation de l'Endpoint d'Inscription

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** La sous-tâche BE-003.1 est terminée. Nous disposons des utilitaires pour hacher les mots de passe et créer des jetons JWT (core/security.py), ainsi qu'un système de dépendance FastAPI pour injecter les sessions de base de données (api/dependencies.py). La couche CRUD (crud/user\_crud.py) est également prête.

Toutes les briques sont en place pour créer notre première route d'API qui écrit des données en base de données de manière sécurisée.

**--- Objectif de la Tâche ---** Votre objectif est de créer l'endpoint d'API REST qui permet à un nouvel utilisateur de s'inscrire sur la plateforme.

1. Créer un nouveau fichier de routeur dans apps/backend/user-service/app/api/routes/users.py.
2. Dans ce fichier, implémenter un endpoint POST /register qui accepte les données d'un nouvel utilisateur via le schéma Pydantic UserCreate.
3. La logique de l'endpoint doit d'abord vérifier si un utilisateur avec le même email existe déjà en utilisant la fonction get\_user\_by\_email. Si c'est le cas, il doit lever une HTTPException avec un code de statut 400.
4. Si l'email est unique, l'endpoint doit hacher le mot de passe fourni en utilisant la fonction get\_password\_hash.
5. Ensuite, il doit appeler la fonction create\_user de la couche CRUD pour sauvegarder le nouvel utilisateur en base de données avec son mot de passe haché.
6. L'endpoint doit retourner les données du nouvel utilisateur créé, en utilisant le schéma UserRead pour s'assurer que le mot de passe haché n'est jamais exposé.
7. Intégrer ce nouveau routeur à l'application FastAPI principale dans app/main.py.
8. Écrire les tests d'intégration correspondants dans un nouveau fichier tests/api/test\_users.py, en couvrant au minimum le cas d'une inscription réussie et le cas d'une tentative d'inscription avec un email déjà existant.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-001.4, BE-003.1.
* **Fichier(s) clé(s) à lire avant de commencer :** app/models/user\_models.py (pour les schémas UserCreate, UserRead), app/crud/user\_crud.py, et app/core/security.py.
* **Information(s) clé(s) nécessaire(s) :** L'endpoint doit être inclus dans un APIRouter avec un préfixe (ex: /users) et des tags (ex: ["Users"]) pour une documentation OpenAPI claire.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 3 : Spécification des Microservices (pour le contrat d'API du user-service).

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier apps/backend/user-service/app/api/routes/users.py.
2. Les modifications apportées au fichier apps/backend/user-service/app/main.py pour inclure le nouveau routeur.
3. Le contenu complet du nouveau fichier de test apps/backend/user-service/tests/api/test\_users.py.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-003.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-003.3 **Titre :** user-service : Implémentation de l'Endpoint de Connexion et de Génération de Token

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** La sous-tâche BE-003.2 est terminée. L'endpoint d'inscription /register est fonctionnel et testé, permettant la création de nouveaux utilisateurs dans la base de données avec des mots de passe hachés.

Cependant, il n'existe actuellement aucun moyen pour un utilisateur enregistré de s'authentifier pour prouver son identité et obtenir un jeton d'accès (JWT) qui sera utilisé pour les requêtes futures.

**--- Objectif de la Tâche ---** Votre objectif est de créer l'endpoint de connexion (login) conforme au standard OAuth2 "Password Flow", qui authentifie un utilisateur et retourne un jeton d'accès.

1. Créer un nouveau fichier app/models/token\_models.py et y définir un schéma Pydantic Token contenant deux champs : access\_token: str et token\_type: str.
2. Dans le fichier de routeur existant (app/api/routes/users.py), ajouter un nouvel endpoint POST /token.
3. Cet endpoint ne doit pas attendre un corps de requête JSON, mais doit utiliser la dépendance OAuth2PasswordRequestForm de FastAPI pour accepter les données de formulaire username (qui sera notre email) et password.
4. La logique de l'endpoint doit authentifier l'utilisateur : d'abord, récupérer l'utilisateur par son email via la fonction get\_user\_by\_email ; ensuite, comparer le mot de passe fourni avec le mot de passe haché en base de données en utilisant la fonction verify\_password.
5. En cas d'échec de l'authentification (utilisateur non trouvé ou mot de passe incorrect), l'endpoint doit lever une HTTPException avec un code de statut 401 Unauthorized et l'en-tête WWW-Authenticate: Bearer.
6. En cas de succès, l'endpoint doit générer un nouveau jeton d'accès JWT en utilisant la fonction create\_access\_token, puis retourner une réponse contenant le jeton au format du schéma Token.
7. Ajouter les tests d'intégration correspondants dans le fichier tests/api/test\_users.py, couvrant le cas d'une connexion réussie, d'une connexion avec un mot de passe erroné, et d'une connexion avec un utilisateur inexistant.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-001.4, BE-003.1, BE-003.2
* **Fichier(s) clé(s) à lire avant de commencer :** app/crud/user\_crud.py, app/core/security.py.
* **Information(s) clé(s) nécessaire(s) :** L'utilisation de OAuth2PasswordRequestForm = Depends() est une exigence pour la conformité au standard et la compatibilité avec les outils de documentation automatique d'API.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 6 : Sécurité et Authentification

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier apps/backend/user-service/app/models/token\_models.py.
2. Les modifications apportées au fichier apps/backend/user-service/app/api/routes/users.py pour ajouter le nouvel endpoint /token.
3. Les tests d'intégration supplémentaires ajoutés au fichier apps/backend/user-service/tests/api/test\_users.py.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-003.4**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-003.4 **Titre :** user-service : Implémentation d'un Endpoint Protégé (/users/me)

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** Les sous-tâches BE-003.2 et BE-003.3 sont terminées. Les utilisateurs peuvent s'inscrire pour créer un compte et se connecter pour obtenir un jeton d'accès JWT valide.

Cependant, il n'existe actuellement aucune route d'API qui utilise ce jeton pour authentifier une requête et retourner des données spécifiques à l'utilisateur connecté. Le jeton est généré mais n'est pas encore exploité pour sécuriser l'accès aux ressources.

**--- Objectif de la Tâche ---** Votre objectif est de créer un endpoint protégé qui ne peut être accédé que par un utilisateur authentifié, et qui lui retourne ses propres informations.

1. Créer une nouvelle dépendance FastAPI get\_current\_user dans le module app/api/dependencies.py.
2. Cette dépendance doit extraire le jeton JWT de l'en-tête Authorization de la requête (en utilisant le schéma OAuth2PasswordBearer).
3. Elle doit ensuite décoder et valider ce jeton en utilisant les fonctions et secrets définis dans core/security.py.
4. En cas de jeton invalide ou expiré, la dépendance doit lever une HTTPException avec un code de statut 401 Unauthorized.
5. Si le jeton est valide, la dépendance doit utiliser l'identifiant de l'utilisateur (contenu dans le sub du jeton) pour récupérer l'objet User correspondant depuis la base de données.
6. Créer un nouvel endpoint GET /users/me dans le routeur users.py.
7. Protéger cet endpoint en l'injectant avec la nouvelle dépendance get\_current\_user. Le corps de la fonction de cet endpoint doit simplement retourner l'objet User fourni par la dépendance.
8. Ajouter les tests d'intégration correspondants dans tests/api/test\_users.py, couvrant l'accès avec un token valide, l'accès avec un token invalide, et l'accès sans token.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-003.3
* **Fichier(s) clé(s) à lire avant de commencer :** app/core/security.py (pour la logique de décodage JWT), app/crud/user\_crud.py (pour récupérer l'utilisateur depuis la BDD).
* **Information(s) clé(s) nécessaire(s) :** La dépendance doit utiliser le schéma OAuth2PasswordBearer pointant vers l'URL /token pour une intégration correcte avec la documentation OpenAPI.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 6 : Sécurité et Authentification

**--- Livrables Attendus ---**

1. Les modifications apportées au fichier apps/backend/user-service/app/api/dependencies.py (ou app/core/security.py) pour ajouter la fonction get\_current\_user.
2. Les modifications apportées au fichier apps/backend/user-service/app/api/routes/users.py pour ajouter le nouvel endpoint protégé /users/me.
3. Les tests d'intégration supplémentaires ajoutés au fichier apps/backend/user-service/tests/api/test\_users.py.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-004.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-004.1 **Titre :** project-service : Initialisation de l'API (main, dépendances, routeur)

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** L'Epic BE-002 (couche de données du project-service) est terminé. Nous disposons des modèles, des fonctions CRUD et des scripts de migration. Le Sprint 1 pour ce service est donc achevé.

Nous démarrons maintenant le Sprint 2 pour le project-service. L'objectif est de construire la couche API REST au-dessus de la couche de données. Actuellement, le service ne contient aucun code pour démarrer un serveur web ou définir des endpoints.

**--- Objectif de la Tâche ---** Votre objectif est de créer le squelette de l'application FastAPI pour le project-service, en répliquant la structure de base déjà mise en place pour le user-service afin de garantir la cohérence architecturale.

1. Créer le fichier principal de l'application apps/backend/project-service/app/main.py et y initialiser une instance de FastAPI.
2. Mettre en place le système d'injection de dépendance pour la base de données, en créant les fichiers app/db/session.py et app/api/dependencies.py. La logique de ces fichiers est standard et peut être répliquée depuis le user-service.
3. Créer le fichier de routeur initial dans app/api/routes/projects.py. Ce fichier doit contenir une instance vide d' APIRouter prête à accueillir les futurs endpoints.
4. Inclure ce nouveau routeur dans l'application FastAPI principale (main.py) avec un préfixe approprié (ex: /projects) et des tags (ex: ["Projects"]).

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** Toutes les tâches de l'Epic BE-002.
* **Fichier(s) clé(s) à lire avant de commencer :** La structure de dossiers du user-service pour inspiration.
* **Information(s) clé(s) nécessaire(s) :** L'objectif est d'établir une fondation API propre avant d'implémenter la logique des endpoints.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 5 : Logique Métier et Services (sur le principe d'une architecture en couches).

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier apps/backend/project-service/app/main.py.
2. Le contenu complet du nouveau fichier apps/backend/project-service/app/db/session.py.
3. Le contenu complet du nouveau fichier apps/backend/project-service/app/api/dependencies.py.
4. Le contenu complet du nouveau fichier apps/backend/project-service/app/api/routes/projects.py (contenant uniquement l'initialisation du routeur).
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-004.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-004.2 **Titre :** project-service : Implémentation de l'Endpoint de Création de Projet

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** La sous-tâche BE-004.1 est terminée. Le project-service dispose d'une application FastAPI de base avec un routeur vide. Par ailleurs, l'IA #1 a finalisé le mécanisme d'authentification complet dans le user-service.

Pour rendre nos microservices autonomes, nous devons maintenant importer et adapter cette logique d'authentification dans le project-service afin de pouvoir protéger ses endpoints. Aucun endpoint pour créer un projet n'existe encore.

**--- Objectif de la Tâche ---** Votre objectif est de créer un endpoint **protégé** qui permet à une entreprise authentifiée de créer un nouveau projet.

1. Rendre le project-service autonome en matière d'authentification. Pour cela, vous devez : a. Copier les modules core/security.py et models/token\_models.py depuis le user-service vers les répertoires correspondants du project-service. b. Copier la logique de la dépendance get\_current\_user et de OAuth2PasswordBearer dans le fichier api/dependencies.py du project-service.
2. Installer les dépendances de sécurité (passlib[bcrypt], python-jose) dans le fichier requirements.txt du project-service.
3. Dans le fichier de routeur app/api/routes/projects.py, implémenter un endpoint POST / qui sera protégé en utilisant la dépendance get\_current\_user.
4. La logique de cet endpoint doit d'abord vérifier que l'utilisateur authentifié (current\_user) a bien le rôle UserRole.COMPANY. Si ce n'est pas le cas, il doit lever une HTTPException avec un code de statut 403 (Forbidden).
5. Si l'utilisateur est autorisé, l'endpoint doit appeler la fonction create\_project de la couche CRUD pour créer le projet en base de données, en utilisant l'ID de l'utilisateur connecté comme owner\_id.
6. L'endpoint doit retourner les données du projet nouvellement créé en utilisant le schéma Pydantic ProjectRead.
7. Écrire les tests d'intégration correspondants dans un nouveau fichier tests/api/test\_projects.py, couvrant la création réussie par un utilisateur COMPANY, la tentative échouée par un utilisateur LEARNER (attendant un code 403), et la tentative sans authentification (attendant un code 401).

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-002.4, BE-004.1, et compréhension de la logique de l'Epic BE-003.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/project-service/app/crud/project\_crud.py, apps/backend/project-service/app/models/project\_models.py. Vous devez également vous inspirer des fichiers de sécurité du user-service.
* **Information(s) clé(s) nécessaire(s) :** La création de projet est une action restreinte qui nécessite une vérification de rôle (autorisation) en plus de l'authentification.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 3 : Spécification des Microservices (Contrat API pour project-service).
* CDC Back-End, Partie 6 : Sécurité et Authentification.

**--- Livrables Attendus ---**

1. Le fichier requirements.txt du project-service, mis à jour.
2. Les nouveaux fichiers de sécurité, de modèles de token et de dépendances dans le project-service.
3. Les modifications apportées au fichier apps/backend/project-service/app/api/routes/projects.py pour ajouter l'endpoint de création.
4. Le contenu complet du nouveau fichier de test apps/backend/project-service/tests/api/test\_projects.py.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-004.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-004.3 **Titre :** project-service : Implémentation des Endpoints de Lecture de Projets

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** La sous-tâche BE-004.2 est terminée. L'endpoint POST /projects est fonctionnel et permet de créer des projets de manière sécurisée.

Cependant, il n'existe actuellement aucun moyen pour les utilisateurs (qu'ils soient authentifiés ou non) de consulter la liste des projets disponibles ou de voir les détails d'un projet spécifique via l'API.

**--- Objectif de la Tâche ---** Votre objectif est de créer les deux endpoints GET publics pour la consultation des projets.

1. Dans le fichier de routeur existant (app/api/routes/projects.py), implémenter un endpoint public GET / pour lister tous les projets.
2. Cet endpoint doit accepter des paramètres de requête optionnels pour la pagination (skip: int = 0, limit: int = 100).
3. Il doit appeler la fonction get\_all\_projects de la couche CRUD et retourner une liste de projets en utilisant le schéma List[ProjectRead].
4. Implémenter un second endpoint public GET /{project\_id} pour récupérer les détails d'un projet spécifique par son ID.
5. Cet endpoint doit appeler la fonction get\_project de la couche CRUD et utiliser le schéma ProjectReadWithDetails comme modèle de réponse pour inclure les compétences associées.
6. Si un project\_id demandé ne correspond à aucun projet en base de données, l'endpoint doit lever une HTTPException avec un code de statut 404 (Not Found).
7. Ajouter les tests d'intégration pour ces deux nouveaux endpoints dans tests/api/test\_projects.py, en couvrant les cas de succès (liste et détail) et le cas d'un projet non trouvé (404).

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-002.4, BE-004.1.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/project-service/app/crud/project\_crud.py (pour les fonctions get\_\*) et apps/backend/project-service/app/models/project\_models.py (pour les schémas de réponse ProjectRead et ProjectReadWithDetails).
* **Information(s) clé(s) nécessaire(s) :** Ces endpoints sont publics et ne requièrent pas la dépendance get\_current\_user.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 3 : Spécification des Microservices (Contrat API pour project-service).

**--- Livrables Attendus ---**

1. Les modifications apportées au fichier apps/backend/project-service/app/api/routes/projects.py pour ajouter les deux endpoints GET.
2. Les tests d'intégration supplémentaires ajoutés au fichier apps/backend/project-service/tests/api/test\_projects.py.
3. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-002.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-002.1 **Titre :** frontend : Services API et Store d'Authentification

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** L'Epic FE-001 (squelette de l'application) et l'Epic BE-003 (API d'authentification) sont terminés. Nous avons une application React avec une structure de dossiers, et des endpoints /users/register et /token fonctionnels et prêts à être consommés.

L'application front-end n'a actuellement aucune logique pour communiquer avec une API externe, ni pour gérer un état global qui permettrait de savoir si un utilisateur est connecté ou non.

**--- Objectif de la Tâche ---** Votre objectif est de créer les fondations logiques côté client pour la communication avec l'API et la gestion de l'état d'authentification.

1. Installer les librairies axios (pour des requêtes HTTP robustes) et zustand (pour une gestion d'état simple et efficace) via npm.
2. Créer un service API de base dans apps/frontend/src/services/apiClient.ts. Ce service doit exporter une instance préconfigurée d'axios avec l'URL de base de notre API, lue depuis les variables d'environnement (VITE\_API\_BASE\_URL).
3. Créer un service dédié à l'authentification dans apps/frontend/src/services/authService.ts. Ce service doit utiliser apiClient et définir deux fonctions asynchrones : register(userData) et login(credentials), qui correspondront aux appels API respectifs.
4. Créer un "store" d'état global pour l'authentification dans apps/frontend/src/store/authStore.ts en utilisant Zustand.
5. Ce store doit contenir au minimum les éléments d'état suivants : token: string | null, user: UserProfile | null (définir une interface UserProfile simple), et isAuthenticated: boolean.
6. Le store doit exposer des "actions" (fonctions) pour mettre à jour l'état : par exemple loginSuccess(token, user) et logout().

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** Epic FE-001 et Epic BE-003.
* **Fichier(s) clé(s) à lire avant de commencer :** Le CDC Back-End pour connaître les structures de données attendues et retournées par les endpoints /register et /token.
* **Information(s) clé(s) nécessaire(s) :** Le projet utilise Zustand pour la gestion d'état, comme spécifié dans le CDC Front-End.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 7 : Gestion de l'État
* CDC Back-End, Partie 3 : Spécification des Microservices (user-service)

**--- Livrables Attendus ---**

1. Le fichier package.json mis à jour avec axios et zustand comme dépendances.
2. Le contenu complet du nouveau fichier src/services/apiClient.ts.
3. Le contenu complet du nouveau fichier src/services/authService.ts.
4. Le contenu complet du nouveau fichier src/store/authStore.ts.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-002.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-002.2 **Titre :** frontend : Création de la Page et du Formulaire d'Inscription

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-002.1 est terminée. Les services pour appeler l'API d'inscription (authService) et le store Zustand pour gérer l'état d'authentification (authStore) sont prêts à être utilisés.

Cependant, il n'existe aucune interface utilisateur (UI) permettant à un visiteur de saisir ses informations et de déclencher le processus d'inscription. La route /register que nous avons créée au Sprint 1 mène à une page vide.

**--- Objectif de la Tâche ---** Votre objectif est de construire le composant React pour le formulaire d'inscription et la page qui l'héberge, en utilisant des pratiques modernes de gestion de formulaire et de validation.

1. Installer les librairies react-hook-form pour la gestion de la logique du formulaire, @hookform/resolvers et zod pour la validation de schémas, via npm.
2. Dans un nouveau fichier apps/frontend/src/lib/validators.ts, définir un schéma de validation avec zod qui correspond aux données attendues par l'API pour l'inscription (UserCreate). Il doit valider le format de l'email, la complexité du mot de passe, etc.
3. Créer des composants d'interface utilisateur (UI) de base, réutilisables et stylisés avec Tailwind CSS, dans apps/frontend/src/components/ui/. Créez au minimum Input.tsx, Button.tsx, et Label.tsx.
4. Créer un composant de formulaire dédié, RegisterForm.tsx, dans apps/frontend/src/components/auth/. Ce composant doit utiliser le hook useForm de react-hook-form et le schéma zod pour gérer l'état, la validation et l'affichage des erreurs du formulaire.
5. Implémenter la fonction de soumission (onSubmit) du formulaire. Cette fonction doit appeler authService.register() avec les données du formulaire. Elle doit également gérer les états de chargement, de succès (ex: afficher une notification et rediriger vers la page de connexion) et d'erreur (ex: afficher un message d'erreur renvoyé par l'API).
6. Créer la page d'inscription finale, RegisterPage.tsx, dans le répertoire apps/frontend/src/pages/. Cette page affichera le composant RegisterForm au centre de l'écran.
7. Mettre à jour le fichier de routage (App.tsx) pour que la route /register pointe bien vers ce nouveau composant RegisterPage.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-002.1
* **Fichier(s) clé(s) à lire avant de commencer :** apps/frontend/src/services/authService.ts (pour la fonction register à appeler) et le CDC Back-End pour le contrat de données de l'API d'inscription.
* **Information(s) clé(s) nécessaire(s) :** Le projet mandate l'utilisation de react-hook-form et zod pour une gestion robuste des formulaires.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 8 : Formulaires et Validation
* CDC Back-End, Partie 3 : Spécification des Microservices (user-service)

**--- Livrables Attendus ---**

1. Le fichier package.json mis à jour avec les nouvelles dépendances (react-hook-form, @hookform/resolvers, zod).
2. Le contenu du nouveau fichier de schémas de validation src/lib/validators.ts.
3. Le code des nouveaux composants UI réutilisables dans src/components/ui/.
4. Le contenu complet du nouveau composant de formulaire src/components/auth/RegisterForm.tsx.
5. Le contenu complet de la nouvelle page src/pages/RegisterPage.tsx.
6. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-002.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-002.3 **Titre :** frontend : Création de la Page et du Formulaire de Connexion

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-002.2 est terminée. L'interface utilisateur pour l'inscription est fonctionnelle. Les utilisateurs peuvent créer un compte, mais il n'existe pas encore de formulaire ou de page leur permettant de se connecter avec leurs nouveaux identifiants. Le authService et le authStore sont prêts à gérer la logique de connexion.

**--- Objectif de la Tâche ---** Votre objectif est de construire le formulaire et la page de connexion, permettant aux utilisateurs de s'authentifier et de faire persister leur session dans l'état global de l'application.

1. Ajouter un schéma de validation zod pour les données du formulaire de connexion (essentiellement email et password) dans le fichier existant apps/frontend/src/lib/validators.ts.
2. Créer un nouveau composant de formulaire dédié, LoginForm.tsx, dans le répertoire apps/frontend/src/components/auth/.
3. Ce composant doit utiliser react-hook-form, le schéma de validation zod correspondant, et les composants UI réutilisables (Input, Button, Label) que nous avons créés dans la tâche précédente.
4. Implémenter la logique de soumission du formulaire (onSubmit). Cette fonction doit appeler authService.login() avec les identifiants fournis par l'utilisateur.
5. En cas de succès de l'appel API, la logique doit invoquer l'action appropriée de notre store Zustand (authStore) pour sauvegarder le jeton JWT et les informations de l'utilisateur dans l'état global de l'application.
6. Après une connexion réussie, l'utilisateur doit être redirigé par programmation vers la page du tableau de bord (/dashboard).
7. En cas d'échec (ex: mot de passe incorrect), un message d'erreur clair doit être affiché à l'utilisateur.
8. Mettre à jour la page LoginPage.tsx (créée comme placeholder au Sprint 1) dans apps/frontend/src/pages/ pour qu'elle affiche le composant LoginForm.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-002.1 (pour authService et authStore), FE-002.2 (pour les composants UI réutilisables).
* **Fichier(s) clé(s) à lire avant de commencer :** apps/frontend/src/services/authService.ts, apps/frontend/src/store/authStore.ts.
* **Information(s) clé(s) nécessaire(s) :** La gestion de la redirection après connexion doit être effectuée côté client en utilisant le hook useNavigate de react-router-dom.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 8 : Formulaires et Validation
* CDC Back-End, Partie 3 (pour le contrat de l'API /token).

**--- Livrables Attendus ---**

1. Les modifications apportées au fichier src/lib/validators.ts pour inclure le schéma de connexion.
2. Le contenu complet du nouveau composant src/components/auth/LoginForm.tsx.
3. Le contenu mis à jour de la page src/pages/LoginPage.tsx.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-002.4**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-002.4 **Titre :** frontend : Persistance de Session et Routes Protégées

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-002.3 est terminée. Un utilisateur peut s'inscrire et se connecter, et son jeton est stocké dans l'état global de l'application via le store Zustand.

Cependant, deux problèmes majeurs subsistent :

1. Si l'utilisateur rafraîchit la page, l'état de Zustand est réinitialisé et l'utilisateur est déconnecté.
2. Toutes les pages de l'application sont actuellement accessibles à tous, même celles qui devraient être réservées aux utilisateurs connectés (comme le tableau de bord).

**--- Objectif de la Tâche ---** Votre objectif est de résoudre ces deux problèmes en rendant la session persistante et en créant un système de routes protégées.

1. Modifier le store d'authentification (apps/frontend/src/store/authStore.ts) pour utiliser le middleware persist de zustand/middleware. Cela sauvegardera automatiquement l'état (token, informations utilisateur) dans le localStorage du navigateur et le restaurera au rechargement de la page.
2. Créer un nouveau composant ProtectedRoute.tsx dans le répertoire apps/frontend/src/components/auth/.
3. Ce composant doit lire l'état isAuthenticated depuis le authStore.
4. Si l'utilisateur n'est pas authentifié (isAuthenticated est false), le composant doit le rediriger automatiquement vers la page de connexion (/login) en utilisant le composant <Navigate /> de react-router-dom.
5. Si l'utilisateur est authentifié, le composant doit simplement afficher ses enfants en utilisant le composant <Outlet />.
6. Mettre à jour la configuration du routeur (App.tsx) pour utiliser ce nouveau composant. Vous devrez créer une structure de routes imbriquées où ProtectedRoute est un parent et toutes les pages nécessitant une authentification (comme une nouvelle page placeholder DashboardPage.tsx) sont ses enfants.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-002.3.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/frontend/src/store/authStore.ts, apps/frontend/src/App.tsx (ou le fichier de routage).
* **Information(s) clé(s) nécessaire(s) :** La persistance doit se faire via le localStorage. Le mécanisme de protection de route doit être intégré à react-router-dom.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 6 : Routage
* CDC Front-End, Partie 7 : Gestion de l'État

**--- Livrables Attendus ---**

1. Le fichier src/store/authStore.ts mis à jour avec le middleware persist correctement configuré.
2. Le contenu complet du nouveau composant src/components/auth/ProtectedRoute.tsx.
3. Le contenu mis à jour du fichier de configuration du routeur (src/App.tsx) montrant la nouvelle structure de routes protégées.
4. La création d'une page de test src/pages/DashboardPage.tsx pour valider que la route /dashboard est bien protégée.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-005.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-005.1 **Titre :** project-service : Logique CRUD pour les Livrables et Client Pub/Sub

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** Le Sprint 2 est terminé. Les services user-service et project-service disposent d'API de base et le front-end gère l'authentification. Nous démarrons le Sprint 3 pour implémenter la soumission de livrables.

Avant de créer l'endpoint API pour la soumission, nous devons d'abord implémenter deux mécanismes côté serveur :

1. La fonction pour insérer un enregistrement de "livrable" dans notre base de données.
2. La fonction pour publier un message dans un bus de messages (Pub/Sub) afin de notifier l'agent d'évaluation qu'un nouveau travail doit être traité.

**--- Objectif de la Tâche ---** Votre objectif est de créer les fonctions de support pour la gestion des livrables.

1. Mettre à jour le fichier requirements.txt du project-service pour y inclure la librairie google-cloud-pubsub.
2. Dans le fichier existant apps/backend/project-service/app/crud/project\_crud.py, ajouter une nouvelle fonction create\_deliverable(db: Session, \*, deliverable\_in: DeliverableCreate, user\_id: UUID, project\_id: UUID) -> Deliverable.
3. Cette fonction devra créer une nouvelle instance du modèle Deliverable et la sauvegarder en base de données.
4. Créer un nouveau module apps/backend/project-service/app/core/messaging.py.
5. Dans ce nouveau module, implémenter une fonction asynchrone publish\_message(topic\_name: str, data: dict).
6. Cette fonction doit utiliser le client Python de Google Cloud Pub/Sub (pubsub\_v1.PublisherClient) pour publier un message contenant les données fournies (au format JSON encodé en bytes) sur le topic spécifié.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** Epic BE-002.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/project-service/app/models/project\_models.py (pour les modèles Deliverable et le schéma DeliverableCreate).
* **Information(s) clé(s) nécessaire(s) :** La fonction publish\_message doit être async pour ne pas bloquer l'exécution de la requête API.

**--- Références aux Documents Clés ---**

* CDC Données (EDR), Partie 3 : Dictionnaire de Données Détaillé (pour le modèle Deliverable).
* CDC Agents IA, Partie 3 : Communication et Déclenchement (qui mandate l'utilisation de Pub/Sub).

**--- Livrables Attendus ---**

1. Le fichier requirements.txt du project-service mis à jour.
2. Les modifications apportées à app/crud/project\_crud.py contenant la nouvelle fonction create\_deliverable.
3. Le contenu complet du nouveau fichier app/core/messaging.py.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-005.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-005.2 **Titre :** project-service : Implémentation de l'Endpoint de Soumission de Livrable

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** La sous-tâche BE-005.1 est terminée. Nous disposons de la fonction create\_deliverable pour écrire en base de données et de la fonction publish\_message pour communiquer via Pub/Sub. Toutes les briques logiques sont prêtes.

Il est maintenant temps de créer le point d'entrée API qui orchestrera ce processus.

**--- Objectif de la Tâche ---** Votre objectif est de créer l'endpoint API protégé qui permet à un apprenant de soumettre un livrable pour un projet auquel il est assigné.

1. Créer une fonction de support dans la couche CRUD (apps/backend/project-service/app/crud/project\_crud.py) nommée is\_user\_assigned\_to\_project(db: Session, user\_id: UUID, project\_id: UUID) -> bool pour vérifier si un utilisateur est bien assigné à un projet.
2. Créer un nouveau module apps/backend/project-service/app/core/storage.py contenant une fonction utilitaire pour téléverser un fichier vers Google Cloud Storage. Cette fonction doit accepter un UploadFile et retourner le chemin GCS du fichier stocké.
3. Dans le fichier de routeur apps/backend/project-service/app/api/routes/projects.py, implémenter un nouvel endpoint POST /{project\_id}/deliverables.
4. Cet endpoint doit être protégé et requérir un utilisateur authentifié (de rôle apprenant).
5. Il doit être capable de gérer une requête multipart/form-data contenant un fichier (file: UploadFile) et des données de formulaire (notes: str = Form()).
6. La logique de l'endpoint doit d'abord valider que l'utilisateur authentifié est bien assigné au projet en utilisant la fonction CRUD créée à l'étape 1. Si ce n'est pas le cas, lever une HTTPException avec un code de statut 403 (Forbidden).
7. Si l'utilisateur est autorisé, la logique doit : a. Téléverser le fichier sur GCS en utilisant la fonction utilitaire de storage.py. b. Créer l'enregistrement du livrable en base de données via la fonction create\_deliverable. c. Publier un événement sur le topic project.deliverable.submitted via la fonction publish\_message. Le payload doit être un JSON contenant { "deliverable\_id", "project\_id", "user\_id", "file\_path\_in\_gcs" }.
8. L'endpoint doit retourner les données du livrable nouvellement créé en utilisant le schéma DeliverableRead.
9. Écrire les tests d'intégration correspondants, en simulant (mockant) les appels externes à GCS et Pub/Sub.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-005.1.
* **Fichier(s) clé(s) à lire avant de commencer :** app/crud/project\_crud.py, app/core/messaging.py, app/models/project\_models.py.
* **Information(s) clé(s) nécessaire(s) :** La gestion des fichiers dans FastAPI se fait via les types UploadFile et File, et les données de formulaire via Form.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 3.2 : Événements (Communication Asynchrone) (pour le format du payload Pub/Sub).

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau module apps/backend/project-service/app/core/storage.py.
2. Les modifications apportées à apps/backend/project-service/app/crud/project\_crud.py pour ajouter la fonction de vérification d'assignation.
3. Les modifications apportées à apps/backend/project-service/app/api/routes/projects.py pour ajouter le nouvel endpoint de soumission.
4. Les tests d'intégration supplémentaires pour ce nouvel endpoint.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : AG-001.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** AG-001.1 **Titre :** evaluation-agent : Initialisation du projet et du Listener Pub/Sub

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** L'Epic BE-005 est terminé. Le project-service publie maintenant un événement sur le topic project.deliverable.submitted. Nous devons créer le service qui consommera cet événement pour lancer le processus d'évaluation.

Le répertoire agents/evaluation-agent/ est actuellement vide. Nous partons de zéro pour construire ce premier agent.

**--- Objectif de la Tâche ---** Votre objectif est de créer le squelette de l'application pour l'evaluation-agent et d'implémenter la logique de base pour écouter les messages Pub/Sub.

1. Créer la structure de dossiers standard pour un agent transactionnel dans agents/evaluation-agent/, incluant les répertoires app/, app/core/, app/logic/, et app/clients/.
2. Créer un fichier requirements.txt à la racine de l'agent et y ajouter la librairie google-cloud-pubsub.
3. Dans apps/agents/evaluation-agent/app/core/config.py, créer une classe de configuration Settings (utilisant Pydantic-Settings) pour charger les variables d'environnement nécessaires : GCP\_PROJECT\_ID et PUBSUB\_SUBSCRIPTION\_ID.
4. Dans apps/agents/evaluation-agent/app/main.py, écrire le script principal asynchrone qui sera le point d'entrée du conteneur.
5. Ce script doit initialiser un SubscriberClient de la librairie google-cloud-pubsub.
6. Il doit implémenter une fonction de callback qui sera exécutée à la réception de chaque message. Pour cette tâche, cette fonction doit simplement : a. Journaliser (avec un print ou un logger) le contenu (.data) et les attributs (.attributes) du message reçu. b. Acquitter le message avec message.ack() pour qu'il ne soit pas renvoyé.
7. Le script doit lancer l'écoute en appelant la méthode subscriber.subscribe() et utiliser une boucle pour maintenir le processus en vie indéfiniment.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-005.
* **Fichier(s) clé(s) à lire avant de commencer :** Le CDC Agents IA pour comprendre l'architecture générale des agents.
* **Information(s) clé(s) nécessaire(s) :** Le nom de la souscription Pub/Sub sera fourni via une variable d'environnement (PUBSUB\_SUBSCRIPTION\_ID).

**--- Références aux Documents Clés ---**

* CDC Agents IA, Partie 2 : Structure des Projets Agents
* CDC Back-End, Partie 3.2 (pour connaître le nom du topic, et donc de la souscription associée)

**--- Livrables Attendus ---**

1. La structure de dossiers complète pour l' evaluation-agent.
2. Le fichier agents/evaluation-agent/requirements.txt contenant la nouvelle dépendance.
3. Le contenu complet du fichier de configuration app/core/config.py.
4. Le contenu complet du script principal app/main.py avec la logique du listener Pub/Sub.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : AG-001.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** AG-001.2 **Titre :** evaluation-agent : Implémentation du Workflow de Traitement de Message

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** La sous-tâche AG-001.1 est terminée. L'agent s'abonne à Pub/Sub, reçoit des messages et les acquitte. Actuellement, il se contente de journaliser le message reçu sans effectuer aucun traitement.

Nous devons maintenant implémenter la logique qui s'exécute après la réception : la collecte de toutes les données contextuelles et la préparation du livrable pour l'analyse.

**--- Objectif de la Tâche ---** Votre objectif est de construire le pipeline de traitement de données à l'intérieur de l'agent.

1. Créer un client API dans apps/agents/evaluation-agent/app/clients/backend\_api.py. Ce client doit contenir des fonctions pour appeler les APIs internes du project-service afin de récupérer les détails d'un projet et d'un livrable par leur ID.
2. Dans un nouveau module apps/agents/evaluation-agent/app/logic/processing.py, implémenter la logique du **"Pipeline de Traitement Commun"** tel que défini dans le CDC Agents IA. Cette fonction extract\_text\_from\_file(gcs\_path: str) doit être capable d'extraire le texte brut d'un fichier stocké sur GCS, en gérant au minimum les formats .pdf et .docx pour cette tâche.
3. Créer un nouveau module apps/agents/evaluation-agent/app/logic/evaluation\_logic.py qui contiendra le workflow principal.
4. Dans evaluation\_logic.py, implémenter une fonction process\_evaluation\_request(message\_data: dict) qui orchestre les étapes suivantes : a. Parser le message Pub/Sub pour extraire le deliverable\_id et le project\_id. b. Utiliser le client API pour obtenir le brief du projet et le chemin GCS du fichier du livrable. c. Utiliser le module de processing pour extraire le texte brut du fichier. d. Journaliser le succès de l'opération, en indiquant par exemple la quantité de texte extraite.
5. Modifier la fonction callback dans app/main.py pour qu'elle appelle cette nouvelle fonction process\_evaluation\_request avec les données du message qu'elle reçoit.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** AG-001.1
* **Fichier(s) clé(s) à lire avant de commencer :** Le CDC Agents IA pour la description du pipeline de traitement.
* **Information(s) clé(s) nécessaire(s) :** Pour cette tâche, nous ne faisons pas encore appel aux modèles d'IA. Nous nous concentrons uniquement sur la collecte et la préparation des données.

**--- Références aux Documents Clés ---**

* CDC Agents IA, Partie 2 : Pipeline de Traitement Commun

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier app/clients/backend\_api.py.
2. Le contenu complet du nouveau fichier app/logic/processing.py.
3. Le contenu complet du nouveau fichier app/logic/evaluation\_logic.py.
4. Les modifications apportées au fichier app/main.py pour intégrer l'appel au nouveau workflow.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : AG-001.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** AG-001.3 **Titre :** evaluation-agent : Intégration de l'Analyse IA et Soumission des Résultats

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** La sous-tâche AG-001.2 est terminée. L'agent est capable de traiter un message Pub/Sub, de collecter le brief du projet et d'extraire le texte propre du livrable. Il dispose de toutes les données nécessaires pour l'analyse.

Cependant, la logique IA elle-même n'est pas encore implémentée. L'agent ne fait actuellement aucun appel à un modèle de langage ou d'embedding et ne soumet aucun résultat au evaluation-service.

**--- Objectif de la Tâche ---** Votre objectif est de finaliser le workflow de l'agent en y intégrant les modèles d'IA pour l'analyse et en envoyant le résultat final au service back-end approprié.

1. Mettre à jour le fichier requirements.txt pour y inclure les librairies de Machine Learning nécessaires : torch, transformers, et sentence-transformers.
2. Créer un nouveau module apps/agents/evaluation-agent/app/clients/llm\_client.py. Il doit contenir une logique qui charge le modèle microsoft/phi-2 et expose une fonction pour générer une réponse textuelle à partir d'un prompt.
3. Créer un nouveau module apps/agents/evaluation-agent/app/clients/embedding\_client.py. Il doit contenir une logique qui charge le modèle sentence-transformers/all-MiniLM-L6-v2 et expose une fonction pour transformer un texte en vecteur (embedding).
4. Mettre à jour la fonction principale process\_evaluation\_request dans apps/agents/evaluation-agent/app/logic/evaluation\_logic.py. Après avoir préparé les données, elle doit maintenant : a. Utiliser l'embedding\_client pour calculer le score de pertinence sémantique. b. Utiliser le llm\_client et le prompt adéquat (défini dans le CDC Agents IA) pour obtenir l'évaluation qualitative au format JSON. c. Combiner tous les résultats pour former le payload final de l'évaluation.
5. Ajouter une nouvelle fonction dans apps/agents/evaluation-agent/app/clients/backend\_api.py pour soumettre ce payload final via un appel POST à l'endpoint interne de l'evaluation-service.
6. Écrire les tests d'intégration pour le workflow complet dans un nouveau fichier tests/test\_evaluation\_logic.py, en simulant (mockant) les réponses des clients IA et l'appel final à l'API back-end.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** AG-001.2.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/agents/evaluation-agent/app/logic/evaluation\_logic.py.
* **Information(s) clé(s) nécessaire(s) :** Le format exact du prompt pour le LLM et du payload JSON final sont définis dans le CDC Agents IA.

**--- Références aux Documents Clés ---**

* CDC Agents IA, Partie 3.1 (pour le détail du workflow, des modèles et des prompts de l'evaluation-agent).

**--- Livrables Attendus ---**

1. Le fichier requirements.txt mis à jour avec les librairies ML.
2. Le contenu complet des nouveaux fichiers app/clients/llm\_client.py et app/clients/embedding\_client.py.
3. Les modifications apportées à app/logic/evaluation\_logic.py pour intégrer les appels aux modèles IA.
4. Les modifications apportées à app/clients/backend\_api.py pour ajouter la fonction de soumission de l'évaluation.
5. Le nouveau fichier de test tests/test\_evaluation\_logic.py.
6. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-003.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-003.1 **Titre :** frontend : Service API Projets et Composant d'Affichage du Brief

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** Le Sprint 3 a commencé. Le project-service du back-end expose maintenant des endpoints fonctionnels pour récupérer les détails d'un projet (GET /api/v1/projects/{projectId}).

L'application front-end, de son côté, est capable de gérer l'authentification et le routage, mais ne dispose d'aucune logique pour récupérer ou afficher les données relatives à un projet.

**--- Objectif de la Tâche ---** Votre objectif est de construire la logique de récupération des données d'un projet et le composant de base pour afficher ces informations.

1. Créer un nouveau répertoire apps/frontend/src/features/projects/ pour héberger toute la logique liée aux projets.
2. Créer un nouveau service API dans apps/frontend/src/services/projectService.ts.
3. Dans ce fichier, implémenter une fonction asynchrone getProjectById(projectId: string) qui utilise notre apiClient (Axios) pour appeler l'endpoint GET /api/v1/projects/{projectId}.
4. Créer un hook React personnalisé dans apps/frontend/src/features/projects/hooks/useProject.ts. Ce hook doit utiliser useQuery de la librairie **TanStack Query** pour appeler la fonction projectService.getProjectById et gérer les états de chargement, d'erreur et de mise en cache des données.
5. Créer un composant de présentation (dit "dumb component") nommé ProjectBrief.tsx dans apps/frontend/src/features/projects/components/.
6. Ce composant doit accepter en props un objet de type projet (utilisez les types générés depuis l'API) et afficher de manière claire le titre du projet, sa description complète, et la liste de ses compétences requises sous forme de badges.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-001, BE-004.
* **Fichier(s) clé(s) à lire avant de commencer :** Le CDC Back-End pour le contrat exact de la réponse de l'API GET /projects/{id}.
* **Information(s) clé(s) nécessaire(s) :** TanStack Query (React Query) est l'outil obligatoire pour gérer l'état serveur, comme défini dans le CDC Front-End.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 2.3 : Stratégie de Gestion de l'État
* CDC Back-End, Partie 3.1 : Catalogue des Endpoints Principaux

**--- Livrables Attendus ---**

1. La nouvelle arborescence de dossiers apps/frontend/src/features/projects/.
2. Le contenu complet du nouveau fichier src/services/projectService.ts.
3. Le contenu complet du nouveau hook src/features/projects/hooks/useProject.ts.
4. Le contenu complet du nouveau composant src/features/projects/components/ProjectBrief.tsx.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-003.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-003.1 **Titre :** frontend : Service API Projets et Composant d'Affichage du Brief

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** Le Sprint 3 a commencé. Le back-end dispose maintenant des endpoints pour consulter les projets (GET /projects/{id}). Le front-end est prêt à être développé, mais il n'a aucune logique pour récupérer ou afficher les données d'un projet.

**--- Objectif de la Tâche ---** Votre objectif est de construire la logique de récupération des données d'un projet et le composant de base pour afficher ces informations.

1. Créer un nouveau répertoire apps/frontend/src/features/projects/ pour héberger toute la logique liée aux projets.
2. Créer un nouveau service API dans apps/frontend/src/services/projectService.ts.
3. Dans ce fichier, implémenter une fonction asynchrone getProjectById(projectId: string) qui utilise notre apiClient (Axios) pour appeler l'endpoint GET /api/v1/projects/{projectId}.
4. Créer un hook React personnalisé dans apps/frontend/src/features/projects/hooks/useProject.ts. Ce hook doit utiliser useQuery de la librairie **TanStack Query** pour appeler la fonction projectService.getProjectById et gérer les états de chargement, d'erreur et de mise en cache des données.
5. Créer un composant de présentation (dit "dumb component") nommé ProjectBrief.tsx dans apps/frontend/src/features/projects/components/.
6. Ce composant doit accepter en props un objet de type projet (utilisez les types générés depuis l'API) et afficher de manière claire le titre du projet, sa description complète, et la liste de ses compétences requises sous forme de badges.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-001, BE-004.
* **Fichier(s) clé(s) à lire avant de commencer :** Le CDC Back-End pour le contrat exact de la réponse de l'API GET /projects/{id}.
* **Information(s) clé(s) nécessaire(s) :** TanStack Query (React Query) est l'outil obligatoire pour gérer l'état serveur, comme défini dans le CDC Front-End.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 2.3 : Stratégie de Gestion de l'État
* CDC Back-End, Partie 3.1 : Catalogue des Endpoints Principaux

**--- Livrables Attendus ---**

1. La nouvelle arborescence de dossiers apps/frontend/src/features/projects/.
2. Le contenu complet du nouveau fichier src/services/projectService.ts.
3. Le contenu complet du nouveau hook src/features/projects/hooks/useProject.ts.
4. Le contenu complet du nouveau composant src/features/projects/components/ProjectBrief.tsx.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-003.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-003.2 **Titre :** frontend : Création du Formulaire de Soumission de Livrable

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-003.1 est terminée. Nous disposons de la logique (useProject hook) pour récupérer les données d'un projet et d'un composant (ProjectBrief) pour les afficher. Le back-end expose un endpoint POST /projects/{id}/deliverables prêt à recevoir des fichiers.

Cependant, aucune interface utilisateur n'existe pour permettre à l'apprenant de sélectionner un fichier, d'ajouter des notes et de soumettre son travail.

**--- Objectif de la Tâche ---** Votre objectif est de construire un formulaire React robuste pour la soumission de livrables, en gérant la logique de téléversement de fichier et l'état de la requête.

1. Mettre à jour le service apps/frontend/src/services/projectService.ts avec une nouvelle fonction submitDeliverable(projectId: string, formData: FormData). Cette fonction doit envoyer une requête POST avec un Content-Type de multipart/form-data.
2. Ajouter un schéma de validation zod dans le fichier apps/frontend/src/lib/validators.ts pour le formulaire de soumission. Ce schéma doit valider la présence d'un fichier et imposer des contraintes (ex: taille maximale de 10MB).
3. Créer un nouveau composant de formulaire, DeliverableSubmissionForm.tsx, dans le répertoire apps/frontend/src/features/projects/components/.
4. Ce composant doit utiliser le hook useForm de react-hook-form (avec le zodResolver) pour gérer l'état et la validation du formulaire.
5. Le formulaire doit contenir un champ de saisie de fichier (<input type="file" />) et un champ de texte (Textarea) pour les notes.
6. La logique de soumission doit être gérée par le hook useMutation de **TanStack Query**. Il appellera la fonction submitDeliverable.
7. Le composant doit refléter l'état de la mutation : afficher un spinner sur le bouton pendant le chargement (isLoading), afficher une notification de succès ou d'erreur, et en cas de succès, invalider la query du projet (useProject) pour forcer le rafraîchissement des données de la page.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-003.1, BE-005.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/frontend/src/services/projectService.ts et apps/frontend/src/lib/validators.ts.
* **Information(s) clé(s) nécessaire(s) :** La soumission de fichiers via Axios avec react-hook-form nécessite la construction d'un objet FormData.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 8 : Formulaires et Validation
* CDC Back-End, Partie 3 (pour le contrat de l'API de soumission de livrable).

**--- Livrables Attendus ---**

1. Le fichier src/services/projectService.ts mis à jour avec la nouvelle fonction submitDeliverable.
2. Le fichier src/lib/validators.ts mis à jour avec le nouveau schéma de validation.
3. Le contenu complet du nouveau composant src/features/projects/components/DeliverableSubmissionForm.tsx.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-003.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-003.3 **Titre :** frontend : Assemblage de la Page de Détail du Projet

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** Les sous-tâches FE-003.1 et FE-003.2 sont terminées. Nous disposons d'un hook useProject pour récupérer les données d'un projet, d'un composant ProjectBrief pour afficher ces données, et d'un composant DeliverableSubmissionForm pour la soumission.

Il nous manque maintenant la "colle" : le composant de page qui orchestre ces éléments pour créer une vue utilisateur unifiée.

**--- Objectif de la Tâche ---** Votre objectif est de créer le composant de page final qui récupère les données d'un projet via l'URL et affiche les composants appropriés en fonction de l'état de la requête.

1. Créer un nouveau composant de page, ProjectDetailPage.tsx, dans le répertoire apps/frontend/src/pages/.
2. Dans ce composant, utiliser le hook useParams de react-router-dom pour extraire le projectId dynamique depuis l'URL.
3. Utiliser notre hook personnalisé useProject(projectId) pour déclencher la récupération des données du projet.
4. Gérer les différents états de la requête retournés par le hook :
   * Si la requête est en cours de chargement (isLoading), afficher un composant Spinner.
   * Si la requête a échoué (isError), afficher un composant Alert avec un message d'erreur.
5. Si les données sont récupérées avec succès (isSuccess), assembler la page en affichant :
   * Le composant ProjectBrief, en lui passant l'objet project récupéré en props.
   * Le composant DeliverableSubmissionForm, en lui passant le projectId en props.
6. Mettre en page ces deux composants de manière logique en utilisant les classes utilitaires de Tailwind CSS.
7. S'assurer que la route /projects/:projectId dans le routeur principal (App.tsx) pointe bien vers ce nouveau composant de page.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-003.1, FE-003.2.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/frontend/src/features/projects/hooks/useProject.ts, apps/frontend/src/features/projects/components/ProjectBrief.tsx, apps/frontend/src/features/projects/components/DeliverableSubmissionForm.tsx.
* **Information(s) clé(s) nécessaire(s) :** L'ID du projet doit être récupéré de l'URL de manière dynamique.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 6 : Routage

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier apps/frontend/src/pages/ProjectDetailPage.tsx.
2. Toutes modifications nécessaires au fichier de routage (src/App.tsx) pour que la route soit fonctionnelle.
3. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-006.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-006.1 **Titre :** portfolio-service : Initialisation, Modèles de Données et Migration

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** Le Sprint 3 est terminé. Nous démarrons le Sprint 4, qui se concentre sur la valorisation du travail des apprenants. La première étape est de construire le microservice qui gérera les portfolios.

Le répertoire apps/backend/portfolio-service/ vient d'être créé et est vide. Nous devons mettre en place toute la couche de données pour ce nouveau service, en incluant ses modèles et son propre système de migration.

**--- Objectif de la Tâche ---** Votre objectif est de créer toute la fondation de la couche de données pour le nouveau portfolio-service.

1. Créer la structure de dossiers standard pour un nouveau microservice à l'intérieur de apps/backend/portfolio-service/.
2. Dans un nouveau fichier app/models/portfolio\_models.py, définir les classes SQLModel pour les tables Portfolio, PortfolioItem, et la table de liaison PortfolioSkill.
3. Ces modèles doivent respecter scrupuleusement le CDC Données, en incluant toutes les colonnes, les types, les contraintes, et les relations ForeignKey et Relationship vers les tables users, projects, et skills.
4. Créer les schémas Pydantic de base nécessaires pour les futures opérations API (ex: PortfolioRead, PortfolioItemCreate, PortfolioItemRead).
5. Initialiser et configurer Alembic spécifiquement pour ce service (dans apps/backend/portfolio-service/alembic/), en liant sa configuration aux nouveaux modèles que vous venez de créer.
6. Générer le script de migration initial qui contient les instructions CREATE TABLE pour toutes les tables liées au portfolio.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** La conception des modèles dans les Epics BE-001 et BE-002 est une référence pour les clés étrangères.
* **Fichier(s) clé(s) à lire avant de commencer :** Aucun fichier de code n'est nécessaire, la source de vérité est la documentation.
* **Information(s) clé(s) nécessaire(s) :** Les relations One-to-One (User-Portfolio) et Many-to-Many (Portfolio-Project) doivent être implémentées correctement avec SQLModel.

**--- Références aux Documents Clés ---**

* **Source de Vérité Absolue :** CDC Données (EDR), Partie 3 : Dictionnaire de Données Détaillé (pour les tables portfolios, portfolio\_items, portfolio\_skills).
* CDC Données (EDR), Partie 7 : Gestion des Versions et des Migrations.

**--- Livrables Attendus ---**

1. La structure de dossiers complète pour le portfolio-service.
2. Le contenu complet du nouveau fichier apps/backend/portfolio-service/app/models/portfolio\_models.py.
3. Le répertoire alembic/ pour ce service, entièrement configuré.
4. Le contenu du script de migration initial généré par Alembic.
5. Un rapport de fin de tâche complet.

### **PROMPT DE Tâche : BE-006.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-006.2 **Titre :** portfolio-service : Implémentation de la Couche CRUD

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** La sous-tâche BE-006.1 est terminée. Les modèles de données et le script de migration pour les tables Portfolio, PortfolioItem, et PortfolioSkill sont prêts.

Cependant, aucune fonction n'existe encore pour manipuler les données dans ces tables. Nous avons besoin de créer une couche d'abstraction CRUD (Create, Read, Update, Delete) pour encapsuler la logique d'interaction avec la base de données.

**--- Objectif de la Tâche ---** Votre objectif est de créer le module Python contenant les fonctions pour les opérations de base sur les entités liées au portfolio.

1. Créer un nouveau répertoire app/crud/ à l'intérieur du portfolio-service (apps/backend/portfolio-service/app/crud/).
2. Dans ce répertoire, créer un nouveau fichier nommé portfolio\_crud.py.
3. Dans portfolio\_crud.py, implémenter les fonctions CRUD essentielles pour les entités Portfolio et PortfolioItem.
4. Les fonctions doivent prendre une session de base de données (Session) comme argument et utiliser les schémas Pydantic/SQLModel pour les opérations.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-006.1.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/portfolio-service/app/models/portfolio\_models.py (pour les modèles et schémas).
* **Information(s) clé(s) nécessaire(s) :** La logique doit correctement gérer les relations entre Portfolio et PortfolioItem.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 5 : Logique Métier et Services.

**--- Livrables Attendus ---**

1. La création du répertoire apps/backend/portfolio-service/app/crud/.
2. Le contenu complet du nouveau fichier portfolio\_crud.py.
3. Ce fichier doit contenir au minimum les fonctions suivantes, entièrement typées :
   * create\_portfolio(db: Session, \*, portfolio\_in: PortfolioCreate) -> Portfolio
   * get\_portfolio\_by\_user\_id(db: Session, user\_id: UUID) -> Optional[Portfolio]
   * add\_item\_to\_portfolio(db: Session, \*, item\_in: PortfolioItemCreate, portfolio\_id: UUID) -> PortfolioItem
   * get\_portfolio\_with\_details(db: Session, user\_id: UUID) -> Optional[Portfolio] (cette fonction doit charger le portfolio, ses items et les skills associés).
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-006.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-006.3 **Titre :** portfolio-service : Implémentation des Endpoints API de Base

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** La sous-tâche BE-006.2 est terminée. La couche CRUD du portfolio-service est fonctionnelle, avec une logique pour créer et lire des portfolios et leurs items.

Cependant, ce service n'expose actuellement aucune interface réseau. Il est inaccessible pour le front-end ou pour les autres services. Nous devons construire la couche API REST par-dessus la couche CRUD.

**--- Objectif de la Tâche ---** Votre objectif est de créer l'application FastAPI et les endpoints de base pour le portfolio-service.

1. Initialiser le squelette de l'application FastAPI pour le portfolio-service (création des fichiers app/main.py, app/db/session.py, app/api/dependencies.py).
2. Intégrer la logique d'authentification (get\_current\_user) dans ce service pour pouvoir protéger les routes, en la copiant depuis un autre service pour garantir l'autonomie.
3. Créer un nouveau fichier de routeur dans apps/backend/portfolio-service/app/api/routes/portfolios.py.
4. Implémenter un endpoint **protégé** GET /me qui retourne le portfolio détaillé (avec ses items et compétences) de l'utilisateur actuellement connecté.
5. Implémenter un endpoint **public** GET /{public\_url\_slug} qui retourne les données d'un portfolio pour sa consultation publique.
6. Implémenter un endpoint **protégé** PUT /me/status qui permet à un utilisateur de changer l'état de visibilité (is\_public) de son portfolio.
7. Écrire les tests d'intégration pour ces nouveaux endpoints dans un nouveau fichier tests/api/test\_portfolios.py.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-006.2.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/portfolio-service/app/crud/portfolio\_crud.py.
* **Information(s) clé(s) nécessaire(s) :** Vous devrez peut-être ajouter de nouvelles fonctions à la couche CRUD pour supporter ces endpoints (ex: get\_portfolio\_by\_slug, update\_portfolio\_status).

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 3 : Spécification des Microservices.
* CDC Données (EDR), Partie 3 : Dictionnaire de Données Détaillé.

**--- Livrables Attendus ---**

1. La structure complète de l'application FastAPI pour le portfolio-service.
2. Le contenu complet du nouveau fichier apps/backend/portfolio-service/app/api/routes/portfolios.py.
3. Le contenu complet du nouveau fichier de test apps/backend/portfolio-service/tests/api/test\_portfolios.py.
4. Toutes les modifications ou ajouts nécessaires au fichier portfolio\_crud.py.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-007.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-007.1 **Titre :** evaluation-service : Création du Service et Endpoint de Consultation

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** Le Sprint 4 a commencé. L'evaluation-agent est capable de stocker des résultats d'évaluation dans la table evaluations de la base de données. Cependant, il n'existe actuellement aucun microservice dédié pour exposer ces données de manière sécurisée au Front-End.

**--- Objectif de la Tâche ---** Votre objectif est de créer le microservice evaluation-service avec un unique endpoint permettant à un utilisateur de consulter une évaluation qui lui appartient.

1. Créer le squelette complet du microservice evaluation-service (apps/backend/evaluation-service/), incluant la structure de dossiers, le Dockerfile, et le requirements.txt.
2. Copier et adapter les modèles de données Evaluation et les schémas Pydantic correspondants (EvaluationRead) depuis le project-service vers ce nouveau service.
3. Implémenter une couche CRUD simple avec une fonction get\_evaluation\_by\_deliverable\_id.
4. Mettre en place la couche API (main.py, dependencies.py, etc.) avec la logique d'authentification get\_current\_user.
5. Créer un endpoint **protégé** GET /deliverables/{deliverable\_id}/evaluation.
6. La logique de l'endpoint doit vérifier que l'utilisateur authentifié est bien l'auteur du livrable (en faisant une jointure ou une seconde requête) avant de retourner les données de l'évaluation, sinon retourner une erreur 403 Forbidden.
7. Écrire les tests d'intégration pour cet endpoint, incluant la vérification des permissions.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** AG-001.3.
* **Fichier(s) clé(s) à lire avant de commencer :** CDC Données pour le schéma de la table evaluations et deliverables.
* **Information(s) clé(s) nécessaire(s) :** La vérification de la propriété du livrable est une étape de sécurité cruciale.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 3 : Spécification des Microservices.
* CDC Données (EDR), Partie 3 : Dictionnaire de Données Détaillé.

**--- Livrables Attendus ---**

1. La structure complète du nouveau service evaluation-service.
2. Le code de l'endpoint GET /deliverables/{deliverable\_id}/evaluation.
3. Les tests d'intégration pour cet endpoint.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-004.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-004.1 **Titre :** frontend : Service API et Composants d'Affichage de l'Évaluation

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** Le back-end expose maintenant un endpoint (GET /deliverables/{id}/evaluation) pour récupérer les résultats d'une évaluation. Le front-end doit maintenant implémenter la logique pour appeler cet endpoint et les composants de base pour afficher les résultats de manière claire et pédagogique.

**--- Objectif de la Tâche ---** Votre objectif est de construire la logique de récupération des données d'évaluation et les composants de base pour afficher ces informations.

1. Créer un nouveau répertoire apps/frontend/src/features/evaluations/ pour héberger la logique de cette fonctionnalité.
2. Créer un nouveau service evaluationService.ts (ou mettre à jour un service existant) avec une fonction getEvaluationForDeliverable(deliverableId: string) pour appeler l'API.
3. Créer un nouveau hook useEvaluation.ts dans src/features/evaluations/hooks/ qui utilise useQuery de TanStack Query pour gérer la récupération des données d'évaluation.
4. Dans src/features/evaluations/components/, créer deux composants de présentation :
   * ScoreCard.tsx : Pour afficher une note chiffrée de manière proéminente (ex: dans un cercle de couleur).
   * FeedbackList.tsx : Un composant qui prend un titre (ex: "Points Forts") et un tableau de chaînes de caractères pour les afficher sous forme de liste.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-007.1 (doit être terminée pour que l'API soit disponible).
* **Fichier(s) clé(s) à lire avant de commencer :** CDC Back-End pour le contrat de la réponse de l'API d'évaluation.
* **Information(s) clé(s) nécessaire(s) :** Les composants doivent être purement présentationnels ("dumb"), leur seule responsabilité est d'afficher les données qu'ils reçoivent en props.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 3.2 : Catalogue des Composants d'UI Fondamentaux.

**--- Livrables Attendus ---**

1. La nouvelle arborescence de dossiers src/features/evaluations/.
2. Le nouveau fichier de service API evaluationService.ts.
3. Le nouveau hook useEvaluation.ts.
4. Les nouveaux composants ScoreCard.tsx et FeedbackList.tsx.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-004.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-004.2 **Titre :** frontend : Assemblage de la Page de Résultat de l'Évaluation

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-004.1 est terminée. Nous disposons du hook useEvaluation pour récupérer les données d'une évaluation, ainsi que des composants ScoreCard et FeedbackList pour afficher ces données.

Il manque maintenant la page qui va orchestrer ces éléments pour présenter un rapport d'évaluation complet et cohérent à l'apprenant.

**--- Objectif de la Tâche ---** Votre objectif est de créer la page de consultation des résultats d'une évaluation.

1. Créer un nouveau composant de page, EvaluationResultPage.tsx, dans le répertoire apps/frontend/src/pages/.
2. Dans ce composant, utiliser le hook useParams de react-router-dom pour extraire l'ID du livrable (deliverableId) depuis l'URL.
3. Utiliser notre hook personnalisé useEvaluation(deliverableId) pour déclencher la récupération des données de l'évaluation correspondante.
4. Gérer les différents états de la requête retournés par le hook :
   * Si la requête est en cours de chargement (isLoading), afficher un composant Spinner.
   * Si la requête a échoué (isError), afficher un composant Alert.
5. Si les données sont récupérées avec succès, assembler la page en affichant :
   * Le composant ScoreCard avec le score global.
   * Le composant FeedbackList pour les "Points Forts".
   * Le composant FeedbackList pour les "Axes d'Amélioration".
6. Organiser la mise en page de ces composants de manière claire et esthétique en utilisant Tailwind CSS.
7. Mettre à jour le routeur de l'application (App.tsx) pour ajouter une nouvelle route **protégée** /deliverables/:deliverableId/evaluation qui pointe vers cette nouvelle page.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-004.1.
* **Fichier(s) clé(s) à lire avant de commencer :** Les composants ScoreCard et FeedbackList, et le hook useEvaluation.
* **Information(s) clé(s) nécessaire(s) :** Cette page doit être protégée ; un utilisateur non connecté ne doit pas pouvoir y accéder.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 6 : Routage

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier apps/frontend/src/pages/EvaluationResultPage.tsx.
2. Les modifications apportées au fichier de routage (src/App.tsx) pour enregistrer la nouvelle route protégée.
3. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-008.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-008.1 **Titre :** messaging-service : Initialisation, Modèles de Données et Migration

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** Le Sprint 4 est terminé. Nous démarrons le Sprint 5 pour ajouter les dernières fonctionnalités majeures du MVP. La première est la messagerie en temps réel, qui nécessitera son propre microservice.

Le répertoire apps/backend/messaging-service/ vient d'être créé et est vide. Nous devons mettre en place toute la couche de données pour ce nouveau service.

**--- Objectif de la Tâche ---** Votre objectif est de créer toute la fondation de la couche de données pour le nouveau messaging-service.

1. Créer la structure de dossiers standard pour un nouveau microservice à l'intérieur de apps/backend/messaging-service/.
2. Dans un nouveau fichier app/models/messaging\_models.py, définir les classes SQLModel pour les tables Conversation, ConversationParticipant (table de liaison), et Message.
3. Ces modèles doivent respecter scrupuleusement le CDC Données, en incluant toutes les colonnes, types, et relations ForeignKey (notamment vers la table users).
4. Créer les schémas Pydantic de base nécessaires pour les futures opérations API (MessageRead, ConversationRead, MessageCreate).
5. Initialiser et configurer Alembic spécifiquement pour ce service (dans apps/backend/messaging-service/alembic/), en liant sa configuration aux nouveaux modèles.
6. Générer le script de migration initial qui contient les instructions CREATE TABLE pour toutes les tables liées à la messagerie.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** La conception du modèle User (BE-001) est une référence pour les clés étrangères.
* **Fichier(s) clé(s) à lire avant de commencer :** Aucun fichier de code n'est nécessaire, la source de vérité est la documentation.
* **Information(s) clé(s) nécessaire(s) :** La table ConversationParticipant est une table de liaison qui mappe les utilisateurs aux conversations.

**--- Références aux Documents Clés ---**

* **Source de Vérité Absolue :** CDC Données (EDR), Partie 3 : Dictionnaire de Données Détaillé (pour les tables conversations, conversation\_participants, messages).

**--- Livrables Attendus ---**

1. La structure de dossiers complète pour le messaging-service.
2. Le contenu complet du nouveau fichier apps/backend/messaging-service/app/models/messaging\_models.py.
3. Le répertoire alembic/ pour ce service, entièrement configuré.
4. Le contenu du script de migration initial généré par Alembic.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-008.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-008.2 **Titre :** messaging-service : Implémentation de la Couche CRUD

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** La sous-tâche BE-008.1 est terminée. Les modèles de données et le script de migration pour les tables Conversation, ConversationParticipant et Message sont prêts.

Il n'existe actuellement aucune fonction pour manipuler les données dans ces tables. Nous devons créer une couche d'abstraction CRUD pour encapsuler la logique d'interaction avec la base de données avant de pouvoir créer les endpoints API et WebSocket.

**--- Objectif de la Tâche ---** Votre objectif est de créer le module Python contenant les fonctions pour les opérations de base sur les entités liées à la messagerie.

1. Créer un nouveau répertoire app/crud/ à l'intérieur du messaging-service.
2. Dans ce répertoire, créer un nouveau fichier nommé messaging\_crud.py.
3. Dans messaging\_crud.py, implémenter les fonctions CRUD essentielles pour les entités Conversation et Message.
4. Les fonctions doivent prendre une session de base de données (Session) comme argument et utiliser les schémas Pydantic/SQLModel pour les opérations.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-008.1.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/messaging-service/app/models/messaging\_models.py.
* **Information(s) clé(s) nécessaire(s) :** La logique de création d'une conversation doit aussi gérer l'ajout des participants dans la table de liaison ConversationParticipant.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 5 : Logique Métier et Services.

**--- Livrables Attendus ---**

1. La création du répertoire apps/backend/messaging-service/app/crud/.
2. Le contenu complet du nouveau fichier messaging\_crud.py.
3. Ce fichier doit contenir au minimum les fonctions suivantes, entièrement typées :
   * create\_conversation(db: Session, \*, conversation\_in: ConversationCreate, participant\_ids: List[UUID]) -> Conversation
   * get\_user\_conversations(db: Session, \*, user\_id: UUID) -> List[Conversation]
   * create\_message(db: Session, \*, message\_in: MessageCreate, author\_id: UUID, conversation\_id: UUID) -> Message
   * get\_messages\_for\_conversation(db: Session, \*, conversation\_id: UUID, skip: int = 0, limit: int = 100) -> List[Message]
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : BE-008.3**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** BE-008.3 **Titre :** messaging-service : Implémentation du WebSocket Manager et des Endpoints

**IA Spécialiste Assignée :** IA #1 - User & Core

**--- Contexte et État Actuel ---** La sous-tâche BE-008.2 est terminée. La couche CRUD du messaging-service est fonctionnelle, avec une logique pour créer et lire des conversations et des messages.

Il manque maintenant la couche API et WebSocket pour exposer cette logique et permettre aux clients de communiquer en temps réel.

**--- Objectif de la Tâche ---** Votre objectif est de construire la couche de communication en temps réel du messaging-service.

1. Créer un nouveau module apps/backend/messaging-service/app/core/connection\_manager.py. Dans ce fichier, implémenter une classe ConnectionManager qui maintiendra un dictionnaire des connexions WebSocket actives pour pouvoir diffuser des messages.
2. Cette classe doit fournir des méthodes pour connect(websocket, user\_id), disconnect(user\_id), et broadcast\_to\_conversation(message, conversation\_id).
3. Créer le routeur apps/backend/messaging-service/app/api/routes/messaging.py.
4. Dans ce routeur, implémenter les endpoints **REST** classiques et protégés pour :
   * GET /conversations : Lister les conversations de l'utilisateur connecté.
   * GET /conversations/{conversation\_id}/messages : Récupérer l'historique des messages d'une conversation.
5. Dans le même routeur, implémenter un endpoint **WebSocket** @router.websocket("/ws").
6. Cet endpoint doit : a. Authentifier l'utilisateur via un token passé en paramètre de l'URL de connexion. b. Accepter la connexion et l'enregistrer dans le ConnectionManager. c. Entrer dans une boucle pour écouter les messages entrants du client (au format JSON). d. Pour chaque message reçu, le sauvegarder en base de données via le CRUD, puis le diffuser à tous les autres participants de la conversation via le ConnectionManager. e. Gérer la déconnexion en retirant le client du ConnectionManager.
7. Écrire les tests d'intégration pour les endpoints REST et les tests unitaires pour la logique WebSocket (en simulant une connexion).

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-008.2.
* **Fichier(s) clé(s) à lire avant de commencer :** apps/backend/messaging-service/app/crud/messaging\_crud.py.
* **Information(s) clé(s) nécessaire(s) :** FastAPI fournit une excellente prise en charge des WebSockets, qui sera l'outil principal pour cette tâche.

**--- Références aux Documents Clés ---**

* CDC Back-End, Partie 3.4 : Communication Temps Réel (WebSockets).

**--- Livrables Attendus ---**

1. Le contenu complet du nouveau fichier app/core/connection\_manager.py.
2. Le contenu complet du nouveau fichier app/api/routes/messaging.py avec les endpoints REST et WebSocket.
3. Les fichiers de test correspondants.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : AG-003.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** AG-003.1 **Titre :** portfolio-agent : Création de l'Agent d'Enrichissement de Portfolio

**IA Spécialiste Assignée :** IA #2 - Project & AI

**--- Contexte et État Actuel ---** Le Sprint 5 a commencé. L'evaluation-service est capable de finaliser une évaluation et le portfolio-service est fonctionnel. Il manque un mécanisme pour connecter ces deux services : un agent qui, lorsqu'une évaluation est réussie, ajoute automatiquement le projet correspondant au portfolio de l'apprenant.

Le répertoire agents/portfolio-agent/ vient d'être créé et est vide.

**--- Objectif de la Tâche ---** Votre objectif est de construire de bout en bout l'portfolio-agent, de la réception de l'événement à l'appel de l'API du portfolio-service.

1. Créer la structure de dossiers standard pour un agent transactionnel dans agents/portfolio-agent/.
2. Mettre en place un listener Pub/Sub dans app/main.py. Cet agent doit s'abonner au topic evaluation.completed (on supposera que l' evaluation-service publie sur ce topic après avoir stocké une évaluation).
3. Créer un client API dans apps/agents/portfolio-agent/app/clients/backend\_api.py. Ce client doit contenir les fonctions pour appeler les APIs du portfolio-service.
4. Implémenter la logique principale dans apps/agents/portfolio-agent/app/logic/portfolio\_logic.py. Cette fonction process\_portfolio\_update doit : a. Parser le message evaluation.completed pour extraire user\_id, project\_id, et evaluation\_id. b. Utiliser le client API pour trouver le portfolio\_id de l'utilisateur. c. Construire un payload pour un nouvel PortfolioItem contenant les informations du projet réussi. d. Appeler l'API du portfolio-service pour ajouter ce nouvel item au portfolio.
5. Intégrer cette logique dans la fonction callback du listener Pub/Sub dans main.py.
6. Écrire les tests unitaires pour la fonction process\_portfolio\_update, en simulant (mockant) les appels au client API.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-006 (le portfolio-service et son API doivent exister), BE-007 (l'evaluation-service doit exister).
* **Fichier(s) clé(s) à lire avant de commencer :** CDC Back-End pour les contrats d'API du portfolio-service.
* **Information(s) clé(s) nécessaire(s) :** L'agent n'a pas besoin de logique IA complexe ; son rôle est transactionnel et basé sur des événements.

**--- Références aux Documents Clés ---**

* CDC Agents IA, Partie 3 : Communication et Déclenchement
* CDC Back-End, Partie 3.1 : Catalogue des Endpoints Principaux

**--- Livrables Attendus ---**

1. La structure de dossiers complète pour le portfolio-agent.
2. Le contenu du client API, de la logique de traitement, et du script main.py.
3. Le fichier de test pour la logique de l'agent.
4. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-005.1**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-005.1 **Titre :** frontend : Service API et Composants d'Affichage du Portfolio

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** Le portfolio-service du back-end est maintenant fonctionnel. Il expose des endpoints pour récupérer le portfolio d'un utilisateur connecté (/me) et un portfolio public via son slug (/{slug}).

Le front-end n'a actuellement aucune logique pour appeler ces endpoints, ni de composants dédiés pour afficher les informations d'un portfolio.

**--- Objectif de la Tâche ---** Votre objectif est de construire la logique de récupération des données du portfolio et les composants de base pour afficher ces informations.

1. Créer un nouveau répertoire apps/frontend/src/features/portfolios/ pour héberger la logique de cette fonctionnalité.
2. Créer un nouveau service portfolioService.ts avec deux fonctions :
   * getMyPortfolio(): Appelle l'endpoint protégé GET /api/v1/portfolios/me.
   * getPublicPortfolioBySlug(slug: string): Appelle l'endpoint public GET /api/v1/portfolios/{slug}.
3. Créer deux hooks React personnalisés dans src/features/portfolios/hooks/ :
   * useMyPortfolio.ts: Utilise useQuery de TanStack Query pour appeler getMyPortfolio.
   * usePublicPortfolio.ts: Utilise useQuery pour appeler getPublicPortfolioBySlug.
4. Dans src/features/portfolios/components/, créer deux composants de présentation :
   * PortfolioHeader.tsx: Affiche le nom de l'utilisateur, une biographie et un commutateur pour rendre le portfolio public/privé.
   * PortfolioItemCard.tsx: Une carte qui affiche un seul projet du portfolio (titre, résumé, compétences, lien vers le résultat de l'évaluation).

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** BE-006.3.
* **Fichier(s) clé(s) à lire avant de commencer :** CDC Back-End pour le contrat des réponses de l'API du portfolio-service.
* **Information(s) clé(s) nécessaire(s) :** Les composants doivent être conçus pour être réutilisables et purement présentationnels.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 3.2 : Catalogue des Composants d'UI Fondamentaux.

**--- Livrables Attendus ---**

1. La nouvelle arborescence de dossiers src/features/portfolios/.
2. Le nouveau fichier de service portfolioService.ts.
3. Les nouveaux hooks useMyPortfolio.ts et usePublicPortfolio.ts.
4. Les nouveaux composants PortfolioHeader.tsx et PortfolioItemCard.tsx.
5. Un rapport de fin de tâche complet.

### **PROMPT DE TÂCHE : FE-005.2**

# **USER PROMPT - ASSIGNATION DE TÂCHE**

**ID de la Tâche :** FE-005.2 **Titre :** frontend : Assemblage de la Page de Portfolio (Privée et Publique)

**IA Spécialiste Assignée :** IA #3 - Frontend

**--- Contexte et État Actuel ---** La sous-tâche FE-005.1 est terminée. Nous disposons des hooks pour récupérer les données de portfolio et des composants de présentation (PortfolioHeader, PortfolioItemCard) pour les afficher.

Il manque maintenant les pages qui vont utiliser ces outils pour créer les vues finales que les utilisateurs verront.

**--- Objectif de la Tâche ---** Votre objectif est de créer les deux pages de consultation de portfolio : la vue privée pour l'utilisateur connecté et la vue publique pour les visiteurs.

1. Créer un nouveau composant de page, MyPortfolioPage.tsx, dans le répertoire apps/frontend/src/pages/.
2. Cette page doit utiliser le hook useMyPortfolio pour récupérer et afficher les données du portfolio de l'utilisateur actuellement connecté.
3. Créer un second composant de page, PublicPortfolioPage.tsx, dans le même répertoire.
4. Cette page doit utiliser le hook useParams pour extraire le slug de l'URL, puis le hook usePublicPortfolio pour récupérer et afficher les données du portfolio public correspondant.
5. Les deux pages doivent gérer correctement les états de chargement (isLoading) et d'erreur (isError).
6. Si les données sont récupérées avec succès, les pages doivent assembler les composants PortfolioHeader et une liste de PortfolioItemCard pour construire la vue.
7. Mettre à jour le routeur de l'application (App.tsx) pour ajouter deux nouvelles routes :
   * Une route **protégée** /portfolio/me qui pointe vers MyPortfolioPage.
   * Une route **publique** /p/:slug qui pointe vers PublicPortfolioPage.

**--- Prérequis et Dépendances ---**

* **Tâche(s) précédente(s) complétée(s) :** FE-005.1.
* **Fichier(s) clé(s) à lire avant de commencer :** Les hooks et composants créés dans la tâche précédente.
* **Information(s) clé(s) nécessaire(s) :** La page MyPortfolioPage doit afficher les contrôles d'administration (comme le commutateur de visibilité), tandis que la page PublicPortfolioPage ne doit pas les afficher.

**--- Références aux Documents Clés ---**

* CDC Front-End, Partie 6 : Routage

**--- Livrables Attendus ---**

1. Le contenu complet des nouveaux fichiers apps/frontend/src/pages/MyPortfolioPage.tsx et apps/frontend/src/pages/PublicPortfolioPage.tsx.
2. Les modifications apportées au fichier de routage (src/App.tsx).
3. Un rapport de fin de tâche complet.