### **Cahier des Charges Technique et d'Implémentation Back-End - SkillForge AI**

* **Version :** 1.0
* **Date :** 8 juin 2025
* **Propriétaire :** Lead Développeur Back-End

### **Partie 1 : Introduction et Rôle dans l'Écosystème**

#### **1.1. Objectif de ce Document**

Ce document est le **guide de construction et la référence unique et complète** pour l'ensemble des composants Back-End du projet SkillForge AI. Il a force de loi pour toutes les décisions de conception, de développement, de test, de déploiement et de maintenance des logiques métier côté serveur.

Son objectif n'est pas de rester théorique, mais de fournir des directives techniques précises et actionnables pour garantir que chaque ligne de code contribue à un système :

* **Robuste :** Fiable, résilient aux pannes et capable de gérer les erreurs de manière prévisible.
* **Performant :** Rapide, optimisé et capable de monter en charge pour répondre à la demande.
* **Sécurisé :** Protégeant les données des utilisateurs et l'intégrité de la plateforme contre les menaces.
* **Maintenable :** Clair, bien structuré et documenté pour faciliter les évolutions futures et l'intégration de nouveaux développeurs.

Ce CDC a pour finalité de permettre à l'équipe Back-End de travailler avec un maximum d'efficacité et d'autonomie, en éliminant l'ambiguïté et en assurant une qualité logicielle non-négociable.

#### **1.2. Audience Cible**

Ce document s'adresse principalement aux profils techniques suivants :

* **Audience Primaire :**
  + **Développeurs Back-End (Python/FastAPI) :** Ce document est leur manuel de référence quotidien. Ils doivent le consulter pour toute nouvelle implémentation ou modification significative.
* **Audience Secondaire :**
  + **Architectes Logiciels :** Pour valider la conformité des implémentations avec la vision architecturale globale définie dans le GAG.
  + **Ingénieurs DevOps/SRE :** Pour comprendre les besoins spécifiques des services Back-End en matière de build, de configuration, de déploiement et de monitoring.
  + **Lead Développeurs Front-End & IA :** Pour comprendre les frontières, les capacités et les contrats d'interaction du système Back-End.
  + **Chefs de Projet Technique :** Pour le suivi et la validation de la mise en œuvre technique.

#### **1.3. Articulation avec le GAG et les Autres Documents**

Ce Cahier des Charges est une pièce maîtresse de l'écosystème documentaire du projet, mais il n'opère pas en vase clos. Il est subordonné au **Guide d'Architecture Générale (GAG)**, qui agit comme le "Hub" (moyeu) dont ce CDC est un "Spoke" (rayon).

Les relations et les frontières avec les autres documents clés sont les suivantes, afin de respecter le principe de **Source Unique de Vérité** :

* **Relation avec le Guide d'Architecture Générale (GAG) :**
  + Le **GAG** définit les principes fondamentaux (ex: "l'architecture est en microservices", "la communication asynchrone est obligatoire pour les tâches longues").
  + Ce **CDC Back-End** **implémente** ces principes (ex: "voici la liste des microservices et leurs responsabilités", "voici le registre des événements asynchrones").
* **Relation avec le CDC Données (EDR) :**
  + Le **CDC Données** est la source de vérité absolue pour le **schéma de la base de données** (tables, colonnes, relations).
  + Ce **CDC Back-End** décrit **comment interagir avec ce schéma** via l'ORM SQLAlchemy, et définit les conventions pour les migrations (Alembic). Il ne redéfinit jamais la structure de la base de données.
* **Relation avec la Spécification OpenAPI (openapi.json) :**
  + La **Spécification OpenAPI** (générée par le code) est le **contrat contractuel** pour la syntaxe, les paramètres et les schémas de réponse de chaque endpoint de l'API REST.
  + Ce **CDC Back-End** définit les **conventions générales de l'API** (versioning, authentification, format des erreurs) et fournit un catalogue de haut niveau pour la compréhension, mais se réfère toujours à la spécification générée comme vérité technique finale.
* **Relation avec le CDC DevOps :**
  + Le **CDC DevOps** décrit **comment** l'infrastructure est provisionnée, comment les pipelines CI/CD fonctionnent et comment le monitoring est configuré.
  + Ce **CDC Back-End** décrit **ce dont les services Back-End ont besoin** de la part de cette infrastructure (ex: "voici la liste des variables d'environnement requises", "voici les étapes de build et de test à intégrer dans le pipeline CI").

### **Partie 2 : Architecture Détaillée des Microservices**

Cette section détaille la structure interne du Back-End. Elle définit les composants, leurs responsabilités et la manière dont ils collaborent pour mettre en œuvre la logique métier de SkillForge AI.

#### **2.1. Vue d'Ensemble Logique et Interactions**

##### **2.1.1. Description Lexicale pour Génération de Diagramme**

Cette description sert de base à la génération du diagramme d'architecture logique du Back-End.

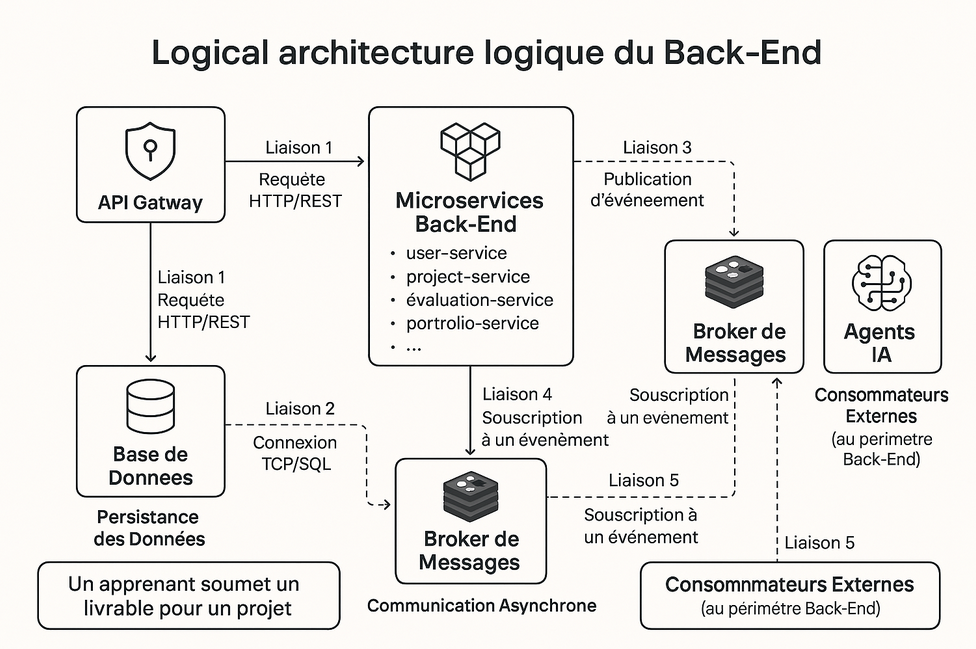
**A. Composants Fondamentaux (Les "Nœuds" du Diagramme) :**

* **Périmètre d'Entrée : API Gateway**
  + **Description :** Le point d'entrée unique et obligatoire pour toutes les communications synchrones provenant de clients externes (Application Front-End). Il se situe à la frontière du système Back-End. Sa responsabilité est la redirection des requêtes, pas la logique métier.
* **Cœur Logique : Le Cluster de Microservices Back-End**
  + **Description :** Un groupe logique de services applicatifs indépendants qui encapsulent la logique métier du projet. Ils sont le cerveau du système. Les services identifiés pour le MVP sont :
    - user-service
    - project-service
    - evaluation-service
    - portfolio-service
    - notification-service
    - messaging-service
* **Persistance des Données : La Base de Données (PostgreSQL)**
  + **Description :** Le système de gestion de base de données relationnelle unique et partagé. Il est la source de vérité pour toutes les données structurées et persistantes de l'application (utilisateurs, projets, évaluations, etc.).
* **Communication Asynchrone : Le Broker de Messages (Redis Pub/Sub)**
  + **Description :** Le système de messagerie qui sert de colonne vertébrale pour toutes les communications asynchrones et événementielles. Il découple les producteurs d'événements des consommateurs.
* **Consommateurs Externes (au périmètre Back-End) : Les Agents IA**
  + **Description :** Bien que faisant partie du projet global, les Agents IA sont considérés comme des consommateurs externes du point de vue de l'architecture Back-End. Ils écoutent les événements émis par le Back-End mais ne sont pas directement appelés par lui de manière synchrone.

**B. Liaisons et Flux (Les "Flèches" du Diagramme) :**

* **Liaison 1 : API Gateway → Microservices Back-End**
  + **Type :** Synchrone (Requête HTTP/REST).
  + **Direction :** Unidirectionnelle. Le Gateway appelle les services, jamais l'inverse.
  + **Description :** Le Gateway route les requêtes entrantes vers le microservice approprié en fonction du chemin de l'URL (ex: /api/v1/projects/\* est routé vers project-service).
* **Liaison 2 : Microservices Back-End ↔ Base de Données (PostgreSQL)**
  + **Type :** Synchrone (Connexion TCP/SQL).
  + **Direction :** Bidirectionnelle. Les services lisent et écrivent dans la base de données.
  + **Description :** Chaque microservice qui nécessite de persister ou de récupérer des données établit une connexion directe à la base de données via l'ORM.
* **Liaison 3 : Microservices Back-End → Broker de Messages (Redis Pub/Sub)**
  + **Type :** Asynchrone (Publication d'événement).
  + **Direction :** Unidirectionnelle (Publish).
  + **Description :** Lorsqu'un service exécute une action nécessitant un traitement ultérieur découplé, il publie un message (événement) sur un topic spécifique dans le Broker.
* **Liaison 4 : Broker de Messages (Redis Pub/Sub) → Microservices Back-End**
  + **Type :** Asynchrone (Souscription à un événement).
  + **Direction :** Unidirectionnelle (Subscribe).
  + **Description :** Un microservice peut s'abonner à des événements (y compris ceux émis par d'autres microservices) pour déclencher des actions en cascade. Par exemple, le notification-service s'abonne à des événements pour savoir quand envoyer une notification.
* **Liaison 5 : Broker de Messages (Redis Pub/Sub) → Agents IA**
  + **Type :** Asynchrone (Souscription à un événement).
  + **Direction :** Unidirectionnelle (Subscribe).
  + **Description :** C'est le principal canal d'activation pour les Agents IA. Ils s'abonnent à des topics spécifiques (ex: project.deliverable.submitted) pour démarrer leurs tâches d'analyse.

##### **2.1.2. Description Détaillée des Interactions**



Pour illustrer comment ces composants collaborent, prenons le scénario métier le plus représentatif : **"Un apprenant soumet un livrable pour un projet"**.

**Étape 1 : Le Traitement Synchrone (Réponse rapide à l'utilisateur)**

1. Le Front-End envoie une requête POST /api/v1/projects/{id}/deliverables avec le fichier et les métadonnées vers l'**API Gateway**.
2. Le Gateway valide le jeton JWT et route la requête vers le **project-service**.
3. Le **project-service** exécute les actions immédiates :
   * Il valide les données de la requête via son modèle Pydantic.
   * Il vérifie les droits de l'utilisateur (cet apprenant a-t-il le droit de soumettre à ce projet ?).
   * Il enregistre les métadonnées du livrable (nom, date, statut "SOUMIS") dans la **Base de Données (PostgreSQL)**.
   * Il télécharge le fichier binaire vers Google Cloud Storage.
4. Le **project-service** renvoie immédiatement une réponse 202 Accepted au Front-End. Le code 202 est utilisé intentionnellement pour signifier "La requête a été acceptée, mais le traitement complet n'est pas terminé". Cela garantit une expérience utilisateur fluide et non bloquante.

**Étape 2 : Le Déclenchement Asynchrone** 5. Immédiatement après avoir renvoyé la réponse 202, le **project-service** construit un message JSON contenant les informations nécessaires (ID du livrable, chemin du fichier, ID de l'utilisateur). 6. Il publie ce message sur le topic project.deliverable.submitted dans le **Broker de Messages (Redis Pub/Sub)**. La responsabilité synchrone du project-service pour cette requête est maintenant terminée.

**Étape 3 : La Chaîne de Traitement Événementielle (Le "travail de fond")** 7. L'**evaluation-agent** (un Agent IA), qui est abonné au topic project.deliverable.submitted, reçoit le message. 8. Il télécharge le fichier depuis Cloud Storage, effectue sa longue et complexe tâche d'analyse et d'évaluation, puis met à jour la ligne correspondante dans la **Base de Données** avec le score, les commentaires et le statut "ÉVALUÉ". 9. Une fois son travail terminé, l'**evaluation-agent** publie à son tour un nouvel événement, evaluation.result.generated, dans le **Broker de Messages**.

10. Le **notification-service**, qui est abonné à evaluation.result.generated, reçoit ce nouvel événement. Il récupère les détails nécessaires depuis la **Base de Données** et envoie une notification (e-mail, in-app) à l'apprenant pour l'informer que son évaluation est disponible.

**Conclusion de ce flux :** Ce pattern architectural (Réponse synchrone rapide → Traitement asynchrone découplé → Notification événementielle) est au cœur de la conception du Back-End. Il garantit la réactivité de l'interface utilisateur tout en permettant des traitements complexes, résilients et scalables en arrière-plan.

#### **2.2. Registre des Microservices**

Chaque microservice est un composant déployable indépendamment, avec un périmètre de responsabilité strict. Cette séparation est la clé de notre agilité et de notre résilience.

##### **2.2.1. user-service**

* **Nom du Service :** user-service
* **Mission :** Gérer l'identité, l'authentification, les profils et le cycle de vie de toutes les entités "utilisateur" (Apprenants, Entreprises, Administrateurs) de la plateforme.
* **Périmètre Fonctionnel :**
  + Fournir les endpoints pour l'inscription des Apprenants et des Entreprises.
  + Gérer la logique de connexion (vérification de l'email et du mot de passe haché avec bcrypt).
  + Générer et signer les jetons d'accès JWT lors d'une connexion réussie.
  + Exposer un endpoint de validation de jeton JWT pour les autres services ou l'API Gateway.
  + Gérer le processus de réinitialisation de mot de passe (génération de jeton de réinitialisation, validation).
  + Gérer les opérations CRUD (Create, Read, Update, Delete) sur les profils utilisateurs (nom, biographie, URL de l'avatar, etc.).
  + Gérer le statut des comptes (ex: PENDING\_VALIDATION pour les entreprises, ACTIVE, BANNED).
  + Publier les événements liés au cycle de vie des utilisateurs (ex: user.account.created).
* **Dépendances Clés :**
  + **Base de Données (PostgreSQL) :** Pour la persistance de toutes les informations des utilisateurs.
  + **Broker de Messages (Redis) :** Pour publier les événements de cycle de vie.
  + **Google Cloud Storage :** Pour stocker les avatars des utilisateurs.

##### **2.2.2. project-service**

* **Nom du Service :** project-service
* **Mission :** Gérer l'intégralité du cycle de vie des projets, depuis leur soumission par une entreprise jusqu'à la gestion des livrables par un apprenant.
* **Périmètre Fonctionnel :**
  + Fournir les endpoints pour les opérations CRUD sur les projets par les entreprises.
  + Gérer les différents statuts d'un projet (DRAFT, PENDING\_VALIDATION, ACTIVE, COMPLETED, ARCHIVED).
  + Implémenter la logique de validation des projets par les administrateurs.
  + Fournir des endpoints de recherche, de filtrage et de consultation des projets pour les apprenants.
  + Gérer l'assignation d'un projet à un apprenant.
  + Gérer les jalons (milestones) définis pour un projet.
  + Fournir l'endpoint de soumission des livrables par les apprenants.
  + Orchestrer le stockage des fichiers de livrables sur Google Cloud Storage.
  + Publier l'événement project.deliverable.submitted après une soumission réussie.
* **Dépendances Clés :**
  + **Base de Données (PostgreSQL) :** Pour la persistance des projets, jalons, et métadonnées des livrables.
  + **Broker de Messages (Redis) :** Pour publier les événements liés aux projets.
  + **user-service :** Pour vérifier les permissions (une entreprise peut-elle créer un projet ? un apprenant peut-il s'y assigner ?).
  + **Google Cloud Storage :** Pour le stockage des fichiers joints aux projets et des livrables soumis.

##### **2.2.3. evaluation-service**

* **Nom du Service :** evaluation-service
* **Mission :** Servir de point de stockage et de gestion unique pour les résultats des évaluations produites par les Agents IA.
* **Périmètre Fonctionnel :**
  + Fournir un endpoint d'API interne et sécurisé, destiné à être appelé exclusivement par les Agents IA pour soumettre les résultats d'une évaluation.
  + Stocker de manière structurée les données d'évaluation : score global, scores par critère, commentaires textuels, points forts, axes d'amélioration.
  + Associer chaque évaluation à un livrable et un utilisateur spécifiques.
  + Fournir les endpoints permettant de consulter les évaluations terminées.
  + Calculer et exposer des statistiques d'évaluation agrégées sur demande (ex: score moyen d'un apprenant).
  + Publier l'événement evaluation.result.generated une fois qu'un résultat est enregistré avec succès.
* **Dépendances Clés :**
  + **Base de Données (PostgreSQL) :** Pour la persistance de toutes les données d'évaluation.
  + **Broker de Messages (Redis) :** Pour notifier le reste du système qu'une évaluation est disponible.

##### **2.2.4. portfolio-service**

* **Nom du Service :** portfolio-service
* **Mission :** Gérer la construction, la persistance et l'exposition des portfolios publics des apprenants.
* **Périmètre Fonctionnel :**
  + Fournir un endpoint d'API interne et sécurisé pour que l'Agent IA de construction de portfolio puisse soumettre le contenu généré.
  + Stocker les éléments du portfolio : descriptions de projet synthétisées, listes de compétences (tags), URL unique et publique.
  + Associer les éléments du portfolio aux projets correspondants qui sont complétés et validés.
  + Gérer le statut public ou privé du portfolio, contrôlé par l'apprenant.
  + Fournir l'endpoint public (non authentifié) pour afficher les données d'un portfolio via son URL unique.
* **Dépendances Clés :**
  + **Base de Données (PostgreSQL) :** Pour stocker les données des portfolios.
  + **project-service et evaluation-service :** Pour récupérer les informations sources (projets réussis, scores) nécessaires à la logique de construction du portfolio.

##### **2.2.5. notification-service**

* **Nom du Service :** notification-service
* **Mission :** Centraliser et orchestrer l'envoi de toutes les notifications aux utilisateurs, sur tous les canaux (e-mail, in-app).
* **Périmètre Fonctionnel :**
  + Être le principal consommateur d'événements métier provenant du **Broker de Messages**. Il s'abonne à une liste exhaustive de topics (ex: evaluation.result.generated, messaging.message.received, etc.).
  + Gérer une bibliothèque de modèles de notifications (templates) pour les e-mails.
  + Intégrer et communiquer avec un fournisseur de services de messagerie tiers (ex: SendGrid, Mailgun) pour l'envoi des e-mails.
  + Stocker l'historique des notifications "in-app" pour chaque utilisateur.
  + Fournir les endpoints permettant à un utilisateur de lister ses notifications "in-app" et de les marquer comme lues.
* **Dépendances Clés :**
  + **Broker de Messages (Redis) :** Sa source d'information principale pour déclencher les envois.
  + **Base de Données (PostgreSQL) :** Pour l'historique et le statut des notifications in-app.
  + **user-service :** Pour récupérer les informations de contact de l'utilisateur (adresse e-mail, préférences de notification).
  + **Service Tiers d'Email :** Dépendance externe critique.

##### **2.2.6. messaging-service**

* **Nom du Service :** messaging-service
* **Mission :** Gérer la fonctionnalité de messagerie interne privée entre les apprenants et les représentants des entreprises dans le contexte d'un projet.
* **Périmètre Fonctionnel :**
  + Fournir les endpoints pour créer une conversation, envoyer un message texte et joindre des fichiers.
  + Gérer la logique des conversations (threads), en les liant à un projet et à ses participants.
  + Stocker l'historique complet des messages de manière persistante.
  + Assurer l'orchestration du stockage des pièces jointes sur Google Cloud Storage.
  + Publier l'événement messaging.message.received pour chaque nouveau message envoyé.
* **Dépendances Clés :**
  + **Base de Données (PostgreSQL) :** Pour la persistance des conversations et des messages.
  + **Broker de Messages (Redis) :** Pour notifier en temps réel qu'un message a été envoyé.
  + **user-service et project-service :** Pour la logique d'autorisation (vérifier qu'un utilisateur a bien le droit de participer à une conversation liée à un projet).
  + **Google Cloud Storage :** Pour stocker les fichiers joints aux messages.

### **Partie 3 : Contrats de Communication et APIs**

Cette section définit les règles et les catalogues de communication. Tout échange de données, qu'il soit synchrone (API REST) ou asynchrone (Événements), doit se conformer strictement aux contrats définis ci-après.

#### **3.1. API REST (Communication Synchrone)**

La communication synchrone est utilisée pour les actions initiées par l'utilisateur et nécessitant une réponse immédiate.

##### **3.1.1. Règles d'Or et Standards**

Conformément au GAG, toutes nos APIs REST respectent les règles suivantes :

1. **Source de Vérité :** La spécification **OpenAPI 3.0** (openapi.json) générée par le code est le seul contrat technique faisant foi. La documentation doit toujours être synchronisée avec le code.
2. **Authentification :** Toutes les routes sécurisées doivent être appelées avec un header Authorization: Bearer <JWT>.
3. **Versioning :** Toutes les routes sont versionnées dans l'URL, avec le préfixe /api/v1/.
4. **Format des Erreurs :** Les réponses d'erreur (4xx/5xx) doivent impérativement suivre la structure { "error": { "code": "CODE\_ERREUR", "message": "Description." } }.

##### **3.1.2. URL de Base par Environnement**

* **Local :** http://localhost:8000
* **Staging :** https://api.staging.skillforge.ai
* **Production :** https://api.skillforge.ai

##### **3.1.3. Catalogue des Endpoints Principaux du MVP**

La liste suivante fournit une vue d'ensemble des routes principales. Les détails des requêtes et des réponses sont dans la spécification OpenAPI.

**Ressource : Authentification & Utilisateurs (user-service)**

* POST /api/v1/auth/register : Inscription d'un nouvel utilisateur (Apprenant ou Entreprise).
* POST /api/v1/auth/login : Connexion d'un utilisateur et récupération d'un JWT.
* POST /api/v1/auth/refresh-token : Obtention d'un nouveau jeton d'accès à partir d'un jeton de rafraîchissement.
* POST /api/v1/auth/request-password-reset : Déclenche l'envoi d'un e-mail de réinitialisation de mot de passe.
* POST /api/v1/auth/reset-password : Finalise la réinitialisation du mot de passe avec un jeton valide.
* GET /api/v1/users/me : Récupère le profil de l'utilisateur actuellement authentifié.
* PUT /api/v1/users/me : Met à jour le profil de l'utilisateur actuellement authentifié.

**Ressource : Projets (project-service)**

* POST /api/v1/projects : Création d'un nouveau projet (requiert le rôle 'entreprise').
* GET /api/v1/projects : Liste les projets actifs, avec filtres et pagination.
* GET /api/v1/projects/{project\_id} : Récupère les détails d'un projet spécifique.
* PUT /api/v1/projects/{project\_id} : Met à jour un projet (requiert d'être le propriétaire ou 'admin').
* POST /api/v1/projects/{project\_id}/assign : Permet à un apprenant de s'assigner à un projet disponible.

**Ressource : Livrables (project-service)**

* POST /api/v1/projects/{project\_id}/deliverables : Soumission d'un nouveau livrable pour un projet (requiert le rôle 'apprenant' et d'être assigné).
* GET /api/v1/projects/{project\_id}/deliverables : Liste les livrables soumis pour un projet.

**Ressource : Évaluations (evaluation-service)**

* GET /api/v1/deliverables/{deliverable\_id}/evaluation : Récupère le résultat de l'évaluation pour un livrable spécifique.

**Ressource : Portfolios (portfolio-service)**

* GET /api/v1/portfolios/me : Récupère les données du portfolio de l'apprenant authentifié.
* PUT /api/v1/portfolios/me/status : Modifie le statut (public/privé) du portfolio.
* GET /api/v1/portfolios/{public\_url} : Endpoint public pour afficher les données d'un portfolio.

**Ressource : Notifications (notification-service)**

* GET /api/v1/notifications : Récupère la liste des notifications in-app de l'utilisateur.
* POST /api/v1/notifications/mark-as-read : Marque une ou plusieurs notifications comme lues.

**Ressource : Messagerie (messaging-service)**

* GET /api/v1/projects/{project\_id}/messages : Récupère les messages d'une conversation liée à un projet.
* POST /api/v1/projects/{project\_id}/messages : Envoie un message dans une conversation.

**Ressource : Administration**

* POST /api/v1/admin/projects/{project\_id}/validate : Valide et active un projet soumis par une entreprise.
* GET /api/v1/admin/users : Liste tous les utilisateurs avec filtres.
* PUT /api/v1/admin/users/{user\_id}/status : Modifie le statut d'un compte utilisateur.

#### **3.2. Événements (Communication Asynchrone)**

La communication asynchrone est utilisée pour les processus longs et pour découpler les services. Le tableau suivant est le **Registre Officiel des Événements** du projet.

| Nom du Topic (Canal) | Service Émetteur | Services Consommateurs | Description du Payload JSON |
| --- | --- | --- | --- |
| user.account.created | user-service | notification-service | { "user\_id": "uuid", "email": "string", "full\_name": "string", "role": "string" }. Envoyé pour déclencher l'e-mail de bienvenue. |
| user.password.reset\_requested | user-service | notification-service | { "user\_id": "uuid", "email": "string", "reset\_token": "string" }. Envoyé pour déclencher l'e-mail avec le lien de réinitialisation. |
| project.project.submitted | project-service | notification-service | { "project\_id": "uuid", "project\_title": "string", "company\_name": "string" }. Envoyé pour notifier les administrateurs qu'un nouveau projet est en attente de validation. |
| project.deliverable.submitted | project-service | evaluation-agent, notification-service | { "deliverable\_id": "uuid", "project\_id": "uuid", "user\_id": "uuid", "file\_path\_in\_gcs": "string" }. L'événement principal qui déclenche le pipeline d'évaluation et notifie l'entreprise. |
| evaluation.result.generated | evaluation-service | notification-service, portfolio-agent | { "evaluation\_id": "uuid", "deliverable\_id": "uuid", "user\_id": "uuid", "project\_id": "uuid", "overall\_score": "float" }. Indique qu'une évaluation est terminée et prête à être consultée. Déclenche la notification à l'apprenant et potentiellement une mise à jour du portfolio. |
| messaging.message.received | messaging-service | notification-service | { "message\_id": "uuid", "conversation\_id": "uuid", "sender\_id": "uuid", "recipient\_id": "uuid", "project\_id": "uuid" }. Permet de notifier un utilisateur qu'il a reçu un nouveau message. |

### **Partie 4 : Persistance et Gestion des Données**

Cette section définit les standards et les conventions pour toute interaction avec les systèmes de stockage de données, incluant la base de données relationnelle et le système de cache.

#### **4.1. Interaction avec la Base de Données**

##### **4.1.1. Référence à la Source de Vérité (CDC Données)**

Il est impératif de rappeler que ce document ne définit PAS le schéma de la base de données. Le **CDC Données (EDR)** est la **source unique de vérité** pour la structure des tables, les types de colonnes, les relations et les contraintes. Ce CDC Back-End définit uniquement les **méthodes et conventions d'interaction** avec ce schéma.

##### **4.1.2. ORM et Accès aux Données**

1. **Technologie Standard :** L'accès à la base de données PostgreSQL DOIT se faire exclusivement via l'ORM **SQLAlchemy** en version **2.0** ou supérieure, en utilisant son paradigme moderne de requêtage.
2. **Driver Asynchrone :** Le driver de base de données à utiliser est **asyncpg**, pour garantir des performances optimales avec le framework asyncio.
3. **Gestion des Sessions :** Toutes les opérations de base de données (lecture, écriture) au sein d'une requête API doivent être exécutées à travers une unique session asynchrone (AsyncSession). Cette session doit être gérée via le système d'injection de dépendances de FastAPI pour garantir qu'elle soit correctement ouverte et fermée pour chaque requête.
4. **Modèles :** L'utilisation de **SQLModel** est recommandée car il combine les modèles SQLAlchemy et les modèles Pydantic, réduisant ainsi la duplication de code entre la couche de persistance et la couche de validation de l'API.

##### **4.1.3. Migrations de Schéma avec Alembic**

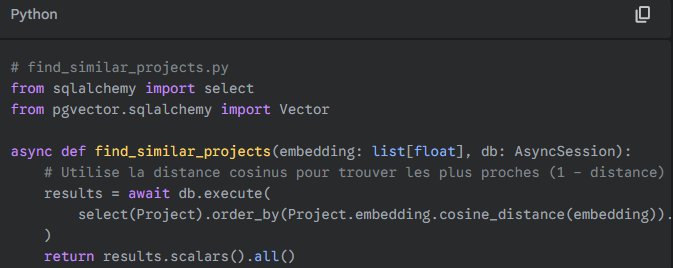
Toute modification du schéma de la base de données est un processus critique qui doit suivre un workflow strict et non-négociable.

1. **Outil Obligatoire :** **Alembic** est le seul outil autorisé pour gérer les migrations de schéma.
2. **Workflow de Migration :**
   * **Étape A - Modification du Modèle :** Le développeur modifie un modèle SQLModel/SQLAlchemy dans le code.
   * **Étape B - Génération du Script :** Le développeur génère un nouveau script de migration avec la commande : alembic revision --autogenerate -m "Description concise et claire de la modification".
   * **Étape C - Revue Manuelle OBLIGATOIRE :** Le développeur doit **impérativement relire et valider** le script de migration généré par Alembic pour s'assurer qu'il correspond exactement à l'intention de la modification et qu'il ne comporte aucun risque de perte de données.
   * **Étape D - Commit :** Le script de migration est commité dans le dépôt Git en même temps que la modification du modèle correspondant.
   * **Étape E - Déploiement :** Le pipeline de CI/CD exécutera automatiquement la commande alembic upgrade head avant le démarrage de la nouvelle version de l'application, appliquant ainsi la migration à la base de données de l'environnement cible.

##### **4.1.4. Utilisation de pgvector pour la Recherche de Similarité**

L'extension pgvector est utilisée pour les fonctionnalités de suggestion de projets.

1. **Type de Données :** Dans les modèles SQLModel/SQLAlchemy, les colonnes destinées à stocker des embeddings DOIVENT utiliser le type Vector importé depuis pgvector.sqlalchemy. La dimension des vecteurs doit être spécifiée (ex: Vector(384) pour all-MiniLM-L6-v2).
2. **Exemple de Requête de Similarité :** Pour trouver les projets les plus similaires à un vecteur donné, la requête doit utiliser une des fonctions de distance fournies par l'extension. Exemple :



#### **4.2. Stratégie de Cache**

Le cache est utilisé pour améliorer la latence et réduire la charge sur la base de données pour les données fréquemment lues.

1. **Technologie :** **Redis** est la technologie de cache unique pour le MVP.
2. **Cas d'Usage Approuvés :**
   * **Données de Profil Utilisateur :** Les données publiques d'un utilisateur, fréquemment demandées, peuvent être mises en cache.
   * **Détails de Projets Actifs :** Les informations d'un projet publié et actif, qui sont rarement modifiées, sont un candidat idéal pour le cache.
   * **Configuration de la Plateforme :** Toute configuration globale stockée en base de données doit être mise en cache au démarrage du service.
   * **Limitation de Débit (Rate Limiting) :** Redis peut être utilisé pour stocker les compteurs de requêtes pour les logiques de limitation de débit.
3. **Convention de Nommage des Clés :** Pour éviter les collisions et faciliter le débogage, toutes les clés de cache DOIVENT suivre le format : <nom\_du\_service>:<type\_de\_ressource>:<identifiant\_unique>.
   * Exemple 1 : user-service:user-profile:f47ac10b-58cc-4372-a567-0e02b2c3d479
   * Exemple 2 : project-service:project-details:c4f5d6e8-2e1a-4b9c-8c6f-3d1b7a9f0e2d
4. **Stratégie d'Invalidation et TTL (Time-To-Live) :**
   * **Stratégie par Défaut : Expiration Basée sur le Temps (TTL).** Chaque clé insérée dans le cache doit avoir un TTL. Les TTL par défaut sont :
     + Données de session/profil : 5 minutes.
     + Données de projet/ressource : 1 heure.
   * **Stratégie d'Invalidation Explicite :** Pour les données critiques, une stratégie d'invalidation explicite doit être implémentée en plus du TTL. Lorsque une ressource est mise à jour (via PUT, POST, DELETE), le code responsable de la modification DOIT explicitement supprimer la clé de cache correspondante. Par exemple, lors d'un appel à PUT /api/v1/projects/{id}, la clé project-service:project-details:{id} doit être effacée de Redis.

### **Partie 5 : Conventions de Développement et Qualité du Code**

#### **5.1. Structure des Projets (Structure de Dossiers)**

Afin d'assurer une cohérence totale entre les microservices, chaque service FastAPI doit impérativement respecter la structure de dossiers suivante. Cette organisation est basée sur les meilleures pratiques et favorise une séparation claire des préoccupations.

**Exemple pour un service (service-name):**

service-name/

├── app/

│ ├── \_\_init\_\_.py

│ ├── main.py # Point d'entrée de l'application FastAPI, middlewares, routers

│ ├── api/ # Couche API : routers et endpoints

│ │ └── v1/

│ │ └── endpoints/

│ │ └── resource.py # Router pour une ressource (ex: projects.py)

│ ├── core/ # Logique de base : configuration, clients externes

│ │ └── config.py # Configuration avec Pydantic-Settings

│ ├── crud/ # Couche d'accès aux données (Create, Read, Update, Delete)

│ │ └── crud\_resource.py # Fonctions d'interaction directe avec l'ORM

│ ├── models/ # Modèles de base de données (SQLAlchemy/SQLModel)

│ │ └── resource\_model.py

│ ├── schemas/ # Schémas de validation de données API (Pydantic)

│ │ └── resource\_schema.py

│ └── services/ # Couche de logique métier (Business Logic)

│ └── business\_service.py

├── tests/ # Ensemble des tests pour ce service

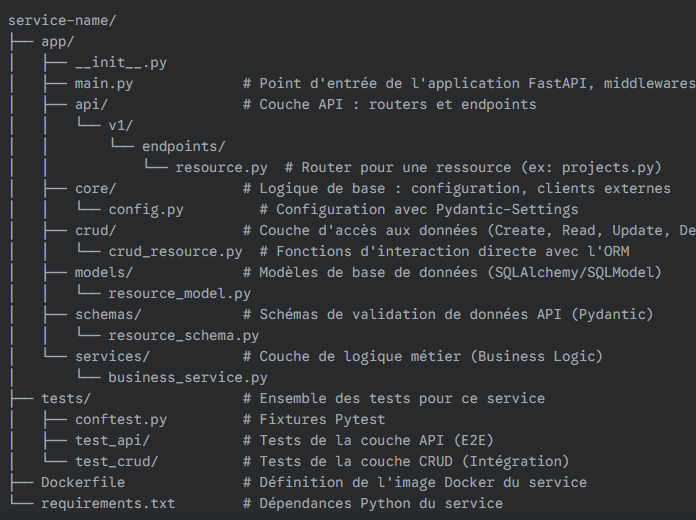
│ ├── conftest.py # Fixtures Pytest

│ ├── test\_api/ # Tests de la couche API (E2E)

│ └── test\_crud/ # Tests de la couche CRUD (Intégration)

├── Dockerfile # Définition de l'image Docker du service

└── requirements.txt # Dépendances Python du service



* **api/** : Contient uniquement la logique de routage et de gestion des requêtes/réponses HTTP.
* **crud/** : Contient la logique d'interaction directe avec la base de données via l'ORM. Aucune logique métier ne doit s'y trouver.
* **services/** : Contient la logique métier complexe, orchestrant les appels à la couche crud et à d'autres services.
* **models/ vs schemas/** : models définit la structure de la table en BDD. schemas définit la forme des données attendues en entrée et en sortie de l'API.

#### **5.2. Standards de Code**

L'homogénéité du code est non-négociable. Nous utilisons une suite d'outils pour l'automatiser et la garantir.

1. **Formatage du Code : black**
   * **Règle :** L'intégralité du code Python doit être formatée avec black en utilisant sa configuration par défaut.
   * **Application :** black sera exécuté via un "pre-commit hook" Git, empêchant tout commit de code non formaté.
2. **Qualité du Code (Linting) : ruff**
   * **Règle :** ruff est notre linter de choix pour sa rapidité et son exhaustivité. Il remplace flake8 et isort.
   * **Application :** Une configuration ruff stricte sera définie à la racine du monorepo. Il sera également exécuté en tant que "pre-commit hook" et dans le pipeline de CI. Toute violation bloquera la pull request.
3. **Typage Statique : mypy**
   * **Règle :** Le typage statique est obligatoire. Toute fonction ou méthode, sans exception, doit avoir ses arguments et son type de retour annotés.
   * **Application :** mypy sera exécuté en mode strict dans le pipeline de CI. Un code qui ne passe pas la validation de mypy sera rejeté.

#### **5.3. Gestion des Dépendances**

Pour garantir des builds déterministes et éviter les conflits, la gestion des dépendances est soumise à des règles strictes.

1. **Fichier par Service :** Chaque service et chaque agent doit posséder son propre fichier requirements.txt.
2. **Versions Épinglées (Pinned Versions) :** Toutes les dépendances listées dans requirements.txt doivent impérativement utiliser un opérateur d'égalité stricte (==).
   * **Exemple :** fastapi==0.111.0, sqlalchemy==2.0.30.
3. **Gestion des Sous-Dépendances :** Il est fortement recommandé d'utiliser un outil comme pip-tools pour compiler un fichier requirements.in (dépendances de haut niveau) en un fichier requirements.txt complet et épinglé, incluant toutes les sous-dépendances.

#### **5.4. Configuration et Secrets**

La gestion de la configuration et des secrets est un pilier de notre sécurité et de notre flexibilité.

1. **Configuration de l'Application : pydantic-settings**
   * **Règle :** La configuration de chaque service (URL de la base de données, niveaux de log, etc.) doit être gérée par une classe héritant de BaseSettings du paquet pydantic-settings.
   * **Source :** Cette classe chargera sa configuration **exclusivement à partir des variables d'environnement**. Cela respecte le principe n°3 du "Twelve-Factor App".
2. **Politique "Zéro Secret en Clair"**
   * **INTERDICTION FORMELLE** de stocker quelque secret que ce soit (clé d'API, mot de passe, secret JWT) en clair dans le code source ou dans un fichier commité sur Git.
   * **TOLÉRANCE LIMITÉE** pour les fichiers .env : ils sont autorisés **uniquement** pour le développement local et doivent être listés dans le fichier .gitignore.
   * **OBLIGATION EN PRODUCTION :** Dans les environnements de staging et de production, les secrets sont chargés au démarrage du service **exclusivement et directement** depuis **Google Secret Manager**. Le CDC DevOps détaille la mise en place de cette infrastructure.

#### **5.5. Logging**

Des logs clairs et exploitables sont indispensables pour le débogage et le monitoring.

1. **Format Obligatoire : JSON Structuré**
   * **Règle :** Tous les logs émis par les services doivent être au format JSON. La bibliothèque python-json-logger est le standard à utiliser.
   * **Justification :** Le format JSON permet une ingestion, un parsage et une recherche efficaces dans notre système de logs centralisé (Google Cloud Logging).
2. **Champs Obligatoires par Entrée de Log :**
   * Chaque enregistrement de log, quel que soit son niveau, doit contenir au minimum les champs suivants :
     + timestamp : La date et l'heure UTC de l'événement.
     + level : Le niveau du log (ex: INFO, WARNING, ERROR).
     + message : Le message de log principal.
     + service\_name : Le nom du microservice qui émet le log (ex: project-service).
     + correlation\_id : Un identifiant unique généré pour chaque requête entrante à l'API Gateway. Cet ID doit être propagé à travers tous les appels de service et les événements asynchrones liés à cette requête initiale, permettant un traçage complet de bout en bout. Un middleware FastAPI sera chargé de sa gestion.

### **Partie 6 : Stratégie de Test**

Cette section définit l'ensemble des pratiques, outils et exigences en matière de tests logiciels pour tous les composants Back-End. L'objectif est d'instaurer une culture de la qualité où chaque développeur est responsable de la validation de son propre code. Nous adoptons le modèle de la **Pyramide de Tests** comme guide conceptuel.

#### **6.1. Framework et Outils**

Pour maintenir une cohérence et une efficacité maximales, l'écosystème de test est standardisé comme suit :

1. **Framework Principal : pytest**
   * **Règle :** pytest est le framework standard et unique pour l'écriture et l'exécution de tous les types de tests en Python.
2. **Tests Asynchrones : pytest-asyncio**
   * **Règle :** Le plugin pytest-asyncio est obligatoire pour tester correctement les coroutines et les logiques basées sur asyncio.
3. **Isolation des Composants (Mocking) : unittest.mock**
   * **Règle :** Pour les tests unitaires, l'isolation des dépendances externes doit être réalisée à l'aide du module unittest.mock de la bibliothèque standard.
4. **Génération de Données de Test : Faker**
   * **Recommandation :** L'utilisation de la bibliothèque Faker est fortement encouragée pour générer des données de test réalistes.

#### **6.2. Typologie des Tests Obligatoires (La Pyramide de Tests)**

Chaque fonctionnalité doit être couverte par plusieurs types de tests, avec un accent mis sur les tests les plus rapides et les moins coûteux à la base de la pyramide.

##### **A. Tests Unitaires (Base de la pyramide)**

* **Objectif :** Valider la correction logique d'une unité de code (une fonction, une méthode) de manière totalement isolée de ses dépendances.
* **Périmètre :** Ils sont rapides, nombreux et couvrent la logique métier. Toutes les dépendances externes (BDD, réseau) DOIVENT être "mockées".
* **Exemple Concret de Script :**
  + **Scénario :** Tester la fonction can\_archive\_project du project\_service.
  + **Fichier :** tests/test\_services/test\_project\_service.py

*import pytest*

*from datetime import datetime, timedelta*

*from unittest.mock import Mock*

*from app.services import project\_service*

*from app.models.project\_model import ProjectStatus*

*@pytest.mark.parametrize(*

*"project\_status, completion\_date, expected\_result",*

*[*

*(ProjectStatus.COMPLETED, datetime.utcnow() - timedelta(days=31), True),*

*(ProjectStatus.ACTIVE, datetime.utcnow() - timedelta(days=31), False),*

*(ProjectStatus.COMPLETED, datetime.utcnow() - timedelta(days=15), False),*

*(ProjectStatus.COMPLETED, None, False),*

*],*

*)*

*def test\_can\_archive\_project(project\_status, completion\_date, expected\_result):*

*"""Vérifie la logique de la fonction can\_archive\_project dans divers scénarios."""*

*# ARRANGE : On simule un objet Project avec les attributs nécessaires.*

*mock\_project = Mock()*

*mock\_project.status = project\_status*

*mock\_project.completed\_at = completion\_date*

*# ACT : On exécute la fonction à tester.*

*can\_be\_archived = project\_service.can\_archive\_project(mock\_project)*

*# ASSERT : On vérifie que le résultat est celui attendu.*

*assert can\_be\_archived is expected\_result*

##### **B. Tests d'Intégration (Milieu de la pyramide)**

* **Objectif :** Vérifier que les composants d'un microservice interagissent correctement avec son infrastructure directe (principalement la base de données).
* **Périmètre :** Ils s'exécutent contre une **base de données de test réelle mais éphémère**, garantissant l'isolation.
* **Exemple Concret de Script :**
  + **Scénario :** Tester la fonction crud.create\_project et vérifier que le projet est bien créé en base de données.
  + **Fichier :** tests/test\_crud/test\_crud\_projects.py

*from sqlalchemy.ext.asyncio import AsyncSession*

*from app.crud import crud\_projects*

*from app.schemas.project\_schema import ProjectCreateSchema*

*# Note: la 'db\_session' est une "fixture" pytest définie dans 'tests/conftest.py'*

*# qui fournit une session de BDD de test propre pour chaque test.*

*async def test\_create\_project\_in\_db(db\_session: AsyncSession):*

*"""Vérifie la création correcte d'un projet dans la base de données."""*

*# ARRANGE : On prépare les données d'entrée via un schéma Pydantic.*

*project\_data = ProjectCreateSchema(*

*title="Nouveau Projet d'Intégration",*

*description="Une description de test.",*

*owner\_id="f47ac10b-58cc-4372-a567-0e02b2c3d479" # ID de l'entreprise propriétaire*

*)*

*# ACT : On appelle la fonction CRUD qui interagit avec la BDD.*

*created\_project = await crud\_projects.create(db=db\_session, obj\_in=project\_data)*

*# ASSERT : On vérifie les résultats.*

*assert created\_project is not None*

*assert created\_project.title == "Nouveau Projet d'Intégration"*

*assert created\_project.id is not None # L'ID a été généré par la BDD.*

##### **C. Tests de Contrat API / End-to-End (Sommet de la pyramide)**

* **Objectif :** Garantir que l'API externe d'un microservice se comporte exactement comme le spécifie son contrat OpenAPI.
* **Périmètre :** Ils simulent des requêtes HTTP réelles vers l'application et valident toute la chaîne de traitement.
* **Exemple Concret de Script :**
  + **Scénario :** Tester l'endpoint POST /api/v1/projects pour la création d'un projet.
  + **Fichier :** tests/test\_api/v1/test\_api\_projects.py

*from httpx import AsyncClient*

*# Note: le 'client' est une "fixture" pytest qui fournit un client HTTP*

*# pour interroger l'application de test. L'authentification est gérée*

*# par une autre fixture qui injecte un header 'Authorization' valide.*

*async def test\_create\_project\_api\_endpoint(client: AsyncClient, valid\_company\_headers: dict):*

*"""Vérifie le endpoint de création de projet de bout en bout."""*

*# ARRANGE : On définit le payload JSON de la requête.*

*project\_payload = {*

*"title": "Nouveau Projet API",*

*"description": "Description via API.",*

*}*

*# ACT : On envoie une requête HTTP POST réelle à notre application de test.*

*response = await client.post(*

*"/api/v1/projects",*

*json=project\_payload,*

*headers=valid\_company\_headers*

*)*

*# ASSERT : On vérifie la réponse HTTP et son contenu.*

*assert response.status\_code == 201 # 201 Created*

*response\_data = response.json()*

*assert response\_data["title"] == "Nouveau Projet API"*

*assert "id" in response\_data*

#### **6.3. Exigences de Qualité et Automatisation**

Pour garantir que cette stratégie est appliquée, nous mettons en place des gardes-fous automatisés.

1. **Couverture de Code (Code Coverage) :**
   * **Règle :** Un taux de couverture de code par les tests (mesuré avec pytest-cov) **strictement supérieur à 85%** est exigé pour chaque microservice.
2. **Intégration Continue (CI) et "Quality Gate" :**
   * **Règle :** Le pipeline de CI sur GitHub Actions exécute automatiquement l'intégralité de la suite de tests pour chaque Pull Request.
   * **Règle absolue :** Une Pull Request ne pourra pas être fusionnée si :
     + Un ou plusieurs tests échouent.
     + Le taux de couverture de code du service descend en dessous du seuil de 85%.

### **artie 7 : Sécurité Détaillée**

#### **7.1. Authentification : Vérification de l'Identité**

L'authentification est le processus qui confirme que l'utilisateur est bien celui qu'il prétend être.

1. **Standard de Jeton : JWT avec RS256**
   * **Règle :** Le standard d'authentification pour les APIs est le **JSON Web Token (JWT)**. L'algorithme de signature à utiliser est **RS256** (signature asymétrique avec une paire de clés RSA).
   * **Justification :** L'utilisation de RS256 est un choix de sécurité délibéré pour un environnement microservices. Seul le user-service possède la clé privée pour **signer** les jetons. Tous les autres services et l'API Gateway détiennent uniquement la clé publique pour **vérifier** la signature. Cela empêche un service compromis de pouvoir générer de nouveaux jetons valides.
2. **Processus de Validation de Jeton**
   * **Règle :** Toute requête arrivant sur un endpoint sécurisé doit faire l'objet d'une validation de son jeton JWT. Ce processus, encapsulé dans une dépendance FastAPI partagée, doit effectuer les vérifications suivantes dans cet ordre :
     1. Extraire le jeton du header Authorization: Bearer <token>.
     2. Vérifier la signature cryptographique du jeton avec la clé publique.
     3. Valider les "claims" standards :
        + exp (Expiration Time) : Le jeton n'est pas expiré.
        + iss (Issuer) : L'émetteur est bien skillforge.ai/auth.
        + aud (Audience) : L'audience est bien skillforge.ai/api.
   * **Action :** En cas d'échec de l'une de ces étapes, la requête doit être immédiatement rejetée avec une erreur HTTP 401 Unauthorized.

#### **7.2. Autorisation : Contrôle des Permissions (RBAC)**

L'autorisation est le processus qui détermine ce qu'un utilisateur authentifié a le droit de faire.

1. **Source de Vérité des Rôles :** Le rôle de l'utilisateur (apprenant, entreprise, admin) est contenu dans le claim role du payload JWT. Ce claim est considéré comme fiable car le jeton a été signé par le user-service.
2. **Implémentation avec des Dépendances FastAPI :**
   * **Règle :** Le contrôle d'accès basé sur les rôles (RBAC) doit être implémenté à l'aide d'une dépendance FastAPI réutilisable. Cette approche garantit une application uniforme et déclarative des règles de sécurité directement dans la signature des endpoints.
   * **Exemple de Mise en Œuvre :**
     + **Création d'une dépendance RoleChecker dans un module de sécurité partagé :**

—------

# app/core/security.py

from fastapi import Depends, HTTPException, status

from app.models.user\_model import User # Notre modèle utilisateur

# 'get\_current\_active\_user' est la dépendance d'authentification

# qui valide le JWT et retourne l'objet User.

from .auth import get\_current\_active\_user

class RoleChecker:

def \_\_init\_\_(self, allowed\_roles: list[str]):

self.allowed\_roles = allowed\_roles

def \_\_call\_\_(self, user: User = Depends(get\_current\_active\_user)):

if user.role not in self.allowed\_roles:

raise HTTPException(

status\_code=status.HTTP\_403\_FORBIDDEN,

detail="The user does not have sufficient privileges for this operation.",

)

return user

—--------

Utilisation sur un endpoint sécurisé :

—-----

# app/api/v1/endpoints/admin.py

from app.core.security import RoleChecker

# Seuls les utilisateurs avec le rôle "admin" peuvent appeler cet endpoint.

@router.post("/projects/{project\_id}/validate")

async def validate\_project(

project\_id: UUID,

admin\_user: User = Depends(RoleChecker(allowed\_roles=["admin"]))

):

# ... la logique métier pour la validation du projet

return {"status": "project validated"}

—------

#### **7.3. Validation des Entrées : Prévention des Injections**

La validation systématique de toutes les données entrantes est notre première et meilleure ligne de défense contre les attaques par injection et la corruption de données.

1. **Rôle Central de Pydantic :**
   * **Règle absolue :** Aucune donnée provenant d'une source externe (corps de requête HTTP, paramètre d'URL, message de broker) ne doit être traitée par la logique métier sans avoir été préalablement validée par un schéma Pydantic.
   * **Principe "Fail-Fast" :** FastAPI applique ce principe par défaut. Si le corps d'une requête ne correspond pas au schéma Pydantic défini pour l'endpoint, FastAPI rejette automatiquement la requête avec une erreur HTTP 422 Unprocessable Entity avant même que notre code ne soit exécuté.
2. **Protection contre les Injections SQL :**
   * **Règle :** L'utilisation systématique de l'ORM SQLAlchemy avec des requêtes paramétrées (ce qui est son mode de fonctionnement par défaut) nous protège efficacement contre les injections SQL. L'écriture de requêtes SQL brutes (raw SQL) est formellement interdite sans une revue de sécurité approfondie.

#### **7.4. Sécurité des Dépendances : Audit Continu**

Une vulnérabilité dans une de nos bibliothèques tierces est une vulnérabilité dans notre propre application.

1. **Problématique :** La chaîne d'approvisionnement logicielle (supply chain) est un vecteur d'attaque majeur.
2. **Solution d'Audit Automatisé :**
   * **Outil :** Nous utiliserons pip-audit pour scanner nos dépendances contre les bases de données de vulnérabilités connues (comme PyPI Advisory Database).
   * **Automatisation Obligatoire :** L'exécution de pip-audit doit être une étape obligatoire dans notre pipeline de CI/CD sur GitHub Actions pour chaque Pull Request.
   * **Règle "Fail-Fast" :** Le pipeline sera configuré pour échouer (--fail-on-vuln) si une vulnérabilité de niveau "élevé" ou "critique" est détectée. Le "merge" de la Pull Request sera bloqué jusqu'à ce que la dépendance vulnérable soit mise à jour ou remplacée.