

Documentation du projet OrganisX

Youssef Tghrayt

17 août 2025

Préface

Ce document a pour objectif de présenter la conception et le développement de l'application **OrganisX**, une solution moderne de gestion de tâches basée sur **.NET 9** et **Angular**.

Il décrit l'architecture hexagonale choisie, l'organisation du backend et du frontend, ainsi que les communications entre microservices via RabbitMQ.

Cette documentation est destinée à la fois aux développeurs qui souhaitent comprendre la structure du projet et aux parties prenantes qui souhaitent une vue globale de l'application.

Auteur : Youssef Tghrayt

Table des matières

1	Introduction	4
1.1	Objectifs du projet	4
1.2	Approche et méthodologie	4
1.3	Vision à long terme	5
2	Documentation Fonctionnelle	6
2.1	Roadmap fonctionnelle	6
2.1.1	MVP 1 : Gestion personnelle	6
2.1.2	MVP 2 : Collaboration	6
2.1.3	MVP 3 : Organisation avancée	6
3	Cahier des charges - MVP 1	7
3.1	Introduction	7
3.2	Objectifs du MVP 1	7
3.3	Périmètre fonctionnel	7
3.3.1	Périmètre, acteurs et scénarios	7
3.3.2	Cas d'utilisation principaux	8
3.3.3	Diagramme d'utilisation (optionnel)	8
3.3.4	Paramètres et données	8
3.4	Exigences techniques	8
3.4.1	Backend	8
3.4.2	Frontend	9
3.5	Exigences non-fonctionnelles	9
3.6	Architecture MVP 1	9
3.6.1	Backend	9
3.6.2	Frontend	10
3.6.3	Schéma d'architecture	10
3.7	Planification / Roadmap	10
3.7.1	Plan de tests	10
3.7.2	Jalons MVP1	11
3.8	Conclusion	11
4	Architecture	12
4.1	Philosophie générale	13
4.2	Structure Backend	14
4.3	Structure Frontend	15
4.4	Communication Frontend/Backend	16
4.5	Architecture microservices dès le MVP	16
4.6	Communication entre microservices	17

4.6.1	Communication synchrone	17
4.6.2	Communication asynchrone (prévue pour les évolutions futures) . .	17
4.6.3	Gestion des contrats et documentation	17
4.7	Monitoring et observabilité	18
4.7.1	Collecte des logs	18
4.7.2	Suivi de la performance et alertes	18
4.7.3	Traçabilité des requêtes	18
4.7.4	Évolutions futures	18

Chapitre 1

Introduction

Dans un contexte où la collaboration et la gestion efficace des tâches prennent une importance croissante, le projet **OrganisX** a pour objectif de fournir une solution moderne, modulaire et extensible permettant d'organiser le travail individuel et collectif.

L'idée initiale est née du constat que les outils existants, bien que puissants, sont souvent soit trop complexes pour un usage personnel, soit trop limités pour un usage collaboratif en équipe. **OrganisX** se positionne comme un compromis entre simplicité et extensibilité, en mettant l'accent sur une architecture claire et évolutive.

1.1 Objectifs du projet

Les objectifs principaux du projet peuvent être résumés comme suit :

- Proposer une application web simple et intuitive pour la gestion des tâches personnelles.
- Introduire progressivement des fonctionnalités collaboratives (partage, équipes, gestion de projets).
- Concevoir une architecture **hexagonale** afin de garantir une séparation claire entre le domaine métier, les services applicatifs et les couches d'infrastructure.
- Documenter systématiquement les choix techniques au travers des **ADR** (Architecture Decision Records).
- Mettre en place une feuille de route incrémentale basée sur des **MVP** (**Minimum Viable Products**) afin de livrer de la valeur rapidement et de manière continue.

1.2 Approche et méthodologie

La conception de **OrganisX** suit une approche incrémentale et agile :

- Découpage du projet en trois MVP successifs :
 1. **MVP1** : gestion des tâches personnelles et authentification.
 2. **MVP2** : introduction de la collaboration (partage, équipes).
 3. **MVP3** : ajout de fonctionnalités avancées d'organisation (projets, notifications, reporting).
- Utilisation de **GitHub** pour le suivi des tâches (issues, milestones, projects).
- Documentation vivante et centralisée dans le projet **OrganisXDocs**, structurée en sections thématiques (architecture, backend, frontend, etc.).

1.3 Vision à long terme

Au-delà des trois premiers MVP, **OrganisX** vise à devenir une plateforme extensible qui pourra accueillir des modules additionnels tels que la gestion des ressources, l'intégration avec des outils tiers (calendriers, messageries), ou encore l'analyse de la productivité grâce à des métriques et des tableaux de bord.

Ainsi, ce projet se veut à la fois un terrain d'expérimentation technique (architecture hexagonale, documentation structurée, microservices) et une solution concrète pouvant évoluer en fonction des besoins utilisateurs.

Chapitre 2

Documentation Fonctionnelle

2.1 Roadmap fonctionnelle

Le développement d'OrganisX est prévu en trois MVPs successifs, permettant de livrer rapidement de la valeur tout en élargissant progressivement le périmètre fonctionnel.

2.1.1 MVP 1 : Gestion personnelle

- Authentification sécurisée (JWT / OAuth2).
- Création, modification et suppression de tâches personnelles.
- Organisation par catégories simples.

2.1.2 MVP 2 : Collaboration

- Partage de tâches et projets entre utilisateurs.
- Gestion des équipes et attribution de rôles.
- Historique des modifications et activité en temps réel.

2.1.3 MVP 3 : Organisation avancée

- Création et gestion de projets complexes.
- Système complet de notifications (RabbitMQ) et alertes personnalisées.
- Reporting et statistiques d'activité.

Chapitre 3

Cahier des charges - MVP 1

3.1 Introduction

Le premier MVP (Minimum Viable Product) du projet **OrganisX** vise à développer une application web simple de gestion de tâches personnelles (type Todo list). Ce MVP permettra de valider les choix techniques, d'expérimenter l'architecture hexagonale mise en place et de tester l'expérience utilisateur de base avant d'ajouter des fonctionnalités avancées (notifications, paiements, gestion d'équipes, etc.).

3.2 Objectifs du MVP 1

- Fournir un système d'authentification basique (inscription et connexion).
- Permettre à un utilisateur connecté de créer, consulter, modifier et supprimer ses propres tâches.
- Assurer une séparation claire entre le frontend (Angular) et le backend (ASP.NET Core en architecture hexagonale).
- Préparer le terrain pour l'évolutivité (microservices futurs, ajout d'équipes, etc.).

3.3 Périmètre fonctionnel

Le MVP 1 inclut uniquement les fonctionnalités essentielles listées ci-dessous.

3.3.1 Périmètre, acteurs et scénarios

Acteurs principaux

Acteur	Description
Utilisateur non authentifié	Peut s'inscrire ou se connecter
Utilisateur authentifié	Gère ses tâches personnelles
Système d'authentification	Émet et valide les jetons, applique les politiques de sécurité

3.3.2 Cas d'utilisation principaux

- **UC1 - S'inscrire** : L'utilisateur crée un compte en fournissant un email et un mot de passe.
- **UC2 - Se connecter** : L'utilisateur accède à l'application via email et mot de passe.
- **UC3 - Créer une tâche** : L'utilisateur ajoute une nouvelle tâche avec un titre, une description (optionnelle) et une date d'échéance.
- **UC4 - Lister mes tâches** : L'utilisateur visualise toutes ses tâches, triées par date de création ou échéance.
- **UC5 - Modifier une tâche** : L'utilisateur met à jour le titre, la description, l'échéance ou l'état d'une tâche (À faire, En cours, Terminé).
- **UC6 - Supprimer une tâche** : L'utilisateur supprime une tâche définitivement.

3.3.3 Diagramme d'utilisation (optionnel)

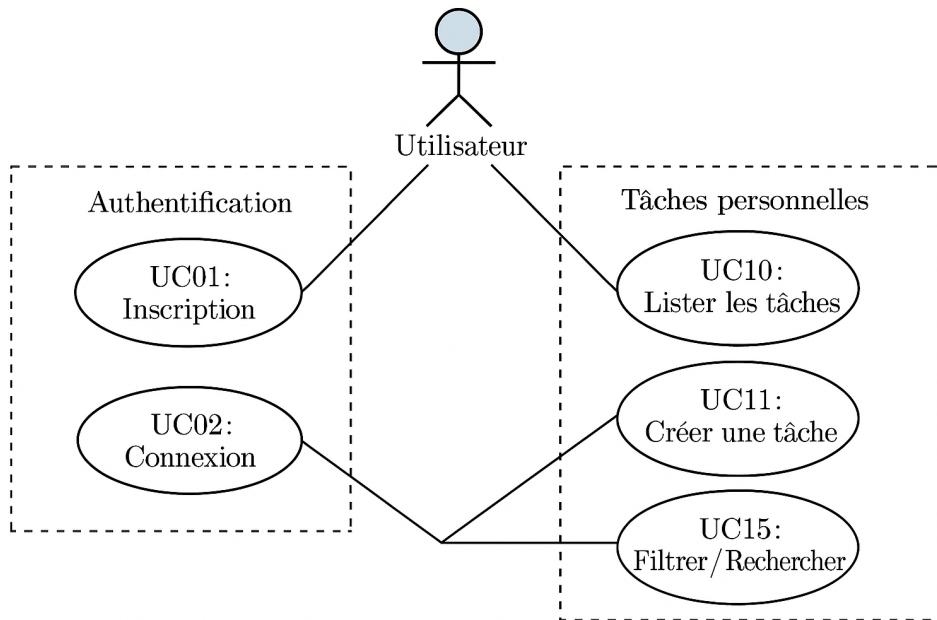


FIGURE 3.1 – Diagramme des cas d'utilisation du MVP 1

3.3.4 Paramètres et données

- **Personnalisation de l'affichage** : Choix du thème, langue et préférences d'interface.
- **Export des données** : Export des tâches au format CSV ou JSON.

3.4 Exigences techniques

3.4.1 Backend

- Langage : C# avec .NET 9.

- Architecture : Hexagonale (Domain, Application, Infrastructure, API).
- Base de données : PostgreSQL.
- Sécurité : Authentification JWT.
- Tests unitaires : xUnit.

Modèle de données simplifié

Entité	Attributs	Contraintes
Utilisateur	id, email, passwordHash, displayName, locale, timeZone, createdAt	Email unique, mot de passe hashé
Tâche	id, userId, title, description, status, priority, dueDate, tags[], createdAt, updatedAt	FK userId, taille max description 2000 caractères

API principales

POST /api/v1/auth/register Créer un compte.

POST /api/v1/auth/login Obtenir tokens d'accès et de rafraîchissement.

GET /api/v1/tasks Lister les tâches avec pagination et filtres.

POST /api/v1/tasks Créer une tâche.

PUT /api/v1/tasks/{id} Modifier une tâche.

DELETE /api/v1/tasks/{id} Supprimer une tâche.

3.4.2 Frontend

- Framework : Angular 20.
- Gestion d'état : services et RxJS.
- Communication : Appels HTTP vers l'API REST.
- UI : TailwindCSS ou Angular Material.

3.5 Exigences non-fonctionnelles

- **Performance** : Les appels API doivent répondre en moins de 500ms en moyenne.
- **Sécurité** : Les mots de passe doivent être hashés (bcrypt).
- **Disponibilité** : 99% pour MVP (mode développement).
- **Expérience utilisateur** : Interface simple et responsive.
- **Évolutivité** : Préparer la modularisation future (ajout de microservices).

3.6 Architecture MVP 1

3.6.1 Backend

- **AuthService** : Gestion des comptes utilisateurs et authentification.
- **TodoService** : Gestion des tâches (CRUD).

3.6.2 Frontend

- Module Auth (login, register).
- Module Todos (liste, création, édition, suppression).
- Shared components (header, forms, etc.).

3.6.3 Schéma d'architecture

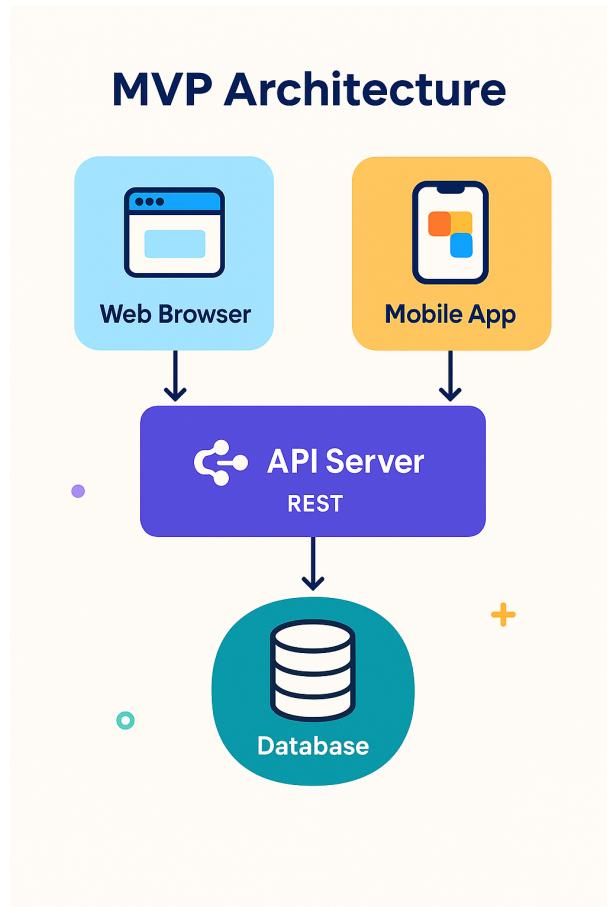


FIGURE 3.2 – Architecture technique du MVP 1

3.7 Planification / Roadmap

1. **Semaine 1** : Mise en place du projet backend (API .NET, PostgreSQL, AuthService).
2. **Semaine 2** : Implémentation du TodoService (CRUD).
3. **Semaine 3** : Mise en place du frontend Angular (Auth, Todos).
4. **Semaine 4** : Intégration frontend-backend + tests utilisateurs.

3.7.1 Plan de tests

- Tests unitaires backend (Domain, Application).
- Tests intégration API Auth et Todo.
- E2E frontend : parcours inscription → création → complétion.

3.7.2 Jalons MVP1

1. **Alpha** (S2-3) : Auth + création/liste tâches.
2. **Beta** (S4-5) : Filtres/recherche, validations, tests intégration.
3. **Release** (S6) : Durcissement sécurité, perfs, doc finalisée.

3.8 Conclusion

Le MVP 1 permettra de valider la faisabilité technique du projet OrganisX et de fournir une base solide pour les fonctionnalités futures. L'approche incrémentale et modulaire garantit que chaque évolution du projet se construit sur des fondations fiables.

Chapitre 4

Architecture

L'architecture du projet **OrganisX** constitue la fondation sur laquelle reposent toutes les fonctionnalités, la qualité et l'évolutivité du système. Elle a été conçue pour répondre aux besoins suivants :

- Séparer clairement le **coeur métier** des préoccupations techniques.
- Assurer une **testabilité élevée** et faciliter la maintenance.
- Permettre une **évolution progressive** vers des solutions distribuées et scalables (microservices, cloud).
- Favoriser l'**adaptabilité technologique** (pouvoir remplacer une base de données, un framework, une librairie sans réécrire la logique métier).

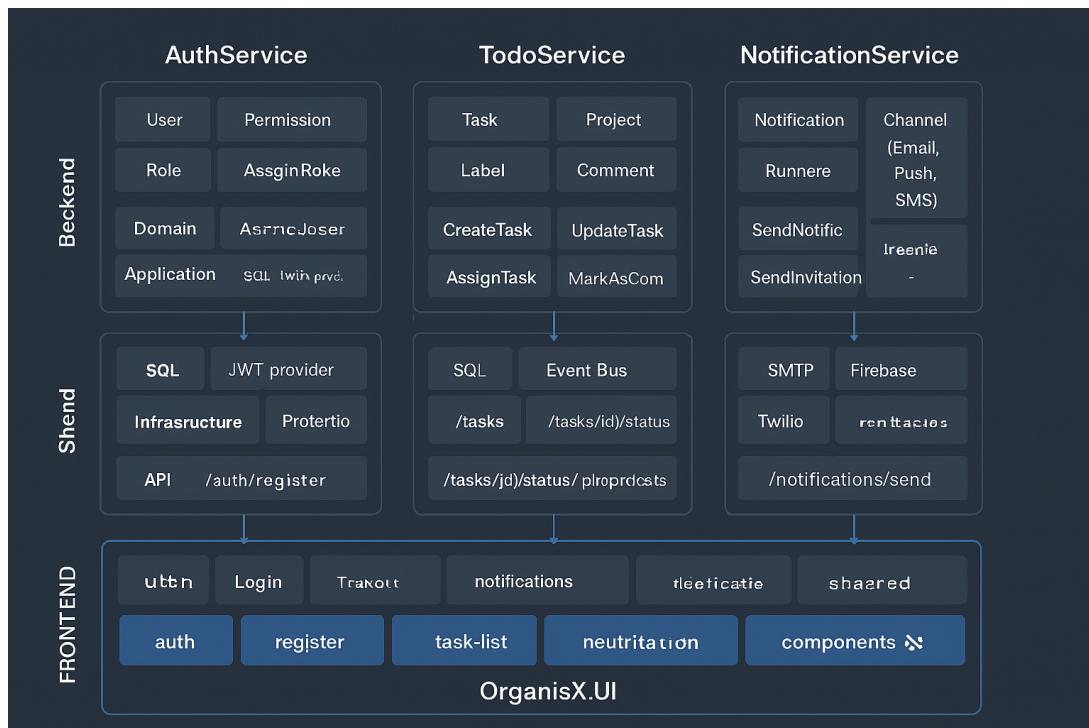


FIGURE 4.1 – Architecture globale du projet OrganisX

4.1 Philosophie générale

Le projet adopte une approche **hexagonale** (également appelée *Ports & Adapters*). Cette architecture permet d'organiser le code autour de son **domaine métier**, tout en rendant les dépendances techniques secondaires et interchangeables. En pratique, cela signifie que le cœur applicatif (les règles métier et les cas d'utilisation) ne dépend d'aucune technologie particulière, mais uniquement d'interfaces (ports). Les implémentations concrètes (adapters) se trouvent en périphérie.

4.2 Structure Backend

Le backend est développé en **.NET 9** et adopte une organisation orientée **microservices**, tout en respectant les principes de l'architecture hexagonale. Chaque service est indépendant, possède sa propre base de données et expose une API REST.

Les microservices principaux sont :

- **OrganisX.AuthService** :
 - Gestion de l'authentification et de l'autorisation via **JWT**.
 - Contient les entités liées aux utilisateurs (User, Role).
 - Peut être remplacé à terme par une solution externe (Azure AD, Keycloak).
- **OrganisX.TaskService** :
 - Gestion du cycle de vie des tâches (création, assignation, suivi).
 - Implémente les règles métier associées (une tâche doit avoir un responsable, des deadlines, etc.).
- **OrganisX.TeamService** :
 - Gestion des équipes et des relations entre membres.
 - Coordination avec TaskService pour affecter les tâches aux bons utilisateurs.
- **OrganisX.ReportingService** (optionnel/futur) :
 - Génération de rapports (productivité, suivi des tâches).
 - Peut consommer des événements provenant des autres microservices.

Chaque microservice respecte la structure interne suivante (dérivée de l'architecture hexagonale) :

- **Domain** : entités métier et invariants.
- **Application** : cas d'utilisation.
- **Infrastructure** : persistance, messaging (RabbitMQ/Azure Service Bus), accès API externes.
- **Api** : contrôleurs REST exposant les fonctionnalités.

La communication entre microservices peut se faire :

- en **synchrone**s (API REST) pour les appels directs,
- en **asynchrone**s (bus de messages) pour les événements métier (ex. création d'une tâche notifiant le service d'équipe).

Domain, Application, Infrastructure et Api.

```

/backend
  └── AuthService
    ├── AuthService.Domain
    ├── AuthService.Application
    ├── AuthService.Infrastructure
    └── AuthService.Api

  └── TodoService
    ├── TodoService.Domain
    ├── TodoService.Application
    ├── TodoService.Infrastructure
    └── TodoService.Api

  └── NotificationService
    ├── NotificationService.Domain
    ├── NotificationService.Application
    ├── NotificationService.Infrastructure
    └── NotificationService.Api

  └── Shared
    └── Shared

```

Cette organisation respecte le principe de **dépendances dirigées vers l'intérieur** : seul le domaine est indépendant, tandis que l'infrastructure dépend de l'application et du domaine.

4.3 Structure Frontend

Le frontend est conçu en **Angular 20**. Il constitue la porte d'entrée utilisateur vers les fonctionnalités d'OrganisX. Son organisation repose sur les principes suivants :

- **Modules fonctionnels** : chaque domaine métier (authentification, gestion des tâches, gestion des équipes) est isolé dans un module Angular.
- **Services Angular** : centralisation de la logique de communication avec le backend via l'API REST.
- **Composants UI** : interfaces réactives permettant une bonne expérience utilisateur.
- **State management** (si besoin futur) : possibilité d'introduire NgRx pour gérer des flux complexes d'état.

```

/freight
└ src/app
    └── auth
        ├── login
        └── register

    └── todos
        ├── task-list
        ├── task-detail
        └── project-view

    └── notifications
        └── notification-list

    └── shared
        ├── components
        └── services

    └── core
        ├── interceptors
        ├── guards
        └── api-clients

```

4.4 Communication Frontend/Backend

L'interaction entre le frontend et le backend se fait via une **API RESTful**, en utilisant le format **JSON**. Quelques caractéristiques principales :

- **Authentification** : sécurisée par **JWT**.
- **Endpoints REST** exposant les cas d'utilisation (ex. `POST /tasks`, `GET /teams`).
- **Validation et erreurs** : centralisées dans le backend et renvoyées au frontend pour affichage clair.

4.5 Architecture microservices dès le MVP

Le projet adopte directement une **architecture microservices**, afin d'assurer une meilleure séparation des responsabilités, une évolutivité indépendante des modules et une meilleure résilience.

Les services principaux prévus dès le MVP sont :

- **AuthService** : gestion des utilisateurs, inscription, connexion et génération de tokens JWT.

- **TaskService** : gestion des tâches (création, mise à jour, suppression, consultation).

La communication entre les services s'effectue via des **API REST sécurisées**. À terme, l'architecture pourra intégrer une communication **asynchrone par messages** (RabbitMQ, Azure Service Bus, Kafka) pour gérer des flux plus complexes.

Cette approche permet de garantir dès le départ :

- Une **scalabilité indépendante** (chaque microservice peut être déployé et scalé séparément).
- Une **flexibilité technologique** (chaque service peut évoluer indépendamment avec ses propres choix techniques).
- Une **résilience accrue** (panne d'un service n'entraîne pas la chute du système global).

4.6 Communication entre microservices

La communication entre microservices est un élément clé de l'architecture. Dans le cadre du **MVP 1**, le choix s'oriente vers une **communication synchrone via API REST sécurisées**, afin de privilégier la simplicité de mise en œuvre. Chaque service expose ses fonctionnalités sous forme d'API documentées (ex. Swagger / OpenAPI).

4.6.1 Communication synchrone

- Les microservices communiquent via **HTTP/HTTPS** avec des formats standards comme **JSON**.
- Un **API Gateway** (optionnel au MVP) peut être introduit pour centraliser l'accès aux différents services, gérer l'authentification et appliquer des règles de routage.
- Exemple : le **TaskService** vérifie auprès de l'**AuthService** si un utilisateur est authentifié avant de permettre la création d'une tâche.

4.6.2 Communication asynchrone (prévue pour les évolutions futures)

Afin d'assurer la scalabilité et la résilience, une communication asynchrone pourra être mise en place dans les versions ultérieures :

- Utilisation de **message brokers** tels que **RabbitMQ**, **Kafka** ou **Azure Service Bus**.
- Gestion d'événements métiers (*UserRegistered*, *TaskCreated*, *TaskCompleted*, etc.).
- Permet de découpler davantage les microservices et d'améliorer la **tolérance aux pannes**.

4.6.3 Gestion des contrats et documentation

- Chaque microservice expose un **contrat d'API** clair (OpenAPI/Swagger).
- La documentation est versionnée avec le code source pour garantir la cohérence.
- Les tests d'intégration incluent des vérifications de compatibilité entre services.

Ainsi, le système combine une **approche simple et pragmatique au démarrage** avec la possibilité d'évoluer vers des **communications asynchrones robustes** lorsque les besoins l'exigeront.

4.7 Monitoring et observabilité

Afin de garantir la fiabilité et la maintenabilité du système, une stratégie de **monitoring centralisé** est mise en place dès le MVP. L'outil choisi est **Telemetry BetterStack**, qui permet de collecter, centraliser et analyser les logs provenant des différents microservices.

4.7.1 Collecte des logs

- Chaque microservice est configuré pour envoyer ses **logs applicatifs** (ex. erreurs, requêtes entrantes, événements métiers) vers la plateforme BetterStack.
- Les logs sont structurés au format **JSON** pour faciliter leur exploitation.
- Les logs sensibles (mots de passe, tokens, données personnelles) sont systématiquement masqués avant l'envoi.

4.7.2 Suivi de la performance et alertes

- BetterStack permet de **surveiller les métriques clés** : temps de réponse des API, nombre d'erreurs, taux de succès.
- Mise en place de **seuils d'alerte** (ex. temps de réponse > 2s, taux d'erreurs > 5%) déclenchant des notifications (email, Slack, etc.).
- Les tableaux de bord offrent une vue consolidée de l'état des microservices.

4.7.3 Traçabilité des requêtes

- Un **correlation ID** unique est généré pour chaque requête utilisateur et propagé entre les microservices.
- Cela permet de tracer le parcours complet d'une requête (de l'AuthService au TaskService, etc.) et de faciliter le **debugging**.

4.7.4 Évolutions futures

- Intégration possible avec des solutions de **metrics** et **APM** (Application Performance Monitoring) comme Prometheus ou Grafana.
- Mise en place de **tracing distribué** via OpenTelemetry pour analyser en profondeur les appels entre microservices.

Ainsi, dès le MVP, le projet bénéficie d'une solution robuste pour la **collecte de logs et le suivi en temps réel**, garantissant une détection rapide des anomalies et une maintenance proactive.