Documentation du projet OrganisX

Youssef Tghrayt

16 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

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

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 **cœur 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).

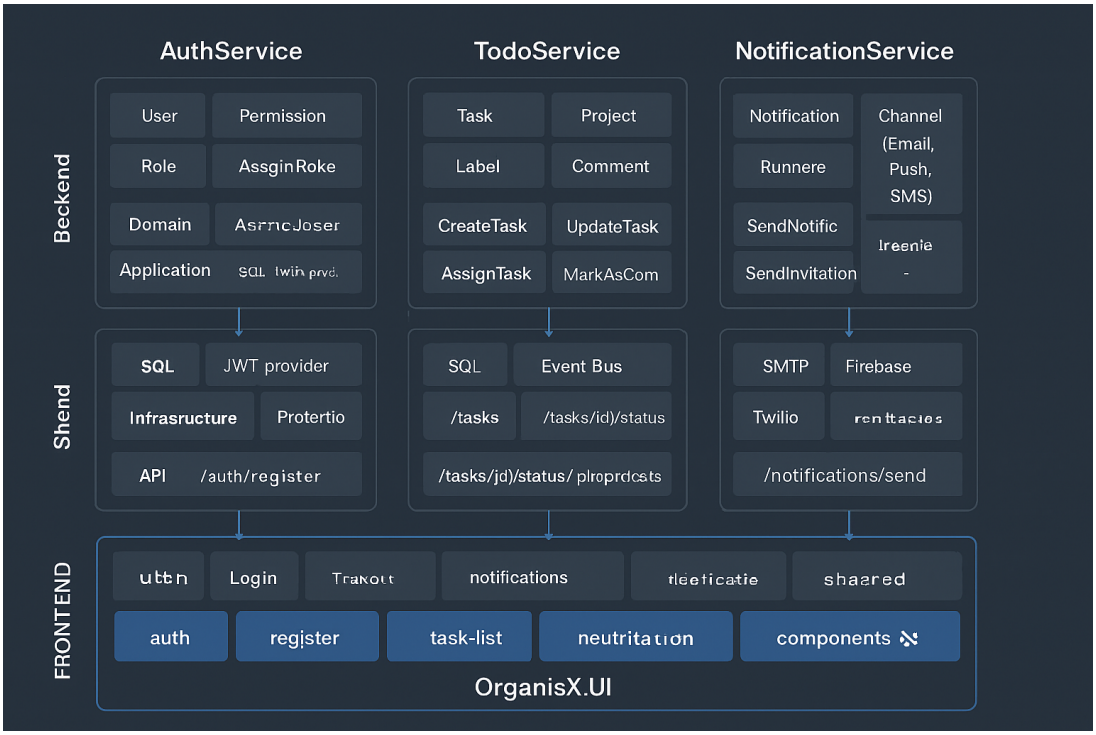


FIGURE 2.1 – Architecture globale du projet OrganisX

2.1 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.

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

`Domain`, `Application`, `Infrastructure` et `Api`.

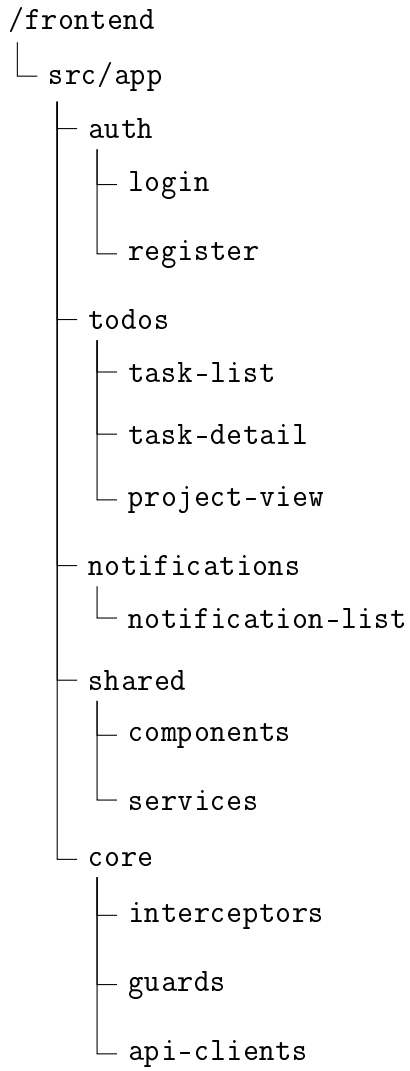
```
/backend
├─ AuthService
│  ├─ AuthService.Domain
│  ├─ AuthService.Application
│  ├─ AuthService.Infrastructure
│  └─ AuthService.Api
├─ TodoService
│  ├─ TodoService.Domain
│  ├─ TodoService.Application
│  ├─ TodoService.Infrastructure
│  └─ TodoService.Api
├─ NotificationService
│  ├─ NotificationService.Domain
│  ├─ 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.

2.3 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.



2.4 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.

2.5 Évolution future : vers les microservices

Le projet démarre sous la forme d'un **monolithe modulaire**. Toutefois, grâce à l'architecture hexagonale, il est possible de faire évoluer certaines parties vers des **microservices indépendants** si la charge augmente. Par exemple :

- Le module d'authentification peut être externalisé en un service dédié.
- La gestion des tâches peut être scalée indépendamment de la gestion des utilisateurs.

— La communication inter-services pourra s'appuyer sur des **messages asynchrones** (ex. RabbitMQ, Azure Service Bus).

Ainsi, l'architecture choisie garantit à la fois la **simplicité initiale** et la **scalabilité future**.