**1. Contexte et Objectifs du Projet**

Le projet vise à automatiser la gestion des dossiers administratifs liés à la naissance, au mariage et à d'autres événements d’état civil à travers une plateforme intelligente. Cette plateforme comportera :

* Un système de traitement automatisé de dossiers.
* Une reconnaissance OCR des documents déposés.
* Une gestion de calendrier pour les rendez-vous et notifications.

L’objectif principal est de **numériser, automatiser et simplifier** les démarches administratives pour les citoyens et les agents administratifs.

**2. Besoins Fonctionnels**

**2.1 Module 1 : Automatisation du traitement des dossiers**

* L’utilisateur choisit le type de dossier à déposer (Naissance, Mariage…).
* Le système suit une modélisation BPMN (Camunda) pour chaque type de processus.
* Le traitement suit un **workflow dynamique** selon le type de dossier.
* Suivi d’avancement de chaque étape pour les agents.

**2.2 Module 2 : Intégration de calendrier et notifications**

* Visualisation des **rendez-vous disponibles** dans un calendrier (Angular Scheduler).
* Prise de rendez-vous par les usagers.
* Envoi automatique de **notifications (e-mail/SMS)** via RabbitMQ.
* Gestion des reports ou annulations.

**2.3 Module 3 : Reconnaissance OCR des documents**

* Téléversement de documents justificatifs (PDF, image).
* Reconnaissance automatique des informations via Tesseract dans un conteneur Docker.
* Extraction des données pour auto-remplir des champs de formulaire.

**3. Besoins Non Fonctionnels**

* Sécurité des données personnelles (RGPD).
* Interface intuitive et responsive.
* Système scalable et modulable.
* Disponibilité 24/7.

**4. Outils et Technologies Utilisées**

| **Composant** | **Technologie choisie** |
| --- | --- |
| Workflow BPMN | Camunda BPMN |
| Back-end | .NET (C#) |
| Front-end | Angular (avec Angular Scheduler) |
| Message broker | RabbitMQ |
| OCR | Tesseract (via Docker) |
| Base de données | PostgreSQL / SQL Server (selon module) |
| Conteneurisation | Docker |

**5. Base de Données – Schéma simplifié**

**📁 DossierAdministratif**

* id
* typeDossier (Naissance, Mariage…)
* dateDepot
* statut
* documentAssocie

**📅 Agenda**

* id
* dateRendezVous
* heure
* utilisateurId
* etat (Confirmé, Annulé…)

**🗂️ PapiersRequis**

* id
* typeDossier
* nomDocument
* cheminFichier
* texteExtrait

**6. Plan du Projet**

| **Étape** | **Durée estimée** | **Responsable** |
| --- | --- | --- |
| Rédaction des spécifications | 1 semaine | Analyste |
| Mise en place de l’architecture | 1 semaine | Architecte |
| Développement module OCR | 2 semaines | Dev OCR + DevOps |
| Développement module workflow | 3 semaines | Dev .NET + BPMN |
| Développement calendrier + notif | 2 semaines | Dev Angular |
| Intégration & tests | 2 semaines | Équipe QA |

Étape 0 : Préparation & planification

├─ Analyse des besoins backend détaillés

├─ Choix des technologies & outils (ASP.NET Core, Camunda, RabbitMQ, PostgreSQL)

├─ Définition de la structure des microservices

└─ Création des repositories Git (un par microservice)

Étape 1 : Mise en place de l’infrastructure backend

├─ Initialiser projets ASP.NET Core pour chaque microservice

├─ Configurer accès base de données (PostgreSQL ou SQL Server)

├─ Configurer l’authentification centralisée (IdentityServer)

└─ Mettre en place la communication interservices (REST, RabbitMQ)

Étape 2 : Développement du CitizenService

├─ Modélisation entités (Citoyen, Auth, rôles)

├─ Création des API CRUD pour gestion citoyens

├─ Tests unitaires des endpoints

└─ Mise en place de la sécurité des API (JWT, scopes)

Étape 3 : Développement du DossierService avec workflow Camunda

├─ Modélisation entités DossierAdministratif, TypeDossier

├─ Intégration Camunda BPMN (déploiement diagrammes)

├─ Implémentation lancement de processus via API

├─ Gestion des tâches utilisateurs (validation, vérification)

└─ Tests d’intégration avec le moteur BPMN

Étape 4 : Développement du ReclamationService

├─ Modélisation entités Réclamation, HistoriqueRequête

├─ API CRUD pour dépôt, modification, suivi des réclamations

├─ Notifications de statut (via RabbitMQ si besoin)

└─ Tests fonctionnels et sécurité

Étape 5 : Développement du AgendaService avec RabbitMQ

├─ Modélisation entités AgendaRendezVous

├─ API gestion rendez-vous (création, modification, suppression)

├─ Publication des événements rendez-vous sur RabbitMQ

├─ Mise en place worker de notification asynchrone

└─ Tests de la chaîne complète (API → RabbitMQ → Worker → Notification)

Étape 6 : Tests d’intégration globale backend

├─ Tester les échanges entre microservices (ex: Reclamation <-> Dossier)

├─ Tester le flux complet d’un dossier (depuis dépôt à validation)

├─ Tester la sécurité inter-services (authentification et autorisation)

└─ Validation performances API

Étape 7 : Documentation & préparation au déploiement

├─ Documenter les APIs (Swagger/OpenAPI)

├─ Préparer Dockerfiles et scripts CI/CD

└─ Validation finale avant intégration frontend

**🔗 Communication entre les composants du projet CitizenService**

**1. Architecture générale**

[Frontend Angular / Flutter]

│

│ REST API (HTTPS / JWT)

▼

[Backend API ASP.NET Core (Microservices)]

├────────────┬───────────────┬─────────────┐

│ │ │ │

▼ ▼ ▼ ▼

[Camunda BPMN] [RabbitMQ] [Base de données] [OCR - Tesseract Docker]

**2. Détails des communications**

**2.1. Frontend ↔ Backend API**

* **Protocoles :** HTTP/HTTPS (REST API)
* **Sécurité :** JWT (JSON Web Tokens) via IdentityServer
* **Actions :**
  + Angular/Flutter envoie des requêtes (GET, POST, PUT, DELETE) pour gérer :
    - Citoyens, dossiers, réclamations
    - Rendez-vous (agenda)
    - Téléversement de documents
  + Backend répond avec JSON (données, statuts, erreurs)

**2.2. Backend API ↔ Base de données**

* **Technologies :** ORM Entity Framework Core (C#)
* **SGBD :** PostgreSQL / SQL Server / MySQL selon le microservice
* **Communication :** Connexion directe via chaîne de connexion sécurisée
* **Actions :**
  + CRUD sur entités (Citoyen, DossierAdministratif, AgendaRendezVous, PapiersRequis…)

**2.3. Backend API ↔ Camunda BPMN**

* **Protocoles :** REST API Camunda (HTTP/HTTPS)
* **Actions :**
  + Backend lance un processus BPMN (POST /process-definition/key/{key}/start)
  + Interagit avec les tâches utilisateur ou automatiques via l’API Camunda
  + Reçoit les callbacks / webhook ou interroge l’état des processus

**2.4. Backend API ↔ RabbitMQ**

* **Protocole :** AMQP (Advanced Message Queuing Protocol)
* **Actions :**
  + Backend publie des événements (ex : création/modification de rendez-vous) dans des **queues/exchanges** RabbitMQ
  + Ces messages sont asynchrones et découpés en tâches indépendantes (notifications, rappels)

**2.5. RabbitMQ ↔ Worker Notification**

* **Protocoles :** AMQP
* **Actions :**
  + Worker consomme les messages dans RabbitMQ
  + Exécute des tâches : envoi email, push notification
  + Peut publier un message retour (statut) dans RabbitMQ ou mettre à jour la base via API

**2.6. Backend API ↔ OCR Tesseract (Docker)**

* **Options de communication :**
  + Appel système Docker CLI pour lancer un conteneur OCR sur un fichier uploadé
  + Ou appel via API HTTP si OCR est exposé comme un microservice
* **Actions :**
  + API transmet le fichier au conteneur OCR
  + Récupère le texte extrait
  + Stocke le résultat dans la base de données PapiersRequis

**3. Résumé du flux**

1. Utilisateur → Frontend Angular/Flutter

2. Frontend → Backend API (via REST, JWT sécurisé)

3. Backend → DB (CRUD direct)

4. Backend → Camunda (lancer process, gérer tâches via REST)

5. Backend → RabbitMQ (publier événements rendez-vous)

6. Worker Notification ← RabbitMQ (consommer events → envoyer mails)

7. Backend → OCR Docker (upload fichiers → extraire texte)

8. Backend → Frontend (renvoyer données actualisées)

**4. Schéma simplifié**

[Frontend Angular/Flutter]

|

| REST + JWT

v

[Backend API ASP.NET Core]

| | |

| | |

v v v

[Camunda] [RabbitMQ] [Base de données]

|

v

[Worker Notification]

[Backend API] ↔ [OCR Tesseract Docker]

**5. Conseils**

* Utilise des **API REST claires et documentées** (Swagger/OpenAPI).
* Pour la communication RabbitMQ, définis des **formats de message JSON standardisés**.
* Assure-toi que les communications sont **authentifiées et sécurisées**.
* Garde la logique métier dans les microservices, la communication entre eux doit être **asynchrone si possible** (ex: RabbitMQ).

Parfait ! Vu que tu travailles **exclusivement sur le module** 🏛️ 3. Microservice Municipalité & Dossiers Citoyens - CitizenService, je vais te refaire un **plan de travail détaillé étape par étape (de 0 à la fin)**, en tenant compte de **tout le cahier des charges fonctionnel & technique**, ainsi que de l’architecture globale.

**✅ Vue d’ensemble du projet : CitizenService**

| **Élément** | **Détail** |
| --- | --- |
| 🎯 Objectif | Centraliser toutes les démarches administratives des citoyens |
| 🔧 Techs | ASP.NET Core (.NET 8), PostgreSQL, RabbitMQ, Angular, Flutter |
| 🔐 Sécurité | IdentityServer avec JWT |
| 🧱 Architecture | Microservices + DDD + TDD |
| 📦 Communication | REST API + RabbitMQ |
| 📂 Base de données | Citoyen, Dossier, TypeDossier, Historique, Réclamations |
| 📲 Front | Angular (Web), Flutter (Mobile) |

**🧭 Feuille de Route Détaillée : Étapes de A à Z**

**🔹 ÉTAPE 0 : Cadrage & Préparation**

**🎯 Objectif :**

Mettre en place les fondations du projet.

**✅ À faire :**

1. **Créer les dossiers projet dans GitHub / GitLab**
2. **Structurer ton projet ASP.NET Core** :
   * API + Domain + Application + Infrastructure (architecture DDD)
3. **Définir les entités métier** :
   * Citoyen, DossierAdministratif, TypeDossier, HistoriqueRequete, ReclamationAdministrative
4. **Modéliser la base PostgreSQL** (MCD + script SQL init)
5. **Configurer le CI/CD** avec GitHub Actions :
   * Build, Test, Dockerize

**🔹 ÉTAPE 1 : Authentification & Sécurité**

**🎯 Objectif :**

Sécuriser les accès à l’API.

**✅ À faire :**

1. Intégrer IdentityServer (depuis le microservice AuthService)
2. Configurer :
   * Authentification par JWT
   * Rôles (CITIZEN, AGENT, ADMIN)
   * Vérification des tokens dans les controllers ASP.NET
3. Middleware pour logs sécurité + tentative de connexions

**🔹 ÉTAPE 2 : Développement du Back-end (ASP.NET Core)**

**1️⃣ Microservice : CitizenService**

**🧱 Domain Layer (DDD) :**

* Entities: Citoyen, DossierAdministratif, etc.
* ValueObjects: Adresse, StatutDossier
* Aggregates: DossierAggregate

**🔧 Application Layer :**

* UseCases :
  + Créer un dossier
  + Modifier un dossier
  + Visualiser papiers requis
  + Télécharger formulaires
  + Consulter statut
  + Créer une réclamation
  + Consulter historique
  + Planifier un rendez-vous

**💾 Infrastructure Layer :**

* Connexion PostgreSQL
* Intégration RabbitMQ (si notif)

**🌐 API Layer :**

* Controllers REST : sécurisés par JWT

**🔹 ÉTAPE 3 : Gestion des Documents (OCR)**

**🎯 Objectif :**

Extraire automatiquement du texte des documents.

**✅ À faire :**

1. Créer un microservice Dockerisé avec **Tesseract OCR**
2. API POST /ocr/analyse : upload fichier → retourne JSON avec texte
3. Enregistrer les documents OCR dans le dossier du citoyen
4. Lier avec CitizenService

**🔹 ÉTAPE 4 : Agenda des rendez-vous**

**🎯 Objectif :**

Permettre aux citoyens de prendre rendez-vous.

**✅ À faire :**

1. Créer l'entité RendezVous
2. API :
   * POST /rendezvous
   * GET /rendezvous/mes
3. Notifications par RabbitMQ vers NotificationService

**🔹 ÉTAPE 5 : Front-End (Angular & Flutter)**

**1️⃣ Angular (Admin / Agent) :**

* Gestion des dossiers
* Recherche citoyen
* Visualisation réclamations
* Planning rendez-vous
* Téléchargement de documents

**2️⃣ Flutter (Citoyen) :**

* Création dossier
* Suivi de statut
* Dépôt de réclamation
* Prise de rendez-vous
* Notifications push

**🔹 ÉTAPE 6 : Observabilité & Logs**

**✅ À faire :**

1. Ajouter **Serilog** dans ASP.NET Core pour logs formatés JSON
2. Configurer **Prometheus** pour exporter les métriques
3. Créer des dashboards dans **Grafana**

**🔹 ÉTAPE 7 : Déploiement & Intégration**

**✅ À faire :**

1. Créer un Dockerfile pour chaque microservice
2. Docker Compose (pour dev local)
3. Helm Charts (pour K8s)
4. Intégrer le pipeline CI/CD pour :
   * Build + Test + Publish
   * Déploiement automatique sur push

**🔹 ÉTAPE 8 : Documentation & Livrables**

**✅ À faire :**

1. Générer la documentation Swagger de l’API
2. Créer README technique par microservice
3. Rédiger un manuel utilisateur (PDF ou Web)

**🗺️ Roadmap Résumée**

| **Étape** | **Action** | **État** |
| --- | --- | --- |
| 0 | Préparation projet, DDD, DB | 🔄 En cours |
| 1 | Authentification / JWT / Sécurité | ⏳ |
| 2 | Développement du back-end | ⏳ |
| 3 | OCR pour documents | ⏳ |
| 4 | Agenda & Notifications | ⏳ |
| 5 | Front Angular + Flutter | ⏳ |
| 6 | Observabilité (Serilog + Grafana) | ⏳ |
| 7 | Déploiement Docker + CI/CD | ⏳ |
| 8 | Documentation + Livrables | ⏳ |

+---------------------+

| FrontOffice |

| Angular + Flutter |

+----------+----------+

|

| (OAuth2 + JWT via API Gateway)

+----------v----------+

| API Gateway |

| (YARP, NGINX, etc.) |

+----------+----------+

|

+----------+----------+-----------------------+

| Microservices Backend (Docker) |

| - user-service (ASP.NET Core + PostgreSQL)|

| - payment-service (Spring Boot + PostgreSQL)|

| - product-service (PHP + MySQL) |

+----------+----------+-----------------------+

| |

| |

REST/gRPC calls Message Broker (RabbitMQ/Kafka)

| |

+----------v----------+-----------------------+

| Database (PostgreSQL, MySQL, SQLServer) |

+-----------------------------------------------+

Pour getway api

+--------------------+ +--------------------+

| | | |

| FrontOffice | | Mobile App (Flutter)|

| (Angular Web App) | | |

+---------+----------+ +----------+---------+

| |

| HTTP Request (avec JWT) | HTTP Request (avec JWT)

| |

v v

+--------------------------------------------------+

| API Gateway |

| (ex: YARP / NGINX / Ocelot) |

| |

| - Valide le token JWT via IdentityServer |

| - Gère le routage des requêtes |

| - Applique CORS, logs, monitoring |

+------------+-------------+------------+----------+

| | |

| | |

| | |

+----------v--+ +------v-----+ +---v------------+

| | | | | |

| user-service| |payment-svc | | product-service|

| (ASP.NET) | | (Spring) | | (PHP Laravel) |

+-------------+ +------------+ +----------------+

| | |

| | |

| | |

+-----v--------+ +-----v---------+ +---v------------+

| PostgreSQL | | PostgreSQL | | MySQL |

+--------------+ +---------------+ +----------------+

La **communication entre les services** dans un système **microservices** comme celui que tu souhaites construire pour **CitizenService** est cruciale pour assurer la cohérence des données, la fluidité des interactions et la scalabilité du système. Voici comment les services de ton projet pourraient communiquer entre eux :

**1. Architecture des Services**

**Types de communication entre les services :**

1. **Communication Synchrone (REST API ou gRPC)**
   * Utilisée pour les interactions immédiates entre services. Par exemple, un service peut envoyer une requête HTTP ou gRPC à un autre service et attendre une réponse immédiate.
2. **Communication Asynchrone (Message Brokers comme RabbitMQ ou Kafka)**
   * Utilisée pour les événements ou notifications qui n'ont pas besoin de réponse immédiate. Les services publient des événements (par exemple, lorsqu’un dossier est validé) et d'autres services peuvent y réagir sans attendre de réponse immédiate.

**2. Flux de Communication entre les Services**

**A. Service des Citoyens → Service des Dossiers Administratifs**

* **Contexte** : Lorsqu'un citoyen soumet un dossier administratif, le **Service des Citoyens** doit valider ses informations personnelles (email, identité, etc.) et, une fois la validation réussie, il crée une entrée dans la base de données. Ensuite, le service envoie une demande de création de dossier au **Service des Dossiers Administratifs**.
* **Type de communication** : **REST API**
* **Exemple de flux** :
  1. Le **Service des Citoyens** expose un **point d'API REST** pour soumettre une demande de dossier administratif. Ce service s'assure que les informations du citoyen sont valides.
  2. Une fois validé, le **Service des Citoyens** appelle un **point d'API REST** du **Service des Dossiers Administratifs** pour créer un dossier avec l'ID du citoyen et le type de dossier demandé.
  3. Le **Service des Dossiers Administratifs** enregistre le dossier et répond au **Service des Citoyens** avec un **statut de confirmation**.

**B. Service des Dossiers Administratifs → Service de Suivi des Demandes**

* **Contexte** : Le **Service des Dossiers Administratifs** doit permettre au **Service de Suivi des Demandes** d'avoir accès à l'état actuel des dossiers afin que les citoyens puissent suivre l'évolution de leurs demandes en temps réel.
* **Type de communication** : **REST API**
* **Exemple de flux** :
  1. Le **Service des Dossiers Administratifs** expose une **API REST** qui permet de récupérer les détails d'un dossier.
  2. Le **Service de Suivi des Demandes** appelle cette **API** pour afficher l'état actuel du dossier (par exemple, "En cours", "Validé", "Rejeté").

**C. Service des Réclamations Administratives → Service des Dossiers Administratifs**

* **Contexte** : Lorsqu'un citoyen soumet une réclamation, cette réclamation est liée à un dossier spécifique. Le **Service des Réclamations Administratives** doit valider que la réclamation est associée à un dossier existant et mettre à jour l'état de ce dossier si nécessaire.
* **Type de communication** : **REST API**
* **Exemple de flux** :
  1. Le **Service des Réclamations Administratives** reçoit une **requête REST** avec l'ID du citoyen et de son dossier.
  2. Il appelle une **API du Service des Dossiers Administratifs** pour vérifier le **statut du dossier** lié à cette réclamation.
  3. Si la réclamation est valide, le **Service des Réclamations** enregistre cette réclamation dans sa base de données et retourne un message de succès au citoyen.

**D. Service des Citoyens → Service des Réclamations Administratives**

* **Contexte** : Un citoyen soumet une réclamation concernant un dossier administratif. Le **Service des Citoyens** fournit une interface pour soumettre cette réclamation, et le **Service des Réclamations** doit la gérer.
* **Type de communication** : **REST API**
* **Exemple de flux** :
  1. Le **Service des Citoyens** reçoit la réclamation et valide les informations.
  2. Le **Service des Citoyens** envoie les détails de la réclamation à un **point d'API du Service des Réclamations Administratives** pour l'enregistrement et le traitement.

**E. Service des Dossiers Administratifs → Service des Rendez-vous**

* **Contexte** : Si un citoyen doit prendre un rendez-vous pour finaliser une démarche (par exemple, pour valider un dossier en personne), le **Service des Dossiers Administratifs** doit communiquer avec le **Service des Rendez-vous** pour réserver un créneau.
* **Type de communication** : **REST API**
* **Exemple de flux** :
  1. Le **Service des Dossiers Administratifs** contacte l'API **Service des Rendez-vous** pour vérifier la disponibilité des créneaux horaires.
  2. Le citoyen choisit un créneau et le **Service des Rendez-vous** enregistre la réservation dans sa base de données.

**F. Communication Asynchrone (RabbitMQ / Kafka)**

* **Contexte** : Pour certaines actions, comme la notification d'un citoyen lorsque l'état de son dossier a changé ou lorsqu'un événement a lieu dans un autre service (par exemple, lorsqu'une réclamation est traitée), tu souhaites utiliser des messages asynchrones pour informer d'autres services.
* **Type de communication** : **RabbitMQ / Kafka**
* **Exemple de flux** :
  1. Lorsqu'un dossier change de statut (par exemple, **"Validé"**), le **Service des Dossiers Administratifs** publie un **événement Kafka/RabbitMQ** pour informer les autres services de la mise à jour.
  2. Le **Service des Réclamations** s'abonne à cet événement pour vérifier si une réclamation liée au dossier doit être traitée en fonction du statut de ce dernier.
  3. Le **Service des Rendez-vous** peut également s'abonner à des événements pour envoyer des notifications aux citoyens lorsqu'un rendez-vous a été confirmé ou annulé.

**G. Notifications Push (SignalR)**

* **Contexte** : Pour fournir une communication en temps réel avec les citoyens, par exemple pour informer immédiatement un citoyen du changement d'état d'un dossier ou d'un rendez-vous, tu peux utiliser **SignalR** pour envoyer des notifications en temps réel via l'interface utilisateur.
* **Type de communication** : **SignalR**
* **Exemple de flux** :
  1. Lorsqu'un événement critique se produit (par exemple, une réclamation est traitée), un service envoie une notification via **SignalR** à l'application frontend (web ou mobile).
  2. Le citoyen voit immédiatement la mise à jour dans son interface utilisateur.

**3. Diagramme de Communication entre Services**

+---------------------+ REST API +--------------------------+

| Service des | <-------------------> | Service des Dossiers |

| Citoyens | | Administratifs |

+---------------------+ +--------------------------+

| |

Event (Kafka/RabbitMQ) |

| |

+---------------------+ Event (Kafka/RabbitMQ) +---------------------------+

| Service des | <-----------------------------> | Service des Réclamations |

| Réclamations | | Administratives |

+---------------------+ +---------------------------+

|

REST API

|

+--------------------------+

| Service des Rendez-vous |

| (Gestion des Rendez-vous) |

+--------------------------+

**Conclusion**

1. **Communication synchrone** via **REST API** est utilisée pour les interactions immédiates entre services (comme la création de dossiers, soumission de réclamations, etc.).
2. **Communication asynchrone** via **RabbitMQ ou Kafka** est utilisée pour la gestion des événements (comme la mise à jour de statut de dossier, notification de réclamation, etc.) afin de garantir une communication décentralisée et réactive sans dépendance immédiate.
3. **SignalR** est utilisé pour la notification en temps réel, permettant une interaction fluide avec les utilisateurs.

Cette approche garantit que ton système est flexible, évolutif et capable de traiter efficacement les opérations tout en respectant une architecture microservices bien conçue. Si tu souhaites des ajustements ou des détails supplémentaires sur un aspect spécifique de la communication, n’hésite pas à demander !

Oui, la structure que j'ai donnée dans ma réponse précédente couvre tous les services principaux mentionnés dans ton projet **CitizenService**. Cependant, pour une meilleure compréhension et pour assurer que tous les services sont bien représentés, je vais clarifier et détailler chaque service en fonction de l'architecture **DDD**, en ajoutant des précisions sur la gestion des événements asynchrones, des interactions avec la base de données, ainsi que la structure des contrôleurs et des tests.

Voici la structure complète pour chaque service, en prenant en compte **tous les services** nécessaires dans ton projet **CitizenService** :

**1. Service des Citoyens**

Ce service est responsable de la gestion des informations personnelles des citoyens et de l'authentification.

**Architecture :**

CitizenService.API (Web API Layer)

├── Controllers

│ ├── CitizenController.cs # API pour l'inscription, la connexion, la récupération d'infos citoyen

├── Models

│ ├── Citizen.cs # Modèle représentant un citoyen

├── Application

│ ├── CommandHandlers # Commande pour créer un citoyen

│ ├── QueryHandlers # Query pour obtenir un citoyen par ID

├── Domain

│ ├── Aggregates

│ │ ├── CitizenAggregate.cs # Agrégat représentant un citoyen (peut contenir des règles de validation)

│ ├── Repositories

│ │ ├── CitizenRepository.cs # Interface pour persister les citoyens

├── Infrastructure

│ ├── CitizenDbContext.cs # Contexte de la base de données (EF Core)

├── Tests

│ ├── CitizenControllerTests.cs # Tests pour les contrôleurs

│ ├── CitizenServiceTests.cs # Tests pour la logique métier des citoyens

│ ├── CitizenRepositoryTests.cs # Tests des interactions avec la base de données

**Services Associés :**

* **CommandHandlers** : Pour les actions comme l'inscription d'un citoyen.
* **QueryHandlers** : Pour récupérer les informations d'un citoyen.

**2. Service des Dossiers Administratifs**

Gère la création, la modification et le suivi des dossiers administratifs des citoyens.

**Architecture :**

DossierService.API (Web API Layer)

├── Controllers

│ ├── DossierController.cs # API pour créer et suivre un dossier administratif

├── Models

│ ├── Dossier.cs # Modèle représentant un dossier administratif

├── Application

│ ├── CommandHandlers # Commande pour créer, valider ou mettre à jour un dossier

│ ├── QueryHandlers # Query pour obtenir un dossier par ID

├── Domain

│ ├── Aggregates

│ │ ├── DossierAggregate.cs # Agrégat pour la gestion des dossiers

│ ├── Repositories

│ │ ├── DossierRepository.cs # Interface de repository pour persister les dossiers

├── Infrastructure

│ ├── DossierDbContext.cs # Contexte de la base de données pour les dossiers

├── Tests

│ ├── DossierControllerTests.cs # Tests pour les contrôleurs des dossiers

│ ├── DossierServiceTests.cs # Tests pour la logique métier des dossiers

│ ├── DossierRepositoryTests.cs # Tests des interactions avec la base de données

**Services Associés :**

* **CommandHandlers** : Pour la création, la validation, et la mise à jour des dossiers administratifs.
* **QueryHandlers** : Pour le suivi des dossiers administratifs par ID.

**3. Service des Réclamations Administratives**

Permet aux citoyens de soumettre des réclamations concernant leurs dossiers administratifs.

**Architecture :**

ReclamationService.API (Web API Layer)

├── Controllers

│ ├── ReclamationController.cs # API pour gérer les réclamations administratives

├── Models

│ ├── Reclamation.cs # Modèle de données représentant une réclamation

├── Application

│ ├── CommandHandlers # Commande pour soumettre et gérer une réclamation

│ ├── QueryHandlers # Query pour récupérer une réclamation par ID

├── Domain

│ ├── Aggregates

│ │ ├── ReclamationAggregate.cs# Agrégat pour la gestion des réclamations

│ ├── Repositories

│ │ ├── ReclamationRepository.cs # Interface de repository pour persister les réclamations

├── Infrastructure

│ ├── ReclamationDbContext.cs # Contexte de la base de données pour les réclamations

├── Tests

│ ├── ReclamationControllerTests.cs # Tests pour les contrôleurs des réclamations

│ ├── ReclamationServiceTests.cs # Tests pour la logique métier des réclamations

│ ├── ReclamationRepositoryTests.cs # Tests des interactions avec la base de données

**Services Associés :**

* **CommandHandlers** : Pour soumettre et traiter les réclamations des citoyens.
* **QueryHandlers** : Pour récupérer et suivre l'état des réclamations.

**4. Service de Suivi des Demandes**

Permet aux citoyens de suivre l'état de leurs demandes administratives en temps réel.

**Architecture :**

SuiviDemandeService.API (Web API Layer)

├── Controllers

│ ├── SuiviController.cs # API pour récupérer l'état des demandes des citoyens

├── Models

│ ├── SuiviDemande.cs # Modèle pour le suivi des demandes

├── Application

│ ├── CommandHandlers # Commande pour mettre à jour ou suivre une demande

│ ├── QueryHandlers # Query pour obtenir le suivi de la demande

├── Domain

│ ├── Aggregates

│ │ ├── SuiviDemandeAggregate.cs # Agrégat pour la gestion des suivis de demande

│ ├── Repositories

│ │ ├── SuiviDemandeRepository.cs # Interface de repository pour persister les suivis

├── Infrastructure

│ ├── SuiviDemandeDbContext.cs # Contexte de la base de données pour les suivis de demande

├── Tests

│ ├── SuiviControllerTests.cs # Tests pour le suivi des demandes

│ ├── SuiviServiceTests.cs # Tests pour la logique métier du suivi des demandes

│ ├── SuiviRepositoryTests.cs # Tests des interactions avec la base de données

**Services Associés :**

* **CommandHandlers** : Pour la mise à jour ou la récupération de l'état des demandes.
* **QueryHandlers** : Pour le suivi en temps réel des demandes par ID.

**5. Service des Rendez-vous**

Gère la planification et la gestion des rendez-vous des citoyens pour diverses démarches administratives.

**Architecture :**

RendezvousService.API (Web API Layer)

├── Controllers

│ ├── RendezvousController.cs # API pour gérer la création, l'annulation et le suivi des rendez-vous

├── Models

│ ├── Rendezvous.cs # Modèle pour représenter un rendez-vous

├── Application

│ ├── CommandHandlers # Commande pour créer et annuler un rendez-vous

│ ├── QueryHandlers # Query pour récupérer un rendez-vous par ID

├── Domain

│ ├── Aggregates

│ │ ├── RendezvousAggregate.cs # Agrégat pour gérer la logique métier des rendez-vous

│ ├── Repositories

│ │ ├── RendezvousRepository.cs # Interface de repository pour persister les rendez-vous

├── Infrastructure

│ ├── RendezvousDbContext.cs # Contexte de la base de données pour les rendez-vous

├── Tests

│ ├── RendezvousControllerTests.cs # Tests pour les contrôleurs des rendez-vous

│ ├── RendezvousServiceTests.cs # Tests pour la logique métier des rendez-vous

│ ├── RendezvousRepositoryTests.cs # Tests des interactions avec la base de données

**Services Associés :**

* **CommandHandlers** : Pour la création, l'annulation et la gestion des rendez-vous.
* **QueryHandlers** : Pour récupérer les rendez-vous en fonction de critères spécifiques (par exemple, par citoyen).

**Conclusion :**

La structure donnée ci-dessus inclut **tous les services nécessaires** dans ton projet **CitizenService**. Chaque service suit les principes **DDD** pour assurer une logique métier cohérente et indépendante tout en étant testé via des pratiques **TDD** pour garantir la qualité du code.

Si tu veux une autre clarification ou un détail supplémentaire sur un service spécifique, n'h

+-----------------------------+

| Plateforme Municipale |

| Intelligente |

+-----------------------------+

|

+------------+------------+

| |

+-------------------+ +------------------+

| Citoyen | | Administrateur |

+-------------------+ +------------------+

|

+-------------------+------------------+

| |

+----------------+ +-------------------+

| Inscription | | Soumettre un Dossier|

+----------------+ +-------------------+

| |

+------------------+ +---------------------+

| Se connecter | | Suivre l'état du Dossier|

+------------------+ +---------------------+

| |

+------------------+ +---------------------+

| Réinitialiser | | Modifier un Dossier |

| mot de passe | +---------------------+

+------------------+ |

| +-------------------+

+-------------------+ | Soumettre une Réclamation |

| Modifier infos | +-------------------+

+-------------------+ |

| +-------------------+

+-------------------+ | Suivre une Réclamation |

| Consulter historique| +-------------------+

| des demandes | |

+-------------------+ +-------------------+

| | Prendre un Rendez-vous |

+-------------------+ +-------------------+

| Recevoir notification | +-------------------+

+-------------------+ |

| +-------------------+

+-------------------+ | Modifier/Réserver |

| Annuler/Réserver | | Rendez-vous |

| un Rendez-vous | +-------------------+

+-------------------+