Dossier Produit — Reglemento

Product Owner — Nicolas

4 août 2025

Table des matières

1 Vision Produit

En tant que PO, mon objectif avec Reglemento est de répondre à un besoin criant exprimé par de nombreux indépendants et PME : comprendre et anticiper les évolutions réglementaires qui les concernent. Le système doit leur offrir des alertes claires, ciblées et compréhensibles pour les aider à rester conformes sans effort inutile.

2 Problème identifié

Les utilisateurs finaux expriment une difficulté à :
— Identifier les obligations légales pertinentes
— Interpréter les textes souvent techniques ou juridiques
— Être notifiés à temps des changements réglementaires majeurs

3 Proposition de valeur

Nous souhaitons livrer une plateforme simple mais puissante permettant à chaque utilisateur :
— de configurer ses préférences réglementaires,
— de recevoir des alertes pertinentes,
— de bénéficier d'un résumé intelligible,
— et d'agir en fonction (archiver, s'informer, déléguer...).

4 Composants techniques prévus

Chef de projet et Tech Lead, voici la stack que je propose :
— **Frontend Angular** (type SPA) : accès client
— **Backend ASP.NET Core (v8)** : API REST, logique métier
— **Scraping Python (FastAPI)** : microservice isolé
— **PostgreSQL** : persistance des données
— **Auth0** : gestion externalisée des identités

5 Fonctionnalités MVP validées

5.1 Configuration du profil utilisateur

— Pays, secteur(s), thématiques sélectionnables
— Stockage en base (table `UserPreferences`)

5.2 Réception d'alertes personnalisées

— Scraper = producteur d'alertes
— Matching des alertes avec profils
— Affichage des alertes dans dashboard utilisateur

5.3 Authentification déléguée (Auth0)

— Intégration OAuth via angular-auth0
— Utilisation de token JWT côté .NET pour sécuriser les endpoints

6 Détail des modules & responsabilités

6.1 Backend (.NET 8)

Organisation modulaire inspirée de Clean Architecture :
— **Domain** : Entités & règles métiers
— **Application** : Cas d'usage (CQRS)
— **Infrastructure** : Auth0, PostgreSQL, emailing futur
— **Web API** : Exposition des endpoints sécurisés
Modules :
— `Users` : persistance et mise à jour du profil
— `Alerts` : génération, diffusion, consultation
— `Admin` : suivi, logs, édition manuelle

6.2 Frontend (Angular)

— `auth` : auth.service.ts + guard + login UI
— `preferences` : configuration de la veille
— `dashboard` : affichage des alertes
— `shared/ui` : composants transversaux

7 Roadmap technique proposée

1. Mise en place des modules auth (Angular/Auth0) + base .NET
2. Implémentation du module `preferences`
3. Création du scraper FastAPI (hors MVP mais nécessaire pour tests)
4. Génération et distribution d'alertes simulées
5. Construction du dashboard utilisateur

8 Schémas de données (MVP)

8.1 API du service principales.

8.1.1 Table `Users`

— `Id` (UUID, PK)
— `Auth0Id` (string, unique)
— `Email` (string)
— `CreatedAt` (datetime)

8.1.2 Table `UserPreferences`

— `Id` (UUID, PK)
— `UserId` (UUID, FK → Users.Id)
— `Countries` (array[string])
— `Sectors` (array[string])
— `Themes` (array[string])
— `UpdatedAt` (datetime)

8.1.3 Table `Alerts`

— `Id` (UUID, PK)
— `Title` (string)
— `Summary` (text)
— `SourceUrl` (string)
— `PublishedAt` (datetime)
— `Country` (string)
— `SectorTags` (array[string])
— `ThemeTags` (array[string])

8.1.4 Table `UserAlerts`

Table de jointure pour historiser les alertes vues ou non.
— `Id` (UUID, PK)
— `UserId` (UUID, FK → Users.Id)
— `AlertId` (UUID, FK → Alerts.Id)
— `IsRead` (bool)
— `ReceivedAt` (datetime)

9 10. Implémentation du scraping avec Python (FastAPI)

Le module de scraping est un microservice indépendant développé en Python, utilisant le framework FastAPI pour exposer une API REST permettant d'intégrer les données extraites dans la base principale.

Architecture technique

— **Framework principal :** FastAPI, choisi pour sa légèreté et sa rapidité.
— **Bibliothèques et APIs de scraping :**
  — `requests` ou `httpx` pour les requêtes HTTP.
  — `BeautifulSoup` (bs4) ou `lxml` pour le parsing HTML.
  — `Scrapy` (optionnel) pour des scrapes plus complexes et asynchrones.
  — **APIs publiques gouvernementales :** où disponibles, privilégier l'utilisation d'APIs REST publiques officielles comme EUR-Lex API pour récupérer les données réglementaires au format structuré.
  — `Selenium` ou `Playwright` pour le scraping dynamique de sites web utilisant beaucoup de JavaScript.

— **Gestion des tâches planifiées :** utilisation de `APScheduler` ou de `Celery` avec un broker Redis pour automatiser les tâches de scraping à intervalles réguliers.
— **Base de données :** PostgreSQL (partagée ou dédiée selon architecture), accessible via `SQLAlchemy` ou `Tortoise ORM`.

Sources officielles et APIs gouvernementales

1. **EUR-Lex** : accès au droit de l'Union européenne (traités, directives, règlements)
   `https://eur-lex.europa.eu/`
   API officielle : `https://eur-lex.europa.eu/content/tools/api.html`
2. **Belgique - Moniteur belge** : publication officielle des lois et règlements belges
   `http://www.ejustice.just.fgov.be/cgi/welcome.pl`
   Pas d'API officielle, scraping HTML nécessaire.
3. **France - Legifrance** : textes législatifs et réglementaires français
   `https://www.legifrance.gouv.fr/`
   API disponible : `https://www.data.gouv.fr/en/datasets/api-legifrance/`
4. **Allemagne - Bundesgesetzblatt** : lois fédérales allemandes
   `https://www.bgbl.de/`
   Pas d'API officielle, scraping nécessaire.
5. **Portail européen des données** : point d'accès centralisé aux données ouvertes gouvernementales
   `https://data.europa.eu/en`
   Contient plusieurs jeux de données réglementaires en open data.

APIs exposées par le service de scraping

Le service FastAPI expose plusieurs endpoints REST sécurisés pour gérer les opérations de scraping et la communication avec l'application principale :

— `POST /scrape/start` : démarre manuellement une tâche de scraping pour une source donnée (URL ou API).
— `GET /scrape/status/task_id` : récupère le statut d'une tâche de scraping (en cours, terminée, erreur).
— `GET /scrape/results/task_id` : récupère les résultats structurés d'une tâche spécifique.
— `POST /data/push` : endpoint utilisé par le service de scraping pour envoyer les données extraites vers l'API principale (authentifié).
— `GET /sources/list` : liste des sources réglementaires disponibles pour le scraping.
— `POST /sources/add` : ajoute une nouvelle source à surveiller.
— `DELETE /sources/delete/source_id` : supprime une source existante.

Chaque endpoint est sécurisé par authentification JWT ou API key, avec gestion des permissions selon les rôles.

Fonctionnalités clés

— **Extraction des sources réglementaires :**
  — Téléchargement des pages officielles depuis les sites gouvernementaux européens et nationaux.

  - Utilisation des APIs publiques si disponibles pour récupérer les données au format JSON ou XML.
  - Extraction du contenu HTML brut stocké dans la table `RawSources` si pas d'API disponible.
- **Nettoyage et structuration des données :**
  - Parsing des contenus pour extraire les titres, dates, résumés, thèmes et secteurs.
  - Enrichissement des données avec métadonnées.
  - Stockage dans la table `ExtractedRegulations`.
- **API REST sécurisée :**
  - Endpoint pour pousser les données vers l'application principale.
  - Authentification via token JWT ou API key.
- **Logging et monitoring :**
  - Journalisation des exécutions dans `ScraperLogs`.
  - Gestion des erreurs et notifications.

Démarrage du développement

1. Initialiser un projet FastAPI simple avec `uvicorn` pour lancer le serveur.
2. Implémenter un endpoint REST `/scrape/start` déclenchant un scraping manuel pour tests.
3. Créer les fonctions de scraping avec `requests` + `BeautifulSoup` ciblant un site de test.
4. Persister le contenu HTML brut dans la base.
5. Ajouter la partie parsing et extraction de données structurées.
6. Mettre en place la planification avec `APScheduler` ou `Celery`.
7. Implémenter un endpoint sécurisé `/data/push` pour envoyer les données vers l'API principale .NET.

9.1 Tables des données

Table `Sources`

Liste des sources à scraper.
- `source_id` (UUID) : identifiant unique de la source
- `name` (string) : nom descriptif de la source (ex : "EUR-Lex")
- `url` (string) : URL de la page ou API à scraper
- `type` (string) : type de source ("API", "HTML", "PDF", etc.)
- `frequency` (int) : fréquence en heures entre chaque scraping
- `last_scraped` (datetime) : date de dernière récupération
- `active` (bool) : source active ou non

Table `RawSources`

Historise les sources réglementaires brutes extraites par scraping.
- `Id` (UUID, PK)
- `SourceUrl` (string)
- `HtmlContent` (text)

— `ScrapedAt` (datetime)
— `Country` (string)
— `SourceType` (enum : `BOE`, `EUR-Lex`, etc.)

Table `ExtractedRegulations`

Données nettoyées et enrichies automatiquement à partir de ***`RawSources`****.*
— `Id` (UUID, PK)
— `RawSourceId` (UUID, FK → RawSources.Id)
— `Title` (string)
— `Summary` (text)
— `DetectedThemes` (array[string])
— `DetectedSectors` (array[string])
— `PublishedAt` (datetime)
— `Language` (string)

Table `ScraperLogs`

Journal des exécutions de tâches de scraping.
— `Id` (UUID, PK)
— `StartedAt` (datetime)
— `FinishedAt` (datetime)
— `Status` (enum : `SUCCESS`, `FAILURE`)
— `Log` (text)

Évolutions possibles

— Intégration de techniques NLP pour améliorer l'extraction thématique.
— Support multilingue (FR, EN, NL, DE, etc.).
— Dashboard de supervision des tâches de scraping.

10 Analyse fonctionnelle – API .NET Core

Objectif de l'API

L'API .NET représente le cœur transactionnel de l'application Reglemento. Elle doit :

1. Centraliser, valider et exposer les données réglementaires extraites via le scrapper Python.
2. Gérer les utilisateurs, profils, abonnements, préférences, filtres et alertes.
3. Offrir une couche métier stable, évolutive, documentée et sécurisée.
4. Servir d'interface unique pour le front Angular et d'éventuels autres clients (mobile, partenaires).

Modules fonctionnels principaux

Module	Description
User Management	Authentification (via Auth0), gestion des utilisateurs, rôles et préférences
Profil Métier	Déclaration du secteur d'activité, taille entreprise, obligations réglementaires
Moteur de Notification	Génération des alertes à partir des règles et des préférences utilisateur
Gestion des Règlements	Stockage, catégorisation, versioning, visualisation des textes
Historique et Logs	Journalisation des interactions utilisateurs, alertes envoyées, modifications
Bridge avec le Scrapper	Endpoint sécurisé pour réception des données extraites par le service Python

Architecture technique recommandée

- **Framework** : ASP.NET Core 8 (LTS)
- **Architecture** : Modulaire, Clean Architecture ou Onion (domain-centric)
- **Design Patterns** : CQRS, Mediator (MediatR), Repository, Unit of Work, DTO/-Mapper (AutoMapper)
- **Sécurité** : Auth0 avec middleware JWT pour sécuriser toutes les routes

Structure proposée :

```
/Reglemento.API           → couche d'exposition (Controllers, Swagger)
/Reglemento.Application   → logique métier, use-cases, CQRS
/Reglemento.Domain        → entités, interfaces, enums, règles métier
/Reglemento.Infrastructure → persistence, EF Core, impl. API externes
/Reglemento.Shared        → constantes, helpers, middlewares communs
```

Technologies et packages recommandés

Fonctionnalité	Librairie
Authentification	Auth0.AspNetCore.Authentication
Architecture CQRS	MediatR, FluentValidation
Persistance	Entity Framework Core (PostgreSQL)
Mapping	AutoMapper
Documentation	Swashbuckle/Swagger
Sécurité	JWT Bearer Auth, CORS, HTTPS enforcement
Background jobs	Hangfire ou Quartz.NET (optionnel)

10.1 Schéma de données.

Table `Users`

- `Id` (UUID, PK)
- `Auth0Id` (string, unique)
- `Email` (string)
- `CreatedAt` (datetime)

Table `UserPreferences`

- `Id` (UUID, PK)
- `UserId` (UUID, FK → Users.Id)
- `Countries` (array[string])
- `Sectors` (array[string])
- `Themes` (array[string])
- `UpdatedAt` (datetime)

Table `Alerts`

- `Id` (UUID, PK)
- `Title` (string)
- `Summary` (text)
- `SourceUrl` (string)
- `PublishedAt` (datetime)
- `Country` (string)
- `SectorTags` (array[string])
- `ThemeTags` (array[string])

Table `UserAlerts`

Table de jointure pour historiser les alertes vues ou non.

- `Id` (UUID, PK)
- `UserId` (UUID, FK → Users.Id)
- `AlertId` (UUID, FK → Alerts.Id)
- `IsRead` (bool)
- `ReceivedAt` (datetime)

Authentification et Sécurité

1. Auth0 pour OIDC, login par email/mot de passe, social login (optionnel)
2. L'API .NET ne gère pas l'identification native, mais valide les JWT fournis par Auth0
3. Middleware d'autorisation pour restreindre certains endpoints à certains rôles (`User`, `Admin`, `Scraper`, etc.)

APIs exposées

Endpoint	Méthode	Description
`/api/users/profile`	GET/PUT	Récupération et mise à jour des préférences utilisateurs
`/api/alerts`	GET	Liste des alertes générées pour l'utilisateur courant
`/api/regulations/search`	GET	Recherche par mot-clé, date, secteur, type de texte
`/api/sources/push`	POST	Endpoint sécurisé (JWT Scraper) pour recevoir les nouvelles données depuis Python
`/api/logs`	GET	Historique des actions d'un utilisateur connecté

Scénarios métier MVP

1. Un utilisateur s'inscrit (via Auth0), choisit son secteur et reçoit sa première alerte.
2. Un administrateur peut consulter les règlements déjà extraits.
3. Le scrapper pousse une nouvelle loi via un POST, qui est vérifiée, stockée et liée aux bons profils.

Contraintes et Vigilances

1. L'API devra être hébergée localement en dev, et sur Azure/App Service ou Docker VPS plus tard.
2. Intégration fine avec Auth0 pour tester les rôles et extraire les infos du `access_token`.
3. Résilience au parsing et à l'intégration des textes réglementaires (doublons, erreurs, mises à jour).
4. Prévoir un mécanisme de mise à jour automatique (background task) pour réévaluer les alertes.

11 Analyse fonctionnelle – Application Angular

Objectif de l'application web

L'application Angular représente l'interface principale des utilisateurs finaux, leur permettant de :

— Créer et gérer leur profil d'entreprise ou d'indépendant.
— Visualiser les alertes réglementaires pertinentes.
— Consulter les textes de lois liés à leur secteur.
— Gérer leurs préférences, notifications et abonnements.

Modules principaux

Module	Description
Onboarding	Inscription via Auth0, sélection du secteur, finalisation du profil
Dashboard	Vue synthétique des alertes, documents récents, actions à entreprendre
Alertes	Liste filtrable, consultation des alertes, marquage comme lues
Règlements	Accès aux textes légaux, fiches synthétiques et historique des changements
Préférences	Configuration des secteurs suivis, types de notifications, fréquence
Administration	(si Admin connecté) : gestion des utilisateurs, validation des textes, logs

Technologies et structure technique recommandées

- **Framework** : Angular 17+ avec TypeScript strict
- **Architecture** : Modulaire (feature modules), routing lazy-loaded
- **State management** : `@ngrx/store` ou `Signal Store` (expérimental) si la logique devient complexe
- **UI** : `Angular Material` ou `Tailwind CSS` avec des composants personnalisés
- **Auth** : SDK Auth0 Angular
- **API** : Consommation des endpoints REST de l'API .NET sécurisée par JWT

Structure proposée :

```
/src/app
  /core            → services (auth, API, interceptors), guards
  /shared          → composants réutilisables, pipes, directives
  /features
    /onboarding    → inscription, choix secteur
    /dashboard     → alertes, résumé
    /regulations   → moteur de recherche
    /preferences   → réglages de l'utilisateur
    /admin         → panel administratif
  /assets          → icônes, logos, styles
```

Authentification et sécurité

- Intégration Auth0 SPA SDK avec redirection après login/logout
- Stockage du token en mémoire (pas de localStorage)
- Interceptor HTTP pour injecter automatiquement le JWT dans les appels vers l'API .NET
- Gestion des rôles (via claims Auth0) pour afficher/masquer certaines sections

Expérience utilisateur (UX)

— Interface claire, accessible, mobile-first
— Alertes mises en évidence avec niveau de criticité
— Mise en favoris, archivage, filtrage avancé
— Notification toast ou email en cas de nouvelle alerte

APIs consommées

Endpoint	**Méthode**	**Utilisation**
`/api/users/profile`	GET/PUT	Préférences utilisateur
`/api/alerts`	GET	Affichage des alertes
`/api/regulations/search`	GET	Moteur de recherche
`/api/logs`	GET	Historique de consultation

Scénarios MVP

— Un utilisateur s'inscrit, configure son profil métier, et arrive sur un dashboard personnalisé.
— Il reçoit automatiquement une alerte correspondant à une mise à jour réglementaire.
— Il peut consulter les détails de cette alerte, marquer le règlement comme lu ou pertinent.

Contraintes et risques

— Le découpage doit anticiper la scalabilité et la maintenance future (multi-tenant ? internationalisation ?)
— Attention à la sécurité des appels API et à l'usurpation de rôles dans le JWT
— Expérience fluide même en conditions de mauvaise connexion (UX offline partiel)

Conclusion

Notre but est d'avoir un premier livrable utilisable en 4 semaines, avec authentification fonctionnelle, dashboard et préférences. Cela permet d'engager un panel d'utilisateurs pour les premiers retours et corrections avant enrichissement futur (rappels, IA de résumé, export PDF...).