

# Rapport Personnel de Projet : Solution IoT ESEO

Auteur : Tarek SAMMMOUD

# Table des matières

<b>1 Description Technique de l'Implémentation</b>	<b>2</b>
1.1 Architecture du Système d'Alertes . . . . .	2
1.2 Focus sur des Implémentations Complexes . . . . .	2
1.2.1 Déclenchement Intelligent et Prévention des Doublons . . . . .	2
1.2.2 Gestion de la Résolution Automatique . . . . .	3
<b>2 Suivi des Défis et Problèmes Rencontrés</b>	<b>5</b>
<b>3 Environnement de Développement [Hors notation]</b>	<b>6</b>
<b>4 Bilan de l'Apprentissage</b>	<b>7</b>

# Chapitre 1

# Description Technique de l'Implémentation

Au sein du groupe, ma responsabilité principale a porté sur la conception et l'implémentation du **Système d'Alertes et de Monitoring**. Cette fonctionnalité constitue le "cerveau" réactif de la plateforme, transformant les données de capteurs en événements critiques gérables par les utilisateurs.

## 1.1 Architecture du Système d'Alertes

Le système repose sur un mécanisme de surveillance en temps réel intégré au flux de données. J'ai structuré cette fonctionnalité autour de trois piliers :

- **Détection Automatique** : Surveillance proactive de chaque nouveau relevé par rapport aux seuils configurés.
- **Gestion du Cycle de Vie** : Passage d'une alerte à travers différents états (**Active**, **Acquittée**, **Résolue**).
- **Monitoring et Statistiques** : Agrégation des données d'alertes pour le dashboard via le `AlerteRepository.cs`.

## 1.2 Focus sur des Implémentations Complexes

### 1.2.1 Déclenchement Intelligent et Prévention des Doublons

L'un des défis était d'éviter la création d'alertes redondantes pour une même anomalie persistante. Cette logique est orchestrée lors de la création d'un relevé.

```
1 private async Task VerifierEtGérerAlertes(Relevé relevé)
2 {
3     var seuils = await
4         _seuilAlerteService.GetActiveBySondeAsync(relevé.SondeId);
5     foreach (var seuil in seuils)
6     {
7         bool estEnAlerte = (seuil.TypeSeuil == TypeSeuil.Minimum &&
8             relevé.Valeur < seuil.Valeur) ||
9                 (seuil.TypeSeuil == TypeSeuil.Maximum &&
10                relevé.Valeur > seuil.Valeur);
11
12         if (estEnAlerte)
13         {
14             // Vérification si une alerte est déjà en cours pour ce
15             // seuil précis
```

```

12     var alerteExiste = await
13         _alerteService.GetActiveBySondeAndSeuilAsync(releve.SondeId,
14             seuil.Id);
15     if (alerteExiste == null)
16     {
17         await _alerteService.CreerAlerteAsync(releve.SondeId,
18             releve.Valeur, seuil);
19     }
20     else
21     {
22         // Tentative de résolution automatique si la valeur est
23         // revenue la normale
24         await
25             _alerteService.ResoudreAlertesSiNecessaireAsync(releve.SondeId,
26                 releve.Valeur, seuil);
27     }
28 }

```

Listing 1.1 – Extrait de la logique dans ReleveService.cs

**Analyse technique :** L'intelligence ici réside dans le couplage entre `ReleveService` et `AlerteService`. Le système ne se contente pas de crier à l'anomalie ; il vérifie d'abord si le problème est déjà connu (`alerteExiste == null`), évitant ainsi de saturer la base de données et l'interface utilisateur de notifications inutiles.

### 1.2.2 Gestion de la Résolution Automatique

La résolution automatique permet au système de se "réparer" tout seul d'un point de vue informationnel dès que les conditions environnementales redeviennent normales.

```

1 public async Task ResoudreAlertesSiNecessaireAsync(Guid sondeId,
2     decimal valeurMesuree, SeuilAlerteDto seuil)
3 {
4     var alertesActives = await
5         _repository.GetActiveAlertesBySondeAndTypeAsync(sondeId,
6             seuil.TypeSeuil);
7
8     foreach (var alerte in alertesActives)
9     {
10        bool retourNormal = (seuil.TypeSeuil == TypeSeuil.Minimum &&
11            valeurMesuree >= seuil.Valeur) ||
12                (seuil.TypeSeuil == TypeSeuil.Maximum &&
13                    valeurMesuree <= seuil.Valeur);
14
15        if (retourNormal)
16        {
17            alerte.Statut = StatutAlerte.Resolue;
18            alerte.DateResolution = DateTime.Now;
19            alerte.Message += $" (Résolue automatiquement
20                {DateTime.Now:HH:mm})";
21            await _repository.UpdateAsync(alerte);
22        }
23    }
24 }

```

Listing 1.2 – Extrait de AlertService.cs

**Analyse technique :** Cette méthode assure la fermeture de la boucle d'alerte. J'ai pris le soin de ne pas écraser le message initial, mais d'y apposer une mention temporelle, garantissant une traçabilité totale sur la durée de l'incident directement dans l'entité `Alerte.cs`.

## Chapitre 2

# Suivi des Défis et Problèmes Rencontrés

- **Cohérence du Mapping** : Pour ce module, j'ai délibérément évité l'utilisation de Mapperly pour certaines transformations complexes (notamment la construction de messages dynamiques). Cela a augmenté la verbosité du code mais a permis d'injecter une logique métier contextuelle plus fine lors de la création d'alertes.
- **Concurrence de Traitement** : Dans un environnement où les relevés arrivent par rafales (via HttpPull), il y avait un risque de race condition. J'ai dû implémenter des vérifications d'existence robustes (AnyAsync) au niveau de l'infrastructure pour garantir l'unicité des alertes actives.
- **Relations SQL Complexes** : La gestion des relations entre Alert, Sonde et SeuilAlerte nécessitait une attention particulière sur le *eager loading*. J'ai optimisé les requêtes dans le Repository pour éviter le problème de performance "N+1" lors de l'affichage du dashboard.

## Chapitre 3

# Environnement de Développement [Hors notation]

- **Outil choisi : Visual Studio 2022.**
- **Raison du choix :** C'est l'IDE que je maîtrise le mieux pour l'écosystème .NET. Sa puissance pour le refactoring et l'intégration native avec SQL Server Object Explorer ont été des facteurs déterminants.

#### Avis Critique :

- **Points Positifs :** L'explorateur de tests et les outils de diagnostic de performance ont été cruciaux pour valider la réactivité du système d'alertes sous charge simulée.
- **Points Négatifs :** La lourdeur de l'application au démarrage et la consommation mémoire parfois excessive sur de longs cycles de développement.

## Chapitre 4

# Bilan de l'Apprentissage

Ce projet a été une étape clé dans mon parcours de développeur .NET :

1. **Maîtrise de la Clean Architecture** : Comprendre comment la logique métier d'un service peut influencer un autre de manière propre (via les interfaces).
2. **Logique d'État (FSM)** : Concevoir un cycle de vie d'entité robuste (**Active -> Acquittée -> Résolue**) avec des transitions automatiques et manuelles.
3. **Optimisation SQL** : Apprendre à utiliser `Include()` et les projections de manière judicieuse pour maintenir la fluidité d'un dashboard de monitoring.