1. Introduction et buts

1.1 Apercu des exigences

Une application Web qui offre des fonctionnalités pour la gestion de chaine de magasins

Exigences Fonctionnelles

Id	Spécification de l'exigence fonctionnelle
EF01	L'application doit permettre au gestionnaire de générer un rapport détaillé des magasins.
EF02	L'application doit permettre à l'employé d'un magasin de consulter le stock disponible.
EF03	L'application doit permettre à l'employé de faire une demande de réapprovisionnement lorsque le stock de son magasin est en rupture.
EF04	L'application doit permettre au gestionnaire de visualiser les performances des magasins dans un tableau de bord.
EF05	L'application doit permettre au responsable de modifier un produit depuis la maison mère.
EF06	L'application doit permettre au responsable de valider une demande de réapprovisionnement.

Les cas d'utilisation du système

Id	Fonction
UC1	Générer un rapport consolidé des ventes
UC2	Consulter le stock central et déclencher un réapprovisionnement
UC3	Visualiser les performances des magasins dans un tableau de bord
UC4	Mettre à jour les produits depuis la maison mère
UC6	Approvisionner un magasin depuis le centre logistique

1.2 Qualité à atteindre

Exigences non-fonctionnelles

1. **Performance** : Les opérations (consultation du stock, enregistrement des ventes) doivent s'exécuter en moins de 2 secondes.
2. **Robustesse** : Le système doit gérer correctement les erreurs de saisie et les accès invalides à la base de données.
3. **Cohérence des données** : Les écritures dans la base de données doivent être atomiques et respecter les transactions pour éviter les incohérences (surtout en cas d'accès concurrent par plusieurs caisses).

4. **Fiabilité** : Le système doit garantir l'intégrité des données même en cas d'interruption du processus (plantage ou arrêt inattendu).
5. **Portabilité** : L'application doit être exécutable dans un conteneur Docker pour assurer une installation uniforme sur toute machine.
6. **Facilité de déploiement** : Le système doit pouvoir être lancé avec une seule commande (`docker compose up`) pour simplifier le test et l'installation.
7. **Maintenabilité** : Le code doit être structuré en couches séparées (présentation, logique métier, persistance) pour faciliter l'évolution future du projet.

1.3 Parties prenantes

Rôle	Attentes
Développeur	Architecture claire pour guider le développement.
Utilisateurs	Interface intuitive pour les opérations quotidiennes.
Employé	Interface utile pour consulter le stock et faire des demandes de réapprovisionnement avec une gestion d'erreurs simple.
Gestionnaire	Interface intuitive pour générer un rapport sur les magasins et visualiser leurs performances avec des indicateurs utiles pour prendre des décisions importantesquotidiennes.
Responsable	Interface intuitive pour valider des demandes de réapprovisionnement en cours et modifier des produits des magaisns.

2 Contraintes

- Technologie back-end : C# avec .NET 8 MVC.
- Base de données : PostgreSQL 15+ via le provider Npgsql.
- Conteneurisation : Docker & Docker Compose.
- CI/CD : GitHub Actions pour build, tests et publication d'image Docker.
- ORM : Entity Framework Core 8.0.
- Tests : xUni.

3 Contexte et portée

3.1 Contexte métier

Cette application Web sert de gestion centralisée pour une chaîne de 4 magasins de détail, avec un stock local propre à chaque point de vente et un stock central partagé.

3.2 Contexte technique

- Serveur Web : ASP.NET Core MVC, hébergé dans un conteneur Docker.
- Clients : Navigateurs Web modernes (Chrome, Edge, Firefox).
- Base de données : PostgreSQL déployée dans un conteneur séparé.
- Accès aux données : EF Core pour les requêtes LINQ et les migrations.
- Authentification : hors périmètre du labo 2.
- Portée : Mise en place des UC1 à UC6, tests unitaires et d'intégration, et automatisation Docker. Les UC7 à UC8 ne sont pas inclus en raison de contraintes de temps de développement.

4 Stratégie de la solution

1. Architecture N-Tiers : Contrôleurs (MVC), Services (logique métier), Data (EF Core, DbContext).
2. Modèles seedés : DataSeeder initialise Produits, Magasins, Stocks locaux et central, Ventes, et pour Lab 2 opc : Demandes.
3. Services : Une interface + implémentation par UC (RapportService, ReapprovisionnementService, PerformanceService, ProduitService).
4. Contrôleurs MVC : Un pour chaque UC, vues Razor pour interface.
5. Tests : xUnit pour services, contrôleurs, intégration avec InMemoryProvider d'EF.
6. Déploiement : Docker Compose lance deux conteneurs (Web + DB) et exécute dotnet ef database update à l'entrée.

5. Vue des blocs de construction

Application Web (ASP.NET Core MVC) :

- Controllers/
- Views/
- wwwroot/
- Services/
- Data/
- Models

Base de données PostgreSQL:

- Tables/Entities: Produit, Vente, StockCentral, DemandeReapprovisionnement, MagasinStockProduit, Magasin

6 Vue d'éxécution

Processus - Modifier un produit

Processus

Processus - Obtenir rapport consolidé

Processus

7 Vue de déploiement

Vue

8 Conceptes transversaux

- Seed de données : DataSeeder appelé dans OnModelCreating pour initialiser la BD.
- Migrations EF Core : centralisées dans le projet Web, appliquées en startup.
- Transaction : EF Core gère implicitement les transactions dans SaveChangesAsync();
- Validation : DataAnnotations + ModelState dans MVC + validation côté client.
- Messages : TempData pour plusieurs redirections (succès/erreur) et ViewData pour passer des informations ponctuelles.

9 Décision architectural

ADR#1 Choix des technologies : C#, PostgreSQL, Entity Framework Core

Status

Acceptée

Contexte

Je dois développer une application de caisse avec une interface console, une persistance robuste, et une compatibilité avec CI/CD et Docker. Il est nécessaire de choisir un langage, une base de données, et une méthode d'accès aux données (ORM) adaptée.

Décision

J'ai décidé d'utiliser :

- Le langage **C#** avec le framework .NET 8, car l'étudiant est déja familier avec ces technologies
- La base de données relationnelle **PostgreSQL**, car elle est open-source, robuste, compatible avec Docker, et bien supportée par Entity Framework. De plus, l'étudiant possède seulement de l'éxperience avec les bases de données relationnelles en PostgreSQL
- L'ORM **Entity Framework Core**, car il s'intègre naturellement avec C#/.NET, simplifie les requêtes, gère les migrations, et prend en charge PostgreSQL via le provider `Npgsql`.

Conséquences

- Plus facile à démarrer grâce à l'expérience existante avec C# et .NET.
- Intégration fluide dans Docker et dans GitHub Actions.
- Meilleure productivité grâce à l'automatisation des migrations EF Core.

ADR#2 Séparation des responsabilités : Présentation et Persistance

Status

Acceptée

Contexte

L'application doit rester maintenable, claire, et évolutive. Pour cela, il est essentiel de bien séparer la couche présentaion, logique métier et le serveur qui contient les données essentielles de l'application. Une architecture N-tier à 3 couches est donc un bon choix

Décision

J'ai choisi de structurer l'application en 3 couches :

- **Présentation** : Les pages web dont l'utilisateur utilise pour intéragir avec l'application (Dossier: Controllers & Views)
- **Logique du Métier** : Contient la logique pour faire les opérations necessaires (Dossier: Services, Models, et Data)
- **Persistance** : accès aux données via l'ORM Entity Framework

Conséquences

- Facilite les tests unitaires indépendants de la console ou de la base de données.
- Meilleure lisibilité et modularité du code.
- Prépare l'architecture à une éventuelle migration vers une API ou une interface graphique.

10 Exigences de qualité

10.1 Fiabilité

Les approbations de demandes (UC6) doivent se faire atomiquement : stock central, stock local et statut doivent être cohérents.

Validation stricte des paramètres (quantités > 0, ID existants).

10.2 Performance

Les pages critiques (rapport UC1, dashboard UC3) doivent répondre en moins de 500 ms en lecture sur un volume seedé (~100 enregistrements).

Les opérations comme générer un rapport, modifier un produit, et visualiser les performances des magasins doivent s'exécuter en moins de 1 s.

10.3 Sécurité

Validation côté serveur de tous les inputs.

10.4 Testabilité

Les controleurs et les services sont testés pour garantir un application fiable.

10.5 Deployabilité

Une seule commande : docker compose up --build.

11 Risques et dettes techniques

Risque: La couverture des tests n'est pas 100%. Il exisite du code que lorsque changé, il y a pas de moyen pour empecher une erreur d'etre mit en prod.

Dette technique identifiée : Pas de gestion du workflow utilisateur (pas d’authentification réelle, droits). À couvrir dans une itération future. Tests unitaires manquantes pour l'application.

12 Glossaire

Terme	Définition
Entity Framework	Un mappeur objet-relationnel (ORM) pour .NET.
Docker Compose	Un outil pour définir et gérer des applications multi-conteneurs Docker.
DTO	Un objet pour transférer des données entre les couches afin de s'assurer le découplage.