Entityframework Core

m2iformation.fr

Programme

1. **RAPPELS DES CONCEPTS DE BASE**

- Rappel l'ORM (Object-Relational Mapping)
  - Rappel mappage objet-relationnel
  - Comparaison avec les approches traditionnelles
- Approche Database First vs Code First
  - Database First : principes et cas d'utilisation.
  - Code First : principes et cas d'utilisation.
  - Comparaison des avantages et des inconvénients.

2. **MODELES ET CONTEXTE**

- Création des Modèles (Entités) et Objet de Contexte
  - Génération de modèles à partir de la base de données existante.
  - Configuration du DbContext pour l'approche Database First.
- Data Annotations et Fluent API
  - Utilisation des Data Annotations pour la validation et la configuration.
  - Configuration avancée avec Fluent API.

Programme

3. **MIGRATION ET GESTION DE LA BASE DE DONNEES**

- Gestion des migrations
  - Création et application des migrations pour synchroniser le modèle et la base de données.
  - Outils de migration en ligne de commande et dans Visual Studio.
- Gestion des Contraintes d'Intégrité Référentielle
  - Implémentation et gestion des relations (one-to-one, one-to-many, many-to-many).
  - Gestion des contraintes de clés étrangères.

4. **TRANSACTIONS ET CONCURRENCE**

- Notion de Transaction
  - Gestion des transactions avec Entity Framework Core.
  - Implémentation des transactions explicites et implicites.
- Verrouillage Optimiste vs Pessimiste
  - Concepts de base et implémentation des deux types de verrouillage.
  - Scénarios d'utilisation et résolution des conflits.

Programme

5. **REFACTORING ET PATTERNS**

- Refactoring et Découplage des Composants
  - Utilisation de l'Inversion de Contrôle (IoC) et de l'Injection de Dépendances (DI).
  - Découplage des couches de l'application pour une meilleure testabilité et maintenabilité.
- Pattern Unit of Work
  - Introduction au pattern Unit of Work.
  - Implémentation et intégration avec Repository Pattern.

6. **OPTIMISATION DES PERFORMANCES ET GESTION DES DONNEES**

- Tracking des Entités avec EF Core
  - Différence entre le suivi des entités et les requêtes no-tracking.
  - Utilisation efficace du suivi des entités pour optimiser les performances.
- Filtrage des Données et Limitation des Résultats
  - Utilisation de LINQ pour filtrer les données.
  - Techniques pour limiter les résultats des requêtes.
- Mise à Jour en Masse
  - Méthodes pour effectuer des mises à jour en masse de manière efficace

Programme

7. **APPROCHE CQRS ET GESTION DE LA CONCURRENCE D'ACCES**

- Approche CQRS (Command Query Responsibility Segregation)
  - Principes de base de CQRS.
  - Implémentation de CQRS avec Entity Framework Core.
- Gestion de la Concurrence d'Accès avec ADO.NET et EF Core
  - Stratégies pour gérer la concurrence d'accès.
  - Comparaison et intégration de ADO.NET avec EF Core.

8. **TECHNIQUES AVANCEES ET RESOLUTION DE PROBLEMES**

- Présentation des Problèmes à Surmonter en Consultation
  - Optimisation des requêtes pour la lecture.
  - Techniques pour éviter les problèmes de performance.
- Présentation des Problèmes à Surmonter en Mise à Jour
  - Gestion des conflits de mise à jour.
  - Stratégies pour minimiser les impacts sur les performances.
- IEnumerable vs IQueryable
  - Différences entre IEnumerable et IQueryable.
  - Scénarios d'utilisation et impact sur les performances.

RAPPELS DES CONCEPTS DE BASE

1 Rappel de l'ORM (Object-Relational Mapping)

1.1 Rappel du Mappage Objet-Relationnel

Le mappage objet-relationnel (ORM) est une technique qui permet de traduire les objets dans le code (classes) en structures relationnelles dans une base de données (tables), et vice versa. L'objectif est de permettre aux développeurs de travailler avec des objets tout en s'appuyant sur une base de données relationnelle sans écrire directement du SQL.

```sql
CREATE TABLE Users (
    Id INT PRIMARY KEY,
    Name NVARCHAR(50),
    Email NVARCHAR(100)
);
```

```csharp
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```

```csharp
// Ajouter un utilisateur
context.Users.Add(new User { Name = "Alice", Email = "alice@example.com" });
context.SaveChanges();

// Récupérer un utilisateur
var user = context.Users.Find(1);

// Supprimer un utilisateur
context.Users.Remove(user);
context.SaveChanges();
```

RAPPELS DES CONCEPTS DE BASE

1.2 Comparaison avec les Approches Traditionnelles

Critère	ORM	Approche Traditionnelle (SQL brut)
Facilité d'utilisation	Les développeurs manipulent des objets directement.	Nécessite de connaître et écrire du SQL.
Abstraction	Cache la logique SQL sous-jacente.	Nécessite une interaction directe avec SQL.
Portabilité	Indépendant du type de base de données (SQL Server, MySQL, etc.).	Les requêtes SQL doivent être adaptées en fonction du SGBD utilisé.
Optimisation	Automatisation qui peut être moins efficace pour des cas complexes.	Contrôle total des requêtes pour optimiser les performances.
Temps de développement	Réduction grâce à la manipulation orientée objets.	Plus long, car tout doit être codé manuellement.

RAPPELS DES CONCEPTS DE BASE

2 Approche Database First vs Code First

Entity Framework Core prend en charge deux principales approches pour travailler avec une base de données : **Database First** et **Code First**.

2.1 Database First : Principes et Cas d'Utilisation

- **Principe :**
  L'approche **Database First** consiste à partir d'une base de données existante. Les modèles d'entité (classes C#) et le DbContext sont générés automatiquement en fonction des tables de la base.
- **Cas d'utilisation :**
  - Vous travaillez avec une base de données existante.
  - La structure de la base ne doit pas être modifiée par l'application.
  - Idéal pour les projets de maintenance.
- **Commandes :**
  Pour générer les classes et le contexte à partir d'une base existante, utilisez la commande suivante dans la console de gestionnaire de package :

```
Scaffold-DbContext "Server=localhost;Database=MyDatabase;User Id=myUser;Password=myPassword;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

- **Exemple :**
  Si une table SQL ressemble à :

```
CREATE TABLE Products (
   ProductId INT PRIMARY KEY,
   Name NVARCHAR(100),
   Price DECIMAL(10, 2)
);
```

EF Core génère automatiquement :

```
public class Product
{
      public int ProductId { get; set; }
      public string Name { get; set; }
      public decimal Price { get; set; }
}
```

```
public class MyDbContext : DbContext
{
      public DbSet<Product> Products { get; set; }
}
```

RAPPELS DES CONCEPTS DE BASE

2.2 Code First : Principes et Cas d'Utilisation

- **Principe :**
  L'approche **Code First** consiste à définir d'abord les modèles (classes C#). EF Core génère ensuite la structure de la base de données à partir de ces modèles.
- **Cas d'utilisation :**
  - Vous démarrez un nouveau projet où la structure de la base est flexible.
  - Vous voulez contrôler l'évolution des modèles via du code.
- **Commandes :**
  Une fois les modèles créés, vous générez les migrations et appliquez les changements avec :

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

RAPPELS DES CONCEPTS DE BASE

- **Exemple :**
  Une classe C# représentant un produit :

```
public class Product
{
   public int ProductId { get; set; }
   public string Name { get; set; }
   public decimal Price { get; set; }
}
```

EF Core génère automatiquement une table SQL :

```
CREATE TABLE Products (
   ProductId INT PRIMARY KEY,
   Name NVARCHAR(100),
   Price DECIMAL(10, 2)
);
```

RAPPELS DES CONCEPTS DE BASE

2.3 Comparaison des Avantages et Inconvénients

Critère	Database First	Code First
Flexibilité	Limité à la structure existante de la base.	Haute, car les modèles définissent la structure.
Apprentissage	Plus simple pour les bases déjà conçues.	Peut nécessiter des connaissances avancées sur EF Core.
Maintenance	Idéal pour les bases de données stables.	Meilleur pour les projets en évolution constante.
Migration	Les changements doivent être reflétés dans le code.	Les migrations gèrent automatiquement les changements.

MODELES ET CONTEXTE

1 Création des Modèles (Entités) et Objet de Contexte

1.1 Génération de modèles à partir de la base de données existante

Lorsque vous utilisez l'approche **Database First**, les entités (modèles) et le DbContext sont générés automatiquement à partir d'une base de données existante.

Étapes pour générer des modèles :

1. Installez les packages nécessaires dans votre projet :

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

2. Exécutez la commande de scaffolding pour générer les modèles et le DbContext :

```
dotnet ef dbcontext scaffold "Server=localhost;Database=MyDatabase;User Id=myUser;Password=myPassword;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

3. Les fichiers générés incluent :
   - Un fichier `MyDbContext.cs` contenant la définition du contexte.
   - Un fichier pour chaque table de la base de données.

MODELES ET CONTEXTE

Exemple :

Si votre base de données contient une table `Products` :

```
CREATE TABLE Products (
    ProductId INT PRIMARY KEY,
    Name NVARCHAR(100),
    Price DECIMAL(10, 2)
);
```

La commande génère une classe C# pour représenter cette table :

```
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

MODELES ET CONTEXTE

Ainsi qu'un DbContext :

```
public partial class MyDbContext : DbContext
{
    public MyDbContext(DbContextOptions<MyDbContext> options)
        : base(options)
    {
    }
    public virtual DbSet<Product> Products { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>(entity =>
        {
            entity.HasKey(e => e.ProductId);
            entity.Property(e => e.Name).IsRequired().HasMaxLength(100);
            entity.Property(e => e.Price).HasColumnType("decimal(10, 2)");
        });
    }
}
```

MODELES ET CONTEXTE

1.2 Configuration du DbContext pour l'approche Database First

Le `DbContext` est le cœur d'Entity Framework Core. Il agit comme une porte d'entrée vers la base de données.

Configuration dans `Program.cs` :

Ajoutez le DbContext au conteneur de services :

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();

app.Run();
```

MODELES ET CONTEXTE

Explication des Méthodes Principales :

Méthode	Description
`DbSet<TEntity>`	Représente une table de la base de données (exemple : `Products`).
`OnModelCreating`	Configure les entités et leurs relations au niveau du modèle avec Fluent API.
`SaveChanges`	Sauvegarde toutes les modifications faites sur les entités dans la base de données.

MODELES ET CONTEXTE

2 Data Annotations et Fluent API

Entity Framework Core offre deux moyens principaux pour configurer vos modèles :

1. **Data Annotations** : Configuration simple directement dans les classes avec des attributs.
2. **Fluent API** : Configuration avancée dans la méthode `OnModelCreating`.

2.1 Utilisation des Data Annotations

Les **Data Annotations** sont des attributs appliqués directement sur les propriétés des modèles pour définir des règles de validation ou de mappage.

MODELES ET CONTEXTE

```
public class Product
{
    [Key] // Indique que cette propriété est la clé primaire
    public int ProductId { get; set; }

    [Required] // Rend cette propriété obligatoire
    [MaxLength(100)] // Définit une longueur maximale
    public string Name { get; set; }

    [Range(0.01, 9999.99)] // Définit une plage valide pour cette propriété
    public decimal Price { get; set; }

    [DataType(DataType.Date)] // Indique que cette propriété représente une date
    public DateTime CreatedDate { get; set; }
}
```

MODELES ET CONTEXTE

Annotation	Description
`Key`	Définit une propriété comme clé primaire.
`Required`	Rend une propriété obligatoire.
`MaxLength`	Définit une longueur maximale pour une chaîne de caractères.
`Range`	Définit une plage de valeurs acceptables.
`DataType`	Spécifie le type de données (Date, Email, etc.).

MODELES ET CONTEXTE

2.2 Configuration avancée avec Fluent API

La **Fluent API** permet de configurer des règles complexes ou de remplacer les annotations.

Exemple :

Le même modèle configuré avec la Fluent API :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>(entity =>
    {
        entity.HasKey(p => p.ProductId); // Définit la clé primaire
        entity.Property(p => p.Name)
              .IsRequired() // Rend obligatoire
              .HasMaxLength(100); // Définit une longueur maximale
        entity.Property(p => p.Price)
              .HasColumnType("decimal(10, 2)"); // Définit le type SQL exact
        entity.Property(p => p.CreatedDate)
              .HasColumnType("date"); // Définit le type SQL pour une date
    });
}
```

MODELES ET CONTEXTE

Comparaison entre Data Annotations et Fluent API :

Critère	Data Annotations	Fluent API
Simplicité	Facile à lire et à écrire dans les classes.	Nécessite d'écrire dans `OnModelCreating`.
Complexité	Limité à des configurations simples.	Permet des configurations avancées.
Séparation des Concerns	Mélange le code métier et la configuration.	Sépare clairement la logique métier.

MIGRATION ET GESTION DE LA BASE DE DONNÉES

1 Gestion des migrations

Les **migrations** en Entity Framework Core permettent de synchroniser les modifications des modèles (entités) avec la base de données sous-jacente. Cette synchronisation est essentielle pour éviter les incohérences entre le code et la base de données.

1.1 Création et application des migrations

Étape 1 : Ajouter une migration

1. Ajoutez une migration lorsque vous modifiez vos modèles. Une migration capture les modifications sous forme de script SQL.

   Commande :

```
dotnet ef migrations add MigrationName
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Exemple : Si vous ajoutez un champ `Description` dans l'entité `Product` :

```
public class Product
{
      public int ProductId { get; set; }
      public string Name { get; set; }
      public decimal Price { get; set; }
      public string Description { get; set; } // Nouveau champ
}
```

```
dotnet ef migrations add AddDescriptionToProduct
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Cela génère un fichier de migration dans le dossier `Migrations` :

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "Description",
        table: "Products",
        type: "nvarchar(max)",
        nullable: true);
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropColumn(
        name: "Description",
        table: "Products");
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Étape 2 : Appliquer une migration

Une fois la migration créée, appliquez-la à la base de données.

Commande :

```
dotnet ef database update
```

Cela exécute la méthode `Up` de la migration, ajoutant le champ `Description` à la table `Products`.

MIGRATION ET GESTION DE LA BASE DE DONNÉES

1.2 Outils de migration

1. **Ligne de commande (CLI) :**
    - **Ajouter une migration :**

```
dotnet ef migrations add MigrationName
```

- **Appliquer une migration :**

```
dotnet ef database update
```

- **Supprimer une migration :**

```
dotnet ef migrations remove
```

2. **Visual Studio :**
    - Ouvrez la **Console du Gestionnaire de Package**.
    - Commandes similaires :

```
Add-Migration MigrationName
Update-Database
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

2 Gestion des Contraintes d'Intégrité Référentielle

Entity Framework Core permet de gérer les relations entre les entités tout en respectant les contraintes d'intégrité référentielles (ex. : clés étrangères).

2.1 Implémentation des relations

1. Relation One-to-One

Une relation un-à-un signifie qu'une entité est associée à exactement une autre entité.

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Exemple :

Une entité `User` possède une entité `UserProfile`.

```
public class User
{
    public int UserId { get; set; }
    public string Name { get; set; }

    public UserProfile Profile { get; set; }
}

public class UserProfile
{
    public int UserProfileId { get; set; }
    public string Bio { get; set; }

    public int UserId { get; set; }
    public User User { get; set; }
}
```

Fluent API :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>()
        .HasOne(u => u.Profile)
        .WithOne(p => p.User)
        .HasForeignKey<UserProfile>(p => p.UserId);
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

2. Relation One-to-Many

Une relation un-à-plusieurs signifie qu'une entité peut être associée à plusieurs autres entités.

Exemple :

Une entité `Category` peut avoir plusieurs `Products`.

```
public class Category
{
    public int CategoryId { get; set; }
    public string Name { get; set; }

    public ICollection<Product> Products { get; set; }
}

public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public int CategoryId { get; set; }
    public Category Category { get; set; }
}
```

Fluent API :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Category>()
        .HasMany(c => c.Products)
        .WithOne(p => p.Category)
        .HasForeignKey(p => p.CategoryId);
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

3. Relation Many-to-Many

Une relation plusieurs-à-plusieurs signifie que plusieurs entités peuvent être associées à plusieurs autres entités.

Exemple :

Une entité `Student` peut être inscrite à plusieurs `Courses`, et un `Course` peut inclure plusieurs `Students`.

Configuration simplifiée (EF Core 5+) :

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public ICollection<Course> Courses { get; set; }
}
public class Course
{
    public int CourseId { get; set; }
    public string Title { get; set; }

    public ICollection<Student> Students { get; set; }
}
```

Fluent API :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>()
        .HasMany(s => s.Courses)
        .WithMany(c => c.Students)
        .UsingEntity(j => j.ToTable("StudentCourses"));
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

2.2 Gestion des contraintes de clés étrangères

Les clés étrangères garantissent l'intégrité référentielle entre les entités liées.

Exemple :

Pour une relation `Product` et `Category`, EF Core configure automatiquement une contrainte de clé étrangère (`CategoryId` dans `Products`).

Problématique : Suppression en cascade

Si vous supprimez une entité parent, les entités enfant associées doivent être supprimées ou protégées.

Solution avec Fluent API :

Configurer la suppression en cascade ou restreindre la suppression :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>()
        .HasOne(p => p.Category)
        .WithMany(c => c.Products)
        .HasForeignKey(p => p.CategoryId)
        .OnDelete(DeleteBehavior.Cascade); // Suppression en cascade
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Problématiques et Solutions

1. Problème : Modifier une relation existante

Si vous modifiez une relation dans les modèles, EF Core pourrait générer des erreurs liées à la base de données existante.

Solution :

- Supprimez d'abord la relation avec une migration.
- Appliquez la nouvelle relation dans une autre migration.

2. Problème : Relations circulaires

Lorsque deux entités se réfèrent mutuellement, une boucle infinie peut se produire.

Solution :

- Utilisez des attributs comme `[JsonIgnore]` pour éviter la sérialisation circulaire.
- Configurez explicitement les relations avec Fluent API.

MIGRATION ET GESTION DE LA BASE DE DONNÉES

L'approche Database First implique de partir d'une base de données existante et d'ajouter des migrations pour gérer des modifications ou synchroniser les changements avec le code.

Étape 1 : Générer les modèles à partir de la base de données

Si ce n'est pas déjà fait, générez les modèles et le contexte à partir d'une base de données existante avec la commande suivante :

```
dotnet ef dbcontext scaffold "Server=localhost;Database=MyDatabase;User Id=myUser;Password=myPassword;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Cela génère :

1. Les entités (classes représentant les tables).
2. Le `DbContext` configuré pour votre base de données.

Exemple de `Product` généré à partir d'une table existante :

```
public partial class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Exemple de `DbContext` :

```
public partial class MyDbContext : DbContext
{
    public MyDbContext(DbContextOptions<MyDbContext> options) : base(options) { }

    public virtual DbSet<Product> Products { get; set; }
}
```

Étape 2 : Ajouter des modifications au modèle généré

Ajoutez un nouveau champ ou une nouvelle entité au code pour étendre les modèles existants.

Exemple : Ajout d'un champ `Description` à la classe `Product`.

```
public partial class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    public string Description { get; set; } // Nouveau champ ajouté
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Étape 3 : Ajouter une migration

Générez une migration qui capture les changements entre le modèle existant et la base.

Commande :

```
dotnet ef migrations add AddDescriptionToProduct
```

Cela crée un fichier de migration dans le dossier `Migrations`.

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Exemple de fichier de migration généré :

```
public partial class AddDescriptionToProduct : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<string>(
            name: "Description",
            table: "Products",
            type: "nvarchar(max)",
            nullable: true);
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "Description",
            table: "Products");
    }
}
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Étape 4 : Appliquer la migration

Appliquez la migration à la base de données pour synchroniser les modifications.

Commande :

```
dotnet ef database update
```

Étape 5 : Vérifiez la base de données

Vérifiez que la colonne `Description` a été ajoutée à la table `Products`.

Exemple de table après modification :

```
CREATE TABLE Products (
    ProductId INT PRIMARY KEY,
    Name NVARCHAR(100),
    Price DECIMAL(10, 2),
    Description NVARCHAR(MAX) NULL
);
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Problématique 1 : Modifications complexes en Database First

Problème :

- Si vous ajoutez de nombreuses modifications complexes (nouvelles entités, relations, champs), il peut être difficile de conserver la synchronisation avec la base existante.

Solution :

- Scindez les modifications en plusieurs migrations.
- Utilisez les fichiers SQL générés par EF Core pour vérifier les scripts avant l'application.

Problématique 2 : Perte de synchronisation entre la base et le modèle

Problème :

- Si des modifications sont apportées directement à la base (sans passer par le modèle), votre code risque de ne plus être aligné.

Solution :

- Regénérez les modèles avec `Scaffold-DbContext` :

```
dotnet ef dbcontext scaffold "Server=localhost;Database=MyDatabase;User Id=myUser;Password=myPassword;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

  - Cette commande met à jour les classes en fonction de la base de données existante.

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Exemple : Ajouter une relation en Database First

Ajoutons une relation entre `Product` et une nouvelle table `Category`.

Étape 1 : Ajouter une nouvelle table à la base de données

Ajoutez une table `Categories` à la base de données :

```
CREATE TABLE Categories (
    CategoryId INT PRIMARY KEY,
    Name NVARCHAR(100)
);

ALTER TABLE Products
ADD CategoryId INT;

ALTER TABLE Products
ADD CONSTRAINT FK_Products_Categories FOREIGN KEY (CategoryId) REFERENCES Categories(CategoryId);
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Étape 2 : Regénérer les modèles

Regénérez les modèles pour inclure la nouvelle relation :

```
dotnet ef dbcontext scaffold "Server=localhost;Database=MyDatabase;User Id=myUser;Password=myPassword;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Étape 3 : Mise à jour du code généré

La relation apparaît dans les modèles :

```
public partial class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
    public int? CategoryId { get; set; } // Nouvelle relation
    public virtual Category Category { get; set; }
}

public partial class Category
{
    public int CategoryId { get; set; }
    public string Name { get; set; }
```

MIGRATION ET GESTION DE LA BASE DE DONNÉES

Étape 4 : Vérifier la relation

Ajoutez des données pour tester la relation :

```
using var context = new MyDbContext(options);

var category = new Category { Name = "Electronics" };
context.Categories.Add(category);

var product = new Product { Name = "Laptop", Price = 999.99m, Category = category };
context.Products.Add(product);

context.SaveChanges();
```

Sujet de TP : Gestion d'une Plateforme de Réservations avec Implémentation du Unit of Work

Contexte

Vous développez une application de gestion de réservations pour une plateforme unique utilisée par plusieurs entreprises. Chaque entreprise peut gérer ses propres salles et réservations, mais tout est stocké dans une seule base de données partagée. L'objectif est de concevoir une application robuste, modulaire, et performante en intégrant les concepts d'Entity Framework Core (EF Core) et les patterns avancés tels que **Unit of Work (UoW)**.

TP

Étape 1 : Rappels des Concepts de Base et Mise en Place

L'objectif est de poser les bases du projet en utilisant EF Core pour connecter l'application à une base de données existante et en ajoutant des entités et relations nécessaires.

Travail Pratique

1. **Création de la Base de Données**
   - Exécutez le script suivant pour initialiser la base de données avec des tables de base.
2. **Configuration de l'Application avec EF Core**
   - Créez un projet .NET Core.
   - Ajoutez les packages nécessaires (`Microsoft.EntityFrameworkCore`, `Microsoft.EntityFrameworkCore.SqlServer`).
   - Configurez un `DbContext` pour gérer les tables de la base.
3. **Modélisation des Entités**
   - Générez des entités EF Core pour représenter les tables `Company`, `Room`, et `Booking`.
   - Configurez les relations dans le `DbContext` :
     - Relation `One-to-Many` entre `Company` et `Room`.
     - Relation `One-to-Many` entre `Room` et `Booking`.

TP

4. **Ajout des Migrations**
   - Créez une migration pour synchroniser les entités avec la base de données.
   - Appliquez la migration.
5. **Validation avec Data Annotations**
   - Ajoutez des contraintes de validation comme `[Required]`, `[MaxLength]`, et `[Range]`.

TRANSACTIONS ET CONCURRENCE

Entity Framework Core (EF Core) gère les transactions de manière intégrée et permet aussi de manipuler des transactions personnalisées pour garantir la cohérence des données dans des scénarios complexes.

1 Notion de Transaction

Une **transaction** regroupe plusieurs opérations sur la base de données en une unité logique. Si une opération échoue, toutes les opérations précédentes sont annulées, garantissant ainsi la cohérence des données.

1.1 Gestion des transactions avec Entity Framework Core

Par défaut, EF Core utilise des **transactions implicites** pour chaque appel à `SaveChanges`. Cependant, dans des scénarios complexes, des **transactions explicites** peuvent être nécessaires pour regrouper plusieurs appels.

TRANSACTIONS ET CONCURRENCE

Transactions implicites

EF Core encapsule automatiquement chaque appel à `SaveChanges` dans une transaction. Si une erreur survient, EF Core annule automatiquement les modifications.

Exemple :

```
using var context = new AppDbContext(options);

var product = new Product { Name = "Laptop", Price = 999.99m };
context.Products.Add(product);

try
{
    context.SaveChanges(); // Transaction implicite
}
catch (Exception ex)
{
    Console.WriteLine($"Erreur lors de la sauvegarde : {ex.Message}");
}
```

TRANSACTIONS ET CONCURRENCE

Transactions explicites

Dans des scénarios nécessitant plusieurs opérations ou des dépendances entre elles, vous pouvez utiliser une transaction explicite.

Exemple :

```
using var context = new AppDbContext(options);

using var transaction = context.Database.BeginTransaction();
try
{
    var category = new Category { Name = "Electronics" };
    context.Categories.Add(category);
    context.SaveChanges(); // Opération 1
    var product = new Product { Name = "Smartphone", Price = 499.99m, CategoryId = category.CategoryId };
    context.Products.Add(product);
    context.SaveChanges(); // Opération 2
    transaction.Commit(); // Confirme la transaction
}
catch (Exception ex)
{
    transaction.Rollback(); // Annule toutes les modifications
    Console.WriteLine($"Erreur lors de la transaction : {ex.Message}");
```

TRANSACTIONS ET CONCURRENCE

Rétablissement automatique après échec

Scénario :

Une application effectue plusieurs écritures dans différentes tables. Si une opération échoue, toutes les modifications doivent être annulées.

Solution :

Utilisez une transaction explicite avec `Rollback` comme dans l'exemple précédent.

TRANSACTIONS ET CONCURRENCE

2 Verrouillage Optimiste vs Pessimiste

Le verrouillage gère les accès concurrents à une même donnée dans une base de données.

Critère	Verrouillage Optimiste	Verrouillage Pessimiste
Principe	Suppose que les conflits sont rares et vérifie à la fin.	Empêche d'autres transactions d'accéder à la donnée.
Performance	Rapide, car pas de verrou durant les opérations.	Plus lent à cause des verrous.
Utilisation	Idéal pour des applications web ou des systèmes multiuser.	Utile pour des systèmes nécessitant une précision élevée.

2.1 Verrouillage Optimiste

Avec le verrouillage optimiste, les modifications sont vérifiées avant d'être sauvegardées. EF Core utilise un champ de version (`RowVersion`) pour détecter les conflits.
Ajoutez un champ `RowVersion` à l'entité :

```
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    [Timestamp] // Champ de version pour le verrouillage optimiste
    public byte[] RowVersion { get; set; }
}
```

Gestion des conflits :
Si deux utilisateurs modifient le même produit simultanément :

1. L'utilisateur 1 charge le produit, modifie son prix et sauvegarde.
2. L'utilisateur 2 tente de sauvegarder, mais EF Core lève une exception.

TRANSACTIONS ET CONCURRENCE

Code de gestion des conflits :

```
try
{
    context.SaveChanges();
}
catch (DbUpdateConcurrencyException ex)
{
    Console.WriteLine("Conflit détecté. Les données ont été modifiées par un autre utilisateur.");
}
```

TRANSACTIONS ET CONCURRENCE

2.2 Verrouillage Pessimiste

Avec le verrouillage pessimiste, un utilisateur bloque une donnée dès qu'il commence à travailler dessus, empêchant les autres d'y accéder.
EF Core ne prend pas en charge directement le verrouillage pessimiste, mais vous pouvez utiliser des requêtes SQL brutes.

```
context.Database.ExecuteSqlRaw("SELECT * FROM Products WITH (UPDLOCK)");
```

Scénario typique :

1. Un utilisateur bloque un produit pour le modifier.
2. Les autres utilisateurs doivent attendre que le verrou soit libéré.

TRANSACTIONS ET CONCURRENCE

Scénarios d'utilisation et résolution des conflits

1. Optimiste : Scénario de mise à jour concurrente

- **Problème :**
  Deux utilisateurs modifient la même donnée simultanément.
- **Solution :**
  Utilisez le champ `RowVersion` pour détecter les conflits et alertez l'utilisateur affecté.

2. Pessimiste : Scénario de réservation

- **Problème :**
  Une ressource (ex. : une place de parking) doit être réservée par un utilisateur à la fois.
- **Solution :**
  Implémentez le verrouillage pessimiste pour bloquer la ressource jusqu'à ce que l'opération soit terminée.

TRANSACTIONS ET CONCURRENCE

Résumé des Commandes et Techniques

Action	Commande ou Technique
Démarrer une transaction	`context.Database.BeginTransaction()`
Confirmer une transaction	`transaction.Commit()`
Annuler une transaction	`transaction.Rollback()`
Verrouillage optimiste	Champ `[Timestamp]` ou gestion de `RowVersion`.
Verrouillage pessimiste	Requêtes SQL brutes avec `WITH (UPDLOCK)`.

TP

Étape 2 : Gestion des Transactions et Concurrence

L'objectif ici est de gérer efficacement les opérations complexes impliquant plusieurs entités et de garantir l'intégrité des données en cas de concurrence.

1. **Transactions Explicites**
   - Implémentez un scénario où une réservation est créée et vérifiez :
     - Que la salle existe.
     - Que la salle est disponible à la date demandée.
   - Utilisez une transaction pour valider ou annuler l'opération en cas d'erreur.
2. **Gestion des Conflits**
   - Ajoutez une colonne `RowVersion` à l'entité `Booking` pour gérer les conflits avec le **verrouillage optimiste**.
   - Simulez deux utilisateurs essayant de réserver la même salle à la même date.

REFACTORING ET PATTERNS

1 Refactoring et Découplage des Composants

Le **refactoring** consiste à améliorer la structure interne d'une application sans changer son comportement externe. Le **découplage** vise à réduire les dépendances directes entre les composants pour faciliter leur testabilité, leur réutilisation et leur maintenance.

1.1 Utilisation de l'Inversion de Contrôle (IoC) et de l'Injection de Dépendances (DI)

Inversion de Contrôle (IoC) est un principe où le contrôle du flux de l'application est inversé : au lieu que les composants contrôlent directement leurs dépendances, ces dernières leur sont fournies par un conteneur ou un orchestrateur.

Injection de Dépendances (DI) est une méthode concrète pour implémenter IoC en injectant les dépendances dans les classes via le constructeur, des méthodes ou des propriétés.

REFACTORING ET PATTERNS

Exemple : Avant Refactoring

Sans IoC ni DI, les classes créent leurs dépendances, ce qui entraîne un couplage fort.

```
public class ProductService
{
    private readonly MyDbContext _context;

    public ProductService()
    {
        _context = new MyDbContext(); // Couplage direct
    }

    public Product GetProductById(int id)
    {
        return _context.Products.Find(id);
    }
}
```

REFACTORING ET PATTERNS

Exemple : Après Refactoring avec DI

La dépendance `MyDbContext` est injectée, permettant une flexibilité accrue.

```
public class ProductService
{
    private readonly MyDbContext _context;

    public ProductService(MyDbContext context)
    {
        _context = context; // Dépendance injectée
    }

    public Product GetProductById(int id)
    {
        return _context.Products.Find(id);
    }
}
```

REFACTORING ET PATTERNS

Configuration dans `Program.cs` :

```
builder.Services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<ProductService>();
```

Avantages :

- **Testabilité :** Permet d'injecter un contexte simulé (`Mock`) lors des tests unitaires.
- **Flexibilité :** Changez facilement l'implémentation de la dépendance.

REFACTORING ET PATTERNS

1.2 Découplage des couches de l'application

Un découplage réussi repose sur une séparation claire entre :

1. **La couche de présentation** (UI),
2. **La couche de service** (logique métier),
3. **La couche de données** (accès à la base).

REFACTORING ET PATTERNS

Exemple d'architecture découpée :

Structure des couches :

- **UI** : Appelle les services.
- **Services** : Contient la logique métier.
- **Repositories** : Accède aux données.

Exemple de code :

```
// Contrôleur (UI Layer)
public class ProductController : ControllerBase
{
    private readonly ProductService _service;

    public ProductController(ProductService service)
    {
        _service = service;
    }

    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        var product = _service.GetProductById(id);
        return product != null ? Ok(product) : NotFound();
    }
}

// Service (Business Logic Layer)
public class ProductService
{
    private readonly IProductRepository _repository;

    public ProductService(IProductRepository repository)
    {
        _repository = repository;
    }

    public Product GetProductById(int id)
    {
        return _repository.GetById(id);
    }
}
```

REFACTORING ET PATTERNS

Avantages :

- Chaque couche a une responsabilité unique.
- Simplifie les tests unitaires pour chaque couche.
- Facilite les modifications et la maintenance.

REFACTORING ET PATTERNS

2 Pattern Unit of Work

Le **pattern Unit of Work** regroupe plusieurs opérations sur les données dans une unité logique, permettant de contrôler les transactions de manière centralisée. Ce pattern fonctionne souvent en tandem avec le **Repository Pattern**.

2.1 Introduction au pattern Unit of Work

Problématique :

- Si chaque repository appelle directement `SaveChanges` sur le contexte, cela peut entraîner des incohérences et des performances médiocres (multiples transactions).

Solution :

- Utiliser le pattern Unit of Work pour centraliser les appels à `SaveChanges` et regrouper plusieurs opérations dans une seule transaction.

REFACTORING ET PATTERNS

2.2 Implémentation du pattern Unit of Work

Interface :

```
public interface IUnitOfWork : IDisposable
{
    IProductRepository Products { get; }
    ICategoryRepository Categories { get; }
    void Save();
}
```

REFACTORING ET PATTERNS

Implémentation :

```
public class UnitOfWork : IUnitOfWork
{
    private readonly MyDbContext _context;
    public UnitOfWork(MyDbContext context)
    {
        _context = context;
        Products = new ProductRepository(_context);
        Categories = new CategoryRepository(_context);
    }
    public IProductRepository Products { get; private set; }
    public ICategoryRepository Categories { get; private set; }
    public void Save()
    {
        _context.SaveChanges();
    }
    public void Dispose()
    {
        _context.Dispose();
```

REFACTORING ET PATTERNS

2.3 Intégration avec le Repository Pattern

Le **Repository Pattern** encapsule les opérations CRUD pour une entité. En combinaison avec Unit of Work, cela fournit une structure propre pour accéder et manipuler les données.

Exemple de Repository :

```
public class ProductRepository : IProductRepository
{
    private readonly MyDbContext _context;
    public ProductRepository(MyDbContext context)
    {
        _context = context;
    }
    public Product GetById(int id)
    {
        return _context.Products.Find(id);
    }
    public void Add(Product product)
    {
        _context.Products.Add(product);
    }
```

REFACTORING ET PATTERNS

Exemple d'utilisation :

Un service utilise Unit of Work pour gérer les opérations :

```
public class OrderService
{
    private readonly IUnitOfWork _unitOfWork;

    public OrderService(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    public void PlaceOrder(Product product, Category category)
    {
        _unitOfWork.Categories.Add(category);
        _unitOfWork.Products.Add(product);

        // Toutes les opérations sont regroupées dans une seule transaction
        _unitOfWork.Save();
    }
```

REFACTORING ET PATTERNS

Problématiques et Solutions

1. **Problème : Multiples appels à `SaveChanges` dans différents repositories.**
   - **Solution :** Utilisez Unit of Work pour regrouper toutes les modifications en une seule transaction.
2. **Problème : Couplage direct entre services et contexte.**
   - **Solution :** Injectez Unit of Work dans les services pour abstraire l'accès aux données.
3. **Problème : Difficulté à tester les interactions avec la base.**
   - **Solution :** Mockez l'interface `IUnitOfWork` pour tester les services sans accéder à la base réelle.

REFACTORING ET PATTERNS

Concept	Avantages
IoC et DI	Facilite la testabilité et réduit le couplage.
Découplage des couches	Permet une séparation claire des responsabilités.
Unit of Work	Regroupe les modifications en une seule transaction pour plus de cohérence.
Repository Pattern	Simplifie l'accès aux données et rend les opérations CRUD réutilisables.

TP

Étape 3 : Refactoring avec le Pattern Unit of Work

Cette étape vise à structurer l'accès aux données en introduisant le pattern **Unit of Work** pour centraliser les transactions et les opérations sur les entités.

1. **Refactoring avec le Repository Pattern**
   - Créez une interface générique `IRepository<T>` définissant les opérations de base (`Add`, `Update`, `Delete`, `GetById`, `GetAll`).
   - Implémentez cette interface dans une classe `Repository<T>`.
2. **Implémentation du Unit of Work**
   - Créez une interface `IUnitOfWork` qui inclut :
     - Des propriétés pour accéder aux repositories (`IRepository<Company>`, `IRepository<Room>`, `IRepository<Booking>`).
     - Une méthode `SaveChangesAsync` pour valider les modifications.
   - Implémentez `IUnitOfWork` dans une classe `UnitOfWork`.

TP

3. **Ajout du Service de Gestion des Réservations**
   - Créez un service `BookingService` utilisant `IUnitOfWork` pour gérer les réservations.
   - Implémentez des méthodes comme `CreateBooking`, `GetBookingsByRoom`, et `CancelBooking`.
4. **Injection de Dépendances**
   - Configurez l'injection de dépendances pour enregistrer `UnitOfWork` et `Repository<T>` dans le conteneur de services.

Principes de Base de CQRS

1. **Séparation des Responsabilités** :
   - **Command** : Toute opération qui modifie l'état de l'application, comme la création, la mise à jour ou la suppression d'une entité.
   - **Query** : Toute opération qui interroge ou lit les données sans modifier l'état.
2. **Modèles Différents** :
   - Le modèle utilisé pour les écritures peut être optimisé pour la cohérence et l'intégrité des données (relationnel, stricte).
   - Le modèle utilisé pour les lectures peut être optimisé pour les performances, comme une base de données NoSQL ou une vue matérialisée.
3. **Bases de Données Séparées ou Partagées** :
   - **Deux Bases de Données (approche idéale)** : Une base pour les écritures et une autre pour les lectures. Cela offre des avantages comme l'optimisation des performances et la séparation totale des préoccupations.
   - **Une Base Unique (approche simplifiée)** : Une seule base avec des schémas ou des tables distincts pour séparer lecture et écriture. Cela simplifie la synchronisation mais peut être moins performant.

Pourquoi Utiliser CQRS ?

1. **Optimisation des Performances** :
   - Permet d'utiliser des bases optimisées pour les requêtes complexes (NoSQL, Elasticsearch) ou les lectures fréquentes.
2. **Évolutivité** :
   - Les lectures et les écritures peuvent être mises à l'échelle indépendamment.
3. **Flexibilité** :
   - Les modèles de lecture peuvent évoluer sans affecter les modèles d'écriture.
4. **Gestion de la Concurrence** :
   - Les écritures sont isolées, ce qui permet de mieux gérer les conflits liés aux accès concurrents.

Synchronisation Entre Lecture et Écriture

1. **Event Sourcing** (méthode idéale avec CQRS) :
   - Chaque écriture produit un événement qui est utilisé pour mettre à jour les modèles de lecture.
   - Ces événements sont persistés dans une base de données d'événements et propagés aux vues de lecture.
2. **Job de Synchronisation** (méthode simplifiée) :
   - Un processus asynchrone lit les modifications dans les modèles d'écriture et met à jour les vues de lecture.
3. **Base Partagée avec Triggers** :
   - Si une seule base est utilisée, des déclencheurs (triggers) ou des transactions peuvent synchroniser les tables.

Implémentation de CQRS avec Entity Framework Core

Exigences

1. Une **base pour l'écriture** utilisant EF Core pour gérer les transactions.
2. Une **base pour la lecture** qui peut être optimisée avec une solution comme Dapper ou un ORM léger.

Synchronisation (Event Sourcing)

1. **Écriture** : Lorsqu'une commande est exécutée, un événement est généré.
2. **Projection** : Un handler consomme l'événement et met à jour la vue de lecture.

Gestion de la Concurrence d'Accès

Stratégies avec EF Core

1. **Optimistic Concurrency** :
    - Ajouter un champ `RowVersion` pour détecter les conflits.

```
public class Order
{
    public int Id { get; set; }
    public string Product { get; set; }
    public int Quantity { get; set; }
    public byte[] RowVersion { get; set; }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Order>()
        .Property(o => o.RowVersion)
        .IsRowVersion();
}
```

Gestion de la Concurrence d'Accès

Gestion avec ADO.NET

1. **Transactions** :
    - Créer une transaction explicite pour gérer les conflits.

```
using (var connection = new SqlConnection("ConnectionString"))
{
    connection.Open();
    using (var transaction = connection.BeginTransaction())
    {
        // Effectuer des lectures et des écritures dans la transaction
        transaction.Commit();
    }
}
```

2. **Verrous Explicites** :
    - Utiliser des commandes SQL pour verrouiller les lignes nécessaires.

```
SELECT * FROM Orders WITH (UPDLOCK, ROWLOCK) WHERE Id = @Id
```

Comparaison ADO.NET vs EF Core pour la Concurrence

Critère	ADO.NET	EF Core
Performance	Plus performant pour des cas simples	Moins performant
Abstraction	Bas niveau	Haut niveau
Concurrence	Contrôle explicite	Gestion automatique

TP

Étape 4 : Introduction de CQRS

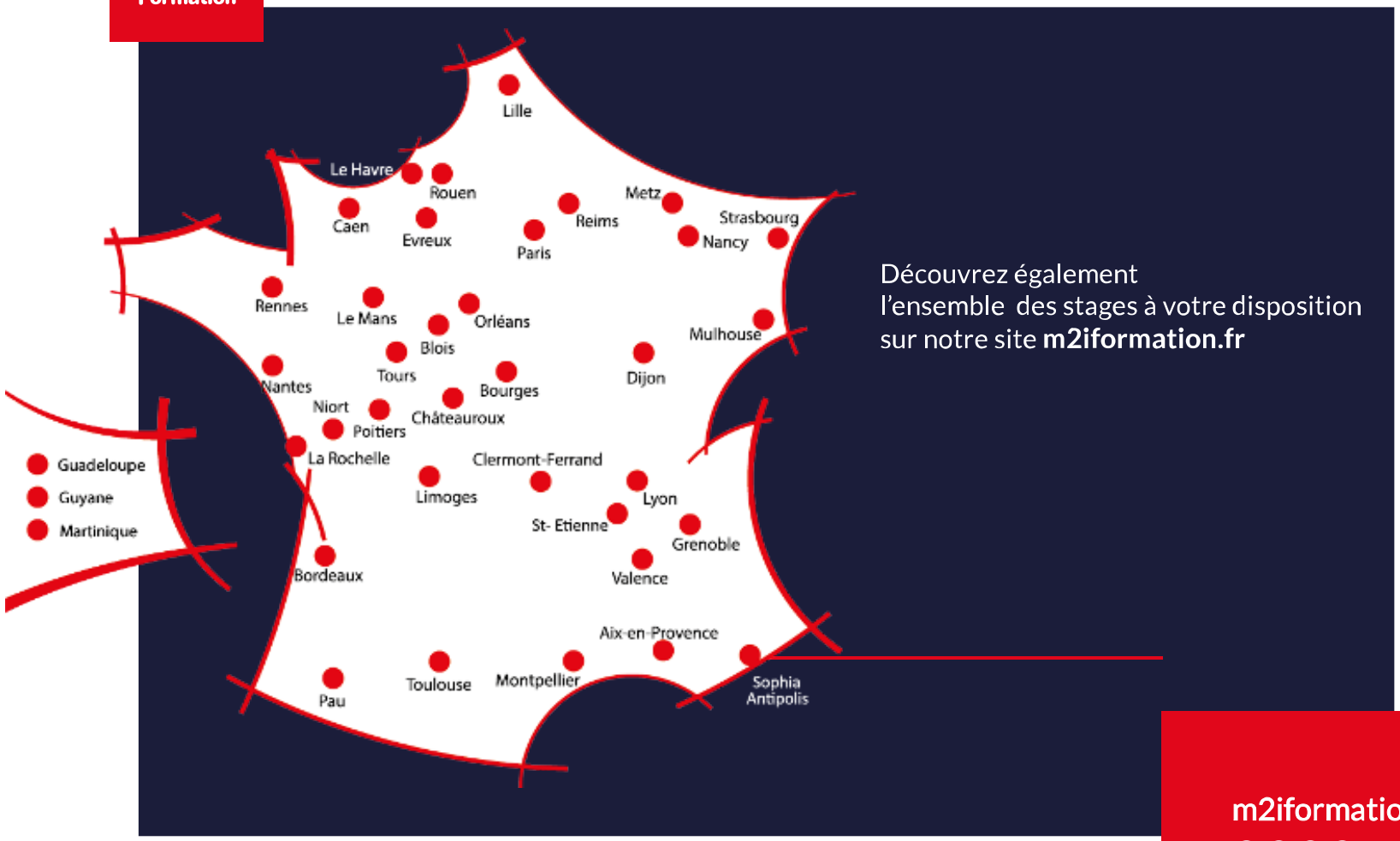
Pour séparer les commandes (écriture) des requêtes (lecture), cette étape introduit le pattern **CQRS** pour simplifier la gestion des opérations complexes.

1. **Séparation Lecture/Écriture**
   - Créez deux interfaces :
     - `ICommandHandler<TCommand>` pour gérer les commandes (ajout, modification, suppression).
     - `IQueryHandler<TQuery, TResult>` pour gérer les requêtes (lecture seule).
   - Implémentez un `CreateBookingCommandHandler` pour gérer la création des réservations.
   - Implémentez un `GetAvailableRoomsQueryHandler` pour récupérer les salles disponibles.
2. **Intégration au Contrôleur**
   - Ajoutez un contrôleur `BookingController` qui utilise les gestionnaires de commandes et requêtes via l'injection de dépendances.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr