Entity Framework (Core)

ESI 5D - ISITECH

C'est parti! →

Entity Framework Core

Pour le développement d'applications modernes

Formation intensive - Jour 1

Appuyez sur Espace pour commencer →

Programme des 4 jours

Jour 1 : Fondamentaux

- Introduction aux ORM et à EF Core
- Configuration de l'environnement
- Modélisation de données
- Context et DbSet

Jour 2 : CRUD et requêtes

- Migrations de base de données
- Opérations CRUD
- Requêtes avancées

Jour 3 : Relations et performances

- Modélisation des relations
- Chargement des données liées
- Optimisation des performances
- Héritage et types complexes

Jour 4 : Architecture

- Patterns de conception avec EF Core
- Tests et bonnes pratiques
- Scénarios avancés
- Projet final

Prérequis

- Connaissances en C# et programmation orientée objet
- Notions de bases de données relationnelles et SQL
- Expérience de base avec Visual Studio ou un autre IDE .NET
- Compréhension des concepts de base du développement web

Planning du jour 1 (Lundi)

Session 1 (9h00-10h30)

Introduction aux ORM et à Entity Framework Core

Session 2 (10h45-12h45)

Configuration de l'environnement et premier projet

Session 3 (13h30-15h30)

Modélisation de données avec EF Core

Session 4 (15h45-17h30)

Context et DbSet

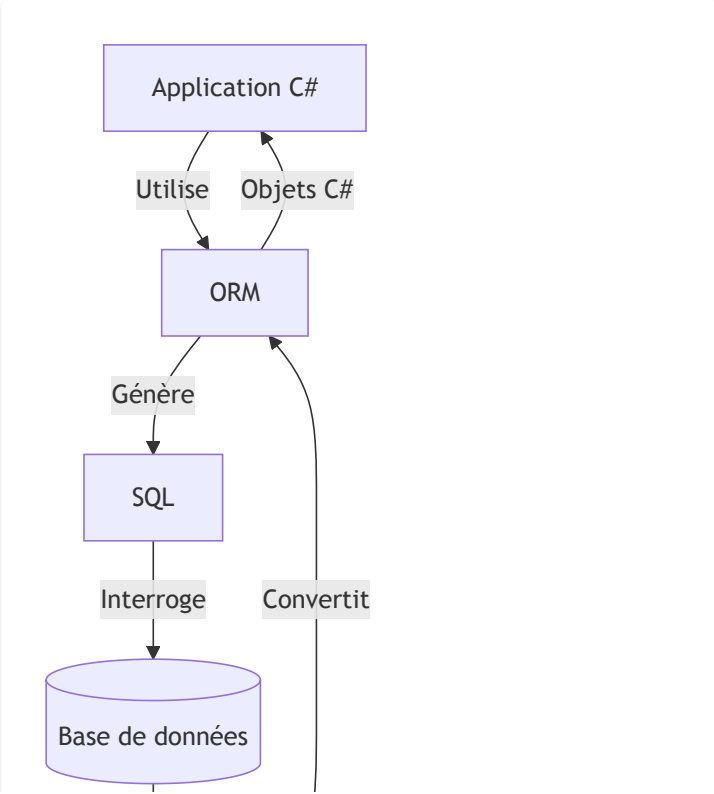
Objectifs de la journée

- Comprendre ce qu'est un ORM et pourquoi l'utiliser
- Mettre en place un environnement de développement
- Créer et configurer un premier projet avec EF Core

Qu’est-ce qu’un ORM?

Object-Relational Mapping

- Technique pour convertir des données entre systèmes incompatibles
- Crée une "couche virtuelle d’objets" qui représente la base de données
- Permet de manipuler des données sans écrire de SQL
- Réduit l’impédance objet-relationnel



Avantages des ORM

- Productivité accrue
- Code plus lisible et maintenable
- Moins de SQL manuel à écrire
- Abstraction de la base de données
- Gestion des migrations de schéma
- Protection contre les injections SQL
- Support du changement de SGBD

Inconvénients des ORM

- Courbe d’apprentissage
- Performances potentiellement moins optimales
- Requêtes complexes parfois difficiles à exprimer
- Risque de génération de requêtes inefficaces
- Sur-utilisation des ressources pour des cas simples
- Perte de contrôle sur le SQL généré

Histoire et évolution d'Entity Framework

EF "Classic" (EF6 et antérieur)

- Première version en 2008 (.NET 3.5)
- Approche "Database First" dominante
- Centré sur l'EDM (Entity Data Model)
- Fortement couplé à ADO.NET
- Performances limitées

EF Core 1.0 - 3.1

- Réécriture complète en 2016
- Open source (GitHub)
- Cross-platform (.NET Core)
- Approche "Code First" privilégiée
- API fluide améliorée
- Performances grandement améliorées

EF Core 5.0+

- Many-to-many sans entité de jointure
- Table par type (TPT) amélioré
- Split queries
- Filtres au niveau requête
- Meilleures performances
- Mappings de tables multiples
- EF Core 8.0: dernière version stable

Différences entre EF6 et EF Core

Architecture

- EF6: Construit sur ADO.NET
- EF Core: Réécriture complète, modulaire

Plateformes

- EF6: Windows uniquement (.NET Framework)
- EF Core: Cross-platform (.NET Core/.NET 5+)

Performance

Fonctionnalités

```
// EF Core: Support des expressions LINQ avancées
var result = await context.Customers
    .Where(c => c.Orders.Any(o => o.Total > 1000))
    .Select(c => new {
        c.Name,
        OrderCount = c.Orders.Count(),
        TotalValue = c.Orders.Sum(o => o.Total)
    })
    .ToListAsync();
```

Installation des outils

Visual Studio

- Visual Studio 2022 (Community, Professional, Enterprise)
- Extensions: ".NET Core Tools", "Entity Framework Tools"

SQL Server

- SQL Server Express ou Developer Edition
- SQL Server Management Studio (SSMS)

.NET SDK

- .NET 8 SDK ou supérieur

Structure d’un projet utilisant EF Core

Organisation recommandée

```
MyProject/
├── MyProject.Core/
│   ├── Entities/
│   │   ├── Customer.cs
│   │   ├── Order.cs
│   │   └── Product.cs
│   ├── Interfaces/
│   │   └── IRepository.cs
│   └── DTOs/
├── MyProject.Infrastructure/
│   ├── Data/
│   │   ├── ApplicationDbContext.cs
│   │   ├── Configurations/
│   │   └── Migrations/
│   └── Repositories/
└── MyProject.API/
    ├── Controllers/
    ├── Program.cs
```

Bonnes pratiques

- Séparation des préoccupations
- Core: domaine et logique métier
- Infrastructure: accès aux données
- API/UI: interface utilisateur

Injection de dépendances

```
// Program.cs
builder.Services.AddDbContext<ApplicationDbContex
    options.UseSqlServer(
        builder.Configuration.GetConnectionStrin
```

Création d'un premier projet

En utilisant Visual Studio

1. Créer un nouveau projet "ASP.NET Core Web API"
2. Configurer le projet (nom, emplacement)
3. Sélectionner le framework (.NET 8.0)
4. Ajouter les packages NuGet EF Core

En utilisant la CLI .NET

```
dotnet new webapi -n MyFirstEfCoreApp
cd MyFirstEfCoreApp
dotnet add package Microsoft.EntityFrameworkCore.Sql
dotnet add package Microsoft.EntityFrameworkCore.Des
dotnet add package Microsoft.EntityFrameworkCore.Too
```

Configuration de la base de données

```
// appsettings.json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyFirstEfCoreDb;Trusted_Connection=True;M
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information"
```

TP : Création d’un premier projet

Objectifs

1. Créer un nouveau projet "ASP.NET Core Web API"
2. Configurer EF Core avec SQL Server
3. Créer une première entité
4. Configurer le DbContext
5. Vérifier la configuration

Étapes détaillées

1. Utilisez Visual Studio ou la CLI .NET
2. Ajoutez les packages NuGet requis
3. Configurez la connexion dans appsettings.json
4. Créez un dossier "Models" ou "Entities"
5. Créez une classe "Product" avec des propriétés
6. Créez une classe "ApplicationDbContext"

Exemple de code d’entité

```
public class Product
```

Pause café ☕

Rendez-vous à 10h45 pour la suite

Modélisation de données avec EF Core

Entités et propriétés

- Classes POCO (Plain Old CLR Objects)
- Propriétés virtuelles pour le lazy loading
- Types de données et conversions
- Types de navigation (références, collections)

```
public class Customer
{
    public int Id { get; set; }  // Clé primaire par

    // Propriétés scalaires
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public DateTime DateOfBirth { get; set; }

    // Propriétés de navigation
    public virtual ICollection<Order> Orders { get;

    // Constructeur - initialiser les collections
    public Customer() => Orders = new HashSet<Order>
}
```

Conventions de nommage

- Id/ClassNameId pour les clés primaires
- ClassNameId pour les clés étrangères
- Pluralisation des noms de tables
- Conventions des index et contraintes

Annotations de données vs. Fluent API

Annotations de données

```
public class Product
{
    [Key]
    public int ProductId { get; set; }

    [Required]
    [MaxLength(100)]
    public string Name { get; set; }

    [Column("unit_price")]
    [Precision(18, 2)]
    public decimal Price { get; set; }

    [MaxLength(500)]
    public string Description { get; set; }

    [ForeignKey("Category")]
    public int CategoryId { get; set; }

    public virtual Category Category { get; set; }
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder
{
    modelBuilder.Entity<Product>(entity =>
    {
        entity.HasKey(e => e.ProductId);

        entity.Property(e => e.Name)
              .IsRequired()
              .HasMaxLength(100);

        entity.Property(e => e.Price)
              .HasColumnName("unit_price")
              .HasPrecision(18, 2);

        entity.Property(e => e.Description)
              .HasMaxLength(500);

        entity.HasOne(e => e.Category)
              .WithMany(c => c.Products)
              .HasForeignKey(e => e.CategoryId);
```

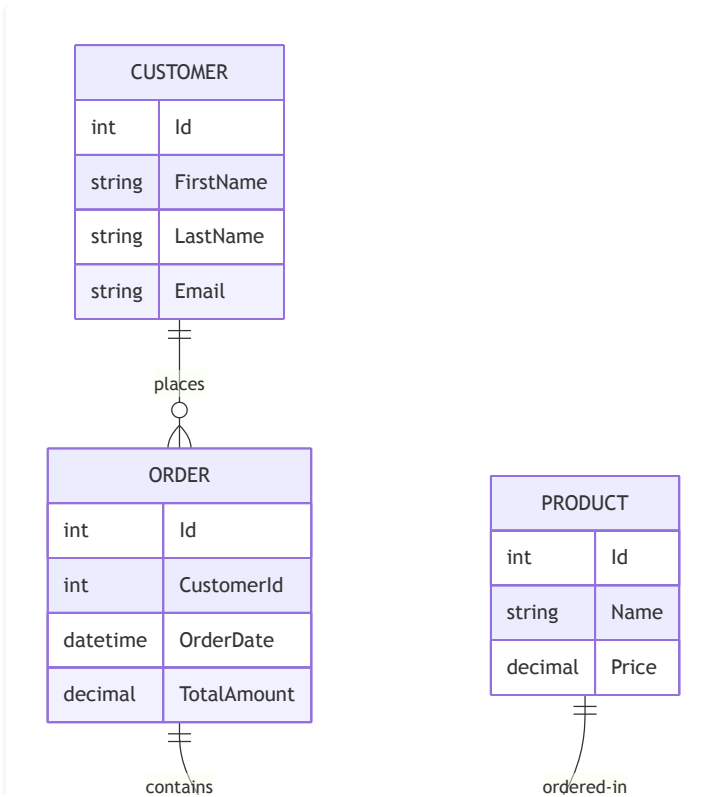
Définition des relations entre entités

Types de relations

- One-to-One (1:1)
- One-to-Many (1:N)
- Many-to-Many (N:M)

Convention vs Configuration

- EF Core peut déduire certaines relations
- La configuration explicite est recommandée
- Importance des propriétés de navigation



TP : Modélisation de données pour une application

Objectifs

1. Créer un modèle de domaine e-commerce
2. Définir les entités et leurs relations
3. Configurer le modèle avec Fluent API
4. Ajouter des validations et contraintes

Entités à créer

- Product (Produit)
- Category (Catégorie)
- Customer (Client)
- Order (Commande)
- OrderItem (Ligne de commande)

Consignes

- Utilisez des noms en anglais pour la compatibilité
- Combinez les Data Annotations et la Fluent API
- Pensez aux contraintes métier (prix > 0, etc.)

Context et DbSet

DbContext

- Représente une session avec la base de données
- Gère les connexions, transactions, et cache
- Convertit les requêtes LINQ en SQL
- Suit les changements des entités

DbSet<T>

- Représente une table ou vue dans la base
- Point d'entrée pour les requêtes
- Permet d'ajouter/supprimer des entités

```
public class ApplicationDbContext : DbContext
{
    // DbSets - un par entité/table
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderItem> OrderItems { get; set; }
    public DbSet<Category> Categories { get; set; }

    // Constructeur pour l'injection de dépendances
    public ApplicationDbContext(
        DbContextOptions<ApplicationDbContext> optio
        : base(options) { }

    // Configuration du modèle
    protected override void OnModelCreating(ModelBui
    {
        base.OnModelCreating(modelBuilder);

        // Appliquer les configurations
        modelBuilder.ApplyConfigurationsFromAssembly
            typeof(ApplicationDbContext).Assembly);
```

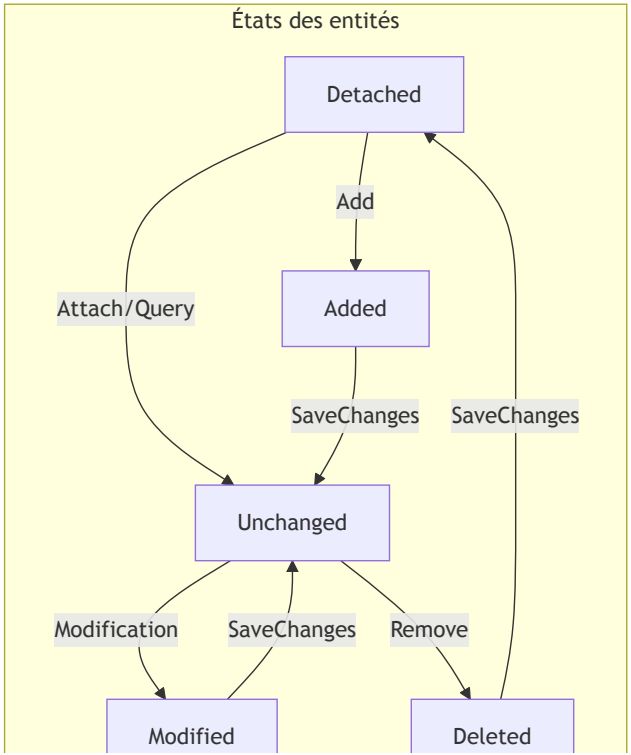
Cycles de vie du contexte

Scénarios d'utilisation

- Web: un contexte par requête HTTP
- Desktop: contexte à vie plus longue
- Services: durée variable selon les cas

Gestion du suivi des entités

- Attached: entités suivies par le contexte
- Detached: entités non suivies
- Added/Modified/Deleted: états des entités
- Unchanged: état initial après chargement



Configuration par convention vs. configuration explicite

Conventions par défaut

- `Id` ou `<type>Id` comme clé primaire
- Clés étrangères basées sur les navigations
- Pluralisation des noms de tables
- Mapping des types .NET vers SQL
- Nullable basé sur les types C#

Configuration explicite

```
// Dans une classe de configuration (approche rec
public class CustomerConfiguration : IEntityTypeC
{
    public void Configure(EntityTypeBuilder<Custc
    {
        builder.ToTable("Customers");

        builder.HasKey(c => c.Id);

        builder.Property(c => c.FirstName)
               .IsRequired()
               .HasMaxLength(50);

        builder.Property(c => c.LastName)
               .IsRequired()
               .HasMaxLength(50);

        builder.Property(c => c.Email)
```

Avantages des conventions

Entity Framework Core

Session Après-midi - Jour 1

Modélisation de données avancée et DbContext

Appuyez sur Espace pour commencer →

Planning de l’après-midi

Session 3 (13h30-15h30)

Modélisation de données avec EF Core

- Modélisation avancée d’entités
- Types de propriétés et conversions
- Shadow properties
- Utilisation des index et contraintes
- Définition de relations complexes

Session 4 (15h45-17h30)

Context et DbSet

- Configuration avancée du DbContext
- Options de DbContext
- Intercepteurs et comportements
- Seed de données
- Travail pratique dirigé

Types de relations

- One-to-One
  - Principal-Dépendant
  - Clé étrangère et propriété de navigation
- One-to-Many
  - Collection d'un côté, référence de l'autre
- Many-to-Many
  - Table de jointure implicite
  - Table de jointure explicite

Types de propriétés et conversions

Types supportés nativement

- Types primitifs (.NET)
- string, decimal, DateTime, Guid
- byte, Enum
- Types nullables (int?, bool?, etc.)

Propriétés complexes

- Types de valeur
- Owned entities
- Collections de types primitifs
- JSON sérialisé

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }

    // Type complexe (owned entity)
    public Address HomeAddress { get; set; }

    // Collection de types primitifs
    public List<string> PhoneNumbers { get; set;

    // Type Enum
    public CustomerStatus Status { get; set; }

    // Propriété avec conversion
    [Column(TypeName = "jsonb")] // Pour PostgreS
    public Dictionary<string, string> Preferences
}

public enum CustomerStatus
{
    Regular,
```

Conversions de valeurs

Conversions automatiques

- Types numériques compatibles
- Chaînes et types numériques
- Enums et valeurs numériques
- DateTime et DateTimeOffset

Conversions personnalisées

- ValueConverter
- Fonctions de conversion
- Types non supportés nativement

```
protected override void OnModelCreating(ModelBuilder
{
    // Conversion d'enum en string
    modelBuilder.Entity<Customer>()
        .Property(c => c.Status)
        .HasConversion<string>();

    // Conversion avec ValueConverter
    modelBuilder.Entity<Product>()
        .Property(p => p.Tags)
        .HasConversion(
            v => string.Join(',', v),
            v => v.Split(',', StringSplitOptions.Rem
                .ToList()
        );

    // Conversion avec fonctions personnalisées
    modelBuilder.Entity<User>()
        .Property(u => u.Password)
        .HasConversion(
            // Stockage: hash du mot de passe
            v => ComputeHash(v),
```

Shadow Properties

Définition

Propriétés qui n'existent pas dans la classe .NET mais sont mappées sur la base de données.

Cas d'utilisation

- Colonnes techniques (audit, traçabilité)
- Mappings sur bases existantes
- Séparation des préoccupations

Avantages

- Classe d'entité plus propre

```
// Configuration de shadow properties
modelBuilder.Entity<BlogPost>()
    .Property<DateTime>("CreatedAt");

modelBuilder.Entity<BlogPost>()
    .Property<DateTime>("LastModified");

modelBuilder.Entity<BlogPost>()
    .Property<string>("CreatedBy");

modelBuilder.Entity<BlogPost>()
    .Property<string>("LastModifiedBy");

// Utilisation dans les requêtes
var posts = context.BlogPosts
    .Where(p =>
        EF.Property<DateTime>(p, "CreatedAt") > somel
    .ToList();

// Définition lors de l'ajout
context.Entry(post)
    .Property("CreatedAt").CurrentValue = DateTime.Ut
context.Entry(post)
```

Définition de relations complexes

Relation One-to-One

```
// Fluent API - approche recommandée
modelBuilder.Entity<User>()
    .HasOne(u => u.Profile)
    .WithOne(p => p.User)
    .HasForeignKey<Profile>(p => p.UserId);
```

Relation One-to-Many

```
modelBuilder.Entity<Blog>()
    .HasMany(b => b.Posts)
    .WithOne(p => p.Blog)
    .HasForeignKey(p => p.BlogId)
    .OnDelete(DeleteBehavior.Restrict);
```

Relation Many-to-Many avec table de jointure personnalisée

```
// Entité de jointure explicite
public class CourseStudent
{
    public int CourseId { get; set; }
    public Course Course { get; set; }

    public int StudentId { get; set; }
    public Student Student { get; set; }

    // Propriétés supplémentaires
    public DateTime EnrollmentDate { get; set; }
    public Grade? Grade { get; set; }
}

// Configuration
modelBuilder.Entity<CourseStudent>()
```

Many-to-Many simplifié (EF Core 5+)

Sans table de jointure explicite

```
// Modèle
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<Course> Courses { get; set; }
}

public class Course
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<Student> Students { get; set; }
}

// Configuration
modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
```

Utilisation des index et contraintes

Index

- Amélioration des performances
- Index simples ou composés
- Index uniques
- Filtrage d'index (SQL Server)

Contraintes

- Contraintes d'unicité
- Contraintes de vérification (check)
- Contraintes par défaut

```
protected override void OnModelCreating(ModelBuil
{
    // Index simple
    modelBuilder.Entity<Customer>()
        .HasIndex(c => c.Email);

    // Index unique
    modelBuilder.Entity<Customer>()
        .HasIndex(c => c.Username)
        .IsUnique();

    // Index composé
    modelBuilder.Entity<Order>()
        .HasIndex(o => new { o.CustomerId, o.Orde

    // Index filtré (SQL Server)
    modelBuilder.Entity<Product>()
        .HasIndex(p => p.Name)
        .HasFilter("[IsDeleted] = 0")
        .HasDatabaseName("IX_Product_Name_NotDele

    // Contrainte de vérification
```

TP : Modélisation avancée pour une application e-commerce

Objectifs

1. Enrichir le modèle de données créé le matin
2. Ajouter des propriétés complexes
3. Configurer des relations avancées
4. Utiliser des index et des contraintes

Nouvelles fonctionnalités à modéliser

- Adresses (livraison, facturation)
- Système d'évaluation des produits
- Gestion des stocks
- Historique des prix
- Catégorisation hiérarchique

Exemple de modélisation pour les adresses

```
// Utilisation d'une owned entity
public class Customer
{
```

Pause café ☕

Rendez-vous à 15h45 pour la dernière session de la journée

Configuration avancée du DbContext

Options de configuration

- Connexion à la base de données
- Logging et journalisation
- Suivi des entités
- Timeout des commandes
- Lazy loading

Méthodes de configuration

- OnConfiguring
- Options dans le constructeur
- Extension methods

```
protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder
            .UseSqlServer(@"Server=(localdb)\mssqllo
            .EnableSensitiveDataLogging()
            .LogTo(Console.WriteLine, LogLevel.Infor
            .EnableDetailedErrors();
    }
}

// Ou via injection de dépendances
services.AddDbContext<ApplicationDbContext>(options
    options.UseSqlServer(
        Configuration.GetConnectionString("DefaultCo
);

// Options avancées
services.AddDbContext<ApplicationDbContext>(options
    options.UseSqlServer(
```

Intercepteurs et comportements

Types d'intercepteurs

- Command interceptors
- Connection interceptors
- Transaction interceptors
- SaveChanges interceptors

Cas d'utilisations

- Audit automatique
- Mesure de performances
- Journalisation avancée
- Modification dynamique des requêtes

```
// Intercepteur de commandes SQL
public class CommandInterceptor : DbCommandIntercepto
{
    private readonly ILogger<CommandInterceptor> _log

    public CommandInterceptor(ILogger<CommandIntercep
    {
        _logger = logger;
    }

    public override InterceptionResult<DbDataReader>
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult<DbDataReader> result)
    {
        _logger.LogInformation(
            "Executing command: {CommandText}",
            command.CommandText);

        return result;
    }

    public override ValueTask<InterceptionResult<DbDa
```

SaveChanges et suivi des entités

SaveChanges

- Mécanisme de persistance
- Transaction implicite
- Validation des contraintes
- Retour du nombre d'entités modifiées

Suivi des entités

- ChangeTracker
- États des entités
  - Added
  - Modified
  - Unchanged
  - Deleted
  - Detached

Personnalisation

- Override de SaveChanges
- Validation personnalisée
- Modifications avant persistance
- Audit automatique

AsNoTracking

- Amélioration des performances
- Requêtes en lecture seule
- Économie de mémoire
- Pas de mise à jour possible

Seed de données

Types de seed

- Seed de développement
- Seed de test
- Seed de production
- Seed de démonstration

Méthodes

- OnModelCreating (données de référence)
- Extensions de ModelBuilder
- Migrations (données évolutives)
- Seed programmatique

```
protected override void OnModelCreating(ModelBuilder
{
    base.OnModelCreating(modelBuilder);

    // Seed de données de référence
    modelBuilder.Entity<Category>().HasData(
        new Category { Id = 1, Name = "Électronique"
        new Category { Id = 2, Name = "Vêtements" },
        new Category { Id = 3, Name = "Alimentation"
    );

    // Seed avec relations
    modelBuilder.Entity<Product>().HasData(
        new Product {
            Id = 1,
            Name = "Smartphone",
            Price = 699.99m,
            CategoryId = 1
        },
        new Product {
            Id = 2,
            Name = "Laptop",
```

TP : Implémentation d'un DbContext personnalisé

Objectifs

1. Configurer un DbContext avancé
2. Implémenter l'audit automatique
3. Configurer le seed de données
4. Gérer les conventions personnalisées

Audit automatique

```
// Exemple à implémenter
public override int SaveChanges(bool acceptAllChange
{
    AuditEntities();
    return base.SaveChanges(acceptAllChangesOnSucces
}

public override Task<int> SaveChangesAsync(
    bool acceptAllChangesOnSuccess,
    CancellationToken cancellationToken = default)
{
    AuditEntities();
    return base.SaveChangesAsync(
        acceptAllChangesOnSuccess,
        cancellationToken);
}

private void AuditEntities()
```

Conventions personnalisées

Types de conventions

- Conventions intégrées
- Conventions personnalisées
- Conventions appliquées à l'ensemble du modèle

Exemples

- Pluralisation des noms de tables
- Nommage personnalisé des clés étrangères
- Soft delete global
- Filtres de requête globaux

```
// Convention personnalisée
public class SoftDeleteConvention : IModelFinaliz
{
    public void ProcessModelFinalizing(
        IConventionModelBuilder modelBuilder,
        IConventionContext<IConventionModelBuilde
    {
        foreach (var entityType in modelBuilder.M
        {
            // Check if entity implements ISoftDe
            if (typeof(ISoftDelete).IsAssignableF
            {
                // Add query filter
                var parameter = Expression.Parame
                var property = Expression.Propert
                    parameter, nameof(ISoftDelete
                var condition = Expression.Equal(
                    property, Expression.Constant
                var lambda = Expression.Lambda(
                    condition, parameter);

                entityType.SetQueryFilter(lambda)
```

Récapitulatif de la journée

Matin

- Introduction aux ORM
- Avantages et inconvénients des ORM
- Histoire et évolution d'EF Core
- Configuration de l'environnement
- Création d'un premier projet

Après-midi

- Modélisation avancée d'entités
- Relations entre entités
- Configuration du DbContext
- Intercepteurs et comportements
- Seed de données

À faire pour demain

- Revoir les concepts clés
- Explorer la documentation officielle
- Terminer les exercices du jour
- Préparer des questions sur les migrations pour demain

Preview de demain

Session 1: Migrations de base de données

- Concept de migrations
- Commandes de base
- Personnalisation des migrations
- Stratégies de déploiement

Session 2 & 3: Opérations CRUD

- Lecture de données
- Méthodes synchrones et asynchrones
- LINQ to Entities
- Création, mise à jour et suppression
- Gestion des transactions
- Suivi des changements

Session 4: Requêtes avancées

- Requêtes avec jointures
- Projections et sélections personnalisées

Questions?

À demain pour la suite du cours !

Merci pour votre attention et votre participation!

Entity Framework Core

Développement d'applications modernes

Introduction à LINQ

Language Integrated Query

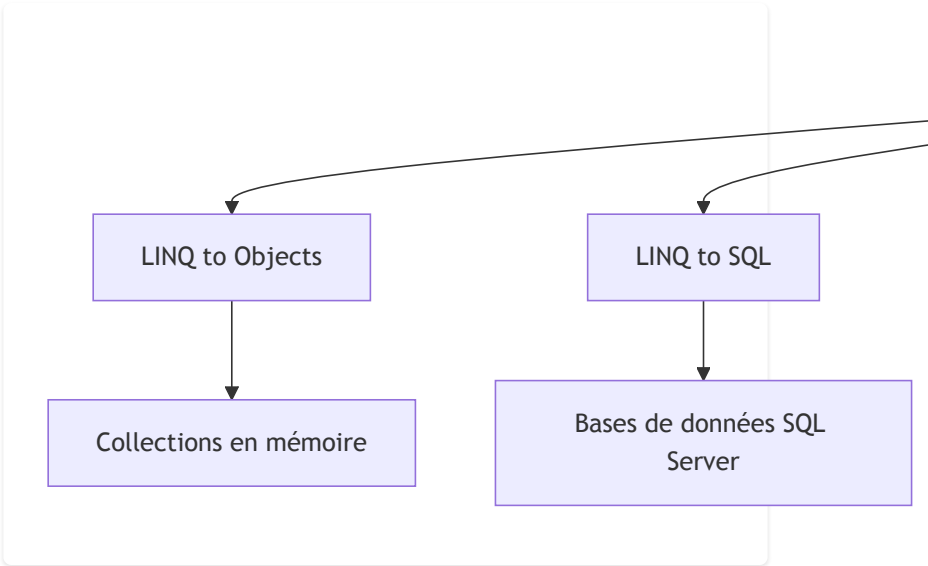
Fondamentaux, contextes d’utilisation et exemples pratiques

Appuyez sur Espace pour commencer →

Qu’est-ce que LINQ ?

Définition

- Language Integrated Query
- Introduit dans C# 3.0 / .NET 3.5 (2007)
- Unifie l’interrogation de différentes sources de données
- Syntaxe déclarative et fortement typée
- Alternative aux requêtes SQL, XPath, etc.



Fondamentaux

- Basé sur des expressions lambda
- Exécution différée (lazy evaluation)
- Fortement typé et vérifié à la compilation

Deux syntaxes, même résultat

Syntaxe de méthodes

```
var result = collection
    .Where(item => item.Price > 100)
    .OrderBy(item => item.Name)
    .Select(item => new {
        item.Name,
        item.Price
    });
```

Avantages

- Plus concise pour les requêtes complexes
- Chaînage fluide des opérations
- Support complet d'IntelliSense
- Utilisable partout en C#

Syntaxe de requêtes

```
var result =
    from item in collection
    where item.Price > 100
    orderby item.Name
    select new {
        item.Name,
        item.Price
    };
```

Avantages

- Plus lisible pour les débutants
- Similaire à SQL (familier)
- Clarté pour certaines opérations
- Meilleure pour les jointures complexes

Sources de données compatibles

Types de collections

- Tableaux (`T[]`)
- Listes (`List<T>`)
- Dictionnaires (`Dictionary<K,V>`)
- Ensembles (`HashSet<T>`)
- Files (`Queue<T>`)
- Piles (`Stack<T>`)
- Tout type implémentant `IEnumerable<T>`

Entity Framework

- DbSet<T>
- IQueryable<T>
- Requêtes traduites en SQL
- Exécution côté serveur DB

Autres sources

- XML (XElement, XDocument)
- DataSet et DataTable
- Fichiers et répertoires
- Événements et observables (Rx)

Cas particuliers

- Types anonymes
- Types dynamiques
- Tuples
- Records (C# 9+)
- Collections immuables
- Span<T> et Memory<T> (partiellement)
- Async Enumerable (IAsyncEnumerable<T>)

Opérations LINQ fondamentales

Filtrage

- `Where` - Filtrer sur un prédicat
- `OfType` - Filtrer par type
- `Take` - Prendre N éléments
- `Skip` - Sauter N éléments
- `TakeWhile` / `SkipWhile` - Conditionnels
- `Distinct` - Supprimer les doublons

Projection

- `Select` - Transformer les éléments
- `SelectMany` - Aplatir des sous-collections
- `Cast` - Conversion de type

Tri

- `OrderBy` / `OrderByDescending` - Tri simple
- `ThenBy` / `ThenByDescending` - Tri multiple
- `Reverse` - Inverser l'ordre

Aggrégation

- `Count` / `LongCount` - Nombre d'éléments
- `Sum` , `Min` , `Max` , `Average` - Calculs
- `Aggregate` - Agrégation personnalisée

Exemples concrets : LINQ to Objects

Filtrage et projection

```
// Données d'exemple
var products = new List<Product>
{
    new Product { Id = 1, Name = "Laptop", Price = 12
    new Product { Id = 2, Name = "Mouse", Price = 25,
    new Product { Id = 3, Name = "Desk", Price = 300,
    new Product { Id = 4, Name = "Chair", Price = 150
    new Product { Id = 5, Name = "Monitor", Price = 4
};

// Produits informatiques (catégorie 1) coûtant plus
var expensiveItProducts = products
    .Where(p => p.CategoryId == 1 && p.Price > 100)
    .Select(p => new {
        ProductName = p.Name,
        Price = $"{p.Price}€"
    });

foreach (var item in expensiveItProducts)
{
    Console.WriteLine($"{item.ProductName}: {item.Pri
```

Tri et agrégation

```
// Statistiques par catégorie
var categoryStats = products
    .GroupBy(p => p.CategoryId)
    .Select(g => new
    {
        CategoryId = g.Key,
        ProductCount = g.Count(),
        TotalValue = g.Sum(p => p.Price),
        AveragePrice = g.Average(p => p.Price),
        MostExpensive = g.OrderByDescending(p => p.Pr
    });

foreach (var stat in categoryStats)
{
    Console.WriteLine($"Catégorie {stat.CategoryId}:'
    Console.WriteLine($"  Nombre de produits: {stat.P
    Console.WriteLine($"  Valeur totale: {stat.Total\
    Console.WriteLine($"  Prix moyen: {stat.AveragePr
    Console.WriteLine($"  Produit le plus cher: {stat
}
```

Jointures et groupements

Types de jointures

- Join (inner join)
- GroupJoin (left join avec groupement)
- Zip (combiner élément par élément)

Syntaxe de méthode

```
var query = customers.Join(
    orders,
    customer => customer.Id,
    order => order.CustomerId,
    (customer, order) => new {
        customer.Name,
        order.OrderDate,
        order.TotalAmount
    }
);
```

Syntaxe de requête

```
var query =
    from c in customers
    join o in orders
    on c.Id equals o.CustomerId
    select new {
        c.Name,
        o.OrderDate,
        o.TotalAmount
    };
```

Left Join (syntaxe de requête)

```
var query =
    from c in customers
    join o in orders
    on c.Id equals o.CustomerId into customerOrders
    from o in customerOrders.DefaultIfEmpty()
```

LINQ to Entities avec Entity Framework

Spécificités

- Traduit en SQL
- Évaluation côté serveur
- Lazy loading vs Eager loading
- Utilise `IQueryable<T>` (pas `IEnumerable<T>`)
- Types de retour différents

Include et ThenInclude

```
// Chargement de relations
var customers = context.Customers
    .Include(c => c.Orders)
        .ThenInclude(o => o.OrderItems)
```

Requête EF Core

```
// Exemple de requête complexe avec EF Core
var topSellingProducts = await context.Products
    .Where(p => p.IsActive)
    .Select(p => new
    {
        p.Id,
        p.Name,
        p.Price,
        CategoryName = p.Category.Name,
        TotalSold = p.OrderItems.Sum(oi => oi.Qua
        Revenue = p.OrderItems.Sum(oi =>
            oi.Quantity * oi.UnitPrice)
    })
    .OrderByDescending(p => p.TotalSold)
    .Take(10)
    .ToListAsync();
```

Execution différée et immédiate

Exécution différée (lazy)

- La requête n'est pas exécutée immédiatement
- Exécutée uniquement quand les résultats sont nécessaires
- Permet l'optimisation automatique
- Peut être étendue après définition

```
// Pas encore exécuté !
var query = products.Where(p => p.Price > 100);

// Ajout de conditions
query = query.OrderBy(p => p.Name);

// Exécution uniquement ici
foreach (var product in query)
{
```

Exécution immédiate

- Force l'évaluation immédiate de la requête
- Transforme `IQueryable` en `IEnumerable` ou autre
- Utile pour mettre en cache les résultats

```
// Méthodes forçant l'exécution immédiate
var list = query.ToList();          // Liste
var array = query.ToArray();        // Tableau
var dict = query.ToDictionary(p => p.Id); // Diction
var lookup = query.ToLookup(p => p.CategoryId); // L

// Opérateurs d'agrégation
var count = query.Count();
var sum = query.Sum(p => p.Price);
var any = query.Any(p => p.Stock > 0);

// Opérateurs d'élément
```

Requêtes plus complexes et cas d'utilisation

Partitionnement et pagination

```
// Pagination simple
var pageSize = 10;
var pageNumber = 2; // Page 2 (basée sur 1)

var pageItems = products
    .OrderBy(p => p.Name)
    .Skip((pageNumber - 1) * pageSize)
    .Take(pageSize)
    .ToList();
```

Opérations ensemblistes

```
// Union, Intersection, Difference
var techProducts = products.Where(p => p.CategoryId
var expensiveProducts = products.Where(p => p.Price

var techOrExpensive = techProducts.Union(expensivePr
```

Requêtes imbriquées

```
// Produits dont le prix est supérieur à la moyenne
var averagePrice = products.Average(p => p.Price);
var aboveAverageProducts = products
    .Where(p => p.Price > averagePrice)
    .OrderBy(p => p.Price)
    .ToList();

// Version en une seule requête
var aboveAverage = products
    .Where(p => p.Price > products.Average(x => x.Pri
    .OrderBy(p => p.Price)
    .ToList();

// Clients ayant commandé au moins un produit de la c
var customersWithTechProducts = context.Customers
    .Where(c => c.Orders.Any(o => o.OrderItems
        .Any(oi => oi.Product.CategoryId == 1)))
    .ToList();
```

Types de données et cas particuliers

Collections spéciales

- ImmutableList<T> ,

  ImmutableArray<T>
- ReadOnlyCollection<T>
- ConcurrentBag<T> ,

  ConcurrentQueue<T>
- ObservableCollection<T>

```
// Retourne une collection en lecture seule
var readOnly = products
    .Where(p => p.Price > 100)
    .Select(p => p.Name)
    .ToList()
    .AsReadOnly();
```

Types anonymes

Tuples et Records

```
// Tuples
var stats = products
    .Select(p => (Name: p.Name, Price: p.Price))
    .OrderBy(x => x.Price)
    .ToList();

// Deconstruction
foreach (var (name, price) in stats)
{
    Console.WriteLine($"{name}: {price}€");
}

// Records (C# 9+)
record ProductSummary(string Name, decimal Price)

var summaries = products
    .Select(p => new ProductSummary(p.Name, p.Pri
    .ToList();
```

Dynamic LINQ (via

LINQ asynchrone et parallèle

Méthodes asynchrones (EF Core)

```
// Versions asynchrones pour Entity Framework
var products = await context.Products
    .Where(p => p.Price > 100)
    .ToListAsync();

var count = await context.Products.CountAsync();
var first = await context.Products.FirstOrDefaultAsy
```

IAsyncEnumerable (C# 8+)

```
// Génération asynchrone des résultats
async IAsyncEnumerable<Product> GetProductsAsync()
{
    await foreach (var product in context.Products.A
    {
        // Traitement potentiellement lourd
        yield return product;
```

PLINQ (Parallel LINQ)

```
// Traitement parallèle
var result = products
    .AsParallel()
    .Where(p => IsExpensive(p)) // Fonction coûteuse
    .OrderBy(p => p.Name)
    .Take(10)
    .ToList();

// Avec options
var result = products
    .AsParallel()
    .WithDegreeOfParallelism(4)
    .WithExecutionMode(ParallelExecutionMode.ForcePa
    .Where(p => IsExpensive(p))
    .ToList();
```

Différences importantes

- PLINQ uniquement pour LINQ to Objects

Performance et optimisation

Bonnes pratiques

- Filtrer avant de projeter
- Éviter les évaluations multiples
- Être attentif à l'exécution différée
- Utiliser AsNoTracking() avec EF Core
- Éviter les requêtes N+1

Déboguer les requêtes

```
// Voir la requête générée (EF Core)
var query = context.Products
    .Where(p => p.Price > 100)
    .Include(p => p.Category);

Console.WriteLine(query.ToQueryString());
```

Optimisations spécifiques

```
// Projection immédiate pour éviter le chargement co
var names = context.Products
    .Where(p => p.CategoryId == 1)
    .Select(p => p.Name)
    .ToList();

// Requête compilée pour réutilisation
private static readonly Func<AppDbContext, int, Task
    EF.CompileAsyncQuery((AppDbContext ctx, int id)
        ctx.Products.Include(p => p.Category)
                    .FirstOrDefault(p => p.Id == id)

// Utilisation
var product = await GetProductById(context, 5);
```

Pagination optimisée

```
// Keyset Pagination (plus efficace que Skip/Take)
public async Task<List<Product>> GetProductsPage(
```

TP : Utilisation avancée de LINQ

Objectifs

1. Créer des requêtes LINQ complexes
2. Appliquer différentes opérations LINQ
3. Optimiser les requêtes existantes
4. Combiner LINQ avec EF Core

Scénarios à implémenter

1. Analyse de données commerciales
2. Rapport de ventes multi-niveaux
3. Recherche et filtrage flexibles
4. Export de données avec projection
5. Analyse statistique d'un jeu de données

Exemple de scénario

```
// Analyse de ventes avec multiples dimensions
public class SalesAnalyzer
{
    private readonly ApplicationDbContext _context;

    public SalesAnalyzer(ApplicationDbContext context)
    {
```

Extensions et bibliothèques LINQ

Bibliothèques populaires

- **System.Linq.Dynamic.Core**
  - Requêtes LINQ dynamiques (par chaînes)
- **MoreLINQ**
  - Extensions LINQ supplémentaires
- **EntityFramework.DynamicFilters**
  - Filtres globaux pour EF Core
- **Z.EntityFramework.Plus**
  - Fonctionnalités avancées pour EF Core
- **Mapster/AutoMapper**

Exemples d'utilisation

```
// Dynamic LINQ
var result = context.Products
    .Where("Price > @0 && CategoryId == @1", 100,
    .OrderBy("Name")
    .Select("new(Id, Name as ProductName, Price)"
    .ToDynamicList();

// MoreLINQ
var batches = products
    .Batch(10) // Grouper par lots de 10
    .Select(batch => ProcessBatch(batch))
    .ToList();

// Z.EntityFramework.Plus
var deleted = context.Products
    .Where(p => p.Discontinued)
    .DeleteFromQuery(); // Suppression sans charg

var updated = context.Products
    .Where(p => p.Price < 10)
```

Récapitulatif

Points clés

- LINQ unifie l'accès aux données
- Syntaxe déclarative et fortement typée
- Adapté à de multiples sources de données
- Execution différée et optimisée
- Extensible pour des besoins spécifiques

Avantages de LINQ

- Code plus concis et lisible
- Moins d'erreurs (typage fort)
- Meilleure réutilisation
- Abstraction de la source de données
- Support de l'IDE et du compilateur
- Requêtes dynamiques possibles

Pour aller plus loin

- Explorez les bibliothèques d'extension
- Appropriez-vous les techniques d'optimisation
- Combinez LINQ avec d'autres fonctionnalités de C#
- Expérimentez avec Expressions Trees

Questions?

Merci pour votre attention !

Documentation

- Microsoft LINQ Documentation
- 101 LINQ Samples
- EF Core Documentation

Bibliothèques

- Dynamic LINQ
- MoreLINQ
- Z.EntityFramework.Plus

Livres et cours

- "LINQ in Action" par Fabrice Marguerie
- "C# in Depth" par Jon Skeet
- Pluralsight: "LINQ Fundamentals"
- Microsoft Learn: ".NET LINQ"

Entity Framework Core

Jour 2 - Mardi Matin

Migrations et opérations CRUD

Appuyez sur Espace pour commencer →

Programme du jour 2

Matin

- Session 1 (9h00-10h30)
  - Migrations de base de données
  - Concept et fonctionnement des migrations
  - Commandes de base pour les migrations
  - Personnalisation des migrations
- Session 2 (10h45-12h45)
  - Opérations CRUD - Lecture de données
  - Méthodes synchrones et asynchrones
  - LINQ to Entities
  - Requêtes de base

Après-midi

- Session 3 (13h30-15h30)
  - Opérations CRUD - Create, Update, Delete
  - Gestion des transactions
  - Suivi des changements (Change Tracking)
- Session 4 (15h45-17h30)
  - Requêtes avancées
  - Requêtes avec jointures
  - Projections et sélections personnalisées
  - Groupement et pagination

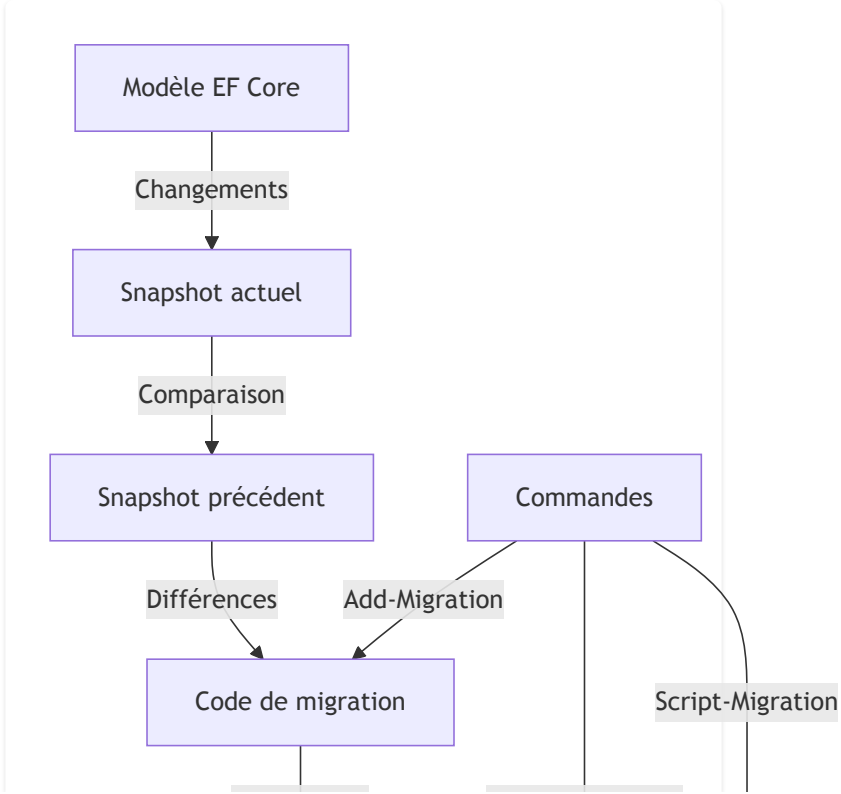
Après-midi

- Modélisation de données
- Types de propriétés et conversions
- Shadow properties
- Relations entre entités
- Configuration du DbContext
- Seed de données

Concept de migrations

Qu'est-ce qu'une migration?

- Représentation des changements du modèle
- Système de contrôle de version pour le schéma
- Code C# généré automatiquement
- Peut être personnalisé manuellement



Avantages des migrations

Évolution incrémentale du schéma

Commandes de base pour les migrations

Package Manager Console (Visual Studio)

```
# Créer une migration
Add-Migration NomDeLaMigration

# Appliquer les migrations
Update-Database

# Revenir à une migration spécifique
Update-Database NomDeLaMigration

# Générer un script SQL
Script-Migration -From MigrationA -To MigrationB

# Supprimer la dernière migration
Remove-Migration
```

CLI .NET Core

```
# Créer une migration
dotnet ef migrations add NomDeLaMigration

# Appliquer les migrations
dotnet ef database update

# Revenir à une migration spécifique
dotnet ef database update NomDeLaMigration

# Générer un script SQL
dotnet ef migrations script MigrationA MigrationB

# Supprimer la dernière migration
dotnet ef migrations remove
```

Options utiles

Structure d’une migration

Fichiers générés

- **YYYYMMDDHHMMSS_NomMigration.cs**
  - Classe de migration avec Up() et Down()
- **YYYYMMDDHHMMSS_NomMigration.Designe**
  - Métadonnées de la migration
- **ApplicationDbContextModelSnapshot.cs**
  - Snapshot actuel du modèle

Méthodes principales

- **Up()** : Applique les changements
- **Down()** : Annule les changements (rollback)

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migra
    {
        migrationBuilder.CreateTable(
            name: "Customers",
            columns: table => new
            {
                Id = table.Column<int>(nullable: fals
                    .Annotation("SqlServer:Identity",
                Name = table.Column<string>(maxLength
                Email = table.Column<string>(maxLengt
                CreatedAt = table.Column<DateTime>(nu
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Customers", x =>
            });

        migrationBuilder.CreateIndex(
            name: "IX_Customers_Email",
            table: "Customers",
            column: "Email",
```

Personnalisation des migrations

Pourquoi personnaliser?

- Opérations non supportées par EF Core
- Manipulation avancée des données
- Optimisations spécifiques
- Migration de données existantes

Points d'extension

- Édition manuelle des méthodes Up/Down
- Ajout de code SQL personnalisé
- Opérations conditionnelles
- Créer des extensions pour MigrationBuilder

```
protected override void Up(MigrationBuilder migratio
{
    // Code généré automatiquement
    // ...

    // SQL brut pour des opérations complexes
    migrationBuilder.Sql(@"
    CREATE PROCEDURE GetTopCustomers
        @Count INT
    AS
    BEGIN
        SELECT TOP(@Count) * FROM Customers ORDER BY
    END
    ");

    // Migration de données
    migrationBuilder.Sql(@"
    UPDATE Products
    SET Price = Price * 1.1
    WHERE CategoryId = 3
    ");
```

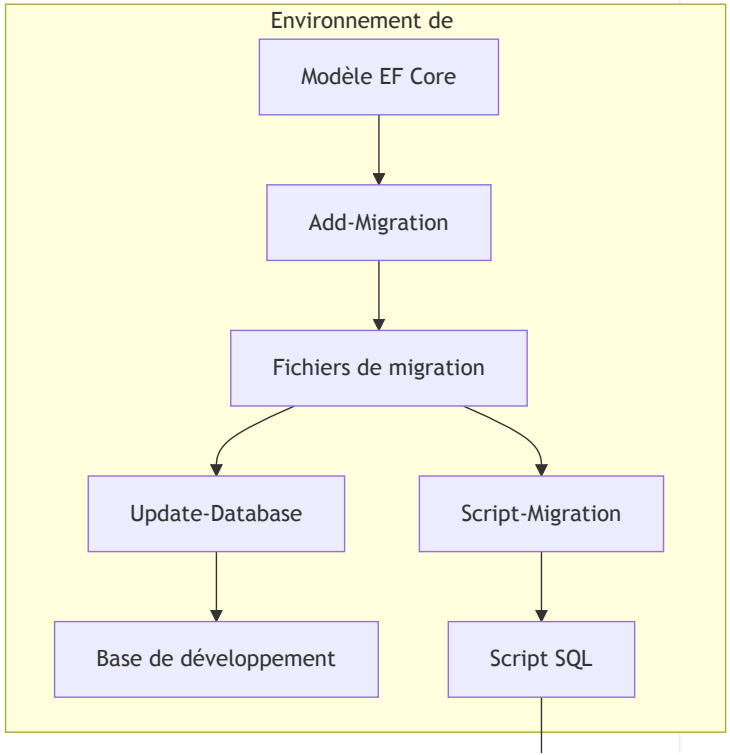
Stratégies de déploiement

Approches de déploiement

- Exécution directe des migrations
- Scripts SQL générés
- Intégration avec DevOps
- Migrations idempotentes

Considérations

- Environnements multiples
- Bases de données de production
- Données existantes
- Rollbacks et récupération



TP : Création et application de migrations

Objectifs

1. Créer une première migration
2. Personnaliser la migration
3. Appliquer la migration
4. Créer une migration pour des modifications
5. Générer un script SQL

Étapes

1. Utilisez `Add-Migration InitialCreate`
2. Examinez les fichiers générés
3. Personnalisez la migration (ajoutez des données initiales)
4. Appliquez avec `Update-Database`
5. Modifiez votre modèle
6. Créez une nouvelle migration
7. Générez un script SQL

Extension pour les plus avancés

- Créez un scénario de rollback
- Implémentez une migration idempotente

Pause café ☕

Rendez-vous à 10h45 pour la suite

Opérations CRUD : Lecture de données

CRUD avec Entity Framework Core

- **C**reate : Ajouter de nouvelles entités
- **R**ead : Lire et interroger des données
- **U**pdate : Mettre à jour des entités existantes
- **D**elete : Supprimer des entités

Lecture de données - Concepts clés

- DbSet comme point d'entrée des requêtes
- LINQ comme langage de requête
- Chargement d'entités et de relations

```
public class ProductService
{
    private readonly ApplicationDbContext _context;
    public ProductService(ApplicationDbContext conte

    // Méthode synchrone - Tous les produits
    public List<Product> GetAllProducts()
    {
        return _context.Products.ToList();
    }

    // Méthode synchrone - Un produit par ID
    public Product GetProductById(int id)
    {
        return _context.Products.Find(id);
    }

    // Méthode asynchrone - Tous les produits
    public async Task<List<Product>> GetAllProductsA
    {
        return await _context.Products
            .AsNoTracking()
```

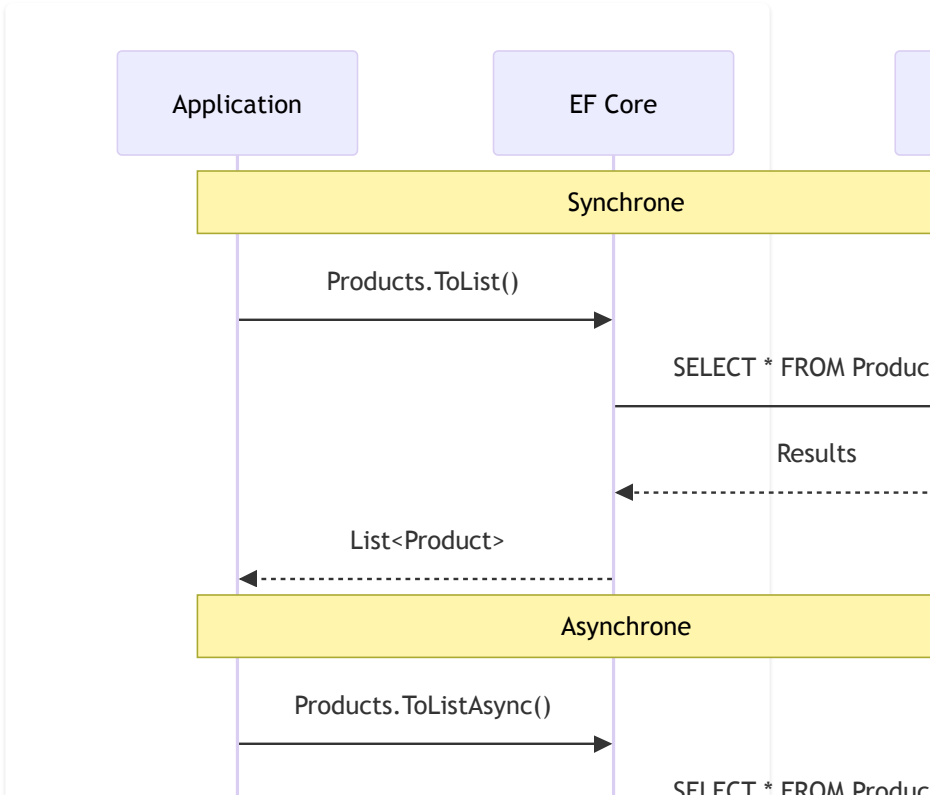
Méthodes synchrones et asynchrones

Méthodes synchrones

- Bloquent le thread pendant l'exécution
- Plus simples à comprendre et à déboguer
- Adéquates pour les applications de bureau
- Exemples: `ToList()`, `First()`, `Single()`

Méthodes asynchrones

- Libèrent le thread pendant l'attente I/O
- Améliorent la capacité de montée en charge
- Recommandées pour les applications web
- Exemples: `ToListAsync()`, `FirstAsync()`



LINQ to Entities

Qu’est-ce que LINQ?

- Language Integrated Query
- Syntaxe unifiée pour interroger diverses sources
- Deux syntaxes: méthodes et requêtes
- Transformé en SQL par EF Core

Avantages de LINQ avec EF Core

- Typage fort et vérification à la compilation
- IntelliSense et refactoring
- Composition de requêtes

Syntaxe de méthodes

```
// Filtrer
var expensiveProducts = context.Products
    .Where(p => p.Price > 100)
    .OrderBy(p => p.Name)
    .ToList();

// Projeter
var productNames = context.Products
    .Select(p => p.Name)
    .ToList();

// Agréger
var averagePrice = context.Products
    .Average(p => p.Price);
```

Syntaxe de requêtes

```
// Équivalent avec la syntaxe de requête
var expensiveProducts = (
```

Opérations LINQ courantes

Filtrage

```
// Filtrage simple
var activeProducts = context.Products
    .Where(p => p.IsActive)
    .ToList();

// Filtrage multiple
var featuredProducts = context.Products
    .Where(p => p.IsActive && p.IsFeatured && p.S
    .ToList();

// Recherche textuelle
var searchResults = context.Products
    .Where(p => p.Name.Contains(searchTerm))
    .ToList();
```

Tri

```
// Tri simple
var sortedProducts = context.Products
```

Pagination

```
var pageSize = 10;
var pageNumber = 2; // Page à base 1

var pagedProducts = context.Products
    .OrderBy(p => p.Name)
    .Skip((pageNumber - 1) * pageSize)
    .Take(pageSize)
    .ToList();
```

Agrégation

```
// Compter
var productCount = context.Products.Count();

// Somme
var totalValue = context.Products.Sum(p => p.Pric

// Moyenne, Min, Max
var averagePrice = context.Products.Average(p =>
```

Requêtes de base et bonnes pratiques

Techniques essentielles

- Utiliser des conditions de filtrage efficaces
- Charger uniquement les données nécessaires
- Projeter vers des DTOs quand approprié
- Éviter les requêtes N+1

Bonnes pratiques

- Préférer `FirstOrDefault` à `First` (exception)
- Utiliser `AsNoTracking` pour les lectures seules
- Composer les requêtes avant exécution

```
// ❌ Problème N+1 - Une requête par catégorie
var products = context.Products.ToList();
foreach (var product in products)
{
    // Une requête SQL à chaque accès
    Console.WriteLine(product.Category.Name);
}

// ✅ Eager Loading - Une seule requête
var productsWithCategories = context.Products
    .Include(p => p.Category)
    .ToList();
foreach (var product in productsWithCategories)
{
    // Pas de requête SQL supplémentaire
    Console.WriteLine(product.Category.Name);
}

// ✅ Projection vers DTO - Données minimales
var productDtos = context.Products
    .Select(p => new ProductDto
    {
```

TP : Implémentation de différentes requêtes de lecture

Objectifs

1. Implémenter diverses requêtes LINQ
2. Utiliser les méthodes synchrones et asynchrones
3. Optimiser des requêtes problématiques
4. Projeter vers des DTOs
5. Implémenter la pagination

Exemple de requête à implémenter

```
// Recherche de produits avec filtres multiples
public async Task<List<ProductDto>> SearchProductsAs
    string searchTerm,
    decimal? minPrice,
    decimal? maxPrice,
    int? categoryId,
    int page = 1,
    int pageSize = 10)
{
    var query = _context.Products.AsQueryable();

    if (!string.IsNullOrEmpty(searchTerm))
        query = query.Where(p => p.Name.Contains(sea
                                 p.Description.Contains

    if (minPrice.HasValue)
```

Récapitulatif de la matinée

Migrations

- Concept de migrations
- Commandes principales
- Personnalisation
- Stratégies de déploiement

Lecture de données (CRUD - Read)

- Méthodes synchrones et asynchrones
- LINQ to Entities
- Opérations de base
- Bonnes pratiques

Après-midi

- Opérations CRUD (Create, Update, Delete)
- Gestion des transactions
- Suivi des changements
- Requêtes avancées

Bon appétit! 🍽️

Rendez-vous à 13h30 pour la suite

Entity Framework Core

Jour 2 - Mardi Après-midi

Opérations CRUD et Requêtes avancées

Appuyez sur Espace pour commencer →

Programme de l’après-midi

Session 3 (13h30-15h30)

Opérations CRUD - Partie 2

- Création de données (Create)
- Mise à jour de données (Update)
- Suppression de données (Delete)
- Gestion des transactions
- Suivi des changements (Change Tracking)

Session 4 (15h45-17h30)

Requêtes avancées

- Requêtes avec jointures complexes
- Projections et sélections personnalisées
- Requêtes avec groupement
- Pagination des résultats
- Techniques d’optimisation

Opérations CRUD - Create

Ajouter une entité

```
var product = new Product
{
    Name = "New Product",
    Price = 29.99m,
    CategoryId = 1
}

</div>

<!--
Notes pour l'enseignant:
- Donnez aux étudiants un document avec les spécific
- Encouragez différentes approches pour les résoudre
- Discutez des compromis entre lisibilité et perform
- Prévoyez environ 60 minutes pour cet exercice
-->

---
```

Ajouter plusieurs entités

```
var products = new List<Product>
{
    new Product { Name = "Product 1", Price = 19.99m
    new Product { Name = "Product 2", Price = 29.99m
    new Product { Name = "Product 3", Price = 39.99m
};

// Ajouter plusieurs entités
context.Products.AddRange(products);

// Persister en une seule transaction
context.SaveChanges();
```

Méthodes asynchrones

```
await context.Products.AddAsync(product);
await context.SaveChangesAsync();
```

Opérations CRUD - Update

Mise à jour d'une entité suivie

```
// Récupérer l'entité
var product = context.Products.Find(1);

// Modifier les propriétés
product.Name = "Updated Name";
product.Price = 39.99m;

// Persister les changements
context.SaveChanges();
```

États des entités

- `EntityState.Added` : Nouvelle entité
- `EntityState.Modified` : Entité modifiée
- `EntityState.Unchanged` : Entité non

Mise à jour d'une entité non suivie

```
// Entité provenant d'ailleurs (ex: API)
var product = new Product
{
    Id = 1,
    Name = "Updated Product",
    Price = 49.99m,
    CategoryId = 2
};

// Méthode 1: Attacher et marquer comme modifiée
context.Products.Attach(product);
context.Entry(product).State = EntityState.Modifi

// Méthode 2: Mise à jour directe
context.Products.Update(product);

// Persister les changements
context.SaveChanges();
```

Opérations CRUD - Delete

Suppression d'une entité suivie

```
// Récupérer l'entité
var product = context.Products.Find(1);

// Supprimer l'entité
context.Products.Remove(product);

// Persister la suppression
context.SaveChanges();
```

Suppression par ID

```
// Créer une référence avec seulement l'ID
var product = new Product { Id = 1 };

// Attacher et marquer comme supprimée
context.Products.Attach(product);
context.Products.Remove(product);
```

Suppression en cascade

```
// Configuration de la relation (dans OnModelCreatin
modelBuilder.Entity<Order>()
    .HasMany(o => o.OrderItems)
    .WithOne(oi => oi.Order)
    .HasForeignKey(oi => oi.OrderId)
    .OnDelete(DeleteBehavior.Cascade);
```

Options de comportement de suppression

- `DeleteBehavior.Cascade` : Suppression en cascade
- `DeleteBehavior.Restrict` : Empêche la suppression
- `DeleteBehavior.SetNull` : Met la FK à null

Gestion des transactions

Transactions implicites

- SaveChanges utilise une transaction implicite
- Toutes les modifications réussissent ou échouent
- Adapté aux cas simples

```
// Transaction implicite
context.Customers.Add(customer);
context.Orders.Add(order);
context.SaveChanges();  // Transaction atomique
```

Transactions explicites

- Contrôle précis du début et de la fin
- Possibilité de faire un rollback manuel
- Peut englober plusieurs SaveChanges

```
// Transaction explicite
using (var transaction = context.Database.BeginTransa
{
    try
    {
        // Première opération
        var customer = new Customer { Name = "New Cus
        context.Customers.Add(customer);
        context.SaveChanges();

        // Utiliser l'ID généré
        var order = new Order
        {
            CustomerId = customer.Id,
            OrderDate = DateTime.Now,
            TotalAmount = 99.99m
        };
        context.Orders.Add(order);
        context.SaveChanges();

        // Valider la transaction
        transaction.Commit();
    }
```

Suivi des changements (Change Tracking)

Fonctionnement du Change Tracker

- Suit les modifications des entités
- Compare avec l'état d'origine
- Détermine les requêtes SQL nécessaires
- Gère l'état des entités

Accès au Change Tracker

```
// Obtenir le Change Tracker
var tracker = context.ChangeTracker;

// Voir les entités suivies
var entries = tracker.Entries().ToList();
```

Optimisations du suivi

```
// Désactiver le suivi pour les requêtes en lecture
var products = context.Products
    .AsNoTracking()
    .ToList();

// Suivi avec identité uniquement
var customers = context.Customers
    .AsTracking(QueryTrackingBehavior.NoTrackingWith
    .ToList();

// Configurer au niveau du contexte
context.ChangeTracker.QueryTrackingBehavior =
    QueryTrackingBehavior.NoTracking;

// Forcer la détection des changements
context.ChangeTracker.DetectChanges();

// Réinitialiser le suivi
context.ChangeTracker.Clear();
```

TP : Implémentation complète des opérations CRUD

Objectifs

1. Créer un service complet pour les opérations CRUD
2. Implémenter les méthodes Create, Read, Update, Delete
3. Utiliser les transactions pour les opérations complexes
4. Optimiser le suivi des changements
5. Gérer les erreurs correctement

Service CRUD à implémenter

```
public interface IProductService
{
    Task<List<Product>> GetAllAsync();
    Task<Product> GetByIdAsync(int id);
    Task<Product> CreateAsync(Product product);
    Task<bool> UpdateAsync(Product product);
    Task<bool> DeleteAsync(int id);
    Task<bool> SoftDeleteAsync(int id);

    // Opération complexe avec transaction
    Task<Order> CreateOrderWithItemsAsync(
        Order order,
        List<OrderItem> items);
}
```

Pause café ☕

Rendez-vous à 15h45 pour la dernière session de la journée

Requêtes avancées

Au-delà des requêtes simples

- Jointures complexes
- Sous-requêtes
- Expressions de table communes (CTE)
- Fonctions de fenêtrage (Window Functions)
- Requêtes d'agrégation avancées

Défis des requêtes complexes

- Traduction SQL efficace
- Performances

```
// Exemple de requête avancée
var topCustomers = await context.Customers
    .Where(c => c.Orders.Count > 5)
    .OrderByDescending(c => c.Orders.Sum(o => o.Tota
    .Take(10)

    // Projection vers un DTO anonyme
    .Select(c => new {
        c.Id, c.Name, c.Email,
        TotalOrders = c.Orders.Count,

        // Calculs et agrégations
        TotalSpent = c.Orders.Sum(o => o.TotalAmount
        AverageOrderValue = c.Orders.Average(o => o.
        FirstOrderDate = c.Orders.Min(o => o.OrderDa
        LastOrderDate = c.Orders.Max(o => o.OrderDat

        // Sous-collection filtrée et transformée
        RecentOrders = c.Orders
            .OrderByDescending(o => o.OrderDate)
            .Take(3)
            .Select(o => new { o.Id, o.OrderDate, o.
```

Requêtes avec jointures

Types de jointures

- Inner Join (jointure interne)
- Left Join (jointure externe gauche)
- Explicit Join (jointure explicite)
- Group Join (jointure avec groupement)

Jointures implicites

```
// Inner join implicite
var products = context.Products
    .Where(p => p.Category.Name == "Electronics")
    .ToList();

// Jointure multi-niveau
var orderItems = context.OrderItems
    .Where(oi => oi.Order.Customer.Country == "Franc
```

Jointures explicites

```
// Jointure explicite (LINQ méthode)
var query = context.Products
    .Join(
        context.Categories,
        product => product.CategoryId,
        category => category.Id,
        (product, category) => new {
            ProductName = product.Name,
            CategoryName = category.Name,
            Price = product.Price
        }
    )
    .Where(x => x.Price > 50)
    .OrderBy(x => x.CategoryName)
    .ThenBy(x => x.Price);

// Jointure explicite (LINQ requête)
var query = from p in context.Products
            join c in context.Categories
                on p.CategoryId equals c.Id
            where p.Price > 50
```

Left Join et Group Join

Left Join (jointure externe gauche)

```
// Left join en LINQ requête
var query = from c in context.Customers
            join o in context.Orders
                on c.Id equals o.CustomerId into cus
            from o in customerOrders.DefaultIfEmpty(
            select new {
                CustomerName = c.Name,
                OrderId = o != null ? o.Id : (int?)n
                OrderDate = o != null ? o.OrderDate
            };

// Left join alternatif avec Include
var customers = context.Customers
    .Include(c => c.Orders)
    .Select(c => new {
        CustomerName = c.Name,
        Orders = c.Orders.Select(o => new {
            o.Id, o.OrderDate, o.TotalAmount
```

Group Join (jointure avec groupement)

```
// Group join en LINQ requête
var query = from c in context.Customers
            join o in context.Orders
                on c.Id equals o.CustomerId into cus
            select new {
                Customer = c,
                OrderCount = customerOrders.Count(),
                TotalAmount = customerOrders.Sum(o =
                Orders = customerOrders.ToList()
            };

// Group join alternatif avec agrégation
var result = context.Customers
    .Select(c => new {
        Customer = c,
        OrderCount = c.Orders.Count,
        TotalAmount = c.Orders.Sum(o => o.TotalAmoun
        Orders = c.Orders.ToList()
```

Projections et sélections personnalisées

Pourquoi projeter?

- Réduire la quantité de données
- Transformer les données pour les clients
- Éviter le chargement d'entités complètes
- Améliorer les performances

Types de projections

- Objets anonymes
- DTOs (Data Transfer Objects)
- Types complexes
- Projections hiérarchiques

```
// Projection simple vers un objet anonyme
var products = await context.Products
    .Select(p => new {
        p.Id,
        p.Name,
        p.Price,
        CategoryName = p.Category.Name,

        // Calculs dans la projection
        DiscountedPrice = p.Price * 0.9m,
        InStock = p.Stock > 0,
        DaysInInventory = EF.Functions.DateDiffDa
    })
    .ToListAsync();

// Projection vers un DTO avec hiérarchie
var customers = await context.Customers
    .Select(c => new CustomerDto
    {
        Id = c.Id,
        FullName = c.FirstName + " " + c.LastName
        Email = c.Email,
```

Requêtes avec groupement

Opérations de groupement

- `GroupBy` : Regrouper des entités par clé
- Agrégations sur les groupes
- Filtres avant ou après groupement
- Projections sur le résultat

Cas d'utilisation

- Rapports et tableaux de bord
- Analyses statistiques
- Données de synthèse
- Regroupements hiérarchiques

```
// Groupement simple
var salesByCategory = await context.Products
    .GroupBy(p => p.CategoryId)
    .Select(g => new {
        CategoryId = g.Key,
        CategoryName = g.First().Category.Name,
        ProductCount = g.Count(),
        TotalStock = g.Sum(p => p.Stock),
        AveragePrice = g.Average(p => p.Price),
        MinPrice = g.Min(p => p.Price),
        MaxPrice = g.Max(p => p.Price)
    })
    .ToListAsync();

// Groupement multi-niveau
var salesByMonth = await context.Orders
    .GroupBy(o => new {
        Year = o.OrderDate.Year,
        Month = o.OrderDate.Month
    })
    .Select(g => new {
        Year = g.Key.Year,
        Month = g.Key.Month,
```

Pagination des résultats

Importance de la pagination

- Améliore les performances
- Réduit la consommation de mémoire
- Améliore l'expérience utilisateur
- Adapté aux grandes quantités de données

Techniques de pagination

- Skip/Take
- Keyset Pagination (cursor-based)
- Pagination côté client vs serveur
- Comptage d'items total

```
// Pagination basique avec Skip/Take
public async Task<List<Product>> GetPagedProductsAsy
    int page, int pageSize)
{
    return await context.Products
        .OrderBy(p => p.Name)
        .Skip((page - 1) * pageSize)
        .Take(pageSize)
        .ToListAsync();
}

// Pagination avec compte total
public async Task<(List<Product> Items, int TotalCou
    GetPagedProductsWithCountAsync(int page, int pag
{
    var query = context.Products.AsQueryable();

    // Obtenir le nombre total sans appliquer Skip/T
    var totalCount = await query.CountAsync();
    var totalPages = (int)Math.Ceiling(totalCount /

    // Récupérer la page demandée
```

TP : Développement de requêtes complexes

Objectifs

1. Implémenter des jointures complexes
2. Créer des projections personnalisées
3. Développer des requêtes avec groupement
4. Mettre en place une pagination efficace
5. Optimiser les performances

Scénarios à implémenter

1. Tableau de bord des ventes par catégorie et par mois
2. Recherche de produits avec filtres multiples et facettes
3. Liste des meilleurs clients avec historique des achats
4. Rapport de stock avec alertes de réapprovisionnement
5. API paginée pour naviguer dans un grand catalogue