Développement d'applications .NET

Développement web avec l'ASP.NET Core : **Accès aux données avec ADO.NET et Entity Framework Core**

Objectifs pédagogiques de cette séance

1) Comprendre les approches d'accès aux données dans .NET
2) Utiliser **ADO.NET** pour interagir directement avec **SQL Server**
3) Maîtriser les opérations CRUD avec **Entity Framework (EF) Core**
4) Maîtriser la structure d'un DbContext EF Core
5) Découvrir le langage d'interrogation **LINQ**
6) Choisir la bonne technologie selon les besoins du projet

Pourquoi plusieurs méthodes d'accès aux données ?

Parce que chaque approche a sa philosophie :

- **Besoins différents**: performance, contrôle ou rapidité de développement.
- **Niveaux d'abstraction variés**: SQL direct (ADO.NET), ORM (EF Core), langage de requêtes (LINQ).
- **Compatibilité des projets**: anciens systèmes vs applications modernes.
- **Flexibilité**: choisir l'outil le plus adapté au contexte et aux contraintes.

Technologie	Niveau d'abstraction	Usage
ADO.NET	Très bas niveau	Performance, contrôles fins
Entity Framework Core	Haut niveau (ORM)	Productivité, mapping automatique

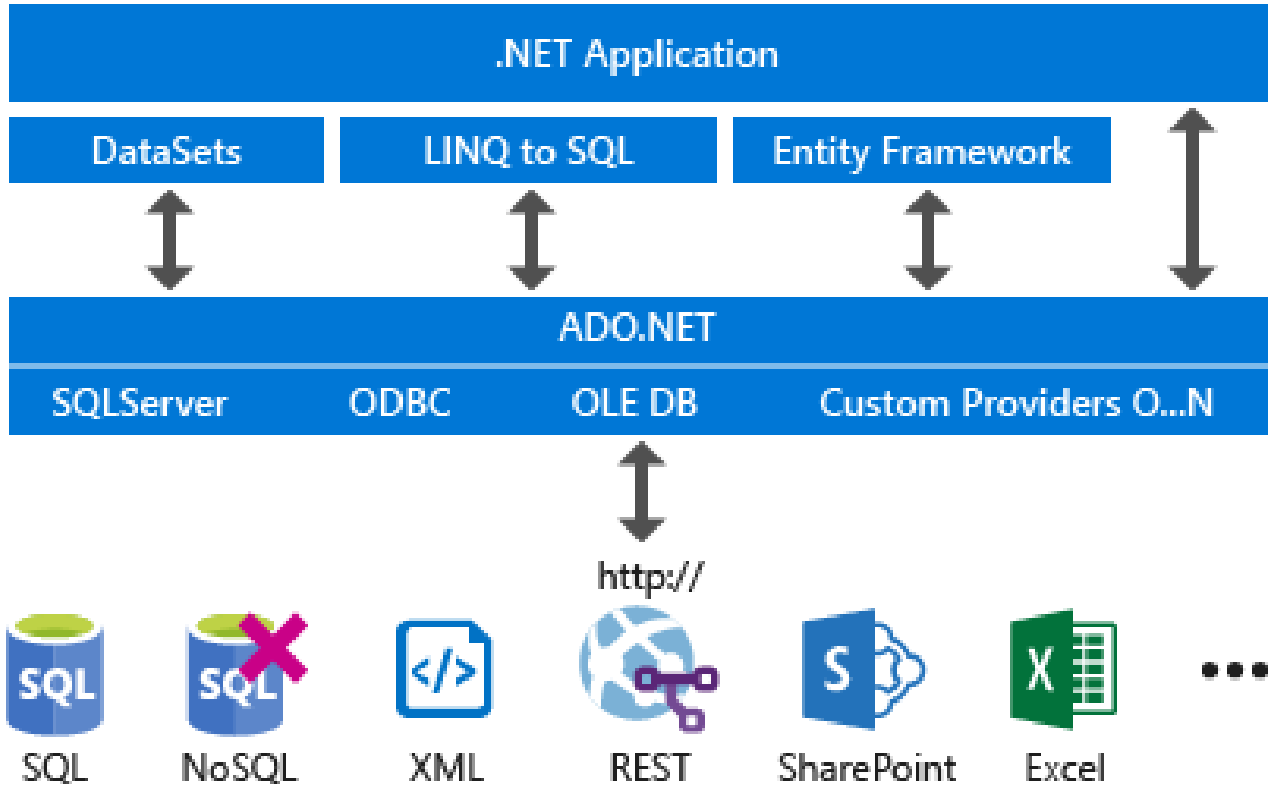
ADO.NET

Qu’est-ce que ADO.NET ?

ADO.NET est la couche **bas niveau** permettant :

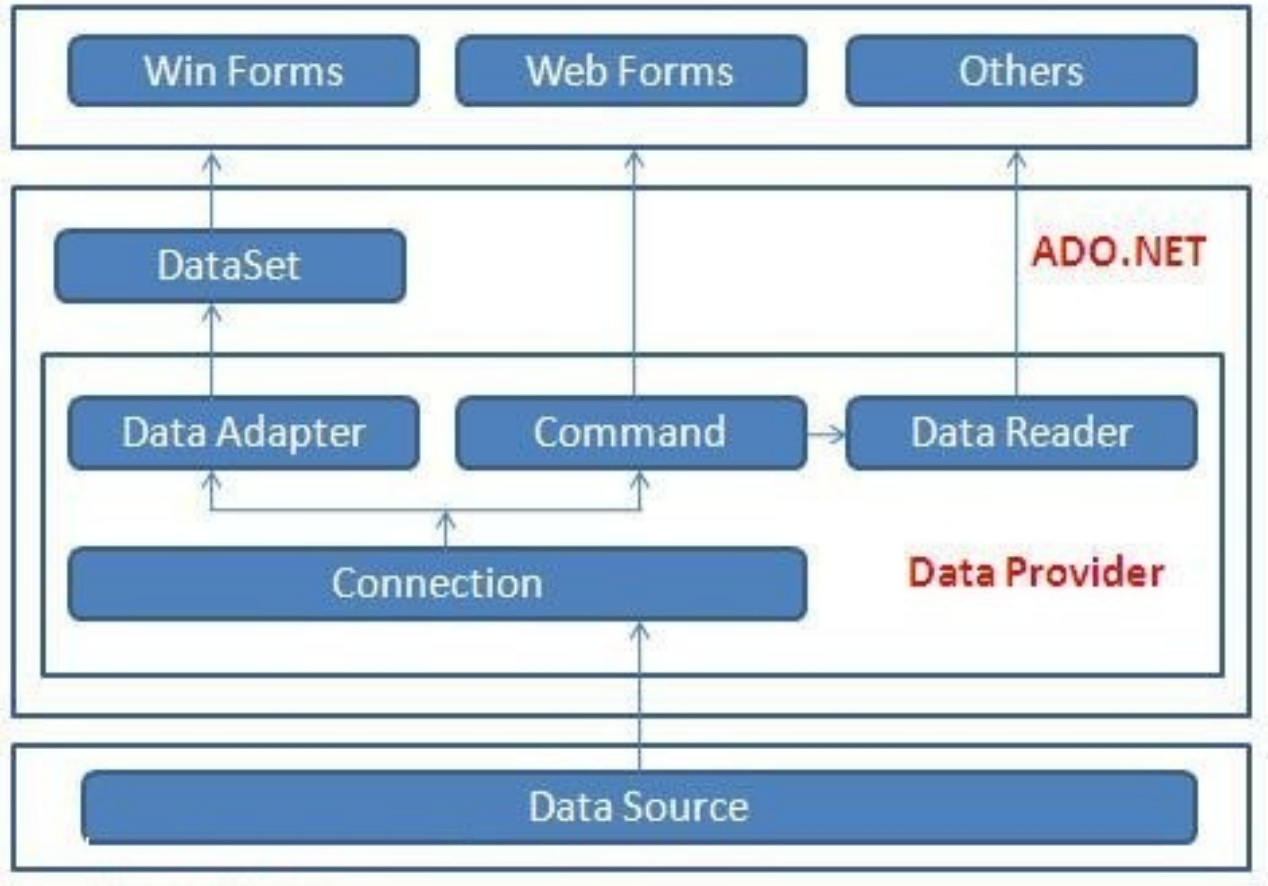
- D’ouvrir des connexions SQL Server
- D’exécuter des commandes SQL
- De lire les résultats avec précision
- De gérer transactions, paramètres et pooling

Il représente l’approche la plus performante pour interagir avec une base de données SQL.



ADO.NET : Composants principaux

- **SqlConnection** : ouverture de session avec la base
- **SqlCommand** : instructions SQL
- **SqlDataReader** : lecture streamée, forward-only
- **SqlDataAdapter** : remplissage DataSet
- **DataSet / DataTable** : stockage déconnecté



ADO.NET : Exemple complet d'une requête SELECT

1) Définir la chaîne de connexion

- Contient les infos nécessaires pour accéder à la base : Serveur, Base de données, Authentification, et Options supplémentaires. (voir le format dans l'app de démo)

2) Créer et ouvrir une connexion

- Établir la liaison avec la base de données.
- Assurer la fermeture automatique via using.

3) Préparer une commande SQL

- Définir la requête (ex. : SELECT Id, Name, Price…).
- Associer la commande à la connexion ouverte.

4) Exécuter la commande

Utiliser une méthode qui renvoie un DataReader pour lire les résultats.

5) Lire les données ligne par ligne

- Avancer dans le résultat avec Read().
- Récupérer les valeurs des colonnes selon leur type (int, string, decimal…).

6) Exploiter les données

- Afficher, stocker, ou traiter les informations récupérées.

```
using var con = new SqlConnection(connString);
con.Open();

var cmd = new SqlCommand("SELECT Id, Name, Price FROM Products", con);
var reader = cmd.ExecuteReader();

while (reader.Read())
{
    Console.WriteLine($"{reader.GetInt32(0)} - {reader.GetString(1)} - {reader.GetDecimal(2)}
}
```

ADO.NET : Exemple INSERT sécurisé (paramétré)

1) Définir la commande SQL avec des paramètres

- La requête utilise des placeholders (ex. : @n, @p) à la place des valeurs.
- Cela évite d'écrire directement les données dans la requête.

2) Créer la commande et l'associer à la connexion

- Préparation de l'objet qui exécutera l'instruction SQL.
- La commande connaît la structure de la requête et les paramètres attendus.

3) Fournir les valeurs des paramètres

- Ajouter chaque valeur via la collection Parameters. Chaque paramètre reçoit : un nom (ex. @n), et une valeur (ex. "Laptop")
- Cette méthode :
  - empêche l'injection SQL
  - garantit que les types sont correctement gérés
  - rend le code plus propre et réutilisable

4) Exécuter la commande

- L'application envoie l'instruction à la base.
- Comme il s'agit d'un INSERT, on utilise une exécution sans retour de résultat.

```
string sql = "INSERT INTO Products (Name, Price) VALUES (@n, @p)";
using var cmd = new SqlCommand(sql, con);

cmd.Parameters.AddWithValue("@n", "Laptop");
cmd.Parameters.AddWithValue("@p", 1599.99M);

cmd.ExecuteNonQuery();
```

ADO.NET : Transactions

1) Démarrer une transaction

- La connexion ouvre une transaction.
- Toutes les opérations suivantes seront exécutées dans le même contexte.
- Objectif : garantir que les modifications sont cohérentes.

2) Exécuter plusieurs opérations SQL

- Plusieurs commandes sont envoyées à la base :
  - Exemple : débiter un compte, puis créditer un autre
- Chaque commande est liée à la même transaction.

3) Vérifier que toutes les opérations réussissent

Tant que chaque commande s'exécute correctement, la transaction peut être validée.

4) Commit de la transaction

Si tout s'est bien passé → Commit(), toutes les modifications deviennent permanentes.

5) (En cas d'erreur) Annulation complète

- Si une opération échoue → Rollback (automatique ou manuel).
- Cela garantit que rien n'est partiellement modifié.
- Principe d'atomicité

```
using var transaction = con.BeginTransaction();
var cmd = new SqlCommand("UPDATE Accounts SET Amount = Amount - 100 WHERE Id=1", con, transact
cmd.ExecuteNonQuery();

cmd = new SqlCommand("UPDATE Accounts SET Amount = Amount + 100 WHERE Id=2", con, transaction)
cmd.ExecuteNonQuery();

transaction.Commit();
```

ADO.NET : Avantages et inconvénients

Avantages

- Performances maximales
- Très précis, prévisible
- Parfait pour reporting, batch processing
- Idéal quand EF génère un SQL inadéquat

Inconvénients

- Beaucoup de code répétitif
- Faible productivité
- Pas de mapping automatique entre objets et tables
- Plus difficile à maintenir en équipe

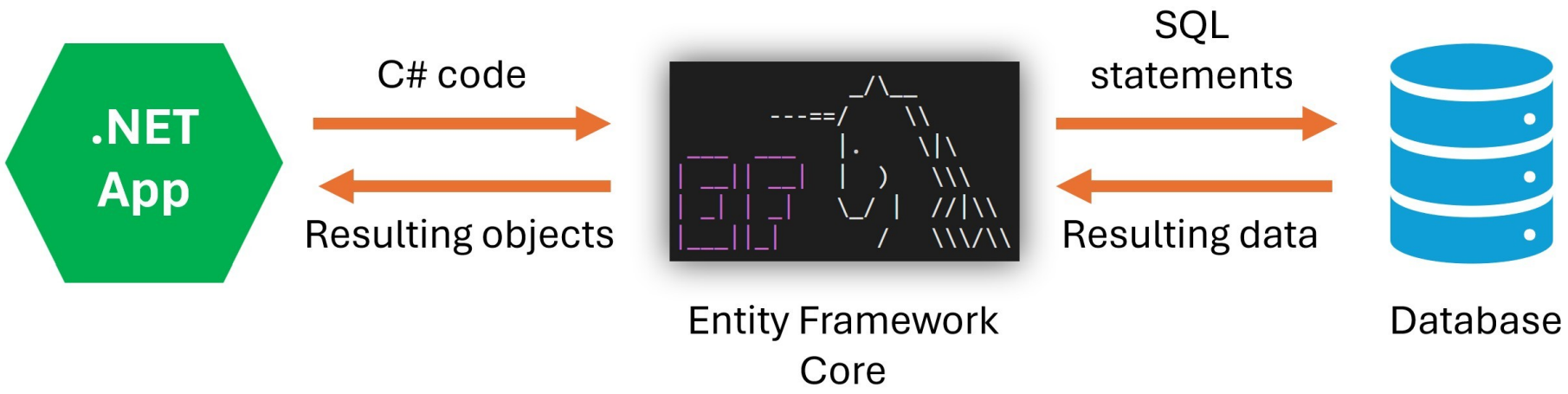
Entity Framework Core

Entity Framework Core : définition

Entity Framework Core est l'**ORM** (Object Relational Mapping) de Microsoft, **successeur** moderne d'**Entity Framework**, réécrit pour offrir de meilleures performances, plus de flexibilité et une compatibilité multiplateforme

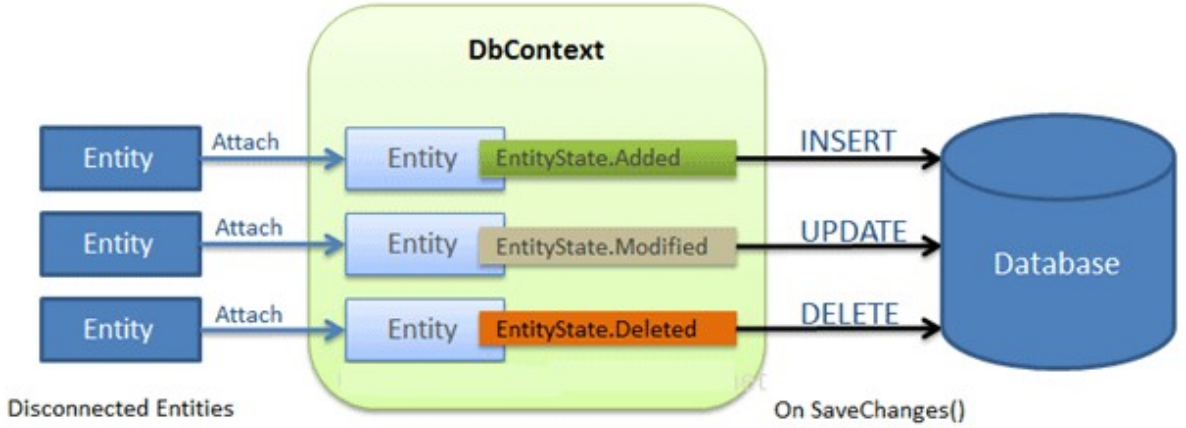
Entity Framework Core permet :

- Mapping automatique entre Objet ↔ Base SQL
- Génération SQL automatique
- Gestion relations (1–1, 1–N, N–N)
- Gestion des migrations
- LINQ pour interroger les données
- Change tracking intelligent



Architecture générale de EF Core

- **DbContext** :
  - représente une session avec la base
  - gère les connexions, le suivi des entités, les requêtes
- **DbSet<TEntity>** :
  - représente une table ou une vue
  - permet d'interroger et de modifier les données
- **Providers** :
  - adaptateurs pour chaque SGBD (SQL Server, SQLite, PostgreSQL, MySQL…)



Pipeline d'exécution d'une requête :

1) Écriture d'une requête LINQ sur un DbSet
2) EF Core traduit la requête LINQ en SQL
3) EF Core utilise ADO.NET pour exécuter le SQL
4) Les résultats sont mappés en objets C#

Installation d'EF Core

Installer les packages NuGet EF Core (pour SQL Server) :

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

EF Core fonctionne avec différents SGBD via providers :

- Microsoft.EntityFrameworkCore.SqlServer : SQL Server
- Microsoft.EntityFrameworkCore.Sqlite : SQLite
- Npgsql.EntityFrameworkCore.PostgreSQL : PostgreSQL
- Pomelo.EntityFrameworkCore.MySql : MySQL/MariaDB
- Microsoft.EntityFrameworkCore.Cosmos : Azure Cosmos DB

Le provider :

- adapte EF Core au dialecte SQL ou au backend de l'SGBD
- influence certaines fonctionnalités (types supportés, fonctions…)

EF Core : création d'entités

- Une entité représente une table ou un enregistrement dans la base
- Propriétés = colonnes
- Doit avoir une clé primaire (par convention : Id ou NomClasseId)

Pourquoi partir des classes ?

- approche **Code-First** : on part du code pour créer le schéma
- plus naturel pour les développeurs C#

Exemple :

```
public class Product
{
    public int Id { get; set; }          // Clé primaire (par convention)
    public string Name { get; set; }     // Nom du produit
    public decimal Price { get; set; }   // Prix
}
```

Le DbContext : cœur d'EF Core

Représente une session de travail avec la base

Principales responsabilités :

- fournir des **DbSet<TEntity>** pour chaque entité
- suivre les changements sur les objets (**tracking**)
- traduire les requêtes **LINQ en SQL**
- gérer **SaveChanges**() pour envoyer les modifications

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
    }

    // Représente la table Products
    public DbSet<Product> Products { get; set; }

    // Config avancée (facultatif au début)
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Exemple : renommer la table
        // modelBuilder.Entity<Product>().ToTable("Products");
    }
}
```

Configuration du DbContext

Le DbContext se configure via :

- un constructeur qui prend DbContextOptions
- la méthode OnConfiguring (pour les scénarios simples)

Ce qu'on y configure :

- la chaîne de connexion
- le provider (SQL Server, SQLite, etc.)

Exemple conceptuel :

- "Dans ce contexte, utilise SQL Server avec cette connexion."
- Dans une API ASP.NET Core, on préfère configurer le DbContext via l'injection de dépendances (voir plus loin).

Chaîne de connexion EF Core

Contient les infos pour se connecter à la base :

- nom du serveur
- nom de la base
- méthode d'authentification
- options (timeout, chiffrement…)

Exemples :

SQL Server : Server=.;Database=MyDb;Trusted_Connection=True;

SQLite : Data Source=mydb.db;

Souvent stockée dans **appsettings.json** pour :

- éviter de mettre les secrets en dur dans le code
- pouvoir changer la config sans recompiler

Enregistrement du DbContext dans ASP.NET Core

EF Core s'intègre nativement avec DI : on enregistre le DbContext dans les services

Avantages :

- EF Core gère le cycle de vie du contexte (par requête HTTP par ex.)
- On injecte simplement AppDbContext dans les contrôleurs / services

Exemple conceptuel :

"Pour ce projet, dès qu'on a besoin d'AppDbContext, crée-le avec SQL Server et cette chaîne de connexion."

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

// Récupérer la chaîne de connexion
var connString = builder.Configuration.GetConnectionString("DefaultConnection");

// Enregistrer le DbContext dans la DI
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(connString));

var app = builder.Build();
// ...
app.Run();
```

Approche Code-First vs Database-First

Code-First :

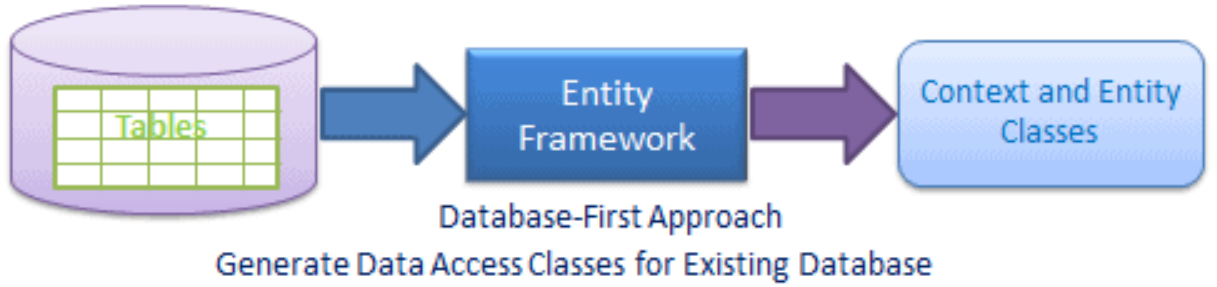
- on définit d'abord les classes C# (entités)
- on laisse EF Core créer / mettre à jour la base via les migrations
- idéal pour nouveaux projets

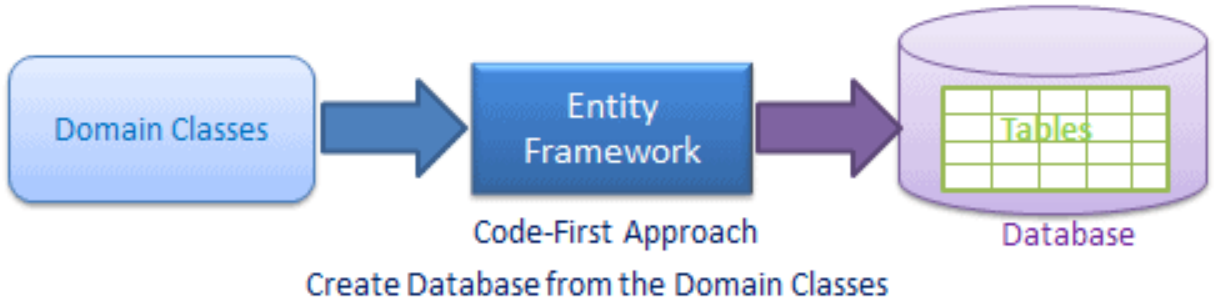
Database-First :

- la base existe déjà
- on génère les classes C# à partir du schéma (dotnet ef dbcontext scaffold)
- pratique pour intégrer un projet existant

EF Core supporte :

- les deux approches
- et même un mix (on part de la DB, puis on évolue avec des migrations).





Migrations : principe

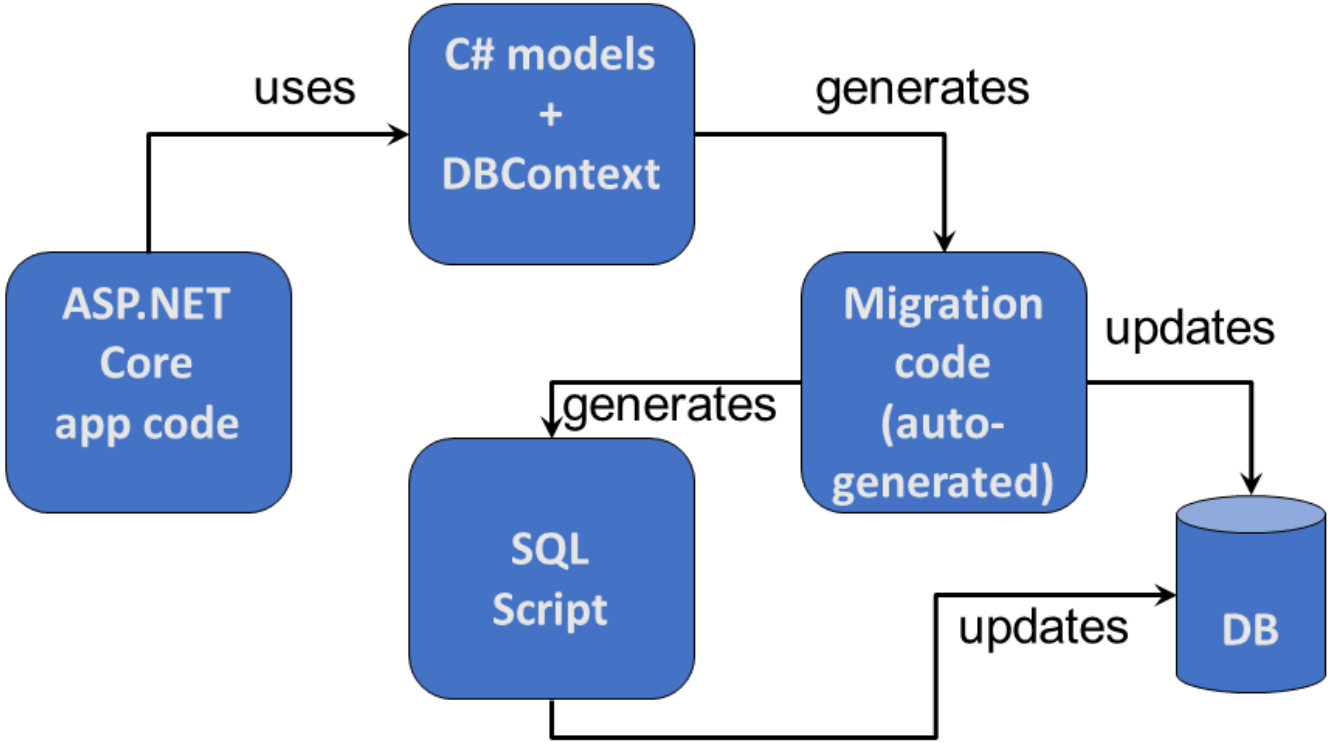
Problème : comment synchroniser évolution des classes ↔ schéma de base ?

Migrations = scripts gérés par EF Core qui décrivent les changements :

- ajout/suppression de tables
- ajout/suppression de colonnes
- changements de types, contraintes, etc.

Avantages :

- historique des changements du schéma
- reproductible (dev, test, prod)
- évite d'écrire tout le SQL à la main



Migrations : commandes de base

Créer une migration : **dotnet ef migrations add InitialCreate**

Appliquer les migrations à la base : **dotnet ef database update**

Modifier le modèle (ex. ajout d'une propriété) :

modifier la classe entité : **dotnet ef migrations add AddPriceToProduct**

dotnet ef database update

Chaque migration contient :

une méthode **Up**() (appliquer les changements)

une méthode **Down**() (annuler les changements)

Opérations CRUD : principe général

Create (ajouter)

1) créer un objet C#
2) l'ajouter au DbSet
3) appeler SaveChanges()

Read (lire)

1) interroger le DbSet avec LINQ
2) ex. filtrer, trier, paginer

Update (modifier)

1) récupérer l'entité
2) modifier ses propriétés
3) SaveChanges()

Delete (supprimer)

1) récupérer l'entité
2) la supprimer du DbSet
3) SaveChanges()

Ajouter un objet avec EF Core

```
using (var context = new AppDbContext(options))
{
    var product = new Product
    {
        Name = "Laptop",
        Price = 1599.99m
    };

    context.Products.Add(product);   // Marque comme "Added"
    context.SaveChanges();           // Génère un INSERT
}
```

- **Add**() : indique qu'on veut insérer
- **SaveChanges**() : envoie le SQL à la base

Lire des données avec LINQ

```
using (var context = new AppDbContext(options))
{
    // Tous les produits
    var allProducts = context.Products.ToList();

    // Filtrer
    var expensive = context.Products
        .Where(p => p.Price > 1000)
        .OrderBy(p => p.Name)
        .ToList();

    // Un seul élément
    var firstLaptop = context.Products
        .FirstOrDefault(p => p.Name == "Laptop");
}
```

- Where, OrderBy, FirstOrDefault → opérateurs LINQ
- Requête traduite en SQL puis exécutée

Modifier une entité

```
using (var context = new AppDbContext(options))
{
    var product = context.Products.FirstOrDefault(p => p.Id == 1);

    if (product != null)
    {
        product.Price = 1499.99m;   // Modification en mémoire

        context.SaveChanges();      // Génère un UPDATE
    }
}
```

- EF Core détecte les propriétés modifiées
- SaveChanges() génère un UPDATE Products SET... WHERE Id = 1

Supprimer une entité

```
using (var context = new AppDbContext(options))
{
    var product = context.Products.FirstOrDefault(p => p.Id == 1);

    if (product != null)
    {
        context.Products.Remove(product); // Marque comme "Deleted"
        context.SaveChanges();            // Génère un DELETE
    }
}
```

Schéma mental : EF Core suit l'état (Added, Modified, Deleted) et adapte l'SQL

Requêtes LINQ simples

EF Core supporte :

- Where (filtrer)
- OrderBy, ThenBy (trier)
- Select (projeter)
- First, FirstOrDefault, Single, Any, Count, etc.

Important :

- tant qu'on n'énumère pas (ToList, First…), la requête n'est pas exécutée
- EF Core traduit la requête LINQ en SQL au moment de l'exécution

Tracking vs No-Tracking

Par défaut, EF Core suit les entités qu'il charge depuis la base, si tu modifies une propriété, SaveChanges() générera un UPDATE

AsNoTracking() :

- désactive le tracking pour cette requête
- utile pour les scénarios de lecture seule (read-only)
- améliore les performances (moins de mémoire)

```
// Avec tracking (par défaut) : EF suit les changements
var productsTracked = context.Products.ToList();

// Sans tracking : meilleur pour lecture seule
var productsNoTracking = context.Products
    .AsNoTracking()
    .ToList();
```

Règle pratique :

- pour les pages/listes de consultation → AsNoTracking()
- pour les scénarios où tu vas modifier l'entité → tracking activé

Relations entre entités

Relations supportées :

- **Un-à-plusieurs (1–N)** : une Category, plusieurs Products
- **Plusieurs-à-plusieurs (N–N)** : un étudiant avec plusieurs cours, un cours avec plusieurs étudiants
- **Un-à-un (1–1)** : ex. User ↔ UserProfile

Représentées via :

- propriétés de navigation (Category, Products)
- clés étrangères (CategoryId)

EF Core peut :

- détecter les relations par convention
- ou via configuration explicite

Relations entre entités : exemple One-To-One

Relation :

- Un **User** possède un seul **UserProfile**
- Un **UserProfile** appartient à un seul **User**

EF Core utilise la **clé primaire** du profil comme **clé étrangère**

```
public class User
{
    public int Id { get; set; }
    public string Username { get; set; }

    // Navigation : One-to-One
    public UserProfile Profile { get; set; }
}

public class UserProfile
{
    // PK = FK vers User
    public int Id { get; set; }

    public string Bio { get; set; }
    public string PhoneNumber { get; set; }

    // Navigation
    public User User { get; set; }
}
```

Relations entre entités : exemple One-To-Many

Relation Category (1) – Products (N)

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

    // 1 catégorie → plusieurs produits
    public List<Product> Products { get; set; }
}

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

    // Clé étrangère
    public int CategoryId { get; set; }
    // Navigation
    public Category Category { get; set; }
}
```

Dans AppDbContext :

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

EF Core détecte la relation par convention (CategoryId + navigation)

Relations entre entités : exemple Many-To-Many

Relation :

- Un **Étudiant** peut suivre **plusieurs Cours**
- Un **Cours** peut avoir **plusieurs Étudiants**

Sans classe de jointure explicite :

- EF Core crée **automatiquement** la **table intermédiaire**.
- EF Core génère Une table StudentCourse (clé composée : StudentId, CourseId)

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

    // Many-to-Many
    public List<Course> Courses { get; set; }
}

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

    // Many-to-Many
    public List<Student> Students { get; set; }
}
```

Avec classe de jointure explicite (optionnel) :

Utile si tu veux ajouter des colonnes (ex: date d'inscription)

```
public class StudentCourse
{
    public int StudentId { get; set; }
    public Student Student { get; set; }

    public int CourseId { get; set; }
    public Course Course { get; set; }

    public DateTime EnrolledAt { get; set; }
}
```

Configuration avec DataAnnotations

On peut annoter les classes et propriétés avec des attributs C#:

- **[Key]** : clé primaire
- **[Required]** : non-null
- **[MaxLength(100)]** : limite de longueur
- **[Column("NomCol")]** : nom de colonne personnalisé
- **[ForeignKey("...")]** : relation explicite
- ...

Avantages :

- configuration proche du code
- simple pour les petits projets

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

public class Product
{
    [Key]                       // Facultatif si Id
    public int Id { get; set; }

    [Required]                  // NOT NULL
    [MaxLength(100)]            // VARCHAR(100)
    public string Name { get; set; }

    [Column(TypeName = "decimal(10,2)")]
    public decimal Price { get; set; }
}
```

Chargement des données : Include / Lazy Loading

Eager loading (Include) :

- charge la relation en même temps que l'entité principale
- ex. produits + catégories en une seule requête

Lazy loading :

- charge la relation à la demande la première fois qu'on y accède
- nécessite proxies et configuration supplémentaire

```
using Microsoft.EntityFrameworkCore;

using (var context = new AppDbContext(options))
{
    var productsWithCategory = context.Products
        .Include(p => p.Category)      // Jointure automatique
        .ToList();
}
```

Règle pratique pour les débutants :

- commencer avec Include (eager loading)
- éviter le lazy loading au début pour ne pas avoir de surprises

Transactions avec EF Core

Pour regrouper plusieurs opérations :

1) débuter une transaction avec **BeginTransaction**()
2) exécuter plusieurs **SaveChanges**()
3) **Commit()** si tout va bien, **Rollback**() en cas de problème

Utilité :

- garantir l'intégrité des données (tout ou rien)
- par exemple : débiter un compte + créditer un autre dans la même transaction

```
using var context = new AppDbContext(options);
using var transaction = context.Database.BeginTransaction();

try
{
    var account1 = context.Accounts.First(a => a.Id == 1);
    var account2 = context.Accounts.First(a => a.Id == 2);

    account1.Amount -= 100;
    account2.Amount += 100;

    context.SaveChanges();  // UPDATE pour les 2 comptes

    transaction.Commit();   // Tout est validé
}
catch
{
    transaction.Rollback(); // Annule toutes les modifs
    throw;
}
```

Exécution de SQL brut

Pour certaines requêtes très spécifiques ou optimisées on peut utiliser du SQL brut avec EF Core

Méthodes principales :

- Database.ExecuteSqlRaw() : commandes sans résultat (UPDATE, DELETE…)
- FromSqlRaw() : requêtes qui retournent des entités

Attention :

- bien gérer les paramètres pour éviter les injections SQL
- respecter la structure de l'entité si on retourne des entités

```
// Commande sans résultat (UPDATE/DELETE)
context.Database.ExecuteSqlRaw(
    "UPDATE Products SET Price = Price * 0.9 WHERE CategoryId = {0}", categoryId);

// Requête avec résultat
var cheapProducts = context.Products
    .FromSqlRaw("SELECT * FROM Products WHERE Price < {0}", 100)
    .ToList();
```