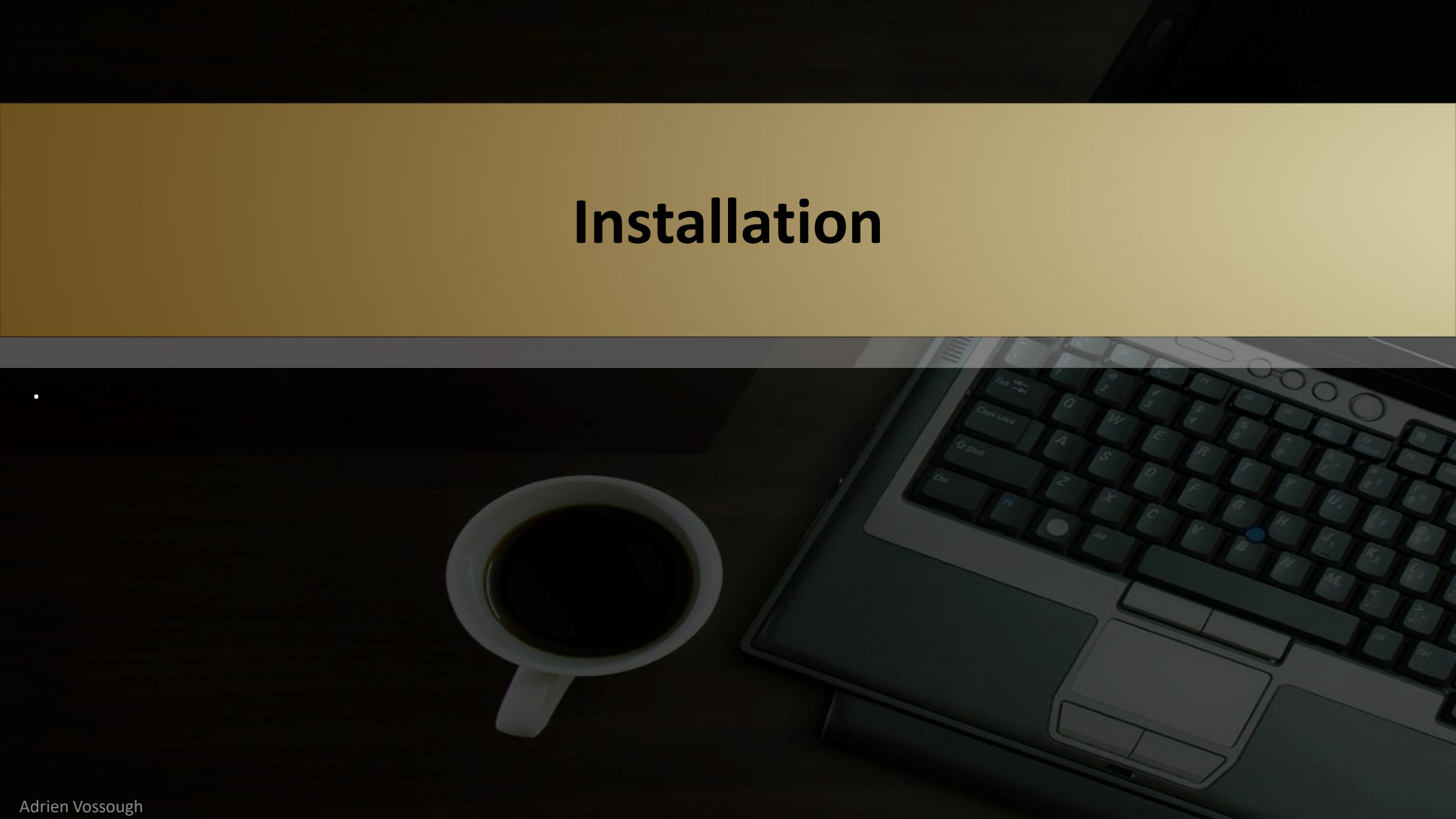

ENTITY
FRAMEWORK
.NET Core

Installation



- Création d’un projet console.

- Rechercher avec Nuget le package :
  Microsoft.EntityFrameworkCore

- Dans la console nuget taper : Install-Package Microsoft.EntityFrameworkCore.Tools

Modélisation

Adrien Vossough

Model First : lorsque la source de données n'existe pas, quand les développeurs commencent à écrire le code d'accès aux données.

Database First : lorsque la source de données existe ou qu'elle est maintenue par une tierce personne (**DBA**, administrateur base de données).

Le Model First est intéressant pour de petites applications (prototypes ou mobile) mais inadaptée aux applications métier.

Les Modèles sont des classes de l'application permettant de manipuler les données issues de la source de données(**SDD**).

avantage :

- géré par l'équipe de développement et ne nécessite pas de scripts ou d'outils externes.
- La structure de la source de données est intégrée à l'application, ce qui simplifie son évolution et sa maintenance ainsi que la gestion avec des outils comme Git.

inconvénient :

- Difficulté d'optimisation de la BDD et de la gestion des procédures stockées
- Compliqué lorsqu'il y a plusieurs BDD en jeu
- Compliqué dans les systèmes multi-applicatifs utilisant la même SDD

Database First, plus classique, la base de données existe avant la couche applicative.

Il existe des outils en ligne de commande pour gérer le DB First :

- des commandes PowerShell intégrées à la console NuGet de Visual Studio (Package Manager Console).
- des commandes intégrées dans .NET Core CLI

Le CLI permet de créer des scripts par des DBA ou des développeurs pour mettre à jours la BDD et être intégré dans chaîne d'intégration continue.

Pour cette approche, il est possible de générer le code ou de le faire manuellement.

La classe **DbContext** d'Entity Framework Core permet les manipulations de données et les enregistrements au niveau de la source de données.

Unit of Work est un design pattern qui permet le maintien d'un ensemble d'objets et la répercussion des modifications vers la SDD.

Ces écritures sont généralement sous la forme d'un ensemble atomique, c'est-à-dire indivisible : lorsqu'une des actions échoue, l'ensemble échoue (voir les transactions).

L'objet nommé **Change Tracker** de **DbContext** gère l'état des objets présents dans le contexte de données, et la méthode **SaveChanges** répercute les modifications dans la SDD.

L'objet **DbContext**, par défaut, n'a pas de fournisseur de données.

L'objet **DbContextOptions** fournit la **configuration** de **DbContext** en le passant au constructeur de ce dernier.

La classe **DbContextOptionsBuilder** permet de fabriquer **DbContextOptions** par ses fonctions :

- **ConfigureWarnings**() définit une fonction qui va gérer les alertes du framework.
- **EnableSensitiveDataLogging**() autorise l'enregistrement de données sensibles (informations dans les entités) dans les logs du framework.
- **UseInternalServiceProvider**() permet de remplacer le fournisseur de service pour le contexte. Doit respecter l'interface **IServiceProvider**
- **UseLoggerFactory**() permet de définir une fabrique d'objets **ILogger** pour le contexte.
- **UseMemoryCache**() pour remplacer le gestionnaire de cache. De type : **IMemoryCache**
- **UseModel**() : modèle objet à utiliser pendant la vie du contexte. Si cette méthode est utilisée, l'étape de configuration du modèle est ignorée.

La configuration de la SDD se fait par des méthodes d'extensions de **DbContextOptionsBuilder** qui sont étendues par les fournisseurs de données.

Avec le fournisseur SQL Server, il faut utiliser (using) **Microsoft.EntityFrameworkCore** dans le fichier configurant le contexte, cela ajoute la méthode **UseSqlServer**()

- Pour SQLite, la méthode est UseSqlite(), pour InMemoryUseInMemory, etc.

Configuration du contexte par son constructeur :

Configuration du contexte par le constructeur

```
using Microsoft.EntityFrameworkCore;

class Program {
    static void Main(string[] args) {
        DbContextOptionsBuilder builder = new DbContextOptionsBuilder();
        builder.UseSqlServer("CHAINE DE CONNEXION");

        using (var context = new Context(builder.Options)) {
            //...
        }
    }
}

public class Context : DbContext {
    public Context(DbContextOptions options): base(options) {
    }
}
```

Il est possible de configurer le contexte en redéfinissant sa méthode **OnConfiguring**() qui reçoit un objet DbContextOptionsBuilder.

Configuration du contexte par redéfinition de OnConfiguring()

```
class Program {
    static void Main(string[] args) {
        DbContextOptionsBuilder optionsBuilder = new DbContextOptionsBuilder();

        using (var context = new Context()) {
            //...
        }
    }
}

public class Context : DbContext {
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        optionsBuilder.UseSqlServer("CHAINE DE CONNEXION");
        base.OnConfiguring(optionsBuilder);
    }
}
```

Une SDD est constituée d'enregistrements sous forme de tables (SQL) ou de collections (NoSQL).

Les données sont exposées par les objets **DbContext** sous forme de collections de type **DbSet<T>**, où T est le type de l'entité.

Si le contexte est lié à une BDD, chaque collection est liée à une table.

Pour associer une **table** à un **DbContext**, il suffit de définir une propriété dont le type est DbSet<Entite>

Liaison du contexte vers une table

```
using Microsoft.Data.Entity;

public class Context : DbContext{
    public DbSet<Client> Clients { get; set; }
}

public class Client{
}
```

Chaque enregistrement dans une BDD est représenté par un objet appelé **entité.**

Les attributs d'une entité correspondent aux différentes valeurs d'une enregistrement et sont définis par des noms et des types.

Table des correspondances de SQL Server.

SQL Server	Type .NET
varchar	string
nvarchar	string
char	string
character	string
nchar	string
text	string
ntext	string
xml	string
tinyint	byte
binary	byte[]
varbinary	byte[]
timestamp	byte[]
rowversion	byte[]
image	byte[]
bit	bool
float	double

SQL Server	Type .NET
dec	decimal
decimal	decimal
money	decimal
numeric	decimal
smallmoney	decimal
int	int
real	float
smallint	short
bigint	long
date	DateTime
datetime	DateTime
datetime2	DateTime
smalldatetime	DateTime
datetimeoffset	DateTimeOffset
time	TimeSpan
uniqueidentifier	Guid

Ajouter un modèle

Il existe trois manière de modéliser de la BDD en objet.

EF utilise de nombreuses conventions de nommage qui si elles sont respectées évitent de la configuration. (Convention plutôt que configuration)

Configurer le modèle
Méthode 1

Exemple d'une entité

```
public class Context : DbContext {
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        base.OnConfiguring(optionsBuilder);
        optionsBuilder.UseSqlServer("CHAINE DE CONNEXION");
    }

    public DbSet<Client> Clients { get; set; }
}

public class Client {
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse { get; set; }
    public string CodePostal { get; set; }
    public string Ville { get; set; }
}
```

Suivre la convention de nommage d'EF permet de relier automatiquement à une certaines structure en base

EF liera cette entité à cette structure en BDD

```
CREATE TABLE[Client] (
    [ClientId] int NOT NULL IDENTITY,
    [Adresse] nvarchar(max),
    [CodePostal] nvarchar(max),
    [Nom] nvarchar(max),
    [Prenom] nvarchar(max),
    [Ville] nvarchar(max),
    CONSTRAINT[PK_Client] PRIMARY KEY([ClientId])
);
```

Il est possible de redéfinir le comportement d'EF par rapport aux entités à l'aide des annotations et/ou de la Fluent API (interface de programmation fluide)

Les espaces de noms des annotations sont :

- System.ComponentModel.DataAnnotations
- System.ComponentModel.DataAnnotations.Schema

Entité avec annotations

```
[Table("client")]
public class Client {
    [Column("id")]
    public int ClientId { get; set; }

    [Column("nom")]
    public string Nom { get; set; }

    [Column("prenom")]
    public string Prenom { get; set; }

    [Column("adresse")]
    public string Adresse { get; set; }

    [Column("code_postal")]
    public string CodePostal { get; set; }

    [Column("ville")]
    public string Ville { get; set; }
}
```

EF liera cette entité à cette structure en BDD

```
CREATE TABLE[client] (
    [id] int NOT NULL IDENTITY,
    [adresse] nvarchar(max),
    [code_postal] nvarchar(max),
    [nom] nvarchar(max),
    [prenom] nvarchar(max),
    [ville] nvarchar(max),

    CONSTRAINT[PK_Client] PRIMARY KEY([id])
);
```

Fluent API passe par une configuration programmatique ce qui permet d'avantage de possibilités que les deux autres méthodes..

Les comportements sont définis au niveau des DbContext, cela permet un comportement différent selon le contexte utilisé.

Entité reconfigurée par Fluent API

```
class Context : DbContext {
    public DbSet<Client> Clients { get; set; }

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

        modelBuilder.Entity<Client>().ToTable("client");
        modelBuilder.Entity<Client>()
            .Property(c => c.ClientId)
            .HasColumnName("id");
        // contrainte de clé primaire
        modelBuilder.Entity<Client>()
             .HasKey(c => c.ClientId);

        modelBuilder.Entity<Client>()
            .Property(c => c.CodePostal)
            .HasColumnName("code_postal")
            .HasMaxLength(5);
    }
}
```

EF liera cette entité à cette structure en BDD

```
CREATE TABLE[client] (
    [id] int NOT NULL IDENTITY,
    [Adresse] nvarchar(max),
    [code_postal] nvarchar(5),
    [Nom] nvarchar(max),
    [Prenom] nvarchar(max),
    [Ville] nvarchar(max),

    CONSTRAINT[PK_Client] PRIMARY KEY([id])
);
```

EF Core utilise dans tous les cas la configuration par convention qui sera écrasée par les annotations ou la configuration par Fluent.

Les priorités des configurations sont : Fluent, annotation et enfin par convention.

L'inclusion d'un model peut se faire de plusieurs manières :

- Par convention, chaque **DbSet<T> de l'objet DbContext** représente une entité, ceci est fait par le service DbSetFinder de l'assembly Microsoft.EntityFrameworkCore.

- Programmatique, en utilisant **ModelBuilder.Entity() ou ses surcharge**.
  Cette méthode doit être utilisée avant l'instanciation interne du modèle : le bon emplacement pour ces appels est dans la méthode **OnModelCreating**() du contexte.

Inclusion du model de façon programmatique

```
public class Context : DbContext {
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        modelBuilder.Entity<Client>();
        // Identique à : modelBuilder.Entity(typeof(Client));

        base.OnModelCreating(modelBuilder);
    }
}
```

L'inclusion par les propriétés DbSet<T> ne permet pas des chargements dynamiques d'entités.

L'inclusion programmatique demande plus de code et il n'est pas possible d'accéder à l'aide de propriétés aux types d'entités concernés : la fonction **DbContext.Set<T>** permet de créer dynamiquement un objet DbSet<T> de manière à obtenir un point d'entrée pour les requêtes.

L'inclusion automatique est lorsqu'une entité ajoutée au contexte dépend d'une autre classe :

Inclusion automatique

```
public class Client {
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public AdressePostale Adresse { get; set; }
}

public class AdressePostale {
    public int Id { get; set; }
    public string CodePostal { get; set; }
    public string Ville { get; set; }
}
```

Si client est ajouté au modèle, AdressePostale le sera aussi comme une inclusion dynamique faite par :
DbContext.Set<AdressePostale>()

Attention : le get et set de la propriété doivent être publics pour que l'inclusion automatique fonctionnent

C'est l'objet de type **PropertyDiscoveryConvention** fournit par le service **CoreConventionBuilderSet** qui gère cette convention et devrait être manipulé que pour l'écriture d'un fournisseur de données.

Empêcher l'inclusion automatique :

Empêcher l'inclusion automatique par annotation

```
using System.ComponentModel.DataAnnotations.Schema;
/* reste du code */
[NotMapped]
public class AdressePostale {
    public int Id { get; set; }
    public string CodePostal { get; set; }
    public string Ville { get; set; }
}

/* reste du code */
```

Empêcher l'inclusion automatique de façon programmatique

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<AdressePostale>();
    base.OnModelCreating(modelBuilder);
}
```

Exclure des propriétés :

Exclusion d'une propriété par annotation

```
public class Client {
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    [NotMapped]
    public string Adresse { get; set; }
    public string Ville { get; set; }
}
```

Exclusion d'une propriété par Fluent API

```
public class Context : DbContext {
    public DbSet<Client> Clients { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        /* reste du code */
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        modelBuilder.Entity<Client>()
                    .Ignore(client => client.Ville);

        base.OnModelCreating(modelBuilder);
    }
}
```

Mapper les tables et les colonnes

Mapper est le fait d'établir une correspondance entre deux éléments.

C'est le fournisseurs de données qui définit le nom des tables, par exemple, le fournisseur de SQL Serveur reprend le nom de la classe comme nom de table.

C'est le type **RelationalEntityTypeAnnotations** de l'assembly **Microsoft.EntityFrameworkCore.Relational** qui gère ce mappage conventionnel.

Il est possible de le redéfinir pour changer la convention de nommage.

Les annotations de **System.ComponentModel.DataAnnotations.Schema** permet aussi de redéfinir le nom d'une table :

Mappage d'une table vers l'entité par annotation

```
[Table("client")]
public class Client{
    /* Code de la classe */
}
```

Il est possible de définir le schéma qui contient la table :

Mappage d'une table vers l'entité par annotation

```
[Table("client", Schema = "ma_base")]
public class Client{
    /* Code de la classe */
}
```

Mappage d'une table vers l'entité par Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder){
    modelBuilder.Entity<Client>().ToTable("client");
    /*
     * Équivalences avec et sans schéma :
     *
     * modelBuilder.Entity<Client>().ToTable("client", "ma_base");
     * modelBuilder.Entity<Client>().ToTable<Client>("client");
     * modelBuilder.Entity<Client>().ToTable<Client>("client", "ma_base");
     */
    base.OnModelCreating(modelBuilder);
}
```

Tout comme pour les tables, c'est le fournisseur de données qui gère le mapping colonnes/propriétés.

Redéfinir le mapping des colonnes :

Mappage d'une propriété vers une colonne par annotation

```
[Table("client")]
class Client {
    [Column(TypeName = "nvarchar(250)")]
    public string Adresse1 { get; set; }
}
```

Les conventions sont gérés pour :

- Le mapping des colonnes par le type **InternalEntityTypeBuilder** utilisé par le type **PropertyDiscoveryConvention** de l'assembly **Microsoft.EntityFrameworkCore**

- Les types de données c'est **RelationalTypeMapper** de l'assembly **Microsoft.EntityFrameworkCore.Relational** et ses enfants au niveau des fournisseurs (**SqlServerTypeMapper** pour SQL Server).

La fonction générique **Property<T>** de **EntityTypeBuilder** retourne un objet **EntityPropertyBuilder**, permettant de modifier le mapping d'une propriété.

Mappage d'une propriété vers une colonne par Fluent

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Client>()
                .Property(c => c.Adresse)
                .HasColumnName("adresse")
                .HasColumnType("nvarchar(120)");

    base.OnModelCreating(modelBuilder);
}
```

Les clés primaires

Une clé primaire d'une entité est une propriété ou un groupe de propriété identifiant de manière unique une instance.

Par convention, la propriété servant de clé se nomme <Nom de la classe>Id ou Id

Clé primaire par convention

```
public class Compte {
    public int Id { get; set; }
    /* reste du code */
}
public class Client {
    public int ClientId { get; set; }
    /* reste du code */
}
```

Si une propriété Id et ClasseId se trouve dans la même entité, la première propriété trouvée sera la clé

La convention :

- est gérée par **KeyDiscoveryConvention** fournit service **CoreConventionBuilderSet** et ne devrait être manipulé que pour l'écriture d'un fournisseur de données.
- Les clés primaires de type **int** ou **Guid** sont générée automatiquement chaque ajout d'objets dans le contexte

Clé primaire par annotation

```
public class Client {
    public int Id { get; set; }
    [Key]
    public int ClePrimaire { get; set; }
    public string Nom { get; set; }
}
```

L'annotation prévaut sur la convention qui est donc ignorée

Plusieurs annotations [Key] pour une même entité provoque une exception **System.InvalidOperationException**

Clé primaire par Fluent

```
protected override void OnModelCreating(ModelBuilder builder) {
    builder.Entity<Client>()
           .HasKey(client => client.ClePrimaire);

    base.OnModelCreating(builder);
}
```

Avec Fluent API, il est possible de créer des clés primaires composites

Clé primaire composite avec Fluent

```
protected override void OnModelCreating(ModelBuilder builder) {
    builder.Entity<Client>()
            .HasKey(client => new {
                client.Id,
                client.ClePrimaire
            });

    base.OnModelCreating(builder);
}
```

Création d'un objet contenant les propriétés de Client servant de clé composite

Les relations

Les relations entre les tables dans une BDD est définie par des clés étrangères, pour les entités, nous retrouverons des références.

Entité principale/parente : Dans une relation, entité référencée

Entité dépendante/enfant : Dans une relation, entité qui référence

Clé principale : Permet d'identifier une entité de manière unique.

Clé étrangère : propriété de l'entité dépendante qui contient la valeur d'une clé principale de manière à la référencer.

Propriété de navigation : propriété dans l'un des types d'entités impliqués dans la relation, et qui contient une ou plusieurs références vers l'autre type d'entité.

Modélisation d'une relation entre deux entités :

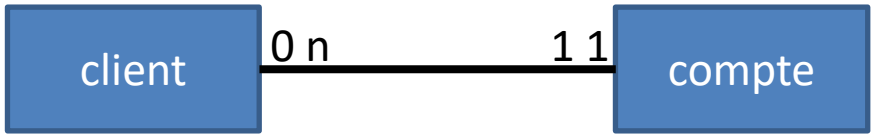
Relations entre deux entités un à plusieurs

```
public class Client {
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }

    public List<Compte> Comptes { get; set; }
}

public class Compte {
    public int CompteId { get; set; }
    public decimal Solde { get; set; }

    public int ClientId { get; set; }
    public Client Client { get; set; }
}
```



Un client a 0 ou plusieurs comptes.
Un compte appartient à un et unique client

Entité principal : Client qui est référencé par Compte
Clé principale : ClientId
Entité dépendante : Compte qui dépend de Client
Clé étrangère : Compte.ClientId, qui contient l'id d'un Client

Propriétés de navigation :

- Compte.Client référence un objet Client qui correspond à Compte.ClientId
- Client.Comptes qui contient l'ensemble des Comptes associés

Conventions utilisées :

- Les propriétés de navigation permettent à EF Core de créer les relations entre entités

EF Core ne sait pas relier les entités lorsqu'elles sont du même types

- la propriété Commande.ClientId est automatiquement comme clé étrangère si un de ces schéma est respecté :
  - <nom de la clé primaire de l'entité principale>
  - <nom de la propriété de navigation><nom de la clé primaire de l'entité principale>

Si la clé étrangère est absente, EF Core génère une propriété virtuelle, appelée **shadow property**, qui est enregistrée comme clé étrangère. Son nom respecte le deuxième schéma

Les conventions sont définies par le type **RelationshipDiscoveryConvention**, de l'assembly **Microsoft.EntityFrameworkCore**

Relations avec annotation

```
public class Compte {
    public int Id { get; set; }
    public decimal Solde { get; set; }

    public int IdClient { get; set; }
    [ForeignKey("IdClient")]
    public Client Client { get; set; }
}
```

Relations avec annotation

```
public class Client {
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }

    [InverseProperty("IdClient")]
    public List<Compte> Comptes { get; set; }
}
```

`[ForeignKey("IdClient")]` De type **ForeignKeyAttribute** définit la clé étrangère

[InverseProperty] permet d'identifier explicitement la propriété associée à l'autre extrémité de la relation. Utile dans les cas complexes où plusieurs liaisons sont faites entre les entités.

Relation entre les entités avec Fluent API :

Relations un à plusieurs avec Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Compte>()
                .HasOne(compte => compte.Client)
                .WithMany(client => client.Comptes);

    // identique à :
    //modelBuilder.Entity<Client>()
    //            .HasMany(client => client.Comptes)
    //            .WithOne(compte => compte.Client);
}
```

WithOne et WithMany sans argument indiquent qu'il existe une propriété de navigation à l'autre extrémité de la relation mais sans existence concrète.

Relations un à plusieurs avec Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Compte>()
                .HasOne(compte => compte.Client)
                .WithMany(client => client.Comptes)
                .HasPrincipalKey(client => client.ClientId)
                .HasConstraintName("Client_Compte_FK")
                .IsRequired()
                .OnDelete(DeleteBehavior.Cascade);
    // Identique à :
    //modelBuilder.Entity<Client>()
    //            .HasMany(client => client.Comptes)
    //            .WithOne(compte => compte.Client)
    //            .HasPrincipalKey(client => client.ClientId
    //            .HasConstraintName("Client_Compte_FK")
    //            .IsRequired()
    //            .OnDelete(DeleteBehavior.Cascade);
}
```

HasConstraintName("Client_Compte_FK") : nom de la contrainte

IsRequired() : valeur non nullable

OnDelete(DeleteBehavior.Cascade) : la supression du client entraîne celle des comptes liés.

Les valeurs possibles :

- DeleteBehavior.Cascade
- DeleteBehavior.SetNull
- DeleteBehavior.Restrict

Les Contraintes

Les propriétés d'une entité ne pouvant contenir la valeur « null » est considéré comme obligatoire, c'est-à-dire les types primitifs.

Les propriétés contenant des références sont par défaut facultatives.

Propriétés obligatoires ou facultatives

```
public class Client
{
    // propriétés  obligatoires.
    public int ClientId { get; set; }
    public bool Actif { get; set; }
    public int Age { get; set; }

    // Ces propriétés sont optionnelles.
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public byte[] Photo { get; set; }
}
```

Propriétés obligatoires ou facultatives

```
public class Client
{
    // propriétés  obligatoires.
    public int ClientId { get; set; }
    [Required]
    public string Nom { get; set; }

    // Ces propriétés sont optionnelles.
    public Nullable<bool> Actif { get; set; }
    public int? Age { get; set; }
    public string Prenom { get; set; }
    public byte[] Photo { get; set; }
}
```

Rendre une propriété obligatoire avec Fluent API :

Propriétés obligatoires avec Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
                .Property(c => c.Nom)
                .IsRequired();

    modelBuilder.Entity<Client>()
                .Property(c => c.Prenom)
                .IsRequired();

    base.OnModelCreating(modelBuilder);
}
```

Définir la longueur maximale :

Longueur maximale d'une propriété : annotation

```
[MaxLength(5)]
public string Nom { get; set; }
```

Longueur maximale d'une propriété : Fluent API

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Client>()
           .Property<string>(nameof(Client.Nom))
           .HasMaxLength(5);
}
```

Cette propriété est ignorée sur les types non compatibles

L'indexation accélère la recherche des valeurs mais ralentit l'écriture de la colonne indexée.
Les clés étrangères, simples ou composées sont, par défaut, indexées par EF.

Passer par Fluent API est obligatoire pour ajouter un index.

Indexation

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Client>()
           .HasIndex(c => c.Nom);
}
```

Index associé à plusieurs propriétés.

Indexation

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Client>()
           .HasIndex(c => new { c.Nom, c.Prenom });
}
```

Ajoute de la contrainte d'unicité à un index et nommage de la contrainte :

Indexation avec contrainte d'unicité

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Client>()
           .HasIndex(c => new { c.Nom, c.Prenom })
           .IsUnique()
           .HasName("IDX_Nom_Prenom");
}
```

Une clé alternative/candidate est une propriété, ou un groupe de propriétés, qui identifie de manière unique une entité.

Une contrainte d'unicité sur un ensemble de champs permettent de créer une clé principale.

Toute propriété (ou groupe de propriétés) utilisée comme clé d'entité principale dans une relation sera une de clé alternative.

Cette convention est gérée par la classe :
Microsoft.EntityFrameworkCore.Metadata.Internal.EntityTypeBuilder

La SDD ou le fournisseur peuvent générer automatiquement des valeurs pour certaines propriétés des entités

Il existe 3 comportements définis par l'énumération DatabaseGeneratedOption situé dans System.ComponentModel.DataAnnotations.Schema :

- None : aucune génération, il faut fournir la valeur

- Identity : la valeur créée lors de l'insertion d'un enregistrement généralement pour les clés primaires de type int, string, Guid ou byte[]

- Computed : la valeur est créée à l'ajout et recréée à chaque modifications de l'entité dans la SDD

La classe KeyConvention gère cette convention.

Génération par annotation

```
public class Client
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; }
}
```

Génération par Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
                .Property(client => client.Id)
                .ValueGeneratedNever();
                // génération à  l'ajout
                //.ValueGeneratedOnAdd();
                // génération à l'ajout et à la mise à jour
                //.ValueGeneratedOnAddOrUpdate();
}
```

Il est possible d'ajouter des colonnes calculées.

Une colonne calculée, est une colonne crée à partir du résultat d'un traitement SQL sur une ou plusieurs colonnes.

La fonction Fluent API, HasComputedColumnSql() permet la création de colonnes calculées

Colonne calculée

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Produit>()
        .Property(client => client.PrixTTC)
        .HasComputedColumnSql("[prix_ht] * 1.2");
}
```

La propriété PrixTTC est configuré pour correspondre au prix de la colonne prix_ht * 1,2

Si aucune valeur n'est fournie, il est possible de spécifier une valeur par défaut pour un champs qui peut être une constante ou le résultat d'une instruction SQL.

Valeur par défaut

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
                .Property(client => client.Nom)
                .HasDefaultValue("Vossough");
}
```

Valeur par défaut fournit par une instruction SQL

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
                .Property(client => client.Email)
                .HasDefaultValueSql("[Nom] + '.' + [Prenom] + '@semifir.com'");
}
```

Scaffolding

Il est possible de générer les entités à partir d'une base de données existante.

Le scaffolding est la génération de code.

Il faut installer une librairie associé au fournisseur d'accès chois et se nomme Design par convention :

Microsoft.EntityFrameworkCore.SqlServer.Design pour SQL Server

Vérifier que le package NuGet nommé **Microsoft.EntityFrameworkCore.Tools** est installé pour pouvoir lancer des commandes de génération en PowerShell.

Pour le vérifier :

- Ouvrir : menu Outils -> Gestionnaire de package NuGet -> Console du Gestionnaire
- Exécuter « get-help Scaffold-DbContext »

Syntaxe : Scaffold-DbContext [-Connection] <String> [-Provider] <String> [-OutputDir <String>] [-Context <String>] [-Schemas <String>] [-Tables <String>] [-DataAnnotations] [-Force] [-Project <String>] [-StartupProject <String>] [-Environment <String>] [<CommonParameters>]

Connection : chaîne de connexion complète de la SDD. Le nom de l'argument est optionnel si sa valeur est la première dans la chaîne d'arguments.

Provider : nom complet du fournisseur de données pour accéder à la SDD.

Exemple : **Scaffold-DbContext -Connection** "Data Source=DESKTOP-RT4EQK1\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=True" **-Provider** Microsoft.EntityFrameworkCore.SqlServer

La commande a pour résultat l'ajout de plusieurs fichiers à la racine du correspondant aux tables de la source de données, le fichier NorthwindContext.cs contient le contexte

OutputDir indique le répertoire de destination des fichiers générés "-OutputDir ./DBModel"

Context permet de changer le nom du contexte créé tel que "-Context Truc" nommera le contexte TrucContext.

Schemas : permet de spécifier les noms des autres schémas pour lesquels doivent être générés des types d'entités. "-Schemas schemaA,schemaB,..."

Script T-SQL pour déplacer une table dans un autre schéma :

```
IF (NOT EXISTS (SELECT * FROM sys.schemas WHERE name = 'monSchema'))
BEGIN
    EXEC ('CREATE SCHEMA [monSchema] AUTHORIZATION [dbo]')
END
ALTER SCHEMA monSchema TRANSFER dbo.Categories
```

Tables filtre les tables pour lesquelles du code doit être généré « -Tables user,compte »

DataAnnotations indique que le code généré devrait utiliser les data annotations tant que possible. Par défaut, les entités générée utilise Fluent API.

Force, force la régénération d'un fichier de code car si un des fichiers qui doit être écrit existe déjà, cela provoque une erreur.

Le CLI .NET sont dans le package NuGet Microsoft.EntityFrameworkCore.Tools.DotNet

Dans une console placé à la racine du projet, taper : dotnet ef

La commande dotnet ef dbcontext scaffold --help pour l'aide

dotnet ef dbcontext scaffold <connection string> <provider> --output-dir <path> --context <name> --data-annotations --force --table <schema.table> --schema <schema> --json --verbose --root-namespace <namespace>

Les deux premiers arguments sont obligatoires

- Le premier, la chaîne de connexion complète à la source de données.
- Le second le fournisseur de données

dotnet ef dbcontext scaffold "Data Source=DESKTOP-RT4EQK1\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=True" Microsoft.EntityFrameworkCore.SqlServer

output-dir est le répertoire de destination pour les fichiers générés. (alias **-o**)

context pour le nom du contexte (alias **-c**)

schema génère les types d'entités associés à un autre schéma. Command répétable pour chaque schéma à ajouter

table sélectionne les tables dont l'entité est à générer, à défaut le fait pour tous les tables. Cette commande est répétable pour chaque table à prendre en compte (alias **-t**)

data-annotations préfère les annotations à Fluent (alias **-a**)

force oblige la régénération des entités (alias **-f**)

json fais un retour sur les fichiers générés au format JSON

verbose fournit d'avantage d'informations pendant l'exécution de la commande

Les migrations

Une migration est une description de la manipulation de la structure de la SDD

Les migrations sont gérés par deux CLI : NuGet et la CLI .NET Core

- Ces outils sont dans les packages NuGet Microsoft.EntityFrameworkCore.Tools et Microsoft.EntityframeworkCore.Tools.DotNet

- Ils permettent trois opérations de base : la création, la suppression et l'application d'une migration

- Ils permettent la génération d'un script différentiel qui décrit les instructions pour aller d'une migration à une autre migration.

La création d'une migration se fait à partir d'un contexte configuré avec un fournisseur et une ou plusieurs entités qui lui sont liées.

Valeur par défaut

```
public class MigrationsContext : DbContext {
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        optionsBuilder.UseSqlServer("CHAINE DE CONNEXION");
    }
    public DbSet<Client> Clients { get; set; }
}
public class Client {
    public int Id { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
}
```

La syntaxe de la commande PowerShell pour créer une migration est :

Add-Migration [-Name] <String> [-OutputDir <String>] [-Context <String>]

La commande attend un nom pour la migration :

- Il est conseillé de le faire court en camelCase sans accent et descriptif de l'action

-**OutputDir** répertoire du projet où créer la migration

-**Context** permet de choisir un contexte si plusieurs sont présents dans le projet

L'exécution de la commande créée deux fichier:

- 20170216101641_Nom.cs : Contient les instructions de modification de la BDD
- ContextModelSnapshot.cs : Contient une description du modèle objet associé à la dernière migration créée.

Ces instructions sont implémentées respectivement dans le corps des méthodes Up et Down.

Fichier 20170216101641_Nom.cs

```
public partial class NomMigration : Migration {
    protected override void Up(MigrationBuilder migrationBuilder) {
        // instructions modifiant  la BDD
    }

    protected override void Down(MigrationBuilder migrationBuilder) {
        // instructions pour annuler les modifications de la fonction UP
    }
}
```

Fichier MigrationsContextModelSnapshot.cs

```
[DbContext(typeof(MigrationsContext))]
partial class MigrationsContextModelSnapshot : ModelSnapshot {
    protected override void BuildModel(ModelBuilder modelBuilder){
        //  création du modelBuilder

        modelBuilder.Entity("ConsoleApplicationMigrations.Client", b => {
            // liste des attributs
        });
    }
}
```

Pour la même opération avec la CLI .NET, dans le répertoire du projet lancer la commande :

- dotnet ef migration add MaMigration

Si nous relançons une commande de migration, les fichiers générés se basent sur les migrations existantes pour générer le code complétmentaire.

Pour appliquer une migration avec PowerShell :

- **Update-Database** [[-Migration] <String>] [-Context <String>]

Sans arguments, l'ensemble des migrations non déployées sont lancées.

Update-Database NomDeLaMigration : place la base de données à l'état de cette migration

Dans la BDD, la table __EFMigrationsHistory contient la liste des migrations appliquées

Avec la CLI .NET, la même opération est exécutée à l'aide de la commande suivante :

- **dotnet ef database update NomDeLaMigration**

La suppression de migration est en LIFO (Last In, First Out).

La commande PowerShell **Remove-Migration** permet de supprimer la dernière migration créée.

Attention :
Remove-Migration ne met pas à jour la BDD

Il est possible de générer des scripts de migrations absolus ou différentiels

Commande PowerShell :

- **Script-Migration** [-From <String> -To <String>] [-Idempotent] [-Context <String>]

Les arguments **From** et **To** définissent la migrations source et de destination utilisées, cela donne un script différentiel.

Si ses arguments sont absents, c'est une

Lorsque ces arguments sont définis, l'outil génère le script différentiel si ils sont absents c'est une créationcompléte pour l'état associé à la dernière migration.

-Idempotent pour générer un script pour une base de données dans un état correspondant à n'importe quelle migration. Ce script est alors à la fois différentiel et absolu.

Exemple de script absolu généré :

Fichier MigrationsContextModelSnapshot.cs

```
IF OBJECT_ID(N'__EFMigrationsHistory') IS NULL
BEGIN
    CREATE TABLE[__EFMigrationsHistory] (
       [MigrationId] nvarchar(150) NOT NULL,
       [ProductVersion] nvarchar(32) NOT NULL,
       CONSTRAINT[PK___EFMigrationsHistory] PRIMARY KEY([MigrationId])
    );
END;
GO
CREATE TABLE[client] (
    # colonnes et contraintes de la table
);
GO
INSERT INTO[__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES(N'20161208014929_AjoutEntiteClient', N'1.1.0-rtm-22752');
GO
CREATE TABLE[compte] (
    # colonnes et contraintes de la table
);
GO
CREATE INDEX[IX_Comptes_ClientId] ON[Comptes] ([ClientId]);
GO
INSERT INTO[__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES(N'20161208015333_AjoutEntiteCommande', N'1.1.0-rtm-22752');
GO
```

Requêtage avec LINQ

Adrien Vossough

LINQ (Language INtegrated-Query)

- est arrivé avec le framework .NET 3.5 en 2007.
- Cette interface permet d'interroger les sources de données compatibles de manière unifiée.
- est ensemble de classes pour le requêtage et une extensions du langage C permettant l'utilisation de mots-clés spécifiques pour créer des requêtes avec une syntaxe proche du SQL.
- est un sucre syntaxique car ses expressions sont transformées par le compilateur en appels de méthodes.
- Permet l'interrogations de différentes SDD : les collections d'objets, base de données SQL Server, les fichiers XML ou toutes source avec un fournisseur LINQ.

Avec Entity Framework Core, les fournisseurs de données sont responsables de l'implémentation de ces extensions. Le fournisseur de données SQL Server peut générer du code SQL optimisé pour le moteur avec lequel il travaille

Simple requête LINQ

Les requêtes LINQ ne gère que les collections implémentant l'interface IEnumerable<T>

- DbSet<T> implémente cette interface

Requête LINQ

```
using (NorthwindContext context = new NorthwindContext()) {
    var requete = from customer in context.Customers
                  where customer.Country == "France"
                  select customer;
}
```

Select obligatoire

Le type de retour est IQueryable<Customer>, mais il est courant d'utiliser l'inférence car la clause select peut créer des objets de type anonyme.

Requête avec l'API LINQ

```
using (NorthwindContext context = new NorthwindContext()) {
    var requete = context.Customers
                  .Where(customer => customer.Country == "France")
                  .Select(customer => customer);
}
```

Select optionnel

LINQ permet d'écrire une requête mais ne l'exécute pas, le résultat est déclenché par :

- Par une boucle **foreach** : opérateur différé
- Par le calcul d'une valeur d'agrégat (**Count, Max**, etc.) ou par de certaines méthodes (**First(),ToList(),** etc.) : opérateurs non différés

Afficher les résultats

```
foreach (var resultat in requete)
{
    Console.WriteLine(resultat.ToString());
}
```

LINQ avancé

Les opérateurs **LINQ** sont disponibles sous la forme de méthodes d'extension sur les objets de type **IEnumerable<T>.** Pour **EF**, les méthodes sont redéfinies sur le type **IQueryable<T>,** qui hérite de **IEnumerable<T>.**

IEnumerable<T> :

- **IEnumerable** pose l'infrastructure pour exécuter le code généré par les boucles **foreach**. Le code **IL** (Intermediate Language) de ces boucles utilise la méthode **GetEnumerator**() du type **IEnumerable** pour obtenir un objet capable de référencer chaque élément de la collection.
- La version générique, **IEnumerable<T>,** depuis .NET 2.0, apporte le typage fort

IQueryable<T> :

- Hérite d'IEnumerable<T>
- Ajoute est un point d'entrée pour l'extension de LINQ à l'aide de fournisseurs de données.
- La propriété Expression des objets IQueryable<T> contient un arbre d'expressions représentant une requête. Lorsque l'objet est énuméré (ex: foreach) cet arbre est passé au fournisseur lié la collection qui traitera la requête

DbSet<T>

- Implémente IQueryable<T>
- Les énumérations d'un objet IQueryable<T> entraînent plusieurs générations et exécutions de requêtes

Une projection transforme un ensemble séquencé de valeur en nouvel séquence.

LINQ définit deux opérateurs de projection : **Select** et **SelectMany**.

L'opérateur **Select** transforme chaque élément d'une collection et retourne une collection d'objets transformés. La séquence source et résultant ont le même nombre d'éléments.

La méthode **Select** accepte en paramètre une fonction de transformation.

SELECT par méthode

```
using (NorthwindContext context = new NorthwindContext()) {
    var requete = context.Customers
                         .Select((c) => new { Nom=c.ContactName, Pays = c.Country });
    // Nom et pays des clients
    foreach (var res in requete) {
        Console.WriteLine($"{res.Nom} / {res.Pays}");
    }
}
```

SELECT par mot-clé

```
var requete = from c in context.Customers
              select new { Nom = c.ContactName, Pays = c.Country};
```

SELECT SQL équivalent

```
SELECT [ContactName], [Country]
FROM [Customers] AS [c]
```

Certains éléments source contiennent des collections d'objets.

Si **Select** est utilisé pour projeter ces collections, le retour sera des listes de listes.

SelectMany permet de combiner cet ensemble pour former une séquence de destination.

Il fonctionne comme le Select, mais la fonction en paramètre doit renvoyer une collection.

SelectMany peut renvoyer un String car c'est une collection de caractère

SelectMany par méthode

```
using (NorthwindContext context = new NorthwindContext())
{
    var requete = context.Customers
            .SelectMany((client) => client.Orders);

    // id des commandes
    foreach (var resultat in requete) {
        Console.WriteLine(resultat.OrderId);
    }
}
```

Requête SQL générée

```
SELECT [client.Orders].[OrderID],
[client.Orders].[CustomerID],
[client.Orders].[EmployeeID], [client.Orders].[Freight],
[client.Orders].[OrderDate], [client.Orders].[RequiredDate],
[client.Orders].[ShipAddress], [client.Orders].[ShipCity],
[client.Orders].[ShipCountry], [client.Orders].[ShipName],
[client.Orders].[ShipPostalCode],
[client.Orders].[ShipRegion],
[client.Orders].[ShipVia], [client.Orders].[ShippedDate]

FROM [Customers] AS [client]

INNER JOIN
[Orders] AS [client.Orders] ON
[client].[CustomerID] = [client.Orders].[CustomerID]
```

Il est possible d'utiliser Select avec SelectMany pour réduire le nombre de colonnes

SelectMany par méthode

```
using (NorthwindContext context = new NorthwindContext()) {

    var requete = context.Customers.SelectMany(
        (client) => client.Orders.Select(
                (commande) => new {Id = commande.OrderId, Date = commande.OrderDate}
        )
    );

    // Id de chaque commande
    foreach (var resultat in requete) {
        Console.WriteLine(resultat.Id);
    }
 }
```

Requête SQL générée

```
SELECT [client.Orders].[OrderID], [client.Orders].[OrderDate]
FROM [Customers] AS [client]
INNER JOIN [Orders] AS [client.Orders] ON [client].[CustomerID] = [client.Orders].[CustomerID]
```

Les opérateurs de partitionnement scinde une collection et retourne un des sous-ensembles résultants.

Ils sont utilisés dans la création de données paginées.

L'opérateur Take retourne un nombre spécifié d'éléments présents en tête de la collection source.

Récupérer les 10 premières valeurs

```
using (var context = new NorthwindContext()) {
    var requete = context.Customers.Take(10);

    // afficher les 10 premiers clients
    foreach (var resultat i
        Console.WriteLine(r
    }
}
```

Requete T-SQL équivalente

```
# retourne les @__p_0 premières valeurs

SELECT TOP(@__p_0) [c].[CustomerID], [c].[Address], [c].[City],
[c].[CompanyName], [c].[ContactName], [c].[ContactTitle],
[c].[Country], [c].[Fax], [c].[Phone], [c].[PostalCode], [c].[Region]
FROM [Customers] AS [c]
```

Skip saute les X premiers éléments

Saute les 20 premières valeurs

```
using (var context = new NorthwindContext())  {
    var requete = context.Customers.Skip(20);

    foreach (var resultat in requete) {
        Console.WriteLine(resultat.ContactName);
    }
}
```

Requete T-SQL équivalente

```
SELECT [c].[CustomerID], [c].[Address], [c].[City], [c].[CompanyName],
[c].[ContactName], [c].[ContactTitle], [c].[Country], [c].[Fax],
[c].[Phone], [c].[PostalCode], [c].[Region]
FROM [Customers] AS [c]
ORDER BY @@ROWCOUNT
OFFSET @__p_0 ROWS
```

OrderBy() et **OrderByDescending()** ordonnent les éléments suivant les valeurs d'une clé donnée.
L'ordonnancement sur les objets complexe est possible s'ils implémentent l'interface **IComparable**

OrderBy : par méthode

```
using (var context = new NorthwindContext()) {
    var requete = context.Customers.OrderBy(client => client.ContactName);
}
```

Requete T-SQL équivalente

```
SELECT *
FROM [Customers] AS[client]
ORDER BY[client].[ContactName];
```

OrderBy : par mot-clé

```
var requete = from client in context.Customers
              orderby client.ContactName ascending
              select client;
```

ascending est optionnel car ascendants par défaut, sinon, **descending**

OrderBy : Autre exemple

```
var requete = context.Customers.OrderBy(client => client.CompanyName.Length);
```

Requete T-SQL équivalente

```
SELECT *
FROM[Customers] AS[client]
ORDER BY LEN([client].[CompanyName])
```

ThenBy() n'est pas défini sur **IQueryable** mais sur **IOrderedQueryable** et opère donc sur une séquence triée avec **OrderBy** ou **OrderByDescending**.

ThenBy : par méthode

```
var requete = context.Employees
                     .OrderBy(employe => employe.Country)
                     .ThenBy(employe => employe.FirstName);
```

Ordonne les employés par pays et ensuite par prénom.

ThenBy : par opérateur

```
var requete = from employe in context.Employees
              orderby employe.Country , employe.FirstName
              select employe;
```

ThenByDescending() pour le tri descendant

ThenByDescending : par opérateur

```
var requete = from employe in context.Employees
              orderby employe.Country , employe.FirstName descending
              select employe;
```

GroupBy() permet de créer une séquence contenant des collections de type **IGrouping<TKey, TValue>.**

Chaque collection regroupe des éléments partageant une même clé.

GroupBy : par méthode

```
using (var context = new NorthwindContext()) {
    var requete = context.Employees.GroupBy(employe =>
employe.Country);

    foreach (var resultat in requete) {
        // clé du groupe
        Console.WriteLine($"Clé : {resultat.Key}");

        // id des employés par pays
        foreach (var employe in resultat) {
            Console.WriteLine($"{employe.EmployeID}");
        }
    }
}
```

Requete T-SQL

```
SELECT *
FROM [Employees] AS [employe]
ORDER BY [employe].[Country]
```

GroupBy : par opérateur

```
var requete = from employe in context.Employees
                  group employe by employe.Country;
```

La clause Select provoque une erreur. Il faut créer une variable de destination pour chacun des groupes avec le mot-clé into

Distinct retire tous les doublons d'un résultat

Distinct : par méthode

```
var requete = context.Customers
                     .Select(customer => customer.Country)
                     .Distinct();
```

Except permet de retourner une séquence exempté des éléments fournis en paramètre.
Il utilise la méthode IEquatable<T>, à défaut, il utilise Equal() hérité d'Object

Except : par méthode

```
// récupére les deux premiers employés
var deuxEmployes = context.Employees.Take(2);
// récupère tous les employés sauf ceux fournis en paramètre
var requete = context.Employees.Except(deuxEmployes);
```

La requête crée est une requête SELECT simple car le filtrage se fait au niveau applicatif

Union permet de combiner deux séquences retournant des objets de même type sans les doublons.

Il détermine l'existence de doublons avec la méthode Equals de IEquatable<T> sinon la fonction Equals héritée dObject.

Union : par méthode

```
// récupère les employés 1 2 3
var employesUnDeuxTrois = context.Employees.Take(3);
// récupère les employés 3 4 5
var employesTroisQuatreCinq = context.Employees.Skip(2).Take(3);
// le résultat est 1 2 3 4 5, l'employé 3 n'est pas en doublon
var requete = employesUnDeuxEtTrois.Union(employesTroisQuatreEtCinq);
```

Intersect retourne les éléments présents dans deux séquences et filtres les autres.

Il utilise IEquatable<T> sinon Equals() hérité d'Object.

var employesUnDeuxEtTrois = context.Employees.Take(3);

Union : par méthode

```
// récupère les employés 1 2 3
var employesUnDeuxTrois = context.Employees.Take(3);
// récupère les employés 3 4 5
var employesTroisQuatreCinq = context.Employees.Skip(2).Take(3);
// Seul l'employé 3 est gardé car présent dans les 2 séquences
var requete = employesUnDeuxEtTrois.Intersect(employesTroisQuatreEtCinq);
```

Where filtre les éléments selon un prédicat passé en paramètre.

Where : par méthode

```
var clientsFrancais  = context.Customers.Where(client => client.Country == "France");
```

Requête SQL généré

```
SELECT * WHERE [client].[Country] = N'France'; # N : indique une chaîne unicode
```

Where : par méthode

```
var clientsA = context.Customers.Where(client =>
        client.Country == "France" && client.City == "Paris");

# Identique à

var clientsB = context.Customers
                    .Where(client => client.Country == "France")
                    .Where(client => client.City == "Paris");
```

Il est possible de faire des recherches avec "Joker".
La requête suivante sera transformé en requête SQL utilisant "%"

Where : par méthode

```
var clients = context.Customers.Where(client => client.City.StartsWith("P") );
```

StartsWith("P") donnera en SQL : WHERE [client].[City] **LIKE** N'**P%'**

Contains("ar") donnera en SQL : WHERE [client].[City] **LIKE** N'**%ar%'**

EndsWith("s") donnera en SQL : WHERE [client].[City] **LIKE** N'**%s'**

Where : par mot-clé

```
var clients = from client in context.Customers
                where client.Country == "France" && client.City == "Paris"
```

Les opérateurs d'agrégation retourne une valeur unique à partir d'une séquence

Count() retourne un entier correspondant au nombre d'élément trouvés

Count : par méthode

```
var nombreClients = context.Customers.Count();
```

Il est possible de lui fournir un prédicat correspondant à une clause Where

Count : par méthode

```
var nombreClients = .Count(client => client.City == "Paris");
```

LongCount() est identique à **Count**() mais retourne un Int64
La requête SQL utilisera COUNT_BIG(*) avec T-SQL

Min() recherche d'une valeur minimale dans un ensemble de données.

Min : par méthode

```
var prixBas = context.Products.Select(p => p.UnitPrice).Min();

# Identique à

var prixBas2 = context.Products.Min(p => p.UnitPrice);
```

Min() autorise des objets complexes implémentant Icomparable et la recherche se fera au niveau applicatif.

Min() avec objet complexe

```
var requete = context.Customers.Min(c => new ClientComparable {
        ContactName = c.ContactName, CompanyName = c.CompanyName
});
```

Max() fonctionne de manière identique à **Min**() mais en retournant la valeur maximale.

Sum() retourne la somme d'une propriété d'un ensemble d'enregistrements.

Il n'accepte que des valeurs numériques ou Nullable<T> (T est numérique)

Sum : par méthode

```
var prixA = context.Products.Select(p => p.UnitPrice).Sum();

# Identique à

var prixB = context.Products.Sum(p => p.UnitPrice);
```

Average() fonctionne comme somme mais retourne la moyenne d'un propriété.

Une jointure est l'association de deux ensembles par le biais d'une propriété commune.

Join() correspond à un INNER JOIN en SQL.

CollectionA.Join(CollectionB, projectionA, projectionB, projectionResultat);

La projection A et B détermine la propriété commune qui sert à la jointure.

Join : par méthode

```
var commandesEtCommerciaux = context.Orders
                .Join(context.Employees,
                    (commande) => commande.EmployeeId,
                    (employe) => employe.EmployeeId,

                    (commande, employe) => new {
                        NumCommande = commande.OrderId,
                        NomEmploye = employe.FirstName + " "+ employe.LastName
                    });
```

Join : Équivalent T-SQL

```
SELECT [commande].[OrderID], [employe].[FirstName] + [employe].[LastName]
FROM [Orders] AS [commande]
INNER JOIN [Employees] AS [employe] ON [commande].[EmployeeID] = [employe].[EmployeeID]
```

Join : par mot-clé

```
var commandesEtCommerciaux =
    from commande in context.Orders
    join employe in context.Employees on commande.EmployeeId equals employe.EmployeeId
    select new {
        NumCommande = commande.OrderId,
        NomEmploye = employe.FirstName + " "+ employe.LastName
    };
```

L'opérateur utilisé n'est pas ==, mais **equals**, qui est un mot-clé utilisé uniquement dans le cadre d'une jointure.

L'opérateur Join est déconseillé pour les relations 1 à n, il est préférable d'utiliser GroupJoin() car nous aurons un objet par jointure

La fonction GroupJoin() est proche de Join() mais pour les relations **un à plusieurs**

GroupJoin : par méthode

```
var commandesParEmploye =
    context.Employees.GroupJoin(context.Orders,
        employe => employe.EmployeeId,
        commande => commande.EmployeeId,
        (employe, commandes) => new {
            NomEmploye = employe.FirstName+ " " + employe.LastName,
            NumCommandes = commandes.Select(commande => commande.OrderId)
        });
```

GroupJoin retourne une liste d'objet contenant : un nom d'employé est une liste d'ID de commandes liées

Le résultat est factorisé, l'employé ne sera pas en doublon dans d'autres objets

GroupJoin : T-SQL

```
SELECT * FROM [Employees] AS [employe]
LEFT JOIN [Orders] AS [commande] ON [employe].[EmployeeID] = [commande].[EmployeeID]
ORDER BY [employe].[EmployeeID]
```

Le groupement se fait par le mot-clé **into** suivi d'un identifiant de variable, cette variable estde type **IEnumerable**<T>, T est le type d'éléments de la séquence jointe et est utilisable dans la suite de la requête.

GroupJoin : par mot-clé

```
var commandesParEmploye =
    from employe in context.Employees
    join commande in context.Orders on employe.EmployeeId equals commande.EmployeeId
        into commandes
    select new {
        NomEmploye = employe.FirstName + " " + employe.LastName,
        NumCommandes = commandes.Select(commande => commande.OrderId)
    };
```

Ces opérateurs peuvent être divisés en deux catégories :

- les opérateurs qui manipulent le type des enregistrements issus de la source de données
- les opérateurs qui opèrent sur le type de la collection qui englobe les résultats.

La conversion des enregistrements s'utilise pour les héritages entre modèle.

Héritage entre modèles

```
public class Personne {
    public int Id { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
}

public class Client : Personne {
    public decimal MontantTotalCommandes { get; set; }
}
```

Personne et Client sont placés sur la même table "Personne" et une colonne pour les différencier est ajoutée

Récupération des personnes

```
foreach (var personne in context.Personnes) {
        Console.WriteLine(personne.Nom);
}
```

La récupération des personnes inclus celle des clients car par héritage, ce sont aussi des personnes.

OfType() permet un filtrage par type, comme récupérer les clients d'une liste de personnes.

OfType : par méthode

```
var clients = context.Personnes.OfType<Client>();
```

OfType : Requête SQL

```
SELECT "p"."Id", "p"."Discriminator", "p"."Nom", "p"."Prenom",
"p"."MontantTotalCommandes"
FROM "Personnes" AS "p"
WHERE "p"."Discriminator" = 'Client'
```

La colonne "Discriminator" indique le type de l'enregistrement.

Équivalent de OfType avec Where Select

```
var clients = context.Personnes
                     .Where(personne => personne is Client)
                     .Select(personne => (Client)personne);
```

Cast() transtype une classe enfant en sa classe parent.

Ici, les objets Client seront récupérés et manipulés comme des objets Personne.

Cast : par méthode

```
var personnesClientes = context.Clients.Cast<Personne>();
```

Requête SQL provoquée

```
SELECT "p"."Id", "p"."Discriminator", "p"."Nom", "p"."Prenom",
"p"."MontantTotalCommandes"
FROM "Personnes" AS "p"
WHERE "p"."Discriminator" = 'Client'
```

Le type IQueryable<T> exécute les requêtes à chaque itération, ce qui est un traitement lourd.

LINQ inclut quatre opérateurs pour construite les collections d'objets IEnumerable<T>.
Rappel : Queryable<T> hérite de IEnumerable<T>.

ToArray() : crée un tableau typé d'enregistrements à partir de la collection source.

ToArray()

```
var clientsFrancais  = context.Customers
                    .Where(client => client.Country == "France")
                    .ToArray();
```

ToList() : retourne les résultatsd'une séquence IQueryable<T> en List<T>

ToList()

```
var clientsFrancais  = context.Customers
                    .Where(client => client.Country == "France")
                    .ToList();
```

ToDictionary() récupère des enregistrements au format KeyValuePair<TKey, TValue>. La clé est le résultat d'une projection et la valeur est le type de l'entité.

ToDictionary()

```
var clients = context.Customers.ToDictionary(client => client.CustomerId);
```

ToLookup() est proche de GroupBy : il génère des groupes en fonction d'une clé qui est calculée à partir d'une projection en paramètre. Les éléments ayant la même clé sont groupés.

ToLookup()

```
var clientsParPays = context.Customers.ToLookup(client => client.Country);
```

.

Les opérateurs de quantification retourne une valeur booléenne selon une condition pour tout ou partie des éléments contenus dans la séquence source.

Any() teste une condition sur les enregistrements et retourne **true** dès qu'un des éléments de la séquence respecte la condition.

Any()

```
var existeClientsFrance = context.Customers
                    .Any(client => client.Country == "France");
```

Requête T-SQL utilisée

```
SELECT CASE
    WHEN EXISTS (
        SELECT 1
        FROM [Customers] AS [client]
        WHERE [client].[Country] = N'France')
    THEN CAST(1 AS BIT) ELSE CAST(0 AS BIT)
END
```

Any() sans paramètre retourne true si la séquence n'est pas vide, sinon false.

All() retourne true si l'ensemble des éléments respecte le prédicat

Any()

```
var tousFrancais = context.Customers
                   .All(client => client.Country == "France");
```

Contains() reçoit un objet en paramètre, il retourne true si l'objet est présent dans une séquence.

Contains()

```
var client = context.Customers.Take(1).ToList()[0];

var clientPresent = context.Customers
                  .Where(client => client.Country == "France")
                  .Contains(premierClient);
```

Contains() est traité côté applicatif.

Les opérateurs d'élément retournent un élément spécifique d'une collection source. Si l'élément est introuvable, il lance une exception.

Ils possèdent tous une implémentation {Nom}OrDefault, qui renvoie null plutôt qu'une exception

First() et **FirstOrDefault**() retourne le premier élément d'une séquence correspondant à un prédicat. Sans ce prédicat, il retourne le 1er élément.

First()

```
var premierClientFrancais = context.Customers
                .First(client => client.Country == "France");
```

Requête associée

```
SELECT TOP(1) *
FROM [Customers] AS [client]
WHERE [client].[Country] = N'France'
```

Last() et **LastOrDefault**() est identique à First mais retourne la dernière valeur mais est effectué par le côté applicatif.

Single() et **SingleOrDefault**() recherchent un élément unique dans une collection source, si 0 ou plusieurs éléments sont trouvés retourne une exception (ou null si OrDefault)

Single()

```
var premiereCommande context.Orders.Take(1).Single();
```

Requête générée

```
SELECT TOP(2) [t].*
FROM (SELECT TOP(@__p_0) *FROM [Orders] AS [o]) AS [t]
```

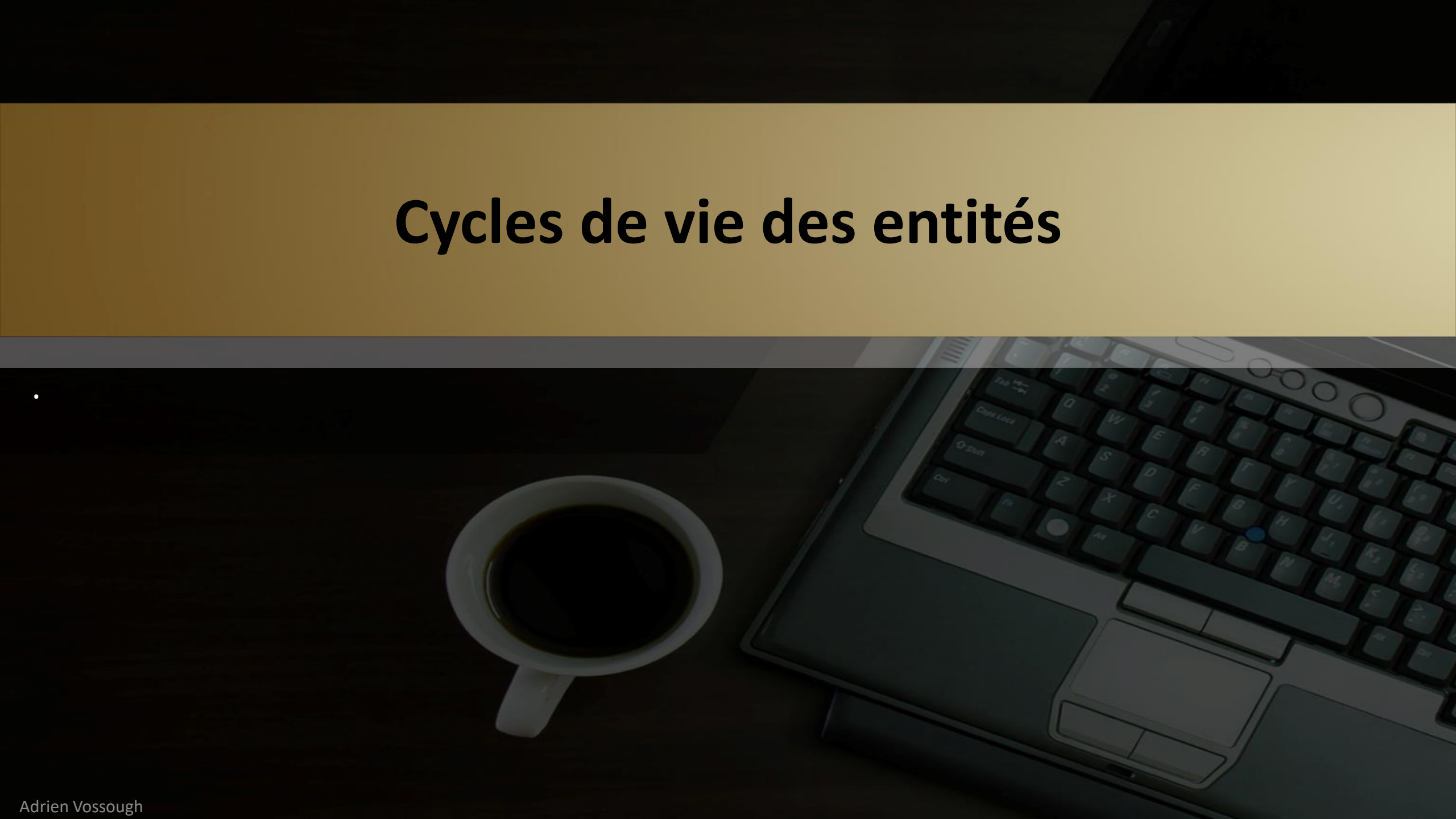
Single attend un paramètre définissant une condition que l'élément retourné doit vérifier.

Cela revient à ajouter un Where à la requête.

Single()

```
var premiereCommande = context.Orders.Single(commande => commande.OrderId == 4523);
```

Cycles de vie des entités



Adrien Vossough

Les entités ont plusieurs états qui détermine le traitement à effectuer par EF

Différents composants de EF se charge de la manipulation dont le Change Tracker ("Surveillant des modifications").

L'énumération **Microsoft.EntityFrameworkCore.EntityState** définit cinq valeurs.

Detached : Entité non reliée à un contexte de données.

- L'entité est créée par le code mais pas sauvegardée
- A été détachée de façon explicite
- L'enregistrement liée à l'entité est supprimée dans la SDD

Unchanged :

- Attachée à un contexte et son contenu est identique à celui de l'enregistrement.
- La validation d'une modification provoque le passage à l'état Unchanged.

Added :

- Lorsque l'entité a été créée et ajoutée au contexte.
- Aucun enregistrement associé n'existe dans la source de données

Modified : Entité liée à un contexte pour laquelle une demande de modification a été faite.

Deleted : Demande de suppression d'une entité liée un contexte de données.

Exemple de code sur l'état d'une entité

Changement d'état d'une entité

```
var client = new Customers(){
    CustomerId = "PABEL",
    Address = "rue de Paris",
    City = "Lille",
    Country = "France"
};
// entité liée au contexte
context.Add(client); // statut passe à Added
context.SaveChanges(); // statut passe à Unchanged
client.PostalCode = "59800";
context.Update(client); // statut passe à Modified
context.SaveChanges(); // statut passe à Unchanged
context.Remove(client); // statut passe à Deleted
context.SaveChanges(); // statut passe à Detached
```

Pour récupérer l'état :

```
EntityEntry clientEntry = context.ChangeTracker.Entries<Customers>().Single();
Console.WriteLine(clientEntry.State);
```

Add() ajoute une entité au contexte. L'entité est considérée comme n'existant pas.

L'état de l'entité ajoutée par un appel à **Add**() est **Added**.

Si l'entité est déjà présente au niveau de la SDD, cela déclenche une exception DbUpdateException à la validation des changements.
Le message sera : Violation of PRIMARY KEY constraint 'PK_Table'. Cannot insert duplicate key in object 'dbo.table'. The duplicate key value is (PABEL). The statement has been terminated.Update

Update() passse l'entité l'état **Modified**. La tentative de validation d'une mise à jour sur une entité inexistante dans la SDD donne une erreur due à un accès concurrent.

Remove() marque une entité comme à supprimer qui passe à l'état **Deleted**. Si l'entité est absente de la SDD, une exception **DbConcurrencyException** est levée.

Remove() sur un objet Added le fait passer à l'état **Detached**.

Les manipulations peuvent se faire sur les collections DbSet<T> du contexte, tel :
context.Customers.Add(client);

Le composant **Change Tracker** a la responsabilité de détecter les changements sur les entités qu'il surveille.

Il est exposé par la propriété **ChangeTracker** du **DbContext**

Propriété "**AutoDetectChangesEnabled**" :

- La détection automatique des changements permet de valider les modifications apportées à des entités qui n'ont pas été explicitement marquées comme modifiées.
- La propriété active (valeur par défaut) ou de désactive

Si **AutoDetectChangesEnabled** est à false : l'oublie d'ajouter un objet modifié à update ne sera pas automatiquement pris en compte.
Les objets enfants ne seront pas non plus insérés **s'ils ne sont pas ajoutés explicitement au contexte**

DetectChanges() du Change Tracker lance une détection manuelle

Détection manuelle

```
context.ChangeTracker.AutoDetectChangesEnabled = false;

context.Add(client);
context.SaveChanges();
client.PostalCode = "59800";

context.ChangeTracker.DetectChanges()

context.SaveChanges();
```

QueryTrackingBehavior : Les entités retournées par des requêtes LINQ sont surveillées par le Change Tracker. Cette propriété du Change Trackerpermet de contrôler ce comportement à l'aide de deux valeurs :

- **QueryTrackingBehavior.TrackAll** : valeur par défaut
- **QueryTrackingBehavior.NoTracking** : empêche le tracking par défaut et peut éviter les sauvegardes par erreur et alléger les temps d'exécution.

La stratégie de détection des changements permet de définir la façon dont les changements sont détectés.

Change Tracker compare les valeurs initiales et actuelles de l'entité et définit son état en fonction du résultat de cette opération. Cette stratégie est appelée **snapshot**.

La stratégie de détection se fait au niveau de **OnModelCreating()** du contexte. Il faut utiliser la méthode **HasChangeTrackingStrategy**() du **ModelBuilder** et lui passer une valeur de l'énumération **ChangeTrackingStrategy**.

Changement de stratégie

```
protected override void OnModelCreating(ModelBuilder builder) {
    builder.HasChangeTrackingStrategy(ChangeTrackingStrategy.ChangedNotifications);
    base.OnModelCreating(modelBuilder);
}
```

L'énumération ChangeTrackingStrategy a quatre valeurs :

Snapshot : valeur par défaut de la stratégie de détection

Il est fait une comparaison "avant/après" sur les propriétés mappées de l'entité, si une valeur est différente, l'entité passe en Modified

Récupérer les valeurs courantes et anciennes

```
var entityEntry = context.ChangeTracker.Entries<Personne>().First();
var valeursCourantes = entityEntry.CurrentValues;
var valeursOrigines = entityEntry.OriginalValues;
```

ChangedNotifications, La logique de décision de Change Tracker pour déterminer l'état est placée dans l'entité.

Toutes les entités doivent implémenter **INotifyPropertyChanged**, et déclenchent l'événement PropertyChanged, le Change Tracker, qui est abonné à cet événement, passe l'état des entités à Modified.

Implémentation de INotifyPropertyChanged

```
public class Personne : INotifyPropertyChanged {
 public event PropertyChangedEventHandler PropertyChanged;
 public int Id { get; set; }
 private string _nom;
 public string Nom {
    get { return _nom; }
    set {
      if (value != _nom) {
          _nom = value;
          OnPropertyChanged(nameof(Nom));
      }
    }
 }

  protected void OnPropertyChanged(string propertyName = null) {
    if (this.PropertyChanged != null) {
     PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
     }
```

Changement de stratégie en ChangedNotifications

```
protected override void OnModelCreating(ModelBuilder builder) {
    builder.HasChangeTrackingStrategy(ChangeTrackingStrategy.ChangedNotifications);
    base.OnModelCreating(modelBuilder);
}
```

ChangingAndChangedNotifications comme ChangedNotifications, ce sont les entités qui indique le changement.

Avec ChangedNotifications et Snapshot, la valeur originale des propriété est enregistrée lorsque l'entité est ajoutée au contexte ou que son état passe à Unchanged suite à une validation.

Avec ChangingAndChangedNotifications, seules certaines propriétés ont leur valeur initiale enregistrée, cette opération est déclenchée unitairement lorsque l'entité émet un événement PropertyChanging.

Les entités doivent implémenter les interfaces INotifyPropertyChanged et INotifyPropertyChanging.

Implémentation de INotifyPropertyChanging, INotifyPropertyChanged

```
public class Personne : INotifyPropertyChanging, INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;
    public event PropertyChangingEventHandler PropertyChanging;
    public int Id { get; set; }

    private string _nom;
    public string Nom {
        get { return _nom; }
        set {
            if (value != _nom) {
                OnPropertyChanging(nameof(Nom));
                _nom = value;
                OnPropertyChanged(nameof(Nom));
            }
        }
    }

    private string _prenom;
    public string Prenom {
        get { return _prenom; }
        set {
            if (value != _prenom) {
                OnPropertyChanging(nameof(Prenom));
                _prenom = value;
                OnPropertyChanged(nameof(Prenom));
            }
        }
    }
    // Fin de la classe
```

```
    // début de la classe
    protected void OnPropertyChanging(string propertyName = null) {
        if (this.PropertyChanging != null)
            PropertyChanging(this, new PropertyChangingEventArgs(propertyName));
    }

    protected void OnPropertyChanged(string propertyName = null){
        if (this.PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

ChangingAndChangedNotificationsWithOriginalValues est une combinaison de **ChangedNotifications** et **ChangingAndChangedNotifications**.

La détection de changements est dans les entités.

Les valeurs originales des propriétés de l'entité sont enregistrées unitairement à chaque déclenchement d'un événement **PropertyChanging**, et la modification de l'état de l'entité est effectuée suite à la réception d'un événement **PropertyChanged** par le **Change Tracker**.

Les entités doivent toutes implémenter **INotifyPropertyChanged** et **INotifyPropertyChanging**.

Attacher et détacher des entités

- Les entités peuvent être gérées par différents contextes.
- Une entité attaché à un contexte est sous surveillance du Change Tracker de celui-ci
- Les entités retournées par LINQ sont par défaut attachées au contexte d'exécution.
- Si le contexte est fermé est l'entité perdure, elle est détachée et peut être rattachée à un autre contexte avec **DbContext.Attach()**, l'entité est à l'état Unchanged.
- Dans le cas où l'entité a été modifiée hors contexte, il faut le passer en Modified manuellement.
- Les entités dépendantes (entité reliée à une entité) sont prises en compte au rattachement.

Changer une entité de contexte

```
Customers client = null;

using (var context = new NorthwindContext()) {
    client = context.Customers.First(c => c.CustomerId == "IDaPlacerICI"); // client attaché à ce contexte
} // fermeture du contexte


using (var context = new NorthwindContext()) {
    client.City = "Paris";
    context.Attach(client); // client attaché au nouveau contexte
    var entry = context.Entry(client);
    entry.State = EntityState.Modified; // passage à Modified
}
```

Validation des changements

La modification d'une entité par **Add**(), **Update**() et **Remove**() n'est validée que par la méthode **SaveChanges**().

SaveChanges()

- Lance la détection des changements, si **AutoDetectChangesEnabled** du Change Tracker est à true.
- La génération de requêtes dépend de l'état de chaque entité.
  - Added, donnera une requête INSERT.
  - Modified, donnera requête UPDATE
  - Deleted, donnera une requête DELETE.
  - Unchanged et Detached ne produisent aucune requête
- Les requêtes sont effectuées dans une transaction implicite (dépend de la SDD)
- Les entités passent à Unchanged ou Detached

Une suppression peut produire une suppression en cascade ou de mise à jour d'enregistrements dépendants.

Ces actions sont faites avant la suppression de l'enregistrement demandée: les entités dépendantes passent à Deleted, ou la clé étrangère prend la valeur null.

Intégrité des données

L’intégrité des données assure la cohérence de l'information, certaines BDD intègrent ce genre de gestion, d'autres non, cela passe donc par la partie applicative.

La concurrence est lorsque plusieurs éléments veulent modifier une même ressource.

Par défaut, le dernier enregistrement écrase les autres modifications.

Des informations modifiées par un utilisateur A seront parfois réinitialisées par un utilisateur B ce qui est généralement problématique

Les manipulations de données de manière optimiste :

Basées sur le fait que deux utilisateurs ne modifie pas le même enregistrement simultanément, et si ce cas se produit, une exception est levée.

Son implémentation est assurée par deux éléments qui peuvent être utilisés séparément : les jetons d'accès concurrents (concurrency tokens) et l'horodatage.

Jetons d'accès concurrents :

Propriétés définies à vérifier lors de la mise à jour d'un enregistrement. Lorsqu'un jeton est modifié, l'enregistrement est considéré comme modifié.

Lors d'accès simultanés, la seconde validation met à jour un enregistrement qui n'existe plus dans son état initial et une exception est levée.

La déclaration d'un jeton d'accès concurrent par l'annotation ConcurrencyCheck.

Déclaration d'un jeton d'accès par annotation

```
public class Personne{
    public int Id { get; set; }

    [ConcurrencyCheck]
    public string Nom { get; set; }
}
```

Requête utilisée

```
UPDATE [Personnes] SET [Nom] = @p0
WHERE [Id] = @p1 AND [Nom] = @p2;
```

Déclaration d'un jeton d'accès par Fluent API

```
protected override void OnModelCreating(ModelBuilder builder){
    builder.Entity<Personne>()
                .Property<string>(nameof(Personne.Nom))
                .IsConcurrencyToken();

    base.OnModelCreating(builder);
}
```

L'horodatage intègre à chaque enregistrement un champ contenant la date de dernière modification. La valeur est générée automatiquement lors de l'ajout ou de la mise à jour d'un enregistrement.

Ce n'est pas obligatoirement un timestamp est pourrait être un entier incrémenté, cela dépend de l'implémentation du fournisseurs de données.

Déclaration d'un horodatage d'accès par annotation

```
public class Personne{
    public int Id { get; set; }
    public string Nom { get; set; }

    [Timestamp]
    public byte[] JetonHorodatage { get;
}
```

Déclaration d'un horodatage d'accès par Fluent API

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Personne>()
                .Property(p => p.JetonHorodatage)
                .ValueGeneratedOnAddOrUpdate()
                .IsConcurrencyToken();
}
```

La concurrence fonctionne comme un jeton de concurrence, une modification en parallèle provoquera une exception " **DbUpdateConcurrencyException**".

Le mode d'accès pessimiste verrouille les enregistrements en cours de modification. Elle nécessite le maintien d'une connexion à la SDD.

Le blocage d'une donnée peut bloquer plusieurs utilisateurs et si l'information n'est pas libérée, cela provoque un deadlock.

Ce mode d'accès est à éviter sauf cas particulier.

Une transaction est un groupe de requêtes, si l'une d'elle n'aboutit pas, cela annule l'ensemble.

EF Core gère les transactions sous deux formes : implicites et explicites.

Les transactions implicites sont utilisées lorsque aucune transaction explicite n'est fournie pour la validation de modifications. Elles ont une portée limitée à une seule validation de changements.

Transaction implicite

```
var client1 = new Customers() {CustomerId = "AVOS", CompanyName = "Semifir"};

var client2 = new Customers() {CustomerId = "AVM", CompanyName = "Gnuside"};

context.Add(client1);
context.Add(client2);

try {
   context.SaveChanges(); // si une exception est levée aucun client n'est sauvegardé
} catch (Exception ex) {
   Console.WriteLine(ex.InnerException.Message);
}
```

Les transactions explicite peuvent encapsuler plusieurs validations.

C'est l'implémentation par le fournisseur de données de IDbContextTransaction qui est chargé de la gestion de transaction.

La fonction BeginTransaction() de la propriété DbContext.Database fournit l'objet transacionnel.

Transaction explicite

```
using (var transaction = context.Database.BeginTransaction()) {
   // code transactionnel
}
```

Transaction explicite

```
using (var context = new NorthwindContext())
using (var transaction = context.Database.BeginTransaction()) {
    try {
        var client1 = new Customers() {
            CustomerId = "CARAL",
            CompanyName = "Caramelyon"
        };
        var client2 = new Customers() {
            CustomerId = "FRANS",
            CompanyName = "France Saucisson"
        };

        context.Add(client1);
        context.SaveChanges();

        context.Add(client2);
        context.SaveChanges();

        transaction.Commit(); // validation de la transaction
    }
    catch (Exception ex) {
        Console.WriteLine(ex.InnerException.Message);
     }
}
```

Si une exception est lancée, c'est l'ensemble des requêtes qui sont annulées

La méthode catch appelle implicitement transaction.Rollback();

Modélisation avancée

Adrien Vossough

Les séquences sont des objets de base de données générant une valeur numérique.

Création d'une séquence

```
public class SequencesContext : DbContext {
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        optionsBuilder.UseSqlServer("CHAINE DE CONNEXION");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        modelBuilder.HasSequence("Sequence_DbRecordNumber");
    }
}
```

Code généré

```
CREATE SEQUENCE [Sequence_DbRecordNumber]
START WITH 1
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
NO CYCLE;
```

CREATE SEQUENCE n'existe pour SQL Server que depuis la version 2012.

Cinq paramètres sont possible pour créer une séquence en SQL.

Création d'une séquence

```
modelBuilder.HasSequence("Sequence_DbRecordNumber")
    .StartsAt(3)  // valeur initiale de type long. Par défaut 1
    .IncrementsBy(2)  // de type int, incrémentation de 2 en 2. Par défaut 1
    .HasMin(2)  // valeur minimale et maximale de type long
    .HasMax(12345)
    .IsCyclic(true); // Indique qu'il faut reprendre à la valeur minimale si le max est atteint

// IsCyclic(false); lance une exception si le max est atteint
```

Il existe une forme générique qui détermine le type généré :
modelBuilder.HasSequence<int>("Sequence")

Utilisation d'une séquence

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Ville>()
        .Property(nameof(Ville.NumeroSeq)) // la séquence donnera la valeur par defaut
        .HasDefaultValueSql("NEXT VALUE FOR Sequence_DbRecordNumber");
}
```

Il est possible de gérer l'héritage dans la SDD.

Les fournisseurs de données relationnels utilise la stratégie nommée Table Per Hierarchy .

L’assembly **Microsoft.EntityFrameworkCore.Relational** propose une implémentation pour les fournisseurs de données.

L'héritage est géré sous forme d'une unique table comprenant l'ensemble des attributs, une colonne supplémentaire est ajouté pour identifier le type.

Héritage

```
public class Personne {
    public int Id { get; set; }
    public string Nom { get; set; }
}

public class Client : Personne {
    public decimal Solde { get; set; }
}
public class InheritanceContext : DbContext{
    public DbSet<Personne> Personnes { get; set; }
    public DbSet<Client> Clients { get; set; }
    // reste du code
}
```

Table résultante

Personnes			
Id	Nom	Solde	Discriminator

Discriminator contiendra la valeur :
Client ou Personne

Modifier le champ Discriminator

```
protected override void OnModelCreating(ModelBuilder modelBuilder){
    modelBuilder.Entity<Personne>()
        .HasDiscriminator("type",typeof(int)) // change le nom et le type
        .HasValue(nameof(Personne), 0)  // modifie la valeur
        .HasValue(nameof(Client), 1);
}
```

Contrairement à EF, EF Core ne peut pas encore mapper des prostocks avec des méthodes.

Utilisation d'une procédure stockée

```
var clients = context.Customers.FromSql("EXECUTE Procedure {0}", "Valeur").ToList();
```

Si les données retournées ne correspondent pas à un type d'entité associé au modèle objet, il faut passer par ADO.NET

Il est conseillé d'encapsuler les procédures stockées dans des méthodes pour les prochaines évolutions d'EF Core.

Les shadow properties sont des propriétés associées au modèle sans implémentation au niveau du type.

Elles sont gérées par le Change Tracker, qui conserve les valeurs associées et leur état.

Les shadow properties permettent l'écriture de code générique, applicable à n'importe quelle entité respectant une condition particulière.

Lorsqu'une relation est découverte à la création du modèle mais la clé étrangère correspondante n'est pas trouvée au niveau de l'entité dépendante, EF enregistre une shadow property pour remplir ce rôle.

Création automatique de shadow properties par EF

```
public class Context : DbContext {
    public DbSet<Personne> Personnes { get; set; }
    public DbSet<Pays> Pays { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder){
        // CODE
    }
}

public class Personne {
    public int Id { get; set; }
    public string Nom { get; set; }
    public Pays Pays { get; set; }
}

public class Pays{
    public int Id { get; set; }
    public string Nom { get; set; }
}
```

Model équivalent géré par EF

```
public class Personne {
    public int Id { get; set; }
    public string Nom { get; set; }
    public Pays Pays { get; set; }
    public int PaysId { get; set; }
}
```

Récupérer les propriétés d'une entité

```
var proprietes = context.Model
      .FindEntityType(typeof(Personne).FullName)
      .GetProperties();
```

Les opérations de la convention peuvent être exécutées explicitement :

- enregistrer la shadow property avec l'id du pays
- configurer l'entité pour utiliser cette propriété comme clé étrangère.

L'enregistrement de la shadow property est effectué à l'aide d'un appel à la fonction Property, définie par le type EntityBuilder, à laquelle sont fournis le type et le nom associés à la propriété.

Création d'un shadow property

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Personne>(entity => {
        // Création de la shadow property
        entity.Property<int?>("PaysId");
        // Équivalent a :
        // entity.Property(typeof(int?), "PaysId");

        // shadow property comme clé étrangère
        entity.HasOne(personne => personne.Pays)
              .WithMany()
              .HasForeignKey("PaysId");
    });
}
```

Propriétés pour groupe d'entités : Garder la date de dernière modification de chaque enregistrement.

Entités

```
public class Personne : IUpTime{
    public int Id { get; set; }
    public string Nom { get; set; }
    public Pays Pays { get; set; }
}

public class Pays {
    public int Id { get; set; }
    public string Nom { get; set; }
}

public class Achat : IUpTime{
    public int Id { get; set; }
    public decimal Prix { get; set; }
}
// sert de marqueur
public interface IUpTime {
}
```

Génération auto. d'un shadow property

```
public class Context : DbContext {
  public DbSet<Personne> Personnes { get; set; }
  public DbSet<Commande> Achats { get; set; }
  public DbSet<Pays> Pays { get; set; }

  protected override void OnConfiguring(DbContextOptionsBuilder optBuilder) {
        // CODE
  }

  protected override void OnModelCreating(ModelBuilder modelBuilder) {
    var typesUpTime = modelBuilder.Model  // récupération des entités marquées
          .GetEntityTypes()
          .Where(type => typeof(IUpTime).IsAssignableFrom(type.ClrType));

    foreach (var type in typesUpTime) {
        modelBuilder.Entity(type.ClrType, entity => {
            entity.Property<DateTime>("LastUpdate")
                  .ValueGeneratedOnAddOrUpdate()
                  .HasComputedColumnSql("GETDATE()");
        });
    }
  }
}
```