

.NET et l'accès aux données

ADO.NET

Depuis sa toute première version, la bibliothèque de classes fournie par la plateforme .NET dispose d'une brique dédiée à la récupération d'informations enregistrées en base de données, ainsi qu'à leur manipulation et à la persistance des modifications qui leur sont apportées. Cet ensemble d'API a pour nom ADO.NET, en référence à l'API ADO (*ActiveX Data Objects*) utilisée en conjonction avec les technologies antérieures à la plateforme .NET.

Bien que leurs noms soient similaires, des différences fondamentales séparent ces deux technologies. Une des plus importantes se trouve dans la manière dont sont définies les interactions entre le cœur de la librairie et les sources de données extérieures. ADO utilise un mécanisme basé sur COM (*Component Object Model*) pour interagir avec des drivers natifs, installés sur la machine. Ceci implique que l'utilisation d'une source de données nécessite, au mieux, d'installer un pilote natif, et dans un cas moins glorieux, de l'écrire avec un langage comme C ou C++, puis de l'installer. Les liens entre les sources de données et le cœur d'ADO.NET sont logés dans des librairies écrites en code managé et respectant un contrat prédéfini : les fournisseurs de données. Ce point permet à cette brique logicielle d'être facilement étendue pour supporter de nouvelles sources de données.

Chaque fournisseur de données encapsule les différents éléments essentiels pour les interactions avec une source de données au sein de quelques classes centrales. Pour le fournisseur de données SQL Server intégré au framework .NET, par exemple :

- Connexion : `SqlConnection`, classe héritée de `DbConnection`.
- Exécution de requêtes : `SqlCommand`, classe héritée de `DbCommand`.
- Paramètres de requêtes : `SqlParameter`, classe héritée de `DbParameter`.
- Transactions : `SqlTransaction`, classe héritée de `DbTransaction`.
- Curseur sur le résultat d'une requête `SELECT` : `SqlDataReader`, classe héritée de `DbDataReader`.
- Collection d'enregistrements : `DataTable`.
- Liaison directe entre la source de données et une `DataTable` : `SqlDataAdapter`, classe héritée de `DbDataAdapter`.

Considérons la table `Clients` définie à l'aide du script SQL suivant :

```
CREATE TABLE [dbo].[Clients](
    [NumClient] [int] NOT NULL PRIMARY KEY,
    [Adresse1] [nvarchar](200) NULL,
    [Adresse2] [nvarchar](200) NULL,
    [Adresse3] [nvarchar](200) NULL,
    [CodePostal] [nvarchar](5) NULL,
    [Nom] [nvarchar](80) NOT NULL,
    [Prenom] [nvarchar](80) NOT NULL,
    [Ville] [nvarchar](100) NULL,
    [MontantTotalAchats] [decimal](12, 2) NULL
)
```

Ce qui suit montre un exemple d'utilisation d'ADO.NET pour la lecture et la mise à jour de données en provenance de cette table. Le fournisseur de données choisi pour l'implémentation est SQL Server, aussi est-il nécessaire d'ajouter une clause using adaptée dans le fichier de code.

```
using System.Data.SqlClient;
```

```
public static void Main(string[] args)
{
    string chaineConnexion = "<ma chaine de connexion>";
    string requeteSelect = @"SELECT * FROM Clients";
    string requeteUpdate = @"UPDATE Clients SET MontantTotalAchats =
MontantTotalAchats + @montant";

    //Affichage de l'identifiant, du nom complet et
    //du montant des achats de tous les clients
    using (var connection = new SqlConnection(chaineConnexion))
    using (var command = connection.CreateCommand())
    {
        command.CommandText = requeteSelect;

        using (var dataReader = command.ExecuteReader())
        {
            while (dataReader.Read())
            {
                var numClient = dataReader.GetInt32(
                    dataReader.GetOrdinal("NumClient"));
                var nom = dataReader.GetString(
                    dataReader.GetOrdinal("Nom"));
                var prenom = dataReader.GetString(
                    dataReader.GetOrdinal("Prenom"));
                var montantAchats = dataReader.GetDecimal(
                    dataReader.GetOrdinal("MontantTotalAchats"));

                Console.WriteLine($"Client n° {numClient} - {prenom} {nom} :
{montantAchats}€");
            }
        }
    }

    //Augmentation de 10€ du montant pour tous les clients
    using (var connection = new SqlConnection(chaineConnexion))
    using (var command = connection.CreateCommand())
    {
        var paramMontant = command.CreateParameter();
        paramMontant.DbType = System.Data.DbType.Decimal;
        paramMontant.ParameterName = "@montant";
        paramMontant.Value = 10;

        command.CommandText = requeteUpdate;

        command.ExecuteNonQuery();
    }
}
```

```
}  
}
```

La complexité de ce code réside en partie dans la problématique de typage des données remontées par l'objet `dataReader`, de type `SqlDataReader`. Pour manipuler des données dont le type est plus précis qu'object, il faut passer par une étape de transtypage, qu'il soit manuel ou encapsulé par les méthodes `GetDecimal`, `GetString`, etc. L'écriture de ce type de code, en plus d'apporter de la complexité, peut également engendrer des erreurs, particulièrement au niveau des chaînes de caractères (requête SQL et noms de champs). De plus, il a peu d'intérêt d'un point de vue fonctionnel et accapare des ressources qui pourraient être utilisées pour apporter de la valeur métier aux applications.

Avec ADO.NET, les développeurs ont les clés pour développer des applications robustes faisant une utilisation intensive des bases de données. Les opérations manuelles de typage fort des données issues de la base de données et l'écriture de requêtes parfois complexes lors de l'ajout ou de la modification d'enregistrements sont néanmoins problématiques. Ces éléments induisent un coût non négligeable en termes de temps de développement et de maintenance pour les projets basés sur cette technologie.

Mappage objet-relationnel

Les faiblesses du modèle traditionnel de manipulation des données, tel qu'il est implémenté dans de nombreuses technologies (.NET, Java, PHP, etc.), ont poussé de nombreux acteurs du monde du développement logiciel à l'élaboration d'un modèle différent, permettant de manipuler les données en suivant le paradigme de la programmation orientée objet (POO) : le mappage objet-relationnel (ORM ou *Object/Relational Mapping*).

Cette technique vise à transformer de manière automatique des données issues de sources de données externes, comme des bases de données, en objets utilisables dans le langage cible. La manipulation de ces objets permet également de générer et d'exécuter de manière automatique les instructions nécessaires à la répercussion des modifications dans la source de données. Ainsi, les actions associées à la source de données sont entièrement encapsulées derrière une façade objet, simplifiant les traitements, et par là même, réduisant les coûts associés au développement d'une solution logicielle couplée à une ou plusieurs sources de données.

Avec l'utilisation d'une solution de mappage objet-relationnel, l'ensemble des enregistrements d'une table peut être représenté par une collection d'objets. Cette collection ne contient en réalité aucune donnée, mais fait l'interface vers la base de données : lorsque l'on effectue une recherche dans la liste, une requête SQL de sélection est générée et exécutée, et les enregistrements retournés sont automatiquement convertis en objets fortement typés.

Considérons le cas d'une table nommée `Client` dont la structure est définie à l'aide de l'instruction SQL suivante :

```
CREATE TABLE Client(
    NumClient int PRIMARY KEY,
    Prenom varchar(50) NOT NULL,
    Nom varchar(50) NOT NULL,
    Adresse1 varchar(100) NOT NULL,
    Adresse2 varchar(100) NOT NULL,
    CodePostal varchar(5) NOT NULL,
    Ville varchar(50) NOT NULL,
    DateNaissance date NULL,
    MontantTotalAchats decimal(10, 2) NOT NULL
)
```

Chacun des enregistrements issus de la table Client pourrait avoir pour modèle la classe C# suivante :

```
public class Client
{
    public int NumClient { get; set; }

    public string Prenom { get; set; }

    public string Nom { get; set; }

    public string Adresse1 { get; set; }

    public string Adresse2 { get; set; }

    public string CodePostal { get; set; }

    public string Ville { get; set; }

    public DateTime? DateNaissance { get; set; }

    public decimal MontantTotalAchats { get; set; }
}
```

En plus de simplifier la lecture des informations issues d'une source de données, le mappage objet-relationnel permet également de faire remonter des modifications (ainsi que des ajouts ou suppressions d'enregistrements) vers la source de données en générant et en exécutant les requêtes SQL correspondantes. La technique de l'ORM offre ainsi au développeur la possibilité de simplifier le code d'accès aux données, tout en lui laissant la possibilité de se concentrer sur les portions de code à forte valeur ajoutée.

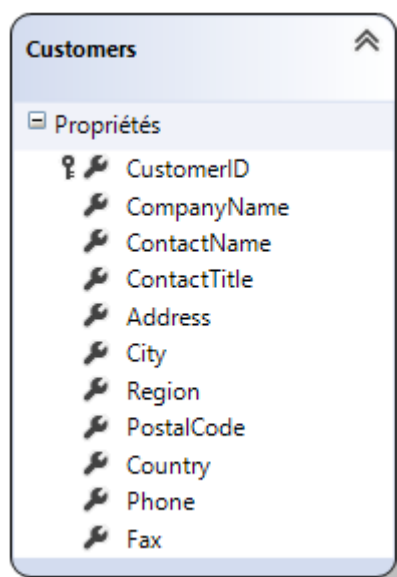
LINQ to SQL

Consciente des défauts d'ADO.NET et de l'évolution en cours au niveau des techniques d'accès aux données, l'équipe en charge du langage C# a travaillé sur la simplification de l'accès aux données par l'intégration du requêtage au langage C#. Le résultat de cet effort est une bibliothèque de mappage objet-relationnel dévoilée à la fin de l'année 2005 sous le nom de DLINQ, et renommée par la suite en LINQ to SQL, qui fait partie intégrante du framework .NET depuis sa version 3.5.

LINQ to SQL tire son nom de la brique logicielle qui lui est associée pour l'écriture de requêtes de sélection : LINQ (*Language-INtegrated Query*). Cette librairie permet de manipuler des données par l'utilisation d'éléments intégrés au langage C# (ou VB.NET), et de manière complètement agnostique : la source des données pourrait aussi bien être un fichier CSV (*Comma-Separated Values*) ou XML, une collection .NET en mémoire ou une base de données.

LINQ to SQL se positionne justement sur ce dernier créneau en fournissant une infrastructure logicielle qui permet de retranscrire les instructions LINQ en instructions SQL compréhensibles par une base de données SQL Server (uniquement).

Les outils associés à LINQ to SQL (SQLMetal et le designer intégré à Visual Studio) facilitent la génération d'un fichier au format XML qui décrit les relations entre les objets .NET et les éléments de la source de données. À partir du contenu de ce fichier, ils peuvent ensuite passer à l'étape de génération du code C#/VB.NET exploitable au sein de l'application. Ainsi, pour un modèle simple, dans lequel une seule table de la base de données Northwind de Microsoft est intégrée, au moins deux fichiers sont créés : un fichier de description (extension .dbml) et un fichier de code contenant deux classes. Avec l'utilisation du designer de Visual Studio, deux fichiers supplémentaires liés à l'aspect visuel du modèle sont également créés.



Le contenu du fichier de description est le suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="Northwind" Class="DataClasses1DataContext"
  xmlns="http://schemas.microsoft.com/linqtosql/dbml/2007">
  <Connection Mode="AppSettings"
    ConnectionString="Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Northwind;
    Integrated Security=True"
    SettingsObjectName="ConsoleApplication1.Properties.Settings"
    SettingsPropertyName="NorthwindConnectionString"
    Provider="System.Data.SqlClient" />
  <Table Name="dbo.Customers" Member="Customers">
    <Type Name="Customers">
      <Column Name="CustomerID" Type="System.String"
        DbType="NChar(5) NOT NULL" IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="CompanyName" Type="System.String"
        DbType="NVarChar(40) NOT NULL" CanBeNull="false" />
      <Column Name="ContactName" Type="System.String"
        DbType="NVarChar(30)" CanBeNull="true" />
```

```

    <Column Name="ContactTitle" Type="System.String"
        DbType="NVarChar(30)" CanBeNull="true" />
    <Column Name="Address" Type="System.String"
        DbType="NVarChar(60)" CanBeNull="true" />
    <Column Name="City" Type="System.String"
        DbType="NVarChar(15)" CanBeNull="true" />
    <Column Name="Region" Type="System.String"
        DbType="NVarChar(15)" CanBeNull="true" />
    <Column Name="PostalCode" Type="System.String"
        DbType="NVarChar(10)" CanBeNull="true" />
    <Column Name="Country" Type="System.String"
        DbType="NVarChar(15)" CanBeNull="true" />
    <Column Name="Phone" Type="System.String"
        DbType="NVarChar(24)" CanBeNull="true" />
    <Column Name="Fax" Type="System.String"
        DbType="NVarChar(24)" CanBeNull="true" />

</Type>
</Table>
</Database>

```

Le code source C# généré par l'exécution de l'outil MSLinkToSQLGenerator (automatique lors de la compilation avec Visual Studio) est, quant à lui, beaucoup plus long, puisqu'il fait presque 400 lignes, commentaires et clauses `using` inclus.

Ici, il est sensiblement diminué, puisqu'il est amputé du code associé aux propriétés `Address`, `City`, `Region`, `PostalCode`, `Country`, `Phone` et `Fax`. Le contenu et le mode de fonctionnement de ces propriétés sont en effet sensiblement identiques à ceux des propriétés conservées.

```

#pragma warning disable 1591
//-----
// <auto-generated>
//     Ce code a été généré par un outil.
//     Version du runtime :4.0.30319.42000
//
//     Les modifications apportées à ce fichier peuvent provoquer un
//     comportement incorrect et seront perdues si
//     le code est regénéré.
// </auto-generated>
//-----

namespace ConsoleApplication1
{
    using System.Data.Linq;
    using System.Data.Linq.Mapping;
    using System.Data;
    using System.Collections.Generic;
    using System.Reflection;
    using System.Linq;
    using System.Linq.Expressions;
    using System.ComponentModel;
    using System;

    [global::System.Data.Linq.Mapping.DatabaseAttribute(Name="Northwind")]

```

```

public partial class DataClasses1DataContext : System.Data.Linq.DataContext
{

    private static System.Data.Linq.Mapping.MappingSource mappingSource =
new AttributeMappingSource();

    #region Définitions de méthodes d'extensibilité
    partial void OnCreated();
    partial void InsertCustomers(Customers instance);
    partial void UpdateCustomers(Customers instance);
    partial void DeleteCustomers(Customers instance);
    #endregion

    public DataClasses1DataContext() :

base(global::ConsoleApplication1.Properties.Settings.Default.NorthwindConnection
String,

        mappingSource)
    {
        OnCreated();
    }

    public DataClasses1DataContext(string connection) :
        base(connection, mappingSource)
    {
        OnCreated();
    }

    public DataClasses1DataContext(System.Data.IDbConnection connection) :
        base(connection, mappingSource)
    {
        OnCreated();
    }

    public DataClasses1DataContext(string connection,
        System.Data.Linq.Mapping.MappingSource
mappingSource) :
        base(connection, mappingSource)
    {
        OnCreated();
    }

    public DataClasses1DataContext(System.Data.IDbConnection connection,
        System.Data.Linq.Mapping.MappingSource
mappingSource) :
        base(connection, mappingSource)
    {
        OnCreated();
    }

    public System.Data.Linq.Table<Customers> Customers
    {
        get
        {
            return this.GetTable<Customers>();
        }
    }
}

```

```

    }
}

[global::System.Data.Linq.Mapping.TableAttribute(Name="dbo.Customers")]
public partial class Customers : INotifyPropertyChanging,
    INotifyPropertyChanged
{
    private static PropertyChangingEventArgs emptyChangingEventArgs = new
    PropertyChangingEventArgs(String.Empty);

    private string _CustomerID;

    private string _CompanyName;

    private string _ContactName;

    private string _ContactTitle;

    #region Définitions de méthodes d'extensibilité
    partial void OnLoaded();
    partial void OnValidate(System.Data.Linq.ChangeAction action);
    partial void OnCreated();
    partial void OnCustomerIDChanging(string value);
    partial void OnCustomerIDChanged();
    partial void OnCompanyNameChanging(string value);
    partial void OnCompanyNameChanged();
    partial void OnContactNameChanging(string value);
    partial void OnContactNameChanged();
    partial void OnContactTitleChanging(string value);
    partial void OnContactTitleChanged();
    #endregion

    public Customers()
    {
        OnCreated();
    }

    [global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_CustomerID",
        DbType="Nchar(5) NOT
    NULL",
        CanBeNull=false,
        IsPrimaryKey=true)]

    public string CustomerID
    {
        get
        {
            return this._CustomerID;
        }
        set
        {
            if ((this._CustomerID != value))
            {
                this.OnCustomerIDChanging(value);
            }
        }
    }
}

```



```

        this.SendPropertyChanging();
        this._CustomerID = value;
        this.SendPropertyChanged("CustomerID");
        this.OnCustomerIDChanged();
    }
}

```

```

[global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_CompanyName",
                                                    DbType="NVarChar(40)
NOT NULL",
                                                    CanBeNull=false)]

```

```

public string CompanyName
{
    get
    {
        return this._CompanyName;
    }
    set
    {
        if ((this._CompanyName != value))
        {
            this.OnCompanyNameChanging(value);
            this.SendPropertyChanging();
            this._CompanyName = value;
            this.SendPropertyChanged("CompanyName");
            this.OnCompanyNameChanged();
        }
    }
}

```

```

[global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_ContactName",
                                                    DbType="NVarChar(30)")]

```

```

public string ContactName
{
    get
    {
        return this._ContactName;
    }
    set
    {
        if ((this._ContactName != value))
        {
            this.OnContactNameChanging(value);
            this.SendPropertyChanging();
            this._ContactName = value;
            this.SendPropertyChanged("ContactName");
            this.OnContactNameChanged();
        }
    }
}

```

```

[global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_ContactTitle",
DbType="NVarChar(30)")]
    public string ContactTitle
    {
        get
        {
            return this._ContactTitle;
        }
        set
        {
            if ((this._ContactTitle != value))
            {
                this.OnContactTitleChanging(value);
                this.SendPropertyChanging();
                this._ContactTitle = value;
                this.SendPropertyChanged("ContactTitle");
                this.OnContactTitleChanged();
            }
        }
    }

    public event PropertyChangingEventHandler PropertyChanging;

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void SendPropertyChanging()
    {
        if ((this.PropertyChanging != null))
        {
            this.PropertyChanging(this, emptyChangingEventArgs);
        }
    }

    protected virtual void SendPropertyChanged(String propertyName)
    {
        if ((this.PropertyChanged != null))
        {
            this.PropertyChanged(this, new
PropertyChangingEventArgs(propertyName));
        }
    }
}

#pragma warning restore 1591

```

L'étude de ce code met en évidence la mise en place de nombreux éléments :

- Le type `DataClasses1DataContext`, qui sert de conteneur de données. C'est ici l'équivalent objet d'une base de données SQL Server.
- L'utilisation de méthodes partielles, internes à `DataClasses1DataContext`, qui offrent des points d'extensibilité via l'écriture de code dans une seconde classe partielle.
- Le type `Customers`, qui est le pendant objet de la table du même nom.

- L'implémentation des interfaces `INotifyPropertyChanged` et `INotifyPropertyChanging` sur le type `Customers` et le déclenchement des événements qui leur sont associés lors de chaque assignation de valeur à une propriété du modèle.
- L'annotation des classes et propriétés du modèle, qui permet au moteur de LINQ to SQL de faire le pont entre les mondes .NET et SQL Server.

Cette mécanique lourde apporte tout de même de nombreuses simplifications d'écriture pour le développeur qui ne manipule plus que des objets .NET parfaitement typés, et ce à l'aide d'un langage souple, fluide et efficace : LINQ.

```
using (var db = new DataClasses1DataContext())
{
    var query = from client in db.Customers
                where client.CompanyName != "Bon app'"
                select client;

    foreach (var client in query)
        Console.WriteLine(client.CompanyName);
}
```

L'arrivée de LINQ to SQL a changé la manière de développer des applications .NET liées à des sources de données, mais cette technologie a certains défauts qui ont pu freiner son adoption à grande échelle :

- Compatibilité SQL Server uniquement.
- Aucun contrôle sur le code SQL généré.

Entity Framework

En parallèle du développement de LINQ to SQL, l'équipe responsable des technologies d'accès aux données avec .NET travaillait également à la mise au point d'un outil de mappage objet-relationnel : Entity Framework. Celui-ci a également été introduit avec le framework .NET 3.5, de sorte que les deux technologies de l'éditeur étaient placées en position de concurrence. Entity Framework, avec notamment sa capacité à agir pour le compte de plusieurs fournisseurs de données là où LINQ to SQL n'est compatible qu'avec SQL Server, a été préféré par les développeurs, et ce malgré les défauts de jeunesse que l'on pouvait lui trouver dans la première version de cette librairie. Par la suite, les efforts de Microsoft se sont concentrés sur Entity Framework, ce qui lui a permis d'évoluer jusqu'à aujourd'hui.

Au fil des versions, le panel de fonctionnalités proposées par cet outil s'est étoffé. Ainsi ont été ajoutées (liste non exhaustive) :

- La personnalisation du code C# généré par l'utilisation de templates T4.
- Le lazy loading, ou chargement paresseux, qui permet de naviguer dans un graphe d'entités en générant les requêtes SQL à la demande.
- L'approche Model First, qui s'ajoute à l'approche Database First présente depuis la première version d'Entity Framework. Model First offre la possibilité de générer la base de données à partir du modèle, plutôt que l'inverse.
- L'approche Code First, qui permet aux développeurs d'écrire leur propre modèle avec C# ou VB.NET pour le connecter à une base de données existante ou créée à la volée par Entity Framework.

- Les migrations, qui assouplissent l'utilisation de Code First par la mise à jour incrémentale de la structure de base de données en fonction du modèle codé.
- Les conventions personnalisées pour Code First.
- Le support de l'injection de dépendances, pour offrir des possibilités de découplage plus importantes

La palette de fonctionnalités implémentées, dans la sixième version d'Entity Framework rend cet outil de mappage objet-relationnel adaptable à de nombreux cas d'utilisation, notamment grâce aux trois approches qu'il permet de supporter.

1. Database First et Model First

Ces deux approches prennent le parti de maintenir une synchronisation entre un modèle objet défini dans un fichier XML (extension .edmx) et une source de données. Comme pour LINQ to SQL, ce fichier de description sert de source pour le processus de génération de code qui, cette fois-ci, s'appuie sur un template T4 (extension .tt). Ce fichier est entièrement personnalisable, voire remplaçable, ce qui permet d'intégrer Entity Framework dans des projets contraints en termes de structuration ou de règles concernant la forme du code produit, notamment.

Le mappage de la table Customers de la base de données Northwind de Microsoft génère donc une description dans un fichier EDMX, et du code définissant le contexte de données et l'entité Customer associée.

Le contenu du fichier EDMX est décrit ci-après.

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="3.0"
xmlns:edmx="http://schemas.microsoft.com/ado/2009/11/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>
      <Schema Namespace="NorthwindModel.Store" Provider="System.Data.SqlClient"
        ProviderManifestToken="2012" Alias="self"

        xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"

        xmlns:customannotation="http://schemas.microsoft.com/ado/2013/11/edm/customannotation"

        xmlns="http://schemas.microsoft.com/ado/2009/11/edm/ssdl">
        <EntityType Name="Customers">
          <Key>
            <PropertyRef Name="CustomerID" />
          </Key>
          <Property Name="CustomerID" Type="nchar" MaxLength="5"
            Nullable="false" />
          <Property Name="CompanyName" Type="nvarchar" MaxLength="40"
            Nullable="false" />
          <Property Name="ContactName" Type="nvarchar" MaxLength="30" />
          <Property Name="ContactTitle" Type="nvarchar" MaxLength="30" />
          <Property Name="Address" Type="nvarchar" MaxLength="60" />
          <Property Name="City" Type="nvarchar" MaxLength="15" />
          <Property Name="Region" Type="nvarchar" MaxLength="15" />
```

```

        <Property Name="PostalCode" Type="nvarchar" MaxLength="10" />
        <Property Name="Country" Type="nvarchar" MaxLength="15" />
        <Property Name="Phone" Type="nvarchar" MaxLength="24" />
        <Property Name="Fax" Type="nvarchar" MaxLength="24" />
    </EntityType>
    <EntityContainer Name="NorthwindModelStoreContainer">
        <EntitySet Name="Customers" EntityType="Self.Customers" Schema="dbo"
store:Type="Tables" />
    </EntityContainer>
</Schema>
</edmx:StorageModels>
<!-- CSDL content -->
<edmx:ConceptualModels>
    <Schema Namespace="NorthwindModel" Alias="Self"
annotation:UseStrongSpatialTypes="false"

xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"

xmlns:customannotation="http://schemas.microsoft.com/ado/2013/11/edm/customannot
ation"

        xmlns="http://schemas.microsoft.com/ado/2009/11/edm">
    <EntityType Name="Customers">
        <Key>
            <PropertyRef Name="CustomerID" />
        </Key>
        <Property Name="CustomerID" Type="String" MaxLength="5"
FixedLength="true" Unicode="true" Nullable="false" />
        <Property Name="CompanyName" Type="String" MaxLength="40"
FixedLength="false" Unicode="true" Nullable="false" />
        <Property Name="ContactName" Type="String" MaxLength="30"
FixedLength="false" Unicode="true" />
        <Property Name="ContactTitle" Type="String" MaxLength="30"
FixedLength="false" Unicode="true" />
        <Property Name="Address" Type="String" MaxLength="60"
FixedLength="false" Unicode="true" />
        <Property Name="City" Type="String" MaxLength="15"
FixedLength="false" Unicode="true" />
        <Property Name="Region" Type="String" MaxLength="15"
FixedLength="false" Unicode="true" />
        <Property Name="PostalCode" Type="String" MaxLength="10"
FixedLength="false" Unicode="true" />
        <Property Name="Country" Type="String" MaxLength="15"
FixedLength="false" Unicode="true" />
        <Property Name="Phone" Type="String" MaxLength="24"
FixedLength="false" Unicode="true" />
        <Property Name="Fax" Type="String" MaxLength="24"
FixedLength="false" Unicode="true" />
    </EntityType>
    <EntityContainer Name="NorthwindEntities2"
        annotation:LazyLoadingEnabled="true">
        <EntitySet Name="Customers" EntityType="Self.Customers" />
    </EntityContainer>
</Schema>
</edmx:ConceptualModels>
<!-- C-S mapping content -->
<edmx:Mappings>

```

```

    <Mapping Space="C-S"
xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
    <EntityContainerMapping
StorageEntityContainer="NorthwindModelStoreContainer"
CdmEntityContainer="NorthwindEntities2">
    <EntitySetMapping Name="Customers">
    <EntityTypeMapping TypeName="NorthwindModel.Customers">
    <MappingFragment StoreEntitySet="Customers">
    <ScalarProperty Name="CustomerID" ColumnName="CustomerID" />
    <ScalarProperty Name="CompanyName" ColumnName="CompanyName" />
    <ScalarProperty Name="ContactName" ColumnName="ContactName" />
    <ScalarProperty Name="ContactTitle" ColumnName="ContactTitle" />
    <ScalarProperty Name="Address" ColumnName="Address" />
    <ScalarProperty Name="City" ColumnName="City" />
    <ScalarProperty Name="Region" ColumnName="Region" />
    <ScalarProperty Name="PostalCode" ColumnName="PostalCode" />
    <ScalarProperty Name="Country" ColumnName="Country" />
    <ScalarProperty Name="Phone" ColumnName="Phone" />
    <ScalarProperty Name="Fax" ColumnName="Fax" />
    </MappingFragment>
    </EntityTypeMapping>
    </EntitySetMapping>
    </EntityContainerMapping>
    </Mapping>
    </edmx:Mappings>
</edmx:Runtime>
<!-- EF Designer content (DO NOT EDIT MANUALLY BELOW HERE) -->
<Designer xmlns="http://schemas.microsoft.com/ado/2009/11/edmx">
    <Connection>
    <DesignerInfoPropertySet>
    <DesignerProperty Name="MetadataArtifactProcessing"
Value="EmbedInOutputAssembly" />
    </DesignerInfoPropertySet>
    </Connection>
    <Options>
    <DesignerInfoPropertySet>
    <DesignerProperty Name="ValidateOnBuild" Value="true" />
    <DesignerProperty Name="EnablePluralization" Value="false" />
    <DesignerProperty Name="IncludeForeignKeysInModel" Value="true" />
    <DesignerProperty Name="UseLegacyProvider" Value="false" />
    <DesignerProperty Name="CodeGenerationStrategy" Value="Aucun" />
    </DesignerInfoPropertySet>
    </Options>
    <!-- Diagram content (shape and connector positions) -->
    <Diagrams></Diagrams>
    </Designer>
</edmx:Edmx>

```

Ce fichier contient quatre sections principales. Trois décrivent respectivement la structure de la source de données, la structure du modèle objet, la relation entre ces deux éléments (autrement dit, le mappage), la quatrième fournissant des informations générales relatives au modèle (options de génération ou données associées à l'affichage dans le designer).

Le modèle est très simplifié, notamment parce qu'il est expurgé des informations intégrées au fichier EDMX. Ce dernier est alors traité au moment de l'exécution de sorte que le contexte de données puisse avoir à disposition toutes les informations nécessaires à son fonctionnement.

```
namespace ConsoleApplication2
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    public partial class NorthwindEntities : DbContext
    {
        public NorthwindEntities()
            : base("name=NorthwindEntities")
        {
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public virtual DbSet<Customers> Customers { get; set; }
    }
}
```

```
namespace ConsoleApplication2
{
    using System;
    using System.Collections.Generic;

    public partial class Customers
    {
        public string CustomerID { get; set; }
        public string CompanyName { get; set; }
        public string ContactName { get; set; }
        public string ContactTitle { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Region { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
        public string Phone { get; set; }
        public string Fax { get; set; }
    }
}
```

Entity Framework répond avec ce mode de fonctionnement au besoin qu'ont les développeurs de contrôler l'ensemble du code qui est sous leur responsabilité, ce qui n'est pas toujours réalisable avec LINQ to SQL.

2. Code First

L'approche Code First va encore plus loin dans la simplification en supprimant tout fichier de mappage : ni DBML, ni EDMX. Toute la description est définie à partir de code, potentiellement généré, mais toujours contrôlable par le développeur. Les éléments particuliers de configuration sont généralement spécifiés à l'aide d'attributs, mais peuvent tout aussi bien être ajoutés à partir de code. Afin de simplifier encore cette tâche de configuration, et de libérer ainsi le modèle d'un maximum d'éléments n'ayant pas de valeur métier, l'approche Code First utilise de nombreuses conventions d'écriture qui permettent à Entity Framework de déterminer les mappages à créer.

Avec Code First, le modèle objet correspondant à l'intégration de la table `Customers` de la base de données Northwind est comparable au code généré par les approches précédentes, puisqu'il suffit de supprimer le déclenchement d'exception présent dans la méthode `OnModelCreating` du type `NorthwindEntities` pour obtenir un "modèle Code First". Des gestions supplémentaires, comme celle des longueurs maximales, peuvent être ajoutées soit par le biais d'annotations, soit dans la méthode `OnModelCreating`, mais le modèle est fonctionnel en l'état.

Dans sa version 6, Entity Framework est un outil de mappage objet-relationnel puissant et mûr, mais son héritage le rend parfois un peu lourd. C'est toutefois un outil de qualité qui peut être utilisé sans difficulté pour des prototypes ou des projets d'envergure, qu'ils utilisent ASP.NET (WebForms ou MVC), WPF ou WinForms.

D'Entity Framework 7 à Entity Framework Core

Initialement révélée au monde sous le nom d'Entity Framework 7, cette version de l'outil de mappage objet-relationnel est conçue sur des fondations bien différentes de la version précédente. Elle répond à des besoins nouveaux, dans un monde nouveau fait de mobilité, d'ouverture et de performance.

1. Genèse

Parmi les objectifs de cette nouvelle version, on trouve notamment le souhait de cibler des sources de données non relationnelles (bases de données NoSQL, par exemple), mais aussi de cibler des plateformes logicielles plus nombreuses : applications Windows Store (Desktop + Mobile), applications .NET Core sur Windows, mais aussi sur Linux et Mac OS X.

Les challenges à surmonter pour faire évoluer Entity Framework 6 vers l'objectif fixé étaient principalement liés à l'historique de la librairie. Elle a été conçue pour être intégrée à un framework .NET monolithique, et sa structure a par conséquent ce même défaut. En interne, elle fait également un usage intensif d'API et de patrons de conception vieillissants, ce qui complique la maintenance et diminue l'évolutivité. Enfin, certains des concepts que cette librairie utilise sont étroitement liés aux sources de données relationnelles, bloquant ainsi le chemin vers les sources de données basées sur des concepts différents.

Pour ces raisons, le choix a été fait de repartir d'une feuille blanche pour l'écriture de cette nouvelle version, permettant ainsi de concevoir différemment les éléments constitutifs de l'outil.

2. Une véritable version 1

En plus des problématiques déjà énoncées, Entity Framework 6 a certains défauts que seule une réécriture complète pouvait arranger, particulièrement pour ce qui concerne la consommation de ressources. Avec l'objectif de l'exécution sur des appareils mobiles ou dans le cloud, la consommation excessive de mémoire ou de cycles processeur peut s'avérer catastrophique.

L'équipe en charge de ce développement a donc réécrit l'ORM avec, entre autres, l'optimisation de cette consommation comme fil rouge. L'extensibilité n'est pas en reste, puisque la librairie est conçue comme un gigantesque jeu de briques coordonné par un moteur d'injection de dépendances. Les perspectives de modification et d'extension s'en trouvent donc multipliées, chaque service pouvant être remplacé aisément directement au niveau du conteneur de services qui est au cœur du système.

L'équipe de développement a annoncé dès les premières conférences publiques concernant Entity Framework 7 que la première version finale ne pourrait supporter le même ensemble de fonctionnalités que son prédécesseur. Certaines ont d'ailleurs été complètement mises de côté, puisque déjà pas ou peu utilisées avec Entity Framework 6. Les plans concernant les fonctionnalités à inclure intègrent entre autres le chargement paresseux, le mappage de procédures stockées ou le développement d'outils graphiques pour la phase de modélisation.

Pour être disponible de la même manière sur toutes les plateformes cibles, Entity Framework 7 a été écrit de manière à tirer parti du framework DNX dont le développement était effectué en parallèle. Celui-ci offre la possibilité d'exécuter du code .NET (incluant le tout nouveau ASP.NET 5) au sein d'applications Windows Store, mais surtout sur des systèmes Linux et Mac OS X grâce au composant CoreCLR. Toutes ces nouveautés ne pouvaient pas être mises en évidence par l'utilisation du schéma standard de numérotation qui suggère que la version la plus récente correspond à une évolution de la version précédente. Ici, il n'est pas seulement question d'une évolution, mais également d'une modification radicale des concepts qui dirigent son développement. Microsoft a donc décidé d'adopter un nouveau nommage pour DNX, Entity Framework 7 et ASP.NET 5 : * Core 1.0. Le 27 juin 2016, les versions finales de ces différents éléments logiciels ont donc été dévoilées sous les noms de .NET Core 1.0, Entity Framework Core 1.0 et ASP.NET Core 1.0.

3. Quand choisir Entity Framework Core ?

Entity Framework Core est une librairie neuve, à laquelle il manque certaines fonctionnalités que l'équipe de développement considère comme critiques pour un outil de mappage objet-relationnel. Pour cette raison, Entity Framework 6 reste le choix le plus approprié pour de nombreuses applications conçues pour le framework .NET (jusqu'à sa version 4.6.2). La décision d'utiliser Entity Framework est aujourd'hui surtout associée à l'environnement dans lequel cette librairie doit être utilisée : les applications qui ciblent .NET Core (Universal Windows Applications ou ASP.NET Core) ne peuvent bénéficier d'Entity Framework 6 qui est dépendant du framework .NET "complet". Toutefois, les nouvelles applications ne nécessitant pas l'utilisation de fonctionnalités non implémentées par Entity Framework Core peuvent inclure cette librairie comme solution d'accès aux données.

Modélisation

Installation d'Entity Framework Core

Entity Framework Core a été conçu pour être entièrement déployé à l'aide du gestionnaire de package NuGet développé par Microsoft et intégré à Visual Studio. Comme APT pour le système Debian, NPM pour Node.js ou pip pour Python, NuGet permet de rendre disponibles à tous des librairies en encapsulant les binaires (ou fichiers de code) ainsi que leurs dépendances au sein d'un unique fichier. Ce dernier est tout simplement une archive au format ZIP dont l'extension est .nupkg et qui contient notamment un manifeste définissant les dépendances associées à la librairie et les environnements pour lesquels elle est disponible.

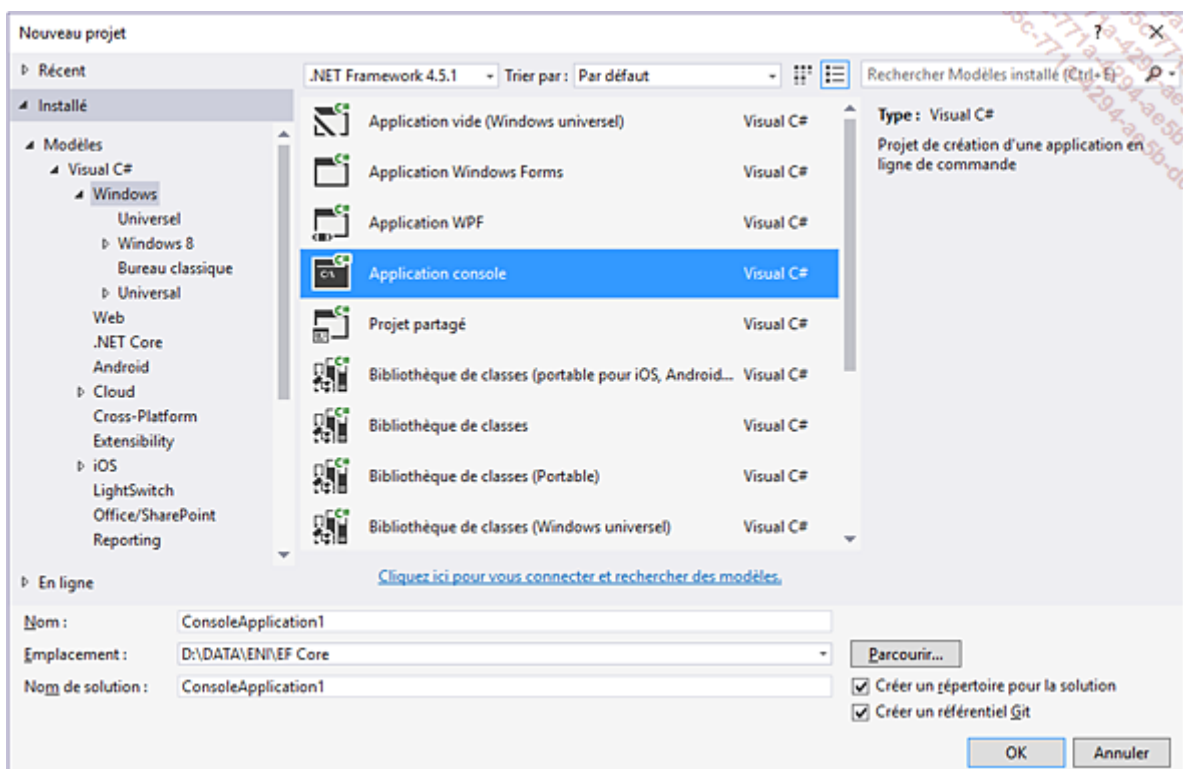
NuGet est principalement utilisé pour la distribution de librairies ciblant la plateforme .NET, mais quelques exceptions existent, notamment dans le cas de bibliothèques JavaScript comme jQuery ou AngularJS.

1. Dans une application .NET

Commençons par la création d'un projet d'application .NET qui utilisera Entity Framework Core. Pour cet exemple, vous utiliserez le modèle Application Console fourni par Visual Studio.

Dans Visual Studio, ouvrez la boîte de dialogue **Nouveau projet** à partir du menu **Fichier - Nouveau - Projet**.

Sélectionnez le modèle **Application console**, renseignez les informations demandées (nom du projet, emplacement...) et validez.

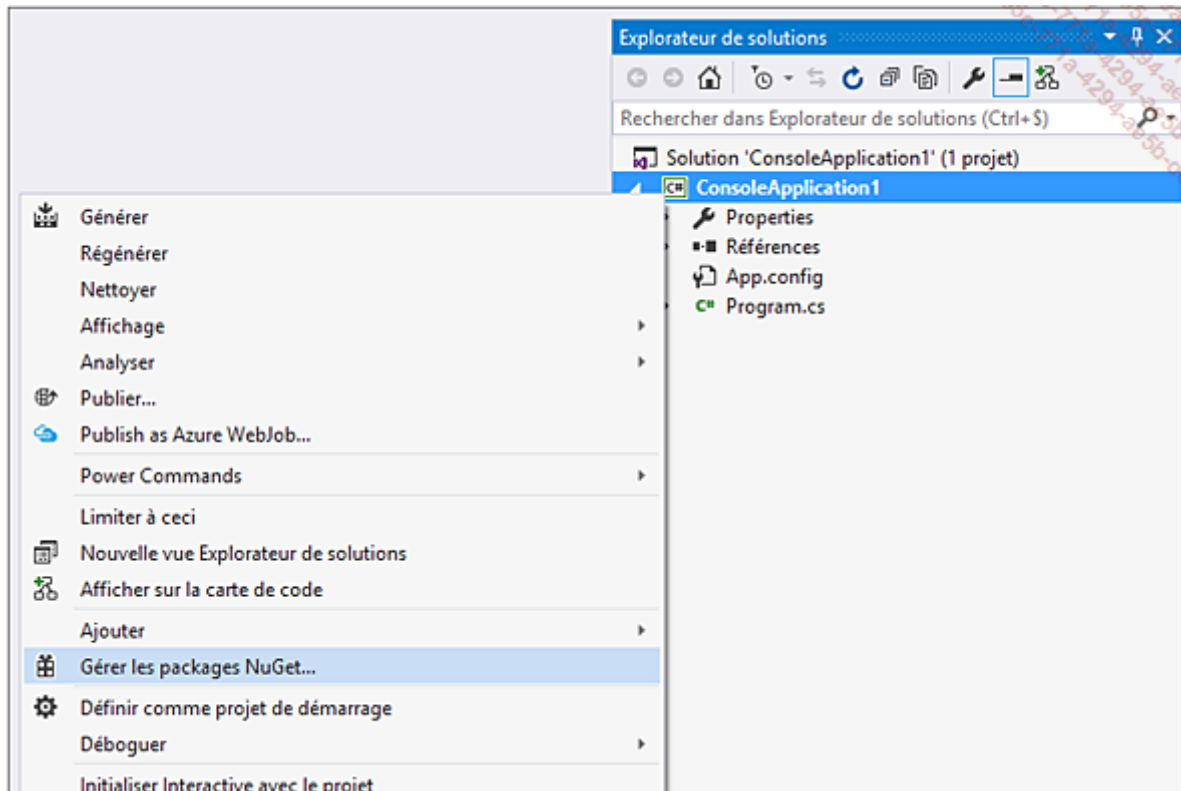


Lors de l'étape de création, il est important de vérifier la version du framework .NET ciblée par l'application. Entity Framework Core n'est en effet compatible qu'avec les versions 4.5.1 et supérieures.

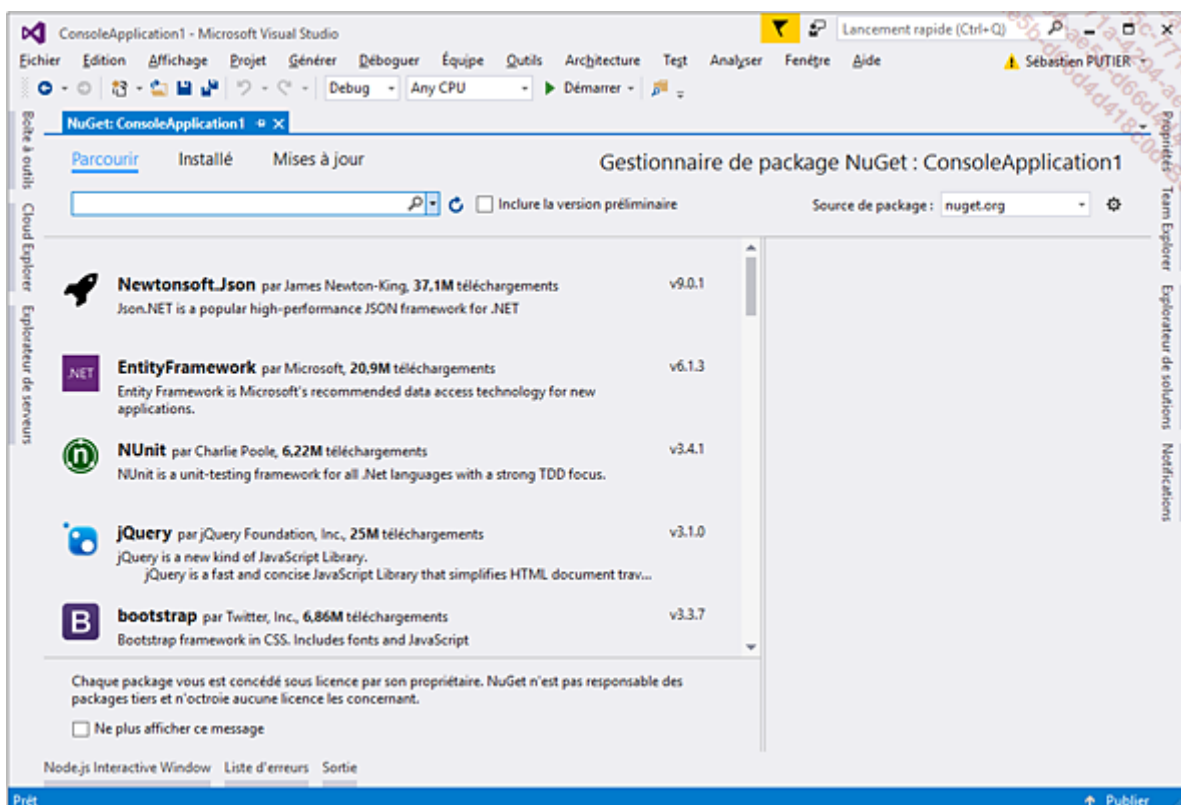
Une fois le projet créé, il faut ajouter le package Microsoft.EntityFrameworkCore à l'aide de NuGet. Deux interfaces pour cet outil sont intégrées à Visual Studio : une interface graphique et une console permettant d'utiliser des lignes de commande.

Interface graphique

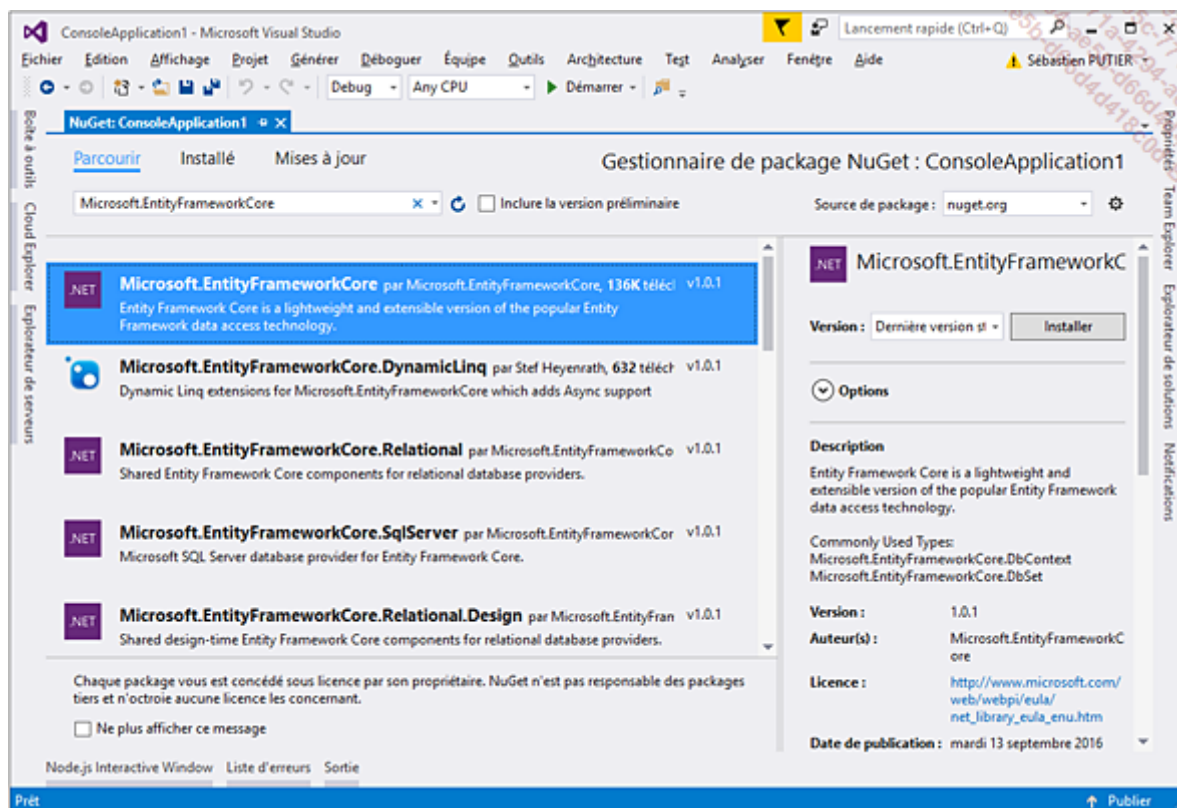
Ouvrez l'interface graphique en sélectionnant, via le menu contextuel du projet, l'option **Gérer les packages NuGet**.



L'onglet **Parcourir** de cette interface permet de lister et d'effectuer des recherches sur les packages NuGet disponibles.



Recherchez **Microsoft.EntityFrameworkCore**. Le premier élément retourné devrait correspondre au package souhaité.



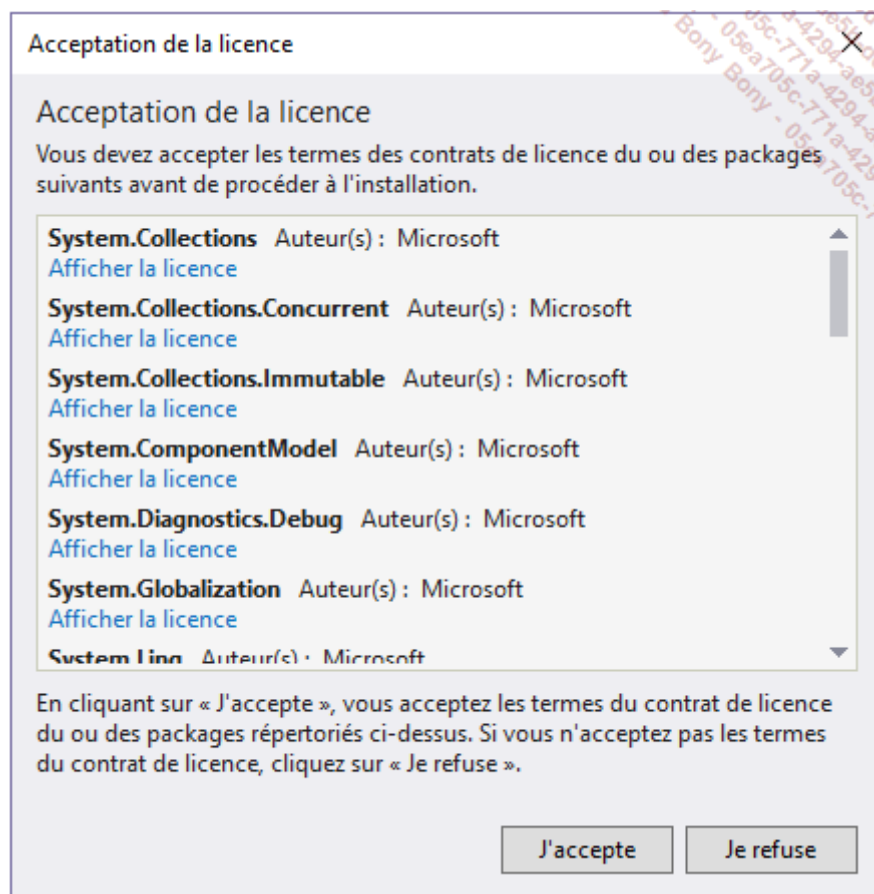
La partie droite de l'interface présente quelques informations relatives au package sélectionné : nom, version courante, description, auteur, licence, etc. Une liste déroulante permet également la sélection de la version souhaitée.

Cliquez sur le bouton **Installer** pour lancer le processus d'intégration du package au projet. La dernière version stable à l'écriture de ces lignes est la version 1.1.0.

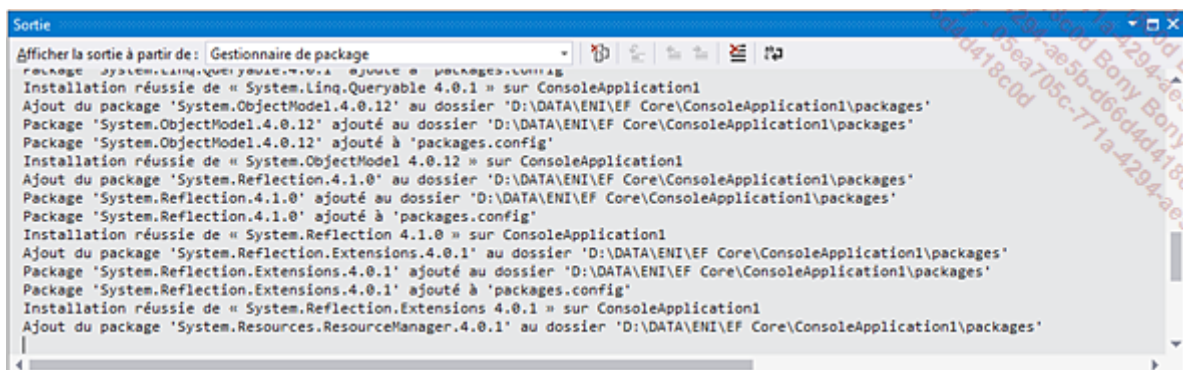
Lors de l'installation, Visual Studio ouvre une fenêtre indiquant les dépendances du package qu'il est nécessaire d'installer.

!images/02-005.png

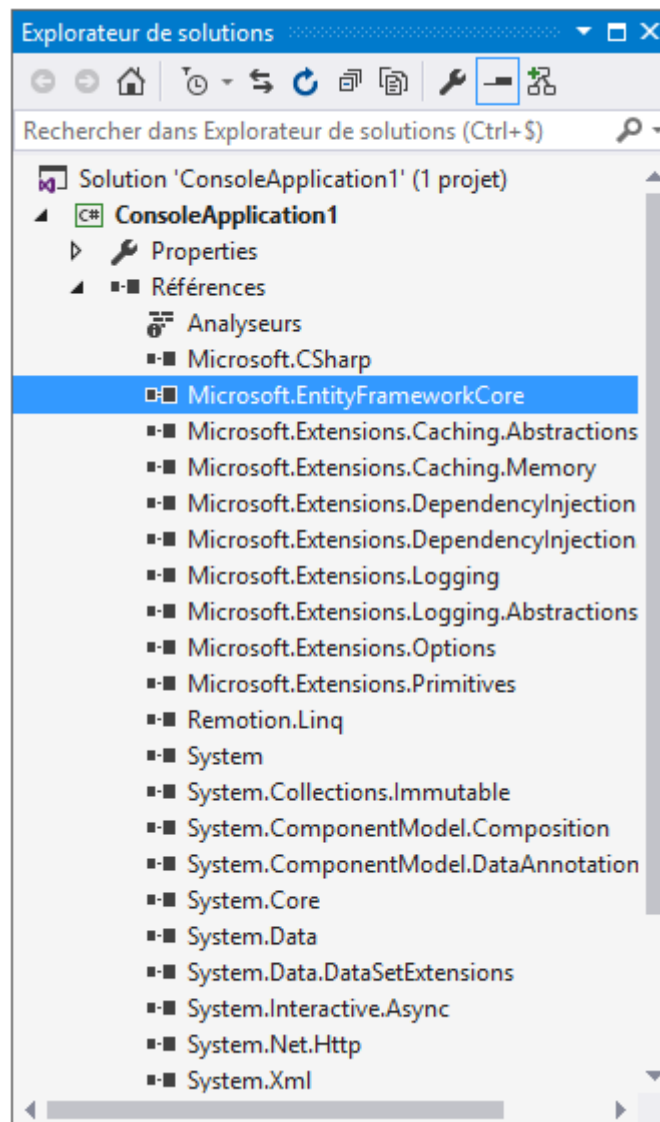
Validez cette liste de modifications, puis acceptez les licences d'utilisation des différentes librairies à installer.



Les différentes opérations effectuées pendant l'installation sont décrites dans la fenêtre **Sortie** de Visual Studio.

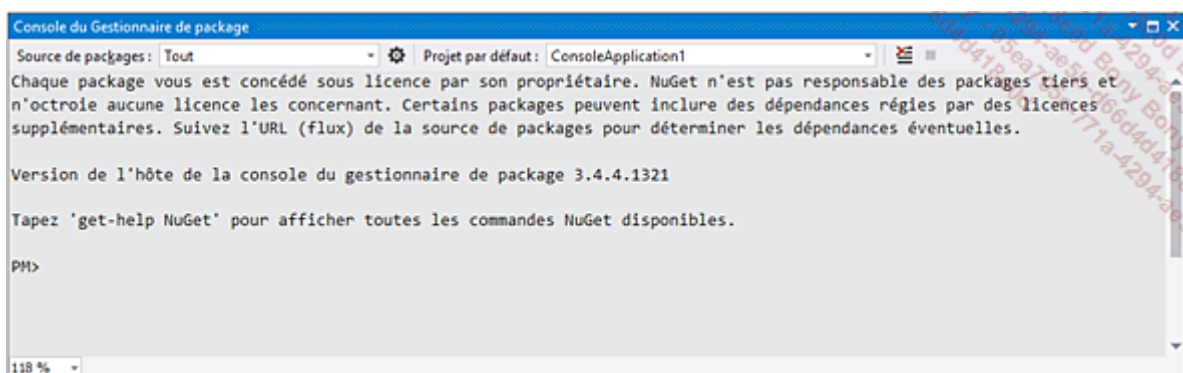


Une fois l'installation terminée, le projet contient les références nécessaires à l'utilisation d'Entity Framework Core. Le nombre de dépendances installées est relativement conséquent en raison de l'architecture modulaire adoptée par .NET Core. La librairie Entity Framework Core étant conçue pour cette plateforme, elle respecte logiquement ce principe de modularité.



Console

Ouvrez la console NuGet à l'aide du menu **Outils - Gestionnaire de package NuGet - Console du Gestionnaire de package**.



À partir de cette console, lancez l'installation du package contenant la bibliothèque principale d'Entity Framework Core à l'aide de la commande suivante :

```
Install-Package Microsoft.EntityFrameworkCore
```

Pour l'installation d'une version spécifique du package, ajoutez le paramètre `version`. Pour installer la version 1.1.0, la commande est la suivante :

```
Install-Package Microsoft.EntityFrameworkCore -Version 1.1.0
```


À ce stade, le package est installé dans le projet par défaut défini dans la console.

Pour préciser le nom du projet pour lequel il faut installer le package, utilisez le paramètre `ProjectName`.

```
Install-Package Microsoft.EntityFrameworkCore -Version 1.1.0  
-ProjectName <nom du projet>
```

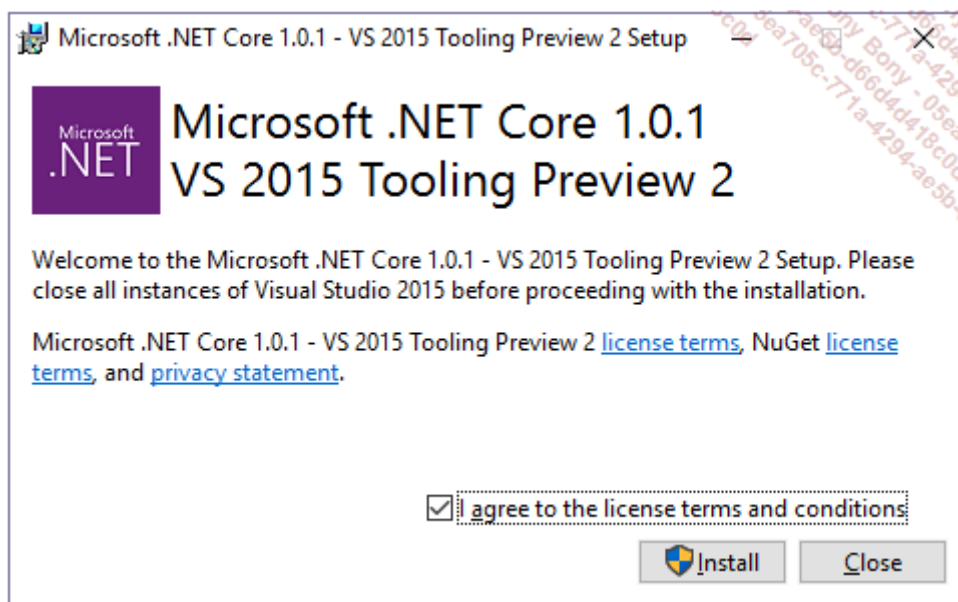
La validation de cette commande déclenche le processus d'installation du package dans le projet.

2. Dans une application .NET Core

L'intégration d'Entity Framework Core dans une application .NET Core tire parti des outils disponibles à travers le SDK .NET Core. Il convient donc d'effectuer l'installation de ce SDK sur la machine de développement, avant de configurer le projet pour inclure la librairie Entity Framework Core.

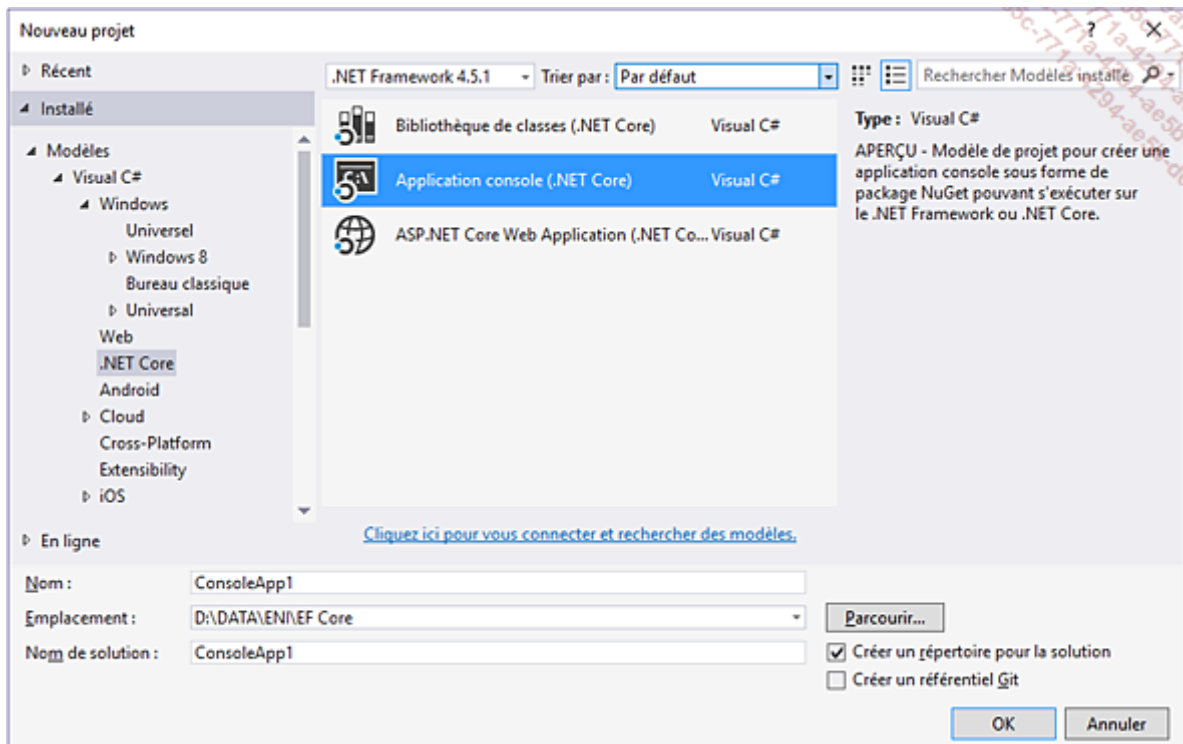
a. Windows

L'installation de .NET Core sur un système d'exploitation Windows est très simple puisque Microsoft fournit des packages contenant les outils nécessaires au développement et à l'exécution des applications basées sur ce framework. Ces packages correspondent à plusieurs cibles : Visual Studio 2015, Visual Studio 2017 ou le SDK seul utilisé en ligne de commande. Ces packages sont tous disponibles à partir du site web <https://www.microsoft.com/net/core>. Ici, l'installation est effectuée pour Visual Studio 2015, puisque c'est l'environnement de développement qui sera utilisé tout au long de cet ouvrage.



Une fois les outils nécessaires installés, de nouveaux modèles de projets ciblant .NET Core peuvent être utilisés dans Visual Studio.

Ouvrez la fenêtre **Nouveau projet** et créez un projet d'application console .NET Core.



L'installation de la librairie Entity Framework Core dans l'application est réalisée de la même manière que dans une application .NET traditionnelle.

À partir du gestionnaire de package NuGet, recherchez puis installez le package Microsoft.EntityFrameworkCore.

Le contenu du fichier project.json, qui décrit le projet et ses dépendances, devrait alors être semblable à ceci.

```
{
  "version": "1.0.1-*",
  "buildOptions": {
    "emitEntryPoint": true
  },
  "dependencies": {
    "Microsoft.EntityFrameworkCore": "1.1.0",
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.1"
    }
  },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": "dnxcore50"
    }
  }
}
```

Entity Framework Core est maintenant disponible dans l'application console.

b. Linux

.NET Core peut être installé sur plusieurs distributions Linux, parmi lesquelles Red Hat Enterprise Linux 7 Server, openSUSE ou Fedora. La description ci-après correspond à la mise en place d'un projet utilisant Entity Framework Core sur une distribution Ubuntu 14.04.

Pour ce système d'exploitation, le SDK est disponible à travers le gestionnaire de package APT.

Pour l'installer, configurez APT de manière qu'il puisse pointer sur le flux Microsoft Open Tech qui lui est dédié. Les commandes suivantes permettent d'exécuter cette opération.


```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/
dotnet-release/ trusty main" > /etc/apt/sources.list.d/dotnetdev.list'
```

```
sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893
```

```
sudo apt-get update
```

```
seb@ubuntu-efcore:~$ sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficma
nager.net/repos/dotnet-release/ trusty main" > /etc/apt/sources.list.d/dotnetdev
.list'
seb@ubuntu-efcore:~$ sudo apt-key adv --keyserver apt-mo.trafficmanager.net --re
cv-keys 417A0893
Executing: gpg --ignore-time-conflict --no-options --no-default-keyring --homedir
/tmp/tmp.x7QPmLv0Pe --no-auto-check-trustdb --trust-model always --keyring /etc
/apt/trusted.gpg --primary-keyring /etc/apt/trusted.gpg --keyserver apt-mo.traf
ficmanager.net --recv-keys 417A0893
gpg: requesting key 417A0893 from hkp server apt-mo.trafficmanager.net
gpg: key 417A0893: public key "MS Open Tech <interop@microsoft.com>" imported
gpg: Total number processed: 1
gpg:             imported: 1 (RSA: 1)
seb@ubuntu-efcore:~$ sudo apt-get update
```

Une fois ces commandes exécutées, téléchargez et installez le SDK à l'aide d'une commande apt-get install classique.

```
sudo apt-get install dotnet-dev-1.0.0-preview2-003131
```

À ce jour, la dernière version disponible du SDK .NET Core pour Linux est en preview. Le numéro de version associé est donc susceptible d'évoluer très vite.

```
seb@ubuntu-efcore:~$ sudo apt-get install dotnet-dev-1.0.0-preview2-003131
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les NOUVEAUX paquets suivants seront installés :
  dotnet-dev-1.0.0-preview2-003131
0 mis à jour, 1 nouvellement installés, 0 à enlever et 5 non mis à jour.
Il est nécessaire de prendre 0 o/17,0 Mo dans les archives.
Après cette opération, 29,9 Mo d'espace disque supplémentaires seront utilisés.
Sélection du paquet dotnet-dev-1.0.0-preview2-003131 précédemment désélectionné.
(Lecture de la base de données... 107350 fichiers et répertoires déjà installés.
)
Preparing to unpack .../dotnet-dev-1.0.0-preview2-003131_1.0.0-preview2-003131-1
_amd64.deb ...
Unpacking dotnet-dev-1.0.0-preview2-003131 (1.0.0-preview2-003131-1) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Paramétrage de dotnet-dev-1.0.0-preview2-003131 (1.0.0-preview2-003131-1) ...
This software may collect information about you and your use of the software, an
d send that to Microsoft.
Please visit http://aka.ms/dotnet-cli-eula for more information.
seb@ubuntu-efcore:~$
```

Pour initialiser un projet .NET Core, exécutez la commande dotnet new. Cette commande est fournie par une interface en ligne de commande (.NET Core CLI) présente dans le SDK installé précédemment.

```

seb@ubuntu-efcore:/$ mkdir /home/seb/efcore
seb@ubuntu-efcore:/$ cd /home/seb/efcore
seb@ubuntu-efcore:~/efcore$ dotnet new
Created new C# project in /home/seb/efcore.
seb@ubuntu-efcore:~/efcore$ ls
Program.cs  project.json
seb@ubuntu-efcore:~/efcore$

```

L'intégration d'Entity Framework Core dans ce projet passe par la modification du fichier project.json qui le décrit.

Ajoutez une référence au package NuGet souhaité dans la section dependencies de ce fichier.

```

seb@ubuntu-efcore:~/efcore$ cat project.json
{
  "version": "1.0.0-*",
  "buildOptions": {
    "debugType": "portable",
    "emitEntryPoint": true
  },
  "dependencies": {
    "Microsoft.EntityFrameworkCore": "1.0.0"
  },
  "frameworks": {
    "netcoreapp1.0": {
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.0.1"
        }
      },
      "imports": "dnxcore50"
    }
  }
}

```

Une fois cette configuration mise à jour, exécutez la commande dotnet restore pour télécharger et installer les packages définis par cette liste de dépendances dans le projet.

Entity Framework Core est maintenant disponible dans le projet.

```

seb@ubuntu-efcore:~/efcore$ dotnet restore
log : Restoring packages for /home/seb/efcore/project.json...
log : Installing Microsoft.EntityFrameworkCore 1.0.0.
log : Writing lock file to disk. Path: /home/seb/efcore/project.lock.json
log : /home/seb/efcore/project.json
log : Restore completed in 8364ms.

```

c. OS X

En premier lieu, il est nécessaire d'obtenir la dernière version d'OpenSSL, ce qui est peut être effectué à l'aide de Homebrew, un gestionnaire de package pour Mac OS X. Les commandes décrites ci-après permettent d'installer ce gestionnaire, de le mettre à jour, puis d'installer et de configurer OpenSSL.

```

~$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

~$ brew update

~$ brew install openssl

~$ mkdir -p /usr/local/lib

~$ ln -s /usr/local/opt/openssl/lib/libcrypto.1.0.0.dylib /usr/local/lib/

~$ ln -s /usr/local/opt/openssl/lib/libssl.1.0.0.dylib /usr/local/lib/

```

Lorsque ce prérequis est satisfait, l'installation du SDK .NET Core est réalisée à l'aide du package d'installation localisé à l'adresse <https://go.microsoft.com/fwlink/?LinkID=835011>.

La suite de la mise en place est comparable à la procédure décrite pour Ubuntu. Il est donc question de quelques lignes de commande de manière à créer et à initialiser un projet .NET Core, suivies de l'ajout du package Microsoft.EntityFrameworkCore à la liste des dépendances de l'application, et enfin de la restauration des packages et du démarrage de l'application.

Pour initialiser un projet .NET Core, exécutez les commandes suivantes :

```

mkdir efcore
cd efcore
dotnet new

```

Éditez le fichier de configuration du projet pour ajouter la dépendance à Entity Framework Core.

Project.json

```

{
  "version": "1.0.0-*",
  "buildOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.EntityFrameworkCore": "1.0.0",
    "Microsoft.NETCore.App": {
      "type": "platform",
      "version": "1.0.0-rc2-3002702"
    }
  },

  "frameworks": {
    "netcoreapp1.0": {
      "imports": "dnxcore50"
    }
  }
}

```

Exécutez les commandes suivantes pour restaurer les dépendances de l'application et l'exécuter.

Deux approches pour deux cas d'utilisation

L'incorporation dans un projet d'une technologie de mappage objet-relationnel comme Entity Framework Core peut être mise en œuvre à tout moment du cycle de développement : en amont, avant que la moindre ligne de code ne soit écrite, ou pendant le développement, suite à la mise en relief d'un défaut de la technologie utilisée ou pour suivre l'évolution de standards de développement. Cette mise en œuvre doit toutefois être effectuée d'une manière adaptée à l'état de la source de données. À cet effet, deux types d'intégrations peuvent être utilisés :

- 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 depuis un certain temps ou qu'elle doit être créée et maintenue par une tierce personne (un DBA - ou *DataBase Administrator* - par exemple).

1. Model First

L'approche Model First est couramment utilisée lors de l'écriture de petites applications, comme des prototypes ou des applications mobiles. Néanmoins, elle n'est pas inadaptée aux applications métier.

Lorsque l'on parle de Model, on fait référence aux classes de l'application permettant de manipuler les données issues de la source de données. Reprenons l'exemple vu à la section Mappage objet-relationnel au chapitre .NET et l'accès aux données. La classe `Client` est l'élément de la couche Model associé à la table `Client` de la base de données. Parler de Model First revient donc à évoquer l'écriture de code .NET qui sera utilisé pour générer la structure d'une source de données.

L'approche Model First contraste donc fortement avec la méthode utilisée traditionnellement pour la mise en place d'une source de données puisqu'elle est entièrement dévolue à l'équipe de développement et ne nécessite pas de scripts ou d'outils externes. De plus, la structure de la source de données est définie et parfaitement intégrée à l'application, ce qui simplifie son évolution et sa maintenance, de même que sa gestion avec les outils de contrôle de code source comme Git ou Team Foundation Server.

Malgré cet avantage indéniable pour les développeurs, cette approche peut également se révéler handicapante dans le cadre de la gestion de la montée en charge d'une application. Lorsque la source de données est une base de données relationnelle, comme SQL Server ou Oracle, aucun DBA n'est associé à sa création et à sa maintenance, entraînant par là la disparition de potentielles optimisations cruciales pour la maîtrise des données et des traitements. C'est d'ailleurs une des raisons pour lesquelles Model First est préféré lors de "petits" développements. Pour parer à cette problématique, il est toutefois possible de mettre en œuvre une seconde approche : Database First.

2. Database First

Le schéma plus classique dans lequel la création de la base de données précède le développement de la couche applicative d'accès aux données peut également être pris en charge par Entity Framework, à l'aide de l'approche Database First. Celle-ci s'appuie sur les mêmes principes d'implémentation que Model First, tout en permettant la génération du code source .NET d'accès aux données.

Pour atteindre cet objectif, Microsoft met à disposition des outils en ligne de commande :

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

Malgré son aspect complexe, l'interface en ligne de commande a des avantages non négligeables en termes de flexibilité. Chaque développeur a la possibilité de créer des scripts correspondant à sa propre utilisation de l'outil. Elle est utilisable par tout membre de l'équipe ne maîtrisant pas le développement .NET, et tout particulièrement par le DBA, qui peut ainsi mettre à jour la couche d'accès aux données à chaque modification structurelle de la source de données. Elle peut également être utilisée dans le cadre d'une chaîne d'intégration continue, sans intervention humaine directe : avant chaque compilation de l'application par Team Foundation Server, Jenkins, ou tout autre outil, il est possible de l'exécuter de manière à s'assurer, par exemple, que le code source .NET correspond parfaitement à une source de données spécifique.

Il n'est toutefois pas obligatoire de générer le code .NET correspondant à la source de données pour utiliser l'approche Database First. Tout modèle Entity Framework Core configuré de manière à correspondre à la source de données peut être utilisé en lieu et place du code généré.

Du modèle de données au modèle objet

Différentes notions relatives au modèle objet utilisé par Entity Framework Core ont été évoquées depuis le début de cet ouvrage. Dans cette section, nous allons mettre en place la structure générale d'un modèle simpliste relatif à la gestion des commandes de clients. Ce modèle sera modifié par la suite de manière à obtenir une couche d'accès répondant à différents besoins.

1. Point d'entrée : DbContext

La classe `DbContext` est un élément central de l'utilisation d'Entity Framework Core. Elle est présente dans tous les projets utilisant cette technologie par le biais d'une classe dérivée. C'est à partir de cette classe que toutes les manipulations de données peuvent être effectuées et enregistrées au niveau de la source de données.

a. Patron de conception Unit of Work

Unit of Work est le nom d'un patron de conception dont l'objectif est le maintien d'un ensemble d'objets et la répercussion de l'ensemble des modifications qui leur sont apportées. Ces écritures sont généralement effectuées sous la forme d'un ensemble atomique, c'est-à-dire indivisible : lorsqu'une des actions échoue, l'ensemble échoue. Ce principe d'indivision est notamment connu des développeurs à travers l'exemple des transactions utilisées par une écrasante majorité de bases de données relationnelles.

Ce mode de fonctionnement est implémenté par la classe `DbContext` au travers d'un objet nommé Change Tracker, qui gère l'état des objets présents dans le contexte de données, ainsi que de la méthode `SaveChanges`. Comme son nom l'indique, cette dernière répercute vers la source de données toutes les modifications qui ont pu être effectuées sur les entités qui lui sont associées. Lorsque le fournisseur de données le permet, cette méthode utilise d'ailleurs les transactions de la source de données pour s'assurer du caractère atomique de la validation des changements.

b. Configuration de l'objet DbContext

Seul, l'objet `DbContext` n'est pas d'une grande utilité. En effet, par défaut il n'est lié à aucun fournisseur de données, et il ne peut donc pas être lié à une source de données "physique".

Il est par conséquent nécessaire de configurer le contexte de données en fournissant a minima les éléments qui sont essentiels pour son bon fonctionnement.

Types DbContextOptions et DbContextOptionsBuilder

Chaque contexte de données doit avoir une instance du type abstrait `DbContextOptions` pour être fonctionnel. C'est en effet cet objet qui porte l'ensemble des informations de configuration nécessaires à la bonne exécution du `DbContext`. Entity Framework Core permet de fournir ces options de deux manières différentes : en passant un objet de type `DbContextOptions` comme argument du constructeur du contexte, ou en le créant dynamiquement pendant la phase de configuration du contexte. Dans les deux cas, le développeur crée cet objet à l'aide d'un objet de type `DbContextOptionsBuilder`. Cette classe définit plusieurs fonctions dédiées à la configuration de services généraux utilisés pendant la vie du contexte de données.

- `ConfigureWarnings`

Définit une procédure à exécuter pour configurer le comportement des alertes levées par le framework.

- `EnableSensitiveDataLogging`

Autorise l'inclusion de données sensibles (informations enregistrées dans les entités) dans les logs du framework.

- `UseInternalServiceProvider`

Par défaut, Entity Framework Core crée un objet implémentant l'interface `IServiceProvider` pour la gestion interne des services utilisables par le contexte de données. Cette méthode permet de le remplacer par un autre objet.

- `UseLoggerFactory`

Définit un objet responsable de l'instanciation des objets de type `ILogger` utilisés par le contexte.

- `UseMemoryCache`

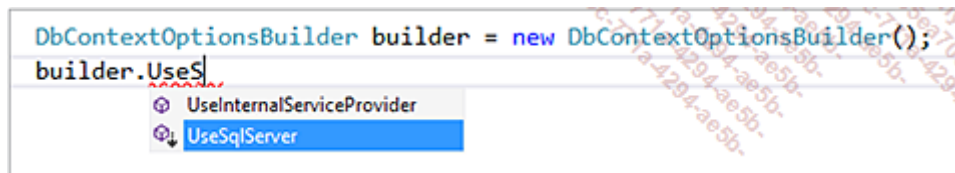
Par défaut, Entity Framework crée un objet implémentant l'interface `IMemoryCache` pour gérer la mise en cache des résultats de requêtes. Cet objet peut être remplacé par une nouvelle implémentation en utilisant cette méthode.

- `UseModel`

Spécifie un modèle objet à utiliser pendant la vie du contexte. Lorsque cette méthode est utilisée, l'étape de configuration du modèle est ignorée.

Il est important de noter que la plus importante des fonctionnalités, à savoir la configuration de la source de données à utiliser, n'est pas exposée par la classe `DbContextOptionsBuilder`. Cette tâche est en effet déléguée aux fournisseurs de données : ils étendent les fonctionnalités du type `DbContextOptionsBuilder` par le biais de méthodes d'extension.

Dans le cas du fournisseur SQL Server, et sous réserve qu'une clause `using Microsoft.EntityFrameworkCore;` soit présente dans le fichier qui définit le contexte de données, il est possible d'utiliser une méthode nommée `UseSqlServer` sur l'objet `DbContextOptionsBuilder` pour configurer la source de données.



Par convention, les fournisseurs implémentent tous cette méthode avec un nom similaire : pour SQLite, la méthode s'appelle `UseSqlite`, pour le fournisseur InMemory, elle s'appelle `UseInMemory`, etc.

Lier la configuration et le contexte

Le moyen le plus simple pour fournir un objet `DbContextOptions` à un contexte de données est d'utiliser le constructeur approprié à l'instanciation du contexte. Le type `DbContext` définit en effet deux constructeurs, dont un qui possède un paramètre de type `DbContextOptions`.

```
public DbContext(DbContextOptions options);  
protected DbContext();
```

Il suffit donc d'exposer publiquement le constructeur paramétré dans le contexte dérivé de manière à pouvoir passer les options de configuration sans difficulté.

```
using Microsoft.EntityFrameworkCore;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        string connectionString = "<ma chaine de connexion>";  
        DbContextOptionsBuilder builder = new DbContextOptionsBuilder();  
        builder.UseSqlServer(connectionString);  
  
        using (var context = new Context(builder.Options))  
        {  
            //...  
        }  
    }  
}  
  
public class Context : DbContext  
{  
    public Context(DbContextOptions options)  
        : base(options)  
    {  
    }  
}
```

Il est également possible d'intégrer la configuration directement dans le code de la classe contexte par le biais d'une procédure particulière : `OnConfiguring`. Cette méthode virtuelle est définie dans la classe `DbContext` et peut être surchargée par ses classes enfants. Elle prend comme paramètre un objet `DbContextOptionsBuilder` sur lequel on peut directement travailler, et elle ne nécessite pas de récupérer l'objet `DbContextOptions` sous-jacent, le framework s'occupant de cette tâche seule après l'exécution de la méthode.


```

class Program
{
    static void Main(string[] args)
    {
        using (var context = new Context())
        {
            //...
        }
    }
}

public class Context : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        string connectionString = "<ma chaine de connexion>";
        optionsBuilder.UseSqlServer(connectionString);

        base.OnConfiguring(optionsBuilder);
    }
}

```

2. Collections d'enregistrements : DbSet

Une source de données, qu'elle soit relationnelle ou non, est constituée d'enregistrements hétérogènes et généralement organisés en groupements. On parle de tables dans le cas de bases de données relationnelles ou de collections dans le cas de bases de données NoSQL (comme MongoDB).

Pour des raisons de simplicité, le terme "table" sera régulièrement utilisé tout au long de cet ouvrage.

Les données de la source sont exposées par les objets `DbContext` sous la forme de collections dont le type est `DbSet<T>`, où T est le type de l'entité. Lorsque le contexte est associé à une base de données, chacune de ces collections est liée à une table. Chaque fournisseur de données pouvant définir son propre modèle de structuration, il est tout à fait envisageable d'avoir une source de données au format CSV pour laquelle chaque collection correspond à un fichier, ou un annuaire LDAP pour lequel chaque type d'élément qu'il contient est associé à une collection .NET.

Pour associer une table à une classe `DbContext`, il suffit de définir une propriété dont le type est `DbSet<T>`. Le paramètre générique de ce type doit correspondre au type d'entité que l'on souhaite associer à la table.


```
using Microsoft.Data.Entity;

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

public class Client
{
}
```

Ici, l'ajout de la propriété `Clients` inclut dans le contexte le type `Client`, et crée automatiquement un lien entre ce type d'entité et une table nommée `Client`.

Nous verrons à la section Configuration du modèle objet - Mappage des tables et colonnes de ce chapitre qu'il existe plusieurs manières de configurer le nom de la table associée à un type d'entité.

3. Les enregistrements, des objets comme les autres

Les sections précédentes nous ont permis de définir l'organisation générale d'une source de données à l'aide des types `DbContext` et `DbSet<T>`, mais les entités n'ont encore aucune existence concrète. Ce sont pourtant elles qui permettent de manipuler les enregistrements de la source de données.

Chaque enregistrement peut être exposé avec Entity Framework Core sous la forme d'un objet .NET. Ces objets sont des instances de classes que l'on appelle "types d'entités". Les données correspondant aux différentes colonnes d'un enregistrement sont associées par défaut à des propriétés publiques du type d'entité correspondant à cet enregistrement.

Chaque propriété porte intrinsèquement deux informations utilisées pour la gestion du mappage entre une table et un type d'entité : son nom et son type.

À ce stade, il est intéressant de comprendre comment sont gérées les associations de types de données.

Chaque fournisseur a la responsabilité de définir les types de données qu'il supporte, tant au niveau des types d'entités que de la source de données sur laquelle il opère. Pour cela, une grille de correspondance entre les types .NET et les types de la source de données est mise en place. Cette table est un élément essentiel du fournisseur de données puisqu'elle est utilisée lors de la génération de chaque requête, qu'elle agisse en lecture ou en écriture sur la source de données. Son existence conditionne l'implémentation de spécificités propres à chaque source de données, comme la gestion du type SQL Server `rowversion`.

La table suivante liste les correspondances de types gérées par le fournisseur de données SQL Server.

Type SQL Server	Type .NET
varchar	string
nvarchar	string
char	string

Type SQL Server	Type .NET
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
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

Configuration du modèle objet

La configuration du modèle objet est une étape essentielle de la modélisation avec Entity Framework Core. C'est en effet à ce stade que sont définies les véritables caractéristiques de chaque élément du modèle, et par ricochet, de la base de données qui lui est associée.

1. Trois manières de faire

Avec Entity Framework Core, le développeur dispose de trois méthodes différentes pour définir la configuration du modèle.

Ci-après, la définition d'un modèle simple qui expose un type d'entité permettant de stocker des informations basiques relatives à un client : nom, prénom, adresse. Nous allons voir comment la configuration du modèle peut influencer sur la base de données qui lui est associée.

```
public class Context : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);

        string connectionString = "<ma chaine de connexion>";
        optionsBuilder.UseSqlServer(connectionString);
    }

    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 Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string ville { get; set; }
}
```

a. Conventions pour simplifier la modélisation

Entity Framework Core intègre nativement de nombreuses conventions destinées à simplifier au maximum la phase de modélisation. Elles couvrent une grande partie du spectre fonctionnel du framework puisqu'elles sont présentes du nommage des tables à la définition des types de colonnes, en passant notamment par les notions de clés primaires ou étrangères.

Lorsqu'un modèle est utilisé sans configuration particulière, ce sont les conventions qui sont utilisées pour établir des liens entre modèle et source de données.

La base de données (ici, SQL Server) associée automatiquement au modèle défini précédemment a une table nommée `Client` dont la structure correspond à l'instruction SQL `CREATE` ci-après.

```
CREATE TABLE [Client] (  
    [ClientId] int NOT NULL IDENTITY,  
    [Adresse1] nvarchar(max),  
    [Adresse2] nvarchar(max),  
    [Adresse3] nvarchar(max),  
    [CodePostal] nvarchar(max),  
    [Nom] nvarchar(max),  
    [Prenom] nvarchar(max),  
    [Ville] nvarchar(max),  
    CONSTRAINT [PK_Client] PRIMARY KEY ([ClientId])  
);
```

Quelques remarques peuvent être faites concernant la relation entre cette table et le modèle décrit précédemment :

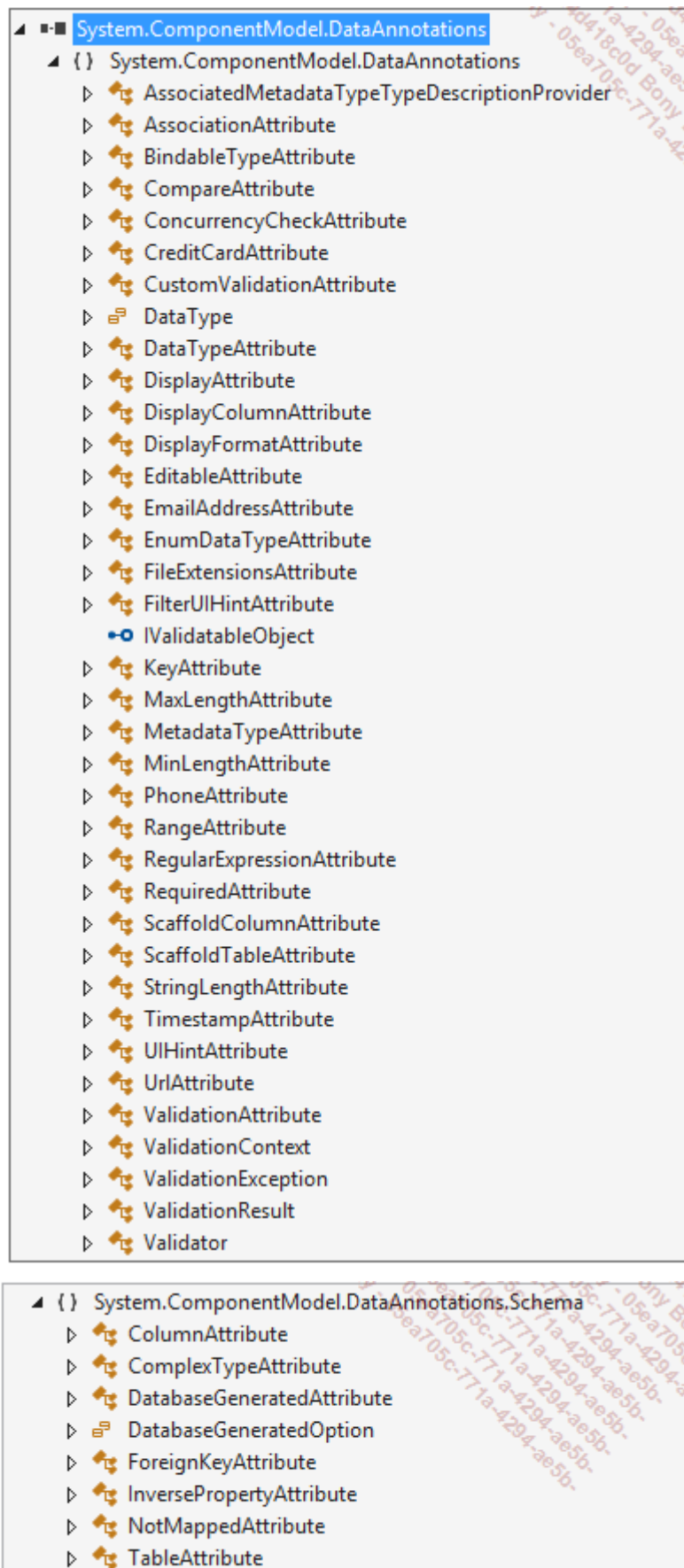
- Le type d'entité `Client` est lié à une table du même nom.
- La propriété `ClientId` est associée à une contrainte de clé primaire.
- Les propriétés de type `string` sont associées à des colonnes de type `nvarchar(max)`.
- Les propriétés du type d'entité et les colonnes de la table portent les mêmes noms.

Ces associations entre le modèle et la base de données sont définies automatiquement à l'aide des conventions d'Entity Framework. Celles-ci peuvent être surchargées à l'aide des data annotations et/ou de la Fluent API (interface de programmation fluide) mises à disposition des développeurs et intégrées à Entity Framework Core.

b. Data annotations

La configuration de modèles objets prend en compte des éléments descriptifs appelés data annotations. Ils se présentent sous la forme d'attributs placés sur les classes ou propriétés qui décrivent le modèle de données.

Les types utilisables dans le cadre des data annotations sont localisés au sein des espaces de noms `System.ComponentModel.DataAnnotations` et `System.ComponentModel.DataAnnotations.Schema`.



Avec l'aide des data annotations, il est possible de définir des comportements de manière simple et expressive. Le type `Client` peut ainsi être modifié pour spécifier le nom de la table et des colonnes qui lui sont associées.

```
[Table("Clients")]
public class Client
{
```

```

[Column("NUM_CLIENT")]
public int ClientId { get; set; }

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

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

[Column("ADR1_CLIENT")]
public string Adresse1 { get; set; }

[Column("ADR2_CLIENT")]
public string Adresse2 { get; set; }

[Column("ADR3_CLIENT")]
public string Adresse3 { get; set; }

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

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

```

Suite à ces modifications, Entity Framework associe désormais le type d'entité `Client` à une table nommée `Clients`. Ci-après le script SQL de création correspondant à cette table.

```

CREATE TABLE [Clients] (
    [NUM_CLIENT] int NOT NULL IDENTITY,
    [ADR1_CLIENT] nvarchar(max),
    [ADR2_CLIENT] nvarchar(max),
    [ADR3_CLIENT] nvarchar(max),
    [CP_CLIENT] nvarchar(max),
    [NOM_CLIENT] nvarchar(max),
    [PRENOM_CLIENT] nvarchar(max),
    [VILLE_CLIENT] nvarchar(max),
    CONSTRAINT [PK_Clients] PRIMARY KEY ([NUM_CLIENT])
);

```

La nature open source et le modèle d'extensibilité utilisé pour le développement d'Entity Framework Core rendent évolutive la liste des annotations : il est possible de créer et d'utiliser de nouvelles data annotations répondant à des besoins spécifiques. Ceci peut se révéler particulièrement utile, notamment lors du développement d'un fournisseur de données. Toutes les sources de données n'offrant pas exactement les mêmes fonctionnalités, il peut être souhaitable de créer des types permettant d'exploiter les comportements spécifiques. On peut notamment évoquer le cas de SQL Server, qui gère nativement un type de données nommé `xml`, dédié, comme son nom l'indique, au stockage spécifique de données au format XML dans une colonne. Aucune data annotation fournie avec Entity Framework Core ne permet à ce jour d'exploiter ce type de données.

c. Fluent API

L'utilisation de la Fluent API fournie par Entity Framework Core est la méthode la plus complexe pour configurer un modèle car elle nécessite de véritablement écrire du code, avec toutes les contraintes de qualité et de testabilité que cela peut entraîner pour une équipe de développement. En contrepartie, elle se révèle être la plus puissante des trois techniques de configuration, notamment parce qu'elle couvre un spectre fonctionnel plus important que les data annotations.

À ce stade, vous pouvez légitimement vous demander où doit se trouver le code de configuration utilisant la Fluent API, et vous avez raison. Nous avons vu avec les data annotations qu'il était nécessaire de décrire les comportements souhaités dans le code des types d'entités. Avec la Fluent API, toute cette pollution peut disparaître de sorte que le modèle ne soit constitué que de classes "simples" : définition du type et des propriétés associées. La configuration est effectuée au niveau de la classe `DbContext` qui utilise ces types d'entités. Ceci ouvre la porte à des scénarios de partage de code bien plus évolués, puisque les mêmes types d'entités peuvent être utilisés par différents objets `DbContext` qui peuvent eux-mêmes être associés à des sources de données complètement différentes.

Il est fréquent aujourd'hui que des applications Windows Store soient couplées à un backend distant accessible via des API web. Ces applications peuvent disposer d'une copie locale des données dans une base de données SQLite, tandis que le service web accède à une base de données SQL Server. Pour ce type de scénarios, la Fluent API peut être un véritable atout qu'il convient de prendre en considération puisqu'elle permet de définir les classes du modèle dans un assembly partagé par l'application Windows et l'application web. Lorsque le nombre d'entités commence à augmenter fortement, c'est autant de temps de développement et de maintenance qui peut être gagné.

La portion de code ci-après configure le type d'entité `Client` à l'aide de la Fluent API de sorte qu'il soit associé à une table nommée `Clients` et limite la longueur de la chaîne de caractères stockée dans la colonne `CodePostal`.

```
class Context : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Client>()
            .ToTable("Clients");

        modelBuilder.Entity<Client>()
            .Property<string>(nameof(Client.CodePostal))
            .HasMaxLength(5);
    }

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

Ce dernier exemple est associé à la table SQL Server décrite par l'instruction `CREATE` suivante :

```
CREATE TABLE [Clients] (
    [ClientId] int NOT NULL IDENTITY,
    [Adresse1] nvarchar(max),
    [Adresse2] nvarchar(max),
    [Adresse3] nvarchar(max),
    [CodePostal] nvarchar(5),
    [Nom] nvarchar(max),
    [Prenom] nvarchar(max),
    [Ville] nvarchar(max),
    CONSTRAINT [PK_Client] PRIMARY KEY ([ClientId])
);
```

d. Combiner les techniques de configuration

Entity Framework Core offre la possibilité aux développeurs de combiner les modes de configuration. Dans l'écrasante majorité des cas de configurations explicites (par l'utilisation des data annotations ou de la Fluent API), une combinaison est effectuée, presque à l'insu du développeur. En effet, Entity Framework Core utilise les conventions avant tout, et le développeur ne surcharge généralement que quelques-unes d'entre elles à l'aide d'attributs ou de code. Dans le cas de l'exemple d'utilisation des data annotations précédent, on peut constater que malgré la modification des noms de champs associés aux propriétés, leur longueur maximale autorisée est toujours `max`. Cette dernière valeur étant attribuée par convention, on a une preuve concrète que la combinaison conventions/annotations est utilisable. La même démonstration peut être faite pour l'exemple concernant la Fluent API. Dans ces deux cas, on peut noter que la configuration explicite est toujours prioritaire par rapport aux conventions.

Un troisième cas n'est pourtant pas couvert : la combinaison conventions/data annotations/Fluent API. Lorsque ces trois techniques sont utilisées en conjonction, la règle utilisée est la suivante :

La Fluent API est prioritaire sur les data annotations et les conventions. Les data annotations sont prioritaires sur les conventions. Enfin, les conventions sont appliquées pour tous les éléments qui ne sont pas spécifiés explicitement.

Le modèle objet suivant utilise les trois modes de configuration disponibles.

```
class Context : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Client>()
            .Property<string>(nameof(Client.CodePostal))
            .HasMaxLength(5);

        modelBuilder.Entity<Client>()
            .Property<string>(nameof(Client.ville))
            .HasColumnType("nvarchar(100)");
    }

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

public class Client
```



```

{
    [Column("NumClient")]
    public int ClientId { get; set; }

    [RequiredAttribute]
    public string Nom { get; set; }

    [RequiredAttribute]
    public string Prenom { get; set; }

    public string Adresse1 { get; set; }

    public string Adresse2 { get; set; }

    public string Adresse3 { get; set; }

    public string CodePostal { get; set; }

    public string ville { get; set; }
}

```

Le script SQL décrivant la table associée présente quelques éléments notables :

- La table contient une colonne nommée `NumClient`, comme spécifié par l'attribut `Column` placé sur la propriété `ClientId`.
- Les colonnes `Nom` et `Prenom` sont marquées comme `NOT NULL`, ce qui est indiqué par l'attribut `Required` placé sur les propriétés correspondantes.
- La colonne `CodePostal` a une longueur maximale de 5 caractères, comme spécifié par le code de configuration placé dans la classe `DbContext`.
- La colonne `ville` a une longueur maximale de 100 caractères, ce qui est spécifié par le code de la méthode `DbContext.OnModelCreating`. On remarque que l'attribut `MaxLength` placé sur la propriété n'est pas pris en compte, car surchargé par la Fluent API.

```

CREATE TABLE [Clients] (
    [NumClient] int NOT NULL IDENTITY,
    [Adresse1] nvarchar(max),
    [Adresse2] nvarchar(max),
    [Adresse3] nvarchar(max),
    [CodePostal] nvarchar(5),
    [Nom] nvarchar(max) NOT NULL,
    [Prenom] nvarchar(max) NOT NULL,
    [ville] nvarchar(100),
    CONSTRAINT [PK_Clients] PRIMARY KEY ([NumClient])
);

```

2. Inclusion/Exclusion de types d'entités

Au début de la section Configuration du modèle objet de ce chapitre, nous avons écrit un modèle objet simpliste qui définit en tout et pour tout un seul type d'entité : `Client`.

```

public class Context : DbContext
{

```

```

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    base.OnConfiguring(optionsBuilder);

    string connectionString = "<ma chaine de connexion>";
    optionsBuilder.UseSqlServer(connectionString);
}

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 Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string ville { get; set; }
}

```

Une des conventions d'Entity Framework Core spécifie que les propriétés d'instance publiques de type `DbSet<T>` présentes dans l'objet `DbContext` représentent des types d'entités. Dans notre cas, le type `Client` est associé au modèle par la présence d'une propriété de type `DbSet<Client>`.

Cette convention d'inclusion est implémentée par le service `DbSetFinder` localisé dans l'assembly `Microsoft.EntityFrameworkCore`.

Il n'est toutefois pas obligatoire de créer une propriété `DbSet<T>` pour effectuer cette inclusion. Les différentes surcharges, génériques ou non, de la méthode `ModelBuilder.Entity` ajoutent un type d'entité à un modèle de manière programmatique. Elles doivent nécessairement être utilisées avant l'instanciation interne du modèle : le bon emplacement pour ces appels est à l'intérieur de la méthode `onModelCreating` du contexte.

```

public class Context : DbContext
{
    protected override void OnConfiguring(...) { ... }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Client>();

        //La ligne suivante est exactement équivalente.
        //modelBuilder.Entity(typeof(Client));

        base.OnModelCreating(modelBuilder);
    }
}

```

Ces deux techniques d'inclusion offrent chacune des avantages et des inconvénients :

- Les propriétés de type `DbSet<T>` sont des points d'entrée idéaux pour toute exécution de requête sur les types d'entités qui leur sont associés : elles sont identifiables rapidement et simplement. En revanche, elles sont toujours présentes dans leur contexte de données, ce qui peut se révéler problématique lorsqu'il doit être dynamique.
- L'inclusion programmatique nécessite évidemment d'écrire plus de code, avec toutes les implications que cela peut avoir en termes de qualité et de maintenabilité. De plus, 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.

Entity Framework Core est également capable de détecter les types d'entités ajoutés indirectement au modèle. Considérons l'entité `Client` définie ci-après :

```
public class Client
{
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }

    public CodePostalVille Localite { get; set; }
}

public class CodePostalVille
{
    public int Id { get; set; }
    public string CodePostal { get; set; }
    public string ville { get; set; }
}
```

Suite à l'association du type d'entité `Client` au modèle de données, Entity Framework Core recherche les propriétés qui lui sont associées. Lorsque le type de l'une d'elles n'est pas reconnu par le fournisseur de données comme étant primitif, c'est-à-dire supporté directement par la source de données, le moteur le considère comme un type d'entité et l'enregistre comme tel.

Le type `CodePostalVille` peut donc être utilisé de la même manière que tout type d'entité enregistré dynamiquement, c'est-à-dire en appelant la fonction `DbContext.Set<CodePostalVille>()`.

La convention de recherche des propriétés dont le type est considéré comme primitif est implémentée par le type `PropertyDiscoveryConvention`. Ce type n'est exposé qu'au travers du service `CoreConventionBuilderSet` et ne devrait être manipulé que pour l'écriture d'un fournisseur de données.

L'inclusion automatique de tous les types non primitifs dans le modèle n'est toutefois pas toujours envisageable. Certaines données peuvent être stockées temporairement dans une entité et ne pas être enregistrées dans une source de données.

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

```

    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string Ville { get; set; }

    public DonneesTemporaires Temp { get; set; }
}

public class DonneesTemporaires
{
    public string AdresseIP { get; set; }
    public string PrenomsEnfants { get; set; }
}

```

Comme son nom l'indique, le type `DonneesTemporaires` contient des informations volatiles que nous n'avons aucune raison d'envoyer à la source de données. Malheureusement, les conventions d'Entity Framework Core l'incluent automatiquement dans le modèle de données du contexte. Pour pallier ce comportement non souhaité, il suffit de décorer le type à exclure du modèle d'un attribut de type `NotMappedAttribute`, localisé dans l'espace de noms `System.ComponentModel.DataAnnotations.Schema`.

```

[NotMapped]
public class DonneesTemporaires
{
    public string AdresseIP { get; set; }

    public string PrenomsEnfants { get; set; }
}

```

Pour exclure un type du modèle de données associé à un contexte, il est également possible d'utiliser l'API de configuration : le type `ModelBuilder` implémente la fonction `Ignore` (sous une forme générique ou non) qui indique explicitement au moteur que le type ne doit pas être enregistré.

```

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

```

3. Inclusion/Exclusion de propriétés

Nous l'avons déjà vu, l'inclusion de propriétés d'un type d'entité dans le modèle objet d'un contexte de données est automatique. La seule règle à respecter pour que l'inclusion d'une propriété soit possible est la suivante : elle doit être publique et accessible en lecture et écriture (`get` et `set`). Ainsi, parmi les propriétés suivantes, seule la seconde pourrait être incluse dans un modèle objet Entity Framework Core.

```

public string Adresse { get; }
public string Adresse { get; set; }
private string Adresse { get; set; }

```

L'exclusion de propriétés nécessite, quant à elle, une indication explicite qui peut prendre la forme de la data annotation [NotMapped] ou de l'appel à la fonction Ignore définie sur le type EntityTypeBuilder. La portion de code ci-après montre comment exclure du modèle les propriétés Adresse2 et Adresse3, respectivement avec la data annotation et avec la Fluent API.

```

public class Context : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        //...
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Client>()
            .Ignore(client => client.Adresse3);

        base.OnModelCreating(modelBuilder);
    }

    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 Adresse1 { get; set; }

    [NotMapped]
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }

    public string CodePostal { get; set; }
    public string ville { get; set; }
}

```

4. Mappage des tables et colonnes

Le lien principal entre un élément (classe ou propriété) du modèle objet et la source de données est défini par l'association de cet élément et d'un nom correspondant à son équivalent au niveau de la source de données.

a. Types d'entités et tables

La responsabilité de la génération du nom de table est déléguée aux fournisseurs de données. Les fournisseurs destinés à l'utilisation de sources de données relationnelles comme SQL Server ou SQLite définissent par défaut que les types d'entités et les tables qui leur sont associées ont des noms identiques.

C'est le type `RelationalEntityTypeAnnotations`, localisé dans l'assembly `Microsoft.EntityFrameworkCore.Relational`, qui est responsable de cette association conventionnelle pour les fournisseurs relationnels supportés par Microsoft.

Ce comportement peut bien évidemment être surchargé explicitement, permettant ainsi de se conformer à des conventions d'équipe ou d'entreprise ou d'associer un modèle objet à une source de données préexistante.

L'utilisation de la data annotation `Table`, dans l'espace de noms `System.ComponentModel.DataAnnotations.Schema`, est le premier moyen d'atteindre ce but. Elle accepte un paramètre de type `string` dont la valeur est le nom souhaité pour la table associée au type d'entité. Ainsi, pour associer le type `Client` à la table `Clients`, on peut décorer le type de la manière suivante.

```
[Table("Clients")]
public class Client
{
    ///...
}
```

Le type `TableAttribute` possède une propriété `Schema` qui permet d'indiquer le schéma associé à la table. Comme avec tous les attributs, on peut valoriser cette propriété directement en la nommant.

```
[Table("Clients", Schema = "dbo")]
public class Client
{
    ///...
}
```

La Fluent API n'est évidemment pas en reste, avec une méthode d'extension `ToTable` définie sur le type `EntityTypeBuilder`. Ce dernier est également le type de retour de la fonction `Entity` vue précédemment, ce qui permet d'en déduire que l'appel à `ToTable` peut être chaîné à l'appel de `ModelBuilder.Entity<T>`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        .ToTable("Clients");

    base.OnModelCreating(modelBuilder);
}
```

La fonction `ToTable` a plusieurs surcharges, génériques ou non, qui offrent les mêmes possibilités que l'attribut `TableAttribute`. Ainsi, on peut utiliser les quatre écritures suivantes pour aboutir à la configuration du type d'entité.

```
modelBuilder.Entity<Client>().ToTable("Clients");
modelBuilder.Entity<Client>().ToTable("Clients", "dbo");
modelBuilder.Entity<Client>().ToTable<Client>("Clients");
modelBuilder.Entity<Client>().ToTable<Client>("Clients", "dbo");
```

b. Propriétés et colonnes

Comme pour les associations types d'entités/tables, la génération du nom d'une colonne liée à une propriété est déléguée au fournisseur de données. Dans le cas des fournisseurs relationnels supportés par Microsoft, la convention utilisée est simple : la propriété et la colonne portent par défaut le même nom.

Cette convention est implémentée au niveau du type `InternalEntityTypeBuilder`, et utilisée au niveau du type `PropertyDiscoveryConvention`. Ces deux types sont inclus dans l'assembly `Microsoft.EntityFrameworkCore`.

L'attribut `ColumnAttribute` redéfinit la valeur conventionnelle attribuée par Entity Framework par la valorisation de sa propriété `Name`. Pour modifier le nom de la colonne associée à la propriété `Adresse1`, il suffit donc de la décorer de la manière suivante :

```
[Column("Adresse")]
public string Adresse1 { get; set; }
```

Cette data annotation peut également être utilisée pour aider à gérer explicitement le type de données que la source doit utiliser pour la colonne. En effet, lorsque la base de données doit être créée à partir du modèle objet, le type de données défini par les conventions d'Entity Framework peut ne pas être en adéquation avec la structure souhaitée au niveau de la source de données.

L'attribut `ColumnAttribute` permet alors d'indiquer explicitement le type de données qui doit être utilisé lors de l'étape de création de la source de données grâce à sa propriété `TypeName`.

```
[Column(TypeName = "nvarchar(120)")]
public string Adresse1 { get; set; }
```

La convention de mappage des types de données est définie au niveau des fournisseurs de données. Pour les fournisseurs relationnels, elle est implémentée en partie par le type `RelationalTypeMapper` de l'assembly `Microsoft.EntityFrameworkCore.Relational` et en partie par les types qui en héritent au niveau des fournisseurs (`SqlServerTypeMapper` ou `SqliteTypeMapper`, par exemple).

La fonction générique `Property<T>` implémentée sur le type `EntityTypeBuilder` renvoie une référence sur une propriété à travers le type `EntityTypeBuilder`, offrant par conséquent la possibilité de manipuler sa configuration. Pour agir sur le nom de la colonne qui lui est associée, on utilise la fonction `EntityTypeBuilder.HasColumnName` qui accepte un paramètre de type `string`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        .Property(c => c.Adresse1)
        .HasColumnName("Adresse");

    base.OnModelCreating(modelBuilder);
}
```

Pour indiquer explicitement le type de données utilisé, on tire parti de la fonction `EntityTypeBuilder.HasColumnName`, dont le paramètre unique correspond au nom du type à utiliser au niveau de la source de données.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<ClientALaCon>()
        .Property(c => c.Adresse1)
        .HasColumnType("nvarchar(120)");

    base.OnModelCreating(modelBuilder);
}
```

5. Définition d'une clé primaire

La clé primaire d'un type d'entité est une propriété, ou un groupe de propriétés, qui contient une valeur unique identifiant chaque instance de ce type. À l'aide de cette clé, il est ainsi possible de manipuler des objets existants de manière unitaire. C'est pour cette raison qu'Entity Framework requiert que chaque type d'entité possède une propriété définie comme clé primaire.

Par convention, pour chaque type, Entity Framework considère qu'une propriété nommée `Id` ou `<Nom de la classe>Id` représente la clé primaire de l'entité.

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

    public string Nom { get; set; }
}
```

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

    public string Nom { get; set; }
}
```

Les classes `Produit` et `Client` définissent toutes deux une clé primaire utilisant les conventions d'Entity Framework. Pour la première, c'est la propriété `Id` qui est prise en compte tandis que la seconde entité a pour clé primaire la propriété `ClientId`.

Lorsqu'une entité définit deux propriétés qui répondent aux attentes d'Entity Framework en termes de conventions, le système considère comme clé primaire la première des propriétés découvertes.

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

    public string Nom { get; set; }
}
```

Ici, la clé primaire est représentée par la propriété `ClientId`. Si l'ordre de déclaration des propriétés `Id` et `ClientId` est inversé, c'est `Id` qui prendra le rôle de clé primaire.

La convention de recherche des clés primaires est implémentée par le type `KeyDiscoveryConvention`. Ce type n'est exposé qu'au travers du service `CoreConventionBuilderSet` et ne devrait être manipulé que pour l'écriture d'un fournisseur de données.

Par convention, les clés primaires de type `int` ou `Guid` sont configurées pour que leur valeur soit générée automatiquement à l'ajout de l'objet au contexte de données.

Data annotations

L'attribut `KeyAttribute` permet de redéfinir explicitement ce comportement en le plaçant sur une des propriétés de l'entité. Ainsi, toute autre propriété qui correspond aux conventions d'Entity Framework est ignorée.

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

    [Key]
    public int MaClePrimaire { get; set; }

    public string Nom { get; set; }
}
```

Le rôle de clé primaire peut être attribué de cette manière à une seule propriété par entité. Lorsque plusieurs annotations `Key` sont présentes au niveau du même type d'entité, l'intégration d'un objet de ce type au contexte de données lève automatiquement une exception de type `System.InvalidOperationException`.

Fluent API

La configuration de la clé primaire d'une entité est effectuée à l'aide de la fonction `HasKey` définie sur le type `EntityTypeBuilder<T>`. Pour attribuer le rôle de clé primaire à la propriété `MaClePrimaire`, on peut écrire le code suivant.

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

    base.OnModelCreating(builder);
}
```

La Fluent API autorise également la création d'une clé primaire composite. Pour cela, l'expression lambda passée en paramètre à la fonction `HasKey` doit retourner un objet de type anonyme contenant les différentes propriétés concernées.

La clé primaire composée des propriétés `Id` et `MaClePrimaire` est définie de la manière suivante.

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

    base.OnModelCreating(builder);
}
```

6. Relations

Les capacités de modélisation d'Entity Framework Core incluent la gestion de relations entre types d'entités. Ces relations, généralement définies au niveau des sources de données par des clés étrangères, sont modélisées avec Entity Framework Core sous la forme de propriétés de navigation.

Commençons par quelques définitions relatives aux différents éléments qui seront utilisés dans cette section. Chacun de ces éléments sera détaillé au fur et à mesure.

- Entité principale : dans une relation entre deux entités, c'est l'entité qui est référencée. On peut aussi la qualifier d'entité parente.
- Clé principale : c'est une propriété qui identifie de manière unique l'entité principale. Elle est associée au rôle de clé primaire ou de clé alternative dans cette entité.
- Entité dépendante : dans une relation entre deux entités, c'est l'entité qui porte une référence vers l'autre. On peut également la qualifier d'entité enfant.
- Clé étrangère : c'est une propriété de l'entité dépendante qui contient la valeur d'une clé principale de manière à la référencer.
- Propriété de navigation : c'est une propriété définie dans l'un des types d'entités impliqués dans la relation, et qui contient une ou plusieurs références vers l'autre type d'entité.

Le code ci-après montre la modélisation typique d'une relation entre deux entités : `Commande` et `Client`. Chaque commande n'est liée qu'à un client, et chaque client est associé à plusieurs commandes.

```
public class Client
```

```

{
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string Ville { get; set; }

    public List<Commande> Commandes { get; set; }
}

public class Commande
{
    public int CommandeId { get; set; }
    public decimal MontantTotalHT { get; set; }
    public decimal MontantTVA { get; set; }

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

```

Les différents éléments définis précédemment sont présents dans cet exemple :

- Entité principale : le type `Client`, qui est référencé par la classe `Commande`.
- Clé principale : la propriété `ClientId`.
- Entité dépendante : la classe `Commande`, qui dépend de l'entité `Client`.
- Clé étrangère : la propriété `Commande.ClientId`, qui contient l'identifiant d'une instance de `Client`.
- Propriétés de navigation : deux propriétés de navigation sont présentes dans ce modèle. `Commande.Client` référence une instance de la classe `Client` qui correspond à la valeur de `Commande.ClientId`. Le type `Client` a, quant à lui, une propriété `Commandes` qui contient l'ensemble des objets de type `Commande` qui lui sont associés.

Cette implémentation utilise de multiples conventions définies par Entity Framework Core. En premier lieu, la relation est découverte par le framework grâce à la présence d'une propriété de navigation. Ici, le modèle en contient deux (`Commande.Client` et `Client.Commandes`), qui sont automatiquement détectées comme représentant les deux extrémités de la même relation.

Les propriétés de navigation sont des propriétés dont le type ne peut être résolu comme étant un type primitif de la source de données.

Une remarque importante : Entity Framework est parfaitement capable de configurer les propriétés de navigation détectées comme appartenant à la même relation, mais n'y parvient pas lorsque plusieurs propriétés de navigation pointant vers le même type d'entité existent à l'intérieur d'un type.

Suite à la découverte de la relation, la propriété `Commande.ClientId` est configurée automatiquement comme clé étrangère. Sa découverte et la configuration qui s'ensuit sont liées à une convention de nommage : toute propriété de l'entité dépendante (ici donc, `Commande`) dont le nom correspond à l'un des schémas suivants est enregistrée comme une clé étrangère :

<nom de la clé primaire de l'entité principale>

<nom de la propriété de navigation><nom de la clé primaire de l'entité principale>

<nom de l'entité principale><nom de la clé primaire de l'entité principale>

Ainsi, dans le cas qui nous intéresse, une propriété nommée `ClientId`, `ClientsClientId` ou `ClientClientId` serait considérée comme une clé étrangère associée à la relation Client <--> Commande découverte plus tôt.

Même si l'existence d'une clé étrangère est obligatoire au niveau de la source de données (avec ou sans contrainte d'intégrité associée), sa présence n'est aucunement obligatoire dans l'entité dépendante du modèle objet. En effet, ce cas de figure est parfaitement géré par Entity Framework Core à l'aide d'une petite astuce : le moteur génère une propriété virtuelle, communément appelée shadow property, qui est enregistrée comme clé étrangère. Son nom est conforme au deuxième schéma évoqué précédemment, et le type correspondant est déduit du type de la clé primaire de l'entité principale.

Dans notre cas, la suppression de la propriété `Commande.ClientId` dans le modèle déclenche à l'exécution la création d'une propriété virtuelle `ClientsClientId` de type `int`.

Toutes ces conventions sont définies par le type `RelationshipDiscoveryConvention`, présent dans l'assembly `Microsoft.EntityFrameworkCore`.

Data annotations

Le comportement par défaut concernant les relations peut bien évidemment être surchargé, notamment à l'aide de data annotations. Deux attributs peuvent ainsi être utilisés pour configurer une relation.

Le premier identifie une clé étrangère de manière explicite, lorsque le nommage de la propriété associée ne lui permet pas d'être enregistrée de manière automatique. Dans l'exemple suivant, la propriété `IdClient` ne peut être découverte par convention car elle ne respecte aucun des schémas de nommage reconnus par Entity Framework Core. La propriété de navigation est donc annotée avec un attribut de type `ForeignKeyAttribute` de façon à être associée à `IdClient`.

```
public class Commande
{
    public int CommandeId { get; set; }
    public decimal MontantTotalHT { get; set; }
    public decimal MontantTVA { get; set; }

    public int IdClient { get; set; }

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

L'annotation `[InverseProperty]` permet, à partir d'une extrémité d'une relation, d'identifier explicitement la propriété associée à l'autre extrémité de la relation. Elle permet notamment de démêler une situation problématique dans laquelle plusieurs propriétés sont candidates pour l'appartenance à une relation à l'intérieur d'un même type. Supposons qu'à chaque commande

puissent être associés deux clients : un client acheteur et un client chez qui la livraison est faite. Le modèle pourrait être modifié comme suit :

```
public class Client
{
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string ville { get; set; }

    public List<Commande> CommandesAchat { get; set; }
    public List<Commande> CommandesLivraison { get; set; }
}

public class Commande
{
    public int CommandeId { get; set; }
    public decimal MontantTotalHT { get; set; }
    public decimal MontantTVA { get; set; }

    public int ClientAchatClientId { get; set; }
    public Client ClientAchat { get; set; }

    public int ClientLivraisonClientId { get; set; }
    public Client ClientLivraison { get; set; }
}
```

La découverte de relations par convention ne peut pas être faite dans ce cas puisque le moteur d'Entity Framework Core est incapable de configurer les relations lorsque plusieurs propriétés de navigation pointent vers un même type d'entité. En l'occurrence, les propriétés `Client.CommandesAchat` et `Client.CommandesLivraison` se télescopent et bloquent le processus de découverte.

Une solution à ce problème est de décorer chacune de ces deux propriétés de navigation avec un attribut de type `InversePropertyAttribute` indiquant le nom de la propriété de navigation qui lui est associée dans le type `Commande`.

```
public class Client
{
    public int ClientId { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string ville { get; set; }

    [InverseProperty("ClientAchat")]
    public List<Commande> CommandesAchat { get; set; }
```

```
[InverseProperty("ClientLivraison")]
public List<Commande> CommandesLivraison { get; set; }
}
```

Fluent API

Pour configurer une relation entre deux types d'entités avec la Fluent API, la première étape est l'identification des propriétés de navigation impliquées.

Pour indiquer la première partie de la relation, le type `EntityTypeBuilder` implémente les fonctions `HasOne` et `HasMany` qui permettent respectivement d'enregistrer une propriété de navigation simple, qui correspond à une seule entité, et une propriété de navigation représentant une collection d'entités.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Commande>()
        .HasOne(commande => commande.Client);

    modelBuilder.Entity<Client>()
        .HasMany(client => client.Commandes);
}
```

Ces méthodes renvoient toutes deux un objet qui permet de créer une référence vers l'autre extrémité de la relation à l'aide des fonctions `WithOne` et `WithMany`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Commande>()
        .HasOne(commande => commande.Client)
        .WithMany(client => client.Commandes);

    modelBuilder.Entity<Client>()
        .HasMany(client => client.Commandes)
        .WithOne(commande => commande.Client);
}
```

Il est à noter que `WithOne` et `WithMany` peuvent toutes deux être utilisées sans argument. Elles indiquent dans ce cas qu'il existe conceptuellement une propriété de navigation à l'autre extrémité de la relation, mais que cette propriété n'a pas d'existence concrète. Ceci permet donc de définir une relation avec une seule propriété de navigation.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Commande>()
        .HasOne(commande => commande.Client)
        .WithMany();

    modelBuilder.Entity<Client>()
        .HasMany(client => client.Commandes)
        .WithOne();
}
```

Il est possible de chaîner à la suite de ces instructions un appel à la fonction `HasForeignKey` qui indique explicitement la propriété qui doit être configurée comme clé étrangère du côté "Many" de la relation (ici, le type d'entité `Commande`).

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Commande>()
        .HasOne(commande => commande.Client)
        .WithMany(client => client.Commandes)
        .HasForeignKey(commande => commande.ClientId);

    modelBuilder.Entity<Client>()
        .HasMany(client => client.Commandes)
        .WithOne(commande => commande.Client)
        .HasForeignKey(commande => commande.ClientId);
}
```

À ce stade, la relation entre les deux types d'entités est configurée et fonctionnelle. Il est toutefois possible d'aller plus loin en ajoutant différents éléments, comme une clé principale alternative ou un nom pour la contrainte d'intégrité associée.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Commande>()
        .HasOne(commande => commande.Client)
        .WithMany(client => client.Commandes)
        .HasPrincipalKey(client => client.ClientId)
        .HasConstraintName("Client_Commande_FK");

    modelBuilder.Entity<Client>()
        .HasMany(client => client.Commandes)
        .WithOne(commande => commande.Client)
        .HasPrincipalKey(client => client.ClientId)
        .HasConstraintName("Client_Commande_FK"); ;
}
```

Une propriété configurée comme clé alternative correspond, en termes relationnels, à une colonne sur laquelle est appliquée une contrainte d'unicité. Elle peut donc être utilisée comme point de terminaison pour la relation. En revanche, il faut savoir que l'utilisation d'une propriété qui n'est pas une clé alternative dans une relation entraîne automatiquement un ajout de cette contrainte au niveau du modèle objet en mémoire.

Les clés alternatives sont évoquées plus en détail à la section éponyme de ce chapitre.

La Fluent API autorise également la configuration du caractère optionnel ou obligatoire d'une relation. Ceci se traduit dans les faits par l'autorisation ou non d'une valeur nulle pour la clé étrangère associée à la relation. Cette contrainte est imposée par l'utilisation de la fonction `IsRequired` qui accepte un argument booléen ayant une valeur par défaut égale à `true`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Commande>()
        .HasOne(commande => commande.Client)
        .WithMany(client => client.Commandes)
        .IsRequired();

    modelBuilder.Entity<Client>()
        .HasMany(client => client.Commandes)
        .WithOne(commande => commande.Client)
        .IsRequired();
}
```

Le dernier élément important qu'il est possible de configurer de cette façon est le comportement que le moteur doit adopter lors de la suppression d'une entité principale. Trois choix s'offrent au développeur : la suppression en cascade, la mise à la valeur `null` des clés étrangères, ou tout simplement la suppression sans toucher aux entités dépendantes.

Cette opération est implémentée par la méthode `onDelete`, qui accepte un argument dont la valeur peut être `DeleteBehavior.Cascade`, `DeleteBehavior.SetNull` ou `DeleteBehavior.Restrict`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Commande>()
        .HasOne(commande => commande.Client)
        .WithMany(client => client.Commandes)
        .onDelete>DeleteBehavior.Cascade);

    modelBuilder.Entity<Client>()
        .HasMany(client => client.Commandes)
        .WithOne(commande => commande.Client)
        .onDelete>DeleteBehavior.Cascade);
}
```

7. Configuration avancée des propriétés du modèle

De nombreux éléments complémentaires peuvent être contrôlés par les modèles de données Entity Framework Core. Ils permettent de valider certaines caractéristiques des données, comme leur existence ou leur longueur, mais peuvent également agir sur la source de données, par la création d'index ou de contraintes d'unicité.

a. Données obligatoires et facultatives

Bon nombre de sources de données, notamment relationnelles, autorisent le contrôle du caractère obligatoire ou non des données qu'elles peuvent contenir. Les bases de données telles que SQL Server, MySQL ou SQLite connaissent notamment la valeur spéciale `NULL`, qui définit l'inexistence d'une donnée pour une colonne d'un enregistrement.

Dans le modèle objet d'Entity Framework, cette absence de donnée est représentée par la valeur spéciale `null`. Les propriétés optionnelles, dont la colonne associée peut ne pas contenir de valeur, doivent pouvoir contenir `null`. Pour cette raison, les propriétés de type valeur (comme `int`, `double`, `bool` ...) sont toujours considérées comme obligatoires, même s'il est indiqué

explicitement qu'elles sont optionnelles. Les propriétés pouvant contenir `null` sont, quant à elles, définies par convention comme étant optionnelles. Ainsi, le type `Client` défini comme suit possède trois propriétés obligatoires, dont sa clé primaire, et sept propriétés optionnelles.

```
public class Client
{
    //Ces propriétés sont obligatoires.
    public int ClientId { get; set; }
    public decimal MontantAchatsTotal { get; set; }
    public bool EstPremium { get; set; }

    // Ces propriétés sont optionnelles.
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string Ville { get; set; }
    public byte[] Photo { get; set; }
}
```

Il existe tout de même une technique simple pour définir une propriété de type valeur comme optionnelle. Le type de retour de la propriété doit, pour cela, être encapsulé à l'aide du type générique `Nullable<T>`. Dans le cas d'un client, le montant total de ses achats peut ne pas avoir de valeur, notamment entre le moment de son enregistrement dans la source de données et celui de sa première commande. La définition de cette propriété devient alors :

```
public Nullable<decimal> MontantAchatsTotal { get; set; }
```

À titre de rappel, en C#, un raccourci syntaxique pour le type `Nullable<T>` est intégré au langage. Il s'agit de l'opérateur `?` placé en position de suffixe pour un type.

```
public decimal? MontantAchatsTotal { get; set; }
```

La propriété `MontantAchatsTotal` ainsi typée est maintenant éligible au rang de propriété optionnelle, et ce, par défaut.

Pour ce qui est de l'opération inverse, à savoir rendre une propriété obligatoire alors que son type accepte les valeurs nulles, il faut recourir à l'usage d'une data annotation : `[Required]`. Cet attribut est simpliste puisqu'il n'accepte aucun paramètre, mais il n'en est pas moins efficace pour résoudre notre problème. La déclaration annotée des propriétés `Nom` et `Prenom` est la suivante :

```
[Required]
public string Nom { get; set; }
[Required]
public string Prenom { get; set; }
```

L'équivalent impératif de cette annotation est implémenté dans le type `EntityTypeBuilder`, sous la forme d'une méthode nommée `IsRequired`.

```
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);
}
```

b. Longueur maximale

Dans Entity Framework Core, la configuration de la longueur maximale d'un champ est utilisée lors de la phase de création d'une base de données. Elle fournit une indication permettant au fournisseur de déterminer le type adéquat pour le stockage de la valeur d'une propriété. Le moteur d'Entity Framework n'effectue pas de validation du respect de cette longueur maximale avant la transmission à la source de données : il est de la responsabilité du développeur de la prendre en charge.

La data annotation dévolue à cette tâche est `[MaxLength]`. Elle attend en paramètre la longueur souhaitée pour la colonne associée à la propriété.

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

L'équivalent impératif de cet attribut est la méthode `HasMaxLength` définie sur le type `PropertyBuilder`.

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

L'assignation d'une longueur maximale n'a pas toujours de sens, comme pour une valeur booléenne ou de type date. C'est pourquoi cette configuration n'est utilisable que sur des propriétés de type "tableau" comme `string` (qui est bien un tableau de `char`) ou `byte[]`. Elle est purement et simplement ignorée lorsqu'elle est utilisée sur une propriété avec le type de laquelle elle n'est pas compatible.

c. Index

Les index sont une composante essentielle des bases de données, relationnelles ou non. Lorsqu'ils sont utilisés correctement, ils peuvent être la clé de la réactivité d'une application par leur implication dans la performance des requêtes qui sont exécutées par la source de données. Entity Framework Core supporte la description d'index dans l'optique de les générer au niveau de la source de données.

Les conventions d'Entity Framework associent automatiquement un index à chaque clé étrangère, qu'elle soit simple ou composée de plusieurs propriétés.

Aucune data annotation n'existe pour la définition d'un index. Par conséquent, la seule manière de surcharger le comportement conventionnel d'Entity Framework est l'utilisation de la Fluent API. Le type `EntityTypeBuilder` implémente la méthode `HasIndex` qui permet de réaliser cette opération. L'expression qu'elle attend en paramètre désigne la propriété qui lui est associée.

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

Il est également possible de spécifier qu'un index est associé à plusieurs propriétés. Pour cela, l'expression de sélection doit renvoyer un objet de type anonyme contenant plusieurs propriétés.

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

Par défaut, les index générés par Entity Framework ne sont pas associés à des valeurs uniques. En d'autres termes, la table `Clients` pourrait contenir plusieurs enregistrements associés au même code postal. Ce comportement n'est pas toujours souhaitable : dans le cas d'une table contenant la liste des villes françaises et de leur code postal, un index sur le couple "code postal/ville" ne devrait pas autoriser de doublons. Pour résoudre ce type de problématique, le type `IndexBuilder` implémente une méthode `IsUnique` qui peut être chaînée à la suite de l'appel à `HasIndex`.

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

Pour laisser au développeur un maximum de contrôle sur les éléments générés au niveau de la source de données, le nom de la contrainte est également configurable à l'aide de la méthode `IndexBuilder.HasName`.

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

d. Clés alternatives

Une clé alternative est une propriété, ou un groupe de propriétés, qui peut être utilisé en tant qu'identificateur unique d'une entité. Dans le monde relationnel, une clé alternative peut être assimilée à une contrainte d'unicité placée sur les champs qui composent la clé. Leur singularité permet donc aux clés alternatives d'être utilisées comme clé d'entité principale dans l'établissement d'une relation entre types d'entités.

C'est d'ailleurs cette spécificité qui est exploitée pour la création conventionnelle de clés alternatives : toute propriété (ou groupe de propriétés) utilisée comme clé d'entité principale lors de la définition d'une relation se voit attribuer le statut de clé alternative de manière automatique.

Cette convention est implémentée au sein du type

`Microsoft.EntityFrameworkCore.Metadata.Internal.EntityTypeBuilder`.

L'exemple suivant montre la mise en place d'une relation entre les types d'entités `Client` et `Commande`. L'identifiant utilisé au niveau des commandes n'est pas, comme précédemment, la propriété `Client.ClientId`, mais `Client.Email`. Les adresses de courrier électronique étant effectivement uniques, l'ensemble est cohérent. L'association est effectuée par l'utilisation de la fonction `HasPrincipalKey` dont le rôle est de définir explicitement la clé d'entité principale utilisée dans une relation. La propriété `Client.Email` est alors enregistrée par le moteur d'Entity Framework Core comme étant une clé alternative.

```
public class AlternateKeyContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
        Database=AlternateKeyDb;Trusted_Connection=True;");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Client>()
            .HasMany(client => client.Commandes)
            .WithOne(commande => commande.Client)
            .HasForeignKey(commande => commande.ClientEmail)
            .HasPrincipalKey(client => client.Email);
    }

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

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

    public string Email { get; set; }

    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
```

```

    public string ville { get; set; }

    public List<Commande> Commandes { get; set; }
}

public class Commande
{
    public int CommandeId { get; set; }
    public decimal MontantTotalHT { get; set; }
    public decimal MontantTVA { get; set; }

    public string ClientEmail { get; set; }
    public Client Client { get; set; }
}

```

L'application impérative du statut de clé alternative à une propriété, ou à un groupe de propriétés, est tout simplement effectuée par un appel à la méthode `EntityTypeBuilder.HasAlternateKey`.

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        .HasAlternateKey(client => client.Email);
}

```

e. Valeurs générées

Entity Framework Core autorise la génération automatique de valeurs par la source de données pour les propriétés de types d'entités. L'implémentation de la logique associée est déléguée aux fournisseurs de données qui ont ainsi la possibilité d'exécuter des traitements qui leur sont propres, mais qui ont aussi la liberté d'ignorer cette fonctionnalité.

La mécanique de génération de valeurs supporte trois comportements différents, définis par l'énumération `DatabaseGeneratedOption` située dans l'espace de noms

`System.ComponentModel.DataAnnotations.Schema` :

- `None` : aucune génération n'est effectuée, il est de la responsabilité du développeur de fournir une valeur valide. C'est le comportement par défaut pour la majorité des propriétés.
- `Identity` : la valeur de la propriété est créée lors de l'insertion d'un enregistrement dans la source de données. C'est le comportement défini par les conventions d'Entity Framework Core pour les clés primaires de type `int`, `string`, `Guid` ou `byte[]`.
- `Computed` : la valeur de la propriété est créée à l'ajout et recrée à chaque répercussion des modifications de l'entité.

Les conventions qui sont associées à la génération de valeurs sont implémentées au niveau du type `KeyConvention`.

L'annotation qui permet d'associer une stratégie de génération à une propriété utilise l'attribut `DatabaseGeneratedAttribute`, auquel est fournie la valeur d'énumération qui identifie la stratégie souhaitée.

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

La Fluent API définit, quant à elle, trois méthodes pour indiquer le comportement attendu pour une propriété, chacune d'elles étant associée à une stratégie de génération.

Pour indiquer qu'aucune génération ne doit être effectuée, on utilise la méthode `PropertyBuilder.ValueGeneratedNever`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        .Property(client => client.Id)
        .ValueGeneratedNever();
}
```

Lorsque la valeur de la propriété doit être générée lors de l'insertion de l'enregistrement associé, on exécute la fonction `PropertyBuilder.ValueGeneratedOnAdd`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        .Property(client => client.Id)
        .ValueGeneratedOnAdd();
}
```

Enfin, la méthode `PropertyBuilder.ValueGeneratedOnAddOrUpdate` est utilisée lorsque la valeur doit être générée lors de l'insertion et à chaque mise à jour de l'enregistrement.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        .Property(client => client.Id)
        .ValueGeneratedOnAddOrUpdate();
}
```

f. Colonnes calculées

Entity Framework Core supporte la notion de colonnes calculées intégrée par les sources de données relationnelles. Cette fonctionnalité permet d'associer une expression SQL à une colonne de manière à obtenir le résultat de l'évaluation de cette expression pour chaque enregistrement.

Les colonnes calculées sont identifiées avec la Fluent API, par l'utilisation de la fonction `HasComputedColumnSql` à laquelle est passée l'expression SQL qui sera la base des évaluations.

```
public class Context : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
```

```

        Database=ComputedColumnDb;Trusted_Connection=True;");
    }

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

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

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

    public string Email { get; set; }

    public string Nom { get; set; }
    public string Prenom { get; set; }
}

```

Ici, la propriété `Email` est configurée de manière que sa valeur corresponde au schéma `<Nom>.<Prenom>@mycompany.fr`.

g. Valeurs par défaut

L'utilisation de valeurs par défaut dans les sources de données relationnelles est monnaie courante. Une colonne bénéficiant de ce mécanisme peut ainsi avoir une valeur valide pour chacun des enregistrements de la table. Ce mécanisme est supporté par Entity Framework Core pour deux types de valeurs différents : les valeurs constantes et les valeurs calculées à partir d'un fragment SQL.

L'assignation d'une valeur par défaut constante à une propriété est exécutée par un appel à la fonction `PropertyBuilder.HasDefaultValue`.

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Client>()
        .Property(client => client.Email)
        .HasDefaultValue("emailgenerique@mycompany.fr");
}

```

Lorsque la valeur par défaut doit être générée à partir d'une expression SQL, la méthode qui doit être appelée est `PropertyBuilder.HasDefaultValueSql`.

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

Scaffolding

Le scaffolding (échafaudage, en français) est une technique dont l'objectif est la génération de code. Plus exactement, il s'agit de générer une structure de base sur laquelle de nouveaux éléments peuvent se greffer, en s'appuyant sur une spécification. Dans le cas d'un projet ASP.NET MVC, cette spécification correspond à quelques paramètres indiquant s'il faut générer une structure vide, un modèle, des éléments d'accès aux données... Pour le cas qui nous intéresse, à savoir l'utilisation d'Entity Framework Core avec une approche Database First, le scaffolding permet la génération d'un modèle objet à partir d'une source de données existante et de nombreux paramètres dont l'objectif est la personnalisation du code généré.

Pour utiliser cette fonctionnalité lors de l'intégration d'Entity Framework Core à une application, il est nécessaire d'installer une librairie complémentaire associée au fournisseur de données choisi. Le nom de cette librairie est construit, par convention, en associant le nom complet du fournisseur de données et le suffixe ".Design". Dans le cas du fournisseur pour SQL Server, dont le nom complet est Microsoft.EntityFrameworkCore.SqlServer, elle a pour nom Microsoft.EntityFrameworkCore.SqlServer.Design.

Une librairie de design contient les instructions utilisées par le fournisseur de données pour générer le code source .NET à partir de la structure de la base de données. Les outils en ligne de commande évoqués à la section Deux approches pour deux cas d'utilisation - Database First tirent tous deux parti des assemblies de design pour effectuer les tâches qui leur sont confiées.

1. Console du gestionnaire de package NuGet

Les commandes PowerShell utilisables avec la console du gestionnaire de package NuGet sont localisées dans un package NuGet nommé Microsoft.EntityFrameworkCore.Tools. Au préalable, il est donc nécessaire de l'installer.

La commande qui nous intéresse dans le cadre du scaffolding s'appelle `Scaffold-DbContext`. L'installation de package devrait intégrer la commande à la console du gestionnaire de package NuGet.

Pour le vérifier, ouvrez cette console à l'aide du menu **Outils - Gestionnaire de package NuGet - Console du Gestionnaire de package** de Visual Studio, et exécutez `get-help NuGet`. La liste de commandes qui s'affiche devrait contenir, entre autres, `Scaffold-DbContext`.

Avant de générer la moindre ligne de code vient une première étape importante et souvent négligée, la phase de lecture et de compréhension de la documentation.

L'aide interactive proposée par PowerShell fournit la liste des arguments acceptés par la commande `Scaffold-DbContext`. Cette section détaille chacun de ces arguments, exemples à l'appui. La source de données utilisée ici est une base de données SQL Server de démonstration fournie par Microsoft : Northwind.

Cette base de données sera de nouveau évoquée au tout début du chapitre Des objets au SQL, avec la marche à suivre pour sa création.

La syntaxe générale d'utilisation de la commande est la suivante.

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

a. Arguments obligatoires : Connection et Provider

-Connection

Cet argument représente la chaîne de connexion complète qui doit être utilisée pour lire les informations de la source de données.

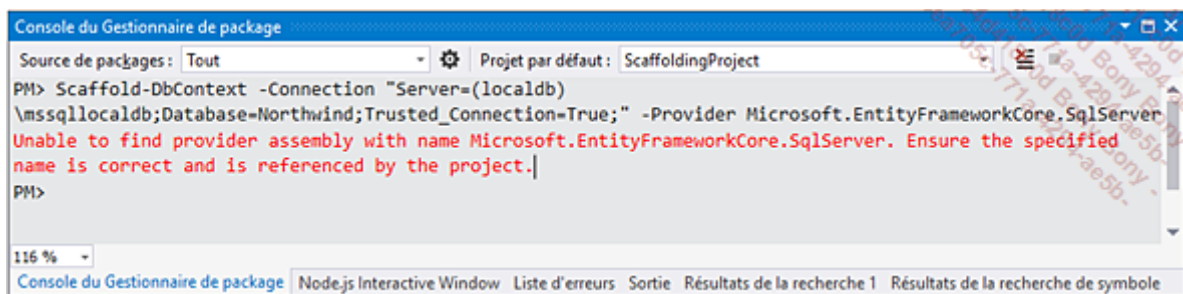
Le nom de l'argument peut être omis lorsque sa valeur est la première dans la chaîne d'arguments.

-Provider

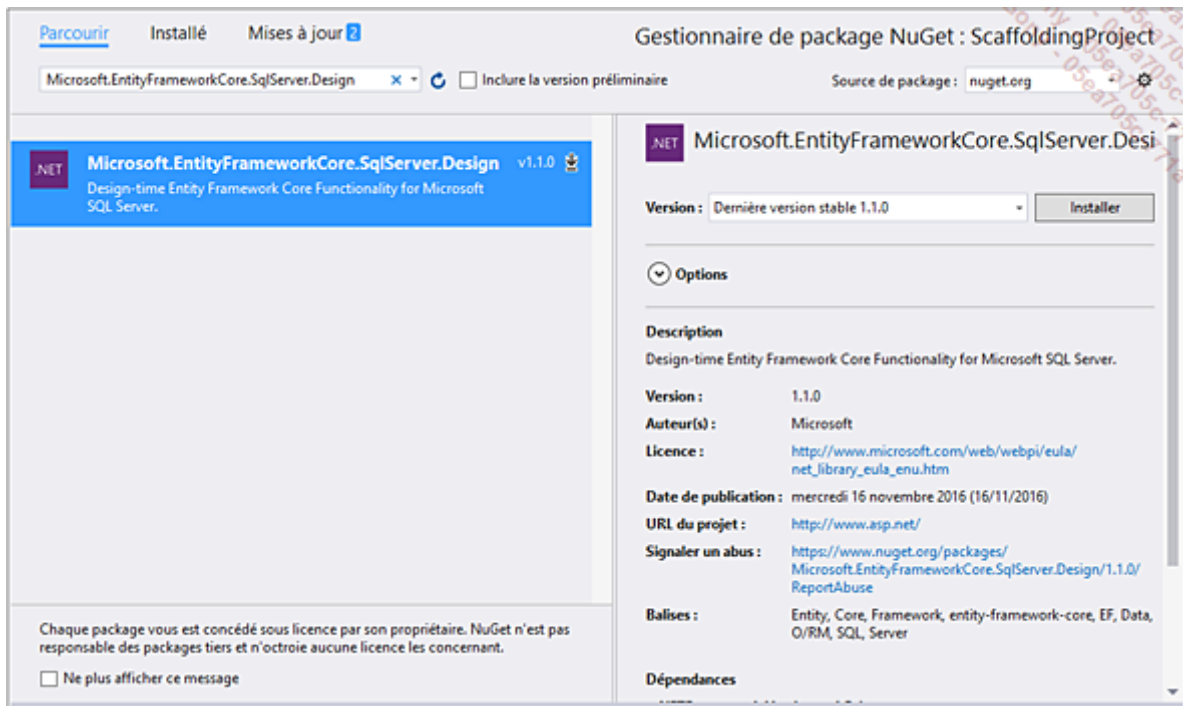
La valeur associée à cet argument indique le nom complet du fournisseur de données qui doit être utilisé pour accéder à la source de données.

```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
```

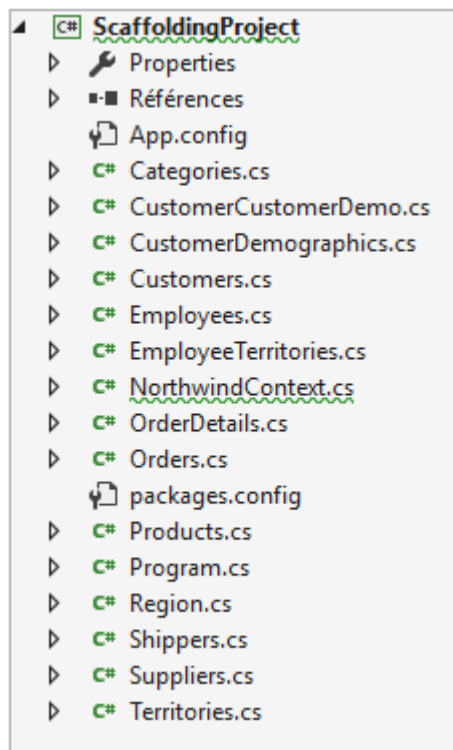
Si l'exécution de cette commande provoque l'affichage d'un message d'erreur indiquant qu'un assembly du provider Microsoft.EntityFrameworkCore.SqlServer n'est pas trouvé, rien d'anormal.



Les éléments nécessaires au scaffolding sont livrés dans un package NuGet distinct, dont le nom est conventionnellement le même que celui du fournisseur suivi du suffixe ".Design". Dans notre cas, il faut installer le package Microsoft.EntityFrameworkCore.SqlServer.Design.



Une fois cette dépendance mise en place, l'exécution de la commande de scaffolding déclenche le processus de génération de code. Elle a pour résultat l'ajout de plusieurs fichiers à la racine du projet sélectionné dans la console du gestionnaire de package. Chacun de ces fichiers de code correspond à une table de la source de données, à l'exception du fichier NorthwindContext.cs, qui contient le contexte de données.

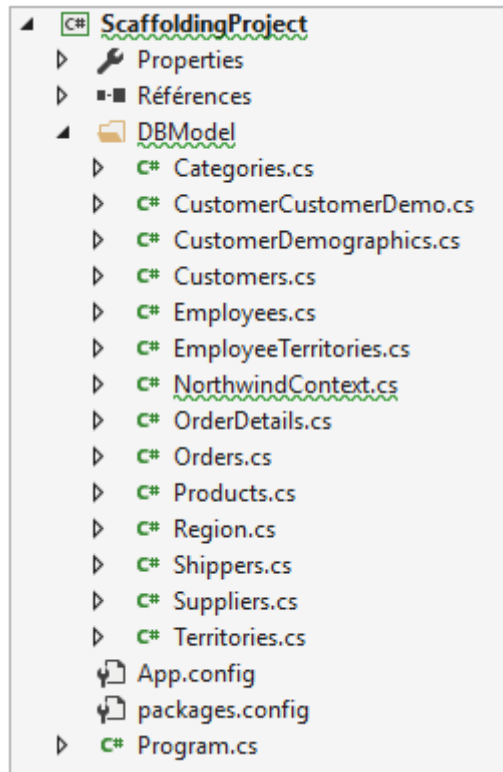


b. OutputDir

Cet argument permet de spécifier le répertoire dans lequel les fichiers générés doivent être placés.

```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
```

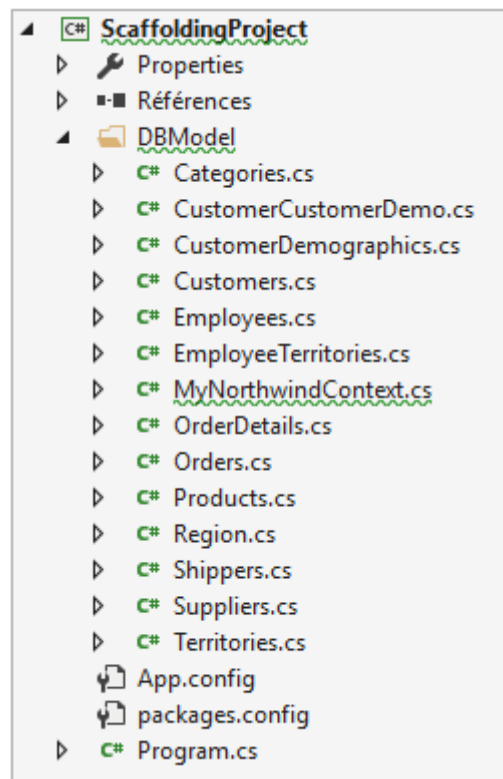
Ici, la commande redirige l'ensemble des fichiers dans le répertoire DBModel du projet sélectionné. Si ce répertoire n'existe pas, il est créé lors de l'opération.



c. Context

Le nom d'une base de données n'est pas forcément en adéquation avec les conventions de nommage utilisées par le code .NET. Il n'est pas rare de trouver une base de données dont le nom est entièrement en majuscules, par exemple. Les contextes de données générés ayant par défaut le nom de la source de données accolé au suffixe "Context", cela peut être gênant. Cette problématique est réglée par l'utilisation de l'argument `-Context`, qui permet de définir le nom du type de contexte généré.

```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
-Context MyNorthwindContext
```



d. Schémas

Lorsque les tables de la base de données n'appartiennent pas toutes au schéma de l'utilisateur connecté, cet argument permet de spécifier les noms des schémas pour lesquels doivent être générés des types d'entités. Les noms de schémas fournis doivent être séparés par des virgules : schema1,schema2,...

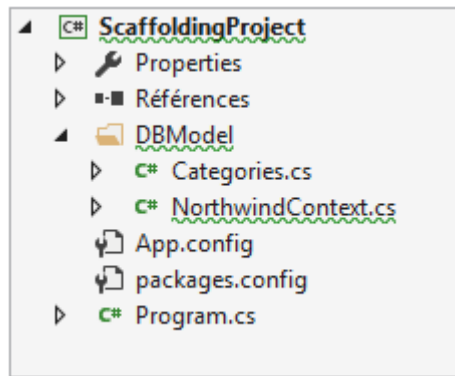
Le script T-SQL suivant permet de créer un schéma et de transférer la table `Categories` vers ce schéma :

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

ALTER SCHEMA monSchema TRANSFER dbo.Categories
```

```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
-Schemas monSchema
```

Cette commande ne génère que le type d'entité associé à la table `Categories`.



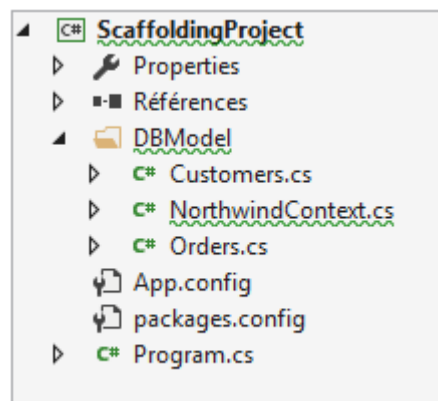
```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
-Schemas monSchema,dbo
```

Dans ce cas, des types d'entités sont créés pour l'ensemble des tables de la base de données.

e. Tables

Cet argument a un comportement comparable au précédent puisqu'il permet de filtrer la liste des tables pour lesquelles du code doit être généré. Comme pour les schémas, les noms des tables concernées sont séparés par des virgules.

```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
-Tables Customers,Orders
```



f. DataAnnotations

L'argument `-DataAnnotations` indique que le code généré devrait utiliser les data annotations tant que possible. Lorsqu'il n'est pas présent, la configuration des types d'entités générée n'utilise que la Fluent API.

Le code suivant montre l'aspect du type d'entité `Customers` et de la portion de configuration associée dans `NorthwindContext` lorsque l'argument `-DataAnnotations` n'est pas présent.

[Customers.cs](#)

```

public partial class Customers
{
    public Customers()
    {
        CustomerCustomerDemo = new HashSet<CustomerCustomerDemo>();
        Orders = new HashSet<Orders>();
    }

    public string CustomerId { get; set; }
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
    public string Fax { get; set; }

    public virtual ICollection<CustomerCustomerDemo>
CustomerCustomerDemo { get; set; }
    public virtual ICollection<Orders> Orders { get; set; }
}

```

NorthwindContext.cs

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // ...
    modelBuilder.Entity<Customers>(entity =>
    {
        entity.HasKey(e => e.CustomerId)
            .HasName("PK_Customers");

        entity.HasIndex(e => e.City)
            .HasName("City");

        entity.HasIndex(e => e.CompanyName)
            .HasName("CompanyName");

        entity.HasIndex(e => e.PostalCode)
            .HasName("PostalCode");

        entity.HasIndex(e => e.Region)
            .HasName("Region");

        entity.Property(e => e.CustomerId)
            .HasColumnName("CustomerID")
            .HasColumnType("nchar(5)");

        entity.Property(e => e.Address).HasMaxLength(60);

        entity.Property(e => e.City).HasMaxLength(15);
    });
}

```

```

        entity.Property(e => e.CompanyName)
            .IsRequired()
            .HasMaxLength(40);

        entity.Property(e => e.ContactName).HasMaxLength(30);

        entity.Property(e => e.ContactTitle).HasMaxLength(30);

        entity.Property(e => e.Country).HasMaxLength(15);

        entity.Property(e => e.Fax).HasMaxLength(24);

        entity.Property(e => e.Phone).HasMaxLength(24);

        entity.Property(e => e.PostalCode).HasMaxLength(10);

        entity.Property(e => e.Region).HasMaxLength(15);
    });

    // ...
}

```

Avec l'utilisation de l'argument `DataAnnotations`, le résultat est bien différent.

```

Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
-DataAnnotations

```

Customers.cs

```

public partial class Customers
{
    public Customers()
    {
        CustomerCustomerDemo = new HashSet<CustomerCustomerDemo>();
        Orders = new HashSet<Orders>();
    }

    [Column("CustomerID", TypeName = "nchar(5)")]
    [Key]
    public string CustomerId { get; set; }
    [Required]
    [MaxLength(40)]
    public string CompanyName { get; set; }
    [MaxLength(30)]
    public string ContactName { get; set; }
    [MaxLength(30)]
    public string ContactTitle { get; set; }
    [MaxLength(60)]
    public string Address { get; set; }
    [MaxLength(15)]

```

```

public string City { get; set; }
[MaxLength(15)]
public string Region { get; set; }
[MaxLength(10)]
public string PostalCode { get; set; }
[MaxLength(15)]
public string Country { get; set; }
[MaxLength(24)]
public string Phone { get; set; }
[MaxLength(24)]
public string Fax { get; set; }

[InverseProperty("Customer")]
public virtual ICollection<CustomerCustomerDemo> CustomerCustomerDemo { get;
set; }
[InverseProperty("Customer")]
public virtual ICollection<Orders> Orders { get; set; }
}

```

NorthwindContext.cs

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // ...

    modelBuilder.Entity<Customers>(entity =>
    {
        entity.HasIndex(e => e.City)
            .HasName("City");

        entity.HasIndex(e => e.CompanyName)
            .HasName("CompanyName");

        entity.HasIndex(e => e.PostalCode)
            .HasName("PostalCode");

        entity.HasIndex(e => e.Region)
            .HasName("Region");
    });

    // ...
}

```

g. Force

La présence de cet argument permet la régénération d'un fichier de code. Sans lui, le processus de génération ne sera pas exécuté et renvoie une erreur si un des fichiers qui doit être écrit existe déjà.


```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
-Force
```

h. Project

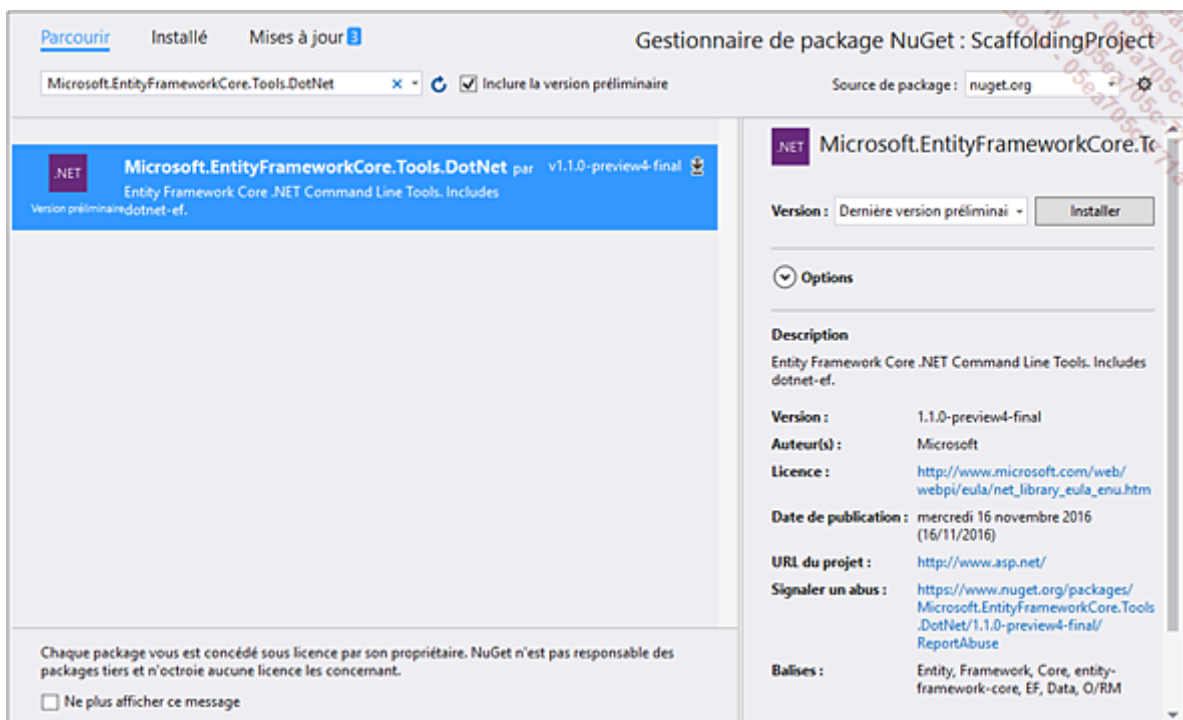
Cet argument définit explicitement le nom du projet à l'intérieur de la solution courante dans lequel doit être effectuée la génération. Si cet argument est omis, c'est le projet sélectionné dans la console du gestionnaire de package NuGet qui est le réceptacle du code généré.

Les deux bibliothèques associées au fournisseur de données choisi doivent être installées dans le projet.

```
Scaffold-DbContext
-Connection "Server=
(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir ./DBModel
-Project ScaffoldingProject
```

2. CLI .NET

Les commandes utilisables avec l'interface en ligne de commande .NET sont localisées dans le package NuGet nommé Microsoft.EntityFrameworkCore.Tools.DotNet. Au préalable, il est donc nécessaire de l'installer.



La commande qui nous intéresse ici est implémentée par l'outil `ef` intégré au projet par l'installation de ce package. Une fois cette opération effectuée, il devrait être utilisable via une invite de commandes.

Pour le vérifier, ouvrez une invite de commandes, placez-vous dans le répertoire de votre projet, puis exécutez la commande `dotnet ef`. Un magnifique animal légendaire et quelques informations contextuelles devraient être affichés.

La commande `dotnet ef dbcontext scaffold --help` permet d'obtenir l'aide associée à la commande de génération de code. Cette section détaille les différents arguments acceptés par l'outil, avec des exemples s'appuyant sur la base de données Northwind de Microsoft.

La syntaxe générale d'utilisation de la commande est décrite ci-après. Lorsque plusieurs manières de spécifier un argument sont disponibles, elles sont séparées par un caractère pipe (|).

```
dotnet ef dbcontext scaffold
    <connection string>
        <provider>
        -o|--output-dir <path>
        -c|--context <name>
        -a|--data-annotations
        -f|--force
        -t|--table <schema.table>
        --schema <schema>
        --json
        --verbose
        --root-namespace <namespace>
```

a. Arguments obligatoires

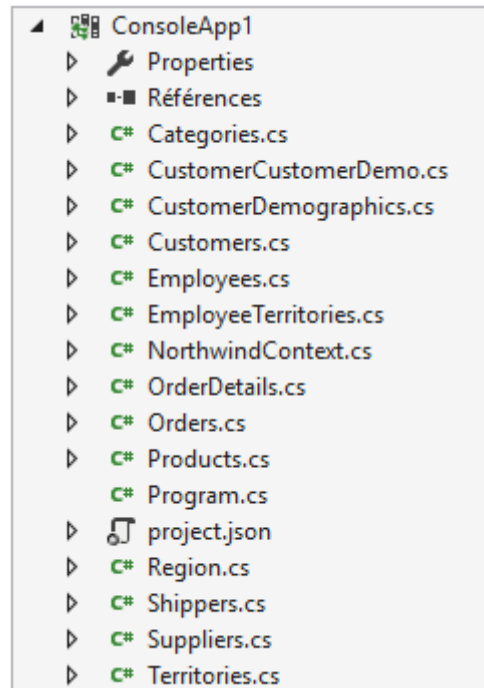
Les deux premiers arguments passés lors de l'exécution de la commande sont obligatoires et leurs valeurs sont utilisées en fonction de leur position :

- Le premier argument passé doit être la chaîne de connexion complète à la source de données.

- Le second correspond, quant à lui, au nom complet du fournisseur de données qui doit être utilisé pour la lecture des informations de la source de données ainsi que pour la génération du modèle objet.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=
Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
```

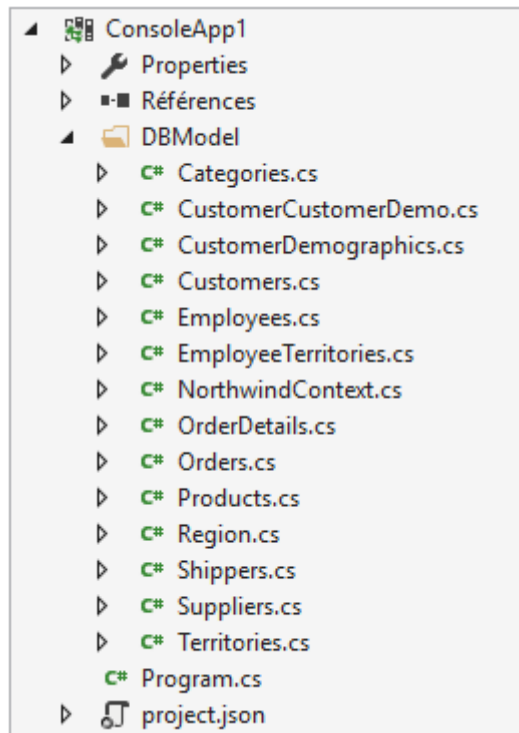
Différents fichiers de code sont ajoutés à la racine du projet en retour. Ils correspondent tous à une table de la source de données, à l'exception d'un, qui est dédié au type du contexte de données.



b. output-dir

L'organisation d'un projet est un élément essentiel pour sa maintenabilité. Générer le modèle objet directement à la racine du projet est par conséquent un véritable problème qu'il convient de régler. L'argument `--output-dir` offre la solution en permettant d'indiquer le répertoire de destination pour les fichiers générés.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
```



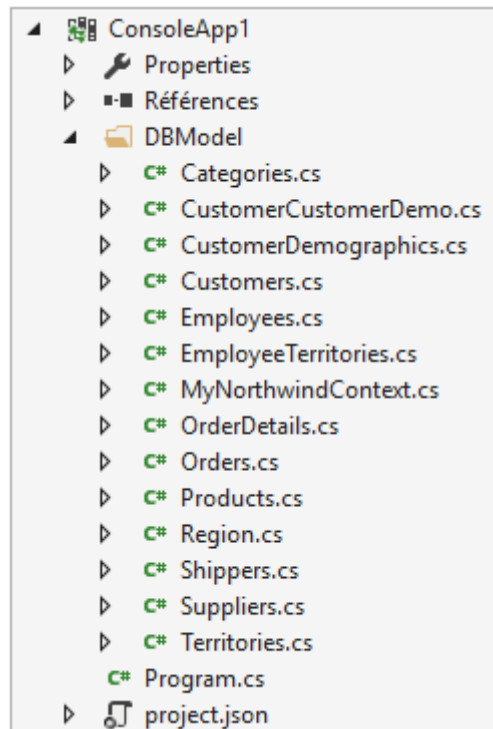
Cet argument peut également être utilisé à l'aide de l'alias `-o` :

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
-o ./DBModel
```

c. context

Pour définir le nom du contexte de données généré par la commande, il convient d'utiliser l'argument `--context` auquel la valeur souhaitée est passée.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
--context MyNorthwindContext
```



Cet argument peut également être utilisé à l'aide de l'alias `-c` :

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
-c MyNorthwindContext
```

d. schema

Cet argument a le même effet que l'argument `Schemas` de la commande PowerShell `scaffold-dbcontext`. Lorsqu'il est renseigné, il permet de générer les types d'entités associés à un certain schéma uniquement. Lorsque plusieurs schémas doivent être utilisés, il faut utiliser l'argument autant de fois que nécessaire.

Le script T-SQL suivant permet de créer un schéma et de transférer la table `Categories` vers ce 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
```

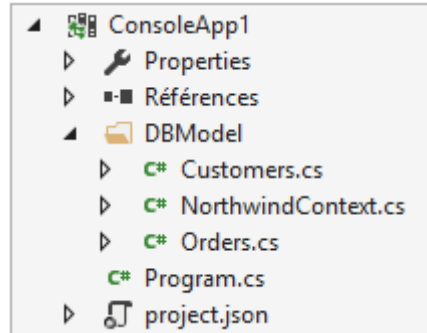
Pour générer les types d'entités associés aux schémas `dbo` et `monSchema`, on peut écrire :

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
--schema dbo --schema monSchema
```

e. table

Comme `--schema`, l'argument `--table` permet de filtrer la liste des tables pour lesquelles la commande doit générer un type d'entité. Il permet en effet de nommer explicitement une table qui doit être incluse dans le traitement, et doit être utilisé pour chaque type d'entité souhaité. À défaut, s'il n'est donc pas présent, toutes les tables sont incluses dans le processus.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
--table Customers --table Orders
```



Ce comportement peut également être défini à l'aide de l'alias `-t` :

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
-t Customers -t Orders
```

f. data-annotations

L'opération de génération produit par défaut des types d'entités entièrement configurés à l'aide de la Fluent API. Ce switch provoque l'utilisation de data annotations pour tous les éléments générés, là où elles sont disponibles.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
--data-annotations
```

Cet argument est également utilisable au travers de son alias, `-a`.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
-a
```

g. force

Par défaut, la génération est bloquée lorsqu'un ou plusieurs fichiers de destination existent déjà. L'argument `--force` permet de lever ce blocage.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
--force
```

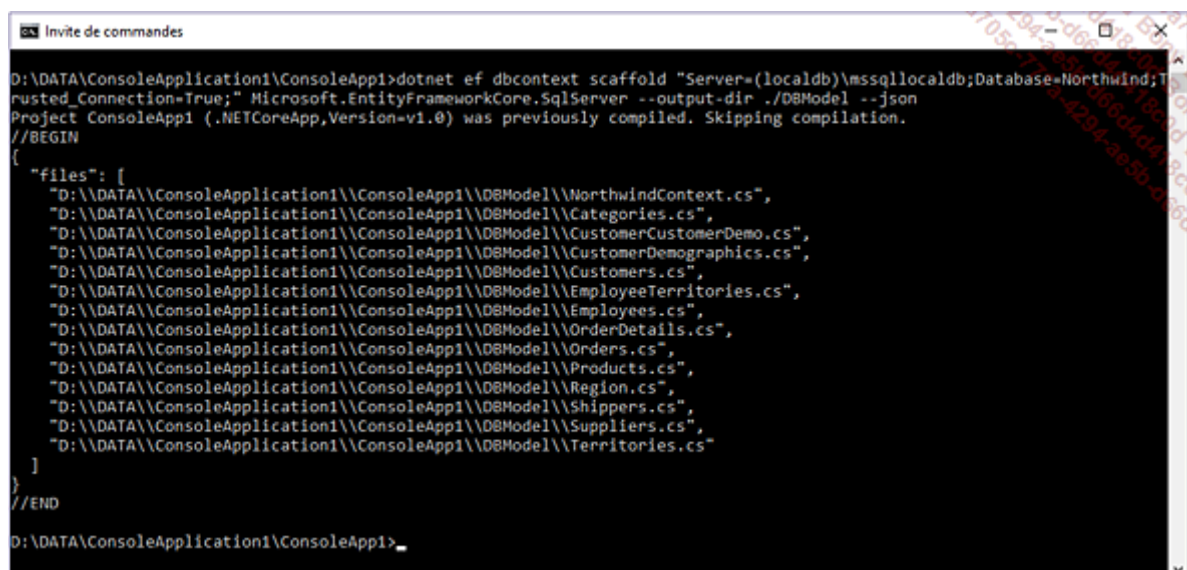
Ce comportement peut également être activé par l'utilisation de l'alias `-f` de cet argument.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
-f
```

h. json

L'argument `--json` offre la possibilité d'obtenir un retour concernant la liste des fichiers manipulés par la commande. Ce retour est présenté sous la forme d'un objet JSON.

```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
--json
```



```
Invite de commandes
D:\DATA\ConsoleApplication1\ConsoleApp1>dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer --output-dir ./DBModel --json
Project ConsoleApp1 (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.
//BEGIN
{
  "files": [
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\NorthwindContext.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Categories.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\CustomerCustomerDemo.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\CustomerDemographics.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Customers.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\EmployeeTerritories.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Employees.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\OrderDetails.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Orders.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Products.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Region.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Shippers.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Suppliers.cs",
    "D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel\Territories.cs"
  ]
}
//END
D:\DATA\ConsoleApplication1\ConsoleApp1>
```

i. verbose

Ce dernier switch permet d'obtenir plus d'informations sur le processus de génération en cours, et ce particulièrement lorsqu'une erreur se produit.


```
dotnet ef dbcontext scaffold
"Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
--output-dir ./DBModel
--verbose
```

```
Selection Invite de commandes
D:\DATA\ConsoleApplication1\ConsoleApp1>dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer --output-dir ./DBModel --verbose
Project ConsoleApp1 (.NETCoreApp,Version=v1.0) will be compiled because Input items removed from last build
Compiling ConsoleApp1 for .NETCoreApp,Version=v1.0
Compilation succeeded.
    0 Warning(s)
    0 Error(s)
Time elapsed 00:00:02.5291666

Setting app base path D:\DATA\ConsoleApplication1\ConsoleApp1\bin\Debug\netcoreapp1.0
D:\DATA\ConsoleApplication1\ConsoleApp1>
```

```
Invite de commandes
D:\DATA\ConsoleApplication1\ConsoleApp1>dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer --output-dir ./DBModel --verbose
Project ConsoleApp1 (.NETCoreApp,Version=v1.0) will be compiled because Input items added from last build
Compiling ConsoleApp1 for .NETCoreApp,Version=v1.0
D:\DATA\ConsoleApplication1\ConsoleApp1\NorthwindContext.cs(25,22): warning CS1030: #warning: 'To protect potentially sensitive information in your connection string, you should move it out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection strings.'
Compilation succeeded.
    1 Warning(s)
    0 Error(s)
Time elapsed 00:00:03.5224261

Setting app base path D:\DATA\ConsoleApplication1\ConsoleApp1\bin\Debug\netcoreapp1.0
System.InvalidOperationException: The following file(s) already exist in directory D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel: NorthwindContext.cs;CustomerCustomerDemo.cs;CustomerDemographics.cs;Customers.cs;Employees.cs;EmployeeTerritories.cs;OrderDetails.cs;Orders.cs;Products.cs;Region.cs;Shippers.cs;Suppliers.cs;Territories.cs. Use the Force flag to overwrite these files.
   at Microsoft.EntityFrameworkCore.Scaffolding.Internal.ReverseEngineeringGenerator.CheckOutputFiles(String outputPath, String dbContextClassName, IModel metadataModel, Boolean overwriteFiles)
   at Microsoft.EntityFrameworkCore.Scaffolding.Internal.ReverseEngineeringGenerator.GenerateAsync(ReverseEngineeringConfiguration configuration, CancellationToken cancellationToken)
   at Microsoft.EntityFrameworkCore.Design.Internal.DatabaseOperations ScaffoldContextAsync(String provider, String connectionString, String outputDir, String dbContextClassName, IEnumerable<1 schemas, IEnumerable<1 tables, Boolean useDataAnnotations, Boolean overwriteFiles, CancellationToken cancellationToken)
   at Microsoft.EntityFrameworkCore.Design.OperationExecutor.<ScaffoldContextImpl>d__22.MoveNext()
   at System.Collections.Generic.EnumerableHelpers.ToArray[T](IEnumerable<1 source, Int32& length)
   at System.Collections.Generic.EnumerableHelpers.ToArray[T](IEnumerable<1 source)
   at Microsoft.EntityFrameworkCore.Design.OperationExecutor.OperationBase.<>c__DisplayClass4_0`1.<Execute>b__0()
   at Microsoft.EntityFrameworkCore.Design.OperationExecutor.OperationBase.Execute(Action action)
The following file(s) already exist in directory D:\DATA\ConsoleApplication1\ConsoleApp1\DBModel: NorthwindContext.cs;Categories.cs;CustomerCustomerDemo.cs;CustomerDemographics.cs;Customers.cs;Employees.cs;EmployeeTerritories.cs;OrderDetails.cs;Orders.cs;Products.cs;Region.cs;Shippers.cs;Suppliers.cs;Territories.cs. Use the Force flag to overwrite these files.
D:\DATA\ConsoleApplication1\ConsoleApp1>
```

Migrations

Les possibilités d'Entity Framework Core incluent, par l'approche Model First, la gestion de la structure des sources de données. Cette capacité s'exprime au travers du concept de migration.

Le terme migration représente le processus de transformation d'une structure de base de données de manière qu'elle corresponde à un modèle donné. Il prend la forme de code .NET décrivant l'ensemble des opérations nécessaires à la mutation de la source de données. Chaque évolution d'un modèle objet peut ainsi donner lieu à la création automatisée d'une migration.

Comme le scaffolding, les migrations sont gérées au travers des deux outils en ligne de commande que sont la console du gestionnaire de package NuGet et la CLI .NET Core. Ces outils sont complétés par les deux packages NuGet Microsoft.EntityFrameworkCore.Tools et Microsoft.EntityFrameworkCore.Tools.DotNet, qui amènent avec eux les outils spécifiques à Entity Framework Core. Les bibliothèques de design associées à chaque fournisseur de données font le pont

entre un projet qui contient un modèle objet et les outils en ligne de commande : ils fournissent les éléments permettant l'interprétation et le traitement du modèle objet.

Dans le cas plus particulier des migrations, les outils permettent d'effectuer trois opérations essentielles : la création d'une migration à partir de l'état courant d'un contexte de données, la suppression d'une migration, et enfin l'application d'une migration à la source de données. Un quatrième traitement très intéressant est également implémenté : la génération d'un script différentiel qui décrit les instructions à exécuter pour aller d'un état associé à une migration à celui résultant d'une seconde migration. Nous manipulerons un modèle client/commande simpliste pour mettre en évidence ces différents comportements.

1. Création d'une migration

La création d'une migration suppose l'existence d'un contexte de données configuré de manière à pouvoir utiliser le fournisseur de données qui lui est associé. Afin que cette migration ait une utilité, il est également nécessaire qu'un ou plusieurs types d'entités existent et soient associés au contexte.

Créez le contexte de données et le type `Client`.

```
public class MigrationsContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
Database=MigrationsDb;Trusted_Connection=True;");
    }

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

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

    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string ville { get; set; }
}
```

La commande PowerShell associée à la création d'une migration est `Add-Migration`. La syntaxe générale d'utilisation de cette commande est la suivante :

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

Lors de son exécution, elle attend un paramètre obligatoire qui est le nom de la migration créée. Les noms des migrations sont en effet les repères qui permettent de les manipuler. Il est par conséquent très important d'utiliser un nom simple et explicite pour ne pas se perdre.

L'argument `-OutputDir` permet de spécifier le répertoire du projet dans lequel la migration doit être créée. Par défaut, un répertoire nommé Migrations, placé à la racine du projet, sert de réceptacle au code généré.

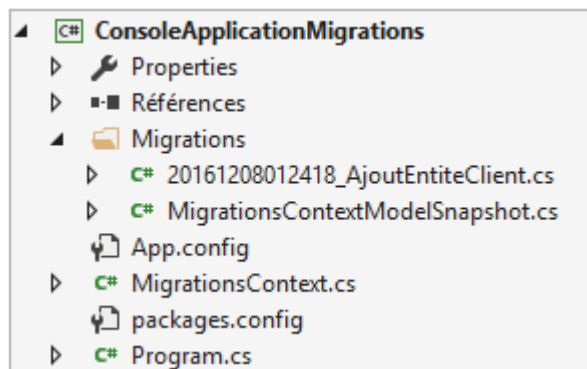
`-Context`, quant à lui, trouve son utilité dans les cas d'utilisation impliquant plusieurs contextes de données. Il indique alors le nom du contexte de données présent dans le projet pour lequel la migration doit être créée.

Créez une migration nommée `AjoutEntiteClient`.

```
Add-Migration AjoutEntiteClient
```

Attention, l'outil supporte mal les caractères accentués !

L'exécution de cette action crée deux fichiers : le nom du premier contient l'horodatage de la création de la migration, suivi du nom de la migration (20161208012418_AjoutEntiteClient.cs, par exemple). Le second s'appelle MigrationsContextModelSnapshot.cs.



Le premier de ces fichiers contient l'ensemble des instructions permettant de modifier la base de données dans les cas de montée en version comme dans les cas de retour arrière. Ces instructions sont implémentées respectivement dans le corps des méthodes `Up` et `Down`.

```
public partial class AjoutEntiteClient : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Clients",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
                        SqlServerValueGenerationStrategy.IdentityColumn),
                Adresse1 = table.Column<string>(nullable: true),
                Adresse2 = table.Column<string>(nullable: true),
                Adresse3 = table.Column<string>(nullable: true),
                CodePostal = table.Column<string>(nullable: true),
                Nom = table.Column<string>(nullable: true),
                Prenom = table.Column<string>(nullable: true),
                ville = table.Column<string>(nullable: true)
            },
        },
    }
}
```

```

        constraints: table =>
        {
            table.PrimaryKey("PK_Clients", x => x.Id);
        });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Clients");
    }
}

```

Le fichier MigrationsContextModelSnapshot.cs contient, quant à lui, une description du modèle objet associé à la dernière migration créée.

```

[DbContext(typeof(MigrationsContext))]
partial class MigrationsContextModelSnapshot : ModelSnapshot
{
    protected override void BuildModel(ModelBuilder modelBuilder)
    {
        modelBuilder
            .HasAnnotation("ProductVersion", "1.1.0-rtm-22752")
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
                SqlServerValueGenerationStrategy.IdentityColumn);

        modelBuilder.Entity("ConsoleApplicationMigrations.Client", b =>
        {
            b.Property<int>("Id")
                .ValueGeneratedOnAdd();

            b.Property<string>("Adresse1");

            b.Property<string>("Adresse2");

            b.Property<string>("Adresse3");

            b.Property<string>("CodePostal");

            b.Property<string>("Nom");

            b.Property<string>("Prenom");

            b.Property<string>("Ville");

            b.HasKey("Id");

            b.ToTable("Clients");
        });
    }
}

```

Pour effectuer la même opération à partir de la CLI .NET, placez-vous dans le répertoire du projet et exécutez la commande.

```
dotnet ef migration add AjoutEntiteClient
```

Modifiez maintenant le modèle pour intégrer les commandes.

```
public class MigrationsContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
        Database=MigrationsDb;Trusted_Connection=True;");
    }

    public DbSet<Client> Clients { get; set; }
    public DbSet<Commande> Commandes { get; set; }
}
```

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

    public string Nom { get; set; }
    public string Prenom { get; set; }

    public string Adresse1 { get; set; }
    public string Adresse2 { get; set; }
    public string Adresse3 { get; set; }
    public string CodePostal { get; set; }
    public string ville { get; set; }

    public List<Commande> Commandes { get; set; }
}
```

```
public class Commande
{
    public int CommandeId { get; set; }
    public decimal MontantTotalHT { get; set; }
    public decimal MontantTVA { get; set; }

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

Ajoutez une nouvelle migration nommée `AjoutEntiteCommande`.

```
Add-Migration AjoutEntiteCommande
```

Le nouveau fichier créé contient les instructions nécessaires au passage de l'état décrit par la migration `AjoutEntiteClient` à l'état actuel du modèle.

```

public partial class AjoutEntiteCommande : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Commandes",
            columns: table => new
            {
                CommandeId = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
                        SqlServerValueGenerationStrategy.IdentityColumn),
                ClientId = table.Column<int>(nullable: true),
                MontantTVA = table.Column<decimal>(nullable: false),
                MontantTotalHT = table.Column<decimal>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Commandes", x => x.CommandeId);
                table.ForeignKey(
                    name: "FK_Commandes_Clients_ClientId",
                    column: x => x.ClientId,
                    principalTable: "Clients",
                    principalColumn: "Id",
                    onDelete: ReferentialAction.Restrict);
            });

        migrationBuilder.CreateIndex(
            name: "IX_Commandes_ClientId",
            table: "Commandes",
            column: "ClientId");
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Commandes");
    }
}

```

2. Mise à jour de la source de données

L'application d'une migration à la source de données est exécutée à l'aide de la commande PowerShell `Update-Database`.

```

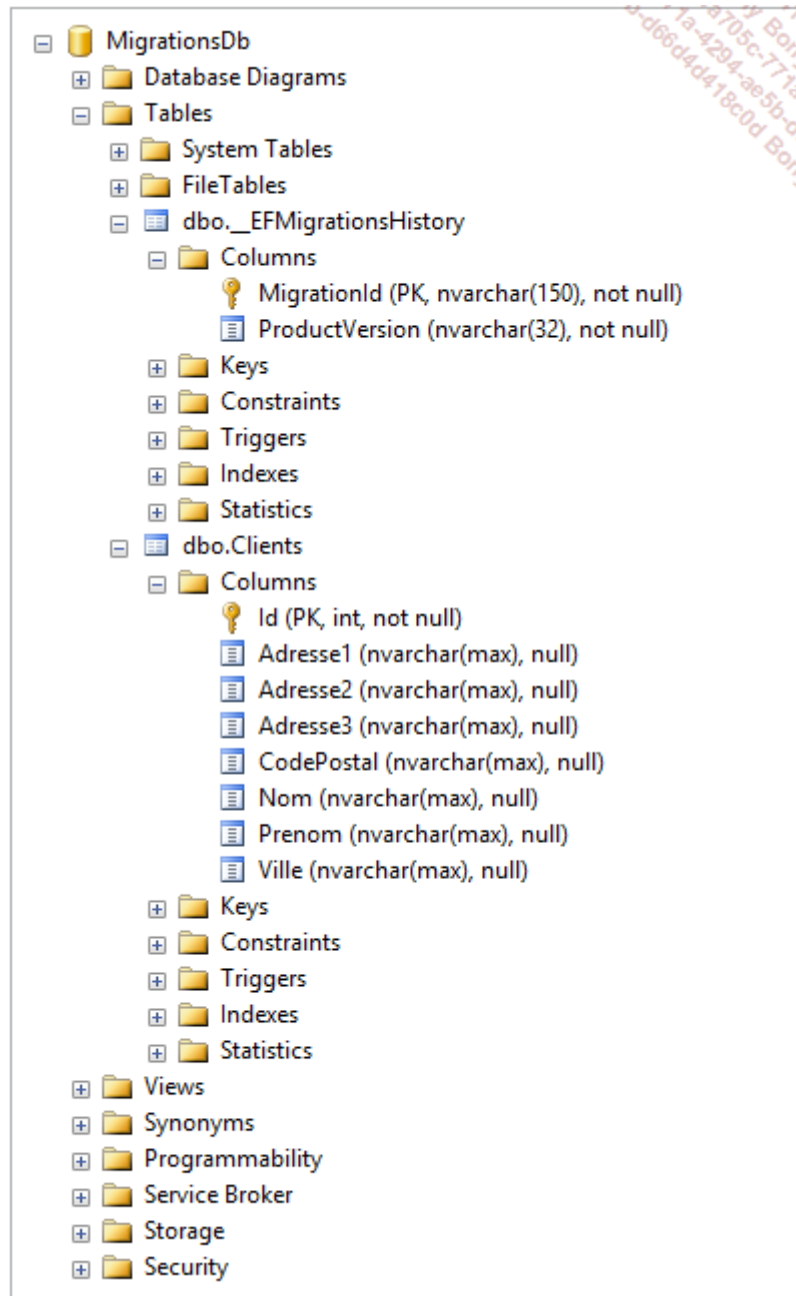
Update-Database [[-Migration] <String>]
                [-Context <String>]

```

Exécutée sans arguments, cette commande répercute sur la source de données l'ensemble des migrations qui n'ont pas encore été déployées. Si vous exécutez cette commande maintenant, la base de données résultante contient les tables `Clients` et `Commandes`.

`Update-Database` permet au développeur de placer la base de données à un point précis de son évolution en acceptant un argument qui correspond au nom d'une migration qui a été créée. Cette migration, par rapport à l'état de la base, peut se situer aussi bien en amont qu'en aval. Si la base est dans l'état `AjoutEntiteCommande`, l'exécution de la commande suivante la remet dans l'état `AjoutEntiteClient`. Elle peut ensuite être de nouveau migrée vers `AjoutEntiteCommande`, ou toute autre migration disponible.

```
Update-Database AjoutEntiteClient
```



On remarque l'existence de la table `__EFMigrationsHistory` qui est générée par l'outil de migrations pour enregistrer des informations concernant les migrations déjà appliquées sur la source de données.

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

```
dotnet ef database update AjoutEntiteClient
```

L'exécution de l'ensemble des migrations nécessite de passer la valeur 0 à la commande, car l'argument associé au nom de la migration est ici obligatoire.

L'argument `Context` de la commande PowerShell définit, quant à lui, le contexte de données pour lequel la migration doit être appliquée, exactement de la même manière qu'avec la commande `Add-Migration`. Il trouve son utilité dans le cas où le projet contient plusieurs contextes de données.

Dans le cas d'une exécution avec la CLI .NET, cet argument s'appelle `-context`.

3. Suppression de migration

La suppression de migration est toujours effectuée en suivant le principe LIFO (*Last In, First Out*). La commande PowerShell `Remove-Migration` ne permet en effet que de supprimer la dernière migration créée. Par analogie, chaque migration peut être associée à une assiette que l'on élimine du haut de la pile. La commande est par conséquent généralement utilisée sans argument.

`Remove-Migration`

Attention : cette commande ne répercute pas la suppression au niveau de la source de données.

Lorsque plusieurs contextes de données existent au sein du projet, il est nécessaire d'indiquer le nom de celui pour lequel la suppression doit être effectuée. L'argument `-Context` est, encore une fois, dédié à cet objectif.

4. Génération de scripts

L'application directe de migrations n'est pas toujours une solution acceptable, notamment en raison de certaines politiques d'entreprises ou de contraintes de déploiement. L'outil en charge des migrations est en mesure de fournir une alternative à ce mode de fonctionnement par la génération des scripts qui sont associés à la répercussion des modifications. Ces scripts ont la particularité de pouvoir être aussi bien absolus que différentiels : ils sont générés en fonction d'un état initial et d'un état final.

L'opération de génération est exécutée au travers de la commande PowerShell `Script-Migration`.

```
script-migration [-From <String> -To <String>]
                [-Idempotent]
                [-Context <String>]
```

Les arguments `From` et `To` permettent de définir les migrations source et destination utilisées. Lorsque ces arguments sont définis, l'outil génère le script différentiel. Lorsque ces valeurs ne sont pas passées, le moteur génère un script de création complet pour l'état associé à la dernière migration.

Le switch `-Idempotent` permet de générer un script qui peut être exécuté sur une base de données qui est dans un état correspondant à n'importe quelle migration. Ce script est alors à la fois différentiel et absolu.

Ci-après le script absolu associé à la migration `AjoutEntiteCommandes`.

```
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 [Clients] (
    [Id] int NOT NULL IDENTITY,
    [Adresse1] nvarchar(max),
    [Adresse2] nvarchar(max),
    [Adresse3] nvarchar(max),
    [CodePostal] nvarchar(max),
    [Nom] nvarchar(max),
    [Prenom] nvarchar(max),
    [Ville] nvarchar(max),
    CONSTRAINT [PK_Clients] PRIMARY KEY ([Id])
);

GO

INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20161208014929_AjoutEntiteClient', N'1.1.0-rtm-22752');

GO

CREATE TABLE [Commandes] (
    [CommandeId] int NOT NULL IDENTITY,
    [ClientId] int,
    [MontantTVA] decimal(18, 2) NOT NULL,
    [MontantTotalHT] decimal(18, 2) NOT NULL,
    CONSTRAINT [PK_Commandes] PRIMARY KEY ([CommandeId]),
    CONSTRAINT [FK_Commandes_Clients_ClientId] FOREIGN KEY ([ClientId])
REFERENCES [Clients] ([Id]) ON DELETE NO ACTION
);

GO

CREATE INDEX [IX_Commandes_ClientId] ON [Commandes] ([ClientId]);

GO

INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
VALUES (N'20161208015333_AjoutEntiteCommande', N'1.1.0-rtm-22752');

GO

```


Des objets au SQL

Modèle d'étude

La quasi-totalité des exemples qui sont décrits dans ce chapitre sont fondés sur la base de données de démonstration Northwind de Microsoft. Il est donc plus que probable que vous ayez besoin d'obtenir une copie locale de cette base si vous souhaitez tester les portions de code que vous découvrirez tout au long de ce chapitre.

Le code source relatif à cet ouvrage, disponible en téléchargement depuis la page Informations générales, contient des scripts SQL qui vous permettront de remplir de véritables sources de données. Ces scripts sont adaptés à différents fournisseurs de données, en fonction de votre préférence :

- SQL Server (testé avec SQL Server 2014 Express LocalDB) ;
- SQLite (testé avec SQLite 3.11.0) ;
- SQL Server Compact Edition 4.0.

Ils sont localisés dans le répertoire Chapitre 03/Scripts et portent le nom de la source de données à laquelle ils sont associés.

Le contexte de données associé à cette base de données, dans son format SQL Server, est décrit ci-après, ainsi que les types d'entités qui sont nécessaires à son bon fonctionnement.

```
public partial class NorthwindContext : DbContext
{
    public virtual DbSet<Categories> Categories { get; set; }
    public virtual DbSet<CustomerCustomerDemo> CustomerCustomerDemo { get; set; }
    public virtual DbSet<CustomerDemographics> CustomerDemographics { get; set; }
    public virtual DbSet<Customers> Customers { get; set; }
    public virtual DbSet<EmployeeTerritories> EmployeeTerritories { get; set; }
    public virtual DbSet<Employees> Employees { get; set; }
    public virtual DbSet<OrderDetails> OrderDetails { get; set; }
    public virtual DbSet<Orders> Orders { get; set; }
    public virtual DbSet<Products> Products { get; set; }
    public virtual DbSet<Region> Region { get; set; }
    public virtual DbSet<Shippers> Shippers { get; set; }
    public virtual DbSet<Suppliers> Suppliers { get; set; }
    public virtual DbSet<Territories> Territories { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        //La chaîne de connexion utilisée ici correspond à une base de données
        SQL Server 2014 Express LocalDB.
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
        Database=Northwind;Trusted_Connection=True;");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Categories>(entity =>
        {
            entity.HasIndex(e => e.CategoryName)
```

```

        .HasName("CategoryName");
    });

modelBuilder.Entity<CustomerCustomerDemo>(entity =>
{
    entity.HasKey(e => new { e.CustomerId, e.CustomerTypeId })
        .HasName("PK_CustomerCustomerDemo");
});

modelBuilder.Entity<Customers>(entity =>
{
    entity.HasIndex(e => e.City)
        .HasName("City");

    entity.HasIndex(e => e.CompanyName)
        .HasName("CompanyName");

    entity.HasIndex(e => e.PostalCode)
        .HasName("PostalCode");

    entity.HasIndex(e => e.Region)
        .HasName("Region");
});

modelBuilder.Entity<EmployeeTerritories>(entity =>
{
    entity.HasKey(e => new { e.EmployeeId, e.TerritoryId })
        .HasName("PK_EmployeeTerritories");
});

modelBuilder.Entity<Employees>(entity =>
{
    entity.HasIndex(e => e.LastName)
        .HasName("LastName");

    entity.HasIndex(e => e.PostalCode)
        .HasName("PostalCode");
});

modelBuilder.Entity<OrderDetails>(entity =>
{
    entity.HasKey(e => new { e.OrderId, e.ProductId })
        .HasName("PK_Order_Details");

    entity.HasIndex(e => e.OrderId)
        .HasName("OrdersOrder_Details");

    entity.HasIndex(e => e.ProductId)
        .HasName("ProductsOrder_Details");

    entity.Property(e => e.Discount)
        .HasDefaultValueSql("0");

    entity.Property(e => e.Quantity)
        .HasDefaultValueSql("1");
}

```

```

        entity.Property(e => e.UnitPrice)
            .HasDefaultValueSql("0");
    });

modelBuilder.Entity<Orders>(entity =>
{
    entity.HasIndex(e => e.CustomerId)
        .HasName("CustomersOrders");

    entity.HasIndex(e => e.EmployeeId)
        .HasName("EmployeesOrders");

    entity.HasIndex(e => e.OrderDate)
        .HasName("OrderDate");

    entity.HasIndex(e => e.ShipPostalCode)
        .HasName("ShipPostalCode");

    entity.HasIndex(e => e.ShipVia)
        .HasName("ShippersOrders");

    entity.HasIndex(e => e.ShippedDate)
        .HasName("ShippedDate");

    entity.Property(e => e.Freight)
        .HasDefaultValueSql("0");
});

```

```

modelBuilder.Entity<Products>(entity =>
{
    entity.HasIndex(e => e.CategoryId)
        .HasName("CategoryID");

    entity.HasIndex(e => e.ProductName)
        .HasName("ProductName");

    entity.HasIndex(e => e.SupplierId)
        .HasName("SuppliersProducts");

    entity.Property(e => e.Discontinued)
        .HasDefaultValueSql("0");

    entity.Property(e => e.ReorderLevel)
        .HasDefaultValueSql("0");

    entity.Property(e => e.UnitPrice)
        .HasDefaultValueSql("0");

    entity.Property(e => e.UnitsInStock)
        .HasDefaultValueSql("0");

    entity.Property(e => e.UnitsOnOrder)
        .HasDefaultValueSql("0");
});

```

```

modelBuilder.Entity<Region>(entity =>

```

```

        {
            entity.Property(e => e.RegionId)
                .ValueGeneratedNever();
        });

        modelBuilder.Entity<Suppliers>(entity =>
        {
            entity.HasIndex(e => e.CompanyName)
                .HasName("CompanyName");

            entity.HasIndex(e => e.PostalCode)
                .HasName("PostalCode");
        });
    }
}

public partial class Categories
{
    public Categories()
    {
        Products = new HashSet<Products>();
    }

    [Column("CategoryID")]
    [Key]
    public int CategoryId { get; set; }
    [Required]
    [MaxLength(15)]
    public string CategoryName { get; set; }
    [Column(TypeName = "ntext")]
    public string Description { get; set; }
    [Column(TypeName = "image")]
    public byte[] Picture { get; set; }

    [InverseProperty("Category")]
    public virtual ICollection<Products> Products { get; set; }
}

public partial class CustomerCustomerDemo
{
    [Column("CustomerID", TypeName = "nchar(5)")]
    public string CustomerId { get; set; }
    [Column("CustomerTypeID", TypeName = "nchar(10)")]
    public string CustomerTypeId { get; set; }

    [ForeignKey("CustomerId")]
    [InverseProperty("CustomerCustomerDemo")]
    public virtual Customers Customer { get; set; }
    [ForeignKey("CustomerTypeId")]
    [InverseProperty("CustomerCustomerDemo")]
    public virtual CustomerDemographics CustomerType { get; set; }
}

public partial class CustomerDemographics
{

```

```

public CustomerDemographics()
{
    CustomerCustomerDemo = new HashSet<CustomerCustomerDemo>();
}

[Column("CustomerTypeID", TypeName = "nchar(10)")]
[Key]
public string CustomerTypeId { get; set; }
[Column(TypeName = "ntext")]
public string CustomerDesc { get; set; }

[InverseProperty("CustomerType")]
public virtual ICollection<CustomerCustomerDemo> CustomerCustomerDemo { get;
set; }
}

public partial class Customers
{
    public Customers()
    {
        CustomerCustomerDemo = new HashSet<CustomerCustomerDemo>();
        Orders = new HashSet<Orders>();
    }

    [Column("CustomerID", TypeName = "nchar(5)")]
    [Key]
    public string CustomerId { get; set; }
    [Required]
    [MaxLength(40)]
    public string CompanyName { get; set; }
    [MaxLength(30)]
    public string ContactName { get; set; }
    [MaxLength(30)]
    public string ContactTitle { get; set; }
    [MaxLength(60)]
    public string Address { get; set; }
    [MaxLength(15)]
    public string City { get; set; }
    [MaxLength(15)]
    public string Region { get; set; }
    [MaxLength(10)]
    public string PostalCode { get; set; }
    [MaxLength(15)]
    public string Country { get; set; }
    [MaxLength(24)]
    public string Phone { get; set; }
    [MaxLength(24)]
    public string Fax { get; set; }

    [InverseProperty("Customer")]
    public virtual ICollection<CustomerCustomerDemo> CustomerCustomerDemo { get;
set; }
    [InverseProperty("Customer")]
    public virtual ICollection<Orders> Orders { get; set; }
}

```

```

public partial class Employees
{
    public Employees()
    {
        EmployeeTerritories = new HashSet<EmployeeTerritories>();
        Orders = new HashSet<Orders>();
    }

    [Column("EmployeeID")]
    [Key]
    public int EmployeeId { get; set; }
    [Required]
    [MaxLength(20)]
    public string LastName { get; set; }
    [Required]
    [MaxLength(10)]
    public string FirstName { get; set; }
    [MaxLength(30)]
    public string Title { get; set; }
    [MaxLength(25)]
    public string TitleOfCourtesy { get; set; }
    [Column(TypeName = "datetime")]
    public DateTime? BirthDate { get; set; }
    [Column(TypeName = "datetime")]
    public DateTime? HireDate { get; set; }
    [MaxLength(60)]
    public string Address { get; set; }
    [MaxLength(15)]
    public string City { get; set; }
    [MaxLength(15)]
    public string Region { get; set; }
    [MaxLength(10)]
    public string PostalCode { get; set; }
    [MaxLength(15)]
    public string Country { get; set; }
    [MaxLength(24)]
    public string HomePhone { get; set; }
    [MaxLength(4)]
    public string Extension { get; set; }
    [Column(TypeName = "image")]
    public byte[] Photo { get; set; }
    [Column(TypeName = "ntext")]
    public string Notes { get; set; }
    public int? ReportsTo { get; set; }
    [MaxLength(255)]
    public string PhotoPath { get; set; }

    [InverseProperty("Employee")]
    public virtual ICollection<EmployeeTerritories> EmployeeTerritories { get; set; }
    [InverseProperty("Employee")]
    public virtual ICollection<Orders> Orders { get; set; }
    [ForeignKey("ReportsTo")]
    [InverseProperty("InverseReportsToNavigation")]
    public virtual Employees ReportsToNavigation { get; set; }
    [InverseProperty("ReportsToNavigation")]

```

```

        public virtual ICollection<Employees> InverseReportsToNavigation { get; set; }
    }
}

public partial class EmployeeTerritories
{
    [Column("EmployeeID")]
    public int EmployeeId { get; set; }
    [Column("TerritoryID")]
    [MaxLength(20)]
    public string TerritoryId { get; set; }

    [ForeignKey("EmployeeId")]
    [InverseProperty("EmployeeTerritories")]
    public virtual Employees Employee { get; set; }
    [ForeignKey("TerritoryId")]
    [InverseProperty("EmployeeTerritories")]
    public virtual Territories Territory { get; set; }
}

[Table("Order Details")]
public partial class OrderDetails
{
    [Column("OrderID")]
    public int OrderId { get; set; }
    [Column("ProductID")]
    public int ProductId { get; set; }
    [Column(TypeName = "money")]
    public decimal UnitPrice { get; set; }
    public short Quantity { get; set; }
    public float Discount { get; set; }

    [ForeignKey("OrderId")]
    [InverseProperty("OrderDetails")]
    public virtual Orders Order { get; set; }
    [ForeignKey("ProductId")]
    [InverseProperty("OrderDetails")]
    public virtual Products Product { get; set; }
}

public partial class Orders
{
    public Orders()
    {
        OrderDetails = new HashSet<OrderDetails>();
    }

    [Column("OrderID")]
    [Key]
    public int OrderId { get; set; }
    [Column("CustomerID", TypeName = "nchar(5)")]
    public string CustomerId { get; set; }
    [Column("EmployeeID")]
    public int? EmployeeId { get; set; }
    [Column(TypeName = "datetime")]
    public DateTime? OrderDate { get; set; }
}

```

```

[Column(TypeName = "datetime")]
public DateTime? RequiredDate { get; set; }
[Column(TypeName = "datetime")]
public DateTime? ShippedDate { get; set; }
public int? ShipVia { get; set; }
[Column(TypeName = "money")]
public decimal? Freight { get; set; }
[MaxLength(40)]
public string ShipName { get; set; }
[MaxLength(60)]
public string ShipAddress { get; set; }
[MaxLength(15)]
public string ShipCity { get; set; }
[MaxLength(15)]
public string ShipRegion { get; set; }
[MaxLength(10)]
public string ShipPostalCode { get; set; }
[MaxLength(15)]
public string ShipCountry { get; set; }

[InverseProperty("Order")]
public virtual ICollection<OrderDetails> OrderDetails { get; set; }
[ForeignKey("CustomerId")]
[InverseProperty("Orders")]
public virtual Customers Customer { get; set; }
[ForeignKey("EmployeeId")]
[InverseProperty("Orders")]
public virtual Employees Employee { get; set; }
[ForeignKey("ShipVia")]
[InverseProperty("Orders")]
public virtual Shippers ShipViaNavigation { get; set; }
}

public partial class Products
{
    public Products()
    {
        OrderDetails = new HashSet<OrderDetails>();
    }

    [Column("ProductID")]
    [Key]
    public int ProductId { get; set; }
    [Required]
    [MaxLength(40)]
    public string ProductName { get; set; }
    [Column("SupplierID")]
    public int? SupplierId { get; set; }
    [Column("CategoryID")]
    public int? CategoryId { get; set; }
    [MaxLength(20)]
    public string QuantityPerUnit { get; set; }
    [Column(TypeName = "money")]
    public decimal? UnitPrice { get; set; }
    public short? UnitsInStock { get; set; }
    public short? UnitsOnOrder { get; set; }
}

```



```

    public short? ReorderLevel { get; set; }
    public bool Discontinued { get; set; }

    [InverseProperty("Product")]
    public virtual ICollection<OrderDetails> OrderDetails { get; set; }
    [ForeignKey("CategoryId")]
    [InverseProperty("Products")]
    public virtual Categories Category { get; set; }
    [ForeignKey("SupplierId")]
    [InverseProperty("Products")]
    public virtual Suppliers Supplier { get; set; }
}

public partial class Region
{
    public Region()
    {
        Territories = new HashSet<Territories>();
    }

    [Column("RegionID")]
    public int RegionId { get; set; }
    [Required]
    [Column(TypeName = "nchar(50)")]
    public string RegionDescription { get; set; }

    [InverseProperty("Region")]
    public virtual ICollection<Territories> Territories { get; set; }
}

public partial class Shippers
{
    public Shippers()
    {
        Orders = new HashSet<Orders>();
    }

    [Column("ShipperID")]
    [Key]
    public int ShipperId { get; set; }
    [Required]
    [MaxLength(40)]
    public string CompanyName { get; set; }
    [MaxLength(24)]
    public string Phone { get; set; }

    [InverseProperty("ShipViaNavigation")]
    public virtual ICollection<Orders> Orders { get; set; }
}

public partial class Suppliers
{
    public Suppliers()
    {
        Products = new HashSet<Products>();
    }
}

```

```

[Column("SupplierID")]
[Key]
public int SupplierId { get; set; }
[Required]
[MaxLength(40)]
public string CompanyName { get; set; }
[MaxLength(30)]
public string ContactName { get; set; }
[MaxLength(30)]
public string ContactTitle { get; set; }
[MaxLength(60)]
public string Address { get; set; }
[MaxLength(15)]
public string City { get; set; }
[MaxLength(15)]
public string Region { get; set; }
[MaxLength(10)]
public string PostalCode { get; set; }
[MaxLength(15)]
public string Country { get; set; }
[MaxLength(24)]
public string Phone { get; set; }
[MaxLength(24)]
public string Fax { get; set; }
[Column(TypeName = "ntext")]
public string HomePage { get; set; }

[InverseProperty("Supplier")]
public virtual ICollection<Products> Products { get; set; }
}

public partial class Territories
{
    public Territories()
    {
        EmployeeTerritories = new HashSet<EmployeeTerritories>();
    }

    [Column("TerritoryID")]
    [MaxLength(20)]
    [Key]
    public string TerritoryId { get; set; }
    [Required]
    [Column(TypeName = "nchar(50)")]
    public string TerritoryDescription { get; set; }
    [Column("RegionID")]
    public int RegionId { get; set; }

    [InverseProperty("Territory")]
    public virtual ICollection<EmployeeTerritories> EmployeeTerritories { get;
set; }
    [ForeignKey("RegionId")]
    [InverseProperty("Territories")]
    public virtual Region Region { get; set; }
}

```

Fournisseurs de données

Les fournisseurs de données sont des bibliothèques complémentaires pour Entity Framework Core qui ont la responsabilité de l'implémentation des fonctionnalités propres à la source de données sur laquelle elles opèrent.

Le principe clé derrière l'existence des fournisseurs de données est l'isolation des traitements spécifiques aux sources de données. Entity Framework Core implémente pour cela une bibliothèque centrale, qui contient les éléments généraux, et qui attend des implémentations particulières de traitements qui sont, eux, gérés par les fournisseurs de données. Ces derniers peuvent donc, d'un point de vue purement technique, être considérés comme des plug-ins, avec la condition suivante : il est nécessaire qu'au moins une de ces bibliothèques d'extension soit présente pour tirer parti de la bibliothèque centrale. Dans le cas contraire, rien ne peut fonctionner.

Conformément à ce que nous venons de voir, les fournisseurs de données pour Entity Framework Core sont livrés sous la forme d'assemblées complémentaires. Ils sont intégrés à une application par l'intermédiaire du gestionnaire de package NuGet. Au jour de l'écriture de ces lignes, huit fournisseurs de données non commerciaux sont disponibles.

SQL Server

Ce fournisseur est utilisé avec le moteur SQL Server, dans ses versions 2008 et supérieures. Il est également compatible avec SQL Azure, la version "cloud" de la base de données.

Auteur : Microsoft.

Package NuGet associé : Microsoft.EntityFrameworkCore.SqlServer.

SQL Server Compact Edition

Ce fournisseur permet l'accès aux bases de données Microsoft SQL Server Compact Edition. Il a la particularité d'être scindé en deux assemblées supportant chacun une version différente de cette base de données.

Auteur : Erik Ejlskov Jensen (<http://erikej.blogspot.fr/>).

Packages NuGet associés :

- EntityFrameworkCore.SqlServerCompact40 pour la version 4.0.
- EntityFrameworkCore.SqlServerCompact35 pour la version 3.5.

SQLite

Ce fournisseur permet d'utiliser Entity Framework Core avec le moteur SQLite à partir de sa version 3.7.

Auteur : Microsoft.

Package NuGet associé : Microsoft.EntityFrameworkCore.SQLite.

PostgreSQL

Ce fournisseur offre la possibilité d'utiliser Entity Framework Core en conjonction avec les bases de données PostgreSQL.

Auteur : Npgsql (<http://www.npgsql.org/>).

Package NuGet associé : Npgsql.EntityFrameworkCore.PostgreSQL.

MySQL (officiel)

Ce fournisseur offre la possibilité d'utiliser Entity Framework Core pour accéder à des bases de données MySQL.

Auteur : MySQL (<http://dev.mysql.com/>).

Package NuGet associé : MySql.Data.EntityFrameworkCore (en version préliminaire actuellement).

MySQL (Pomelo)

Ce fournisseur est dédié aux interactions avec des bases de données MySQL.

Auteur : Pomelo Foundation (<https://github.com/PomeloFoundation/Pomelo.EntityFrameworkCore.MySql>).

Package NuGet associé : Pomelo.EntityFrameworkCore.MySql.

MySQL (Sapient Guardian)

Ce fournisseur permet de manipuler des bases de données MySQL.

Auteur : Noah Potash (<http://outbreaklabs.com/>).

Package NuGet associé : SapientGuardian.EntityFrameworkCore.MySql.

In Memory

Ce dernier fournisseur travaille avec des données en mémoire uniquement. Les données sont donc réinitialisées à chaque démarrage d'une application utilisant ce fournisseur. Pour cette raison, il ne devrait être utilisé qu'à des fins de tests (prototypes, tests unitaires, tests fonctionnels...).

Cette liste ne prend en compte que les fournisseurs de données présentés dans la documentation d'Entity Framework Core. D'autres fournisseurs peuvent bien évidemment être disponibles ou en cours de développement.

Requêtage avec LINQ

Avec la sortie du framework .NET en version 3.5 en fin d'année 2007, Microsoft a introduit une nouvelle manière de manipuler les données, qu'elles soient locales ou issues d'une source de données externe. Cette petite révolution se présente sous un nom simple : LINQ (*Language INtegrated-Query*). Cette interface permet d'interroger toutes les sources de données compatibles d'une manière complètement unifiée, tout en offrant l'avantage non négligeable de vérifier la validité des requêtes dès la phase de compilation.

1. API et extension à C#

LINQ est une brique logicielle constituée, d'une part, d'une bibliothèque de classes fournissant des capacités de requêtage et, d'autre part, d'un jeu d'extensions pour le langage C# permettant l'utilisation de mots-clés spécifiques pour l'écriture de requêtes avec une syntaxe proche de celle de SQL.

L'extension du langage est un sucre syntaxique qui vise à simplifier l'écriture des requêtes, mais les expressions utilisant la syntaxe LINQ sont traduites par le compilateur en appels de méthodes conventionnels. On peut en déduire que toute requête LINQ peut être réécrite avec des appels de méthodes. L'inverse n'est pas vrai, certaines méthodes n'ayant pas d'équivalent dans la syntaxe LINQ. Il n'existe notamment aucun mot-clé traduisant la méthode `FirstOrDefault` dans l'extension LINQ de C#.

L'existence de ces deux syntaxes a pour avantage de laisser au développeur le choix d'utiliser celle avec laquelle il se sent le plus à l'aise. Certains préfèrent en effet écrire des requêtes de manière fluide en n'utilisant que des mots-clés du langage. D'autres, dont l'auteur de cet ouvrage, ont plus d'affinités avec l'utilisation de fonctions et de procédures.

LINQ est capable de travailler sur différentes sources de données allant de la collection d'objets locale jusqu'aux bases de données SQL Server en passant par les fichiers XML ou n'importe quelle autre source pour laquelle existe un fournisseur LINQ. Toutes ces sources de données peuvent être interrogées à l'aide des mêmes requêtes LINQ grâce au travail effectué par les fournisseurs de données. Un fournisseur LINQ transforme les données de la source qui lui correspond en objets .NET, et inversement. Ainsi, les requêtes LINQ ne travaillent que sur des objets .NET fortement typés, la communication avec les sources de données étant sous la responsabilité des fournisseurs.

La structure de LINQ lui permet d'être étendu de manière à supporter toujours plus de sources de données différentes. Dans le cadre d'Entity Framework Core, les fournisseurs de données sont responsables de l'implémentation de ces extensions. Le fournisseur de données SQL Server peut ainsi générer du code SQL optimisé pour le moteur avec lequel il travaille, tandis que le fournisseur SQLite peut générer un code SQL respectant les limitations du moteur.

2. Première requête LINQ

Avant d'explorer en détail les différentes méthodes de LINQ qui peuvent être utilisées pour écrire des requêtes, nous allons étudier un premier exemple qui vous permettra de mieux comprendre ce qu'est LINQ et comment l'utiliser.

Les requêtes LINQ doivent respecter certaines règles pour être valides. Les sources de données compatibles avec cette brique logicielle sont les collections implémentant l'interface `IEnumerable<T>`. Les propriétés décrites dans la classe `NorthwindContext` sont, quant à elles, de type `DbSet<T>`, qui implémente entre autres cette interface.

Lorsque l'on utilise la syntaxe intégrée à C#, les requêtes LINQ ont une forme proche de la syntaxe SQL. Elles sont constituées de plusieurs éléments qui définissent la source de données à utiliser, les données à récupérer et les filtres et traitements nécessaires à la récupération de résultats pertinents. Seuls les deux premiers éléments sont indispensables : une clause `from` permet de définir la collection source tandis que la clause `select` permet de définir un objet qui doit être retourné pour chaque enregistrement correspondant aux critères de recherche.

```
from <variable> in <collection source>
...
select <objet à retourner>
```

Cette première requête récupère la liste des clients situés en France.

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

Le type de retour de cette requête est `IQueryable<Customer>`, mais il est courant d'utiliser l'inférence de type avec les requêtes LINQ, notamment car la clause `select` peut créer des objets de type anonyme.

Nous étudierons ce type de retour plus en détail à la section LINQ dans le détail - `IEnumerable<T>` et `IQueryable<T>` : les différences, qui traite de la différence entre les interfaces `IEnumerable<T>` et `IQueryable<T>`.

L'instanciation de l'objet `NorthwindContext` peut être incluse dans un bloc `using` car ce type implémente l'interface `IDisposable`. Cette manière de faire est considérée comme une bonne pratique car elle permet de libérer les connexions utilisées dès que possible.

Lorsqu'elle est écrite à l'aide des méthodes fournies par l'API LINQ, cette méthode prend la forme suivante :

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

Dans ce cas, la fonction `select` est optionnelle : pour chaque élément retourné par la fonction `where`, elle renvoie ledit élément. Dans ce cas particulier, il est possible de simplifier l'écriture en terminant l'instruction après l'appel de la méthode `where`.

À ce stade, malgré les apparences, la variable `requete` ne contient aucun objet de type `Customer`. Elle ne contient qu'une définition de la requête qui sera en réalité exécutée de manière paresseuse, c'est-à-dire uniquement lorsque ses données de retour devront être lues. Ce mécanisme permet notamment d'exécuter plusieurs fois la même requête LINQ sans avoir besoin de la redéfinir.

Ici, l'exécution de l'opération de recherche est déclenchée par l'utilisation d'une boucle `foreach`, mais elle peut également être lancée par le calcul d'une valeur d'agrégat (`Count` ou `Max`, notamment) ou par l'utilisation de certaines méthodes récupérant un ou plusieurs éléments sous une forme spécifique (`First` et `ToList`, par exemple). On parle dans ce cas d'opérateurs non différés, par opposition aux opérateurs différés, qui ne sont exécutés que lorsque l'énumération des résultats est effectuée.

```
//Affichage du nom et de l'adresse de chaque client
//On affiche également le pays pour vérification.
foreach (var resultat in requete)
{
    Console.WriteLine(resultat.Nom);
    Console.WriteLine("\t" + resultat.Adresse);
    Console.WriteLine("\t" + resultat.Ville);
    Console.WriteLine("\t" + resultat.Pays);
}
```

Le résultat de l'exécution de ce code est présenté ci-après. On constate que chacun des clients est bien localisé en France.

```
Frédérique Citeaux
    24, place Kléber
    Strasbourg
    France
Laurence Lebihan
    12, rue des Bouchers
    Marseille
    France
Janine Labrune
    67, rue des Cinquante Otages
    Nantes
    France
Martine Rancé
    184, chaussée de Tournai
    Lille
    France
Carine Schmitt
    54, rue Royale
    Nantes
    France
Daniel Tonini
    67, avenue de l'Europe
    Versailles
    France
Annette Roulet
    1 rue Alsace-Lorraine
    Toulouse
    France
Marie Bertrand
    265, boulevard Charonne
    Paris
    France
Dominique Perrier
    25, rue Lauriston
    Paris
    France
Mary Saveley
    2, rue du Commerce
    Lyon
    France
Paul Henriot
```

LINQ dans le détail

Dans un monde de plus en plus gouverné par les données, la simplicité de LINQ est un atout non négligeable pour l'écriture d'applications robustes et efficaces. Sa puissance est d'autant plus grande qu'il propose de nombreux opérateurs permettant d'effectuer une grande variété de traitements.

Tous les opérateurs définis par LINQ sont disponibles sous la forme de méthodes d'extension utilisables sur les objets de type `IEnumerable<T>`. Dans le cas d'Entity Framework, les méthodes utilisées sont plus exactement redéfinies sur le type `IQueryable<T>`, qui hérite notamment de `IEnumerable<T>`. Nous commencerons par expliquer la différence entre ces deux interfaces, puis nous détaillerons les différents opérateurs définis par Entity Framework Core. Lorsque l'un d'eux est disponible aussi bien sous la forme d'une méthode qu'en tant que mot-clé du langage C#, les deux modes d'écriture seront détaillés.

1. IEnumerable et IQueryable : les différences

Depuis sa toute première version, la bibliothèque de classes du framework .NET intègre une interface nommée `IEnumerable`, dont l'objectif est de fournir l'infrastructure nécessaire à l'exécution du code généré par l'utilisation de boucles `foreach`. Le code IL (*Intermediate Language*) associé à ces boucles utilise en effet la méthode `GetEnumerator()` du type `IEnumerable` pour obtenir un objet capable de référencer chaque élément de la collection. Le pendant générique de cette interface, `IEnumerable<T>`, a fait son apparition avec l'arrivée de .NET 2.0, toujours dans le même objectif, mais en ajoutant le support du typage fort sur les éléments de la collection itérée.

La librairie `System.Linq`, présente depuis le framework 3.5, inclut quant à elle une interface nommée `IQueryable<T>`, qui est destinée à fournir un point d'entrée pour l'extension de LINQ à l'aide de fournisseurs de données. Cette interface hérite d'`IEnumerable<T>`, ce qui implique un fonctionnement similaire. La propriété `Expression` des objets implémentant `IQueryable<T>` contient un arbre d'expressions représentant une requête. Lorsque l'objet est énuméré, typiquement à l'aide d'une boucle `foreach`, cet arbre est passé au fournisseur associé à la collection courante afin qu'il traite la requête de la manière la plus adaptée à la source de données.

Le type `DbSet<T>` utilisé lors de la phase de modélisation est une implémentation concrète de l'interface `IQueryable<T>`. Les collections définies dans les classes de contexte sont liées aux fournisseurs de données par l'intermédiaire de cette interface. À l'énumération, le fournisseur peut ainsi générer des requêtes SQL, des appels à des services OData, effectuer une recherche en mémoire, ou toute autre opération pertinente en liaison avec la source de données. Ce comportement implique également que de multiples énumérations d'un objet `IQueryable<T>` entraînent plusieurs générations et exécutions de requêtes, quelle que soit la source de données cible.

En résumé, il est très important de comprendre que ce n'est pas la création d'une variable de type `IQueryable<T>` qui exécute une requête SQL. Cette dernière est générée et exécutée de manière différée, à l'utilisation de la variable : avec une boucle `foreach` ou avec un opérateur renvoyant un objet d'un type différent de `IQueryable<T>`. Il ne faut donc pas confondre `IQueryable<T>` et

`IEnumerable<T>` : malgré leurs similarités, faire l'erreur d'utiliser un objet `IQueryable<T>` comme un `IEnumerable<T>` peut coûter cher.

2. Opérateurs de projection

La projection est le processus permettant de transformer les éléments présents dans une séquence de manière à constituer une nouvelle séquence contenant les éléments transformés. Le type de destination ne doit pas nécessairement être déclaré explicitement : comme tout objet instancié en C#, il peut être d'un type anonyme. LINQ définit deux opérateurs de projection :

`Select` et `SelectMany`.

a. Select

L'opérateur `Select` transforme indépendamment chaque élément d'une collection et renvoie une nouvelle collection contenant les objets qui résultent de la transformation. La séquence source et la séquence retournée comptent toujours le même nombre d'éléments.

La méthode `Select` accepte comme paramètre une fonction permettant d'effectuer l'opération de transformation sur un élément de la collection. Elle est appelée autant de fois qu'il y a d'objets dans la séquence pour aboutir à la génération de la nouvelle séquence.

L'exemple suivant récupère les noms et pays des clients enregistrés dans la base de données.

```
using (NorthwindContext context = new NorthwindContext())
{
    var requete = context.Customers
        .Select((client) => new
        {
            Nom = client.ContactName,
            Pays = client.Country
        });

    //Affichage du nom et du pays de chaque client
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.Nom} - {resultat.Pays}");
    }
}
```

L'exécution de cette portion de code affiche une liste de 91 clients (volontairement réduite ici).

```
Maria Anders - Germany
Ana Trujillo - Mexico
Antonio Moreno - Mexico
Thomas Hardy - UK
Christina Berglund - Sweden
Hanna Moos - Germany
Frédérique Citeaux - France
Martín Sommer - Spain
Laurence Lebihan - France

...

Palle Ibsen - Denmark
```

```
Mary Saveley - France
Paul Henriot - France
Rita Müller - Germany
Pirkko Koskitalo - Finland
Paula Parente - Brazil
Karl Jablonski - USA
Matti Karttunen - Finland
Zbyszek Piestrzeniewicz - Poland
```

Le SQL généré et exécuté pour récupérer ces résultats est le suivant :

```
SELECT [client].[ContactName], [client].[Country]
FROM [Customers] AS [client]
```

On peut remarquer deux éléments importants dans cette requête :

- Les colonnes placées dans la clause `SELECT` sont au nombre de deux, et correspondent aux propriétés utilisées pour créer les objets de la séquence résultante.
- Un alias portant le nom du paramètre de la fonction de la transformation est créé pour la table `Customers`. Ce point a notamment son importance lorsqu'il s'agit d'écrire des requêtes joignant plusieurs tables.

La requête peut être réécrite pour utiliser l'extension LINQ du langage C#. Le mot-clé associé est `select`.

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

b. SelectMany

Dans certains cas, il peut être nécessaire que la projection d'un élément source retourne plusieurs objets que l'on souhaite intégrer dans la séquence retournée par la requête. Ces cas ne sont pas couverts par l'opérateur `Select` : il retournerait une séquence contenant à son tour plusieurs collections.

```
var requete = context.Customers
               .Select(client => client.Orders);
```

 (extension) IQueryable<ICollection<Orders>> IQueryable<Customers>.Select<Customers, ICollection<Orders>>> (System.Linq.Expressions.Expression<Func<Customers, ICollection<Orders>>> selector) (+ 3 surcharges)
Projette chaque élément d'une séquence dans un nouveau formulaire.
Exceptions :
ArgumentNullException

Il est possible de combiner l'ensemble de ces "sous-collections" pour former la séquence de destination. Pour cela, il convient d'utiliser l'opérateur `SelectMany`. Celui-ci fonctionne de la même manière que l'opérateur `Select`, à une petite différence près : la fonction qui lui est passée en paramètre doit renvoyer une collection.

Attention : la fonction passée en paramètre à l'opérateur `SelectMany` peut renvoyer une chaîne de caractères, car ce type est considéré comme une collection de caractères. Ce comportement est rarement désirable !

La portion de code ci-après affiche l'identifiant et la date de chaque commande en prenant la liste des clients comme point d'entrée. Il serait bien évidemment plus simple à tous niveaux d'interroger directement la collection `context.Orders`, mais cette requête pourra notamment être modifiée de manière à filtrer les clients pour lesquels on récupère les commandes.

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

    //Affichage de l'identifiant et de la date de chaque commande
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.OrderId} -
{resultat.OrderDate?.ToString("dd/MM/yyyy")}");
    }
}
```

La requête SQL générée est nettement plus complexe que la précédente :

```
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]
```

Cette instruction `SELECT` effectue exactement le traitement attendu : elle renvoie la liste des commandes pour chacun des clients. On peut néanmoins lui reprocher de ramener bien trop de données au vu de l'affichage que l'on effectue : quatorze champs alors que deux seraient suffisants. Pour parer à ce problème, il est possible de combiner les opérateurs `Select` et `SelectMany` :

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

```

    }));

    //Affichage de l'identifiant et de la date de chaque commande
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.Id} -
{resultat.Date?.ToString("dd/MM/yyyy")}");
    }
}

```

Pour un résultat parfaitement identique quant à l'affichage, l'instruction SQL exécutée est bien plus simple :

```

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

```

3. Opérateurs de partitionnement

Les opérateurs de partitionnement ont pour objectif de scinder une collection en fonction de l'argument qui leur est passé et de retourner un des sous-ensembles résultants de l'opération. Ils sont notamment utilisés dans la création de données paginées.

a. Take

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

L'affichage du nom et de la société des dix premiers clients présents dans la table `Customers` est effectué à l'aide du code suivant :

```

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

    //Affichage du nom et de la société de chaque client
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.ContactName} - '{resultat.CompanyName}'");
    }
}

```

Ci-après, le résultat affiché dans la console.

```
Maria Anders - 'Alfreds Futterkiste'  
Ana Trujillo - 'Ana Trujillo Emparedados y helados'  
Antonio Moreno - 'Antonio Moreno Taquería'  
Thomas Hardy - 'Around the Horn'  
Christina Berglund - 'Berglunds snabbköp'  
Hanna Moos - 'Blauer See Delikatessen'  
Frédérique Citeaux - 'Blondesdds1 père et fils'  
Martín Sommer - 'Bólido Comidas preparadas'  
Laurence Lebihan - 'Bon app''  
Elizabeth Lincoln - 'Bottom-Dollar Markets'
```

Le code SQL généré tire ici parti de l'opérateur `TOP()` disponible en T-SQL pour ne récupérer que le nombre d'éléments souhaités. Ce nombre est indiqué à travers l'utilisation du paramètre `@__p_0`.

```
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]
```

b. Skip

L'opérateur `skip` ignore un nombre défini d'éléments placés en tête de la collection source et retourne une séquence contenant l'ensemble des autres éléments.

La table `Customers` contient 91 enregistrements. Pour afficher le nom et la société des dix derniers clients présents, il faut donc ignorer les 81 premiers enregistrements.

```
using (var context = new NorthwindContext())  
{  
    var requete = context.Customers  
        .Skip(81);  
  
    //Affichage du nom et de la société de chaque client  
    foreach (var resultat in requete)  
    {  
        Console.WriteLine($"{resultat.ContactName} - '{resultat.CompanyName}'");  
    }  
}
```

Ci-après, le résultat affiché dans la console.

```
Helvetius Nagy - 'Trail's Head Gourmet Provisioners'  
Palle Ibsen - 'Vaffeljernet'  
Mary Saveley - 'Victuailles en stock'  
Paul Henriot - 'Vins et alcools Chevalier'  
Rita Müller - 'Die wandernde Kuh'  
Pirkko Koskitalo - 'Wartian Herkku'  
Paula Parente - 'Wellington Importadora'  
Karl Jablonski - 'White Clover Markets'  
Matti Karttunen - 'Wilman Kala'  
Zbyszek Piestrzeniewicz - 'Wolski Zajazd'
```

Le SQL généré utilise une clause T-SQL `OFFSET` afin d'ignorer le nombre d'éléments passé via le paramètre `@__p_0`, soit 81.

```
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
```

4. Opérateurs de tri

Les opérateurs de tri permettent d'ordonner les éléments de la collection source, notamment pour éviter les aléas liés à l'utilisation de clusters de données. En effet, lorsque les données proviennent de plusieurs sources, une même requête non triée et exécutée plusieurs fois peut donner plusieurs résultats différents. Les données peuvent également être triées sur commande d'un utilisateur, de manière à lui fournir les données sous une forme qu'il peut exploiter.

a. OrderBy

L'opérateur `OrderBy`, comme son nom l'indique, ordonne les éléments issus de la collection source en comparant les valeurs d'une clé donnée.

Propriété

La clé fournie pour le tri est généralement une propriété des objets comparés.

L'exemple de code suivant affiche une liste des entreprises clientes triée par ordre alphabétique.

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

    //Affichage du nom de la société
    foreach (var resultat in requete)
    {
        Console.WriteLine(resultat.CompanyName);
    }
}
```

Le résultat (volontairement raccourci) de son exécution correspond parfaitement à ce qui est attendu.

```
Alfreds Futterkiste
Ana Trujillo Emparedados y helados
Antonio Moreno Taquería
Around the Horn
Berglunds snabbköp
...

Tradição Hipermercados
Trail's Head Gourmet Provisioners
```

```
Vaffeljernet  
Victuailles en stock  
Vins et alcools Chevalier  
Wartian Herkku  
Wellington Importadora  
White Clover Markets  
Wilman Kala  
Wolski Zajazd
```

L'analyse du code SQL généré montre l'utilisation d'une clause `ORDER BY` sur la colonne `CompanyName` pour gérer l'ordonnancement des éléments.

```
SELECT [client].[CustomerID], [client].[Address],  
       [client].[City], [client].[CompanyName],  
       [client].[ContactName], [client].[ContactTitle],  
       [client].[Country], [client].[Fax], [client].[Phone],  
       [client].[PostalCode], [client].[Region]  
FROM [Customers] AS [client]  
ORDER BY [client].[CompanyName]
```

Cet opérateur est intégré au langage C# sous la forme du mot-clé `orderby`.

```
using (var context = new NorthwindContext())  
{  
    var requete = from client in context.Customers  
                  orderby client.CompanyName  
                  select client;  
  
    //Affichage du nom de la société  
    foreach (var resultat in requete)  
    {  
        Console.WriteLine(resultat.CompanyName);  
    }  
}
```

Il est possible d'adjoindre à cette clause le mot-clé `ascending`, mais cette notation est optionnelle : les tris sont ascendants par défaut.

```
from client in context.Customers  
orderby client.CompanyName ascending  
select client;
```

Le tri peut également être effectué relativement au résultat d'une projection plus complexe. Il est par exemple possible de trier les clients en fonction de la longueur du nom de leur entreprise.

```

using (var context = new NorthwindContext())
{
    var requete = context.Customers
        .OrderBy(client => client.CompanyName.Length);

    //Affichage du nom de la société
    foreach (var resultat in requete)
    {
        Console.WriteLine(resultat.CompanyName);
    }
}

```

Le résultat produit par l'exécution de ce code est évidemment très prévisible.

```

Bon app'
QUICK-Stop
Que Delícia
North/South
Wilman Kala
Vaffeljernet
Maison Dewey
Ernst Handel
Hanari Carnes
B's Beverages
Queen Cozinha
Rancho grande
Santé Gourmet
Simons bistro

...

Lonesome Pine Restaurant
Pericles Comidas clásicas
Bólido Comidas preparadas
Vins et alcools Chevalier
Cactus Comidas para llevar
Centro comercial Moctezuma
Hungry Coyote Import Store
Rattlesnake Canyon Grocery
Magazzini Alimentari Riuniti
Hungry Owl All-Night Grocers
Laughing Bacchus Wine Cellars
Furia Bacalhau e Frutos do Mar
Trail's Head Gourmet Provisioners
Ana Trujillo Emparedados y helados
FISSA Fabrica Inter. Salchichas S.A.

```

L'élément le plus important dans l'exécution de cette opération se cache dans le code SQL généré.


```
SELECT [client].[CustomerID], [client].[Address], [client].[City],
[client].[CompanyName], [client].[ContactName],
[client].[ContactTitle], [client].[Country], [client].[Fax],
[client].[Phone], [client].[PostalCode], [client].[Region]
FROM [Customers] AS [client]
ORDER BY LEN([client].[CompanyName])
```

On note l'utilisation d'une fonction T-SQL spécifique permettant de récupérer la longueur des chaînes de caractères directement au niveau de la source de données. Les fournisseurs de données ont en effet toute latitude pour la transformation des arbres d'expressions qui leur sont passés : ils peuvent ainsi exploiter au maximum les capacités des moteurs sous-jacents en traitant des éléments très spécifiques, comme la propriété `Length` du type `string`. Ce type d'optimisation est bien évidemment dépendant des fonctionnalités de la source de données.

IComparable

Dans la majorité des cas, le tri est effectué sur une propriété simple des objets de la séquence source, mais il peut parfois être nécessaire d'ordonner les éléments en fonction d'un objet complexe. L'opérateur `orderBy` permet d'effectuer ce type d'opération à la condition que l'objet complexe utilisé pour le tri implémente l'interface `IComparable`.

Cette fonctionnalité est déjà utilisée implicitement depuis le début de la section concernant `orderBy` : les types alphanumériques et numériques (entre autres) implémentent nativement l'interface `IComparable`.

Pour montrer l'application de cette fonctionnalité, l'exemple ci-après trie la liste des clients en fonction de la somme des longueurs de leurs noms de contact et d'entreprise. Ce tri pourrait également parfaitement être exécuté en fournissant une expression à l'opérateur `orderBy`.

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

    //Affichage de la société de chaque client
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.ContactName} - {resultat.CompanyName}");
    }
}
```

```
class ClientComparable : IComparable
{
    public string ContactName { get; set; }
    public string CompanyName { get; set; }

    public int CompareTo(object obj)
    {

```

```

        var client2 = obj as ClientComparable;

        var thisLength = this.ContactName.Length + this.CompanyName.Length;
        var client2Length = client2.ContactName.Length +
client2.CompanyName.Length;

        if (thisLength > client2Length) return 1;
        else if (thisLength == client2Length) return 0;
        return -1;
    }
}

```

Ce code produit la sortie suivante :

```

Horst Kloss - 'QUICK-Stop'
Liz Nixon - 'The Big Cheese'
Liu Wong - 'The Cracker Box'
Palle Ibsen - 'Vaffeljernet'
Laurence Lebihan - 'Bon app''
Roland Mendel - 'Ernst Handel'
Mario Pontes - 'Hanari Carnes'
Simon Crowther - 'North/South'
Yang Wang - 'Chop-suey Chinese'
Matti Karttunen - 'Wilman Kala'
Thomas Hardy - 'Around the Horn'

...

Francisco Chang - 'Centro comercial Moctezuma'
Frédérique Citeaux - 'Blondesdds1 père et fils'
Patricio Simpson - 'Cactus Comidas para llevar'
Lino Rodriguez - 'Furia Bacalhau e Frutos do Mar'
Patricia McKenna - 'Hungry Owl All-Night Grocers'
Yoshi Tannamuri - 'Laughing Bacchus Wine Cellars'
Giovanni Rovelli - 'Magazzini Alimentari Riuniti'
Guillermo Fernández - 'Pericles Comidas clásicas'
Ana Trujillo - 'Ana Trujillo Emparedados y helados'
Diego Roel - 'FISSA Fabrica Inter. Salchichas S.A.'
Helvetius Nagy - 'Trail's Head Gourmet Provisioners'

```

Le code SQL généré est très simple, le traitement de comparaison étant effectué du côté applicatif, et non au niveau de la source de données.

```

SELECT [client].[CustomerID], [client].[Address], [client].[City],
[client].[CompanyName], [client].[ContactName],
[client].[ContactTitle], [client].[Country], [client].[Fax],
[client].[Phone], [client].[PostalCode], [client].[Region]
FROM [Customers] AS [client]

```

b. OrderByDescending

L'opérateur `OrderByDescending` fonctionne exactement de la même manière que l'opérateur `OrderBy`, à la différence qu'il renvoie les données dans l'ordre inverse, comme le montre l'exemple de code suivant.

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

    //Affichage du nom de la société
    foreach (var resultat in requete)
    {
        Console.WriteLine(resultat.CompanyName);
    }
}
```

Son exécution retourne bien la liste des clients triée par ordre alphabétique inverse.

```
wolski Zajazd
wilman Kala
white Clover Markets
wellington Importadora
wartian Herkku
vins et alcools Chevalier
victuailles en stock
vaffeljernet
Trail's Head Gourmet Provisioners
Tradição Hipermercados

...

Bólido Comidas preparadas
Blondesddsl père et fils
Blauer See Delikatessen
Berglunds snabbköp
Around the Horn
Antonio Moreno Taquería
Ana Trujillo Emparedados y helados
Alfreds Futterkiste
```

Ce tri inverse est effectué par l'ajout d'un mot-clé `DESC` dans la requête SQL générée.

```
SELECT [client].[CustomerID], [client].[Address], [client].[City],
[client].[CompanyName], [client].[ContactName],
[client].[ContactTitle], [client].[Country], [client].[Fax],
[client].[Phone], [client].[PostalCode], [client].[Region]
FROM [Customers] AS [client]
ORDER BY [client].[CompanyName] DESC
```

Comme l'opérateur `OrderBy`, `OrderByDescending` peut être utilisé avec toute clé complexe dont le type implémente `IComparable`.

```
using (var context = new NorthwindContext())
{
    var requete =
        context.Customers
            .OrderByDescending(client => new ClientComparable
            {
                ContactName = client.ContactName,
                CompanyName = client.CompanyName
            });

    //Affichage de la société de chaque client
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.ContactName} - {resultat.CompanyName}");
    }
}
```

Le type `ClientComparable` utilisé pour créer la clé de tri est implémenté de la manière suivante :

```
class ClientComparable : IComparable
{
    public string ContactName { get; set; }
    public string CompanyName { get; set; }

    public int CompareTo(object obj)
    {
        var client2 = obj as ClientComparable;

        var thisLength = this.ContactName.Length + this.CompanyName.Length;
        var client2Length = client2.ContactName.Length +
            client2.CompanyName.Length;

        if (thisLength > client2Length) return 1;
        else if (thisLength == client2Length) return 0;
        return -1;
    }
}
```

Cet opérateur est également intégré au langage C# : il suffit d'ajouter le mot-clé `descending` à la suite d'une clause `orderby`.

```
using (var context = new NorthwindContext())
{
    var requete = from client in context.Customers
                  orderby client.CompanyName descending
                  select client;

    //Affichage du nom de la société
    foreach (var resultat in requete)
    {
        Console.WriteLine(resultat.CompanyName);
    }
}
```

c. ThenBy

À la différence des opérateurs vus jusqu'ici, l'opérateur `ThenBy` n'est pas défini sur l'interface `IQueryable`, mais sur le type `IOrderedQueryable`. Son utilisation suppose donc que la séquence sur laquelle il opère ait été préalablement triée à l'aide des opérateurs `OrderBy` ou `OrderByDescending`.

`ThenBy` permet d'ajouter un niveau de tri sur la séquence source. Il est ainsi possible d'ordonner les éléments de chaque groupe correspondant à une valeur de clé du tri précédent. L'exemple suivant montre les effets de cet opérateur en triant la liste des employés par pays, puis par ordre alphabétique sur leur prénom.

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

    //Affichage du pays et du prénom des employés
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.Country} - {resultat.FirstName}");
    }
}
```

La liste des employés est affichée sous la forme suivante :

```
UK - Anne
UK - Michael
UK - Robert
UK - Steven
USA - Andrew
USA - Janet
USA - Laura
USA - Margaret
USA - Nancy
```

On constate bien que l'ensemble des employés habitant au Royaume-Uni est trié par ordre alphabétique, et de même pour les employés américains.

La requête SQL exécutée ressemble fortement à une requête issue de l'utilisation de l'opérateur `OrderBy`, mais la colonne `FirstName` utilisée à travers l'opérateur `ThenBy` est ajoutée à la clause de tri.

```
SELECT [employee].[EmployeeID], [employee].[Address],
[employee].[BirthDate], [employee].[City], [employee].[Country],
[employee].[Extension], [employee].[FirstName],
[employee].[HireDate], [employee].[HomePhone],
[employee].[LastName], [employee].[Notes], [employee].[Photo],
[employee].[PhotoPath], [employee].[PostalCode],
[employee].[Region], [employee].[ReportsTo], [employee].[Title],
[employee].[TitleOfCourtesy]

FROM [Employees] AS [employee]

ORDER BY [employee].[Country], [employee].[FirstName]
```

Avec la syntaxe intégrée à C#, l'opérateur `ThenBy` n'est pas explicite. Le tri multiple est exprimé dans une clause `orderby`, en précisant la liste des différents champs sur lesquels il doit être effectué.

```
using (var context = new NorthwindContext())
{
    var requete = from employe in context.Employees
                  orderby employe.Country , employe.FirstName
                  select employe;

    //Affichage du pays et du prénom de l'employé
    foreach (var resultat in requete)
    {
        Console.WriteLine(resultat.Country + "-" + resultat.FirstName);
    }
}
```

d. ThenByDescending

L'opérateur `ThenByDescending` effectue le même traitement et a un effet presque identique à l'opérateur `ThenBy`. Il permet d'ajouter un niveau de tri descendant à une séquence préalablement ordonnée.

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

    //Affichage du pays et du prénom des employés
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.Country} - {resultat.FirstName}");
    }
}
```

L'utilisation de `ThenByDescending` en lieu de place de l'opérateur `ThenBy` dans l'exemple précédent donne le résultat suivant :

```
UK - Steven
UK - Robert
UK - Michael
UK - Anne
USA - Nancy
USA - Margaret
USA - Laura
USA - Janet
USA - Andrew
```

Comme `ThenBy`, l'opérateur `ThenByDescending` est intégré au langage C# : on l'utilise implicitement en ajoutant le mot-clé `descending` à la suite de l'un des champs de tri secondaires d'une clause `orderby`.

```
using (var context = new NorthwindContext())
{
    var requete = from employe in context.Employees
                  orderby employe.Country,
                          employe.FirstName descending
                  select employe;

    //Affichage du pays et du prénom de l'employé
    foreach (var resultat in requete)
    {
        Console.WriteLine(resultat.Country + "-" + resultat.FirstName);
    }
}
```

5. Opérateur de regroupement

Le seul opérateur de regroupement défini par l'API LINQ est `GroupBy`. Il permet de créer une séquence contenant des collections de type `IGrouping<TKey, T>Value>`. Chacune d'elles contient un ensemble d'éléments issus de la collection source et partageant une même clé qui correspond généralement à une valeur de propriété. La création des clés est définie par une projection passée en paramètre à l'opérateur.

L'exemple suivant affiche la liste des employés, groupés en fonction de leur pays.

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

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

        //Affichage du nom complet des employés
    }
}
```

```

foreach (var employe in resultat)
{
    Console.WriteLine($"{employee.FirstName} {employee.LastName}");
}
}
}

```

La sortie produite dans la console explicite le comportement de l'opérateur.

```

Clé : UK
    Steven Buchanan
    Michael Suyama
    Robert King
    Anne Dodsworth
Clé : USA
    Laura Callahan
    Nancy Davolio
    Andrew Fuller
    Janet Leverling
    Margaret Peacock

```

Le premier objet de groupement contenu dans la séquence de sortie correspond à l'ensemble des employés dont la propriété Country a pour valeur "UK". Le second est, quant à lui, associé aux employés américains.

```

SELECT [employee].[EmployeeID], [employee].[Address],
[employee].[BirthDate], [employee].[City], [employee].[Country],
[employee].[Extension], [employee].[FirstName], [employee].[HireDate],
[employee].[HomePhone], [employee].[LastName], [employee].[Notes],
[employee].[Photo], [employee].[PhotoPath], [employee].[PostalCode],
[employee].[Region], [employee].[ReportsTo], [employee].[Title],
[employee].[TitleofCourtesy]
FROM [Employees] AS [employee]
ORDER BY [employee].[Country]

```

Le code SQL généré contient une clause `ORDER BY` qui permet d'ordonner les éléments de manière que le moteur de LINQ puisse constituer efficacement les groupes attendus.

L'opérateur GroupBy est également intégré à C# sous la forme d'expressions `group... by`.

```

using (var context = new NorthwindContext())
{
    var requete = from employe in context.Employees
                  group employe by employe.Country;

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

        //Affichage du nom complet des employés
        foreach (var employe in resultat)
        {
            Console.WriteLine($"{employee.FirstName} {employee.LastName}");
        }
    }
}

```



```

    }
}
}

```

Lorsque cet opérateur est utilisé de cette manière au sein d'une requête LINQ, l'ajout d'une clause `select` provoque une erreur de compilation.

```

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

```

Pour pouvoir ajouter une clause `select`, il est nécessaire de créer une variable de destination pour le groupe créé, à l'aide du mot-clé `into`.

```

var requete = from employe in context.Employees
              group employe by employe.Country into groupeEmployes
              select groupeEmployes;

```

6. Opérateurs d'ensembles

Les opérateurs d'ensemble ont pour objectif de produire une séquence à partir d'une collection source en fonction de l'absence ou de la présence d'éléments identiques dans cette collection ou dans une autre.

a. Distinct

L'opérateur `Distinct` retourne une séquence correspondant aux éléments uniques présents dans la collection source en éliminant tout doublon. Cet opérateur est notamment utilisé pour s'assurer de l'unicité des données renvoyées par des requêtes utilisant des jointures, mais il peut aussi s'avérer très utile dans des cas plus simples, par exemple pour récupérer les différentes valeurs qui sont associées à une colonne dans la source de données.

Cet opérateur permet de récupérer très simplement l'ensemble des pays où se situent les entreprises clientes.

```

using (var context = new NorthwindContext())
{
    var requete = context.Customers
                    .Select(customer => customer.Country)
                    .Distinct();

    //Affichage des pays
    foreach (var country in requete)
    {
        Console.WriteLine(country);
    }
}

```

Le résultat obtenu permet d'affirmer que les 91 clients sont répartis dans 21 pays.

```

Argentina
Austria
Belgium
Brazil
Canada

```

Denmark
Finland
France
Germany
Ireland
Italy
Mexico
Norway
Poland
Portugal
Spain
Sweden
Switzerland
UK
USA
Venezuela

Le code SQL produit est également très simple, puisqu'il tire parti du mot-clé `DISTINCT` géré par les bases de données relationnelles.

```
SELECT DISTINCT [customer].[Country]
FROM [Customers] AS [customer]
```

b. Except

Cet opérateur produit une séquence contenant les éléments de la collection source qui ne sont pas présents dans la collection qui lui est fournie en paramètre.

Pour déterminer la présence d'un élément dans les deux collections, l'opérateur `Except` utilise en premier lieu la méthode `IEquatable<T>.Equals` si les objets de la collection source implémentent cette interface. À défaut, il appelle la méthode `Equals` héritée du type `Object`, ou une surcharge, pour vérifier l'égalité d'éléments.

L'exemple ci-après affiche, dans cet ordre :

- La liste complète des employés.
- Les noms des deux premiers employés, à l'aide de l'opérateur `Take`.
- La liste complète des employés amputée des deux premiers, avec l'opérateur `Except`.

```
using (var context = new NorthwindContext())
{
    var tousEmployes = context.Employees;

    Console.WriteLine("Tous les employés : ");
    foreach (var employe in tousEmployes)
    {
        Console.WriteLine($"{employe.FirstName} {employe.LastName}");
    }

    var deuxPremiersEmployes = context.Employees
        .Take(2);

    Console.WriteLine();
    Console.WriteLine("Deux premiers employés : ");
    foreach (var employe in deuxPremiersEmployes)
```

```

{
    Console.WriteLine($"{employee.FirstName} {employee.LastName}");
}

var requete = context.Employees
    .Except(deuxPremiersEmployes);
Console.WriteLine();
Console.WriteLine("Tous les employés sauf les deux premiers : ");
foreach (var employe in requete)
{
    Console.WriteLine($"{employee.FirstName} {employee.LastName}");
}
}

```

Le résultat affiché est le suivant :

```

Tous les employés :
Nancy Davolio
Andrew Fuller
Janet Leverling
Margaret Peacock
Steven Buchanan
Michael Suyama
Robert King
Laura Callahan
Anne Dodsworth

Deux premiers employés :
Nancy Davolio
Andrew Fuller

Tous les employés sauf les deux premiers :
Janet Leverling
Margaret Peacock
Steven Buchanan
Michael Suyama
Robert King
Laura Callahan
Anne Dodsworth

```

L'analyse du code SQL généré pour l'opérateur `Except` est relativement simple : l'instruction `SELECT` récupère l'ensemble des employés. Le traitement effectué par cet opérateur est en effet à ce jour entièrement exécuté en mémoire.

```

SELECT [e].[EmployeeID], [e].[Address], [e].[BirthDate], [e].[City],
[e].[Country], [e].[Extension], [e].[FirstName], [e].[HireDate],
[e].[HomePhone], [e].[LastName], [e].[Notes], [e].[Photo],
[e].[PhotoPath], [e].[PostalCode], [e].[Region], [e].[ReportsTo],
[e].[Title], [e].[TitleOfCourtesy]
FROM [Employees] AS [e]

```

c. Union

La combinaison de deux séquences d'objets de même type peut être effectuée à l'aide de l'opérateur `Union`. La collection retournée contient l'ensemble des éléments présents dans les deux séquences sources, sans doublon.

La portion de code suivante montre l'effet de cet opérateur en affichant, dans cet ordre :

- Les trois premiers employés à l'aide de l'opérateur `Take`.
- Les noms des employés placés en troisième, quatrième et cinquième position dans la table en utilisant une combinaison des opérateurs `Skip` et `Take`.
- Les cinq premiers employés, en combinant les deux listes précédentes avec l'opérateur `Union`.

```
using (var context = new NorthwindContext())
{
    var employesUnDeuxEtTrois = context.Employees.Take(3);

    Console.WriteLine("Employés n° 1, 2 et 3 : ");
    foreach (var employe in employesUnDeuxEtTrois)
    {
        Console.WriteLine($"{employe.FirstName} {employe.LastName}");
    }

    Console.WriteLine();

    var employesTroisQuatreEtCinq = context.Employees.Skip(2).Take(3);

    Console.WriteLine("Employés n° 3, 4 et 5 : ");
    foreach (var employe in employesTroisQuatreEtCinq)
    {
        Console.WriteLine($"{employe.FirstName} {employe.LastName}");
    }

    Console.WriteLine();
    Console.WriteLine("Union des deux ensembles :");
    var requete = employesUnDeuxEtTrois.Union(employesTroisQuatreEtCinq);

    foreach (var employe in requete)
    {
        Console.WriteLine($"{employe.FirstName} {employe.LastName}");
    }
}
```

Le résultat affiché à l'écran est le suivant :

```
Employés n° 1, 2 et 3 :
Nancy Davolio
Andrew Fuller
Janet Leverling

Employés n° 3, 4 et 5 :
Janet Leverling
Margaret Peacock
```

Steven Buchanan

Union des deux ensembles :

Nancy Davolio

Andrew Fuller

Janet Leverling

Margaret Peacock

Steven Buchanan

Comme `Except`, l'opérateur `Union` détermine l'existence de doublons en utilisant la méthode `Equals` de l'interface `IEquatable<T>` si le type des éléments implémente cette interface, ou, à défaut, la fonction `Equals`, héritée du type `Object`.

d. Intersect

L'intersection de deux ensembles de données correspond à la liste des éléments présents aussi bien dans le premier ensemble que dans le second. Le calcul de l'intersection de deux jeux d'enregistrements peut être réalisé en utilisant l'opérateur `Intersect`.

Comme pour `Except` et `Union`, les comparaisons effectuées sont basées sur l'utilisation de la méthode `IEquatable<T>.Equals` lorsque les objets de la collection source implémentent l'interface `IEquatable<T>`. Si cette implémentation n'existe pas, `Except` fait son œuvre en tirant parti de la méthode `Equals` que tous les objets .NET possèdent.

Les collections `employesUnDeuxEtTrois` et `employesTroisQuatreEtCinq`, définies au début de l'exemple de code qui suit, correspondent respectivement aux trois premiers employés listés dans la table `Employees` et aux employés placés aux positions 3, 4 et 5 de cette même table.

L'utilisation de l'opérateur `Intersect` sur ces deux collections renvoie une collection contenant un unique élément : le troisième employé.

```
using (var context = new NorthwindContext())
{
    var employesUnDeuxEtTrois = context.Employees.Take(3);

    Console.WriteLine("Employés n° 1, 2 et 3 : ");
    foreach (var employe in employesUnDeuxEtTrois)
    {
        Console.WriteLine($"{employe.FirstName} {employe.LastName}");
    }

    Console.WriteLine();

    var employesTroisQuatreEtCinq = context.Employees.Skip(2).Take(3);

    Console.WriteLine("Employés n° 3, 4 et 5 : ");
    foreach (var employe in employesTroisQuatreEtCinq)
    {
        Console.WriteLine($"{employe.FirstName} {employe.LastName}");
    }

    Console.WriteLine();
    Console.WriteLine("Intersection des deux ensembles :");
    var requete = employesUnDeuxEtTrois.Intersect(employesTroisQuatreEtCinq);
```

```

foreach (var employe in requete)
{
    Console.WriteLine($"{employe.FirstName} {employe.LastName}");
}
}

```

Le résultat de l'exécution est le suivant :

```

Employés n° 1, 2 et 3 :
Nancy Davolio
Andrew Fuller
Janet Leverling

Employés n° 3, 4 et 5 :
Janet Leverling
Margaret Peacock
Steven Buchanan

Intersection des deux ensembles :
Janet Leverling

```

7. Opérateur de filtrage

Un seul opérateur de filtrage est implémenté par LINQ : `where`. Les enregistrements renvoyés par cette fonction sont choisis sur la base de l'évaluation d'un prédicat qui doit lui être passé en paramètre. Le premier exemple de l'utilisation de cet opérateur recherche l'ensemble des clients français enregistrés dans la base de données.

```

using (var context = new NorthwindContext())
{
    var clientsFrancais
        = context.Customers
            .Where(client => client.Country == "France");

    Console.WriteLine("Liste des clients français");
    foreach (var client in clientsFrancais)
    {
        Console.WriteLine($"{client.CompanyName} - {client.City}");
    }
}

```

Le nombre d'entreprises retournées dans ce cas est de 11.

Blondesdds1 père et fils - Strasbourg
Bon app' - Marseille
Du monde entier - Nantes
Folies gourmandes - Lille
France restauration - Nantes
La corne d'abondance - Versailles
La maison d'Asie - Toulouse
Paris spécialités - Paris
Spécialités du monde - Paris
Victuailles en stock - Lyon
Vins et alcools Chevalier - Reims

La requête SQL générée par Entity Framework et exécutée par la source de données comporte une clause `WHERE` correspondant au prédicat défini plus haut.

```
SELECT [client].[CustomerID], [client].[Address], [client].[City],  
[client].[CompanyName], [client].[ContactName],  
[client].[ContactTitle], [client].[Country], [client].[Fax],  
[client].[Phone], [client].[PostalCode], [client].[Region]  
FROM [Customers] AS [client]  
WHERE [client].[Country] = N'France'
```

Le caractère `N` préfixant la chaîne de caractères dans la comparaison indique que l'on souhaite utiliser Unicode pour son encodage. Sans cet indicateur, le moteur de base de données utilise l'encodage non-Unicode du serveur pour évaluer le résultat du prédicat. Le type `string` correspondant à des chaînes de caractères Unicode, il est tout à fait cohérent que le SQL généré soit compatible avec cet encodage.

La clause générée peut bien évidemment être complexifiée par l'écriture d'un prédicat incluant plusieurs comparaisons ou par le chaînage de plusieurs instructions de filtrage.

```
using (var context = new NorthwindContext())  
{  
    var clientsLyonnais1  
        = context.Customers  
            .where(client => client.Country == "France"  
                && client.City == "Lyon");  
  
    Console.WriteLine("Liste des clients lyonnais");  
    foreach (var client in clientsLyonnais1)  
    {  
        Console.WriteLine($"{client.CompanyName} - {client.City}");  
    }  
  
    var clientsLyonnais2  
        = context.Customers  
            .where(client => client.Country == "France")  
            .where(client => client.City == "Lyon");  
  
    Console.WriteLine("Liste des clients lyonnais");  
    foreach (var client in clientsLyonnais2)  
    {  
        Console.WriteLine($"{client.CompanyName} - {client.City}");  
    }  
}
```

```
}  
}
```

Dans les deux cas, le code SQL généré ainsi que le résultat de son exécution sont strictement identiques.

victuailles en stock - Lyon

```
SELECT [client].[CustomerID], [client].[Address], [client].[City],  
[client].[CompanyName], [client].[ContactName],  
[client].[ContactTitle], [client].[Country], [client].[Fax],  
[client].[Phone], [client].[PostalCode], [client].[Region]  
FROM [Customers] AS [client]  
WHERE ([client].[Country] = N'France') AND ([client].[City] = N'Lyon')
```

Le fournisseur de données est ici capable d'interpréter l'intention du développeur et utilise la combinaison de prédicats dans le cadre de la clause `WHERE` pour transcrire la requête LINQ en SQL. Ce cas n'est pas isolé : les fournisseurs de données relationnels sont en mesure de transformer des expressions plus complexes en SQL, notamment concernant la comparaison de chaînes de caractères avec jokers.

```
using (var context = new NorthwindContext())  
{  
    var clients  
        = context.Customers  
            .Where(client => client.Country == "France"  
                && client.City.StartsWith("L")  
                && client.City.Contains("yo")  
                && client.City.EndsWith("n"));  
  
    Console.WriteLine("Liste de clients");  
    foreach (var client in clients)  
    {  
        Console.WriteLine($"{client.CompanyName} - {client.City}");  
    }  
}
```

Bien évidemment, un seul client satisfait à tous ces critères : celui qui est basé à Lyon et que nous avons déjà trouvé dans l'exemple précédent. Ici, ce n'est pas tant le résultat qui est intéressant que la manière de l'obtenir. Les fonctions `Startswith`, `Contains` et `Endswith` trouvent toutes leur équivalent en SQL par l'utilisation de l'opérateur `LIKE` et de jokers (le caractère `%` en SQL). Les fournisseurs relationnels tirent parti de cette capacité en générant un code, certes plus complexe, mais ayant l'avantage de déporter les calculs vers la source de données.


```

SELECT [client].[CustomerID], [client].[Address], [client].[City],
[client].[CompanyName], [client].[ContactName],
[client].[ContactTitle], [client].[Country], [client].[Fax],
[client].[Phone], [client].[PostalCode], [client].[Region]
FROM [Customers] AS [client]
WHERE ((([client].[Country] = N'France')
    AND [client].[City] LIKE N'L' + N'%')
    AND [client].[City] LIKE (N%' + N'o') + N'%')
    AND [client].[City] LIKE N%' + N'n'

```

La fonction `where` fait également partie de la liste des opérateurs directement intégrés à C# au travers du mot-clé `where`.

```

using (var context = new NorthwindContext())
{
    var clients = from client in context.Customers
        where client.Country == "France"
            && client.City == "Lyon"
        select client;

    Console.WriteLine("Liste de clients lyonnais");
    foreach (var client in clients)
    {
        Console.WriteLine($"{client.CompanyName} - {client.City}");
    }
}

```

8. Opérateurs d'agrégation

Les opérateurs d'agrégation ont pour objectif de calculer une valeur unique à partir d'une séquence source.

a. Count

L'opérateur `Count` se présente sous la forme d'une fonction dont la valeur de retour, correspondant au nombre d'éléments comptés, est de type `int`. Deux variantes de cet opérateur sont disponibles. La première n'attend aucun paramètre et renvoie le nombre total d'éléments présents dans la collection. Le code ci-après présente l'implémentation d'une opération de comptage des clients enregistrés.

```

using (var context = new NorthwindContext())
{
    var nombreClients = context.Customers.Count();

    Console.WriteLine($"Nombre total de clients : {nombreClients}");
}

```

Nombre total de clients : 91

L'implémentation paramétrable de cet opérateur accepte, quant à elle, qu'un prédicat lui soit fourni, de manière à n'effectuer le calcul qu'en se basant sur les éléments respectant cette condition. D'un point de vue technique, le paramètre attendu est une fonction qui a en entrée un objet de la collection source et doit renvoyer une valeur booléenne. Ici, on souhaite dénombrer les clients habitant à Londres.

```
using (var context = new NorthwindContext())
{
    var nombreClientsLondres
        = context.Customers
            .Count(client => client.City == "London");

    Console.WriteLine($"Nombre total de clients londoniens :
{nombreClientsLondres}");
}
```

Nombre total de clients londoniens :6

Les deux formes de cet opérateur tirent parti, dans le cas des fournisseurs relationnels, de la construction `COUNT()` intégrée au langage SQL. Il est toutefois important de noter que dans le second cas, à savoir le comptage des clients londoniens, Entity Framework est capable de transformer le prédicat en clause `WHERE`, de manière à conserver un maximum de calcul au niveau de la base de données.

```
SELECT COUNT(*)
FROM [Customers] AS [c]
```

```
SELECT COUNT(*)
FROM [Customers] AS [client]
WHERE [client].[City] = N'London'
```

b. LongCount

L'opérateur `LongCount` est presque exactement identique à l'opérateur `Count`. La différence entre les deux réside dans le type de données renvoyé : le type de retour de `LongCount` est `long` (ou `Int64`, donc sur 64 bits), alors que `Count` ne renvoie qu'une valeur de type `int` (ou `Int32`). `LongCount` permet par conséquent de dénombrer beaucoup plus d'éléments.

Cet opérateur peut lui aussi tirer parti des capacités de la source de données, et notamment de SQL Server. Ce moteur de base de données intègre une construction particulière destinée à effectuer le comptage d'un grand nombre d'enregistrements : `COUNT_BIG()`.

```

using (var context = new NorthwindContext())
{
    var nombreClients = context.Customers
                            .LongCount();

    Console.WriteLine($"Nombre total de clients : {nombreClients}");

    var nombreClientsLondres
        = context.Customers
            .LongCount(client => client.City == "London");

    Console.WriteLine($"Nombre de clients londoniens : {nombreClientsLondres}");
}

```

Le code SQL généré pour ces deux requêtes ne diffère du code généré pour l'opérateur `Count` que par le mot-clé utilisé par le moteur SQL Server pour effectuer l'opération.

```

SELECT COUNT_BIG(*)
FROM [Customers] AS [c]

```

```

SELECT COUNT_BIG(*)
FROM [Customers] AS [client]
WHERE [client].[City] = N'London'

```

c. Min

L'emploi de l'opérateur `Min` est associé à la recherche d'une valeur minimale dans un ensemble de données. Dans sa forme la plus simple, la fonction `Min` n'attend aucun paramètre : elle travaille directement sur les données de la séquence source. Pour cette raison, avec Entity Framework Core, les cas d'utilisation de cette implémentation incluent généralement une opération de projection préalable.

Une surcharge de cet opérateur offre la possibilité d'intégrer cette projection en tant que paramètre, de manière à pouvoir sélectionner l'élément sur lequel les comparaisons doivent être effectuées.

Le code suivant utilise les deux versions de l'opérateur `Min` pour parcourir la table `Products` à la recherche du prix le moins élevé, et affiche dans les deux cas un montant de 2,50 dollars.

```

using (var context = new NorthwindContext())
{
    var prixLePlusBas1
        = context.Products
            .Select(produit => produit.UnitPrice)
            .Min();

    Console.WriteLine($"Prix le plus bas : {prixLePlusBas1}");

    var prixLePlusBas2
        = context.Products
            .Min(produit => produit.UnitPrice);
}

```

```
Console.WriteLine($"Prix le plus bas : {prixLePlusBas2}");
}
```

Le code généré montre qu'Entity Framework Core transcrit l'opérateur `Min` en utilisant le mot-clé SQL `MIN`, et ce pour les deux requêtes décrites ici.

```
SELECT MIN([produit].[UnitPrice])
FROM [Products] AS [produit]
```

Le cas général d'utilisation pour cet opérateur implique des données numériques, mais il est tout à fait capable de gérer des types différents, voire inattendus. On peut par exemple l'utiliser pour trouver le premier nom de produit dans l'ordre alphabétique.

```
using (var context = new NorthwindContext())
{
    var premierNomProduitOrdreAlpha = context.Products
        .Min(produit => produit.ProductName);
    Console.WriteLine($"Le premier nom de produit dans l'ordre
        alphabétique est {premierNomProduitOrdreAlpha}");
}
```

Le premier nom de produit dans l'ordre alphabétique est Alice Mutton

La fonction `Min` autorise également des projections renvoyant des objets complexes qui implémentent l'interface `IComparable`, tout comme l'opérateur `OrderBy`. On peut donc réutiliser le type `ClientComparable` écrit plus tôt pour récupérer le client dont la somme des longueurs de ses noms de contact et d'entreprise est la plus petite.

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

    Console.WriteLine($"{requete.ContactName} - {requete.CompanyName}");
}
```

Le résultat affiché correspond bien aux résultats obtenus avec l'utilisation conjointe de l'opérateur `OrderBy` et du type `ClientComparable`.

Horst Kloss - QUICK-Stop

En revanche, dans ce cas, le SQL généré correspond à une simple projection puisque la logique de comparaison ne peut pas être transcrite dans un format compréhensible par la source de données. Le calcul est donc effectué par l'application, et non par la base de données.

```
SELECT [client].[ContactName], [client].[CompanyName]
FROM [Customers] AS [client]
```

d. Max

Comme on peut s'y attendre, l'opérateur `Max` a beaucoup de points communs avec l'opérateur `Min`. Ils opèrent en effet tous deux de la même manière, à la différence que l'opérateur `Max` renvoie la valeur maximale du résultat d'une projection de données. La recherche du montant le plus élevé est donc quasiment identique à celle effectuée avec l'opérateur `Min`.

```
using (var context = new NorthwindContext())
{
    var prixLePlusCher1
        = context.Products
            .Select(produit => produit.UnitPrice)
            .Max();

    Console.WriteLine($"Prix le plus élevé : {prixLePlusCher1}");

    var prixLePlusCher2
        = context.Products
            .Max(produit => produit.UnitPrice);

    Console.WriteLine($"Prix le plus élevé : {prixLePlusCher2}");
}
```

Prix le plus élevé : 263,5000

Le code SQL généré indique clairement que l'opérateur `Max` est transformé de manière à être supporté nativement par la source de données, avec le mot-clé `MAX`.

```
SELECT MAX([produit].[UnitPrice])
FROM [Products] AS [produit]
```

De la même manière que l'opérateur `Min`, `Max` peut être utilisé sur des données non numériques, telles que des chaînes de caractères, des dates ou des objets complexes implémentant l'interface `IComparable`. Ce dernier cas est toutefois à part, puisque ces types ne sont pas supportés nativement par la source de données. La logique de comparaison n'est donc pas déportée et est exécutée par l'application elle-même.

e. Sum

Le calcul de somme sur une collection d'enregistrements est implémenté par l'opérateur `Sum`. À l'instar des opérateurs `Min` et `Max`, il peut être utilisé de deux façons :

- Sans paramètre : il opère alors directement sur les données présentes dans la collection source, généralement préparées à l'aide d'une projection.
- Avec un paramètre : la projection permettant de sélectionner les données qui doivent être prises en compte pour le calcul est intégrée à l'appel de la fonction.

L'exemple ci-après calcule, à l'aide de cet opérateur, le montant total des frais de port facturés pour les commandes passées par les clients.

```
using (var context = new NorthwindContext())
{
    var totalFrais1 = context.Orders
        .Select(commande => commande.Freight)
        .Sum();

    Console.WriteLine($"Montant total des frais de port : {totalFrais1}");

    var totalFrais2 = context.Orders
        .Sum(commande => commande.Freight);

    Console.WriteLine($"Montant total des frais de port : {totalFrais2}");
}
```

Le résultat de l'exécution de ces deux requêtes est identique :

```
Montant total des frais de port : 64942,6900
```

Les bases de données relationnelles supportent nativement la construction SQL `SUM` utilisée pour transformer cet opérateur.

```
SELECT SUM([commande].[Freight])
FROM [Orders] AS [commande]
```

À l'inverse des opérateurs `Min` et `Max`, `Sum` ne supporte que des valeurs numériques : `int`, `long`, `double`, `float`, `decimal`, ainsi que le type `Nullable<T>`, où `T` est un des types numériques cités précédemment. Il peut ainsi renvoyer indifféremment une valeur numérique ou une valeur égale à `null`.

f. Average

`Average` est un opérateur comparable à `Sum` dans son utilisation, mais il effectue pour sa part un calcul légèrement plus complexe. Il produit en effet la valeur moyenne des données numériques contenues dans la séquence source sur laquelle il agit.

Tout comme la fonction `Sum`, `Average` est disponible sous deux formes : sans paramètre et avec un paramètre permettant d'effectuer une projection des données sources vers une forme sur laquelle il peut travailler. Les types supportés par cet opérateur sont ici encore `int`, `long`, `double`, `float`, `decimal` ainsi que `Nullable<T>`, où `T` est un des types numériques cités précédemment.

Le code ci-après calcule le prix moyen des produits enregistrés dans la base de données.

```
using (var context = new NorthwindContext())
{
    var prixMoyenProduits1
        = context.Products
            .Select(produit => produit.UnitPrice)
            .Average();
}
```

```

Console.WriteLine($"Prix moyen des produits : {prixMoyenProduits1}");

var prixMoyenProduits2
    = context.Products
        .Average(produit => produit.UnitPrice);

Console.WriteLine($"Prix moyen des produits : {prixMoyenProduits2}");
}

```

Ce qui produit à l'écran l'affichage suivant :

```
Prix moyen des produits : 28,86636363636363636363636363636
```

Le langage SQL intègre nativement une fonction nommée `AVG` capable d'effectuer un calcul de valeur moyenne. Malheureusement, il n'est pas pris en compte à ce jour par les fournisseurs de données pour déporter le calcul vers la source de données. Il est donc entièrement exécuté par l'application, après la récupération des données nécessaires. Le code SQL effectue uniquement cette dernière opération :

```

SELECT [produit].[UnitPrice]
FROM [Products] AS [produit]

```

9. Opérateurs de jointure

La jointure de deux collections correspond à l'association de ces deux ensembles par le biais d'une propriété commune aux types d'objets qu'elles contiennent. Sans l'utilisation de jointures, il serait nécessaire d'effectuer cette opération manuellement, en parcourant une collection à la recherche d'un objet correspondant à chacun des éléments de l'autre séquence. LINQ implémente deux opérateurs permettant de mettre en place ces jointures : `Join` et `GroupJoin`.

a. Join

L'opérateur `Join` associe deux collections sur la base de deux projections qui permettent d'identifier les points de jointure à utiliser. Le résultat de cette jointure est constitué à partir d'une troisième opération de projection dont les arguments sont une paire d'éléments provenant des deux collections sources et marqués comme correspondants.

L'exemple de code suivant effectue une jointure entre les collections `Orders` et `Employees` afin de retourner le numéro de chaque commande ainsi que le nom de l'employé qui lui est associé.

```

using (var context = new NorthwindContext())
{
    var commandesEtCommerciaux
        = context.Orders
            .Join(context.Employees,
                commande => commande.EmployeeId,
                employe => employe.EmployeeId,
                (commande, employe) => new
                {
                    NumCommande = commande.OrderId,
                    NomEmploye = employe.FirstName + " "
                        + employe.LastName
                });
}

```

```

foreach (var result in commandesEtCommerciaux)
{
    Console.WriteLine($"Commande n° {result.NumCommande}
                        enregistrée par {result.NomEmploye}");
}

```

Le résultat affiché par l'exécution de cette portion de code, volontairement raccourci ici (il y a 830 enregistrements dans la table `orders`, et autant de lignes affichées), est le suivant :

```

Commande n° 10258 enregistrée par Nancy Davolio
Commande n° 10270 enregistrée par Nancy Davolio
Commande n° 10275 enregistrée par Nancy Davolio
Commande n° 10285 enregistrée par Nancy Davolio
...
Commande n° 10265 enregistrée par Andrew Fuller
Commande n° 10277 enregistrée par Andrew Fuller
Commande n° 10280 enregistrée par Andrew Fuller
Commande n° 10295 enregistrée par Andrew Fuller
Commande n° 10300 enregistrée par Andrew Fuller
Commande n° 10307 enregistrée par Andrew Fuller
...
Commande n° 10970 enregistrée par Anne Dodsworth
Commande n° 10978 enregistrée par Anne Dodsworth
Commande n° 11016 enregistrée par Anne Dodsworth
Commande n° 11017 enregistrée par Anne Dodsworth
Commande n° 11022 enregistrée par Anne Dodsworth
Commande n° 11058 enregistrée par Anne Dodsworth

```

L'analyse de la requête SQL exécutée pour obtenir ce résultat correspond à ce que l'on pouvait attendre : l'opérateur `JOIN` du langage SQL est utilisé pour transcrire l'appel à la méthode `Join` de LINQ. Plus précisément ici, il s'agit d'une jointure de type `INNER JOIN`.

```

SELECT [commande].[OrderID], ([employee].[FirstName] + N' ') +
[employee].[LastName]
FROM [Orders] AS [commande]
INNER JOIN [Employees] AS [employee] ON [commande].[EmployeeID] =
[employee].[EmployeeID]

```

L'utilisation d'une clause `INNER JOIN` retranscrit des résultats les éléments de la collection source lorsque aucune correspondance n'est trouvée dans la seconde séquence. Il est possible de gérer ce type de problématique à l'aide de l'opérateur `GroupJoin`.

L'opérateur `Join` est un des opérateurs ayant une contrepartie intégrée au langage C#. Sans surprise, la construction qui lui est associée est `join... on....`. L'exemple ci-après reprend la requête précédente en utilisant cette syntaxe.

```

using (var context = new NorthwindContext())
{
    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
};

foreach (var result in commandesEtCommerciaux)
{
    Console.WriteLine($"Commande n° {result.NumCommande}
                      enregistrée par {result.NomEmploye}");
}
}

```

Tout d'abord, on remarque que le mot-clé `join` est suivi d'une définition de collection semblable à ce que l'on peut trouver lors de l'écriture d'une clause `from`. Elle définit une variable représentant un enregistrement en cours de traitement à l'aide de la syntaxe `<variable> in <collection>`.

Le point notable suivant est situé dans l'opération de comparaison effectuée entre les deux points de jointure. L'opérateur utilisé pour le test d'égalité n'est pas `==`, comme on pourrait s'y attendre, mais `equals`, qui est un mot-clé spécial utilisé uniquement dans le cadre d'une jointure.

Enfin, la projection utilisée pour la création du résultat de l'opération est déportée vers la clause `select` de la requête.

b. GroupJoin

L'opérateur `Join` peut rapidement atteindre ses limites, notamment lorsqu'il est question de jointures pour lesquelles un élément de la collection source correspond à plusieurs éléments de la seconde séquence. La fonction `GroupJoin` est alors à privilégier car celle-ci est conçue pour répondre à cette problématique.

```

using (var context = new NorthwindContext())
{
    var commerciauxEtCommandes
        = 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)
                });

    foreach (var result in commerciauxEtCommandes)
    {
        Console.WriteLine($"Employé : {result.NomEmploye}");

        foreach (var numCommande in result.NumCommandes)

```

```

    {
        Console.WriteLine($"{\tCommande n° {numCommande}");
    }
}
}

```

Le résultat renvoyé par l'opérateur `GroupJoin` est ici une collection d'objets ayant une propriété `NomEmploye` de type `string` et une propriété `NumCommandes` qui est une collection d'entiers. Les résultats ont donc été factorisés de manière à n'avoir qu'une seule fois le nom de l'employé, et non une fois par enregistrement comme avec la méthode `Join`.

Le code SQL associé à cette requête est le suivant :

```

SELECT [employee].[EmployeeID], [employee].[Address], [employee].[BirthDate],
[employee].[City], [employee].[Country], [employee].[Extension],
[employee].[FirstName], [employee].[HireDate], [employee].[HomePhone],
[employee].[LastName], [employee].[Notes], [employee].[Photo],
[employee].[PhotoPath], [employee].[PostalCode], [employee].[Region],
[employee].[ReportsTo], [employee].[Title], [employee].[TitleOfCourtesy],
[commande].[OrderID], [commande].[CustomerID], [commande].[EmployeeID],
[commande].[Freight], [commande].[OrderDate], [commande].[RequiredDate],
[commande].[ShipAddress], [commande].[ShipCity], [commande].[ShipCountry],
[commande].[ShipName], [commande].[ShipPostalCode], [commande].[ShipRegion],
[commande].[ShipVia], [commande].[ShippedDate], ([employee].[FirstName] +
N' ') + [employee].[LastName]
FROM [Employees] AS [employee]
LEFT JOIN [Orders] AS [commande] ON [employee].[EmployeeID] =
[commande].[EmployeeID]
ORDER BY [employee].[EmployeeID]

```

La liste des champs intégrés à la clause `SELECT` est plutôt impressionnante en regard des données que l'on souhaite manipuler in fine. La raison de ce comportement est simple : le groupement est une opération effectuée en mémoire par l'application. Entity Framework, au stade de la génération de code SQL, ne sait pas quels sont les champs qui seront nécessaires pour la suite des opérations, et ce malgré l'utilisation de l'opérateur `SELECT` pour la génération de la collection `NumCommandes`. Il lui est donc nécessaire de récupérer toutes les informations qui pourront être utilisées ultérieurement.

La clause `INNER JOIN` générée lors de l'appel à la méthode `Join` est ici remplacée par une clause `LEFT JOIN` de manière à toujours retourner les enregistrements correspondant à la collection source. La collection de résultats correspondant à la seconde séquence est laissée vide lorsque aucun élément associé n'a été trouvé.

Le résultat affiché dans la console (sensiblement écourté encore une fois) est représenté ci-après.

```

Employé : Nancy Davolio
    Commande n° 10258
    Commande n° 10270
    Commande n° 10275
    Commande n° 10285
    Commande n° 10292
    Commande n° 10293
    ...
Employé : Andrew Fuller

```

```

    Commande n° 11073
    Commande n° 11070
    Commande n° 11059
    Commande n° 11060
    Commande n° 11053
    ...
    Employé : Anne Dodsworth
    ...
    Commande n° 10978
    Commande n° 11022
    Commande n° 11016
    Commande n° 11017
    Commande n° 11058

```

La syntaxe utilisant l'extension C# de LINQ intègre également cet opérateur sous la forme d'un ajout à la clause `join... on...`. Le groupement est exprimé en ajoutant le mot-clé `into` suivi d'un identifiant de variable après la définition de jointure. La variable créée est de type `IEnumerable<T>`, où `T` est le type d'éléments de la séquence jointe et est utilisable dans la suite de la requête.

```

using (var context = new NorthwindContext())
{
    var commerciauxEtCommandes
        = 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)
          };

    foreach (var result in commerciauxEtCommandes)
    {
        Console.WriteLine($"Employé : {result.NomEmploye}");

        foreach (var numCommande in result.NumCommandes)
        {
            Console.WriteLine($"  \tCommande n° {numCommande}");
        }
    }
}

```

10. Opérateurs de conversion

LINQ fournit différents opérateurs dont l'objectif est la manipulation du type de données renvoyé par une requête. Ces opérateurs peuvent être divisés en deux catégories : les opérateurs qui manipulent le type des enregistrements issus de la source de données, et les opérateurs qui opèrent sur le type de la collection qui englobe les résultats. Le premier de ces groupes est constitué des opérateurs `OfType` et `Cast`.

a. Manipulation du type des enregistrements

Ces deux opérateurs sont traités simultanément car ils permettent tous deux de transformer les enregistrements issus d'une requête d'un type vers un autre. Leurs modes de fonctionnement sont néanmoins différents.

Leur explication nécessite également l'utilisation d'un modèle de données différent : ils ne prennent leur intérêt que lorsqu'une relation d'héritage est présente dans le modèle. Ici, nous étudierons donc le cas d'une relation d'héritage entre un type nommé `Personne` et un second type dont le nom est `Client`. La relation est évidente : un client est une personne.

La relation d'héritage est traitée en détail au chapitre Modélisation avancée de cet ouvrage.

Modèle de données

Pour des raisons de simplicité, la source de données utilisée ici est SQLite. Il convient donc d'installer le package NuGet Microsoft.EntityFrameworkCore.Sqlite pour reproduire les exemples décrits dans cette section.

```
public class SqliteContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite(@"Data source=mydb.db");
    }

    public DbSet<Personne> Personnes { get; set; }

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

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; }
}
```

La base de données est initialisée dans le répertoire de l'application et valorisée à l'aide de la portion de code ci-après.

```
using (var context = new SqliteContext())
{
    context.Database.EnsureCreated();

    //Si aucun enregistrement n'existe dans la table Personnes
    if (context.Personnes.Count() == 0)
```

```

{
    var personne1 = new Personne { Nom = "NomPersonne1",
Prenom = "PrenomPersonne1" };
    var personne2 = new Personne { Nom = "NomPersonne2",
Prenom = "PrenomPersonne2" };
    var personne3 = new Personne { Nom = "NomPersonne3",
Prenom = "PrenomPersonne3" };
    var personne4 = new Personne { Nom = "NomPersonne4",
Prenom = "PrenomPersonne4" };

    context.Add(personne1);
    context.Add(personne2);
    context.Add(personne3);
    context.Add(personne4);
}

//Si aucun enregistrement n'existe dans la table Clients
if (context.Clients.Count() == 0)
{
    var client1 = new Client { Nom = "NomClient1",
Prenom = "PrenomClient1", MontantTotalCommandes = 101 };
    var client2 = new Client { Nom = "NomClient2",
Prenom = "PrenomClient2", MontantTotalCommandes = 102 };
    var client3 = new Client { Nom = "NomClient3",
Prenom = "PrenomClient3", MontantTotalCommandes = 103 };
    var client4 = new Client { Nom = "NomClient4",
Prenom = "PrenomClient4", MontantTotalCommandes = 104 };

    context.Add(client1);
    context.Add(client2);
    context.Add(client3);
    context.Add(client4);
}

context.SaveChanges();
}

```

Maintenant que la source de données est initialisée et utilisable, vérifions le contenu des tables associées aux deux objets `DbSet` du modèle.

```

using (var context = new SQLiteContext())
{
    Console.WriteLine("-----");
    Console.WriteLine("Personnes");
    Console.WriteLine("-----");
    foreach (var personne in context.Personnes)
    {
        Console.WriteLine($"{personne.Nom} {personne.Prenom}");
    }

    Console.WriteLine("-----");
    Console.WriteLine("Clients");
    Console.WriteLine("-----");

    foreach (var client in context.Clients)

```

```

{
    Console.WriteLine($"{client.Nom} {client.Prenom}");
}
}

```

Le résultat obtenu à l'écran est le suivant :

```

-----
Personnes
-----
NomPersonne1 PrenomPersonne1
NomPersonne2 PrenomPersonne2
NomPersonne3 PrenomPersonne3
NomPersonne4 PrenomPersonne4
NomClient1 PrenomClient1
NomClient2 PrenomClient2
NomClient3 PrenomClient3
NomClient4 PrenomClient4
-----
Clients
-----
NomClient1 PrenomClient1
NomClient2 PrenomClient2
NomClient3 PrenomClient3
NomClient4 PrenomClient4

```

On remarque d'emblée que la collection `Personnes` renvoie les objets de type `Personne` que nous avons créés plus tôt, ainsi que l'ensemble des clients enregistrés. C'est ce comportement qui ouvre la porte à l'utilisation des opérateurs concernés par cette section.

OfType

L'opérateur `OfType` peut être considéré comme un opérateur de filtrage, au même titre que `where`. Il permet de récupérer, à partir d'une collection source, l'ensemble des éléments dont le type est ou dérive de son type paramètre. Dans le cas décrit ici, son utilisation peut exécuter le filtrage par type sur la collection `Personnes` et ainsi ne renvoyer que la liste des clients sous la forme d'une collection de type `IQueryable<Client>`.

Pour rappel, chaque élément de code générique attend un ou plusieurs types paramètres lors de son utilisation. Ces types sont indiqués entre deux chevrons à la suite du nom de la fonction ou du type générique. Les deux exemples suivants mettent en évidence les types paramètres.

```

var list = new List<string>();
public DbSet<Client> Clients { get; set; }

```

Il est à noter que les types paramètres peuvent être omis dans certains cas. On parle alors d'inférence de type : le compilateur déduit les types paramètres de l'utilisation faite de l'élément de code générique.

```

using (var context = new SQLiteContext())
{
    var clients = context.Personnes
                        .OfType<Client>();

    Console.WriteLine("Clients dans le DbSet Personnes : ");

    foreach (var client in clients)
    {
        Console.WriteLine($"{client.Nom} {client.Prenom}");
    }
}

```

Comme attendu, l'affichage résultant ne liste que des clients.

```

Clients dans le DbSet Personnes :
NomClient1 PrenomClient1
NomClient2 PrenomClient2
NomClient3 PrenomClient3
NomClient4 PrenomClient4

```

Sous le "capot", les choses sont un peu plus compliquées que la vérification d'un prédicat. Le modèle de travail est en réalité associé à une base de données ne contenant qu'une table `Personnes`. Entity Framework Core différencie le type des enregistrements de cette table à l'aide d'une colonne discriminante. L'utilisation de l'opérateur `OfType` tire parti de cette structuration et génère une requête contenant une clause `WHERE` opérant sur la colonne discriminante de la table.

```

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

```

Il est à noter que fonctionnellement, le résultat de cette exécution peut être retrouvé à l'aide des opérateurs `Where` et `Select`.

```

using (var context = new SQLiteContext())
{
    var clients = context.Personnes
                        .Where(personne => personne is Client)
                        .Select(personne => (Client)personne);

    Console.WriteLine("OfType = Where + Select");

    foreach (var client in clients)
    {
        Console.WriteLine($"{client.Nom} {client.Prenom}");
    }
}

```

La requête générée par ces appels est légèrement différente, mais contient elle aussi la notion de filtrage sur le discriminant grâce à la transcription de l'opérateur C# `is`.

```
SELECT "personne"."Id", "personne"."Discriminator", "personne"."Nom",
"personne"."Prenom", "personne"."MontantTotalCommandes"
FROM "Personnes" AS "personne"
WHERE "personne"."Discriminator" IN ('Client', 'Personne') AND
("personne"."Discriminator" = 'Client')
```

Cast

L'opérateur `cast` travaille dans le sens opposé : il effectue un transtypage sur des objets issus d'entités enfants afin de les convertir vers le type de leur classe mère. Dans notre cas d'étude, son utilisation permet de manipuler les objets de type `Client` de la même manière que tout objet `Personne`.

```
using (var context = new SqliteContext())
{
    var personnesClientes = context.Clients
        .Cast<Personne>();

    Console.WriteLine("Clients transtypés en objets Personne");

    foreach (var personne in personnesClientes)
    {
        Console.WriteLine($"{personne.Nom} {personne.Prenom}");
    }
}
```

L'analyse de la requête SQL générée par la librairie montre qu'Entity Framework recherche exactement les mêmes objets que lors de l'utilisation de l'opérateur `OfType`. Seul l'effet est différent puisque la variable `personne` est une collection de type `IQueryable<Personne>`.

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

```
Clients transtypés en objets Personne
NomClient1 PrenomClient1
NomClient2 PrenomClient2
NomClient3 PrenomClient3
NomClient4 PrenomClient4
```

L'appel de la fonction `Cast<Client>` sur le `DbSet Personnes` entraîne, quant à lui, la levée d'une exception, puisqu'il est impossible de transformer un objet `Personne` en `Client` à l'aide d'un transtypage.

b. Manipulation du type des collections d'enregistrements

Le type `IQueryable<T>`, par son mode d'évaluation paresseux, est particulièrement pratique pour la construction progressive de requêtes. Ce fonctionnement peut en revanche s'avérer gênant, notamment lorsqu'il est question d'énumérations multiples des données, puisque chacune d'entre elles implique une nouvelle exécution de la requête sous-jacente. La sérialisation des objets

`IQueryable<T>` est également complexe puisqu'il faut généralement parcourir l'ensemble de l'arbre d'expressions qu'il maintient pour obtenir le résultat escompté.

Pour ces raisons (entre autres), LINQ inclut quatre opérateurs dédiés à la construction de collections à partir d'un objet implémentant `IEnumerable<T>`, ce qui est le cas de tout objet `IQueryable<T>`. Leur mission implique que leur fonctionnement n'est pas relié à la source de données : dans le cas d'une base de données relationnelle, ils n'ont pas de représentation SQL. Ils travaillent donc en énumérant l'ensemble des enregistrements retournés et créent une nouvelle collection locale qui référence ces éléments.

ToArray

Comme son nom l'indique, l'opérateur `ToArray` crée un tableau fortement typé d'enregistrements à partir de la collection source.

```
using (var context = new NorthwindContext())
{
    var clientsFrancais
        = context.Customers
            .where(client => client.Country == "France")
            .ToArray();

    Console.WriteLine($"Type de la collection clientsFrancais :
{clientsFrancais.GetType()}");
    Console.WriteLine();

    foreach (var client in clientsFrancais)
    {
        Console.WriteLine($"{client.CompanyName} - {client.City}");
    }
}
```

```
Type de la collection clientsFrancais :
ConsoleApplication1.Customers[]
```

```
Blondesdds1 père et fils - Strasbourg
Bon app' - Marseille
Du monde entier - Nantes
Folies gourmandes - Lille
France restauration - Nantes
La corne d'abondance - Versailles
La maison d'Asie - Toulouse
Paris spécialités - Paris
Spécialités du monde - Paris
Victuailles en stock - Lyon
vins et alcools Chevalier - Reims
```

ToList

Comme `ToArray`, l'opérateur `ToList()` énumère les enregistrements retournés par une séquence `IQueryable<T>` et retourne une collection, ici de type `List<T>`, contenant ces éléments.

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

```

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

    Console.WriteLine($"Type de la collection clientsFrancais :
{clientsFrancais.GetType()}");
    Console.WriteLine();

    foreach (var client in clientsFrancais)
    {
        Console.WriteLine($"{client.CompanyName} - {client.City}");
    }
}

```

Type de la collection clientsFrancais :
System.Collections.Generic.List`1[ConsoleApplication1.Customers]

Blondesdds1 père et fils - Strasbourg
Bon app' - Marseille
Du monde entier - Nantes
Folies gourmandes - Lille
France restauration - Nantes
La corne d'abondance - Versailles
La maison d'Asie - Toulouse
Paris spécialités - Paris
Spécialités du monde - Paris
Victuailles en stock - Lyon
vins et alcools Chevalier - Reims

ToDictionary

L'opérateur `ToDictionary` permet de transformer un ensemble d'enregistrements en une séquence de données de type `KeyValuePair<TKey, TValue>`, où `TKey` est le type de la clé et `TValue` est le type de l'élément issu de la collection source. Chacune des clés est calculée à partir d'une expression de projection passée en paramètre à l'opérateur et doit être unique.

```

using (var context = new NorthwindContext())
{
    var clients
        = context.Customers
            .Take(10)
            .ToDictionary(client => client.CustomerId);

    Console.WriteLine($"Type de la collection clients :
{clients.GetType()}");
    Console.WriteLine();

    foreach (var client in clients)
    {
        Console.WriteLine($"Clé : {client.Key} - Entreprise :
{client.Value.CompanyName}");
    }
}

```

```
}
```

```
Type de la collection clients : System.Collections.Generic.  
Dictionary`2[System.String,ConsoleApplication1.Customers]
```

```
Clé : ALFKI - Entreprise : Alfreds Futterkiste  
Clé : ANATR - Entreprise : Ana Trujillo Emparedados y helados  
Clé : ANTON - Entreprise : Antonio Moreno Taquería  
Clé : AROUT - Entreprise : Around the Horn  
Clé : BERGS - Entreprise : Berglunds snabbköp  
Clé : BLAUS - Entreprise : Blauer See Delikatessen  
Clé : BLONP - Entreprise : Blondesdds1 père et fils  
Clé : BOLID - Entreprise : Bólido Comidas preparadas  
Clé : BONAP - Entreprise : Bon app'  
Clé : BOTTM - Entreprise : Bottom-Dollar Markets
```

ToLookup

L'opérateur `ToLookup` est comparable dans son fonctionnement à l'opérateur `GroupBy` : il génère des groupements d'enregistrements en fonction d'une clé. Cette clé est calculée à partir d'une expression de projection passée en paramètre à l'opérateur. Lorsque deux éléments ont une clé considérée comme égale, ils sont ajoutés au même groupement.

```
using (var context = new NorthwindContext())  
{  
    var clientsParPays  
    = context.Customers  
        .Where(client => client.CompanyName.StartsWith("F"))  
        .ToLookup(client => client.Country);  
  
    Console.WriteLine($"Type de la collection clients :  
{clientsParPays.GetType()}");  
    Console.WriteLine();  
  
    foreach (var groupeClients in clientsParPays)  
    {  
        Console.WriteLine(groupeClients.Key);  
  
        foreach (var client in groupeClients)  
        {  
            Console.WriteLine($"\\t{client.CompanyName}");  
        }  
    }  
}
```

```
Brazil  
    Familia Arquibaldo  
Spain  
    FISSA Fabrica Inter. Salchichas S.A.  
France  
    Folies gourmandes  
    France restauration  
Sweden  
    Folk och fä HB
```

```
Germany
    Frankenversand
Italy
    Franchi S.p.A.
Portugal
    Furia Bacalhau e Frutos do Mar
```

c. Conversion de type de collection : les écueils à éviter

Les types résultant de l'utilisation des opérateurs `ToList`, `ToArray`, `ToDictionary` et `ToLookup` implémentent tous l'interface `IEnumerable<T>`. Par conséquent, l'API LINQ (To Objects, précisément) peut être utilisée à son tour pour effectuer différentes opérations sur les collections obtenues. Il faut donc rester vigilant quant à leur emploi afin d'éviter que des traitements ne soient exécutés localement par l'application alors qu'ils pourraient être déportés et traités, généralement de manière plus efficace, par la source de données.

À l'inverse, il est important de savoir appeler ces opérateurs de conversion de manière à éviter les énumérations multiples de l'objet `IQueryable<T>`. Celles-ci causent l'exécution de la même requête à plusieurs reprises et peuvent donc entraîner une dégradation des performances de l'application.

11. Opérateurs de quantification

Les opérateurs appartenant à cette catégorie ont tous pour particularité de retourner une valeur booléenne en fonction de l'évaluation d'une condition pour tout ou partie des éléments contenus dans la séquence source.

a. Any

L'opérateur `Any` évalue une condition sur chacun des enregistrements d'une collection source et renvoie `true` dès qu'un des éléments de la séquence respecte la condition. L'exemple suivant montre son fonctionnement pour le test d'existence d'un client localisé au Nicaragua.

```
using (var context = new NorthwindContext())
{
    var existeClientsNicaragua
        = context.Customers
            .Any(client => client.Country == "Nicaragua");

    Console.WriteLine(existeClientsNicaragua );
}
```

La requête SQL générée tire parti du mot-clé `EXISTS` afin de remonter uniquement une valeur scalaire (booléenne, en l'occurrence).

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

L'opérateur `Any` possède également une implémentation n'acceptant pas de paramètre. Dans ce contexte, il évalue implicitement l'existence d'éléments dans la collection source. Il retourne donc `false` lorsque cette séquence est vide, et `true` dans le cas contraire.

Pour vérifier s'il existe des clients dans la source de données, on écrit donc :

```
using (var context = new NorthwindContext())
{
    var existeClients = context.Customers
                           .Any();

    Console.WriteLine(existeClients);
}
```

Ce qui produit une requête comparable à celle de l'exemple précédent.

```
SELECT CASE
    WHEN EXISTS (
        SELECT 1
        FROM [Customers] AS [c])
    THEN CAST(1 AS BIT) ELSE CAST(0 AS BIT)
END
```

b. All

L'opérateur `All` s'assure qu'un prédicat passé en paramètre est vérifié pour l'ensemble des enregistrements d'une séquence source.

```
using (var context = new NorthwindContext())
{
    var tousClientsSontFrancais
        = context.Customers
               .All(client => client.Country == "France");

    Console.WriteLine(tousClientsSontFrancais);
}
```

Le code exécuté au niveau de la source de données tire lui aussi parti de l'instruction `EXISTS`, mais sous sa forme négative (`NOT EXISTS`). En effet, si on considère que tous les éléments doivent respecter une condition, alors le simple fait qu'un élément ne vérifie pas la condition permet d'affirmer que le résultat souhaité est `false`. On retombe donc plus ou moins dans le schéma d'utilisation de l'opérateur `Any`.

```
SELECT CASE
    WHEN NOT EXISTS (
        SELECT 1
        FROM [Customers] AS [client]
        WHERE NOT (([client].[Country] =
N'France') AND [client].[Country] IS NOT NULL))
    THEN CAST(1 AS BIT) ELSE CAST(0 AS BIT)
END
```

c. Contains

La détermination de l'appartenance d'un objet à une séquence d'éléments est déléguée à l'opérateur `Contains`. Celui-ci accepte en paramètre un objet qu'il recherche dans une collection source. Lorsque cet objet est trouvé, il renvoie `true`.

```
using (var context = new NorthwindContext())
{
    var premierClient = context.Customers.Take(1).ToList()[0];

    var premierClientEstClientFrancais
        = context.Customers
            .where(client => client.Country == "France")
            .Contains(premierClient);

    Console.WriteLine(premierClientEstClientFrancais);
}
```

La requête SQL générée montre que le traitement est en réalité effectué par l'application. La méthode `Contains` énumère sa collection source et tente de retrouver l'objet qui lui a été passé en paramètre.

12. Opérateurs d'élément

Les opérateurs appartenant à cette catégorie retournent un élément spécifique d'une collection source. Lorsque l'élément recherché est introuvable, ces opérateurs lèvent une exception. Chacun d'eux possède également une implémentation, dont le nom est accolé au suffixe `OrDefault`, qui évite ce comportement en renvoyant `null` lorsque aucune valeur ne peut être retournée.

Il est à noter que l'utilisation de chacun de ces opérateurs déclenche l'énumération de la collection source, ce qui implique pour le type `IQueryable<T>` l'exécution d'une requête SQL `SELECT`.

a. First et FirstOrDefault

L'opérateur `First` permet de récupérer le premier élément d'une séquence correspondant à un prédicat. L'ordre dans lequel la recherche est effectuée est exactement celui qui est associé à la collection source, et par extension, à la requête SQL générée. L'exemple de code suivant affiche le nom et la ville du premier client français.

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

    Console.WriteLine("Premier client français");
    Console.WriteLine($"{premierClientFrancais.CompanyName} -
{premierClientFrancais.City}");
}
```

L'affichage résultant de son exécution est simple :

Premier client français
Blondesdds1 père et fils - Strasbourg

Dans les coulisses, l'opérateur `First` modifie l'arbre d'expressions maintenu par l'objet `IQueryable<T>`. Aussi, lors de la génération de la requête SQL, Entity Framework est capable d'interpréter sa présence et intègre à l'instruction `SELECT` envoyée à la source de données un filtre `TOP(1)` et une clause `WHERE` correspondant au prédicat d'entrée.

```
SELECT TOP(1) [client].[CustomerID], [client].[Address],  
[client].[City], [client].[CompanyName], [client].[ContactName],  
[client].[ContactTitle], [client].[Country], [client].[Fax],  
[client].[Phone], [client].[PostalCode], [client].[Region]  
FROM [Customers] AS [client]  
WHERE [client].[Country] = N'France'
```

Il est également possible d'utiliser l'opérateur `First` sans lui fournir de condition en paramètre. Dans ce cas, il se contente de ramener le premier élément de la séquence source. Au niveau SQL, ce comportement se traduit par l'utilisation d'une clause `TOP(1)`.

```
using (var context = new NorthwindContext())  
{  
    var premierClient = context.Customers  
        .First();  
  
    Console.WriteLine("Premier client ");  
    Console.WriteLine($"{premierClient.CompanyName} -  
{premierClient.City}");  
}
```

`FirstOrDefault` est utilisé exactement de la même manière que `First`, mais retourne une valeur nulle lorsque aucun élément ne peut être renvoyé.

```
using (var context = new NorthwindContext())  
{  
    var premierClientBerjallien =  
        context.Customers  
            .FirstOrDefault(client => client.City ==  
                "Bourgoin-Jallieu");  
  
    if (premierClientBerjallien != null)  
    {  
        Console.WriteLine("Premier client berjallien");  
        Console.WriteLine($"{premierClientBerjallien.CompanyName}");  
    }  
    else  
    {  
        Console.WriteLine("Aucun client n'est localisé à Bourgoin-Jallieu");  
    }  
}
```

b. Last et LastOrDefault

L'opérateur `Last` est comparable à `First`. En effet, quand `First` renvoie le premier élément d'une séquence, `Last` renvoie le dernier. L'utilisation des deux opérateurs est strictement identique puisque `Last` peut être appelé sans paramètre ou avec un paramètre de type prédicat.

```
using (var context = new NorthwindContext())
{
    var derniereCommande =
        context.Orders
            .Last();

    Console.WriteLine($"La dernière commande a le numéro
{derniereCommande.OrderId}");
}

using (var context = new NorthwindContext())
{
    var dernierEmployeAmericain = context.Employees
        .Last(employe => employe.Country == "USA");

    Console.WriteLine("Dernier employé américain");
    Console.WriteLine($"{dernierEmployeAmericain.FirstName} -
{dernierEmployeAmericain.LastName}");
}
```

En revanche, l'analyse du code généré par Entity Framework montre que l'opérateur `Last` n'est pas transcrit vers le langage SQL. Le traitement associé est entièrement effectué en mémoire, par l'application. Il peut donc être plus intéressant d'utiliser l'opérateur `GroupBy` en conjonction avec l'opérateur `First` afin de maintenir les traitements au niveau de la source de données.

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

L'opérateur `LastOrDefault` est, quant à lui, utilisé lorsque `Last` peut opérer sur une séquence vide, ou lorsque le prédicat fourni en paramètre n'est jamais vérifié parmi les enregistrements de la séquence source.

```
using (var context = new NorthwindContext())
{
    var dernierEmployeChilien =
        context.Employees
            .LastOrDefault(client => client.Country == "Chili");

    if (dernierEmployeChilien != null)
    {
        Console.WriteLine("Dernier client chilien");
        Console.WriteLine($"{dernierEmployeChilien.FirstName}-
{dernierEmployeChilien.LastName}");
    }
}
```



```

else
{
    Console.WriteLine("Aucun employé n'est basé au Chili");
}
}

```

c. Single et SingleOrDefault

L'utilisation de l'opérateur `Single` est associée à la recherche d'un élément unique dans une collection d'enregistrements. Comme pour `First` et `Last`, il existe deux implémentations de `Single`.

La première ne possède aucun argument et renvoie le seul et unique élément contenu dans une collection source.

```

using (var context = new NorthwindContext())
{
    var premiereCommande =
        context.Orders
            .Take(1)
            .Single();

    Console.WriteLine($"Identifiant de la première commande :
{premiereCommande.OrderId}");
}

```

La transcription de cet opérateur dans une requête SQL est faite d'une manière assez astucieuse. On sait que l'on ne veut récupérer qu'un seul enregistrement issu de la collection source. Pour vérifier cette condition, le meilleur moyen est d'essayer de récupérer deux enregistrements dans ladite collection. Ainsi, il est possible de récupérer et de valider le nombre d'éléments demandés en une seule passe, sans surcoût en termes de temps processeur ou de bande passante. Pour réaliser cette opération, l'expression associée à l'opérateur `Single` est traduite par une encapsulation de la requête source dans une instruction `SELECT TOP(2)`.

```

SELECT TOP(2) [t].*
FROM (
    SELECT TOP(@__p_0) [o].[OrderID], [o].[CustomerID], [o].[EmployeeID],
[o].[Freight], [o].[OrderDate], [o].[RequiredDate], [o].[ShipAddress],
[o].[ShipCity], [o].[ShipCountry], [o].[ShipName], [o].[ShipPostalCode],
[o].[ShipRegion], [o].[ShipVia], [o].[ShippedDate]
    FROM [Orders] AS [o]
) AS [t]

```

La seconde implémentation de l'opérateur `Single` attend un paramètre définissant une condition que l'élément retourné doit vérifier.

```
using (var context = new NorthwindContext())
{
    var premiereCommande =
        context.Orders
            .Single(commande => commande.OrderId == 10248);

    Console.WriteLine($"Date de la commande 10248 :
        {premiereCommande.OrderDate?.ToShortDateString()
            ?? "non renseignée"}");
}
```

L'ajout de cette condition se traduit par l'ajout d'une clause `WHERE` dans la requête SQL générée.

```
SELECT TOP(2) [commande].[OrderID], [commande].[CustomerID],
[commande].[EmployeeID], [commande].[Freight], [commande].[OrderDate],
[commande].[RequiredDate], [commande].[ShipAddress],
[commande].[ShipCity], [commande].[ShipCountry], [commande].[ShipName],
[commande].[ShipPostalCode], [commande].[ShipRegion],
[commande].[ShipVia], [commande].[ShippedDate]
FROM [Orders] AS [commande]
WHERE [commande].[OrderID] = 10248
```

L'utilisation de `Single` est génératrice d'exceptions lorsque plusieurs éléments peuvent être retournés. Ce comportement peut être adouci à l'aide de `SingleOrDefault`, qui renvoie une valeur nulle lorsque la séquence source ne contient pas un unique élément correspondant au critère de recherche.

```
using (var context = new NorthwindContext())
{
    var seulClient =
        context.Customers
            .SingleOrDefault();

    if (seulClient != null)
        Console.WriteLine($"Seul client enregistré : {seulClient.CompanyName}");
}
```

Cycle de vie des entités

De leur création à leur suppression, les entités passent par différents états qui déterminent la manière dont Entity Framework doit les traiter. Nous verrons dans cette section quels sont ces états et comment les manipuler. Nous jetterons également un œil de l'autre côté du miroir pour comprendre comment Entity Framework Core les gère, puis pour agir de manière à mettre les entités hors de sa portée, ou au contraire, à les placer sous sa surveillance.

1. Cinq types de manipulations, cinq états

Les différentes manipulations de données effectuées au travers d'Entity Framework sont orchestrées à l'aide de différents composants, dont le Change Tracker (que l'on pourrait traduire par "Surveillant des modifications"). Avant de détailler son rôle et son fonctionnement, il est important de comprendre les états que peut avoir une entité, ainsi que les actions qui mènent à son changement d'état.

L'énumération `Microsoft.EntityFrameworkCore.EntityState` définit cinq valeurs qui couvrent l'ensemble du cycle de vie des entités.

Detached

Cette valeur est associée aux entités qui ne sont pas attachées à un contexte de données. L'état associé à une entité est généralement égal à `Detached` lorsque l'entité a été supprimée de la source de donnée (à l'aide d'Entity Framework) ou que le développeur détache impérativement l'entité. Il est également possible d'arriver à cet état par une demande de suppression d'une entité qui n'a pas d'enregistrement associé au niveau de la source de données, mais qui est sous la surveillance d'un contexte de données (état `Added`).

Unchanged

L'état d'une entité est défini à `Unchanged` lorsqu'elle est attachée à un contexte de données et que son contenu est identique à celui de l'enregistrement qu'elle représente. C'est notamment le cas juste après l'exécution d'une requête de sélection sur la source de données. La validation d'un ajout ou d'une modification sur une entité entraîne également un basculement de son état vers `Unchanged`.

Added

Une entité a pour état `Added` lorsqu'elle a été créée et ajoutée au contexte de données. Aucun enregistrement associé n'existe à ce stade dans la source de données. Ce sera chose faite pendant la prochaine validation de changements : une requête d'ajout sera envoyée à la source de données pour enregistrer les informations de l'entité.

Modified

L'état `Modified` est associé aux entités sous la surveillance d'un contexte pour lesquelles une demande de modification a été faite. Il indique qu'une requête de modification doit être envoyée à la source de données à la prochaine validation de changements.

Deleted

Ce statut résulte de la demande de suppression d'une entité surveillée par un contexte de données. Il indique qu'elle doit être physiquement supprimée à la prochaine validation de changements effectuée par le développeur.

L'exemple suivant montre l'ensemble des étapes permettant d'accéder à ces différents états.

```
using (var context = new NorthwindContext())
{
    var client = new Customers()
    {
        CustomerId = "PABEL",
        CompanyName = "Pastéis de Belém",
        Address = "84, Rua de Belém",
        PostalCode = "1300 - 085",
        City = "Lisboa",
        Country = "Portugal"
    };

    //Ajout de client au contexte de données
    //La surveillance de l'entité commence ici.
```

```

context.Add(client);
//Récupération de l'entrée associée à l'entité dans le Change Tracker du
contexte
EntityEntry clientEntry = context.ChangeTracker.Entries<Customers>
().Single();
Console.WriteLine($"Ajout de l'entité dans le contexte. Son statut est
{clientEntry.State}");

context.SaveChanges();
Console.WriteLine($"Données de l'entité enregistrées en base.
Son statut est {clientEntry.State}");

client.Phone = "+351 21 363 74 23";
client.Fax = "+351 21 363 80 78";

context.Update(client);
Console.WriteLine($"Demande de modification de l'entité.
Son statut est {clientEntry.State}");

context.SaveChanges();
Console.WriteLine($"Données de l'entité enregistrées en base.
Son statut est {clientEntry.State}");

context.Remove(client);
Console.WriteLine($"Demande de suppression de l'entité.
Son statut est {clientEntry.State}");

context.SaveChanges();
Console.WriteLine($"Suppression enregistrée en base.
Son statut est {clientEntry.State}");
}

```

Cette portion de code affiche les lignes suivantes.

```

Ajout de l'entité dans le contexte. Son statut est Added
Données de l'entité enregistrées en base. Son statut est Unchanged
Demande de modification de l'entité. Son statut est Modified
Données de l'entité enregistrées en base. Son statut est Unchanged
Demande de suppression de l'entité. Son statut est Deleted
Suppression enregistrée en base. Son statut est Detached

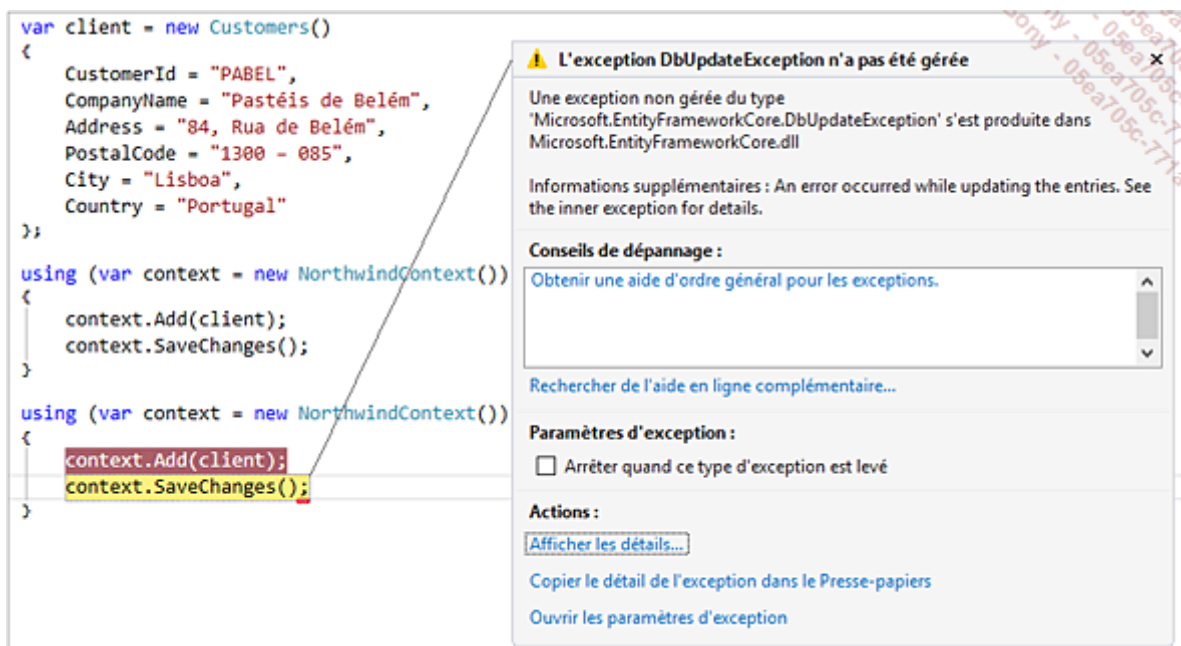
```

La lecture de ce code montre une relation entre l'utilisation de certaines méthodes du type `DbContext` et les changements d'état d'une entité.

Add

La méthode `Add` ajoute une entité au contexte de données. À ce stade, l'entité est considérée comme n'existant pas. Manipulation des entités : ajoutera au niveau de la source de données. C'est le développeur qui a la responsabilité de n'ajouter que des entités neuves, de manière à créer des enregistrements sans déclencher de conflits au niveau des contraintes de clé primaire. L'état de l'entité ajoutée par un appel à `Add` est `Added`.

Toute tentative d'ajout d'une entité déjà présente au niveau de la source de données déclenche une exception de type `DbUpdateException` au moment de la validation des changements.

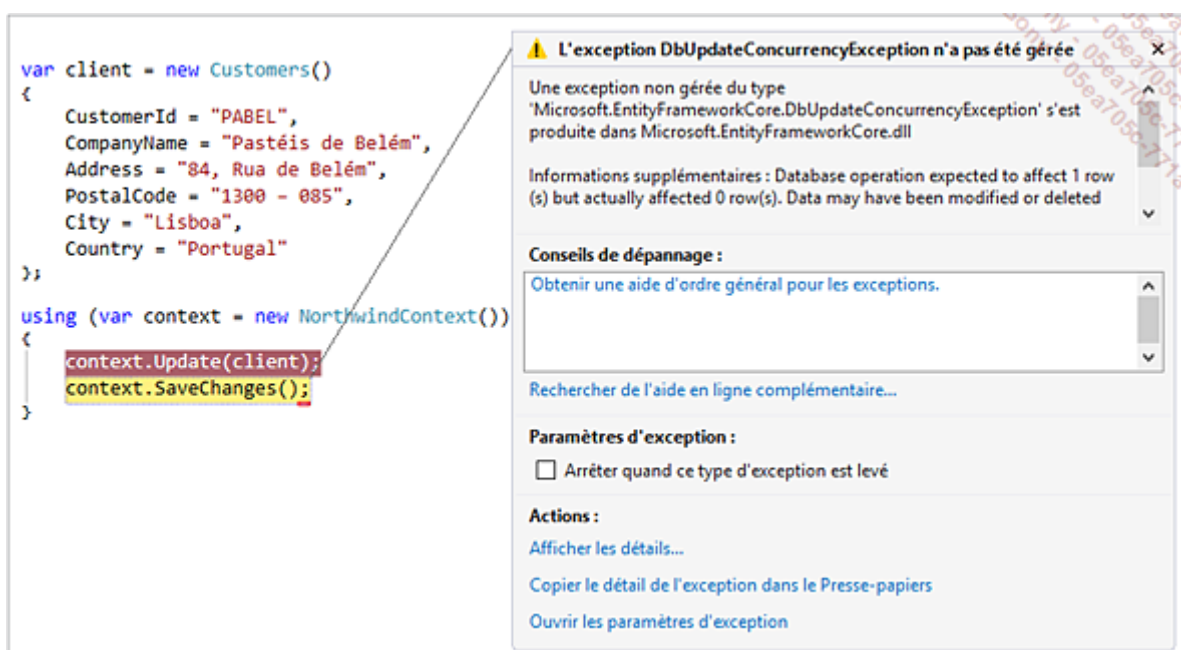


Le message exact remonté dans la propriété `InnerException.Message` de cette exception est ici :

Violation of PRIMARY KEY constraint 'PK_Customers'. Cannot insert duplicate key in object 'dbo.Customers'. The duplicate key value is (PABEL). The statement has been terminated.

Update

L'utilisation de la méthode `Update` assigne à une entité l'état `Modified`. Elle est alors implicitement considérée comme reflétant un enregistrement de la source de données sur lequel une modification doit être répercutée. Il est donc du ressort du développeur de ne placer dans cet état que des entités associées à un enregistrement du côté de la source de données. En effet, la tentative de validation d'une mise à jour sur une entité inexistante du côté de la source de données est traduite par Entity Framework comme une erreur due à un accès concurrent.



Remove

`Remove` est une méthode destinée à marquer une entité comme supprimée. L'état d'une telle entité est `Deleted`. Comme la méthode `Update`, `Remove` présuppose que chaque entité qui lui est passée correspond à un enregistrement de la source de données. Le développeur a donc la responsabilité de vérifier que cette association existe. À défaut, une exception de type `DbConcurrencyException` pourra être levée.

Un cas particulier d'utilisation est toutefois à noter : lorsque `Remove` est utilisée pour supprimer un objet dont l'état est `Added`, aucun enregistrement associé n'existe au niveau de la source de données. L'état de cette entité est alors simplement valorisé avec la valeur `Detached`.

Ces différentes manipulations peuvent également être exécutées à partir de collections `DbSet<T>`.

```
using (var context = new NorthwindContext())
{
    var client = new Customers()
    {
        CustomerId = "PABEL",
        CompanyName = "Pastéis de Belém",
        Address = "84, Rua de Belém",
        PostalCode = "1300 - 085",
        City = "Lisboa",
        Country = "Portugal"
    };

    context.Customers.Add(client);
    context.SaveChanges();

    context.Customers.Update(client);
    context.SaveChanges();

    context.Customers.Remove(client);
    context.SaveChanges();
}
```

2. Détection des changements

La détection de changements est un processus clé dans le fonctionnement d'Entity Framework Core, puisque c'est grâce à lui que la librairie est capable de générer les requêtes de mises à jour exécutées pour répercuter les manipulations de données au niveau de la source de données.

C'est le composant nommé Change Tracker qui a la responsabilité de la détection de changements sur les entités qu'il surveille. Il est exposé publiquement par le type `DbContext` au travers de la propriété `ChangeTracker`, et nous l'avons d'ailleurs manipulé plus tôt pour obtenir l'état des entités. Les différentes propriétés et méthodes de cet objet permettent de modifier son comportement ainsi que d'implémenter des logiques de manipulation de données personnalisées.

a. Détection automatique : 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 `AutoDetectChangesEnabled` permet d'activer ou de désactiver ce comportement de manière à imposer plus de rigueur ou à autoriser plus de flexibilité dans l'écriture du code. Le risque associé à sa désactivation réside bien évidemment dans la possibilité qu'une modification ne soit pas répercutée au niveau de la source de données. Par défaut, la détection automatique est activée.

Les quatre exemples qui suivent montrent le comportement adopté par Entity Framework en fonction de l'état d'activation de la détection automatique ainsi que de l'utilisation ou non de la méthode `update`. Pour tous ces exemples, la variable `client` utilisée est identique à celle que nous avons définie plus tôt.

```
var client = new Customers()
{
    CustomerId = "PABEL",
    CompanyName = "Pastéis de Belém",
    Address = "84, Rua de Belém",
    PostalCode = "1300 - 085",
    City = "Lisboa",
    Country = "Portugal"
};
```

Détection automatique activée et utilisation de la méthode Update

```
using (var context = new NorthwindContext())
{
    context.ChangeTracker.AutoDetectChangesEnabled = true;

    context.Add(client);
    context.SaveChanges();

    client.Phone = "+351 21 363 74 23";
    client.Fax = "+351 21 363 80 78";

    client.Orders.Add(new Orders {
        CustomerId = client.CustomerId,
        EmployeeId = 1 /* Nancy Davolio */,
        OrderDate = DateTime.Today });

    context.Update(client);
    context.SaveChanges();
}
```

L'instruction de validation des changements déclenche ici l'exécution des requêtes suivantes.

Les éléments dont le nom commence par `@p` sont des paramètres de requête.

```
UPDATE [Customers] SET [Address] = @p0, [City] = @p1, [CompanyName] = @p2,
[ContactName] = @p3, [ContactTitle] = @p4, [Country] = @p5, [Fax] = @p6,
[Phone] = @p7, [PostalCode] = @p8, [Region] = @p9
WHERE [CustomerID] = @p10;
```

```

SELECT @@ROWCOUNT;

INSERT INTO [Orders] ([CustomerID], [EmployeeID], [OrderDate],
[RequiredDate], [ShipAddress], [ShipCity], [ShipCountry], [ShipName],
[ShipPostalCode], [ShipRegion], [ShipVia], [ShippedDate])
VALUES (@p11, @p12, @p13, @p14, @p15, @p16, @p17, @p18, @p19, @p20, @p21,
@p22);

SELECT [OrderID], [Freight]
FROM [Orders]
WHERE @@ROWCOUNT = 1 AND [OrderID] = scope_identity();

```

Quatre requêtes sont générées et envoyées à la source de données lors de l'exécution de ce code.

La requête `UPDATE` contient l'ensemble des données de l'entité afin de mettre à jour la totalité des champs de l'enregistrement.

Une requête `INSERT` enregistre les informations de la commande que l'on a associée au client.

Les deux autres requêtes sont liées à la gestion d'accès concurrentiels.

Détection automatique désactivée et utilisation de la méthode Update

```

using (var context = new NorthwindContext())
{
    context.ChangeTracker.AutoDetectChangesEnabled = false;

    context.Add(client);
    context.SaveChanges();

    client.Phone = "+351 21 363 74 23";
    client.Fax = "+351 21 363 80 78";

    client.Orders.Add(new Orders {
        CustomerId = client.CustomerId,
        EmployeeId = 1 /* Nancy Davolio */,
        OrderDate = DateTime.Today });

    context.Update(client);
    context.SaveChanges();
}

```

Le code SQL généré lors de l'appel à `SaveChanges` est le suivant.

```

UPDATE [Customers] SET [Address] = @p0, [City] = @p1, [CompanyName] = @p2,
[ContactName] = @p3, [ContactTitle] = @p4, [Country] = @p5, [Fax] = @p6,
[Phone] = @p7, [PostalCode] = @p8, [Region] = @p9
WHERE [CustomerID] = @p10;

SELECT @@ROWCOUNT;

```

Ce script est nettement plus court que le précédent. La requête `UPDATE` est toujours présente, mais la requête `INSERT` ne l'est pas.

Le contexte d'exécution permet de déduire que cette absence est liée à la désactivation de la détection automatique des changements. Et en effet, c'est bien ce processus qui permet d'inclure les entités enfants. Le détecteur de modifications traverse le graphe de chaque entité qui lui est attachée à la recherche de différences entre l'état initial et l'état actuel d'une valeur, et marque chaque entité altérée (quelle que soit sa place dans le graphe) avec le statut `Modified`.

Détection automatique activée, mais pas d'utilisation de la méthode Update

```
using (var context = new NorthwindContext())
{
    context.ChangeTracker.AutoDetectChangesEnabled = true;

    context.Add(client);
    context.SaveChanges();

    client.Phone = "+351 21 363 74 23";
    client.Fax = "+351 21 363 80 78";

    client.Orders.Add(new Orders {
        CustomerId = client.CustomerId,
        EmployeeId = 1 /* Nancy Davolio */,
        OrderDate = DateTime.Today });

    context.SaveChanges();
}
```

À la validation des modifications, le script suivant est généré, puis exécuté.

```
UPDATE [Customers] SET [Fax] = @p0, [Phone] = @p1
WHERE [CustomerID] = @p2;

SELECT @@ROWCOUNT;

INSERT INTO [Orders] ([CustomerID], [EmployeeID], [OrderDate],
[RequiredDate], [ShipAddress], [ShipCity], [ShipCountry], [ShipName],
[ShipPostalCode], [ShipRegion], [ShipVia], [ShippedDate])
VALUES (@p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10, @p11, @p12, @p13, @p14);

SELECT [OrderID], [Freight]
FROM [Orders]
WHERE @@ROWCOUNT = 1 AND [OrderID] = scope_identity();
```

On retrouve ici les deux requêtes `UPDATE` et `INSERT` permettant d'enregistrer les modifications apportées au client ainsi qu'à sa commande.

Le nombre de champs concernés par la mise à jour des données est bien moindre que dans les deux exemples précédents. On en déduit que l'inclusion de tous les champs de l'entité `Customers` dans la requête `UPDATE` est le fruit de l'utilisation de la méthode `Update`. Le détecteur de changements permet d'affiner et d'optimiser les requêtes exécutées en n'incluant que les données modifiées dans la liste d'instructions à destination de la source de données.

Détection automatique désactivée et pas d'utilisation de la méthode Update

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

```

{
    context.ChangeTracker.AutoDetectChangesEnabled = false;

    context.Add(client);
    context.SaveChanges();

    client.Phone = "+351 21 363 74 23";
    client.Fax = "+351 21 363 80 78";

    client.Orders.Add(new Orders {
        CustomerId = client.CustomerId,
        EmployeeId = 1 /* Nancy Davolio */,
        OrderDate = DateTime.Today });

    context.SaveChanges();
}

```

Ici, strictement aucune requête n'est générée pour répercuter les modifications au niveau de la base de données. Le développeur doit introduire un déclenchement manuel de la détection de changements de manière à se retrouver dans le cas de l'exemple précédent.

En résumé

L'appel à la méthode `update` entraîne la génération d'une requête incluant l'ensemble des données scalaires de l'entité concernée, que la détection automatique soit activée ou non. Les entités enfants ne sont pas prises en compte et doivent faire l'objet d'instructions explicites pour être prises en compte (`Add`, `Update` ...), sauf si la détection automatique de changements est activée.

Avec la détection automatique des changements, le graphe d'entités complet est traversé au moment de la validation des modifications, de manière à répercuter les modifications relatives à toutes les entités concernées au niveau de la source de données.

Lorsque la détection automatique des changements est désactivée, il est nécessaire d'indiquer explicitement les opérations à exécuter en appelant la méthode `update` ou en déclenchant manuellement la détection des changements.

b. Détection manuelle : DetectChanges

La détection de changements est déclenchée manuellement à l'aide de la méthode `DetectChanges` du Change Tracker.

```

using (var context = new NorthwindContext())
{
    context.ChangeTracker.AutoDetectChangesEnabled = false;

    context.Add(client);
    context.SaveChanges();

    client.Phone = "+351 21 363 74 23";
    client.Fax = "+351 21 363 80 78";

    client.Orders.Add(new Orders {
        CustomerId = client.CustomerId,
        EmployeeId = 1 /* Nancy Davolio */,

```

```

        OrderDate = DateTime.Today });

    context.ChangeTracker.DetectChanges();
    context.SaveChanges();
}

```

Le processus déclenché à l'aide de cette méthode est identique à celui qui est exécuté lorsque la détection est automatique. Le code SQL généré est donc exactement le même que dans le troisième cas décrit dans la section précédente (Détection automatique activée, mais pas d'utilisation de la méthode `Update`).

```

UPDATE [Customers] SET [Fax] = @p0, [Phone] = @p1
WHERE [CustomerID] = @p2;

SELECT @@ROWCOUNT;

INSERT INTO [Orders] ([CustomerID], [EmployeeID], [OrderDate],
[RequiredDate], [ShipAddress], [ShipCity], [ShipCountry], [ShipName],
[ShipPostalCode], [ShipRegion], [ShipVia], [ShippedDate])
VALUES (@p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10, @p11, @p12, @p13, @p14);

SELECT [OrderID], [Freight]
FROM [Orders]
WHERE @@ROWCOUNT = 1 AND [OrderID] = scope_identity();

```

La méthode `DetectChanges` permet de séparer l'exécution de la détection et la validation des changements, ce qui peut ouvrir la porte à certains scénarios d'utilisation, comme l'affichage des modifications pour les confirmer avant de les valider.

QueryTrackingBehavior

Par défaut, toutes les entités retournées par des requêtes LINQ sont incluses dans la liste des entrées surveillées par le Change Tracker. La propriété `QueryTrackingBehavior` permet de contrôler ce comportement à l'aide de deux valeurs d'énumération :

- `QueryTrackingBehavior.TrackAll` : c'est la valeur par défaut, qui indique au moteur que les données retournées doivent être surveillées.
- `QueryTrackingBehavior.NoTracking` : cette valeur configure le contexte de données de manière qu'il ne surveille jamais implicitement les entités issues de requêtes LINQ. Elle est utile pour l'implémentation de scénarios dans lesquels les données issues de la source sont en lecture seule. Aucun traitement relatif à la détection de changements n'est alors exécuté sur les entités lues, ce qui évite les mises à jour non intentionnelles et diminue également le temps processeur utilisé par Entity Framework.

c. Stratégies de détection des changements

Jusqu'ici, nous avons vu différents moyens de contrôler QUAND est exécutée la détection de changements, mais le mode de fonctionnement (le COMMENT) reste le même : le 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 modification de la stratégie de détection n'est pas effectuée au niveau d'une instance d'un contexte de données, mais relativement à un modèle dans son ensemble, ce qui implique qu'elle se révèle « impactante » pour l'application.

Ce changement doit être effectué au niveau de la méthode `OnModelCreating` du contexte de données. Pour cela, il faut utiliser la méthode `HasChangeTrackingStrategy` du type `ModelBuilder`, en lui passant une valeur de l'énumération `ChangeTrackingStrategy`.

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

    base.OnModelCreating(modelBuilder);
}
```

L'énumération `ChangeTrackingStrategy` expose quatre valeurs : `Snapshot`, dont nous avons déjà évoqué le fonctionnement, `ChangedNotifications`, `ChangingAndChangedNotifications` et `ChangingAndChangedNotificationsWithOriginalValues`. Nous allons détailler le mode de fonctionnement et les implications liés à chacune de ces valeurs. Pour cela, nous partirons à chaque fois d'un modèle de données plus concis et simple que Northwind, utilisant une base de données SQLite. Vous pouvez bien sûr l'adapter à la source de données de votre choix.

```
public class ChangeTrackingStrategyContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite(@"Data source=mydb.db");
    }

    public DbSet<Personne> Personnes { get; set; }
}

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

    public string Nom { get; set; }

    public string Prenom { get; set; }

    public string ville { get; set; }

    public string Pays { get; set; }
}
```

Snapshot

La stratégie `Snapshot` est utilisée par défaut pour la détection des changements. Les exemples précédents utilisent donc le mode de fonctionnement détaillé ici. Pour utiliser explicitement ce mode de fonctionnement, il faut intégrer un appel à `ModelBuilder.HasChangeTrackingStrategy` dans la méthode `OnModelCreating` du contexte.

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.HasChangeTrackingStrategy(
        ChangeTrackingStrategy.Snapshot);

    base.OnModelCreating(modelBuilder);
}
```

Lorsqu'il est déclenché, le processus de détection fait une comparaison "avant/après" sur chacune des valeurs des propriétés mappées de l'entité. Toute comparaison révélant une inégalité entre la valeur initiale et la valeur actuelle entraîne le passage de l'entité au statut `Modified`. L'exemple de code suivant montre l'utilisation de la stratégie `Snapshot` avec l'exécution d'une détection manuelle, de manière à suivre l'état de l'entité et de la propriété modifiée.

```
using (var context = new ChangeTrackingStrategyContext())
{
    context.ChangeTracker.AutoDetectChangesEnabled = false;

    var personne = new Personne
    {
        Nom = "PUTIER",
        Prenom = "Seb"
    };

    context.Add(personne);
    context.SaveChanges();

    var entityEntry = context.ChangeTracker.Entries<Personne>().First();
    var propertyEntry = entityEntry.Property<string>(nameof(Personne.Prenom));
    Console.WriteLine($"Entité enregistrée - Etat de l'entité :
{entityEntry.State}");
    Console.WriteLine($"Propriété Prenom - Valeur initiale :
{propertyEntry.OriginalValue} -
Valeur Actuelle : {propertyEntry.CurrentValue}");

    personne.Prenom = "Sébastien";
    Console.WriteLine($"Propriété Prenom enregistrée - Etat de l'entité :
{entityEntry.State}");
    Console.WriteLine($"Propriété Prenom - Valeur initiale :
{propertyEntry.OriginalValue} -
Valeur Actuelle : {propertyEntry.CurrentValue}");

    context.ChangeTracker.DetectChanges();
    Console.WriteLine($"Détection déclenchée - Etat de l'entité :
{entityEntry.State}");
    Console.WriteLine($"Propriété Prenom - Valeur initiale :
{propertyEntry.OriginalValue} -
Valeur Actuelle : {propertyEntry.CurrentValue}");

    context.SaveChanges();
    Console.WriteLine($"Entité enregistrée - Etat de l'entité :
{entityEntry.State}");
    Console.WriteLine($"Propriété Prenom - Valeur initiale :
{propertyEntry.OriginalValue} -
```

```
        Valeur Actuelle : {propertyEntry.CurrentValue}");  
    }
```

L'affichage provoqué par l'exécution de ce code montre bien que l'état de l'entité ne passe à `Modified` qu'une fois le processus de détection exécuté, alors que la valeur de la propriété `Prenom` change avant la détection.

```
Entité enregistrée - Etat de l'entité : Unchanged  
Propriété Prenom - Valeur initiale : Seb - Valeur Actuelle : Seb  
  
Propriété Prenom modifiée - Etat de l'entité : Unchanged  
Propriété Prenom - Valeur initiale : Seb - Valeur Actuelle : Sébastien  
  
Détection déclenchée - Etat de l'entité : Modified  
Propriété Prenom - Valeur initiale : Seb - Valeur Actuelle : Sébastien  
  
Entité enregistrée - Etat de l'entité : Unchanged  
Propriété Prenom - Valeur initiale : Sébastien - Valeur Actuelle : Sébastien
```

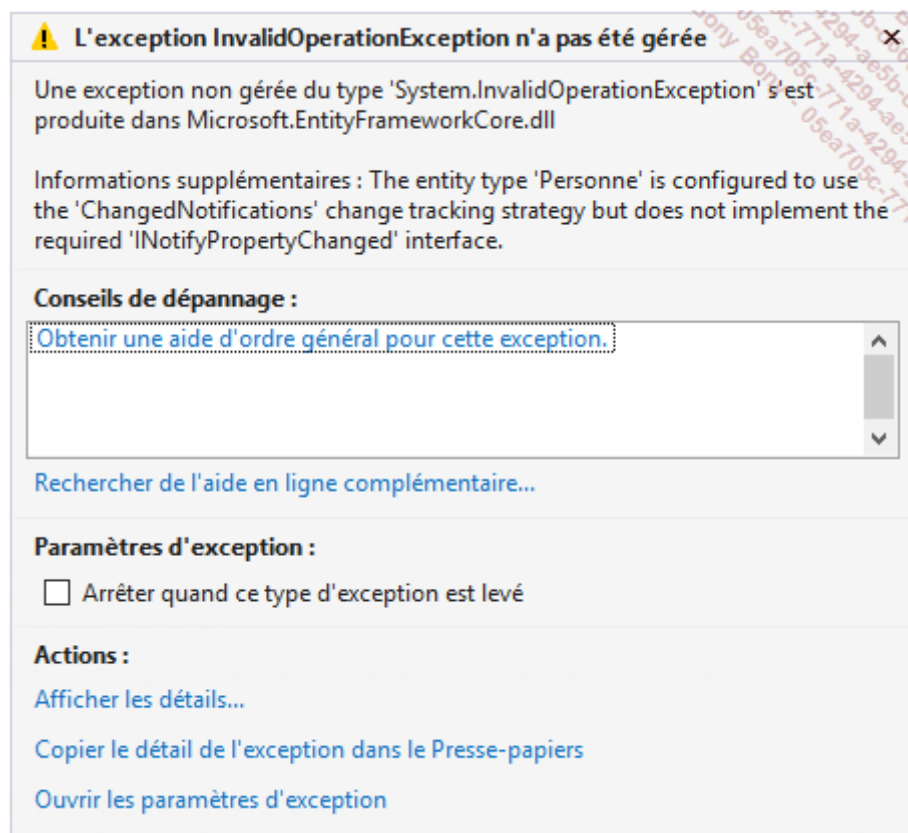
ChangedNotifications

Cette stratégie transforme la gestion des détections de changements en rendant les entités actrices de l'évolution de leur état. La logique de décision utilisée par le Change Tracker pour déterminer l'état de l'entité est déportée dans le code du type d'entité, avec un canal de communication entre ces éléments qui utilise l'interface `INotifyPropertyChanged`. Le principe est simple : lorsque l'entité déclenche un événement `PropertyChanged`, le Change Tracker, qui est abonné à cet événement, passe l'état de l'entité à `Modified`.

Pour utiliser cette stratégie, la première étape correspond à l'appel de la méthode `ModelBuilder.HasChangeTrackingStrategy` dans la méthode `OnModelCreating` du type de contexte.

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

À ce stade, exécuter l'exemple décrit pour la stratégie `Snapshot` déclenche une exception liée à la non-implémentation de l'interface requise `INotifyPropertyChanged` dans le type `Personne`. Il n'est donc absolument pas facultatif d'implémenter cette interface : TOUS les types d'entités inclus dans le modèle, même implicitement, doivent le faire.



Afin de se conformer aux besoins de cette stratégie, le type `Personne` est donc modifié pour implémenter l'interface `INotifyPropertyChanged`.

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

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

    private string _prenom;
    public string Prenom
    {
        get { return _prenom; }
        set
        {
            if (value != _prenom)
            {
                _prenom = value;
                OnPropertyChanged(nameof(Prenom));
            }
        }
    }
}
```

```

    }
}

public event PropertyChangedEventHandler PropertyChanged;

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

```

L'exécution de l'exemple vu précédemment se passe maintenant convenablement et produit le résultat suivant :

```

Entité enregistrée - Etat de l'entité : Unchanged
Propriété Prenom - Valeur initiale : Seb - Valeur Actuelle : Seb

Propriété Prenom enregistrée - Etat de l'entité : Modified
Propriété Prenom - Valeur initiale : Seb - Valeur Actuelle : Sébastien

Détection déclenchée - Etat de l'entité : Modified
Propriété Prenom - Valeur initiale : Seb - Valeur Actuelle : Sébastien

Entité enregistrée - Etat de l'entité : Unchanged
Propriété Prenom - Valeur initiale : Sébastien - Valeur Actuelle : Sébastien

```

L'effet attendu au niveau de la source de données est vérifié, puisque l'enregistrement associé à l'entité est présent dans la base de données, mais l'affichage indique une modification de comportement par rapport à la stratégie `Snapshot`. En effet, la stratégie par défaut maintient l'état `Unchanged` jusqu'à l'exécution de la méthode `DetectChanges`, tandis que l'état passe à `Modified` dès l'assignation d'une nouvelle valeur à la propriété `Prenom` avec la stratégie actuelle.

Le code SQL est également généré en tenant compte des notifications envoyées par l'entité puisqu'il ne met à jour que les champs pour lesquels la propriété correspondante a déclenché l'événement `PropertyChanged`.

```

UPDATE "Personnes" SET "Prenom" = @p0
WHERE "Id" = @p1;

```

Si la propriété `Nom` avait également été modifiée, la requête de mise à jour le refléterait :

```

UPDATE "Personnes" SET "Nom" = @p0, "Prenom" = @p1
WHERE "Id" = @p2;

```

Avec la stratégie `ChangedNotifications`, la détection de changements est donc plus instantanée et plus flexible puisqu'elle est implémentée et exécutée directement par les types d'entités. En revanche, la modification de ce comportement a un coût non négligeable puisqu'elle nécessite l'implémentation (et la maintenance !) de l'interface `INotifyPropertyChanged` sur l'ensemble des entités du modèle.

ChangingAndChangedNotifications

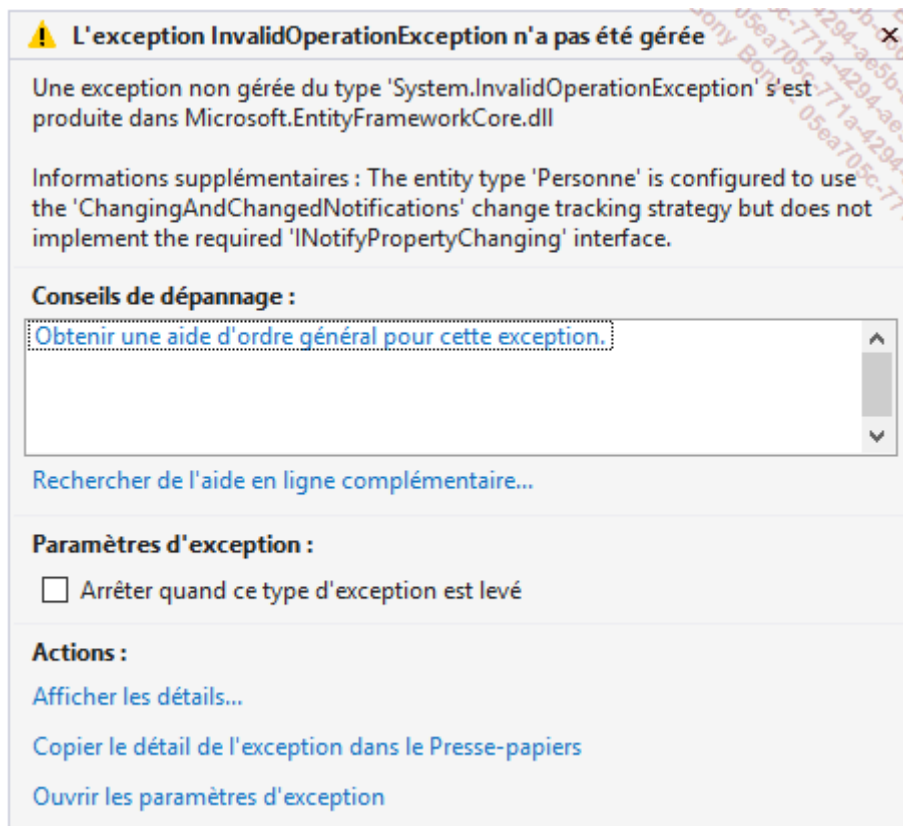
Comme la stratégie précédente, `ChangingAndChangedNotifications` permet de rendre les entités actives au niveau de la détection de changements. Cette stratégie est relativement identique à `ChangedNotifications`, puisque le passage d'une entité à l'état `Modified` est ici aussi conditionné par le déclenchement d'un événement `PropertyChanged`. En revanche, la gestion des valeurs originales de propriétés est bien plus particulière. Avec `ChangedNotifications` et `Snapshot`, la valeur originale de chaque propriété mappée est enregistrée lorsque l'entité est ajoutée au contexte ou que son état passe à `Unchanged` suite à une validation de modifications. Lorsque l'on utilise la stratégie `ChangingAndChangedNotifications`, seules certaines propriétés voient leur valeur initiale enregistrée, et cette opération est déclenchée unitairement lorsque l'entité émet un événement `PropertyChanging`.

Comme pour les stratégies précédentes, il faut tout d'abord activer cette stratégie dans la méthode `OnModelCreating` du type de contexte.

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.HasChangeTrackingStrategy(
        ChangeTrackingStrategy.ChangingAndChangedNotifications);

    base.OnModelCreating(modelBuilder);
}
```

Comme on peut l'extrapoler à partir de son nom, la stratégie `ChangingAndChangedNotifications` requiert que chaque type d'entité intégré au modèle objet implémente les deux interfaces `INotifyPropertyChanged` et `INotifyPropertyChanging`. À défaut, une exception indiquant ce prérequis est levée par Entity Framework Core.



L'implémentation de la classe `Personne` est reprise de l'exemple précédent, avec l'ajout de l'interface `INotifyPropertyChanging`.

```

public class Personne : INotifyPropertyChanging, INotifyPropertyChanged
{
    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));
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    public event PropertyChangingEventHandler PropertyChanging;

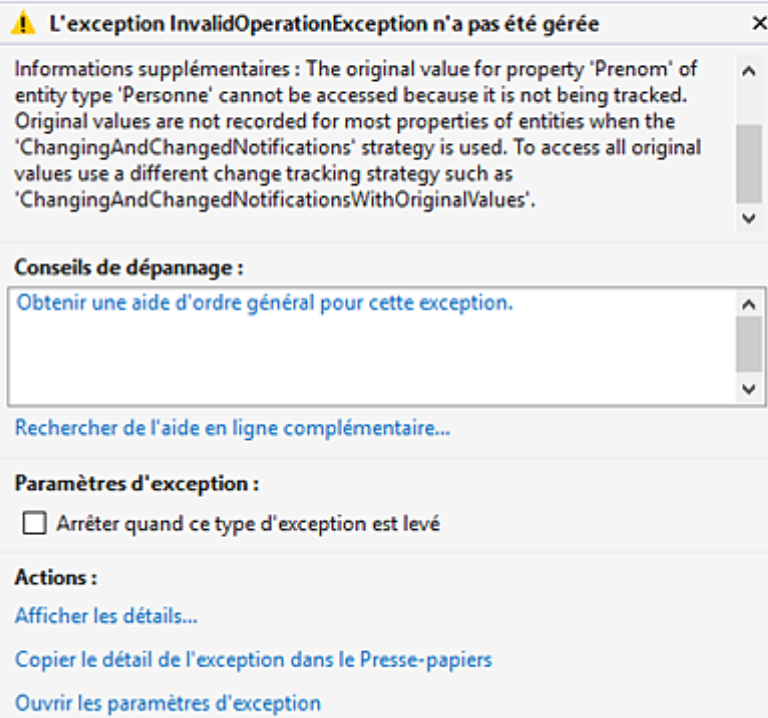
    protected void OnPropertyChanging(string propertyName = null)
    {
        if (this.PropertyChanging != null)
            PropertyChanging(this, new PropertyChangingEventArgs(propertyName));
    }

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

```

En l'état, l'exécution du code écrit plus tôt pour permettre la visualisation des états déclenche très rapidement une exception indiquant que la valeur originale de la propriété `Prenom` n'est pas disponible.

```
var entityEntry = context.ChangeTracker.Entries<Personne>().First();
var propertyEntry = entityEntry.Property<string>(nameof(Personne.Prenom));
Console.WriteLine($"Entité enregistrée - Etat de l'entité : {entityEntry.State}");
Console.WriteLine($"Propriété Prenom - Valeur initiale : {propertyEntry.OriginalValue}");
```



Cette exception est liée au fait qu'avec la stratégie `ChangingAndChangedNotifications`, seuls deux types de propriétés voient leur valeur originale enregistrée par le Change Tracker : les jetons d'accès concurrents et les membres d'une clé étrangère. L'accès à toute autre valeur originale génère une `InvalidOperationException`. Il est donc nécessaire d'épurer quelque peu le code d'exemple afin de vérifier le mode de fonctionnement de la stratégie.

```
using (var context = new ChangeTrackingStrategyContext())
{
    context.ChangeTracker.AutoDetectChangesEnabled = false;

    var personne = new Personne
    {
        Nom = "PUTIER",
        Prenom = "Seb"
    };

    context.Add(personne);
    context.SaveChanges();

    var entityEntry = context.ChangeTracker.Entries<Personne>().First();
    var propertyEntry = entityEntry.Property<string>(nameof(Personne.Prenom));
    Console.WriteLine($"Entité enregistrée - Etat de l'entité : {entityEntry.State}");

    personne.Prenom = "Sébastien";
    Console.WriteLine($"Propriété Prenom enregistrée - Etat de l'entité : {entityEntry.State}");

    context.ChangeTracker.DetectChanges();
}
```

```

        Console.WriteLine($"Détection déclenchée - Etat de l'entité : {entityEntry.State}");

        context.SaveChanges();
        Console.WriteLine($"Entité enregistrée - Etat de l'entité : {entityEntry.State}");
    }

```

À l'exécution, l'affichage montre que le comportement de la stratégie actuelle est identique à celui de la stratégie `ChangedNotifications` : l'entité passe à l'état modifié dès l'assignation d'une nouvelle valeur à la propriété `Prenom`.

```

Entité enregistrée - Etat de l'entité : Unchanged
Propriété Prenom enregistrée - Etat de l'entité : Modified
Détection déclenchée - Etat de l'entité : Modified
Entité enregistrée - Etat de l'entité : Unchanged

```

ChangingAndChangedNotificationsWithOriginalValues

La dernière stratégie, au nom un peu long, est une combinaison de `ChangedNotifications` et `ChangingAndChangedNotifications`. Comme pour ces deux stratégies, la détection de changements est déportée au niveau des 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. Pour répondre à ces besoins, les entités du modèle doivent donc toutes implémenter les interfaces `INotifyPropertyChanged` et `INotifyPropertyChanging`.

L'association de la stratégie avec le modèle est encore une fois la première opération à réaliser.

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasChangeTrackingStrategy(ChangeTrackingStrategy.

    ChangingAndChangedNotificationsWithOriginalValues);

    base.OnModelCreating(modelBuilder);
}

```

Pour vérifier le comportement de cette stratégie, nous allons réutiliser l'implémentation du type `Personne` telle qu'écrite pour la stratégie `ChangingAndChangedNotifications`, ainsi que le code de visualisation des états que nous manipulons depuis l'étude de la stratégie `Snapshot`. À l'exécution, nous retrouvons les éléments attendus : la valeur originale de la propriété `Prenom` ainsi que la modification d'état dès l'assignation d'une nouvelle valeur à cette même propriété.

```
Entité enregistrée - Etat de l'entité : Unchanged
Propriété Prenom - valeur initiale : Seb - valeur Actuelle : Seb

Propriété Prenom enregistrée - Etat de l'entité : Modified
Propriété Prenom - valeur initiale : Seb - valeur Actuelle : Sébastien

Détection déclenchée - Etat de l'entité : Modified
Propriété Prenom - valeur initiale : Seb - valeur Actuelle : Sébastien

Entité enregistrée - Etat de l'entité : Unchanged
Propriété Prenom - valeur initiale : Sébastien - valeur Actuelle : Sébastien
```

3. Attacher et détacher des entités

La manipulation d'entités à travers différents contextes de données peut se révéler indispensable, notamment pour l'implémentation de scénarios d'utilisation déconnectés. Le cas d'entités sérialisées pour la communication entre une application mobile et une API web est de nos jours très courant, et correspond tout à fait à la problématique posée. Ce type de manipulation peut parfaitement être implémenté avec Entity Framework Core à travers les notions d'entités attachées et détachées.

Attacher une entité à un contexte, c'est placer cette entité sous la surveillance du Change Tracker de ce contexte. Une fois l'entité attachée, le Change Tracker est en mesure de vérifier les modifications qui lui sont apportées et de les répercuter au niveau de la base de données.

Chaque entité créée par une requête LINQ est par défaut attachée au contexte qui a exécuté la requête. À partir du moment où le contexte est fermé et si la variable représentant l'entité est toujours accessible (déclarée en dehors du bloc `using`), alors son état relativement à un Change Tracker est indéfini, puisque l'état est une notion que seul le Tracker connaît. On en déduit qu'elle n'est plus attachée au contexte, donc qu'elle est forcément détachée.

Cette entité peut être manipulée au niveau d'une nouvelle instance de contexte de données. Pour cela, on peut utiliser la méthode `DbContext.Attach`, dont l'objectif est justement de placer une entité sous la surveillance de son Change Tracker. Cette opération place toutefois l'entité dans l'état `unchanged`, ce qui est techniquement correct puisque le Change Tracker ne connaît pas l'entité. Mais ce comportement peut être erroné d'un point de vue métier si l'entité a été modifiée avant d'être attachée au contexte. Il est alors nécessaire de passer manuellement son état à la valeur `Modified` pour que le déroulement des opérations soit conforme à ce qui est attendu. L'exemple suivant montre cet enchaînement de changements d'état.

```
Customers client = null;

Console.WriteLine("----- Premier contexte -----");
using (var context = new NorthwindContext())
{
    client = context.Customers
        .First(c => c.CustomerId == "SPECB");
    Console.WriteLine($"Etat (1er contexte) : {context.Entry(client)?.State}");
}

Console.WriteLine("----- Second contexte -----");
using (var context = new NorthwindContext())
{
```

```

Console.WriteLine($"Etat : {context.Entry(client)?.State}");
client.City = "Lyon";

Console.WriteLine($"Etat après modification :
{context.Entry(client)?.State}");

context.Attach(client);
Console.WriteLine($"Etat après attachement :
{context.Entry(client)?.State}");

var entry = context.Entry(client);
entry.State = EntityState.Modified;

Console.WriteLine($"Etat après modification manuelle :
{context.Entry(client)?.State}");
}

```

Ci-après, l'affichage produit par l'exécution de ce fragment de code :

```

----- Premier contexte -----
Etat (1er contexte) : Unchanged
----- Second contexte -----
Etat : Detached
Etat après modification : Detached
Etat après attachement : Unchanged
Etat après modification manuelle : Modified

```

Plus étonnant, la gestion des entités dépendantes. Si une commande est ajoutée à la collection `client.orders` alors que `client` a un statut d'entité détachée, elle est quand même prise en compte par le Change Tracker lors de l'appel à la méthode `DbContext.Attach`. Modifions le code de l'exemple précédent de manière qu'il affiche également l'état d'une commande ajoutée à la collection `orders` de l'entité `client`.

```

Customers client = null;

Console.WriteLine("----- Premier contexte -----");
using (var context = new NorthwindContext())
{
    client = context.Customers.First(c => c.CustomerId == "SPECB");
    Console.WriteLine($"Etat (1er contexte) : {context.Entry(client)?.State}");
}

var commande = new Orders()
{
    OrderDate = DateTime.Today,
    ShipAddress = client.Address,
    ShipCity = client.City,
    ShipCountry = client.Country
};

client.Orders.Add(commande);

Console.WriteLine("----- Second contexte -----");

```

```

using (var context = new NorthwindContext())
{
    Console.WriteLine($"Etat : {context.Entry(client)?.State}");
    Console.WriteLine($"Etat commande : {context.Entry(commande)?.State}");
    client.City = "Lyon";

    Console.WriteLine($"Etat après modification : {context.Entry(client)?.State}");
    Console.WriteLine($"Etat commande : {context.Entry(commande)?.State}");

    context.Attach(client);
    Console.WriteLine($"Etat après attachement : {context.Entry(client)?.State}");
    Console.WriteLine($"Etat commande : {context.Entry(commande)?.State}");

    var entry = context.Entry(client);
    entry.State = EntityState.Modified;

    Console.WriteLine($"Etat après modification manuelle : {context.Entry(client)?.State}");
    Console.WriteLine($"Etat commande : {context.Entry(commande)?.State}");
}

```

On voit clairement à l'affichage que l'état de la commande est `Added` dès lors que le client est de nouveau attaché à un contexte de données.

```

----- Premier contexte -----
Etat (1er contexte) : Unchanged
----- Second contexte -----
Etat : Detached
Etat commande : Detached
Etat après modification : Detached
Etat commande : Detached
Etat après attachement : Unchanged
Etat commande : Added
Etat après modification manuelle : Modified
Etat commande : Added

```

Validation des changements

Si les actions d'ajout, de mise à jour ou de suppression par les méthodes `Add`, `Update` et `Remove` du contexte modifient l'état d'un objet, elles ne sont toutefois d'aucune utilité sans leur répercussion sur la source de données. Cette dernière tâche, la validation des changements, est assurée par la méthode `SaveChanges` de la classe `DbContext`, déjà utilisée à de nombreuses reprises dans la section Cycle de vie des entités.

Malgré sa simplicité apparente, `SaveChanges` déclenche plusieurs traitements de manière à générer et à exécuter les ordres à destination de la source de données. Le premier d'entre eux est le lancement de la détection des changements, lorsque la propriété `AutoDetectChangesEnabled` du Change Tracker a pour valeur `true`.

La génération de requêtes qui s'ensuit s'appuie sur l'état défini pour chaque entité. Lorsqu'une entité est dans l'état `Added`, une requête `INSERT` est créée. Pour l'état `Modified`, c'est une requête `UPDATE` qui est générée, et dans le cas de l'état `Deleted`, c'est une requête `DELETE`. Lorsqu'une entité est dans l'état `Unchanged` ou `Detached`, aucune opération n'est exécutée pour celle-ci.

L'exécution de l'ensemble de ces requêtes est effectuée dans le contexte d'une transaction implicite lorsque la source de données le permet. Si tout se passe bien, la transaction implicite est validée, et l'état des entités est passé à `Unchanged` ou `Detached` afin d'indiquer qu'elles reflètent l'état de la source de données.

Certains cas plus complexes peuvent toutefois se produire, notamment en ce qui concerne les suppressions. En fonction de la configuration de chaque clé étrangère, la librairie peut se retrouver dans une situation de suppression en cascade ou de mise à jour d'enregistrements dépendants. Dans ces deux cas, le traitement est effectué en amont de la suppression d'enregistrement demandée initialement : l'entité dépendante est passée à l'état `Deleted`, ou la valeur `null` est affectée à la clé étrangère.

L'exemple suivant expose les différents types de manipulations et l'effet de la méthode `SaveChanges` vis-à-vis des entités en fonction de leur état.

```
using (var context = new NorthwindContext())
{
    var client = new Customers()
    {
        CustomerId = "PABEL",
        CompanyName = "Pastéis de Belém",
        Address = "84, Rua de Belém",
        PostalCode = "1300 - 085",
        City = "Lisboa",
        Country = "Portugal"
    };

    //Ajout de l'entité client au contexte
    context.Add(client);

    var clientEntityEntry =
        context.ChangeTracker.Entries<Customers>().First();
    Console.WriteLine($"Add() => Etat de l'entité client :
{clientEntityEntry.State}");

    context.SaveChanges();
    Console.WriteLine($"SaveChanges() => Etat de l'entité client :
{clientEntityEntry.State}");

    //Modification de l'entité client et création d'une commande
    client.Phone = "+351 21 363 74 23";
    client.Fax = "+351 21 363 80 78";
    context.Update(client);
    Console.WriteLine($"Update() => Etat de l'entité client :
{clientEntityEntry.State}");

    var commande =
        new Orders
```



```

    {
        CustomerId = client.CustomerId,
        EmployeeId = 1 /* Nancy Davolio */,
        OrderDate = DateTime.Today
    };
    client.Orders.Add(commande);

    var commandeEntityEntry = context.ChangeTracker.Entries<Orders>().First();
    Console.WriteLine($"client.Orders.Add() => Etat de l'entité client :
{clientEntityEntry.State}");
    Console.WriteLine($"                        => Etat de l'entité commande :
{commandeEntityEntry.State}");

    context.SaveChanges();

    Console.WriteLine($"SaveChanges() => Etat de l'entité client :
{clientEntityEntry.State}");
    Console.WriteLine($"                        => Etat de l'entité commande :
{commandeEntityEntry.State}");

    //Suppression du client
    context.Remove(client);

    Console.WriteLine($"Remove() => Etat de l'entité client :
{clientEntityEntry.State}");
    Console.WriteLine($"                        => Etat de l'entité commande :
{commandeEntityEntry.State}");
    context.SaveChanges();

    Console.WriteLine($"SaveChanges() => Etat de l'entité client :
{clientEntityEntry.State}");
    Console.WriteLine($"                        => Etat de l'entité commande :
{commandeEntityEntry.State}");
}

```

L'exécution de ce code produit l'affichage suivant, que nous allons commenter :

```

Add() => Etat de l'entité client : Added
SaveChanges() => Etat de l'entité client : Unchanged
Update() => Etat de l'entité client : Modified
client.Orders.Add() => Etat de l'entité client : Modified
                        => Etat de l'entité commande : Added
SaveChanges() => Etat de l'entité client : Unchanged
                        => Etat de l'entité commande : Unchanged
Remove() => Etat de l'entité client : Deleted
                        => Etat de l'entité commande : Unchanged
SaveChanges() => Etat de l'entité client : Detached
                        => Etat de l'entité commande : Unchanged

```

- Appel de la méthode `Add` : l'entité `client` est mise en état de surveillance par le Change Tracker. Elle est placée par défaut dans l'état `Added`, puisque le contexte ne sait pas si cet objet a une correspondance au niveau de la base de données. Comme nous savons qu'il n'en a pas, tout va bien.

- Premier appel à `SaveChanges` : l'entité `client` est enregistrée dans la source de données à l'aide d'une requête `INSERT`. L'opération ayant réussi, Entity Framework change son état pour refléter la base de données : elle passe à `Unchanged`.
- Appel de la méthode `Update` : l'entité `client` est automatiquement passée à l'état `Modified`.
- Exécution de `client.Orders.Add` : l'entité `commande` est ajoutée à la liste des commandes du client. Par la mécanique des propriétés de navigation, elle est ajoutée au `DbSet Orders`, et est donc mise sous la surveillance du Change Tracker, qui la place à l'état `Added`.
- Deuxième appel à `SaveChanges` : l'entité `client`, marquée comme modifiée, est mise à jour par l'exécution d'une requête SQL `UPDATE`. L'entité `commande` est, quant à elle, insérée dans la table `orders`. Ces deux entités voient leur état passé à `Unchanged` dès que les changements sont répercutés au niveau de la base de données.
- Appel de la méthode `Remove` : l'entité `client` est passée à l'état `Deleted`. L'entité `commande` reste à l'état `Unchanged` car aucune configuration de suppression en cascade n'existe pour cette relation.
- Dernier appel à `SaveChanges` : la suppression de l'entité `client` est validée en base de données, son état est donc passé à `Detached` pour refléter l'état de la source de données. L'enregistrement associé à la commande n'est pas modifié puisque son état avait pour valeur `Unchanged`.

Pour modifier le comportement relatif à la suppression de l'enregistrement `Customers` de sorte que l'enregistrement `Orders` soit supprimé également, il faut configurer la suppression en cascade pour cette relation. Le code ci-après présente en gras l'ajout à apporter à la méthode `OnModelCreating` de `NorthwindContext` pour atteindre ce but.

```
protected override void OnModelCreating(ModelBuilder builder)
{
    // ...

    builder.Entity<Orders>(entity =>
    {
        entity.HasIndex(e => e.CustomerId)
            .HasName("CustomersOrders");

        entity.HasIndex(e => e.EmployeeId)
            .HasName("EmployeesOrders");

        entity.HasIndex(e => e.OrderDate)
            .HasName("OrderDate");

        entity.HasIndex(e => e.ShipPostalCode)
            .HasName("ShipPostalCode");

        entity.HasIndex(e => e.ShipVia)
            .HasName("ShippersOrders");

        entity.HasIndex(e => e.ShippedDate)
            .HasName("ShippedDate");

        entity.Property(e => e.Freight)
            .HasDefaultValueSql("0");
    });
}
```

```

        entity.HasOne(e => e.Customer)
            .WithMany(c => c.Orders)
            .OnDelete(DeleteBehavior.Cascade);
    });

    // ...

}

```

L'affichage produit permet de valider ce comportement : l'entité `commande`, dont l'état après l'appel à `Remove` vaut `Unchanged`, est passée à l'état `Detached` après la validation de cette modification. L'enregistrement correspondant n'existe plus en base de données.

```

Add() => Etat de l'entité client : Added
SaveChanges() => Etat de l'entité client : Unchanged
client.Orders.Add() => Etat de l'entité client : Modified
                    => Etat de l'entité commande : Added
SaveChanges() => Etat de l'entité client : Unchanged
                    => Etat de l'entité commande : Unchanged
Remove() => Etat de l'entité client : Deleted
                    => Etat de l'entité commande : Unchanged
SaveChanges() => Etat de l'entité client : Detached
                    => Etat de l'entité commande : Detached

```

Assurer l'intégrité des données

L'intégrité des données est une notion essentielle pour la totalité des sources de données, qu'elles soient relationnelles, orientées document, basées sur des fichiers texte, ou même tout simplement en mémoire. Certaines d'entre elles, les bases de données relationnelles, intègrent des mécanismes avancés dont l'objectif est uniquement d'assurer la cohérence des informations qu'elles contiennent : c'est notamment le cas des contraintes de clés étrangères. D'autres mécanismes peuvent également être impliqués pour la gestion des accès concurrents ou l'exécution d'ensembles d'instructions sous une forme atomique.

1. Accès concurrents

La problématique de l'accès en écriture à un enregistrement par plusieurs utilisateurs est récurrente dans les environnements client-serveur. Il existe trois manières de procéder pour gérer ce type de cas.

a. Loi du plus fort

Ce mode de fonctionnement représente le comportement par défaut d'Entity Framework Core. Il n'est pas à proprement parler une méthode de gestion des accès concurrents puisqu'il laisse les utilisateurs modifier librement les données avec, pour seule règle, que le dernier qui met à jour un enregistrement gagne, que l'enregistrement ait été modifié par un autre utilisateur ou non. C'est généralement ce comportement que l'on souhaite éviter par l'utilisation des contrôles d'accès concurrents optimistes ou pessimistes.

Pour montrer les effets de ce mode de fonctionnement, le code ci-après simule la mise à jour d'un même enregistrement par deux utilisateurs via deux threads exécutés en parallèle. Le contexte de données utilisé ne comporte qu'un type d'entité et est configuré pour fonctionner avec SQL Server.

```

public class ConcurrencyContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
        Database=ConcurrencyContextDB;Trusted_Connection=True;");
    }

    public DbSet<Personne> Personnes { get; set; }
}

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

    public string Nom { get; set; }

    public string Prenom { get; set; }
}

```

```

//Initialisation de la donnée en base
using (var context = new ConcurrencyContext())
{
    context.Database.EnsureCreated();

    var personne = new Personne()
    {
        Prenom = "Michel",
        Nom = "MARTIN",
    };

    context.Add(personne);
    context.SaveChanges();
}

//Création des deux tâches qui vont s'exécuter en parallèle
var tache1 = new Task(() =>
{
    using (var context = new ConcurrencyContext())
    {
        var personne = context.Personnes.First();
        Console.WriteLine("Tâche 1 : Enregistrement chargé");

        personne.Nom = "DUPOND";

        System.Threading.Thread.Sleep(1000);

        context.SaveChanges();
        Console.WriteLine("Tâche 1 : Michel DUPOND enregistré");
    }
});

var tache2 = new Task(() =>

```

```

{
    using (var context = new ConcurrencyContext())
    {
        var personne = context.Personnes.First();
        Console.WriteLine("Tâche 2 : Enregistrement chargé");

        personne.Nom = "DUPUIS";

        context.SaveChanges();
        Console.WriteLine("Tâche 2 : Michel DUPUIS enregistré");
    }
});

//Exécution des deux tâches
tache1.Start();
tache2.Start();

Task.WaitAll(tache1, tache2);

//Lecture de la donnée en base après exécution des deux tâches
using (var context = new ConcurrencyContext())
{
    var personne = context.Personnes.First();

    Console.WriteLine($"Les deux tâches de MAJ sont terminées.
                        Nom complet : {personne.Prenom} {personne.Nom}");
}

```

L'exécution de ce code produit l'affichage suivant :

```

Tâche 1 : Enregistrement chargé
Tâche 2 : Enregistrement chargé
Tâche 2 : Michel DUPUIS enregistré
Tâche 1 : Michel DUPOND enregistré
Les deux tâches de MAJ sont terminées. Nom complet : Michel DUPOND

```

On remarque clairement que la tâche 1 modifie sans encombre un enregistrement déjà mis à jour par la tâche 2.

Les deux requêtes de mise à jour générées et exécutées par Entity Framework sont construites sur le même modèle. Le seul critère permettant de définir l'enregistrement qui doit être mis à jour est son identifiant. Celui-ci ne variant pas, il n'y a aucun conflit.

```

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

```

b. Contrôle d'accès concurrents optimiste

Le comportement par défaut d'Entity Framework peut rapidement poser des problèmes : les enregistrements sont mis à jour sans contrôle, les utilisateurs ne comprennent pas la disparition des informations qu'ils ont pourtant enregistrées, les développeurs s'arrachent les cheveux.

Entity Framework intègre la possibilité d'encadrer les manipulations de données de manière optimiste : on table sur le fait que les utilisateurs ne vont pas modifier le même enregistrement simultanément, et si ce cas se produit, une exception est levée. L'implémentation de ce comportement est assurée par deux éléments qui peuvent être utilisés séparément : les jetons d'accès concurrents (concurrency tokens) et l'horodatage.

Jetons d'accès concurrents

Les jetons d'accès concurrents sont des propriétés définies comme des valeurs à vérifier lors de la mise à jour d'un enregistrement. Ils servent de points de repère : lorsqu'un jeton d'accès concurrent est modifié, l'enregistrement est considéré comme modifié. Ainsi, lors d'accès simultanés, la seconde validation met un jour un enregistrement qui n'existe plus dans son état initial et une exception est levée.

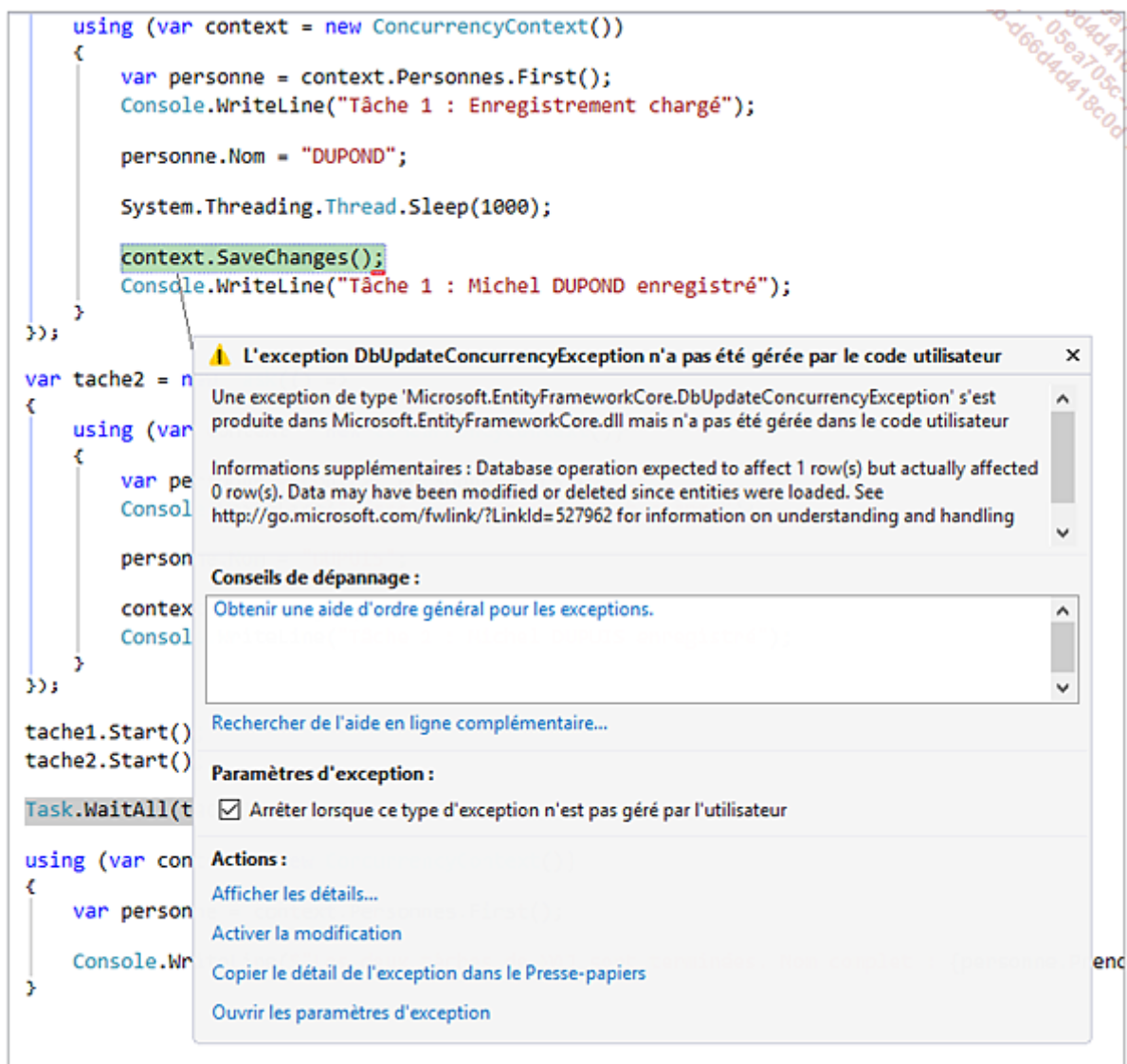
Le rôle de jeton d'accès concurrent est attribué à une propriété par le biais de la data annotation `ConcurrencyCheck`.

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

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

    public string Prenom { get; set; }
}
```

Le code exécuté précédemment se comporte maintenant de manière bien différente, puisqu'une `DbUpdateConcurrencyException` est levée par l'instruction `SaveChanges` de la tâche 1.



Le message indique qu'aucune ligne n'a été mise à jour par la requête SQL générée. Ceci s'explique par le fait que l'instruction `UPDATE` contient une clause `WHERE` quelque peu modifiée : elle tient maintenant compte de la valeur du champ `Nom` qui est identifié comme jeton d'accès concurrent dans le modèle objet.

```

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

```

Comme toutes les configurations de propriétés, celle-ci peut être réalisée à l'aide de la Fluent API. La fonction dédiée à cet usage est définie sur le type `PropertyBuilder` et se nomme `IsConcurrencyToken`. Dans notre modèle, pour indiquer que la propriété `Nom` est un jeton d'accès concurrent, la méthode `OnModelCreating` doit être implémentée de la manière suivante.

```

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

    base.OnModelCreating(builder);
}

```

Horodatage

La technique de l'horodatage intègre à chaque enregistrement un champ dont le contenu est la date de la dernière modification portant sur ses données. Ce champ est un cas particulier de jeton d'accès concurrent dont la valeur est générée automatiquement lors de l'ajout ou de la mise à jour d'un enregistrement.

Dans les faits, la colonne définie pour l'horodatage n'a aucune obligation de contenir un timestamp (identificateur de la date de modification) puisque l'implémentation de ce comportement est déléguée aux différents fournisseurs de données. Il pourrait tout à fait être associé à une valeur entière unique à l'échelle d'une base de données, par exemple. Pour le fournisseur SQL Server, le type de données .NET utilisé habituellement pour les champs d'horodatage est `byte[]`. Dans ce contexte, il est associé au niveau de la base de données à une colonne dont le type est `rowversion` (qui remplace la syntaxe dépréciée du type `timestamp`).

Les propriétés utilisées pour l'horodatage sont décorées d'un attribut `Timestamp`.

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

    public string Nom { get; set; }

    public string Prenom { get; set; }

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

L'exécution du code de mise à jour concurrente vu précédemment produit exactement le même résultat que lors de la définition du champ `Nom` comme jeton d'accès concurrent. Une exception de type `DbUpdateConcurrencyException` est déclenchée à la validation des changements effectuée par la tâche 1.

La requête de mise à jour générée est légèrement plus complexe que celle que nous avons vue avec un jeton d'accès concurrent simple puisqu'elle tire parti de fonctionnalités spécifiques au moteur SQL Server. Pourtant, le principe de base est identique puisque la clause `WHERE` se base sur une égalité entre le timestamp passé en paramètre et celui qui est associé à l'enregistrement recherché pour valider la mise à jour. Le code supplémentaire permet de récupérer la valeur générée automatiquement par la base de données pour le champ `JetonHorodatage` afin qu'Entity Framework actualise la valeur de ce jeton dans l'entité .NET.

```
DECLARE @inserted0 TABLE ([JetonHorodatage] varbinary(8));
UPDATE [Personnes] SET [Nom] = @p0
OUTPUT INSERTED.[JetonHorodatage]
INTO @inserted0
WHERE [Id] = @p1 AND [JetonHorodatage] = @p2;
SELECT [JetonHorodatage] FROM @inserted0;
```


Pour qualifier une propriété de jeton d'horodatage à l'aide de la Fluent API, on utilise deux méthodes : `IsConcurrencyToken`, que nous avons vue précédemment, et `ValueGeneratedOnAddOrUpdate`, qui indique à Entity Framework que la génération de la valeur de timestamp est déléguée à la base de données.

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

c. Contrôle d'accès concurrents pessimiste

Ce mode de gestion des accès concurrents consiste à verrouiller les enregistrements de la source de données qui sont en cours de mise à jour, de manière qu'ils ne puissent pas être manipulés par plusieurs utilisateurs simultanément. Cette technique prévient effectivement 100 % des conflits, mais elle nécessite généralement de maintenir une connexion à la source de données pendant toute la durée de la manipulation des données. De plus, son fonctionnement bloquant entrave la bonne marche de l'application puisque les enregistrements sont verrouillés pour une durée indéterminée : les utilisateurs en attente sont dans une position particulièrement gênante. Enfin, chaque verrouillage doit systématiquement aller de pair avec un déverrouillage, ce qui peut ne jamais arriver. L'enregistrement est dans ce cas dans un état de deadlock (cadenassage) et ne peut être déverrouillé que par un administrateur de base de données. La gestion pessimiste a donc un avantage non négligeable sur les autres modes de fonctionnement, mais impose un certain nombre de contraintes à prendre en compte pendant toute la vie de l'application.

Avec l'avènement de l'ère de la mobilité, la gestion pessimiste a atteint ses limites et peut être considérée comme obsolète, même si certains cas sont encore adaptés à son utilisation. Le contrôle d'accès concurrents pessimiste n'est donc pas utilisable avec Entity Framework Core.

2. Transactions

Certaines sources de données, notamment les bases de données relationnelles, autorisent le groupement de manipulations de données en un seul ensemble dont l'exécution est effectuée de manière atomique : une transaction.

Le principe de fonctionnement d'une transaction est relativement simple. Une transaction encapsule un ensemble d'opérations qui doivent être exécutées. Lorsque toutes ces opérations sont effectuées avec succès, la transaction est validée. À la moindre erreur déclenchée dans sa portée, la transaction est interrompue et toutes les modifications effectuées sous sa surveillance sont annulées.

Entity Framework Core inclut la gestion des transactions sous deux formes : implicites et explicites.

a. Implicites

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.

Pour vérifier le fonctionnement de ces transactions, la portion de code suivante tente d'insérer deux clients dans la table `Customers`. Le second client a pour clé primaire "FRANS", qui est déjà clé primaire d'un enregistrement : la contrainte non respectée déclenche une exception à l'insertion du second enregistrement. Une erreur s'étant produite dans la portée d'une transaction, celle-ci annule les modifications qui ont pu être faites sous sa surveillance. En l'occurrence, le premier enregistrement créé ne sera pas maintenu dans la table `Customers`.

```
using (var context = new NorthwindContext())
{
    var client1 = new Customers()
    {
        CustomerId = "CARAL",
        CompanyName = "Caramelyon"
    };

    var client2 = new Customers()
    {
        CustomerId = "FRANS",
        CompanyName = "France Saucisson"
    };

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

    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Une erreur s'est produite pendant l'enregistrement des données");
        Console.WriteLine(ex.InnerException.Message);
    }
}

using (var context = new NorthwindContext())
{
    var nbClients_CARAL =
        context.Customers.Count(
            client => client.CustomerId == "CARAL");
    Console.WriteLine($"Nombre de clients avec Id = CARAL : {nbClients_CARAL}");
}
```

L'affichage produit en sortie confirme l'explication : le client Caramelyon, ayant pour valeur de clé primaire "CARAL", n'est pas enregistré dans la base de données.

```
Une erreur s'est produite pendant l'enregistrement des données
Violation of PRIMARY KEY constraint 'PK_Customers'. Cannot insert
duplicate key in object 'dbo.Customers'. The duplicate key value
is (FRANS).
The statement has been terminated.
Nombre de clients avec Id = CARAL : 0
```

b. Explicites

Les transactions peuvent être utilisées de manière explicite de façon à pouvoir encapsuler plusieurs validations de changements dans une seule transaction. Elles sont représentées dans Entity Framework Core par le type `IDbContextTransaction` dont l'implémentation est spécifique à chaque fournisseur de données. Pour cette raison, la création d'une transaction n'est pas effectuée par l'utilisation d'un constructeur, mais par un appel à la fonction `BeginTransaction` exposée à travers la propriété `DbContext.Database`. L'objet retourné implémente l'interface `IDisposable`, c'est pourquoi cette méthode est généralement utilisée en conjonction avec la structure C# `using`.

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

Les différentes opérations qui doivent être exécutées dans le contexte de cette transaction sont ensuite incorporées au sein de ce bloc `using`. À la suite de ces instructions, toujours dans ce bloc, il faut valider explicitement la transaction à l'aide de sa méthode `Commit`. L'annulation de la transaction est optionnelle : lorsque l'objet `IDbContextTransaction` est nettoyé par l'appel de sa méthode `Dispose`, la méthode `Rollback` est appelée automatiquement si aucun commit n'a été effectué.

Reprenons le code de l'exemple précédent pour inclure l'utilisation d'une 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();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Une erreur s'est produite pendant l'enregistrement des données");
        Console.WriteLine(ex.InnerException.Message);
    }
}
```

```

    }
}

using (var context = new NorthwindContext())
{
    var nbClients_CARAL =
        context.Customers.Count(client => client.CustomerId == "CARAL");
    Console.WriteLine($"Nombre de clients avec Id = CARAL : {nbClients_CARAL}");
}

```

Un appel à la méthode `Rollback` de la transaction peut être inséré dans le bloc `catch`, de manière à rendre ce comportement explicite.

```

...
catch (Exception ex)
{
    Console.WriteLine("Une erreur s'est produite pendant
l'enregistrement des données");
    Console.WriteLine(ex.InnerException.Message);

    transaction.Rollback();
}

```

L'exécution de ce code produit exactement le même affichage que lors de l'utilisation d'une transaction implicite, et ce malgré l'appel réussi à `context.SaveChanges` pour le client Caramelyon.

```

Une erreur s'est produite pendant l'enregistrement des données
Violation of PRIMARY KEY constraint 'PK_Customers'. Cannot insert
duplicate key in object 'dbo.Customers'. The duplicate key value
is (FRANS).
The statement has been terminated.
Nombre de clients avec Id = CARAL : 0

```

Visualisation du code SQL généré

Un des reproches régulièrement faits aux précédentes versions d'Entity Framework est la difficulté d'obtenir les instructions SQL générées par le moteur. Cette problématique est maintenant réglée puisque Entity Framework Core intègre un point d'entrée accessible simplement pour l'ajout d'une fonctionnalité de log. Il s'agit d'un service enregistré dans le cœur du moteur et implémentant l'interface `ILoggerFactory`. Il permet de renvoyer des instances d'objets implémentant l'interface `ILogger`, et dont le but est l'enregistrement d'informations liées au déroulement de l'exécution du code d'Entity Framework Core.

Le patron de conception Factory permet de centraliser les instantiations d'un ou de plusieurs types d'objets au sein d'une classe unique. Il est très utilisé au sein de la librairie Entity Framework Core pour tirer parti au maximum du mécanisme d'injection de dépendances.

Entre la factory et l'objet `ILogger`, il existe un intermédiaire : le fournisseur d'objets `ILogger`. Cet intermédiaire est utilisé pour pouvoir associer plusieurs types concrets implémentant `ILogger` à l'unique instance d'`ILoggerFactory` maintenue en mémoire par Entity Framework Core. Le fournisseur implémente l'interface `ILoggerProvider` et est enregistré dans la factory par un

appel à la méthode `ILoggerFactory.AddProvider`. Cet enregistrement n'est effectué qu'une fois, généralement au démarrage de l'application, car la factory est un singleton conservé en mémoire tout au long de l'exécution. Il est partagé par tous les objets `DbContext` (qui peuvent être de type différent !) utilisés pendant la vie de l'application.

La première étape est la création d'un type implémentant le type `ILogger`, puis il faut créer un type implémentant `ILoggerProvider`, et enfin, il faut enregistrer ce type au niveau du service `ILoggerFactory`. Ce service est récupéré à partir d'un contexte de données instancié. Le code ci-après montre ces différentes étapes pour la mise en place d'un logger qui affiche dans la console les informations remontées par Entity Framework Core.

Ici, le choix est fait d'imbriquer le type `MyLogger` dans `MyLoggerProvider` en tant que classe privée pour éviter les instantiations directes.

```
public class MyLoggerProvider : ILoggerProvider
{
    public ILogger CreateLogger(string categoryName)
    {
        return new MyLogger();
    }

    public void Dispose()
    { }

    private class MyLogger : ILogger
    {
        public bool IsEnabled(LogLevel logLevel)
        {
            return true;
        }

        public void Log<TState>(LogLevel logLevel, EventId eventId,
                                TState state, Exception exception,
                                Func<TState, Exception, string> formatter)
        {
            Console.WriteLine(formatter(state, exception));
        }

        public IDisposable BeginScope<TState>(TState state)
        {
            return null;
        }
    }
}
```

Le code suivant est exécuté une seule et unique fois, au démarrage de l'application.

```
using (var db = new NorthwindContext())
{
    var serviceProvider = db.GetInfrastructure<IServiceProvider>();
    var loggerFactory = serviceProvider.GetService<ILoggerFactory>();
    loggerFactory.AddProvider(new MyLoggerProvider());
}
```

Lors de l'exécution d'une requête LINQ (entre autres), de nombreuses informations devraient être affichées dans la console. Ci-après, l'affichage réalisé lors de l'exécution du premier exemple concernant l'opérateur Select :

```
using (NorthwindContext context = new NorthwindContext())
{
    var requete = context.Customers
                        .Select((client) => new
                        {
                            Nom = client.ContactName,
                            Pays = client.Country
                        });

    //Affichage du nom et du pays de chaque client
    foreach (var resultat in requete)
    {
        Console.WriteLine($"{resultat.Nom} - {resultat.Pays}");
    }
}
```

```
Compiling query model:
'from Customers client in DbSet<Customers>
select new <>f__AnonymousType3<string, string>(
    client.ContactName,
    client.Country
)'
Optimized query model:
'from Customers client in DbSet<Customers>
select new <>f__AnonymousType3<string, string>(
    client.ContactName,
    client.Country
)'
TRACKED: False
(QueryContext queryContext) =>
IEnumerable<<>f__AnonymousType3<string, string>>
_Select(
    source: IEnumerable<ValueBuffer> _ShapedQuery(
        queryContext: queryContext,
        shaperCommandContext: SelectExpression:
            SELECT [client].[ContactName], [client].[Country]
            FROM [Customers] AS [client]
        ,
        shaper: ValueBufferShaper
    )
    ,
    selector: (ValueBuffer client) =>
new <>f__AnonymousType3<string, string>(
    try { (string) object client.get_Item(0) } catch
(Exception) { ... } ,
    try { (string) object client.get_Item(1) } catch
(Exception) { ... }
)
```

```
)

Opening connection to database 'Northwind' on
server '(localdb)\mssqllocaldb'.
Executed DbCommand (48ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
SELECT [client].[ContactName], [client].[Country]
FROM [Customers] AS [client]
Maria Anders - Germany

// ... Ici, il y a 89 autres clients.

Zbyszek Piestrzeniewicz - Poland
Closing connection to database 'Northwind' on
server '(localdb)\mssqllocaldb'.
```

La sortie produit donc des informations relatives à la transcription de la requête LINQ en langage SQL, la requête SQL générée elle-même, mais aussi des informations concernant la connexion à la source de données.