

# **Mappage objet-relationnel**

# Dictionnaire de données

---

Le **dictionnaire des données** permet, sous réserve de disposer des droits nécessaires, d'accéder aux informations concernant les bases de données hébergées par un serveur.

```
string conStr = "Provider=SQLOLEDB;Data Source=
                INFO-DORMEUR;Uid=ETD;Pwd=ETD";
OleDbConnection dbCon = new OleDbConnection(conStr);
dbCon.Open();
    // liste des bases
DataTable database =
    dbCon.GetOleDbSchemaTable(OleDbSchemaGuid.Catalogs,
        null);
    // liste des tables
DataTable tables =
    dbCon.GetOleDbSchemaTable(OleDbSchemaGuid.Tables,
        new object[] {null, null, null, "TABLE" });
    // etc. (plus d'information sur Moodle...)
```

# Accès à la base en C#

---

Nous avons créé une application **Windows Form**, connectée à la base `MusiqueSQL`, et utilisant les tables `Musicien`, `Composer` et `Oeuvre`.

Pour effectuer le lien (**mappage**) entre les tables et notre application, nous avons créé deux classes, `Musicien` et `Oeuvre`, grâce auxquelles nous avons pu « convertir » des lignes de tables en objets...

```
while (reader.Read())  
{  
    int id = Convert.ToInt32(reader.GetInt32(0));  
    string nom = reader.GetString(1);  
    [...]  
    Musicien m = new Musicien(id, nom, prénom);  
    listBox1.Items.Add(m);  
}
```

# Entity Framework (EF)

---

Le module **Entity Framework** de **ADO.NET**, introduit avec la version 3.5 de **.Net**, permet d'automatiser ce mappage entre base de données relationnelle et application orientée objet.

Il est intégré comme un outil de Visual Studio : menu **Projet/Ajouter un nouvel élément/ADO.NET Entity Data model**, puis sélection de la base, des tables, des vues ou des procédures stockées.

Cela génère **une classe partielle** par table sélectionnée (en principe...), et une classe de gestion du mappage (fichier suffixé par `.Designer.cs`) associé à un fichier suffixé par `.edmx` qui décrit le schéma relationnel de la base.

# La classe partielle Musicien générée

---

Avec import de **Musicien**, **Oeuvre** et **Composer** :

```
public partial class Musicien    {  
    public Musicien()    {  
        this.Oeuvre = new HashSet<Oeuvre>();    }  
    public int Code_Musicien { get; set; }  
    public string Nom_Musicien { get; set; }  
    public string Prénom_Musicien { get; set; }  
    public Nullable<int> Année_Naissance { get; set; }  
    public Nullable<int> Année_Mort { get; set; }  
    public Nullable<int> Code_Pays { get; set; }  
    public byte[] Photo { get; set; }  
    public virtual ICollection<Oeuvre> Oeuvre  
        { get; set; }  
}
```

# LINQ (to SQL) : accès à la base

---

**LINQ** (**L**anguage **I**ntegrated **Q**uery) est un composant **.NET** qui étend **C#** (et d'autres langages .NET) en offrant la possibilité d'utiliser une syntaxe « de type **SQL** » (**Select**) pour interroger toutes sortes de structures de données.

```
// Les trois parties d'une requête LINQ :  
// 1. Source de données.  
// Généralement une table (ou jointure)  
// 2. Création de la requête.  
// renvoie un System.Data.Objects.ObjectQuery  
var musiciensXX = from m in musiqueSQL.Musicien  
                  where m.Année_Naissance > 1900  
                  select m;  
// 3. Récupération/traitement du résultat de la requête.  
foreach (Musicien m in musiciensXX) {  
    Console.Write(m.Nom_Musicien + ", ");  
}
```

# LINQ (to SQL) : autre exemple

---

Récupération d'un genre à partir de son libellé (genre) :

```
string genre = Console.ReadLine() ;  
var gen = from g in musiqueSQL.Genre  
         where g.Libellé_Abrégé == genre  
         select g;
```

Récupération de tous les albums du genre ainsi sélectionné :

```
foreach (Album alb in gen.First().Album)  
{  
    string s = alb.Titre_Album;  
    // traitement...  
}
```

# Liste des musiciens : avant / après... (1)

---

## Avant :

```
OleDbConnection connection = new OleDbConnection();
connection.ConnectionString = "....";
connection.Open();
string SQL =
    "Select Code_Musicien, Nom_Musicien from Musicien;";
OleDbCommand comm = new OleDbCommand(SQL, connection);
OleDbDataReader reader = comm.ExecuteReader();
while (reader.Read())
{
    Musicien m = new Musicien();
    m.Code_Musicien = reader.GetInt(0);
    m.Nom_Musicien = reader.GetString(1);
    Console.WriteLine(m.Code_Musicien.ToString() + " " +
        m.Nom_Musicien);
}
```



## Liste des musiciens : avant / après... (2)

---

### Après :

```
MusiqueSQLEntities musiqueSQL = new MusiqueSQLEntities();  
var lesmusiciens = from m in musiqueSQL.Musicien  
                    orderby m.Nom_Musicien  
                    select m;  
foreach (Musicien m in lesmusiciens) {  
    Console.WriteLine(m.Code_Musicien.ToString() + " " +  
        m.Nom_Musicien);  
}
```

**Avantages.** La syntaxe est vérifiée **dès la compilation**, et non lors de l'accès au serveur, à l'exécution, pour une requête SQL de type String (OleDb).

De plus, on peut « parcourir » l'ensemble des musiciens sans gérer soi-même une liste.

# LINQ : quelques compléments

---

## Récupérer uniquement certaines colonnes :

```
var lesNomsPrénoms = from m in musiqueSQL.Musicien
    select new { n = Nom_Musicien, p = Prénom_Musicien } ;
foreach (var m in lesNomsPrénoms) {
    Console.WriteLine(m.n + " " + m.p); }
```

## Récupérer uniquement certaines lignes :

```
var lesMusiciensXX = from m in musiqueSQL.Musicien
    where m.Année_Naissance > 1900 select m;
```

## Jointures :

```
var lesNomsEtPays = from m in musiqueSQL.Musicien
    join p in musiqueSQL.Pays
    on m.Code_Pays equals p.Code_Pays
    select new {nom = m.Nom_Musicien, pays = p.Nom_Pays};
```

# Opérations de mise à jour

---

Les **modifications** de la base de données (INSERT, UPDATE et DELETE) se feront par des modifications classiques des structures de données internes (modification d'un objet, création d'un nouvel objet ou suppression).

La **validation** des modifications effectuées se fera grâce à la méthode `SaveChanges` du gestionnaire de données :

```
musiqueSQL.SaveChanges();
```

**Attention.** *Cette méthode ne prend pas en charge l'ensemble des validations (contraintes référentielles, destruction en cascade...) qui devront être gérées par le programmeur lors de la construction ou de la suppression des objets. Exécution de type tout ou rien : toutes les opérations réussissent ou échouent...*

## Et maintenant ?...

---

*Proposez une nouvelle version  
de votre application (gestion  
des deux listbox) utilisant  
Entity Framework et LINQ...*

*Quelques indications pour bien démarrer...*

# Configurer le projet EF + LINQ

---

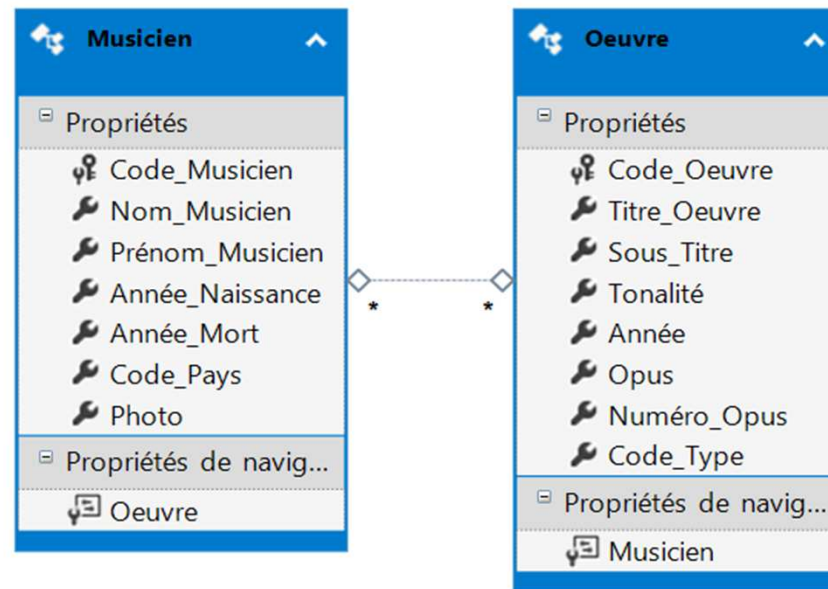
(1)

1. Dans Visual Studio, créer un nouveau projet **Application Windows Forms (.NET Framework) C#**.
2. Menu **Projet / Ajouter un nouvel élément...** puis, dans le volet de gauche, choisir **Éléments Visual C# / Données** et, dans le volet central, choisir **ADO.NET Entity Data Model**, puis cliquer sur **Ajouter**.
3. Choisir **EF Designer à partir de la base de données**, puis **Suivant**.
4. Cliquer sur **Nouvelle connexion...**, puis Source de données : **Microsoft SQL Server**, **Continuer**, Nom du serveur : *info-dormeur*, Authentification : **SQL Server**, **ETD/ETD**, Nom de la base de données : **MusiqueSQL**, puis **OK**.
5. Cocher **Oui, inclure les données sensibles de connexion**.
6. Choisir **Entity Framework 6.x**.
7. Dérouler **Tables, dbo**, puis choisir les tables **Musicien**, **Composer** et **Œuvre** puis bouton **Terminer**. (Répondre **OK** aux messages « Avertissement de sécurité ».)

# Configurer le projet EF + LINQ

(2)

7. Examiner le schéma généré (fichier **Model1.edmx** par défaut) :



8. Examiner les classes **Musicien** et **Oeuvre** générées... Comprenez-vous le rôle de l'attribut **oeuvre** dans la classe **Musicien** ?

**Note.** Un **clic droit** dans la fenêtre du schéma généré (puis **mettre à jour le modèle à partir de la base de données**) permet d'importer si besoin des tables oubliées...