



ADO.NET

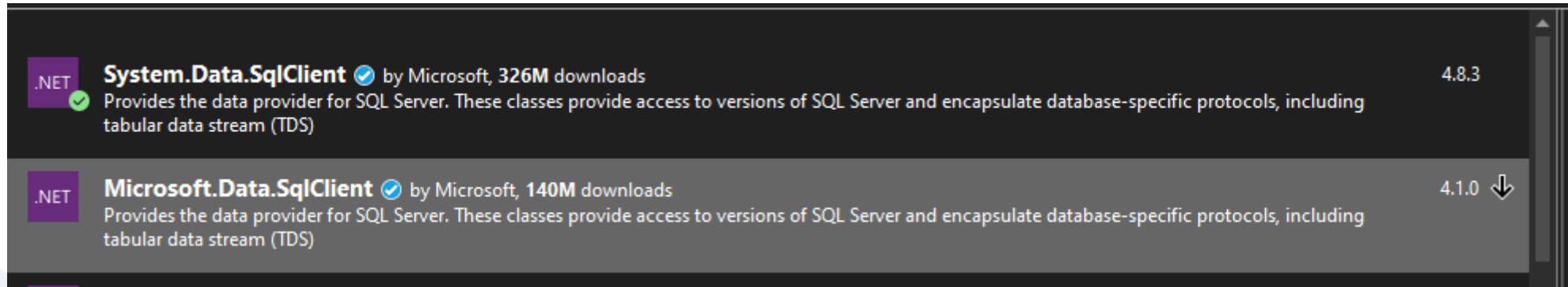
Manipulation des Données en C#

L'Accès aux données

Pour accéder aux données en usant d'ADO.NET, il faut tout d'abord utiliser le Package NuGet SqlClient, qui peut alors soit être :

- **System.Data.SqlClient**
- **Microsoft.Data.SqlClient**

Le choix se pose généralement sur la date de la dernière modification du package ou sur les préférences personnelles.



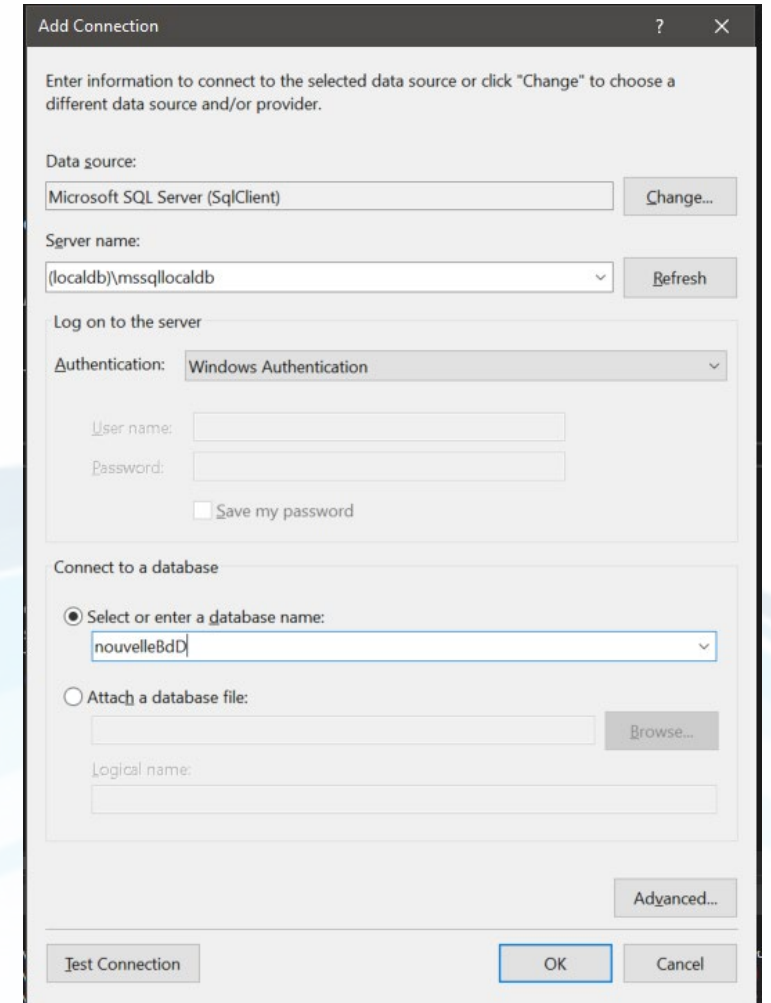
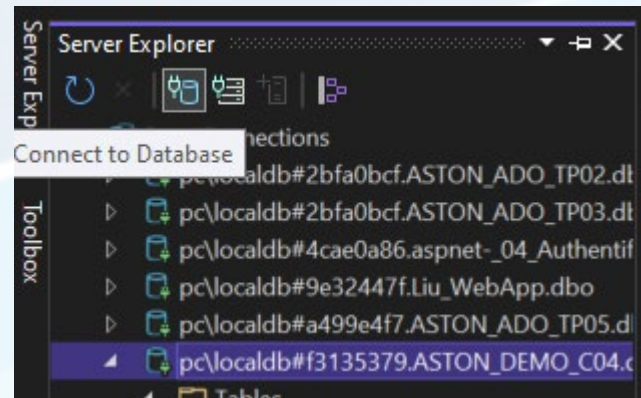
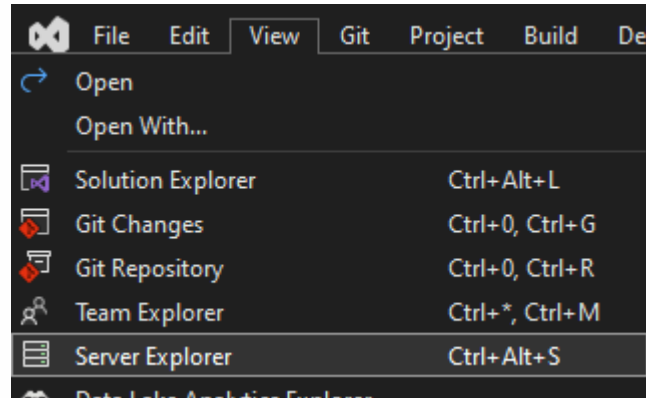
```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

*Ne pas oublier le **using***

Créer une base de données

Pour créer une base de données, rien de plus simple, il suffit de se servir de l'instance locale **MSSQLLocalDB**.

On peut accéder au serveur en passant par l'explorateur des serveurs (**CTRL+ALT+S**), puis il suffit de créer une nouvelle connexion vers la base de données locale **SQL Server Express**



SqlConnection

Une fois le package ajouté au projet, il faudra créer un objet de type *SqlConnection* afin d'accéder aux données. Pour créer un objet de ce type, il faut disposer d'une chaîne de connexion. Cette chaîne est trouvable facilement en observant les propriétés de notre base de donnée.

Il faut faire attention lors de l'utilisation d'une connexion SQL, et il vaut mieux entourer les différentes expressions d'un *try...catch...finally* dans le but d'être sûr d'avoir une connexion fermée à l'issue des différentes manipulations.

```
private string connectionString = @"Data Source=(localdb)\mssqllocaldb;Initial Catalog=ASTON_DEMO_C04;Integrated Security=True";  
1 reference
```

```
SqlConnection connection = new SqlConnection(@"Data Source=(localdb)\mssqllocaldb;Initial Catalog=ASTON_DEMO_C04;"  
    + "Integrated Security=True");
```

```
SqlConnection connection = new SqlConnection(connectionString);
```

Récupération de la Connection String

[-] Identity	
(Name)	ASTON_DEMO_C04
[-] Connection	
Connection String	Data Source=(localdb)\mssqllo
Provider	.NET Framework Data Provider
State	Open

SqlCommand

La **SqlCommand** est une classe permettant d'accéder à des commandes SQL et de sécuriser ces dernières via l'ajout de **Parameters**, tel que :

```
SqlCommand cmd = connection.CreateCommand();

cmd.CommandText = "INSERT INTO dogs (Name, CollarColor, NbrOfLegs, MasterId) OUTPUT INSERTED.Id "
    + "VALUES (@name, @collarColor, @nbrOfLegs, @masterId)";
cmd.Parameters.Add(new SqlParameter("@name", dogName));
cmd.Parameters.Add(new SqlParameter("@collarColor", dogCollarColor));
cmd.Parameters.Add(new SqlParameter("@nbrOfLegs", dogNbrOfLegs));
cmd.Parameters.Add(new SqlParameter("@masterId", dogMasterId));
```

La commande est créée à partir de la connexion, et possèdera plusieurs paramètres en fonction des besoin. Attention, ces **variables** devront être uniques, sous peine de lever une exception. De plus, il est recommandé de **libérer** la commande une fois cette dernière utilisée.

```
cmd.Dispose();
```


ExecuteNonQuery

La **SqlCommand** peut ensuite être envoyée en BdD sans retour particulier de paramètres sauf le nombre de lignes modifiées.

Pour cela, on a recour à *ExecuteNonQuery()*, comme ci-dessous :

```
Console.WriteLine("Table des chiens introuvable ! Création...");
cmd2.CommandText = "CREATE TABLE [dbo].[dogs] ("
    + "[Id] INT NOT NULL PRIMARY KEY IDENTITY (1,1),"
    + "[Name] NVARCHAR(50) NOT NULL,"
    + "[CollarColor] NVARCHAR(50) NOT NULL,"
    + "[NbrOfLegs] INT NOT NULL,"
    + "[Master] INT NOT NULL,"
    + "PRIMARY KEY CLUSTERED ([Id] ASC),"
    + "CONSTRAINT[FK_dogs_masters] FOREIGN KEY([MasterId]) REFERENCES[dbo].[masters]([Id])"
    + ")";

int nbRows = cmd2.ExecuteNonQuery();
cmd2.Dispose();
Console.WriteLine("Table des chiens créée !");
```

ExecuteNonQuery va renvoyer un nombre de lignes affectées, qu'il est généralement utile de stocker dans une variables en vue de la réalisation d'une condition basée sur la valeur de cette dernière.

ExecuteScalar

La **SqlCommand** peut aussi être utilisée via **ExecuteScalar()** dans le but de récupérer une seule valeur obtenue, comme par exemple l'Id ajoutée :

```
cmd.CommandText = "INSERT INTO dogs (Name, CollarColor, NbrOfLegs, MasterId) OUTPUT INSERTED.Id "
+ "VALUES (@name, @collarColor, @nbrOfLegs, @masterId)";
cmd.Parameters.Add(new SqlParameter("@name", dogName));
cmd.Parameters.Add(new SqlParameter("@collarColor", dogCollarColor));
cmd.Parameters.Add(new SqlParameter("@nbrOfLegs", dogNbrOfLegs));
cmd.Parameters.Add(new SqlParameter("@masterId", dogMasterId));

try
{
    connection.Open();

    if (connection.State == ConnectionState.Open)
    {
        int id = (int) cmd.ExecuteScalar();

        cmd.Dispose();

        if (id != 0) Console.WriteLine($"Le chien a été ajouté ! ID : {id}");
    }
}
```

ExecuteScalar va renvoyer la valeur de la première colonne du résultat, qu'il faudra caster sous la forme d'un Int pour obtenir Ici la valeur de l'Id sous peine d'avoir un problème de typage.

ExecuteReader

Enfin, la **SqlCommand** peut servir à obtenir une ou plusieurs lignes de multiples valeurs depuis la BdD via l'utilisation de **ExecuteReader()** qui va renvoyer un **DataReader**, sur lequel on peut ensuite itérer ou non en fonction du nombre de lignes souhaitées :

```
SqlConnection connection = new SqlConnection(connectionString);
SqlCommand cmd = connection.CreateCommand();
cmd.CommandText = "SELECT dogs.Id, Name, CollarColor, NbrOfLegs, masters.FirstName, masters.LastName"
    + "FROM dogs INNER JOIN masters ON dogs.MasterId = masters.Id;";

try
{
    connection.Open();

    if (connection.State == ConnectionState.Open)
    {
        SqlDataReader reader = cmd.ExecuteReader();
        cmd.Dispose();
        while (reader.Read())
        {
            Console.WriteLine($"{reader.GetInt32(0)}. Nom : {reader.GetString(1)}, Couleur du collier : {reader.GetString(2)}, "
                + "Nombre de pattes : {reader.GetInt32(3)}, Maître : {reader.GetString(4)} {reader.GetString(5)}");
        }
        reader.Close();
    }
}
```

Attention, il est nécessaire de **fermer le DataReader** sous peine d'avoir des problèmes de **leak**.

Les Datasets

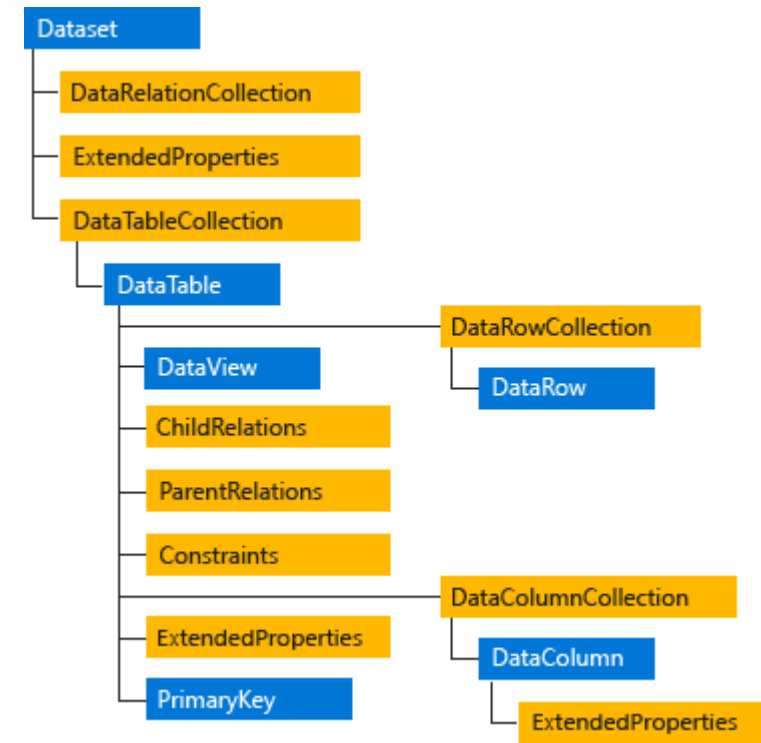
Un **Dataset** est en réalité une classe du C# servant à représenter une Base de données.

Il est en effet constitué de multiples **DataTables**, elles mêmes constituées de **DataRows** qui représentent les multiples lignes des données. Il est également possible d'accéder à des **DataColumns** et à des **DataView** si besoin.

L'utilisation de Datasets permet de travailler de façon déconnectée de la Base de données dans le but d'alléger la charge de cette dernière. Pour ce faire, il faut Remplir le Dataset au lancement de l'application et modifier la source des données à la fermeture de l'application.

Pour ce faire, nous allons avoir besoin de un ou de plusieurs **SqlDataAdapter** qui serviront à établir la jonction entre la source et notre programme en se basant sur les tables que l'on a déclaré au préalable.

```
_dogDataset = new DataSet("dogs");  
_dogDataset.Tables.Add(new DataTable("dogs"));  
_dogDataset.Tables.Add(new DataTable("masters"));
```



Les SqlDataAdapter

Le SqlDataAdapter est une classe servant à faire la jointure entre la source de données et les Datasets. Elle doit pour cela avoir accès à plusieurs requêtes alimentées par des commandes SQL sous la forme de chaîne de caractère, et des paramètres demandés après en avoir spécifié le typage.

```
SqlCommand cmd = _sqlConnection.CreateCommand();  
cmd.CommandText = "SELECT Id, Name, CollarColor, NbrOfLegs, MasterId FROM dogs;";  
_dogAdapter.SelectCommand = cmd;  
_dogAdapter.Fill(_dogDataset.Tables["dogs"]);  
_dogAdapter.TableMappings.Add("dogs", "dogs");
```

La première commande est donc celle permettant d'alimenter le Dataset via un **SELECT**. Il faut ensuite se servir de la méthode **Fill()** qui prendra pour paramètre la DataTable que l'on souhaite remplir.

```
DataColumn masterId = _dogDataset.Tables["dogs"].Columns["MasterId"];  
DataColumn idMaster = _dogDataset.Tables["masters"].Columns["Id"];  
DataRelation masterId_IdMaster = new DataRelation("MasterId_IdMaster", idMaster, masterId);  
_dogDataset.Relations.Add(masterId_IdMaster);
```

Il est possible par la suite de réaliser des relations concernant les différentes tables de notre Dataset, via l'utilisation des **DataColumn** et des **DataRelation**, puis de leur ajout à la collection de relations du **Dataset**.

Les SqlDataAdapter

Une fois les manipulations de données effectuées dans l'application, il faudra modifier la source de données pour avoir une réelle persistance de ces dernières. Pour ce faire, il faudra alimenter les trois requêtes **INSERT**, **DELETE** et **UPDATE** du SqlDataAdapter dans le but de lui permettre d'utiliser la méthode Update qui lui est propre.

Cette méthode va se charger de son côté de modifier les valeurs qui ont réellement été modifiées et qui ont levé des événements au cours de l'exécution du programme. Les valeurs ainsi modifiées vont être stockées dans une liste des changements qu'il est également possible d'exporter au besoin via l'utilisation d'une méthode dédiée du Dataset (**GetChanges()**) dans le but potentiel d'exporter ensuite ces modifications sous la forme d'un XML par exemple.

```
SqlCommand insertDog = new SqlCommand("INSERT INTO dogs (Name, CollarColor, NbrOfLegs, MasterId) VALUES (@name, @collarColor, @nbrOfLegs, @masterId)", _sqlConnection);
insertDog.Parameters.Add("@name", SqlDbType.NVarChar, 50, "Name");
insertDog.Parameters.Add("@collarColor", SqlDbType.NVarChar, 50, "CollarColor");
insertDog.Parameters.Add("@nbrOfLegs", SqlDbType.Int, 11, "NbrOfLegs");
insertDog.Parameters.Add("@masterId", SqlDbType.Int, 11, "MasterId");
```

```
SqlCommand deleteDog = new SqlCommand("DELETE FROM dogs WHERE Id = @id;", _sqlConnection);
deleteDog.Parameters.Add("@id", SqlDbType.Int, 11, "Id");

_dogAdapter.InsertCommand = insertDog;
_dogAdapter.UpdateCommand = updateDog;
_dogAdapter.DeleteCommand = deleteDog;

_dogAdapter.Update(_dogDataset, "dogs");
```

Les DataRow

Les **Datarow** sont une classe d'ADO.NET qui permet l'accès aux données sous la forme de champs. Pour accéder à un champ, il faut ainsi faire appel à une méthode générique prenant comme paramètre le nom de la colonne que l'on souhaite.

Pour parcourir les lignes d'une DataTable, il convient d'utiliser en règle générale Linq et la conversion de cette dernière sous la forme d'un Enumerable que l'on peut ainsi placer dans une boucle de type foreach.

Une fois cela fait, il est possible soit de simplement afficher les valeurs, soit de les stocker dans des classes via l'utilisation des constructeurs ou de la notation entre accolades et des propriétés settables.

```
foreach(var row in _dogDataset.Tables["dogs"].AsEnumerable())
{
    Console.WriteLine($"{row.Field<int>("Id")}. Nom : {row.Field<string>("Name")}, Couleur du collier : {row.Field<string>("CollarColor")},
}
```

```
Dog newDog = new Dog()
{
    Id = row.Field<int>("Id"),
    Name = row.Field<string>("Name"),
    CollarColor = row.Field<string>("CollarColor"),
    NbrOfLegs = row.Field<int>("NbrOfLegs"),
    MasterId = row.Field<int>("MasterId")
};
```