ADO.NET

Manipulation des Données en C#

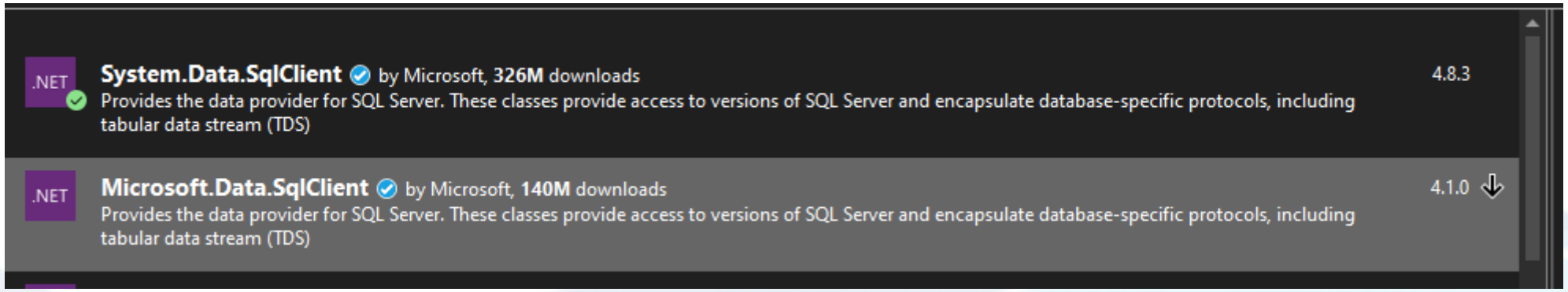
Antoine Dieudonné

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.



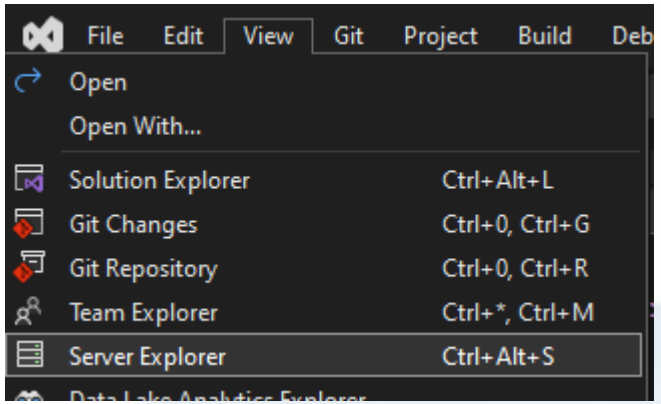

```
DONet
1   using System;
2   using System.Collections.Generic;
3   using System.Data;
4   using System.Data.SqlClient;
5   using System.Linq;
6   using System.Text;
7   using System.Threading.Tasks;
8
```

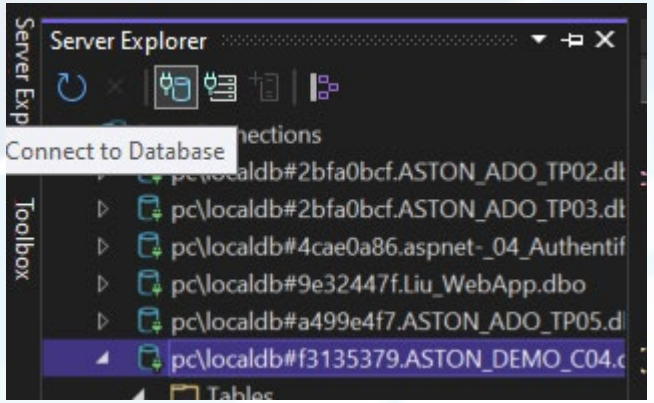
*Ne pas oublier le **using***

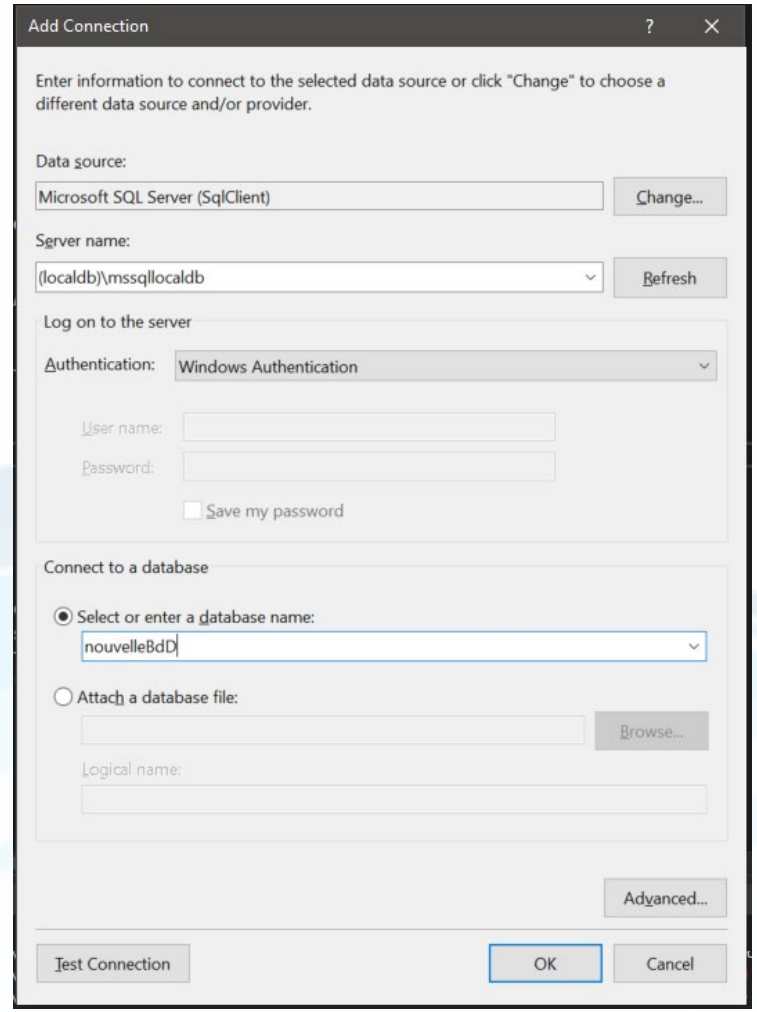
Créer une base de données

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

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







SqlConnection

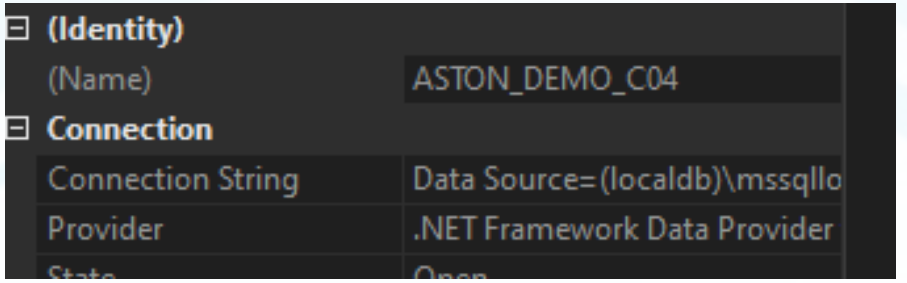
Une fois le package ajouté au projet, il faudra créer un objet de type ***SqlConnection*** afin d'accéder aux données. Pour créer un objet de ce type, il faut disposer d'une chaine de connexion. Cette chaine est trouvable facilement en observant les propriétés de notre base de donnée.

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 sur 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

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 crée 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 recourd 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**.