Cyril Seguenot

2017

Résumé

Support d’accompagnement d’une formation à ADO.Net faite en présentiel

Formation à ADO.Net

Support de cours

ADO.Net

Table des matières

[1 Le mode connecté 2](#_Toc481103829)

[1.1 Gestion de la connexion 2](#_Toc481103830)

[1.2 Exécution d’une requête 5](#_Toc481103831)

[1.3 Récupération du résultat dans une liste 6](#_Toc481103832)

[1.4 Affichage des données 7](#_Toc481103833)

[1.5 Autres types de requêtes 8](#_Toc481103834)

[1.6 Transactions 9](#_Toc481103835)

[1.7 Requêtes de masse 10](#_Toc481103836)

[2 Le mode déconnecté 13](#_Toc481103837)

[2.1 Création d’une source de données 13](#_Toc481103838)

[2.2 Récupération de données 17](#_Toc481103839)

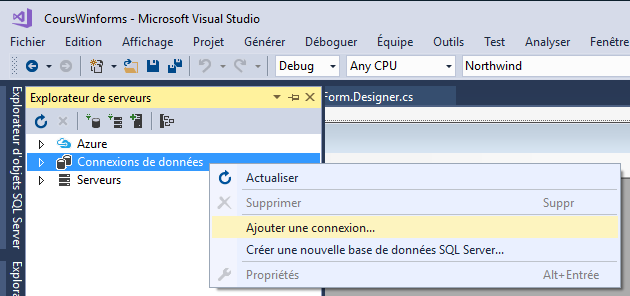
# Le mode connecté

## Gestion de la connexion

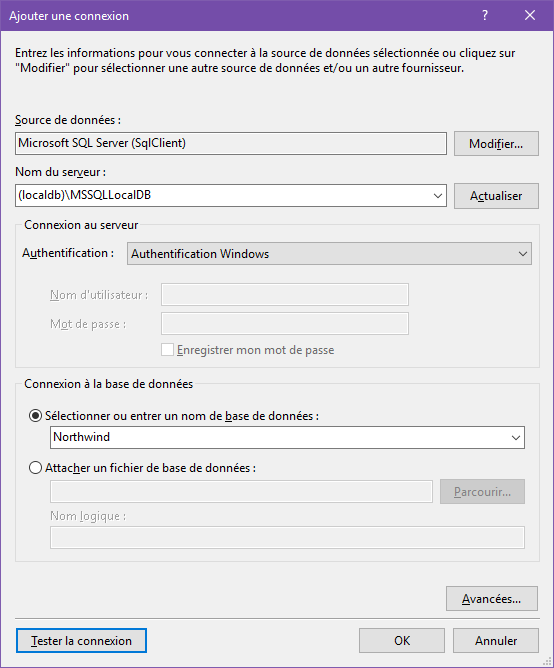
### Obtention d’une chaîne de connexion

Pour qu’une application puisse travailler avec une base de données, elle doit s’y connecter. Il faut donc créer une connexion. Pour cela :

* Ouvrir le panneau « explorateur de serveurs » de Visual Studio
* Dans le menu contextuel de la branche « Connexion de données », cliquer sur « Ajouter une connexion »



Ceci ouvre la fenêtre suivante, dans laquelle il faut sélectionner l’instance SQL Server, et la base que l’on souhaite utiliser.



Nous avons sélectionné ici l’instance SQL Server « (localDB)MSSQLLocalDB » avec authentification Windows, ainsi que la base de données Northwind disponible sur cette instance.

Le bouton Tester la connexion permet de s’assurer que la connexion est valide avant d’aller plus loin.

De retour dans le panneau « Explorateur de serveurs », nous pouvons voir la connexion créée :

|  |  |
| --- | --- |
|  | Dans ce panneau, le menu contextuel de chaque connexion permet d’ouvrir ou fermer la connexion, de la modifier, ou d’afficher ses propriétés.  Dans les propriétés, on peut notamment consulter la chaîne de connexion.  Une connexion peut bien entendu être utilisée dans plusieurs projets différents. |

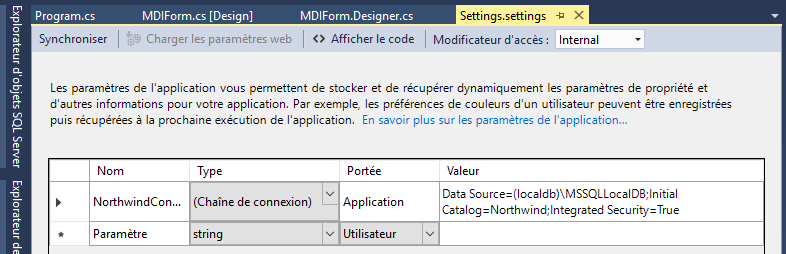
En affichant les propriétés, de la connexion, on a accès à la chaîne de connexion, qui ressemble à ceci :

Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Northwind;Integrated Security=True

C’est cette chaîne que nous utiliserons dans l’application pour nous connecter à la base et exécuter des requêtes.

### Stockage dans les paramètres de l’application

Afin de pouvoir accéder à la chaîne de connexion facilement et à de multiples endroits dans le code, il faut la stocker dans les paramètres de l’application. Pour cela, ouvrir la liste des paramètres en double-cliquant sur « Settings.settings » dans le dossier « Properties » :



Dans cette liste, ajouter une entrée de type « Chaine de connexion » et saisir la valeur obtenue précédemment depuis l’explorateur de serveur.

### Pool de connexions

Les connexions de base de données sont créées et contenues dans un pool. Lorsque l’application nécessite une connexion, ADO.Net extrait une connexion disponible du pool. Lorsque l’application ferme la connexion, celle-ci est rendue au pool et de nouveau disponible pour un prochain besoin. Pour des connexions gérées dans un pool, l’ouverture et la fermeture de connexion ne sont donc pas des opérations coûteuses. Fermer une connexion ne déconnecte pas de la base de données, mais retourne simplement la connexion au pool. L’ouverture d’une connexion implique juste l’obtention d’une connexion déjà ouverte du pool. Il est donc inutile, et même néfaste, de garder une connexion plus longtemps qu’on n’en a réellement besoin. Il faut ouvrir la connexion juste avant l’exécution de la commande, et la fermer juste après.

## Exécution d’une requête

Voici un exemple de code illustrant l’utilisation des objets ADO.Net pour récupérer des données depuis une table de la base :

using System;

using System.Data.SqlClient;

namespace Northwind

{

   /// <summary>

   /// Classe d'accès aux données

   /// </summary>

   public static class DAL

   {

      // Récupération de la liste des catégories de produits

      public static void GetCatégories()

      {

         // On récupérer la chaîne de connexion stockée dans le fichier App.config

         var connectString= Properties.Settings.Default.NorthwindConnectionString;

         // On écrit la requête SQL à exécuter

         string queryString = "SELECT CategoryId, CategoryName, Description FROM Categories";

         // On crée une connexion à partir de la chaîne de connexion

         using (var connect = new SqlConnection(connectString))

         {

            // On créé une commande à partir de la requête,

            // et en utilisant la connexion définies précédemment

            var command = new SqlCommand(queryString, connect);

            // On ouvre la connexion

            connect.Open();

            // On exécute la requête en récupérant son résultat   
 // dans un objet SqlDataRedader

            using (SqlDataReader reader = command.ExecuteReader())

            {

               // On lit et on affiche les lignes de résultat en boucle

               while (reader.Read())

               {

                  Console.WriteLine(String.Format("{0}, {1}, {2}",

                     reader[0], reader[1], reader[2]));

               }

            }

         }

         // Le fait d'avoir créé la connexion dans une instruction using

         // permet de la fermer automatiquement à la fin du bloc using

      }

   }

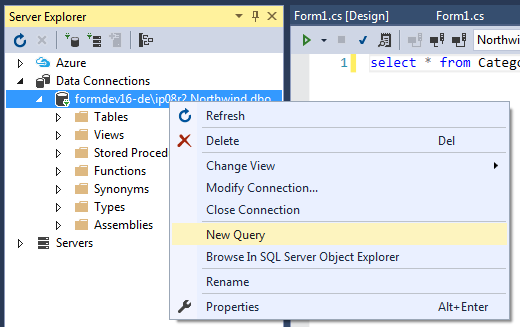
}

Les classes SqlConnection et SqlCommand sont définies dans l’espace de noms System.Data.SqlClient.

La classe SqlDataReader extrait les lignes une par une de la base de données, et ne conserve pas de verrou sur une ligne après la récupération des données. Ceci facilite beaucoup les accès concurrents de plusieurs utilisateurs à la base de données, ce qui est très important pour une application massivement multi-utilisateurs.

**Astuce :**

La requête SQL peut être écrite et exécutée dans une interface spécifique de Visual Studio, semblable à celle de SQL Server Management Studio. Il suffit pour cela d’ouvrir le panneau Explorateur de Serveur, et de sélectionner « New query » dans le menu contextuel de la base :



## Récupération du résultat dans une liste

Le code précédent ne fait qu’afficher les données récupérées dans la console de sortie.

Pour pouvoir exploiter les données, il faut les stocker dans une collection d’objets :

public static List<Personne> GetPersonnes()

{

   // Liste de personnes pour stocker le résultat de la requête

   var listPersonnes = new List<Personne>();

   var connectString = Properties.Settings.Default.NorthwindConnectionString;

   string queryString = @"select EmployeeID, TitleOfCourtesy, LastName, FirstName, BirthDate, Region, ReportsTo from Employees";

   using (var connect = new SqlConnection(connectString))

   {

      var command = new SqlCommand(queryString, connect);

      connect.Open();

      using (SqlDataReader reader = command.ExecuteReader())

      {

         while (reader.Read())

         {

            GetPersonnesFromDataReader(listPersonnes, reader);

         }

      }

   }

   return listPersonnes;

}

Cette méthode exécute une requête qui récupère une liste de personnes et stocke le résultat dans une liste générique, puis retourne cette liste.

La méthode GetPersonneFromDataReader permet de traduire chaque ligne du DataReader en objet Personne et de l’ajouter à la liste. Voici son code :

private static void GetPersonnesFromDataReader(List<Personne> lstPersonnes, SqlDataReader reader)

{

   var pers = new Personne();

   pers.Id = reader.GetInt32(0);       // ou bien (int)reader["EmployeeID"];

   pers.Titre = reader.GetString(1);  // ou  (string)reader["TitleOfCourtesy"];

   pers.Nom = reader.GetString(2);     // ...

   pers.Prénom = reader.GetString(3);

   pers.DateNais = reader.GetDateTime(4);

   if (!reader.IsDBNull(5))

      pers.Région = reader.GetString(5);

   if (!reader.IsDBNull(6))

      pers.IdManager = reader.GetInt32(6);

   lstPersonnes.Add(pers);

}

Il est important de noter que certaines colonnes du DataReader peuvent avoir la valeur DBNull, si le champ correspondant est nullable dans la base de données. Dans ce cas, il faut obligatoirement vérifier si la valeur vaut DBNull avant de la stocker dans l’objet, sinon cela provoque une erreur.

## Affichage des données

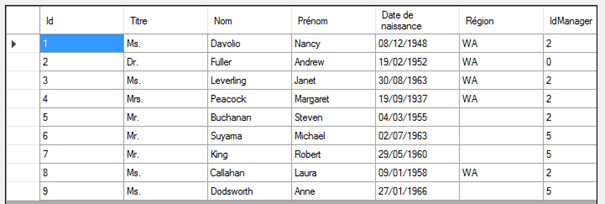
Winforms fournit des contrôles spécialisés dans l’affichage et la saisie de données (dans la section Données de la boite à outils).

Pour afficher des données tabulaires, le contrôle le plus adapté est le DataGridView. Son utilisation est très simple, puisqu’il suffit d’écrire la ligne suivante pour afficher les données :

dgvPersonnes.DataSource = DAL.GetPersonnes();

dgvPersonnes est le nom du contrôle DataGridView, et DAL.GetPersonnes renvoie une liste générique d’objets Personne.

Voici le visuel obtenu :



Les libellés des en-têtes de colonnes sont par défaut définis par les noms des propriétés de la classe utilisée (ici, la classe Personne). Pour les Personnaliser, on peut décorer les propriétés de l’attribut [DisplayName] :

public class Personne

{

   public int Id { get; set; }

   public string Titre { get; set; }

   public string Nom { get; set; }

   public string Prénom { get; set; }

   [DisplayName("Date de naissance")]

   public DateTime DateNais { get; set; }

   public string Région { get; set; }

   public int IdManager { get; set; }

}

…ou bien définir les libellés des en-têtes de colonne explicitement :

grid.Columns[0].HeaderText = "Identifiant";

## Autres types de requêtes

Les requêtes qui ne ramènent qu’un seul résultat, et non pas un tableau peuvent être exécutées de la façon suivante :

public static int GetNbProduits()

{

   int nbProd;

   var connectString = Properties.Settings.Default.NorthwindConnectionString;

   string queryString = @"select COUNT(\*) from Products";

   using (var connect = new SqlConnection(connectString))

   {

      var command = new SqlCommand(queryString, connect);

      connect.Open();

      nbProd = (int)command.ExecuteScalar();

   }

   return nbProd;

}

On utilise ici la méthode ExecuteScalar de la commande.

Les requêtes de type insert, update, delete sont exécutées de la façon suivante :

var command = new SqlCommand(req1, connect);

command.ExecuteNonQuery();

Comme ce type de requête ne renvoie aucun résultat, il n’y a rien à traiter en retour...

## Transactions

Pour s’assurer que plusieurs requêtes sont soit toutes validées, soit aucune, il faut les placer dans une transaction. L’exemple de code ci-dessous illustre la mise en œuvre d’une transaction avec ADO.Net :

public static void SupprimerCommande(int idCmde)

{

   // Préparation des requêtes et paramètres

   string req1 = @"delete from OrderDetails where OrderId = @id";

   var param1 = new SqlParameter("@id", DbType.Int32);

   param1.Value = idCmde;

   string req2 = @"delete from Orders where OrderId = @id";

   var param2 = new SqlParameter("@id", DbType.Int32);

   param2.Value = idCmde;

   using (var connect = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      // Ouverture de la connexion

      connect.Open();

      // Début de la transaction

      SqlTransaction tran = connect.BeginTransaction();

      try

      {

         // Création et exécution des commandes, en leur affectant la transaction

         var com1 = new SqlCommand(req1, connect, tran);

         com1.Parameters.Add(param1);

         com1.ExecuteNonQuery();

         var com2 = new SqlCommand(req2, connect, tran);

         com2.Parameters.Add(param2);

         com2.ExecuteNonQuery();

         // Validation de la transaction s'il n'y a pas eu d'erreur

         tran.Commit();

      }

      catch(Exception)

      {

         // Annulation de la transaction en cas d'erreur

         tran.Rollback();

         throw;   // Remontée de l'erreur à l'appelant

      }

   }

De façon générale, les requêtes insert, update et delete portant sur plusieurs lignes doivent être placées dans des transactions, pour assurer la cohérence des données. Mais pour plus de sécurité, il est recommandé de le faire systématiquement, même si les requêtes ne portent que sur une seule ligne.

## Requêtes de masse

Dans une application de gestion, il est fréquent d’avoir des écrans de saisies en tableau, de sorte que l’utilisateur puisse saisir plusieurs enregistrements d’affilée rapidement.

On peut enregistrer les données dans la base après validation de chaque ligne, mais cela n’est pas optimal du point de vue des performances, car on exécute un grand nombre de requêtes unitaires d’insertion. Le moteur de base de données fonctionne de façon optimale sur les traitements de masse. Pour insérer des données en masse, il est possible d’utiliser une requête du type :

insert ... select ... from MemTable

… où MemTable représente une table mémoire. Ceci ne fonctionne que si le type de la table mémoire a été préalablement défini en base au moyen d’une instruction :

create type TypeMemTable as table (...)

Le code ci-dessous montre un exemple complet d’enregistrement d’une liste de personnes dans la table Employees, en utilisant une requête d’insertion de masse.

// Insertion d'une liste de personne en base avec requête de masse

public static void AjouterPersonnes(List<Personne> listPers)

{

   // Ecriture de la requête d'insertion en masse

   // Le paramètre @table contiendra les enregistrements à insérer

   string req = @"

      insert Employees (LastName, FirstName, TitleOfCourtesy)

      select Nom, Prenom, Titre from @table";

   // Création du paramètre de type table mémoire

   // /!\ Le type TypeTablePersonne doit être créé au préalable dans la base

   var param = new SqlParameter("@table", SqlDbType.Structured);

   DataTable tablePers = GetDataTableForPersonnes(listPers);

   param.TypeName = "TypeTablePersonne";

   param.Value = tablePers;

   using (var cnx = new SqlConnection(Settings.Default.NorthwindConnectionString))

   {

      // Ouverture de la connexion et début de la transaction

      cnx.Open();

      SqlTransaction tran = cnx.BeginTransaction();

      try

      {

         // Création et exécution de la commande

         var command = new SqlCommand(req, cnx, tran);

         command.Parameters.Add(param);

         command.ExecuteNonQuery();

         // Validation de la transaction s'il n'y a pas eu d'erreur

         tran.Commit();

      }

      catch (Exception)

      {

         tran.Rollback(); // Annulation de la transaction en cas d'erreur

         throw;   // Remontée de l'erreur à l'appelant

      }

   }

}

Le paramètre @table est de type SqlDbType.Structured. Il s’agit d’une table mémoire, dont les colonnes correspondent à celles d’un type utilisateur défini au préalable dans la base. Ce type est nommé ici TypeTablePersonne.

La méthode GetDataTableForPersonne surlignée en jaune permet de construire la table mémoire, et de la charger avec les données de la liste passée en paramètre. Voici son code :

// Création et remplissage d'une table mémoire à partir d'une liste de personnes

private static DataTable GetDataTableForPersonnes(List<Personne> listPers)

{

// Créaton d'une table mémoire

DataTable table = new DataTable();

// Création des colonnes Nom et Prenom de type chaîne et ajout à la table

var colNom = new DataColumn("Nom", typeof(string));

colNom.AllowDBNull = false;

table.Columns.Add(colNom);

var colPrénom = new DataColumn("Prenom", typeof(string));

colPrénom.AllowDBNull = false;

table.Columns.Add(colPrénom);

// Pour les colonnes nullables, on peut utiliser une syntaxe plus courte

// car AllowDBNull à la valeur True par défaut

table.Columns.Add(new DataColumn("Titre", typeof(string)));

/\* Si la colonne de clé primaire n'est pas auto-incrémentée,

on peut définir une contrainte de clé primaire sur la table

Ceci affecte automatiquement Unique = True et AllowDBNull = False

sur la ou les colonne(s) définie(s) comme clé \*/

//table.PrimaryKey = new DataColumn[] { col1, col2 };

// Chargement de la liste des personnes dans la table mémoire

foreach (var p in listPers)

{

// Création d'une ligne de table

DataRow ligne = table.NewRow();

ligne["Nom"] = p.Nom;

ligne["Prenom"] = p.Prénom;

// Pour une colonne nullable dans la base,

// il faut affecter la valeur DBNull à la place de null

if (p.Titre != null)

ligne["Titre"] = p.Titre;

else ligne["Titre"] = DBNull.Value;

// Ajout de la ligne dans la table

table.Rows.Add(ligne);

}

      return table;

}

Les tables mémoires sont représentées par la classe DataTable, et leurs colonnes par la classe DataColumn, tous deux définis dans l’espace de noms System.Data. Pour chaque colonne de la table, il faut définir au minimum un nom et un type de données. On peut définir en plus des contraintes de type « unique » et « not null ».

On peut définir la clé primaire de la table au moyen de sa propriété PrimaryKey, qui est un tableau de DataColumn. Toutes les colonnes ajoutées à ce tableau verront automatiquement leurs propriétés affectées de la façon suivante : AllowDbNull = False et Unique = True.

Lorsqu’on charge les données dans la table, il faut faire attention à leur caractère nullable : les valeurs null de .net et de SQL Server ne sont pas compatibles ; il faut donc faire soi-même la transcription de l’une en l’autre à l’aide de DbNull.Value.

NB/ En C#, tous les types références (c’est-à-dire les classes) sont nullables. Les types valeur (int, double, DateTime…etc.) peuvent être rendus nullables s’ils sont déclarés de la façon suivante :

int? Id;

Dans la mesure du possible, il est plus simple de définir dans la base de données les colonnes de type numérique comme non nullables, avec une valeur par défaut 0. Ceci évite d’avoir à gérer la valeur null dans le code C#, et évite également d’avoir à utiliser la fonction IsNull dans les requêtes SQL qui font des opérations mathématiques sur ces colonnes.

Enfin, voici le code SQL de création du type table dont le nom est spécifié comme valeur de la propriété TypeName du paramètre de requête :

create type TypeTablePersonne as table

(

Nom nvarchar(20) not null,

Prenom nvarchar(10) not null,

Titre nvarchar(25)

)

Ses noms de colonnes doivent correspondre aux noms des objets DataColumns créés dans la méthode GetDataTableForPersonnes.

**Mise à jour et suppression de masse :**

La technique exposée précédemment peut bien entendu être appliquée avec des requêtes de mise à jour ou de suppression de masse, du style :

delete e

from Employees e

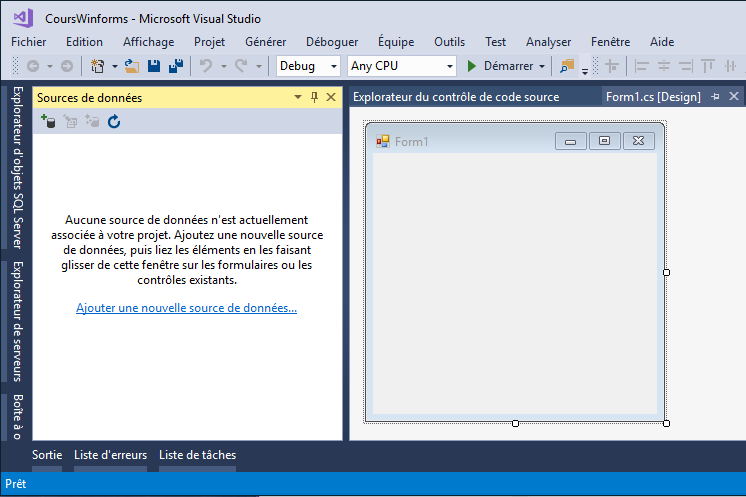
inner join @table t on e.EmployeeID = t.Id

# Le mode déconnecté

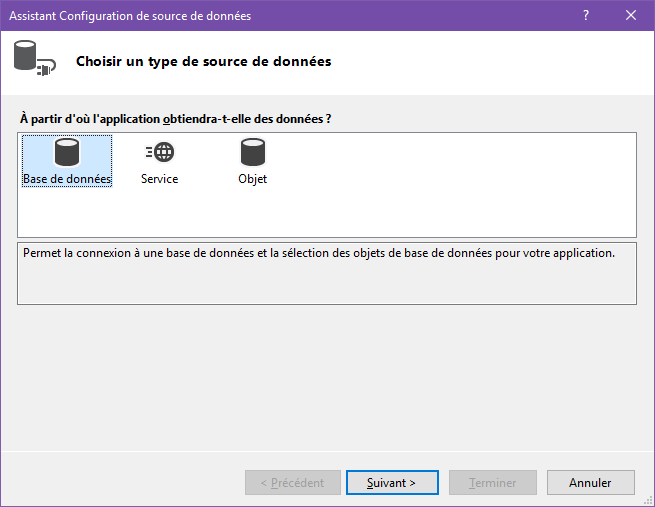
## Création d’une source de données

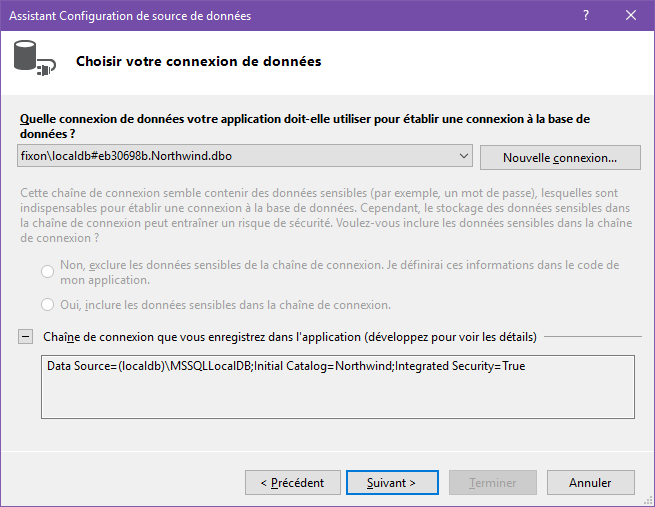
Pour fonctionner en mode déconnecté, l’application à besoin d’une source de données locale. Pour créer cette source, dans le panneau « Source de données », cliquer sur le lien « Ajouter une nouvelle source de données ».

NB/ Le panneau Source de données est accessible via le menu Affichage \ Autres fenêtres \ Sources de données.

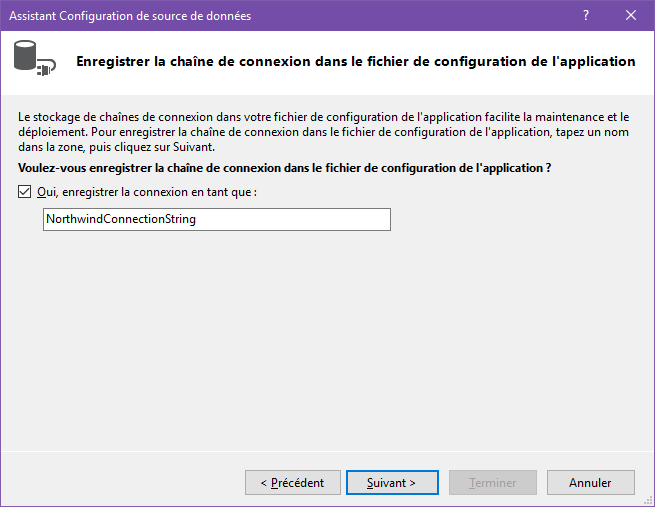


Le clic sur le lien ouvre la fenêtre suivante, dans laquelle on sélectionne « Base de données », puis à l’étape suivante « Dataset » :



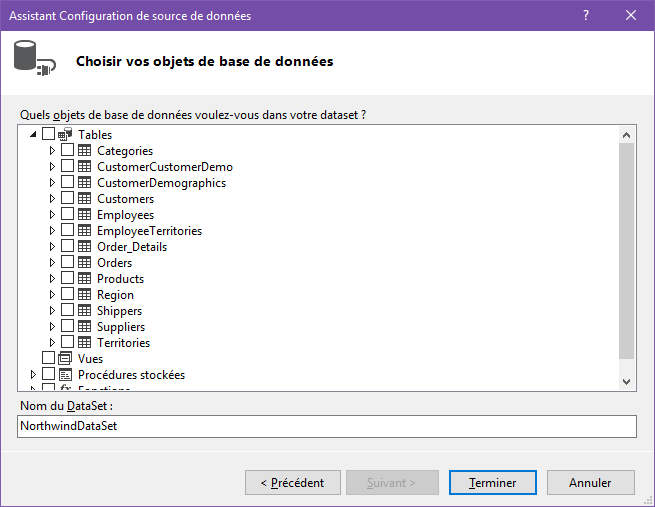
L’étape suivante nous propose de sélectionner ou de créer une connexion, qui nous permettra d’accéder à la base de données :

Un clic sur le bouton « Nouvelle connexion » ouvre la boîte de dialogue vue dans le précédent chapitre. Si on crée une nouvelle connexion, l’étape suivante permet de nommer et d’enregistrer la chaîne de connexion :

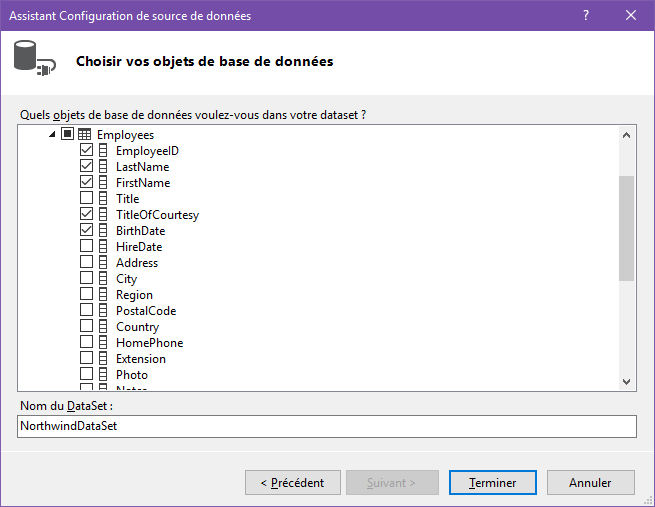


La chaîne de connexion est stockée dans une section spécifique du fichier App.config de l’application.

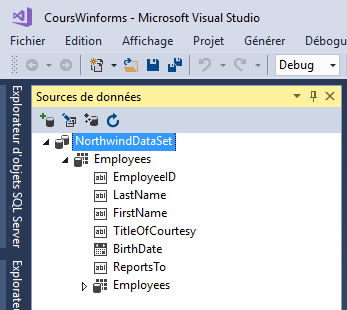
Dans la base de données, nous pouvons sélectionner les objets que nous souhaitons utiliser (tables, vues, procédures stockées, fonctions…)



Pour chaque table, nous pouvons également choisir la liste des champs que nous souhaitons utiliser., comme le montre l’écran ci-dessous :



Lorsqu’on clique sur le bouton Terminer, le Dataset apparaît dans la liste des sources de données :



NB/ Après création, le dataset peut être modifié pour ajouter ou enlever des éléments. Il suffit pour cela de le sélectionner et de cliquer sur « Configurer le dataset à l’aide de l’assistant » dans son menu contextuel

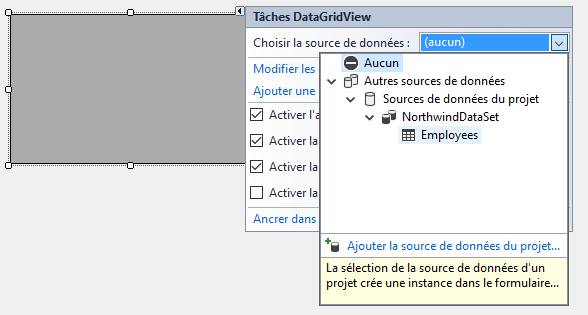
## Récupération de données

Voyons maintenant comment afficher les données de la source de données créée précédemment.

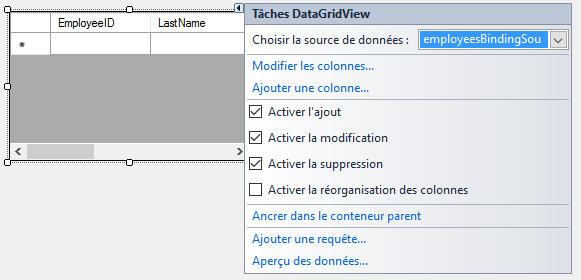
Dans la fenêtre principale de l’application, faire glisser un composant DataGridView.

Visual Studio affiche automatiquement une fenêtre de configuration, dans laquelle nous pouvons sélectionner la source de données qui alimentera la grille.

Sélectionnons la table Employees du dataset que nous avons créé plus haut :

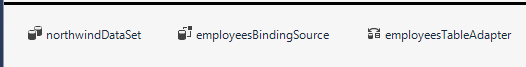


Visual Studio configure automatiquement la grille en créant des colonnes correspondant à celles du dataset :



Nous pouvons également activer ou non l’ajout, la modification et la suppression des données au travers de la grille.

Remarquons que Visual Studio a également généré automatiquement le composant suivants dans la fenêtre :



* Le premier correspond au dataset que nous avons créé plus haut.
* Le second est un objet de type BindingSource, qui permet de faire le lien entre la source de données et les composants d’affichage.
* Le troisième est un objet TableAdapter qui permet de remplir le Dataset à partir d’une requête SQL