

# Accès aux bases de données à l'aide de connecteur ADO.NET.

---

MOHAMED AIROUCHE

# Les opérations CRUD dans une base de données

Le terme CRUD est un **acronyme** des noms des quatre opérations de base de la gestion de la persistance des données. Utiliser une base de données peut se résumer à quatre opérations principales :

## Les opérations CRUD

*Create* : créer des données

*Read* : lire des données

*Update* : modifier des données

*Delete* : supprimer des données

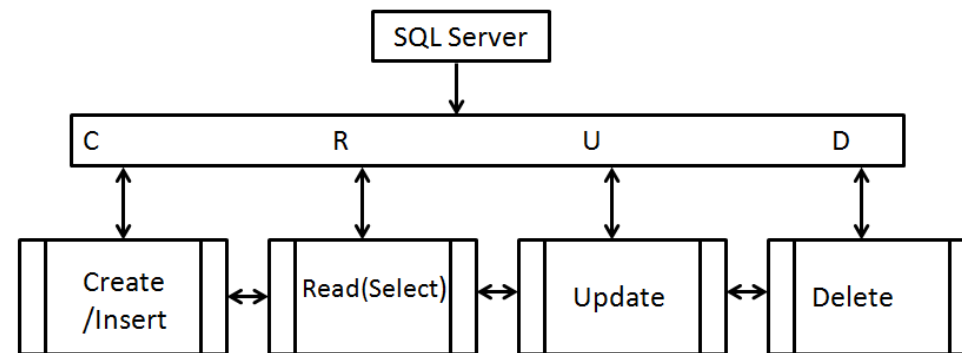
## Les commandes en SQL

**Create avec la commande INSERT**

**Read avec la commande SELECT**

**Update avec la commande UPDATE**

**Delete avec la commande DELETE**



<https://openclassrooms.com/fr/courses/4055451-modelisez-et-implementez-une-base-de-donnees-relationnelle-avec-uml/4459270-oh-crud>

# Les opérations CRUD et Scaffolding avec Visual Studio

---

## **Scaffolding (l'échafaudage)**

Le Scaffolding, c'est la capacité que possèdent Visual Studio et ASP.NET MVC pour vous générer du code répétitif permettant de créer, lire, mettre à jour et supprimer des données dans votre application. En analysant votre modèle, Visual Studio est capable d'en déduire les champs pour réaliser des formulaires classiques et pour générer des actions de contrôleur.

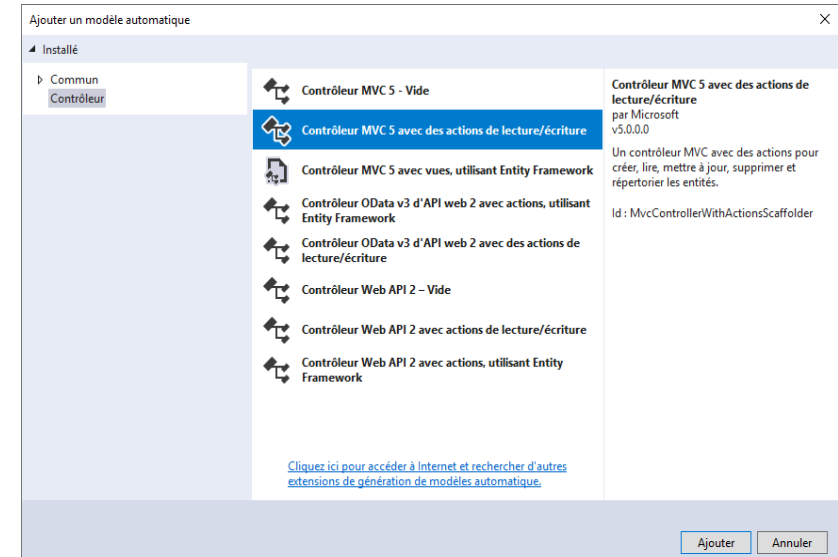
- ✓ **Scaffolding de contrôleur**
- ✓ **Scaffolding de vues**
- ✓ **Scaffolding de contrôleur en utilisant Entity Framework**

<https://openclassrooms.com/fr/courses/1730206-apprenez-asp-net-mvc/2098936-scaffolding>

# Les opérations CRUD avec Contrôleur ASP.NET MVC

Visual Studio 2017 permet d'ajouter un Contrôleur avec des fonctionnalités de génération automatique des méthodes d'actions suivantes :

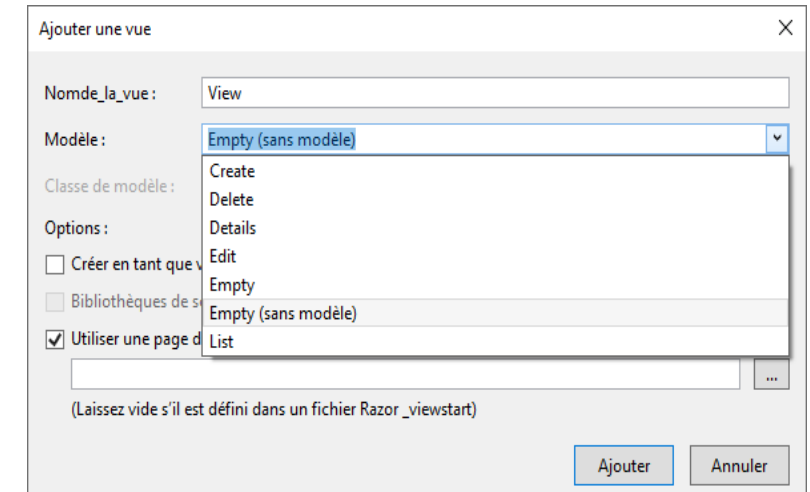
- Create (Créer): les méthodes d'action (GET, POST) pour insérer des données.
- Edit (Modifier) : les méthodes d'action (GET, POST) pour la mise à jour des données.
- Delete (Supprimer): les méthodes d'action (GET, POST) pour supprimer un enregistrement.
- Details (Détails) : la méthode d'action pour afficher un enregistrement,
- List (Liste) : la méthode d'action pour afficher une liste d'enregistrements.



# Les opérations CRUD et les vues en ASP.NET MVC

Les fonctionnalités de génération de modèles de Visual Studio 2017 fournissent différents modèles de page. Parmi ces modèles :

- Create (Créer): générer un formulaire pour insérer des données.
- Edit (Modifier) : générer un formulaire de mise à jour des données.
- Delete (Supprimer): générer un formulaire pour afficher un enregistrement avec un bouton pour confirmer la suppression.
- Details (Détails) : générer un formulaire pour afficher un enregistrement avec deux boutons, un pour aller pour modifier le formulaire et un pour supprimer la page d'enregistrement affichée.
- List (Liste) : générer un tableau HTML pour afficher une liste d'enregistrements.
- Empty (Vide): générer une page vide sans utiliser de modèle.



# Exemple

---

## Développer une application qui permet de gérer des employés :

On souhaite créer une application qui permet de gérer des employés. Chaque employé est défini par son identifiant, son nom, son prénom, son sexe, son département et sa ville.

L'application doit permettre de :

- Afficher tous les employés.
- Afficher les détails d'un employé.
- Saisir et ajouter un nouvel employé.
- Éditer et modifier un employé.
- Supprimer un employé.

Les employés sont stockés dans une collection d'employés de type *Dictionary* dont la clé de chaque employé est représentée par son identifiant. L'application se compose de deux couches :

- ✓ La couche Web basée sur Asp.net MVC qui contient l'entité Employee.
- ✓ La couche Métier qui contient :
  - L'interface IEmployeeDataAccess.
  - Une implémentation de cette interface.

# Exemple

---

## Création d'une application ASP.NET MVC :

- Créez une application ASP.NET MVC **DemoEmployee**.
- Un principe à suivre lors du développement est la **séparation des responsabilités**. Selon ce principe, l'application doit être divisée en fonction des types des tâches qu'il effectue.
- La séparation des responsabilités est un aspect fondamental de l'utilisation des couches dans les architectures d'applications. Dans l'idéal, les règles et la logique métier doivent se trouver dans un projet distinct, qui ne doit pas dépendre d'autres projets dans l'application. Cette séparation permet de garantir que le modèle de l'application est facile à tester et peut évoluer sans être étroitement couplé à des détails d'implémentation de bas niveau.
- Dans un scénario de projet unique, la séparation des préoccupations s'obtient par l'utilisation de dossiers. La logique métier doit résider dans les services et les classes contenus dans le dossier Modèles. Nous utilisons cette approche pour notre exemple.

<https://docs.microsoft.com/fr-fr/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>

# Exemple

---

## Ajout du modèle à l'application :

- Cliquez avec le bouton droit sur le dossier Modèles et sélectionnez Ajouter >> Classe. Nommez votre classe Employee.cs. Cette classe contiendra les propriétés de modèle Employee.
- Ouvrez Employee.cs et mettez-y le code suivant. Puisque nous ajoutons les validations requises aux champs de la classe Employee, nous devons donc utiliser DataAnnotations.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Web;

namespace DemoEmployee.Models
{
    public class Employee
    {
        public int ID { get; set; }
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }
        [Required]
        public string Gender { get; set; }
        [Required]
        public string Department { get; set; }
        [Required]
        public string City { get; set; }
    }
}
```



# Exemple

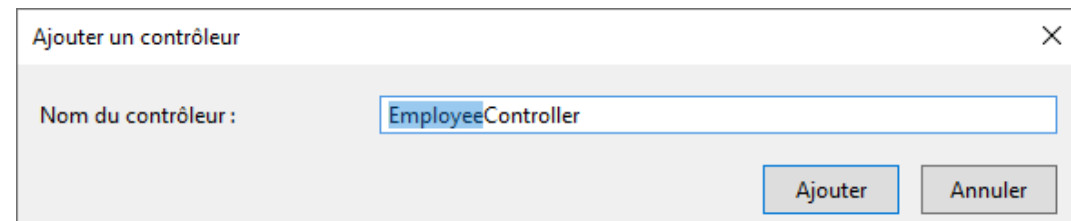
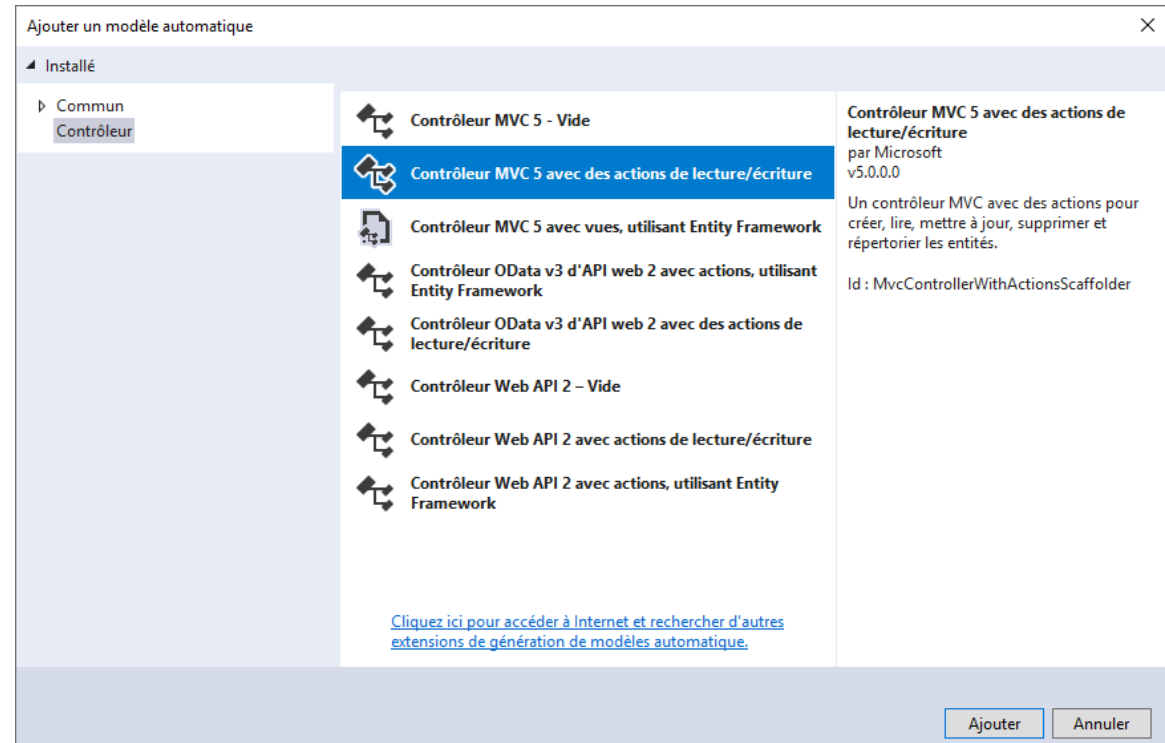
## Ajout du contrôleur à l'application :

Faire un clic droit sur le dossier «Controllers» du projet >> Ajouter >> Contrôleur.

Sélectionner le modèle Contrôleur MVC 5 avec des actions de lecture/écriture.

Visual studio 2017 vas créer automatiquement les méthodes d'actions suivantes :

- Index
- Create (GET, POST)
- Edit (GET, POST)
- Delete (GET, POST)
- Details

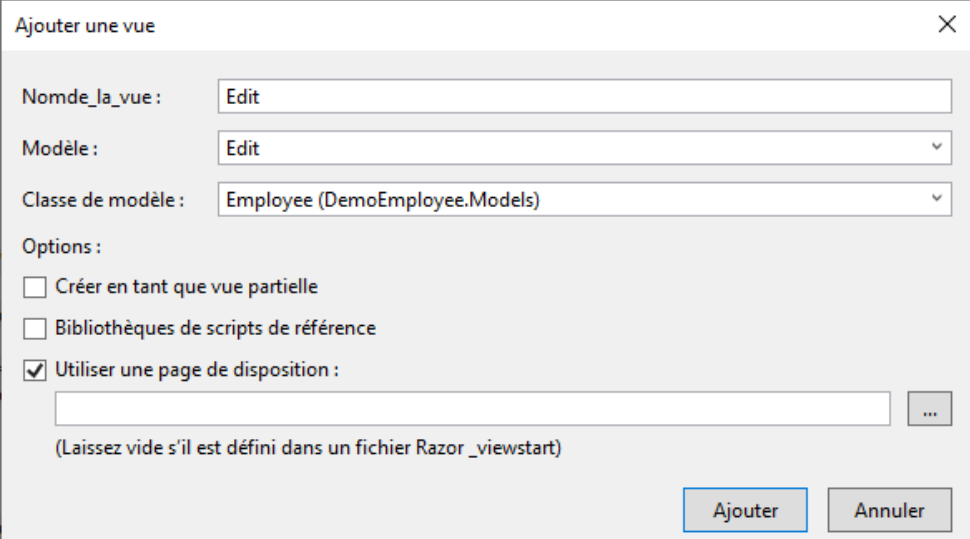


# Exemple

## Ajout des vues associées aux méthodes d'actions du contrôleur à l'application:

Pour chaque méthode d'action de contrôleur ajoutez une vue en utilisant la génération des vues automatique de Visual studio. Au final on doit se trouver avec les vues suivantes :

- Index.cshtml
- Create.cshtml
- Edit.cshtml
- Delete.cshtml
- Details.cshtml



Ajouter une vue

Nom de la vue : Edit

Modèle : Edit

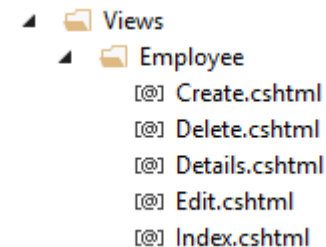
Classe de modèle : Employee (DemoEmployee.Models)

Options :

- ☐ Créer en tant que vue partielle
- ☐ Bibliothèques de scripts de référence
- ☒ Utiliser une page de disposition :

(Laissez vide s'il est défini dans un fichier Razor \_viewstart)

Ajouter Annuler



Views

- Employee
  - Create.cshtml
  - Delete.cshtml
  - Details.cshtml
  - Edit.cshtml
  - Index.cshtml

# Exemple

---

## Ajout du dossier pour la couche métier :

Ajoutez un nouveau dossier au projet. Nommez-le : Metier.

Ajoutez une interface au dossier Metier. Nommez-la : `IEmployeeDataAccess.cs`. Cette interface contiendra les opérations de la logique métier pour réaliser les différentes fonctionnalités de l'application.

Les interfaces peuvent avoir plusieurs implémentations qui peuvent être échangées en fonction des besoins. L'utilisation des interfaces permet aux applications d'être faiblement couplées avec ces interfaces, plutôt qu'avec des implémentations spécifiques, ce qui les rend plus faciles à étendre, mettre à jour et tester.

Le but d'utilisation de l'interface est d'avoir un couplage faible entre les modules ou les couches de l'application.

```
using DemoEmployee.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DemoEmployee.Metier
{
    interface IEmployeeDataAccess
    {
        IEnumerable<Employee> GetAllEmployees();
        Employee GetEmployeeData(int ID);
        Employee AddEmployee(Employee emp);
        void UpdateEmployee(Employee emp);
        void DeleteEmployee(int ID);
    }
}
```

# Exemple

---

## Ajout d'une classe pour implémenter l'interface :

Ajoutez une classe au dossier Metier. Nommez-la : EmployeeDataAccessImp1.cs. Cette classe contiendra l'implémentation de l'interface pour réaliser les différentes fonctionnalités de l'application en utilisant les données dans un dictionnaire.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using DemoEmployee.Models;

namespace DemoEmployee.Metier
{
    public class EmployeeDataAccessImp1 : IEmployeeDataAccess
    {
        public Employee AddEmployee(Employee emp)
        {
            throw new NotImplementedException();
        }

        public void DeleteEmployee(int ID)
        {
            throw new NotImplementedException();
        }
    }
}
```

//Suite de la classe EmployeeDataAccessImp1

```
        public IEnumerable<Employee> GetAllEmployees()
        {
            throw new NotImplementedException();
        }

        public Employee GetEmployeeData(int ID)
        {
            throw new NotImplementedException();
        }

        public void UpdateEmployee(Employee emp)
        {
            throw new NotImplementedException();
        }
    }
}
```

# Exemple

---

## Ajout du code dans la classe qui implémente l'interface :

Ajoutez du code aux différentes méthodes pour que l'application puisse gérer des employés qui sont stockés dans une collection d'employés de type *Dictionary* (afficher tous les employés, saisir et ajouter un nouvel employé, éditer et modifier un employé, supprimer un employé).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using DemoEmployee.Models;

namespace DemoEmployee.Metier
{
    public class EmployeeDataAccessImp1 : IEmployeeDataAccess
    {
        static int count = 0;
        static private Dictionary<int, Employee> employees = new Dictionary<int, Employee>();
        public Employee AddEmployee(Employee emp)
        {
            count++;
            emp.Id = count;
            employees[emp.Id] = emp;
            return emp;
        }
    }
}
```

# Exemple

---

```
//Suite de la classe EmployeeDataAccessImp1

    public void DeleteEmployee(int ID)
    {
        employees.Remove(ID);
    }

    public IEnumerable<Employee> GetAllEmployees()
    {
        return employees.Values;
    }

    public Employee GetEmployeeData(int Id)
    {
        Employee emp;
        employees.TryGetValue(Id, out emp);
        return emp;
    }

    public void UpdateEmployee(Employee emp)
    {
        employees[emp.Id] = emp;
    }
}
```

# Base de données locale SQL Server Express

---

## Base de données locale SQL Server Express

La base de données locale est une version allégée de serveur de base de données SQL Server Express. La base de données locale s'exécute dans un mode d'exécution spécial de SQL Server Express qui vous permet d'utiliser des bases de données en tant que fichiers *.mdf*. En règle générale, les fichiers de base de données de base de données locale peuvent être conservés dans le dossier *App\_Data* d'un projet Web.

SQL Server Express n'est pas recommandé pour une utilisation dans les applications Web de production. En particulier, la base de données locale ne doit pas être utilisée pour la production avec une application Web, car elle n'est pas conçue pour fonctionner avec IIS. Toutefois, une base de données de base de données locale peut être facilement migrée vers SQL Server ou SQL Azure.

Dans Visual Studio 2017, la base de données locale est installée par défaut dans Visual Studio.

## Création d'une chaîne de connexion et utilisation de SQL Server LocalDB

La façon la plus simple d'utiliser LocalDB est de se connecter à l'instance en utilisant la chaîne de connexion suivante: `Server=(localdb)\MSSQLLocalDB;Integrated Security=true`.

Pour vous connecter à une base de données spécifique en utilisant le nom de fichier, connectez-vous à l'aide d'une chaîne de connexion suivante :

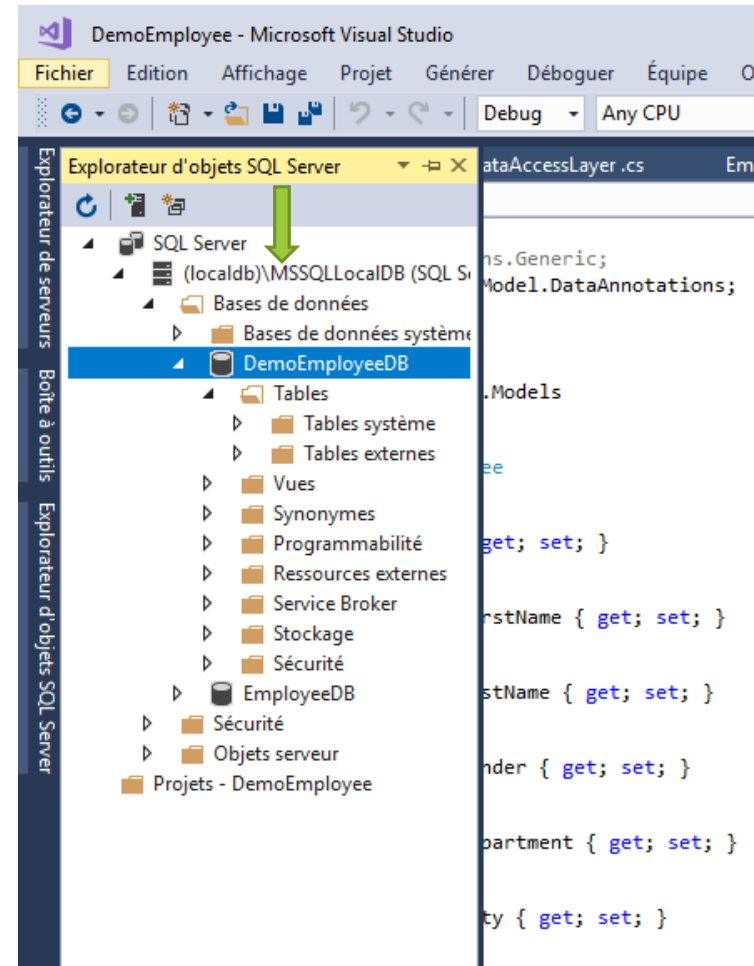
`Server=(LocalDB)\MSSQLLocalDB;Integrated Security=true;AttachDbFileName=D:\Data\MyDB1.mdf`.

<https://docs.microsoft.com/fr-ca/aspnet/mvc/overview/getting-started/introduction/creating-a-connection-string>

# Base de données locale SQL Server Express

## Création d'une base de données dans la Base de données locale SQL Server Express

- Faire un clic droit par la souris sur votre base donnée Local au lieu indiqué ci-dessus.
- Sélectionnez «Nouvelle requête ...» et ajoutez la requête de création de la base de donnée DemoEmployees et de la table tbEmployee.



<https://docs.microsoft.com/fr-ca/aspnet/mvc/overview/getting-started/introduction/creating-a-connection-string>



# Base de données locale SQL Server Express

## Ajout d'une table dans la base de donnée

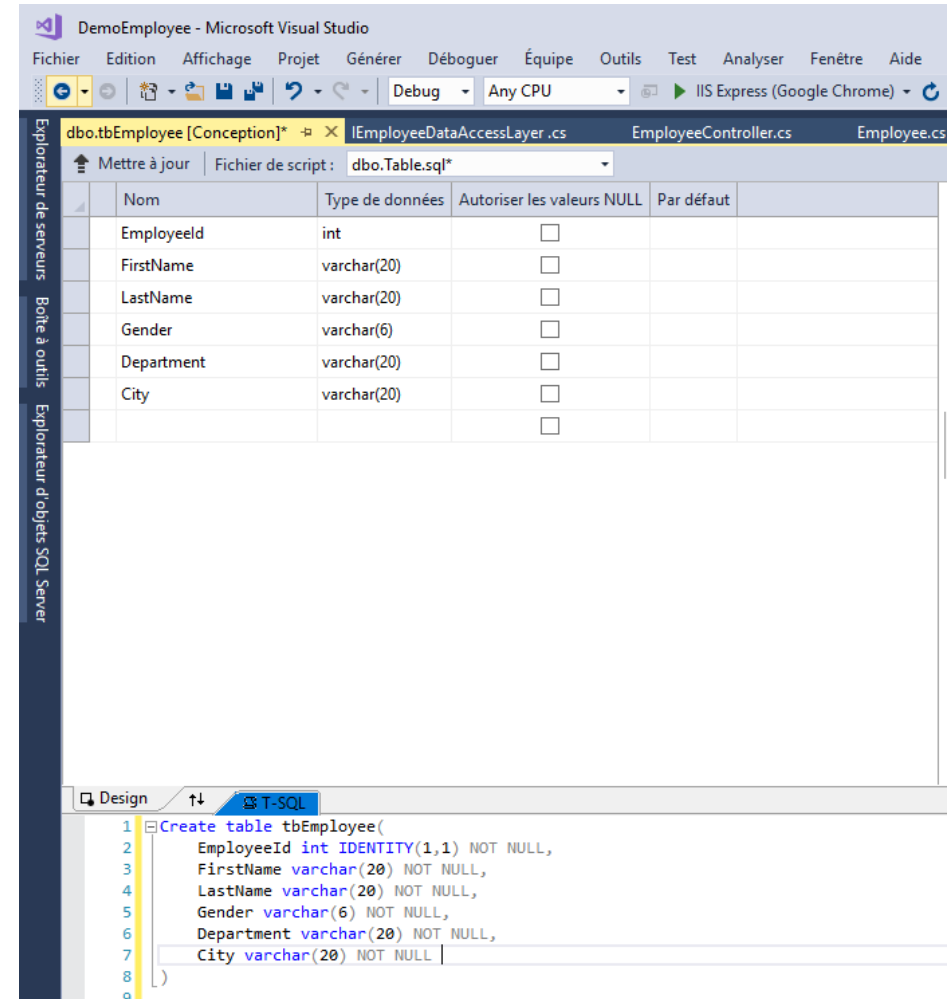
```
Create Database [DemoEmployeeDB]  
Go
```

```
USE [DemoEmployeeDB]  
GO
```

```
SET ANSI_NULLS ON  
GO
```

```
SET QUOTED_IDENTIFIER ON  
GO
```

```
CREATE TABLE [dbo].[tbEmployee] (  
    [EmployeeId] INT IDENTITY (1, 1) NOT NULL,  
    [FirstName] VARCHAR (20) NOT NULL,  
    [LastName] VARCHAR (20) NOT NULL,  
    [Gender] VARCHAR (6) NOT NULL,  
    [Department] VARCHAR (20) NOT NULL,  
    [City] VARCHAR (20) NOT NULL  
);
```



<https://docs.microsoft.com/fr-ca/aspnet/mvc/overview/getting-started/introduction/creating-a-connection-string>

# Base de données locale SQL Server Express

---

## Ajout des procédures dans la base donnée

- ✓ La procédure pour insérer un nouvel employé comme enregistrement dans la base de donnée

```
Create procedure spAddEmployee
(
    @FirstName VARCHAR(20),
    @LastName VARCHAR(20),
    @Gender VARCHAR(6),
    @Department VARCHAR(20),
    @City VARCHAR(20)
)
as
Begin
    Insert into tbEmployee (FirstName, LasstName, Gender, Department, City)
    Values (@FirstName, @LastName, @Gender,@Department, @City)
End
```

<https://docs.microsoft.com/fr-ca/aspnet/mvc/overview/getting-started/introduction/creating-a-connection-string>

# Base de données locale SQL Server Express

---

- ✓ La procédure pour modifier un enregistrement employee dans la base de donnée

```
Create procedure spUpdateEmployee
(
    @EmpId INTEGER ,
    @FirstName VARCHAR(20) ,
    @LastName VARCHAR(20) ,
    @Gender VARCHAR(6) ,
    @Department VARCHAR(20) ,
    @City VARCHAR(20)
)
As
begin
    Update tbEmployee
    set FirstName=@FirstName,
    LastName=@LastName,
    Gender=@Gender,
    Department=@Department,
    City=@City
    where EmployeeId=@EmpId
End
```

<https://docs.microsoft.com/fr-ca/aspnet/mvc/overview/getting-started/introduction/creating-a-connection-string>

# Base de données locale SQL Server Express

---

- ✓ La procédure pour supprimer un enregistrement employee dans la base de donnée

```
Create procedure spDeleteEmployee
(
    @EmpId int
)
As
begin
    Delete from tbEmployee where EmployeeId=@EmpId
End
```

- ✓ La procédure pour afficher tous les enregistrements employee dans la base de donnée

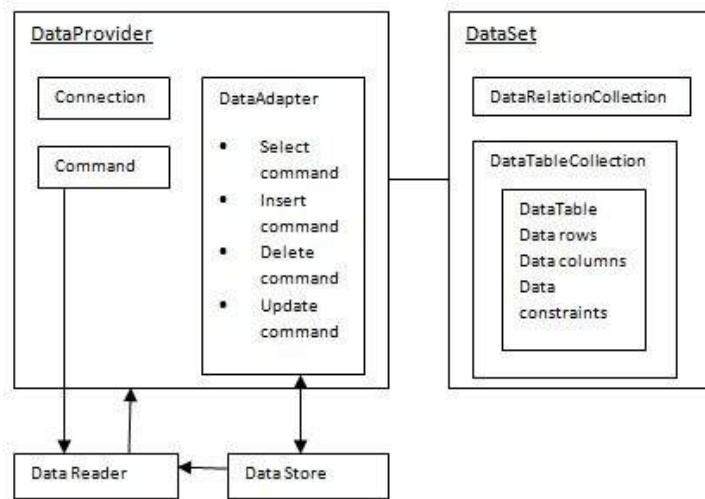
```
Create procedure spGetAllEmployees
as
Begin
    select *
    from tbEmployee
End
```

<https://docs.microsoft.com/fr-ca/aspnet/mvc/overview/getting-started/introduction/creating-a-connection-string>

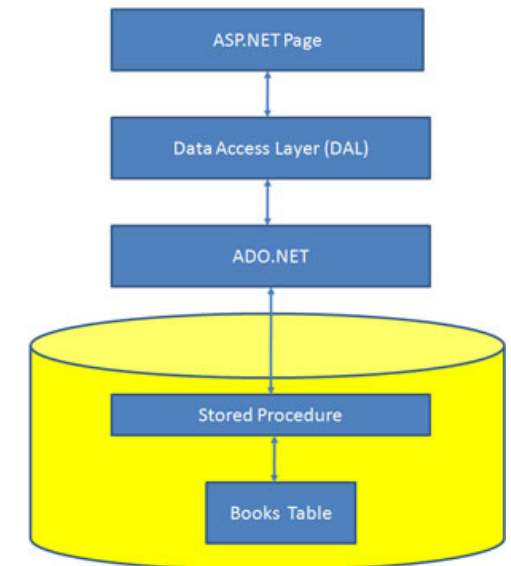
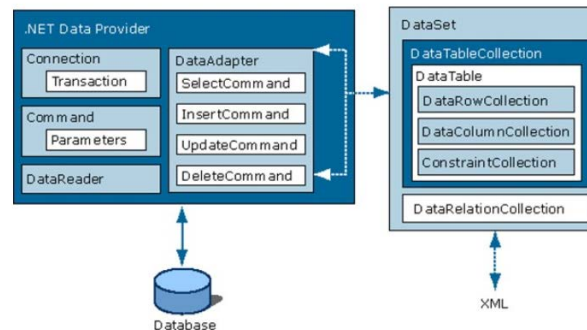
# ADO.NET pour l'accès au données

ADO.NET est un ensemble de classes qui exposent les services d'accès aux données pour les programmeurs .NET Framework. Partie intégrante du .NET Framework, il permet d'accéder à des données relationnelles, XML et d'application. ADO.NET répond à divers besoins en matière de développement, en permettant notamment de créer des clients de bases de données frontaux et des objets métier de couche intermédiaire utilisés par des applications.

ADO.NET fournit un pont entre la couche métier et la base de données. Les objets ADO.NET encapsulent toutes les opérations d'accès aux données.



ADO.NET Architecture Diagram



[https://www.tutorialspoint.com/asp.net/asp.net\\_ado\\_net.htm](https://www.tutorialspoint.com/asp.net/asp.net_ado_net.htm)

<https://docs.microsoft.com/fr-ca/dotnet/framework/data/adonet/>

# Exemple

---

## Extension de l'application en ajoutant une deuxième classe qui implémente l'interface :

Ajoutez une classe au dossier Metier. Nommez-le :

EmployeeDataAccessImp2.cs. Cette classe contiendra l'implémentation les opérations de la logique métier pour réaliser les différentes fonctionnalités de l'application en utilisant les données dans une base de données. Cette classe implémente l'interface IEmployeeDataAccess.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using DemoEmployee.Models;

namespace DemoEmployee.Metier
{
    public class EmployeeDataAccessImp2 : IEmployeeDataAccess
    {
        public Employee AddEmployee(Employee emp)
        {
            throw new NotImplementedException();
        }

        public void DeleteEmployee(int ID)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<Employee> GetAllEmployees()
        {
            throw new NotImplementedException();
        }

        public Employee GetEmployeeData(int ID)
        {
            throw new NotImplementedException();
        }

        public void UpdateEmployee(Employee emp)
        {
            throw new NotImplementedException();
        }
    }
}
```

# Exemple

---

## Ajout du code dans la classe qui implémente l'interface :

Ajoutez du code aux différentes méthodes pour que l'application puisse gérer des employés qui sont stockés dans une base de donnée (afficher tous les employés, saisir et ajouter un nouvel employé, éditer et modifier un employé, supprimer un employé).

Méthode qui permet d'ajouter un nouvel employé

```
public Employee AddEmployee(Employee emp)
{
    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand("spAddEmployee", con);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.AddWithValue("@FirstName", emp.FirstName);
        cmd.Parameters.AddWithValue("@LastName", emp.LastName);
        cmd.Parameters.AddWithValue("@Gender", emp.Gender);
        cmd.Parameters.AddWithValue("@Department", emp.Department);
        cmd.Parameters.AddWithValue("@City", emp.City);
        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
    }
    return emp;
}
```

# Exemple

---

Méthode qui permet de supprimer un employé.

```
public void DeleteEmployee(int Id)
{
    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand("spDeleteEmployee", con);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.AddWithValue("@EmpId", Id);
        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
    }
}
```



# Exemple

---

Méthode qui permet d'afficher tous les employés.

```
public IEnumerable<Employee> GetAllEmployees()
{
    List<Employee> lstemployee = new List<Employee>();
    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand("spGetAllEmployees", con);
        cmd.CommandType = CommandType.StoredProcedure;
        con.Open();
        SqlDataReader rdr = cmd.ExecuteReader();
        while (rdr.Read())
        {
            Employee employee = new Employee();
            employee.Id = Convert.ToInt32(rdr["EmployeeID"]);
            employee.FirstName = rdr["FirstName"].ToString();
            employee.LastName = rdr["LastName"].ToString();
            employee.Gender = rdr["Gender"].ToString();
            employee.Department = rdr["Department"].ToString();
            employee.City = rdr["City"].ToString();
            lstemployee.Add(employee);
        }
        con.Close();
    }
    return lstemployee;
}
```

# Exemple

---

Méthode qui permet d'afficher les information d'un employé.

```
public Employee GetEmployeeData(int Id)
{
    Employee employee = new Employee();
    using (SqlConnection con = new SqlConnection(connectionString))
    {
        string sqlQuery = "SELECT * FROM tbEmployee WHERE EmployeeID= " + Id;
        SqlCommand cmd = new SqlCommand(sqlQuery, con);
        con.Open();
        SqlDataReader rdr = cmd.ExecuteReader();
        while (rdr.Read())
        {
            employee.Id = Convert.ToInt32(rdr["EmployeeID"]);
            employee.FirstName = rdr["FirstName"].ToString();
            employee.LastName = rdr["LastName"].ToString();
            employee.Gender = rdr["Gender"].ToString();
            employee.Department = rdr["Department"].ToString();
            employee.City = rdr["City"].ToString();
        }
    }
    return employee;
}
```

# Exemple

---

Méthode qui permet de modifier les information d'un employé.

```
public void UpdateEmployee(Employee emp)
{
    using (SqlConnection con = new SqlConnection(connectionString))
    {
        SqlCommand cmd = new SqlCommand("spUpdateEmployee", con);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.AddWithValue("@EmpId", emp.Id);
        cmd.Parameters.AddWithValue("@FirstName", emp.FirstName);
        cmd.Parameters.AddWithValue("@LastName", emp.LastName);
        cmd.Parameters.AddWithValue("@Gender", emp.Gender);
        cmd.Parameters.AddWithValue("@Department", emp.Department);
        cmd.Parameters.AddWithValue("@City", emp.City);
        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
    }
}
```