Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Créer des services de données ADO .NET

Version 1.0


James RAVAILLE
http://blogs.dotnet-france.com/jamesr

Jean-Christophe VASSELON

Sommaire

1 Introduction 3
2 Création d'un service de données ADO .NET 4
2.1 Création d'un site Web 4
2.2 Création du service de données ADO .NET 4
2.3 Choix de la source de données 6
3 Utilisation du Framework Entity comme source de données 7
4 Définition des droits 11
4.1 Définition de droits sur les entités 12
4.2 Création d'opérations de services et gestion de leurs droits 13
5 Exposition de données non persistantes 15
5.1 Présentation de l'exercice 15
5.1.1 La classe ProcessView 15
5.1.2 Création du service Métier 16
5.1.3 Exposition du service métier via le service de données ADO .NET 17
5.2 Utilisation de l'interface IUpdatable 18
6 Introduction à la sécurité avec ADO.Net Data Services 29
6.1 L'attribut QueryInterceptor 29
6.2 L'attribut ChangeInterceptor 30
7 Conclusion 32

1 Introduction

Dans la continuité du chapitre précédent, ce chapitre nous permet d'approfondir nos connaissances dans la création d'un service de données ADO .NET, en abordant les points suivants :

- Création d'une source de données, exposée au travers du service de données.
- Implémentation des droits d'accès sur les entités et les opérations.
- La création d'opérations personnalisées.
- L'exposition de données non persistantes via les interfaces *IQueryable* et *IUpdatable*.

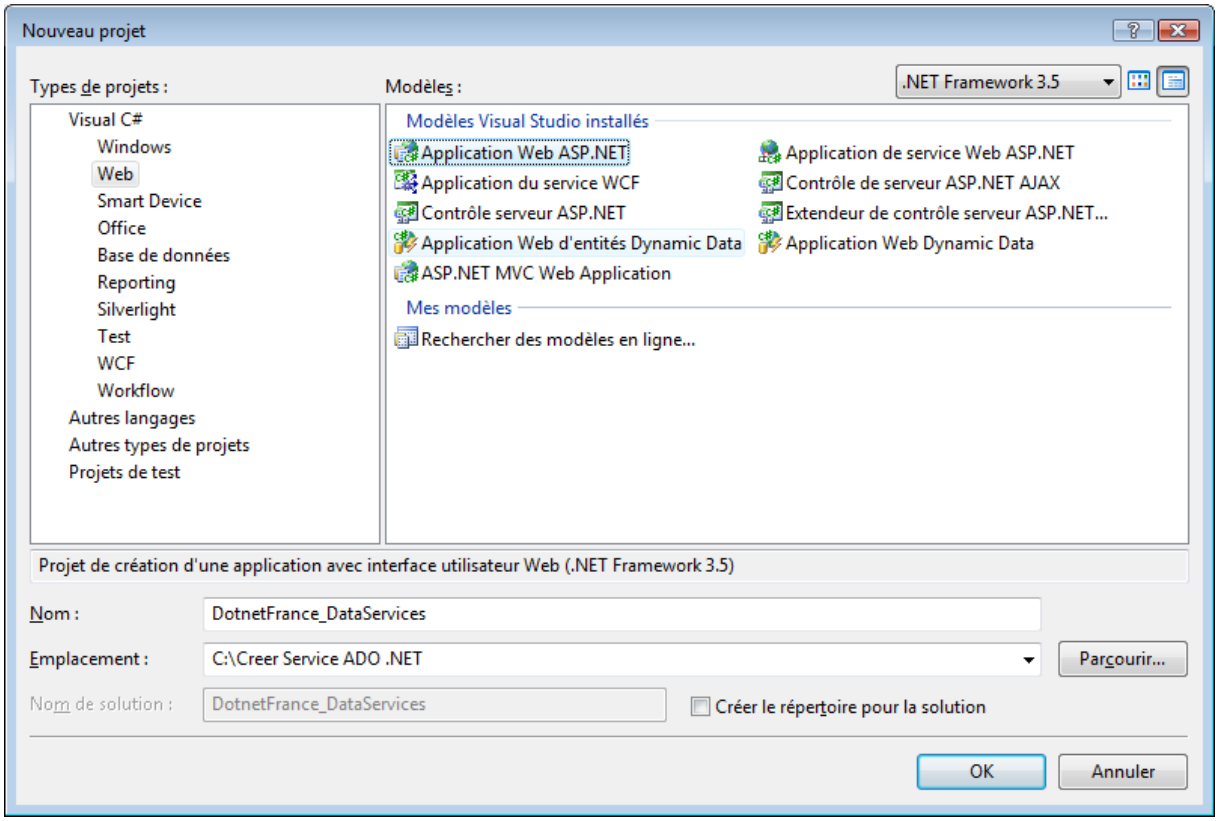
2 Création d'un service de données ADO .NET

2.1 Création d'un site Web

Pour créer notre service de données ADO .NET, nous devons créer un projet Web. Dans Visual Studio, il existe deux types de projet Web :

- Les applications Web.
- Les sites Web.

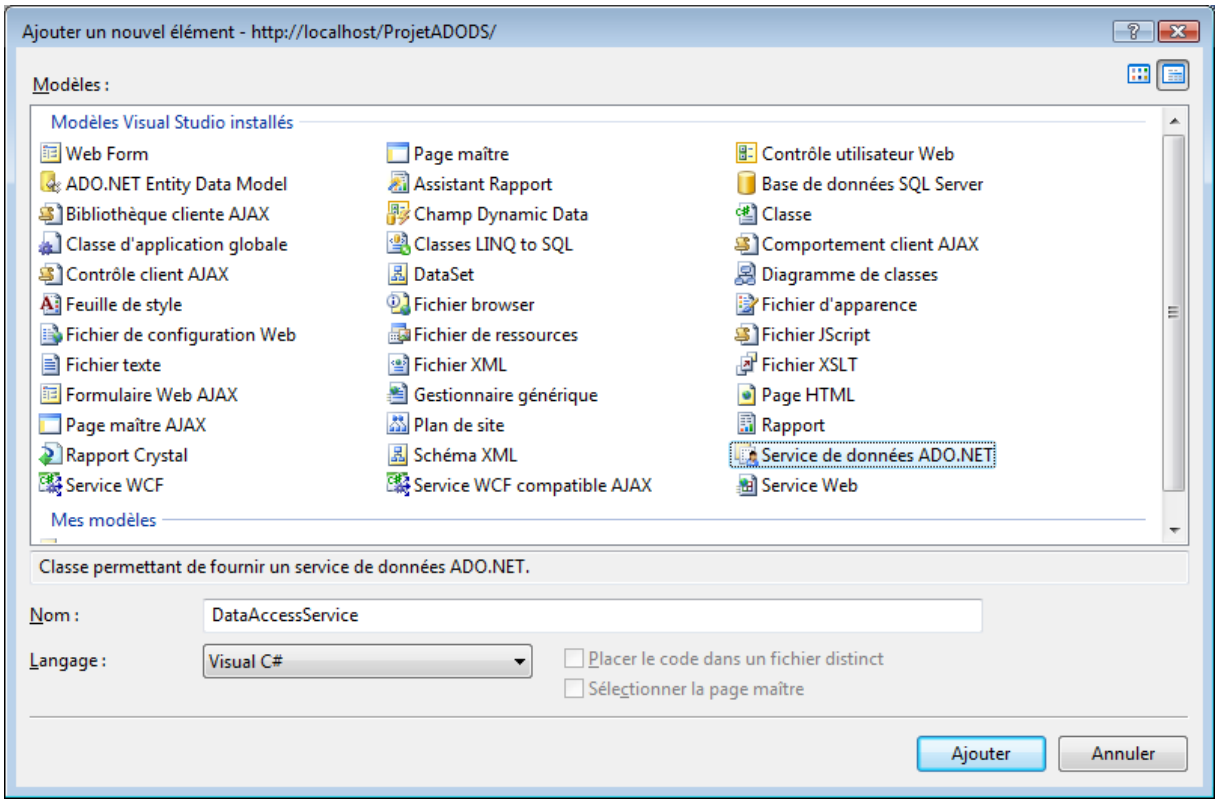
Voici un exemple, où nous allons créer une application Web. Nous appellerons ce projet *DotnetFrance_DataServices* :



2.2 Création du service de données ADO .NET

Créons notre service de données ADO .NET :

A la racine de l'application ASP .NET, ajoutons un nouvel élément de type « Services de données ADO.Net », que nous appellerons *DataAccessService* :



Par défaut, la classe du service de données ADO .NET, contient le bloc d'instructions suivant :

```
// C#

using System;
using System.Data.Services;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel.Web;

public class DataAccessService : DataService< /* TODO : placez ici le nom
de votre classe source de données */ >
{
    // Cette méthode n'est appelée qu'une seule fois pour initialiser les
stratégies au niveau des services.
    public static void InitializeService(IDataServiceConfiguration
config)
    {
        // TODO : définissez des règles pour indiquer les jeux d'entités
et opérations de service visibles, pouvant être mis à jour, etc.
        // Exemples :
        // config.SetEntitySetAccessRule("MyEntityset",
EntitySetRights.AllRead);
        // config.SetServiceOperationAccessRule("MyServiceOperation",
ServiceOperationRights.All);
    }
}
```

```
' VB.Net

Imports System
Imports System.Data.Services
Imports System.Collections.Generic
Imports System.Linq
Imports System.ServiceModel.Web

Public Class DataAccessService
    Inherits DataService(Of 'TODO : placez ici le nom de votre classe
                                                    source de données )
    ' Cette méthode n'est appelée qu'une seule fois pour initialiser les
                                    stratégies au niveau des services.
    Public Shared Sub InitializeService(ByVal config As
                                                    IDataServiceConfiguration)
        ' TODO : définissez des règles pour indiquer les jeux d'entités
      et opérations de service visibles, pouvant être mis à jour, etc.
        ' Exemples :
        'config.SetEntitySetAccessRule("MyEntityset",
                                                    EntitySetRights.AllRead)
        'config.SetServiceOperationAccessRule("MyServiceOperation",
                                            ServiceOperationRights.All)
    End Sub
End Class
```

2.3 Choix de la source de données

Le service de données que nous allons créer, doit utiliser une source de données « compatible ». Il est possible :

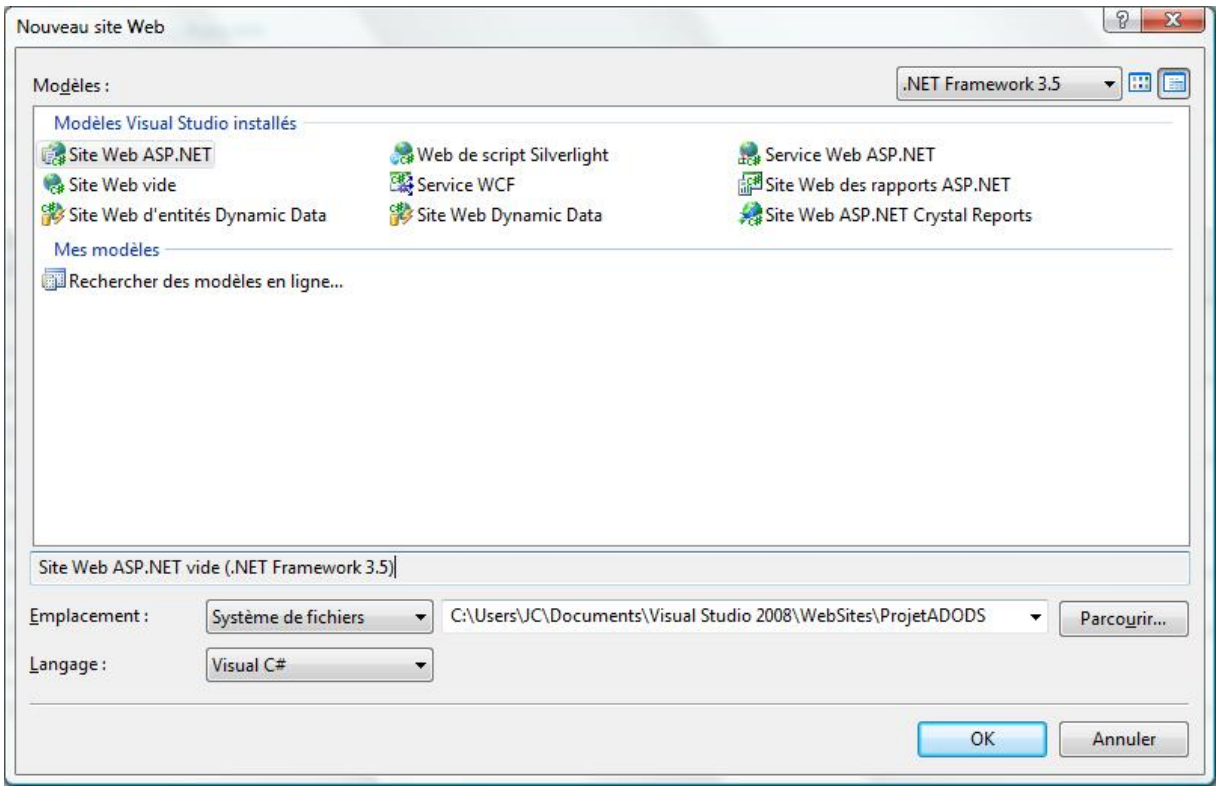
- D'utiliser le Framework Entity, proposée par Microsoft depuis la version 3.5 du Framework .NET.
- De créer sa propre source de données. Dans ce cas, les classes exposées doivent implémenter les interfaces suivantes du Framework .NET :
    - *System.Linq.Queryable*, de manière à pouvoir être requêtée.
    - *System.Linq.IUpdatable*, de manière à pouvoir être mise à jour.

Dans ce cours, nous mettrons en œuvre successivement ces deux sources de données.

3 Utilisation du Framework Entity comme source de données

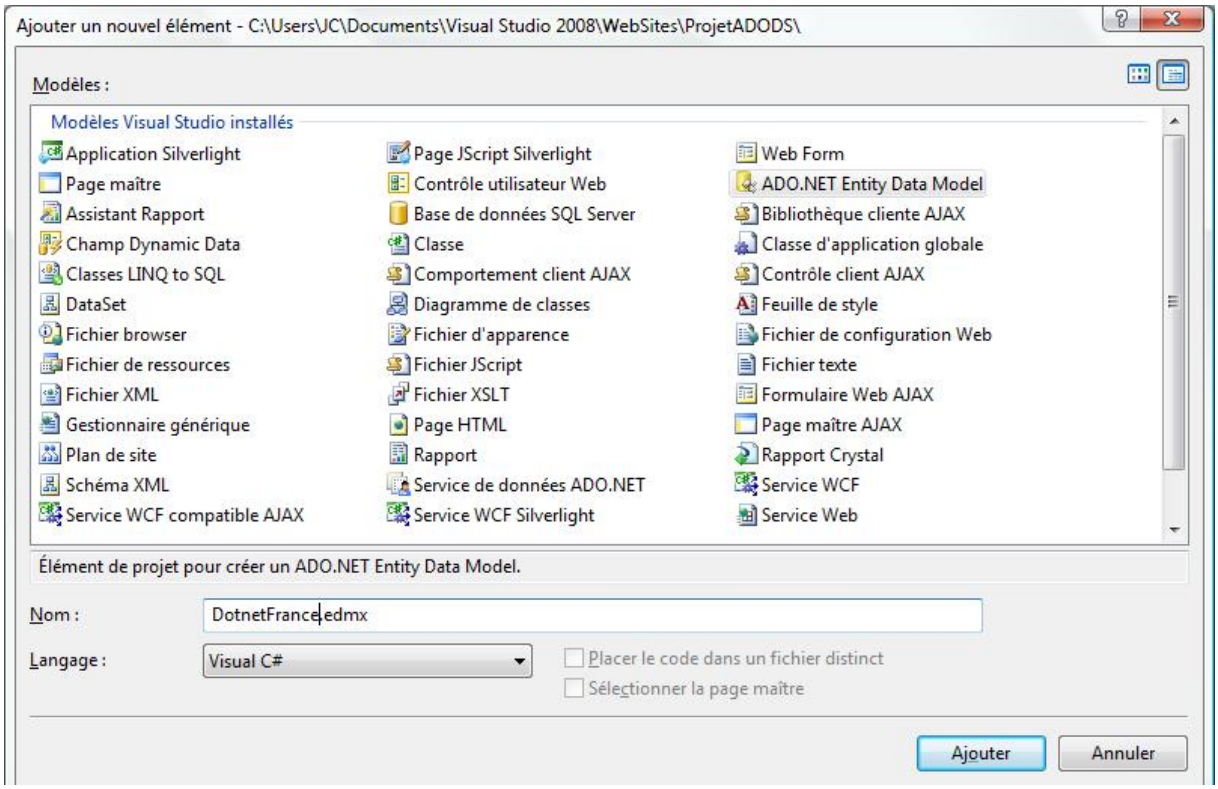
Dans ce chapitre nous allons créer un site web ASP.Net dans lequel nous intégrerons notre service de données ADO .Net. Cela nous permettra de faire quelques tests dans un navigateur, sans entrer trop en détail dans la consommation du service.

Créons donc un nouveau site web ASP.Net dans Visual Studio 2008 :

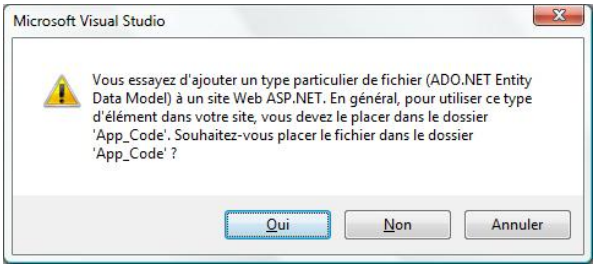


Créons ensuite notre Entity Data Model pour accéder à nos données. En effet, l'Entity Data Model est un composant du Framework Entity, qui permet d'exposer des données contenues dans une source de données, sous forme d'entités.

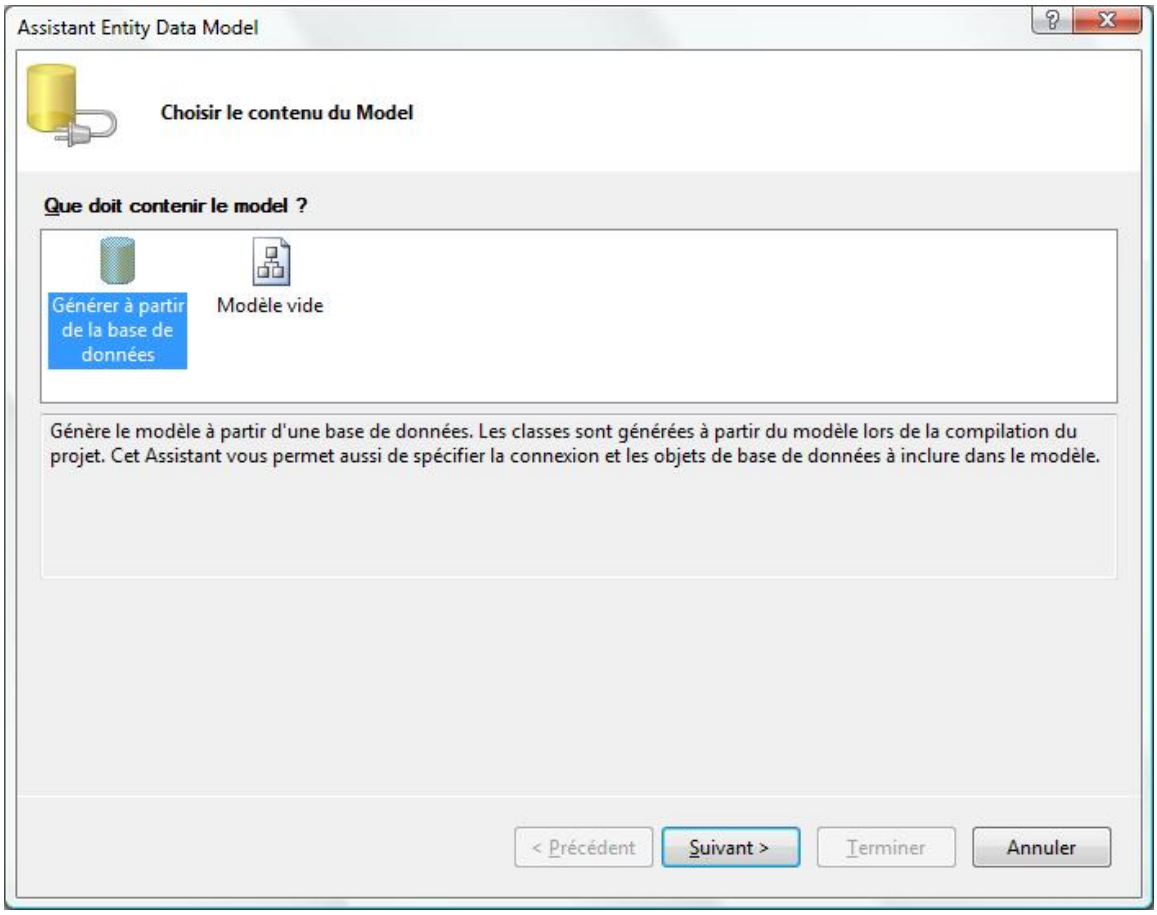
Via le menu contextuel sur notre site web dans l'explorateur de solution, cliquez sur votre projet Web puis sur « ajouter un nouvel élément… ». Cliquez ensuite sur l'item « ADO.Net Entity Data Model ». La fenêtre suivante apparaît :



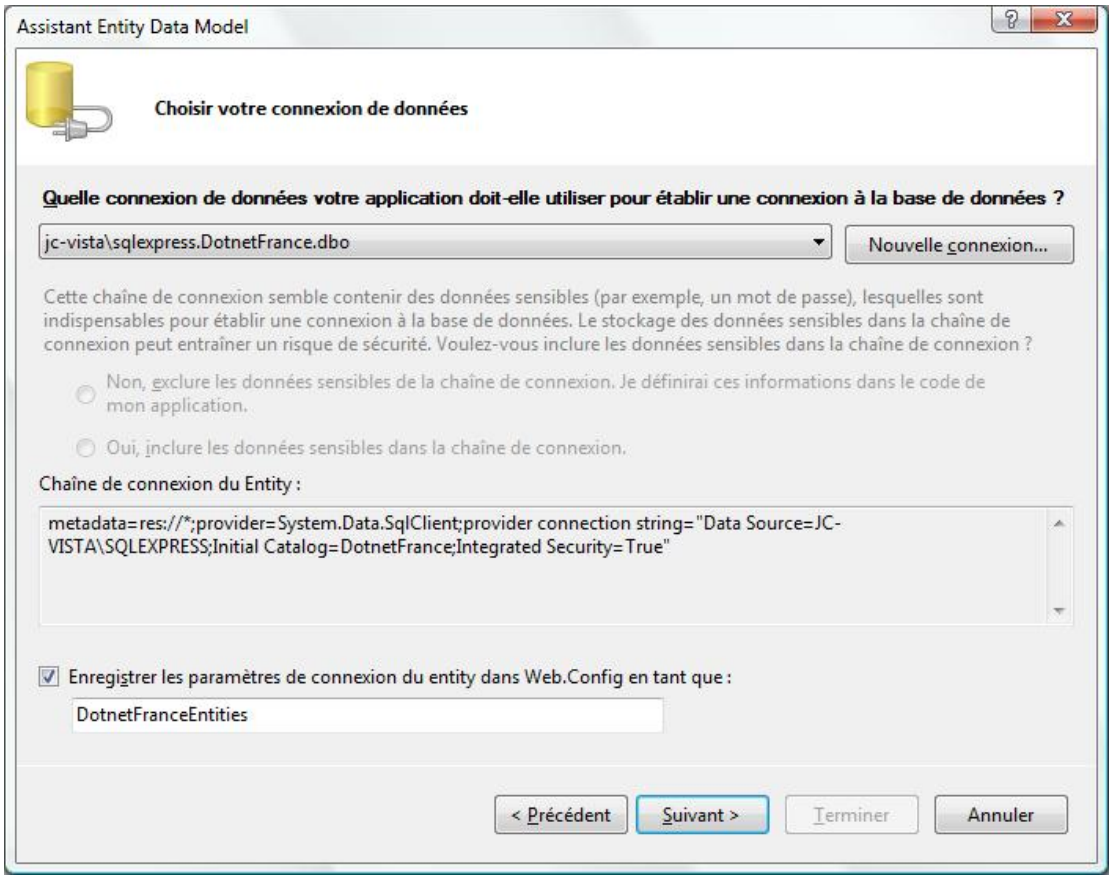
Une fenêtre s'ouvre, cliquez sur oui.



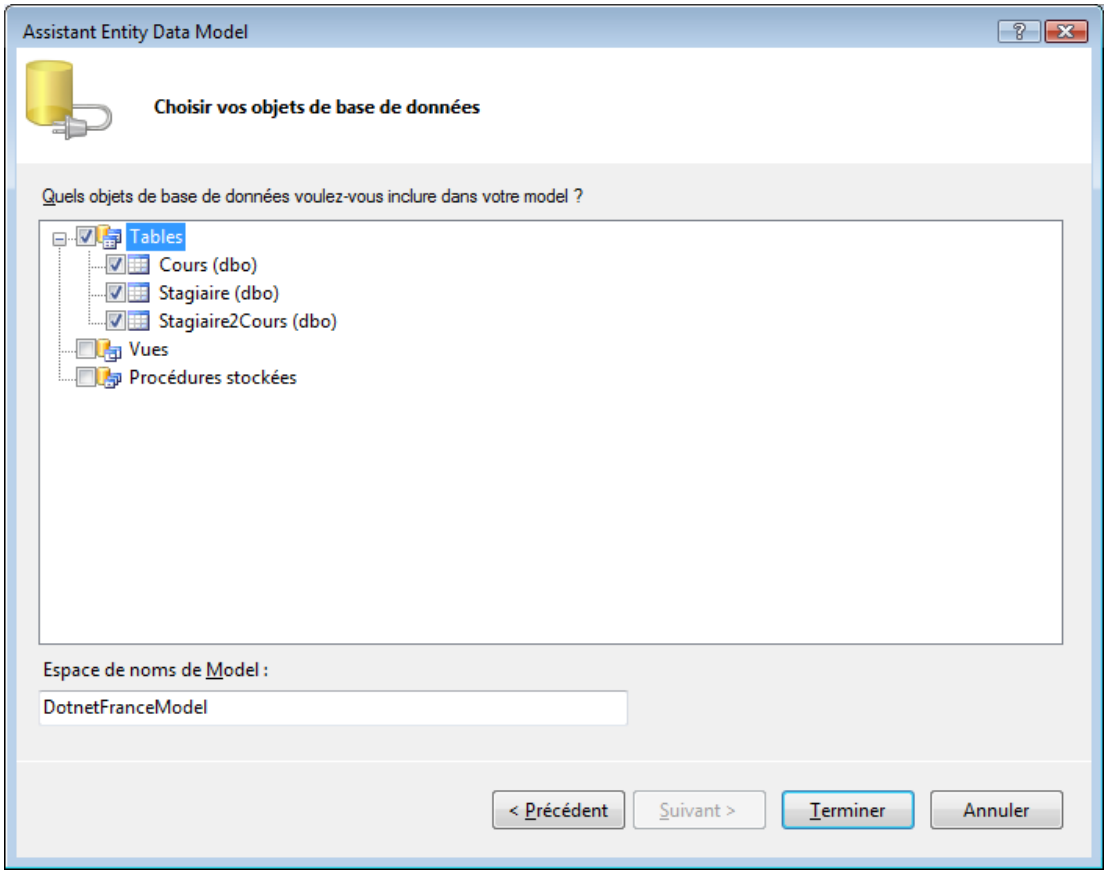
Une fenêtre de l'assistant permettant la création d'un Entity Data Model apparaît. Choisissez donc l'option pour générer depuis votre base de données.



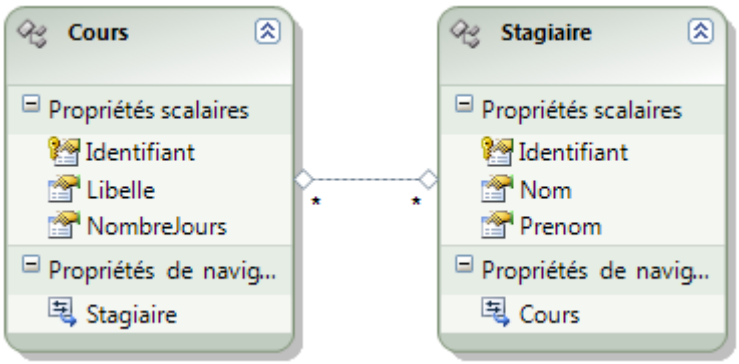
Dans la fenêtre, choisissez l'item « Générer à partir de la base de données », puis cliquez sur le bouton « Suivant ». La fenêtre suivante apparaît :



Dans la liste des connexions, choisissez la connexion permettant d'accéder à votre base de données. Si elle n'existe pas, vous pouvez la créer en cliquant sur le bouton « Nouvelle Connexion ». Une fois la connexion sélectionnée, cliquez sur le bouton « Suivant » :



Sélectionner l'ensemble des tables de la base données. Puis cliquez sur le bouton Terminer. On obtient donc le modèle d'entités suivant :



Pensez aussi à vérifier la chaîne de connexion stockée dans le fichier de configuration. Elle a été créée à partir des informations de la connexion. Vous pouvez toujours changer ces paramètres : nom de l'instance SQL attaquée, nom de la base de données, type d'authentification (Windows, SQL Server …).

4 Définition des droits

Nous allons donc maintenant nous occuper de personnaliser ce service. Commençons d'abord par mettre en place notre source de données. Donc notre modèle d'entités dans l'objet DataService.

```
// C#

public class WebDataService :
DataService<DotnetFranceModel.DotnetFranceEntities>
{
    // Cette méthode n'est appelée qu'une seule fois pour initialiser les
stratégies au niveau des services.
    public static void InitializeService(IDataServiceConfiguration
config)
    {
        // TODO : définissez des règles pour indiquer les jeux d'entités
et opérations de service visibles, pouvant être mis à jour, etc.
        // Exemples :
        // config.SetEntitySetAccessRule("MyEntityset",
EntitySetRights.AllRead);
        // config.SetServiceOperationAccessRule("MyServiceOperation",
ServiceOperationRights.All);
    }
}
```

```
' VB.Net

Public Class WebDataService
    Inherits DataService(Of DotnetFranceModel.DotnetFranceEntities)
    ' Cette méthode n'est appelée qu'une seule fois pour initialiser les
stratégies au niveau des services.
    Public Shared Sub InitializeService(ByVal config As
IDataServiceConfiguration)
        ' TODO : définissez des règles pour indiquer les jeux d'entités
et opérations de service visibles, pouvant être mis à jour, etc.
        ' Exemples :
        ' config.SetEntitySetAccessRule("MyEntityset",
EntitySetRights.AllRead);
        ' config.SetServiceOperationAccessRule("MyServiceOperation",
ServiceOperationRights.All);
    End Sub
End Class
```

Nous allons maintenant pouvoir nous intéresser aux deux lignes de configuration commentées. Elles permettent :

- De définir les droits d'accès sur les entités.
- De définir des droits d'accès sur les opérations personnalisées, qu'il sera possible d'exécuter.

4.1 Définition de droits sur les entités

L'instruction *config.SetEntitySetAccessRule* nous permet de gérer des droits d'accès aux entités. Elle accepte deux paramètres :

- Une chaîne portant le nom de l'entité sur laquelle nous souhaitons appliquer des droits.
- Les droits que l'on souhaite lui appliquer.

Voici les différentes valeurs que peut prendre EntitySetRights :

Valeur	***Description***
None	Permet d'interdire tout accès aux données
ReadSingle	Permet de lire des éléments de données uniques
ReadMultiple	Permet de lire des groupes de données
WriteAppend	Permet de créer des éléments de données dans des groupes de données
WriteReplace	Permet de remplacer des données
WriteDelete	Permet de supprimer des éléments de données dans des groupes de données
WriteMerge	Permet de fusionner des données
AllRead	Permet de lire des données
AllWrite	Permet d'écrire des données
All	Permet de créer, lire, mettre à jour et supprimer des données

On peut appliquer ces valeurs :

- De manière unitaire :

C#:

```
config.SetEntitySetAccessRule("*", EntitySetRights.All);
```

VB.Net:

```
config.SetEntitySetAccessRule("*", EntitySetRights.All);
```

- De manière combinatoire :

C#:

```
config.SetEntitySetAccessRule("*",EntitySetRights.WriteDelete|EntitySetRi
ghts. WriteAppend);
```

VB.Net:

```
config.SetEntitySetAccessRule("*",EntitySetRights.WriteDelete          Or
EntitySetRights. WriteAppend)
```

- Sur plusieurs éléments :

C#:

```
config.SetEntitySetAccessRule("Stagiaire",EntitySetRights.WriteDelete|Ent
itySetRights. WriteAppend);
config.SetEntitySetAccessRule("Cours",EntitySetRights.AllRead);
```

VB.Net:

```
config.SetEntitySetAccessRule("Stagiaire",EntitySetRights.WriteDelete   Or
EntitySetRights. WriteAppend);
config.SetEntitySetAccessRule("Cours",EntitySetRights.AllRead);
```

4.2 Création d'opérations de services et gestion de leurs droits

L'instruction *config.SetServiceOperationAccessRule* permet de gérer des opérations. Elle accepte aussi deux paramètres :

- Une chaîne portant le nom de l'opération.
- Les droits que l'on souhaite lui appliquer.

Voici les différentes valeurs que peut prendre ServiceOperationRights :

Valeur	***Description***
None	Permet d'interdire tout accès aux données
ReadSingle	Permet de lire un élément de données unique grâce à l'opération
ReadMultiple	Permet de lire plusieurs éléments de données grâce à l'opération
AllRead	Permet de lire des éléments de données uniques ou multiples déployés par l'opération
All	Permet d'utiliser tous les droits attribués à l'opération de service

De la même manière que pour EntitySetRights, nous pouvons l'appeler de plusieurs manières :

- De manière unitaire :

C#:

```
config.SetServiceOperationAccessRule("MonOperation",
ServiceOperationRights.All);
```

VB.Net:

```
config.SetServiceOperationAccessRule("MonOperation",
ServiceOperationRights.All);
```

- Sur plusieurs opérations :

C#:

```
config.SetServiceOperationAccessRule("MonOperation",
ServiceOperationRights.All);
config.SetServiceOperationAccessRule("UneAutreOperation",
ServiceOperationRights.ReadSingle);
```

VB.Net:

```
config.SetServiceOperationAccessRule("MonOperation",
ServiceOperationRights.All);
config.SetServiceOperationAccessRule("UneAutreOperation",
ServiceOperationRights.ReadSingle);
```

Voyons maintenant la création d'une opération. Une opération est une méthode :

- Qui peut être une procédure ou une fonction. S'il s'agit d'une fonction, alors elle ne peut retourner un type de données :
  - De type primitif, tel qu'une chaîne de caractères, un nombre numérique ou un booléen.
  - Correspondant au type d'une entité du modèle de données, exposé par le service de données ADO .NET.
  - Implémentant l'interface *IEnumerable<T> ou IQueryable<T>*, où *T* d'une entité du modèle de données, exposé par le service de données ADO .NET.
- N'acceptant que des paramètres d'entrée (C# - pas de paramètres *out*), où chaque type des paramètres doit être un type primitif.
- Définie avec la métadonnée :
  - *WebGet*, pour pouvoir demander son exécution en mode GET.
  - *WebInvoke*, pour pouvoir demander son exécution en mode POST, PUT ou DELETE.

Les métadonnées *WebGet* et *WebInvoke* permettent de définir des propriétés intrinsèques du service de données, telles que le format de la requête et de la réponse envoyée, …

Par exemple, créons une fonction de tri des 10 derniers cours par ordre décroissant des identifiants.

```
// C#

[WebGet]
public IQueryable<Cours> triParId()
{
return from c in this.CurrentDataSource.Cours
      where c.Identifiant =< 10
      orderby c.Identifiant descending
      select c;
}
```

```
'VB.Net

<WebGet()> _
Public Function triParId() As IQueryable(Of Cours)
    Return From c In Me.CurrentDataSource.Cours _
        Where c.Identifiant =< 10 _
        Order By c.Identifiant Descending _
        Select c
End Function
```

Nous déclarerons donc notre opération de la manière suivante :

C# ou VB.Net :

```
config.SetServiceOperationAccessRule("triParId",
ServiceOperationRights.All);
```

Si vous voulez testez l'opération, faites un clic droit sur votre Service.svc puis « afficher dans le navigateur ». Rajoutez à la fin de votre url le nom de votre opération.
Dans notre cas, on obtiendra quelque chose du type : http://localhost:49627/ProjetADODS/WebDataService.svc/triParId.

5 Exposition de données non persistantes

5.1 Présentation de l'exercice

Nous allons voir dans cette partie comme exposer des données, de manière autre qu'au travers du Framework Entity. Pour montrer un exemple concret, nous allons exposer des informations sur les processus s'exécutant sur le serveur.

5.1.1 La classe ProcessView

Nous commençons par créer une classe nommée *ProcessView*, qui correspond au type de données, utilisé pour retourner des données sur les processus. Cette classe ne définit que deux propriétés (en C#, vous remarquerez l'utilisation d'accesseurs simplifiés) :

```
// C#

namespace DotnetFrance_DataServices.Processes
{
    public class ProcessView
    {
        public int ProcessViewID { get; set; }
        public string ProcessName { get; set; }
    }
}
```

```
' VB .NET

Namespace Processes
    Public Class ProcessView
        Private _ProcessViewID As Integer
        Public Property ProcessViewID() As Integer
            Get
                Return _ProcessViewID
            End Get
            Set(ByVal value As Integer)
                _ProcessViewID = value
            End Set
        End Property

        Private _ProcessName As String
        Public Property ProcessName() As String
            Get
                Return _ProcessName
            End Get
            Set(ByVal value As String)
                _ProcessName = value
            End Set
        End Property
    End Class
End Namespace
```

5.1.2 Création du service Métier

Pour requêter une source de données, il faut qu'elle soit exposée au travers de l'interface *System.Linq.IQueryable* du Framework .NET, afin de pouvoir utiliser les opérateurs de requête LINQ standard. Dans notre cas, nous allons écrire une requête LINQ, retournant une liste d'objet *ProcessView*. Ainsi, pour exécuter cette requête, il nous faudra utiliser la méthode *AsQueryable*.

```
// C#

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Diagnostics;

namespace DotnetFrance_DataServices.Processes
{
    public class DonneesPerso
    {
        public DonneesPerso()
        {
        }

        public IQueryable<ProcessView> GetListeProcessus
        {
            get
            {
                return Process.GetProcesses().Select(
                    p => new ProcessView
                    {
                        ProcessViewID = p.Id,
                        ProcessName = p.ProcessName
                    }).AsQueryable();
            }
        }
    }
}
```

```
' VB .NET

Namespace Processes
    Public Class DonneesPerso
        Public Sub New()
        End Sub

        Public ReadOnly Property Processus() As IQueryable(Of
ProcessView)
            Get
            Return Process.GetProcesses().Select(Function(p) New
ProcessView()).AsQueryable()
            End Get
        End Property
    End Class
End Namespace
```

5.1.3 Exposition du service métier via le service de données ADO .NET

Dans la classe code-behind de notre service de données ADO .NET :

```
// C#

using System;
using System.Collections.Generic;
using System.Data.Services;
using System.Linq;
using System.ServiceModel.Web;
using System.Web;

using DotnetFrance_DataServices.Processes;

namespace DotnetFrance_DataServices
{
    public class DataAccessService : DataService<DonneesPerso>
    {
        // Cette méthode n'est appelée qu'une seule fois pour initialiser
les stratégies au niveau des services.
        public static void InitializeService(IDataServiceConfiguration
config)
        {
            config.SetEntitySetAccessRule("*", EntitySetRights.All);
            config.SetServiceOperationAccessRule("*",
ServiceOperationRights.All);
        }
    }
}
```

```
' VB.Net

Imports System
Imports System.Data.Services
Imports System.Linq
Imports System.Diagnostics

Public Class WebDataService
    Inherits DataService(Of DonneesPersos)
    Public Shared Sub InitializeService(ByVal config As
IDataServiceConfiguration)
        config.SetEntitySetAccessRule("*", EntitySetRights.All)
        config.SetServiceOperationAccessRule("*",
ServiceOperationRights.All)
    End Sub
End Class
```

De la même manière, si vous souhaitez tester l'affichage de cette source de données, cliquez droit sur votre service, puis sur afficher dans le navigateur. Et ajoutez "GetListeProcessus" après votre service. Une adresse de ce type :

http://localhost:49627/ProjetADODS/WebDataService.svc/Processus

Nous allons voir dans la partie suivante comment modifier de telles données avec un autre exemple.

5.2 Utilisation de l'interface IUpdatable

Voyons donc comment appliquer les concepts du CRUD (Create, Read, Update, Delete) sur nos données. Nous allons prendre un exemple similaire au précédent et requêter sur un objet personnalisé.

Voici le code de notre service :

```
using System;
using System.Data.Services;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel.Web;

public class WebDataService :
DataService<context.InformationsDataContext>
{
    public static void InitializeService(IDataServiceConfiguration
                                                                config)
    {
        config.SetEntitySetAccessRule("*",  EntitySetRights.All);
    }
}
```

```
Imports System
Imports System.Data.Services
Imports System.Collections.Generic
Imports System.Linq
Imports System.ServiceModel.Web

Public Class WebDataService
    Inherits DataService(Of context.InformationsDataContext)
    Public Shared Sub InitializeService(ByVal config As
                                            IDataServiceConfiguration)
        config.SetEntitySetAccessRule("*", EntitySetRights.All)
    End Sub
End Class
```

Nous allons créer une classe de données personnalisées, et nous créerons le service de données depuis son contexte de données.

Intéressons nous au code de cette classe de données. Elle sera similaire à notre table Stagiaire, avec une liste contenant identifiants, noms et prénoms de plusieurs stagiaires.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Services.Common;
using System.Linq.Expressions;

namespace Information
{
    [DataServiceKey("id")]
    public class Informations
    {
        public int id { get; set; }
        public string Nom { get; set; }
        public string Prenom { get; set; }


        public static List<Informations> GetInformation()
        {
           return new List<Informations>() {
               new Informations() {id=1, Nom="Deroux", Prenom = "Alain"},
               new Informations() {id=2, Nom="Ravaille", Prenom="James"},
               new Informations(){id=3, Nom="Siron", Prenom="Karl"}
            };
        }
    }
}
```

Lors de la création d'un objet IQueryable personnalisé il faut penser à ce qu'il ait une "clef d'entité". Pour ce faire, il suffit d'ajouter la métadonnée "DataServiceKey" à votre classe en lui fournissant comme argument le nom de votre clef d'entité. Par exemple dans mon cas :

En VB.Net :

```
<DataServiceKey("id")>
```

En C# :

```
[DataServiceKey("id")]
```

Créons ensuite le contexte de données :

```
// C#

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Services;
using System.Reflection;
using Information;

namespace context
{
    public class InformationsDataContext : IUpdatable
    {
        private static List<Informations> _Informations;

        public IQueryable<Informations> Information
        {
            get
            {
                return _Informations.AsQueryable();
            }
        }

        static InformationsDataContext()
        {
            _Informations = Informations.GetInformation();
        }
      //#Implémentation de l'interface IUpdatable#
     }
}
```

```
' VB .NET

Imports System
Imports System.Collections.Generic
Imports System.Linq
Imports System.Web
Imports System.Data.Services
Imports System.Reflection
Imports Information

Namespace context
    Public Class InformationsDataContext
        Implements IUpdatable
        Private Shared _Informations As List(Of Informations)

        Public ReadOnly Property Information() As IQueryable(Of
Informations)
            Get
                Return _Informations.AsQueryable()
            End Get
        End Property

        Shared Sub New()
            _Informations = Informations.GetInformation()
        End Sub
    End Class
    '#Implémentation de l'interface IUpdatable#
End Namespace
```

Implémentons maintenant l'interface *IUpdatable<T>*, puis faisons quelques modifications pour implémenter les fonctions qui nous intéressent.

Dans la zone commentée au-dessus on retrouvera un bloc de code de la forme suivante :

```
        public object CreateResource(string containerName, string
                                                            fullTypeName)
        {
            var objType = Type.GetType(fullTypeName);
            var resourceToAdd = Activator.CreateInstance(objType);
            _Informations.Add((Informations)resourceToAdd);
            return resourceToAdd;
        }

        public void DeleteResource(object targetResource)
        {
            _Informations.Remove((Informations)targetResource);
        }

        public object GetResource(IQueryable query, string fullTypeName)
        {
            object result = null;
            var enumerator = query.GetEnumerator();
            while (enumerator.MoveNext())
            {
                if (enumerator.Current != null)
                {
                    result = enumerator.Current;
                    break;
                }
            }
            if (fullTypeName != null &&
                        !fullTypeName.Equals(result.GetType().FullName))
            {
                throw new DataServiceException();
            }
            return result;
        }

        public object GetValue(object targetResource, string
                                                            propertyName)
        {
            var targetType = targetResource.GetType();
            var targetProperty = targetType.GetProperty(propertyName);
            return targetProperty.GetValue(targetResource, null);
        }

        public void SetValue(object targetResource, string propertyName,
                                                    object propertyValue)
        {
            Type targetType = targetResource.GetType();
            PropertyInfo property = targetType.GetProperty(propertyName);
            property.SetValue(targetResource, propertyValue, null);
        }
```

```
        public object ResolveResource(object resource){
            return resource;
        }
        public void SaveChanges(){}
        public void SetReference(object targetResource, string
                                    propertyName, object propertyValue){
            throw new NotImplementedException();
        }
        public object ResetResource(object resource){
            throw new NotImplementedException();
        }
        public void ClearChanges(){
            throw new NotImplementedException();
        }
        public void AddReferenceToCollection(object targetResource,
                        string propertyName, object resourceToBeAdded){
            throw new NotImplementedException();
        }
        public void RemoveReferenceFromCollection(object targetResource,
                        string propertyName, object resourceToBeRemoved){
            throw new NotImplementedException();
        }
```

```
Public Function CreateResource(ByVal containerName As String, ByVal
                                    fullTypeName As String) As Object
    Dim objType = Type.[GetType](fullTypeName)
    Dim resourceToAdd = Activator.CreateInstance(objType)
    _Informations.Add(DirectCast(resourceToAdd, Informations))
    Return resourceToAdd
End Function
Public Sub DeleteResource(ByVal targetResource As Object)
    _Informations.Remove(DirectCast(targetResource, Informations))
End Sub
Public Function GetResource(ByVal query As IQueryable, ByVal fullTypeName
                                                    As String) As Object
    Dim result As Object = Nothing
    Dim enumerator = query.GetEnumerator()
    While enumerator.MoveNext()
        If enumerator.Current IsNot Nothing Then
            result = enumerator.Current
            Exit While
        End If
    End While
    If fullTypeName IsNot Nothing AndAlso Not
                fullTypeName.Equals(result.[GetType]().FullName) Then
        Throw New DataServiceException()
    End If
    Return result
End Function
Public Function GetValue(ByVal targetResource As Object, ByVal
                                    propertyName As String) As Object
    Dim targetType = targetResource.[GetType]()
    Dim targetProperty = targetType.GetProperty(propertyName)
    Return targetProperty.GetValue(targetResource, Nothing)
End Function
Public Sub SetValue(ByVal targetResource As Object, ByVal propertyName As
String, ByVal propertyValue As Object)
    Dim targetType As Type = targetResource.[GetType]()
    Dim [property] As PropertyInfo = targetType.GetProperty(propertyName)
            [property].SetValue(targetResource, propertyValue, Nothing)
End Sub
```

```
Public Function ResolveResource(ByVal resource As Object) As Object
    Return resource
End Function
Public Sub SaveChanges()
End Sub
Public Sub SetReference(ByVal targetResource As Object, ByVal
                  propertyName As String, ByVal propertyValue As Object)
    Throw New NotImplementedException()
End Sub
Public Function ResetResource(ByVal resource As Object) As Object
    Throw New NotImplementedException()
End Function
Public Sub ClearChanges()
    Throw New NotImplementedException()
End Sub
Public Sub AddReferenceToCollection(ByVal targetResource As Object, ByVal
           propertyName As String, ByVal resourceToBeAdded As Object)
    Throw New NotImplementedException()
End Sub
Public Sub RemoveReferenceFromCollection(ByVal targetResource As Object,
ByVal propertyName As String, ByVal resourceToBeRemoved As Object)
    Throw New NotImplementedException()
End Sub
```

Voici un bref récapitulatif des méthodes de l'interface IUpdatable :

Nom	Description	Arguments
AddReferenceToCollection()	Ajoute la valeur passée en argument à la collection	***Object targetResource*** : définit la propriété ***string propertyName*** : nom de la propriété de collection à laquelle la ressource sera ajoutée ***Object resourceToBeAdded*** : Objet opaque représentant la ressource à ajouter
ClearChanges()	Annule une modification	
CreateResource()	Créer une ressource	***String containerName*** : nom du jeu d'entités auquel la ressource appartient ***String fullTypeName*** : nom complet du type de données
DeleteResource()	Supprime une ressource	***Object TargetResource*** : Ressource à supprimer
GetResource()	Accesseur à une ressource	***IQueryable query*** : requête linq vers la ressource qui nous intéresse ***String fullTypeName*** : Nom de type complet de la ressource
GetValue()	Accesseur à la valeur d'une propriété	***Object targetResource*** : représente la ressource ***String propertyName*** : nom de la propriété

RemoveReferenceFromCollection()	Supprime une valeur de la collection	***Object targetResource*** : objet cible ***String propertyName*** : nom de la propriété ***Object ResourceToBeRemoved*** : valeur de la propriété à supprimer
ResetResource()	Met à jour une ressource	***Object Resource*** : ressource à mettre à jour
ResolveResource()	Retourne l'instance d'une ressource	***Object Resource*** : Resource dont on doit retourner l'interface
SaveChanges()	Sauvegarde toutes les opérations effectuées	
SetReference()	Définit la référence de propriété sur l'objet spécifié	***Object targetResource*** : objet définissant la propriété ***String propertyName*** : Nom de la propriété devant être mise à jour ***Object propertyValue*** : Valeur de la propriété à mettre à jour
SetValue()	Affecte une valeur de propriété spécifique à une propriété spécifiée	***Object targetResource*** : objet définissant la propriété ***String propertyName*** : Nom de la propriété devant être mise à jour ***Objet propertyValue*** : Valeur de la propriété pour la mise à jour

Testons maintenant nos méthode dans une application console DOS :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Services.Client;
using ConsoleApplication1.ServiceReference1;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            var proxy = new InformationsDataContext(new
Uri("http://localhost:52857/IUpdatableT/WebDataService.svc/"));
            proxy.MergeOption = MergeOption.AppendOnly;
            var StInfos = new Informations
            {
                id = 4,
                Nom = "Vaselon",
                Prenom = "Jean-Christophe",
            };
            proxy.AddToInformation(StInfos);
            proxy.SaveChanges();
            Afficher(proxy);

            StInfos.Nom = "Vasselon";
            proxy.UpdateObject(StInfos);
            proxy.SaveChanges();
            Afficher(proxy);

            proxy.DeleteObject(StInfos);
            proxy.SaveChanges();
            Afficher(proxy);

            Console.Read();
        }
        private static void Afficher(InformationsDataContext proxy)
        {
            var info = from infos in proxy.Information select infos;
            foreach (Informations infos in info)
            {
                Console.WriteLine(
                    "Stagiaire n°{0}, Nom : {1}, Prenom : {2}",
                    infos.id,
                    infos.Nom,
                    infos.Prenom
                    );
            }
            Console.WriteLine();
        }
    }
}
```

```
Imports System
Imports System.Collections.Generic
Imports System.Linq
Imports System.Text
Imports System.Data.Services.Client
Imports ConsoleApplication1.ServiceReference1

Namespace ConsoleApplication1
    Class Program
        Private Shared Sub Main(ByVal args As String())
            Dim proxy = New InformationsDataContext(New
      Uri("http://localhost:52857/IUpdatableT/WebDataService.svc/"))
            proxy.MergeOption = MergeOption.AppendOnly
            Dim StInfos = New Informations()
            proxy.AddToInformation(StInfos)
            proxy.SaveChanges()
            Afficher(proxy)

            StInfos.Nom = "Vasselon"
            proxy.UpdateObject(StInfos)
            proxy.SaveChanges()
            Afficher(proxy)

            proxy.DeleteObject(StInfos)
            proxy.SaveChanges()
            Afficher(proxy)
            Console.Read()
        End Sub
        Private Shared Sub Afficher(ByVal proxy As
                                              InformationsDataContext)
            Dim info = From infos In proxy.Information _
                Select infos
            For Each infos As Informations In info
                Console.WriteLine("Stagiaire n°{0}, Nom : {1}, Prenom :
                            {2}", infos.id, infos.Nom, infos.Prenom)
            Next
            Console.WriteLine()
        End Sub
    End Class
End Namespace
```

Ce code fait simplement un ajout d'un nouveau stagiaire, sauvegarde cette manipulation puis l'affiche.

MergeOption.AppendOnly nous évite de recharger les données déjà stockées dans le contexte.

Ensuite nous créonsun nouvel objet de type Informations dans lequel on stockera les informations d'un nouveau stagiaire.

Il nous suffira ensuite de le passer en argument aux fonctions qui nous intéresse.

En faisant une analogie avec un tableau, l'objet StInfos représente une ligne contenant tous les champs de notre tableau. De cette manière on sait exactement quoi modifier ou ajouter, comment et où placer nos données dans ce tableau.

A chaque modification, on appelle la méthode *SaveChanges()* pour l'enregistrer dans la source réelle de données.

La fonction *Afficher()* est là pour illustrer l'exemple en montrant les modifications.

Nous retrouvons donc le principe du CUD (Create, Update, Delete) avec les 3 méthodes :

- AddToObject() : Pour la création d'une nouvelle ressource
- Update() : Pour la mise à jour
- DeleteObject() : Pour la suppression

6 Introduction à la sécurité avec ADO.Net Data Services

Comme nous avons le voir, n'importe qui a accès à nos ressources et à leurs modifications. Nous pouvons leur attribuer des droits spécifiques aux ressources (All, AllRead, etc), mais l'utilisation des interceptors est plus intéressante.

Les interceptors se placent dans le code de notre service de données ADO .NET. On peut les cumuler en fonction de ce qu'on veut intercepter.

Leur principal intêret est qu'ils se déclenchent lors d'une action précise et retournent les résultats que l'on veut que le service retourne dans ce cas précis.

On trouve deux types d'interceptors :

- Les QueryInterceptors : ils se déclenchent lors de la recherche d'une ressource
- Les ChangeInterceptors : ils se déclenchent lors de la modification ou de l'ajout d'une ressource.

Voyons maintenant comment les créer.

6.1 L'attribut QueryInterceptor

L'attribut *QueryInterceptor* permet de limiter l'affichage des données. Ainsi nous pourrons limiter l'affichage de certaines données lors de recherches précises.

```
//C#
using System;
using System.Data.Services;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel.Web;
using System.Linq.Expressions;
using DotNetFranceModel;

public class DataAccessService : DataService<DotNetFranceEntities>
{
      public static void InitializeService(IDataServiceConfiguration
                                                                  config)
    {
        config.SetEntitySetAccessRule("*", EntitySetRights.All);
        config.SetServiceOperationAccessRule("*",
                                            ServiceOperationRights.All);
    }

    [QueryInterceptor("Cours")]
    public Expression<Func<Cours, bool>>
    InterceptionCours()
    {
        return c => c.Identifiant <= "5";
        //Retournera les 5 derniers cours
    }
}
```

```
'VB.Net
Imports System
Imports System.Data.Services
Imports System.Collections.Generic
Imports System.Linq
Imports System.ServiceModel.Web
Imports System.Linq.Expressions

Public Class Service1
    Inherits DataService(Of TestAstoriaModel.TestAstoriaEntities)
    Public Shared Sub InitializeService(ByVal config As
IDataServiceConfiguration)
        config.SetEntitySetAccessRule("*", EntitySetRights.All)
        config.SetServiceOperationAccessRule("*",
ServiceOperationRights.All)
    End Sub

<QueryInterceptor("Cours")> _
Public Function InterceptionCours() As Expression(Of Func(Of Cours,
Boolean))
    Return Function(c) c.Identifiant <= "5"
    'Retournera les 5 derniers cours
End Function

End Class
```

/ ! \ Ne pas oublier d'inclure le namespace System.Linq.Expressions !

On met le nom de la base que l'on souhaite intercepter dans l'attribut QueryInterceptor.

Et pour finir, on retourne ce que l'on veut que notre application affiche lors du requêtage d'une telle table. Dans cet exemple simple, on ne retournera que les 5 derniers cours.

6.2 L'attribut ChangeInterceptor

L'attribut ChangeInterceptor permet de lever une exception lors d'une opération indésirable. Nous pouvons par exemple l'utiliser pour contrôler l'inscription d'un stagiaire :

```
//C#
[ChangeInterceptor("Stagiaire")]
public void ModificationPseudo(Stagiaire st,
                                        UpdateOperations Nom)
{
    if (nomStagiaire == UpdateOperations.Add || nomStagiaire ==
                                                UpdateOperations.Change)
    {
        ObjectQuery<Stagiaire> query =
                  (ObjectQuery<Stagiaire>)
                  from q in this.CurrentDataSource.Stagiaire
                  where q.Nom == st.Nom
                  select q;
        if (query.Count<Stagiaire>() != 0)
            throw new DataServiceException("Stagiaire déjà présent dans
                                                        la base");
    }
}
```

```
'VB.Net
<ChangeInterceptor("Stagiaire")> _
Public Sub ModificationPseudo(ByVal st As Stagiaire, ByVal Nom As
                                                        UpdateOperations)
    If Nom = UpdateOperations.Add OrElse Nom = UpdateOperations.Change
    Then
        Dim query As ObjectQuery(Of Stagiaire) = DirectCast(From q In
                  Me.CurrentDataSource.Stagiaire _
                  Where q.Nom = st.Nom _
                  Select q, ObjectQuery(Of Stagiaire))

        If query.Count(Of Stagiaire)() <> 0 Then
            Throw New DataServiceException("Stagiaire déjà présent dans
                                                        la base")
        End If
    End If
End Sub
```

Dans ce cas, si la requête trouve un nom identique entré dans la base, elle empêche la mise à jour et retourne l'exception. A vous de penser à la traiter ensuite dans votre application en fonction de vos besoin.

7 Conclusion

Dans ce cours, nous avons pu voir les différentes manières de créer un service :

- Avec des données persistantes au travers du Framework Entity.
- Avec des données non persistantes grâce à l'interface *System.Linq.IQueryable*.

Nous avons aussi vu comment apporter des modifications de ces données en implémentant l'interface *IUpdatable*. Nous nous sommes intéressés aux autorisations et aux opérations ainsi qu'aux Interceptors pour pouvoir mieux maitriser les flux de données.

Dans les prochains chapitres, nous approfondirons la consommation de ces services de données ADO .NET.