Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

ASP .NET Dynamic Data : développement avancé

Version 1.0

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

Sommaire

1 Introduction

1.1 Présentation

Dans le cours précédent intitulé « ASP .NET Dynamic Data : premiers pas », nous avons vu comment créer une simple application avec ASP.NET Dynamic Data. Dans ce cours, nous allons poursuivre le développement de cette application, afin d'aborder des notions plus avancées, permettant de la personnaliser.

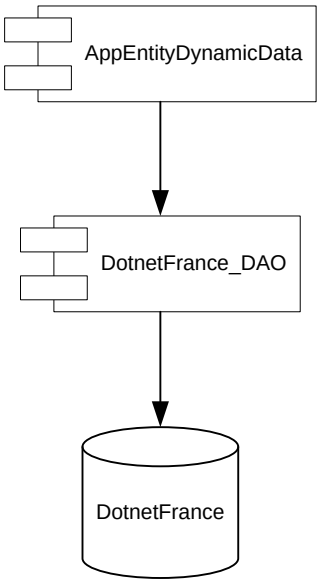
Les concepts de personnalisation présentés ci-dessous, sont tous applicables à une application ASP.NET Dynamic Data qui se base sur un modèle de données créé avec le Framework Entity. Nous attirons votre attention sur le fait que si vous utiliser LINQ To SQL comme modèle de données, alors certaines de ces personnalisations ne seront pas applicables tel qu'elles sont présentées dans ce document, mais de toutes autres manières non abordées dans ce cours.

1.2 Pré-requis

Pour comprendre le contenu de ce cours, nous vous recommandons d'avoir lu les cours d'introduction et « premiers pas » sur ASP .NET Dynamic Data, publiés sur Dotnet-France.

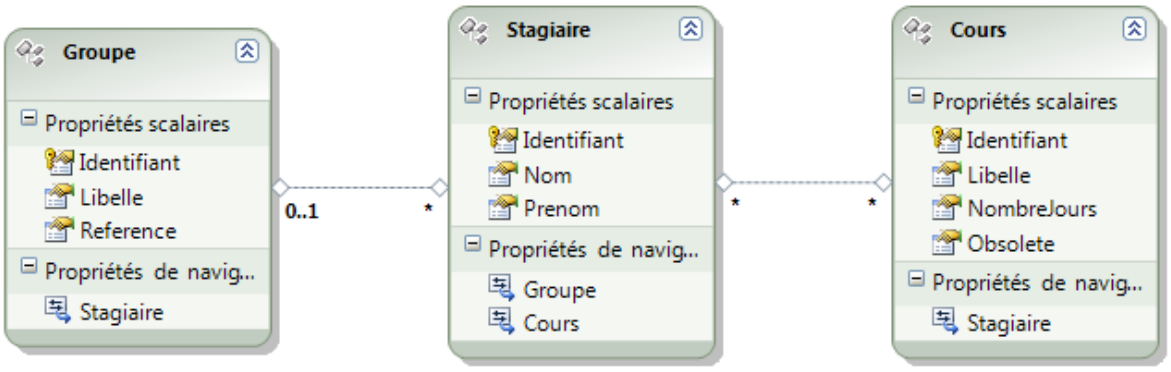
1.3 Quelques rappels sur l'application

Voici l'architecture de l'application sur laquelle nous allons travailler dans ce cours :



Le projet *AppentityDynamicData* est notre application ASP .NET Dynamic Data, que nous allons personnaliser dans ce cours. Le projet *DotnetFrance_DAO* est notre composant d'accès aux données, utilisant le Framework Entity pour consulter et gérer les données de la base de données SQL Server nommée *DotnetFrance*.

Aussi, le modèle de données de l'application est le suivant :


Groupe
Propriétés scalaires
Identifiant
Libelle
Reference
Propriétés de navig...
Stagiaire
0..1
*
Stagiaire
Propriétés scalaires
Identifiant
Nom
Prenom
Propriétés de navig...
Groupe
Cours
*
*
Cours
Propriétés scalaires
Identifiant
Libelle
NombreJours
Obsolete
Propriétés de navig...
Stagiaire

2 Limiter l'accès aux entités du modèle de données

2.1 Présentation

Dans ce chapitre, nous allons modifier le comportement de l'application, afin :

- Définir les entités que l'utilisateur peut consulter.
- Définir les champs à afficher et à masquer.
- De définir les opérations CRUD qu'il est possible de réaliser sur les entités.

2.2 Interdire l'accès aux données d'une entité

Par défaut, lorsque nous lançons la page d'accueil, l'écran suivant apparaît :

Partons du principe que nous ne voulons pas afficher la table (l'entité *Cours*, et ne pouvoir accéder à aucune information sur les cours au travers d'une autre entité, par exemple au travers des stagiaires.

La table *Cours* est affichée car dans le fichier *Global.asax*, l'instruction suivante :

```
// C#

model.RegisterContext(typeof(DotnetFranceEntities), new
ContextConfiguration() { ScaffoldAllTables = true });
```

```
' VB .NET

model.RegisterContext(GetType(DotnetFranceEntities), New
ContextConfiguration() With {.ScaffoldAllTables = True})
```

… permet d'accéder aux données de toutes les tables.

Alors pour interdire la consultation des données concernant les cours des stagiaires, dans le composant *DotnetFrance_DAO*, nous allons ajouter une classe partielle étendant l'entité *Cours*. Sur la définition de la classe, nous allons utiliser la métadonnée *System.ComponentModel.DataAnnotations.ScaffoldTable*, en passant la valeur *False* en paramètre. Voici le contenu du fichier Cours.cs / Cours.vb :

```
// C#

namespace DotnetFrance_DAO
{
    [System.ComponentModel.DataAnnotations.ScaffoldTable(false)]
    public partial class Cours
    {
    }
}
```

```
' VB .NET

<System.ComponentModel.DataAnnotations.ScaffoldTable(False)> _
Partial Public Class Cours

End Class
```

Pour utiliser la méta-donnée *System.ComponentModel.DataAnnotations.ScaffoldTable*, il est nécessaire de référencer le composant *System.ComponentModel.DataAnnotations.dll* dans notre projet.

A noter qu'il est possible d'effectuer l'inverse. Dans le fichier global.asax, il est possible d'écrire l'instruction :

```
// C#

model.RegisterContext(typeof(DotnetFranceEntities), new
ContextConfiguration() { ScaffoldAllTables = false });
```

```
' VB .NET

model.RegisterContext(GetType(DotnetFranceEntities), New
ContextConfiguration() With {.ScaffoldAllTables = false })
```

Puis, sur pour chaque entité du modèle pour lesquelles ont souhaite autoriser la consultation des données, on effectue la même opération que ci-dessus, en fournissant le paramètre *True* à la méta-données *ScaffoldTable*.

2.3 Masquer des champs

Par défaut, lors de l'affichage de listes d'entités ou d'un formulaire de détail d'une entité (pour simple consultation ou modification), toutes les propriétés des entités sont affichées. C'est par exemple le cas de la propriété *Identifiant* des stagiaires, qui est affichée dans la liste des stagiaires et dans le formulaire de détail d'un stagiaire. Etant donné qu'il s'agit d'une donnée purement technique dont le seul but est d'identifier un stagiaire, nous allons la masquer.

Pour ce faire, dans le même esprit que l'affichage / masquage d'une table, nous devons appliquer à la propriété *Identifiant* de l'entité *Stagiaire* la métadonnée *System.ComponentModel.DataAnnotations.ScaffoldColumn*. La difficulté que nous allons rencontrer

est que cette propriété a été généré dans le modèle d'entités *dotnetFrance.edmx*. Voici son implémentation :

```
// C#

private int _Identifiant;

[global::System.Data.Objects.DataClasses.EdmScalarPropertyAttribute(Entit
yKeyProperty=true, IsNullable=false)]
[global::System.Runtime.Serialization.DataMemberAttribute()]
public int Identifiant
{
    get
    {
        return this._Identifiant;
    }
    set
    {
        this.OnIdentifiantChanging(value);
        this.ReportPropertyChanging("Identifiant");
        this._Identifiant = global::System.Data.Objects.DataClasses
                .StructuralObject.SetValidValue(value);
        this.ReportPropertyChanged("Identifiant");
        this.OnIdentifiantChanged();
    }
}
```

```
' VB .NET

Private _Identifiant As Integer

<Global.System.Data.Objects.DataClasses.EdmScalarPropertyAttribute(Entity
KeyProperty:=True, IsNullable:=False), _
 Global.System.Runtime.Serialization.DataMemberAttribute()> _
Public Property Identifiant() As Integer
    Get
        Return Me._Identifiant
    End Get
    Set(ByVal value As Integer)
        Me.OnIdentifiantChanging(value)
        Me.ReportPropertyChanging("Identifiant")
        Me._Identifiant = Global.System.Data.Objects.DataClasses
                .StructuralObject.SetValidValue(value)
        Me.ReportPropertyChanged("Identifiant")
        Me.OnIdentifiantChanged()
    End Set
End Property
```

Si nous modifions ces blocs de code, alors ces modifications seront effacées lors de la prochaine mise à jour du modèle de données. Le moyen proposé par le Framework .NET le suivant :

- Créer une nouvelle classe appelée classe de méta-donnée que nous appellerons *Stagiaire_Metadata*.
- Dans cette classe, on spécifie au travers de la méta-donnée *System.ComponentModel.DataAnnotations.ScaffoldColumn*, que la propriété *Identifiant* ne doit pas être affichée, en passant en paramètre la valeur *False* :

```
// C#

public class Stagiaire_Metadata
{
    [System.ComponentModel.DataAnnotations.ScaffoldColumn(false)]
    public int Identifiant;
}
```

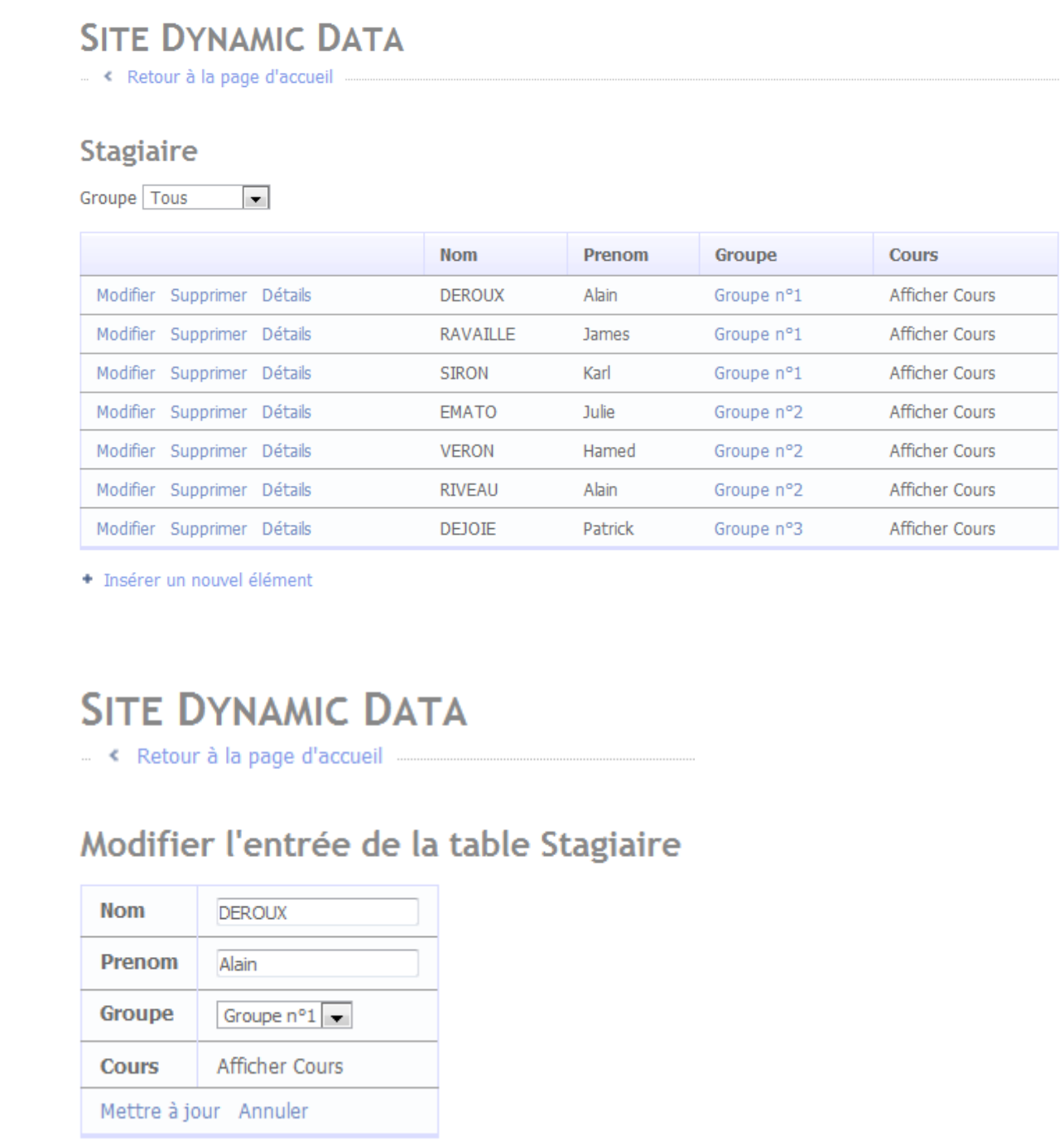
```
' VB .NET

Public Class Stagiaire_Metadata
    <System.ComponentModel.DataAnnotations.ScaffoldColumn(False)> _
    Public Identifiant As Integer
End Class
```

- On associe la classe de méta-donnée *Stagiaire_Metadata* à la l'entité *Stagiaire*, en appliquant la méta-donnée *System.ComponentModel.DataAnnotations.MetadataType* à cette dernière :

```
// C#

[System.ComponentModel.DataAnnotations.MetadataType(typeof(Stagiaire_Metadata))]
public partial class Stagiaire
{
}
```

```
' VB .NET

<System.ComponentModel.DataAnnotations.MetadataType(GetType(Stagiaire_Metadata))> _
Partial Public Class Stagiaire

End Class
```

Une fois les projets compilés et l'application exécutée, on obtient les résultats suivant :



CQFD : on peut observer que observer que l'identifiant n'apparaît pas.

3 La personnalisation des pages et des contrôles utilisateurs

3.1 Présentation

Afin d'adapter l'application à différents besoins, il est possible de la personnaliser. Nous allons présenter les personnalisations suivantes :

- Modifications du modèle de page.

3.2 Sélection des entités à afficher dans la page d'accueil

La page d'accueil de notre application ASP .NET Dynamic Data est la page Default.aspx. Dans son état actuel, cette page affiche la liste de toutes les entités du méta-modèle, à savoir *Groupe*, *Stagiaire* et *Cours*. Nous ne souhaitons afficher que l'entité Groupe.

Positionnons-nous alors dans la méthode *RegisterRoutes* du fichier global.asax.

3.3 Personnalisation des modèles de page

Les modèles de pages sont des pages ASP .NET utilisées lors de l'exécution de l'application, permettant de naviguer, consulter, ajouter, modifier … les données des entités. Elles sont contenues dans le répertoire *DynamicData\PageTemplates* de l'application. Il est possible de les mettre à jour dans le but de les adapter à notre besoin et de les personnaliser. Ces modèles de page sont les suivantes :

- *List.aspx* : cette page permet d'afficher l'ensemble des données d'une entité du méta-modèle dans une grille de données (contrôle *GridView*), avec la possibilité de filtrer et trier les données pour chaque colonne constituant une clé étrangère ou booléenne.
- *ListDetails.aspx* : cette page propose des fonctionnalités similaires à la page *List.aspx*. En supplément, elle propose et un contrôle de type *DetailsView* pour afficher les données de la ligne sélectionnée, ainsi que pour l'ajout d'un nouvel enregistrement.
- *Details.aspx* : cette page permet de consulter le détail d'une ligne.
- *Edit*.aspx : cette page permet de modifier le contenu d'une ligne.
- *Insert*.aspx : cette page permet de créer une nouvelle instance d'une entité et de la persister dans la base de données.

3.3.1 Modification de l'ensemble des pages

Toutes ces pages sont des pages de contenu, qui fournissent du contenu à la Master Page nommé *Site.master* et située à la racine de l'application, dans laquelle elles s'exécutent. En modifiant cette Master Page, vous modifiez l'affichage de toutes ces pages.

Pour illustrer ce propos, nous allons apporter à la page maître les modifications suivantes :

- Ajouter le logo de Dotnet-France en entête de page.
- Modifier le nom de l'application en l'appelant Dotnet-France Student Manager.

Pour appliquer ces modifications, positionnons-nous dans le code XHTML de la page maître, et apportons les modifications suivantes :

```
// XHTML

Bloc de code initial :
<h1><span class="allcaps">Site Dynamic Data</span></h1>


Bloc de code modifié :
<h1><span class="allcaps" id="TitrePage">Dotnet-France Student
Manager</span></h1>
```

Après avoir copié une image dans le répertoire DynamicData/Content/Images, ajoutons dans la feuille de style CSS, la règle CSS suivante :

```
// CSS

#TitrePage
{
    background-
image:url("DynamicData/Content/Images/LogoDotnetFrance.jpg");
    background-repeat:no-repeat;
    padding-left:150px;
}
```

Nous obtenons ainsi l'entête suivante pour chacune de nos pages :

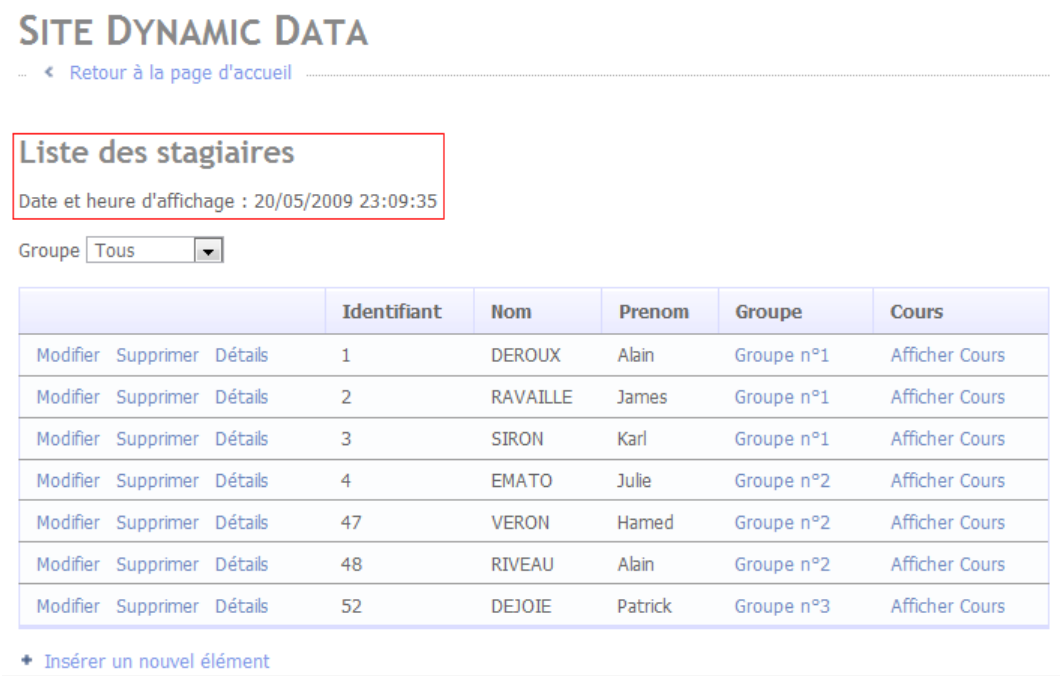
3.3.2 Création de nouveaux modèles de page

Les modèles de pages existant sont utilisées pour toutes les entités, sans exploiter les éventuelles spécificités du modèle de données utilisé par l'application. Pour ce faire, une autre alternative que la modification de ces modèles de page s'offre à vous : créer vos propres modèles de page.

Le répertoire DynamicData\CustomPages de votre application ASP .NET Dynamic Data doit contenir les modèles de page que vous créez. Voici les actions que nous allons réaliser pour créer un nouveau modèle de page pour l'entité Stagiaire :

- Commençons alors par créer un répertoire nommé *Stagiaire* (nom de l'entité pour laquelle nous voulons créer un modèle de page) dans ce répertoire.
- Copions le fichier List.aspx contenu dans le répertoire des modèles de page par défaut (DynamicData\PageTemplates), dans ce répertoire. Renommons la classe code-behind (attention, elle est partielle et est aussi référencée dans la directive @Page de la page .Aspx) de la page ASP .NET. Le nom du fichier doit l'un des noms suivants : List | Details | Edit | Insert | ListDetails

- Personnalisons ce nouveau fichier, en effectuant les modifications suivantes :
  - Le titre de la page, au lieu du nom des entités affichées, doit être « Liste des stagiaires ».
  - Sous ce titre, la date et l'heure d'affichage doit être inscrite.



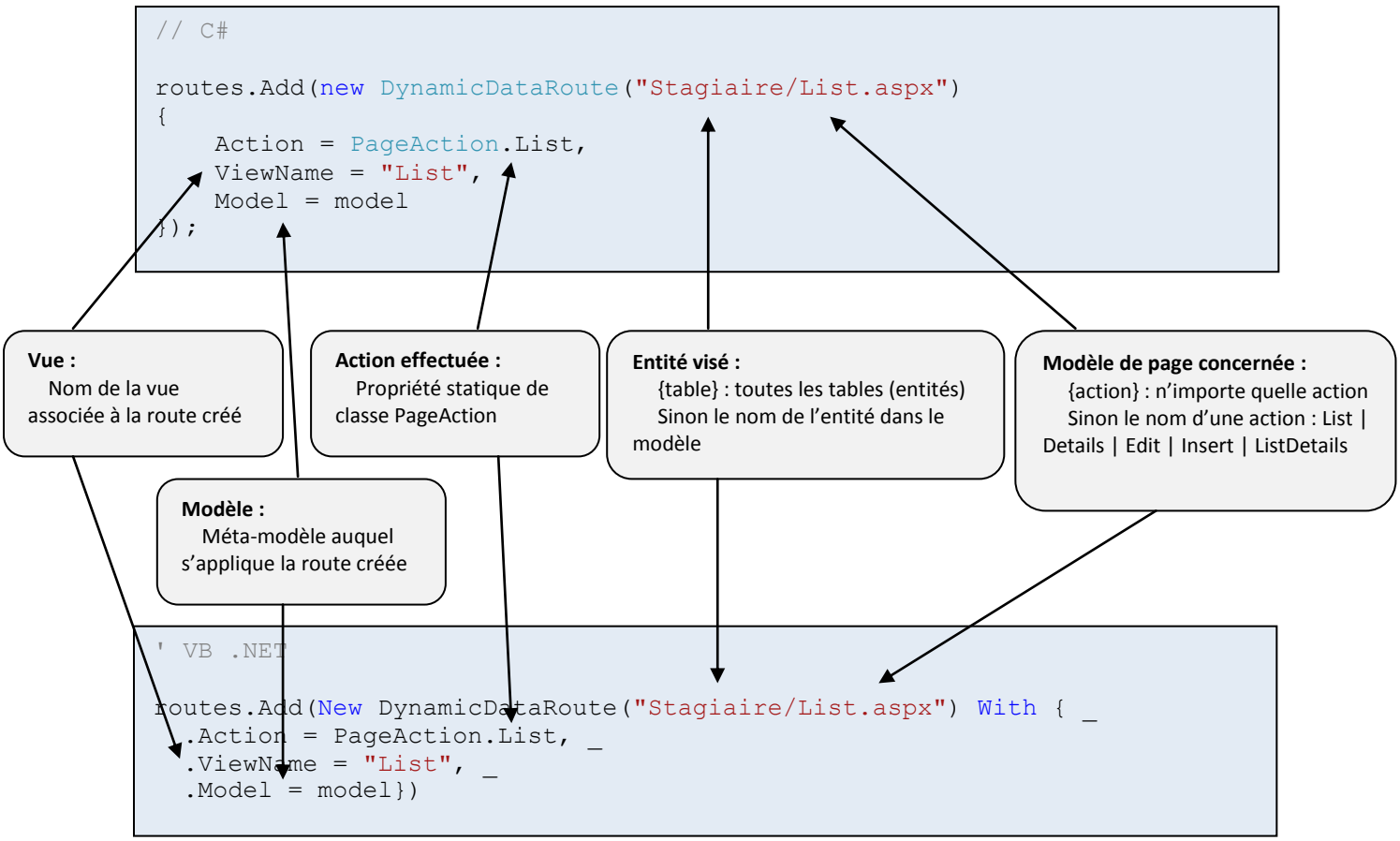
Pour que ce modèle de page soit pris en compte dans l'application, il est nécessaire de personnaliser le routage.

3.3.3 Personnalisation du routage

ASP .NET Dynamic Data utilise le routage d'URL pour réaliser le mapping entre les URLs reçues et les pages à fournir. Le mécanisme de génération de modèles automatique (activé grâce à la valorisation de la propriété *ScaffoldAllTables* de la classe *ContextConfiguration* à *True*, dans le fichier *global.asax*) des tables permet de déduire la page et la table à renvoyer en fonction de l'URL reçue. L'utilisation du mécanisme de routage présente l'avantage suivant : l'URL demandée n'a pas à correspondre au chemin d'accès physique dans l'application.

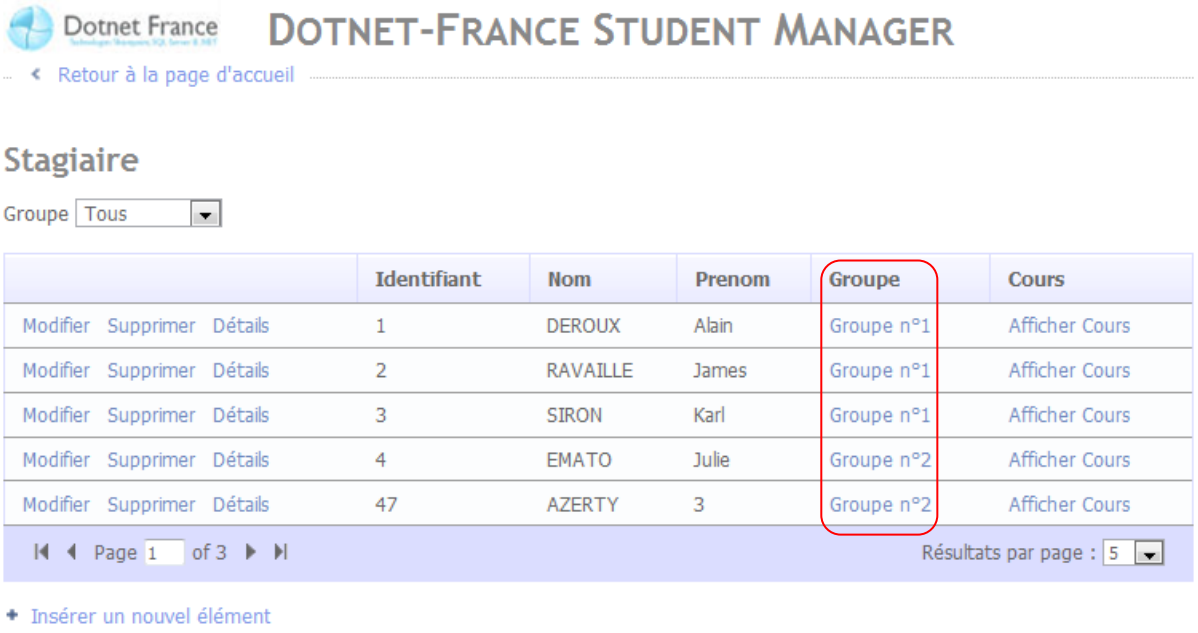
Par défaut, ASP .NET Dynamic Data utilise un modèle de page différent pour chaque action (affichage d'une liste d'entités, affichage d'une entité, ajout d'une entité, modification d'une entité) pour l'ensemble des entités. Nous détaillerons ces modèles de page lorsque nous étudierons leur personnalisation.

Ce routage peut être personnalisé, afin de permettre l'utilisation de modèles de page personnalisés. Voici un exemple de création d'une route, qui permet de rediriger l'utilisateur vers la page DynamicData/PageTemplates/Stagiaire/List.aspx, lors de l'affichage d'une liste de stagiaires :



3.4 Personnalisation des propriétés de navigation

Pour naviguer entre les entités du méta-modèle, ASP .NET Dynamic Data utilise des propriétés de navigation. Par défaut, pour chaque stagiaire, une propriété *Groupe* permet d'accéder aux informations du groupe auquel est rattaché le stagiaire. Cette dernière propriété est implémentée sous forme d'un lien hypertext dont le libellé est le libellé du groupe :

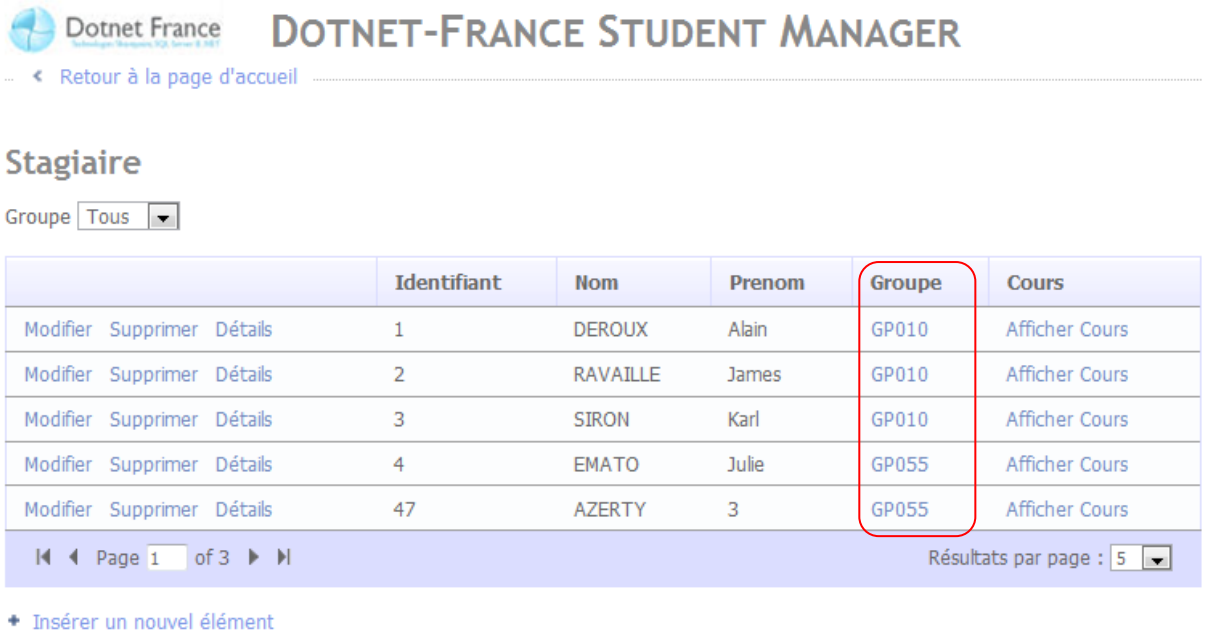


Il est possible de modifier ce libellé, en choisissant une autre propriété de l'entité *Groupe*. Pour ce faire, il nous suffit d'appliquer la métadonnée *System.ComponentModel.DataAnnotations.DisplayColumn* à l'entité *Groupe*. Dans le projet *DotnetFrance_DAO*, créons alors une classe partielle étendant la classe Groupe. Appliquons à cette classe cet attribut, en précisant le nom du champ à afficher (en l'occurrence la référence du groupe), comme indiqué ci-dessous :

```
// C#

namespace DotnetFrance_DAO
{
    [System.ComponentModel.DataAnnotations.DisplayColumn("Reference")]
    public partial class Groupe
    {
    }
}
```

```
' VB

<System.ComponentModel.DataAnnotations.DisplayColumn("Reference")> _
Public Class Groupe

End Class
```

On obtient ainsi le résultat suivant :



3.5 Personnalisation des contraintes sur les données

Au travers du méta-modèle, il est possible de récupérer les contraintes de données positionnées sur le modèle de données défini. Il est possible de personnaliser ces contraintes, de

manière à soit les renforcer, soit les alléger. Par exemple, nous allons lors de la création d'un cours, interdire de saisir un nombre de jours nul ou négatif pour spécifier la durée du cours.

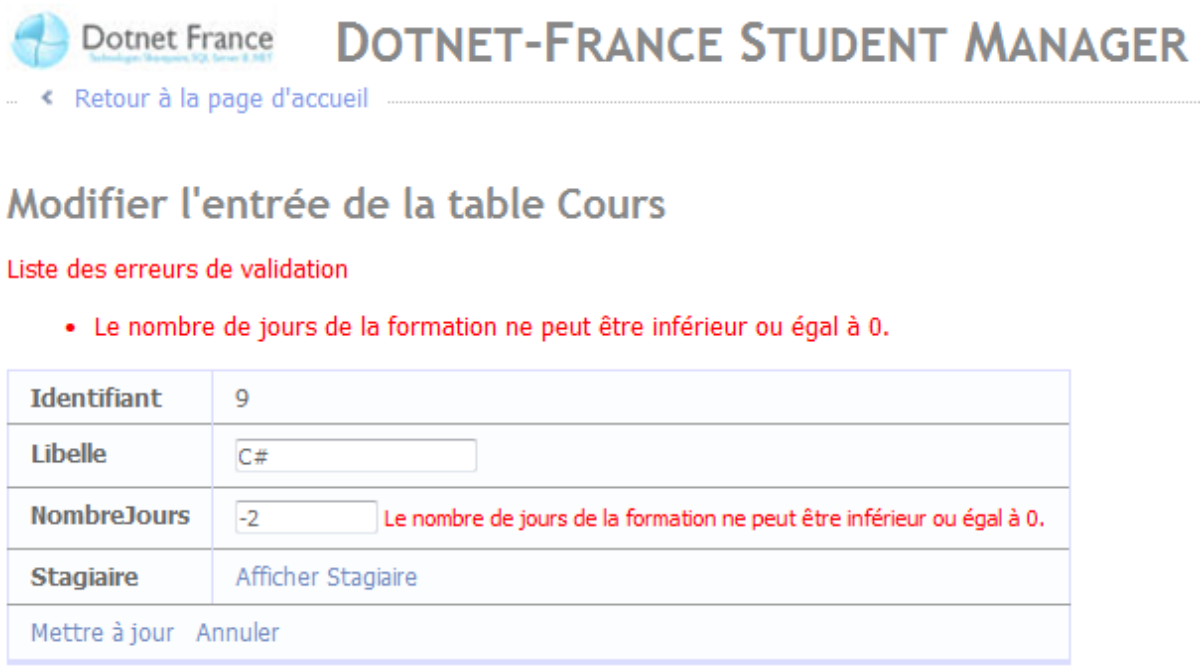
Pour ce faire, nous allons créer une classe partielle étendant la classe *Cours* du projet *DotnfetFrance_DAO*, dans laquelle nous allons implémenter la méthode partielle *OnNombreJoursChanging*, afin d'exécuter du code lors de la modification de la durée d'un cours :

```
// C#

namespace DotnetFrance_DAO
{
    public partial class Cours
    {
        partial void OnNombreJoursChanging(int value)
        {
            if ((this.EntityState != EntityState.Detached) && (value<=0))
            {
                throw new Exception("Le nombre de jours du cours ne peut
être inférieur ou égal à 0.");
            }
        }
    }
}
```

```
' VB
Partial Public Class Cours

    Private Sub OnNombreJoursChanging(ByVal value As Integer)
        If Me.EntityState <> EntityState.Detached AndAlso value <= 0 Then
            Throw New Exception("Le nombre de jours du cours ne peut être
inférieur ou égal à 0.")
        End If
    End Sub

End Class
```

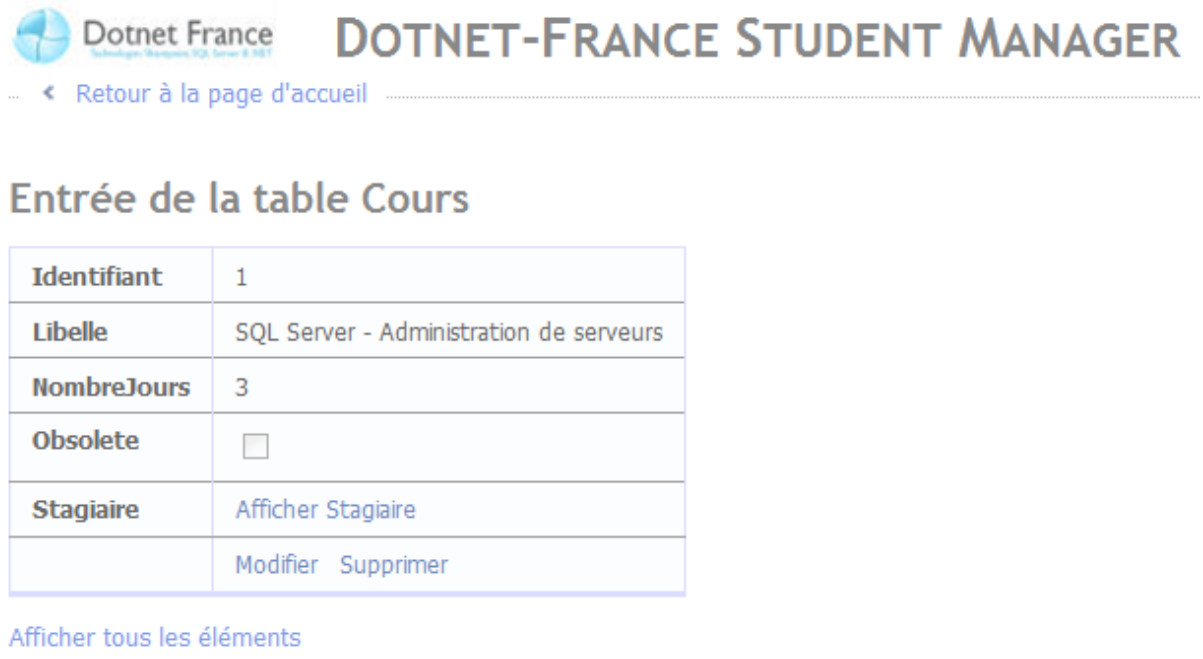
Dans le test spécifié ci-dessus, on test l'état du cours pour lequel on souhaite modifier le nombre de jours, afin d'éviter que l'exception soit levée lors de la demande d'ajout d'un cours.

Ainsi, par exemple, si on modifie la durée d'un cours existant, on obtient le résultat suivant :



3.6 Personnalisation de l'affichage des données dans les champs

Un cours peut être d'actualité ou obsolète. Cette information est gérée par le champ *Obsolete* de l'entité *Cours*, de type booléen. Par défaut, lors de la modification d'un cours, ce champ booléen est affiché au travers d'un contrôle CheckBox :



Nous allons donc personnaliser ce champ en affichant deux boutons radios dont le libelle est oui/non.

Le répertoire Dynamicdata\FieldTemplates de notre application ASP .NET Dynamic Data contient deux ensembles de contrôles utilisateurs spécifiques, dont la classe de base est *System.Web.DynamicData.FieldTemplateUserControl* :

- Un premier ensemble de contrôles utilisé pour l'affichage des informations sur les entités dans les grilles de données. Par exemple, le contrôle utilisateur nommé *Boolean.ascx* permet d'afficher une case à cocher pour une propriété de type booléen d'une entité dans une grille.
- Un second ensemble de contrôles utilisé pour l'affichage du détail pour modifier une entité donnée. Ils portent le même nom que les contrôles du premier ensemble suffixé par la chaîne de caractères « _Edit ». Par exemple le contrôle utilisateur nommé *Boolean_Edit.ascx* permet d'afficher dans une vue détail pour modifier ou ajouter une propriété de type booléen.

Pour répondre au besoin décrit ci-dessus, nous allons modifier le code du contrôle utilisateur *Boolean_Edit.ascx*, de manière à remplacer le contrôle de type Checkbox par deux contrôles de type bouton radio. Voici le code « natif » de ce contrôle :

```
// XHTML

<%@ Control Language="C#" CodeBehind="Boolean_Edit.ascx.cs"
Inherits="AppDynamicData.Boolean_EditField" %>

<asp:CheckBox runat="server" ID="CheckBox1" />
```

```
// C#

public partial class Boolean_EditField :
System.Web.DynamicData.FieldTemplateUserControl
{
    protected override void OnDataBinding(EventArgs e)
    {
        base.OnDataBinding(e);

        object val = FieldValue;
        if (val != null)
            CheckBox1.Checked = (bool)val;
    }

    protected override void ExtractValues(IOrderedDictionary dictionary)
    {
        dictionary[Column.Name] = CheckBox1.Checked;
    }

    public override Control DataControl
    {
        get
        {
            return CheckBox1;
        }
    }
}
```

```
' VB

Partial Class Boolean_EditField
    Inherits System.Web.DynamicData.FieldTemplateUserControl

    Public Overrides ReadOnly Property DataControl() As Control
        Get
            Return CheckBox1
        End Get
    End Property

    Protected Overrides Sub OnDataBinding(ByVal e As EventArgs)
        MyBase.OnDataBinding(e)
        Dim val As Object = FieldValue
        If (Not (val) Is Nothing) Then
            CheckBox1.Checked = CType(val, Boolean)
        End If
    End Sub

    Protected Overrides Sub ExtractValues(ByVal dictionary As
IOrderedDictionary)
        dictionary(Column.Name) = CheckBox1.Checked
    End Sub

End Class
```

Et voici le code de contrôle modifié :

```
// XHTML

<%@ Control Language="C#" CodeBehind="Boolean_Edit.ascx.cs"
Inherits="AppDynamicData.Boolean_EditField" %>

<asp:RadioButton GroupName="Boolean_Edit" runat="server"
ID="RbtAffirmatif" Text="Oui" />
 
<asp:RadioButton GroupName="Boolean_Edit" runat="server" ID="RbtNegatif"
Text="Non" />
```

```
// C#

public partial class Boolean_EditField :
System.Web.DynamicData.FieldTemplateUserControl
{
    protected override void OnDataBinding(EventArgs e)
    {
        base.OnDataBinding(e);

        object val = FieldValue;
        bool bval;
        if (val != null)
        {
            bval = Convert.ToBoolean(val);
            RbtAffirmatif.Checked = bval;
            RbtNegatif.Checked = !bval;
        }
    }

    protected override void ExtractValues(IOrderedDictionary dictionary)
    {
        dictionary[Column.Name] = RbtAffirmatif.Checked;
    }

    public override Control DataControl
    {
        get
        {
            return RbtAffirmatif;
        }
    }
}
```

```
' VB

Imports System.Web.DynamicData
Partial Class Boolean_EditField
    Inherits System.Web.DynamicData.FieldTemplateUserControl

    Protected Overrides Sub OnDataBinding(ByVal e As EventArgs)
        MyBase.OnDataBinding(e)

        Dim val As Object = FieldValue
        Dim bval As Boolean
        If val IsNot Nothing Then
            bval = Convert.ToBoolean(val)
            RbtAffirmatif.Checked = bval
            RbtNegatif.Checked = Not bval
        End If
    End Sub

    Protected Overrides Sub ExtractValues(ByVal dictionary As
IOrderedDictionary)
        dictionary(Column.Name) = RbtAffirmatif.Checked
    End Sub

    Public Overrides ReadOnly Property DataControl() As Control
        Get
            Return RbtAffirmatif
        End Get
    End Property
End Class
```

La classe code-behind de ce contrôle permet de fournir des informations sur le choix de l'utilisateur, mais aussi de cocher le radio bouton adéquat, en fonction de la valeur de la propriété *Obsolete* du cours affiché.

On obtient ainsi le résultat suivant :

4 Conclusion

Dans ce cours, nous avons personnalisé une simple application ASP .NET Dynamic Data, de manière à montrer comment l'adapter à nos besoins. Vous avez pu rendre compte des nombreuses possibilités de personnalisation, tant que l'affichage des données, que leur gestion.

Sans aucun doute, ASP .NET Dynamic Data permet de créer des applications de gestion de données personnalisées, avec un faible effort de développement.