

Développement d'une application de gestion de contacts avec ASP.NET MVC (C#)

Dans cette série de tutoriaux, nous allons construire de bout en bout une application complète de gestion de contacts. Cette application vous permettra de stocker les informations – noms, numéro de téléphone et adresses emails – pour une liste de personnes.

Nous allons construire cette application à travers plusieurs étapes. Chacune d'entre elles allant enrichir notre application MVC. Le but de cette construction par étape est de vous permettre de comprendre la raison derrière chaque changement.

Etape 1 – Création de l'application. Dans cette 1ère étape, nous allons créer l'application de gestion de contacts de la manière la plus simple qui soit. Nous allons mettre en place le support d'opérations classiques vers la base de données : création, lecture, mise à jour et suppression d'enregistrements.

Etape 2 – Rendre l'application plus attrayante. Ici, nous allons améliorer l'apparence de l'application en modifiant la page maitre par défaut d'ASP.NET MVC ainsi que le CSS l'accompagnant.

Etape 3 – Ajout de la validation de formulaires. Dans cette 3ème étape, nous allons ajouter une logique simple de validation. Nous allons empêcher les utilisateurs de soumettre un formulaire sans remplir certains champs obligatoires. Nous allons également valider les adresses email et numéros de téléphone.

Etape 4 – Rendre l'application faiblement couplée. Ici, nous allons profiter de plusieurs modèles de développement logiciel (Design Patterns) pour maintenir et modifier plus facilement notre application. Par exemple, nous allons revoir l'application pour utiliser 2 patterns connus sous le nom de « Repository pattern » et « Dependency Injection pattern »

Etape 5 – Créer des tests unitaires. Dans cette 5ème étape, afin de rendre encore plus simple la gestion du code source, nous allons ajouter des tests unitaires. Nous allons utiliser pour cela un « Mock Object Framework » puis fabriquer des tests unitaires pour nos contrôleurs et logiques de validation.

Etape 6 – Utiliser un développement guidé par les tests. Ici, nous allons ajouter de nouvelles fonctionnalités d'abord en écrivant des tests unitaires puis ensuite en écrivant du code pour ces mêmes tests unitaires. Dans cette étape, nous allons ajouter la notion de groupes de contacts.

Etape 7 – Ajouter le support d'Ajax. Dans cette dernière phase, nous allons améliorer la réponse et la performance de notre application en y ajoutant le support d'Ajax.

Etape 1 – Création de l'application

Le projet ASP.NET MVC

Lancez Visual Studio et choisissez « File, New Project ». Lorsque la fenêtre « New Project » apparaît (Figure 1), choisissez le type de projet « Web » puis le modèle « ASP.NET MVC Web Application ». Nommez votre projet ContactManager et cliquez sur OK

Faites bien attention à avoir retenu .NET Framework 3.5 dans la liste déroulante en haut à droite de la fenêtre « New Project ». Sinon, vous ne verrez pas le modèle de projet ASP.NET MVC apparaître.

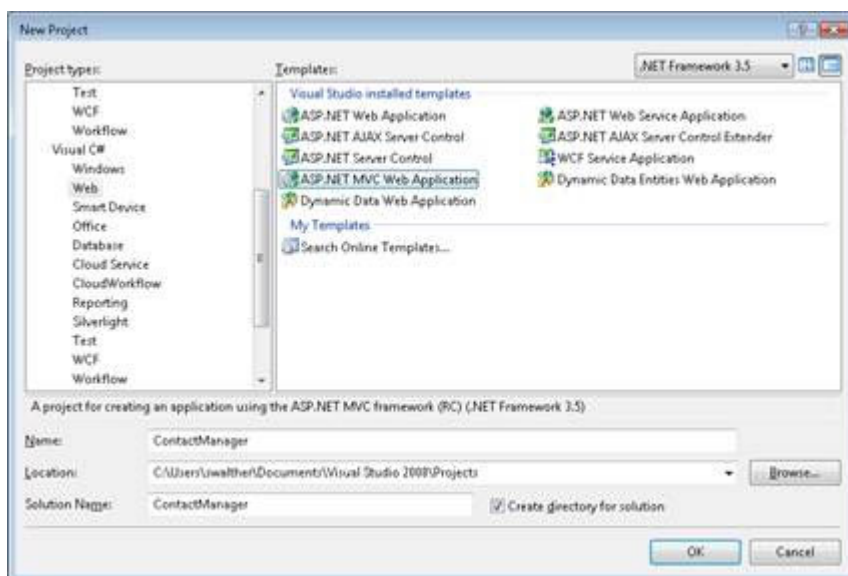


Figure 01: La fenêtre de création du projet

La fenêtre « Create Unit Test Project » apparaît. Vous pouvez utiliser cette fenêtre pour indiquer que vous souhaitez créer et ajouter un projet de tests unitaires à votre solution pendant que vous êtes en train de créer une application ASP.NET MVC. Bien que nous n'utiliserons pas de tests unitaires dans cette étape, choisissez bien l'option « Yes, create a unit test project » car nous le ferons dans une étape suivante. D'ailleurs, ajoutez un projet de tests pendant la phase de création d'une solution ASP.NET MVC est bien plus simple qu'après.

Comme Visual Web Developer ne supporte pas les projets de type tests, vous n'aurez pas la fenêtre d'ajout d'un projet de tests qui vous sera proposé.

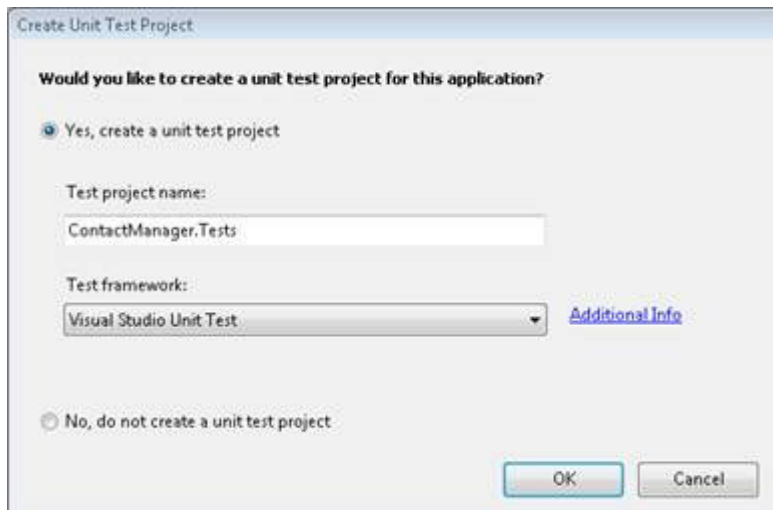


Figure 02: La fenêtre d'ajout d'un projet de tests unitaires

L'application ASP.NET MVC apparaît dans l'explorateur de solution Visual Studio (Figure 3). Si vous ne voyez pas l'explorateur de solution, vous pouvez l'ouvrir en choisissant le menu « View, Solution Explorer ». Vous noterez que la solution contient 2 projets : le projet ASP.NET MVC lui-même et le projet de tests. Le projet MVC s'appelle ContactManager et le projet de tests s'appelle ContactManager.Tests.

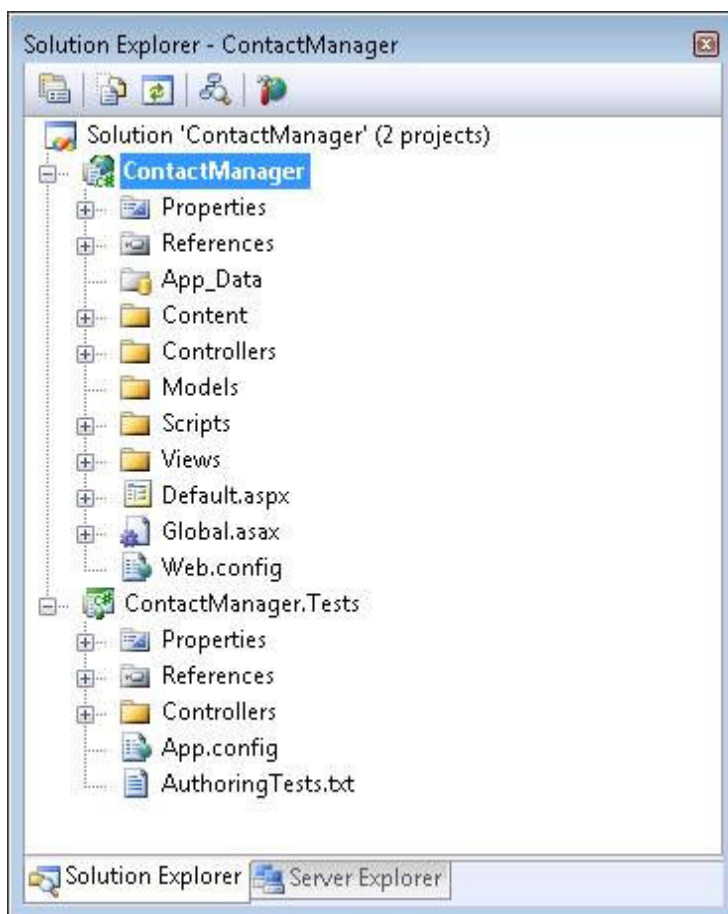


Figure 03: L'explorateur de solution

Suppression des fichiers d'exemples

Le modèle de projet ASP.NET MVC contient des exemples d'implémentation pour des contrôleurs et des vues. Avant de continuer à créer notre application, vous devez supprimer ces fichiers. Vous pouvez les supprimer en vous rendant dans l'explorateur de solution, en cliquant droit sur un fichier ou un répertoire et en retenant l'option « Delete ».

Vous devrez alors supprimer les fichiers suivant depuis le projet ASP.NET MVC :

- \Controllers\HomeController.cs
- \Views\Home\About.aspx
- \Views\Home\Index.aspx

Ainsi que le fichier suivant depuis le projet de tests:

- \Controllers\HomeControllerTest.cs

Création de la base de données

Notre application Contact Manager est une application web orientée base de données. Nous utilisons une base de données pour stocker les informations de nos contacts.

Le framework ASP.NET MVC fonctionne avec l'ensemble des bases de données modernes du marché comme Microsoft SQL Server, Oracle, MySQL et IBM DB2. Dans ce tutorial, nous utiliserons une base Microsoft SQL Server. D'ailleurs, lorsque vous installez Visual Studio, on vous offre le choix d'installer en option Microsoft SQL Server Express qui est une version gratuite de Microsoft SQL Server.

Créez une nouvelle base de données en cliquant-droit sur le répertoire App_Data dans l'explorateur de solution et choisissez l'option Add, New Item. Dans la fenêtre Add New Item, cliquez sur la catégorie Data puis le modèle SQL Server Database (Figure 4). Nommez la nouvelle base de données ContactManagerDB.mdf et cliquez sur OK.

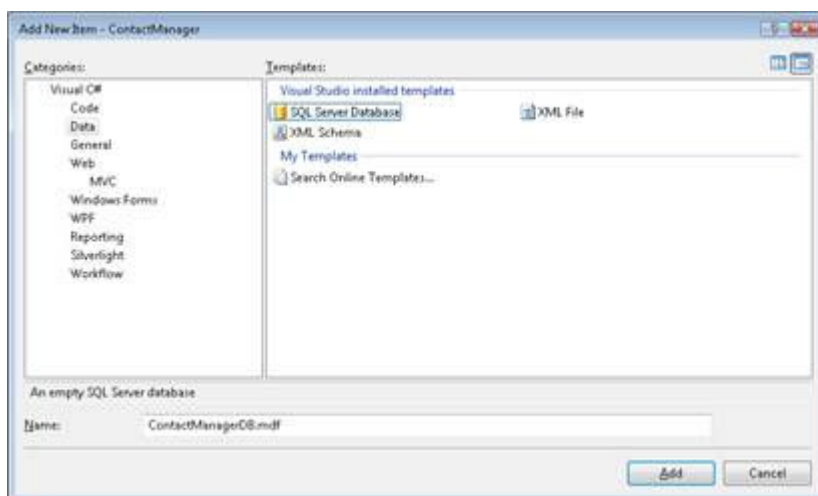


Figure 04: Création d'une nouvelle base de données Microsoft SQL Server Express

Après avoir créé la nouvelle base, cette dernière apparaît dans le répertoire App_Data dans l'explorateur de solution. Double-cliquez sur ContactManagerDB.mdf

La fenêtre explorateur de serveur (Server Explorer) est appelée explorateur de base de données (Database Explorer) sous Microsoft Visual Web Developer.

Vous pouvez utiliser la fenêtre Server Explorer pour créer de nouveaux objets dans la base comme des tables, des vues, des triggers et des procédures stockées. Cliquez-droit sur le répertoire Tables et sélectionnez l'option Add New Table. L'assistant de création de tables apparaît (Figure 5)

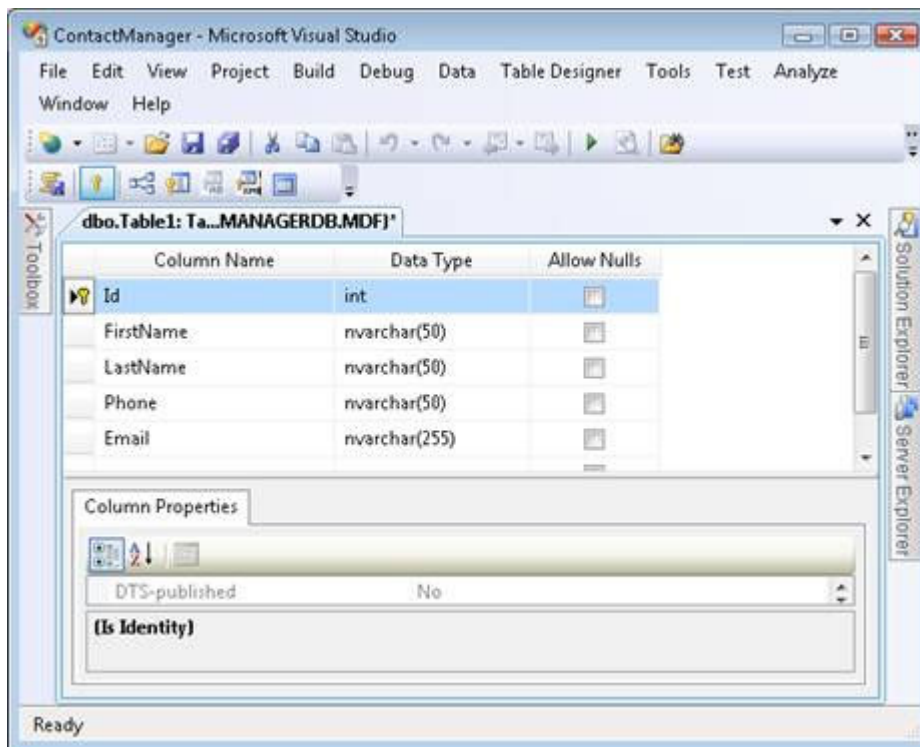


Figure 05: L'assistant de création de tables

Nous devons créer une table contenant les colonnes suivantes :

Nom de colonne	Type de donnée	Autorise les valeurs nulles
Id	int	false
FirstName	nvarchar(50)	false
LastName	nvarchar(50)	false
Phone	nvarchar(50)	false
Email	nvarchar(255)	false

La première colonne, Id, est spéciale. Vous devez marquer la colonne Id comme la colonne d'identité et de clé primaire. Pour cela, vous devez vous rendre dans les propriétés de la colonne (Column Properties au bas de la figure 5) et descendre vers le bas pour positionner une valeur dans Identity Specification. Positionnez la propriété (Is Identity) à Yes

Vous marquerez une colonne comme clé primaire en la sélectionnant et en cliquant sur le bouton avec une icône de clé dans la barre de Visual Studio. Une fois l'opération effectuée, une icône en forme de clé apparaît prêt de la colonne retenue (voir Figure 5).

Après avoir fini de créer la table, cliquez que le bouton de sauvegarde (le bouton avec une icône en forme de lecteur de disquettes) afin de sauvegarder la table. Nommez là Contacts.

Après avoir fini la sauvegarde de la table Contacts, vous devriez être en mesure d'ajouter des enregistrements dans celle-ci. Cliquez-droit sur la table dans la fenêtre Server Explorer et choisissez Show Table Data. Entrez un ou plusieurs contacts dans la grille fournie.

Création du modèle de données

Une application ASP.NET MVC est constituée de 3 briques: des modèles, des vues et des contrôleurs. Nous allons commencer par créer un modèle de classes représentant notre table Contacts créée dans la section précédente.

Dans ce tutorial, nous allons utiliser Microsoft Entity Framework pour générer automatiquement le modèle de classes depuis la base de données.

Le framework ASP.NET MVC n'est lié en aucune façon à l'utilisation de Microsoft Entity Framework. Vous pouvez utiliser ASP.NET MVC avec d'autres technologies d'accès aux données comme NHibernate, LINQ to SQL ou ADO.NET.

Suivez ces étapes pour créer le modèle de classes de données :

1. Cliquez-droit sur le répertoire Models dans l'explorateur de solution et choisissez Add, New Item. La fenêtre Add New Item apparaît (Figure 6)

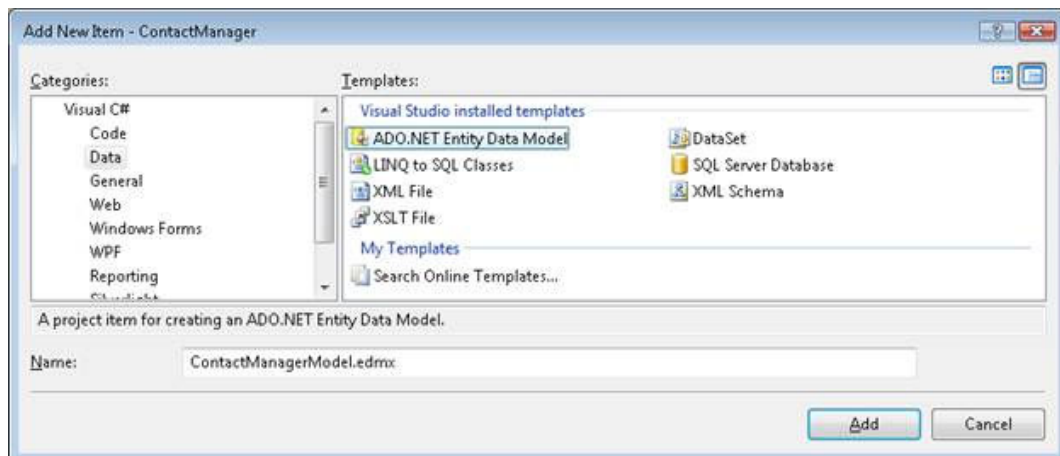


Figure 06: La fenêtre d'ajout d'un nouvel élément

2. Retenez la catégorie Data et le modèle ADO.NET Entity Data Model. Nommez votre modèle de données ContactManagerModel.edmx et cliquez sur le bouton Add. L'assistant Entity Data Model apparaît (Figure 7)

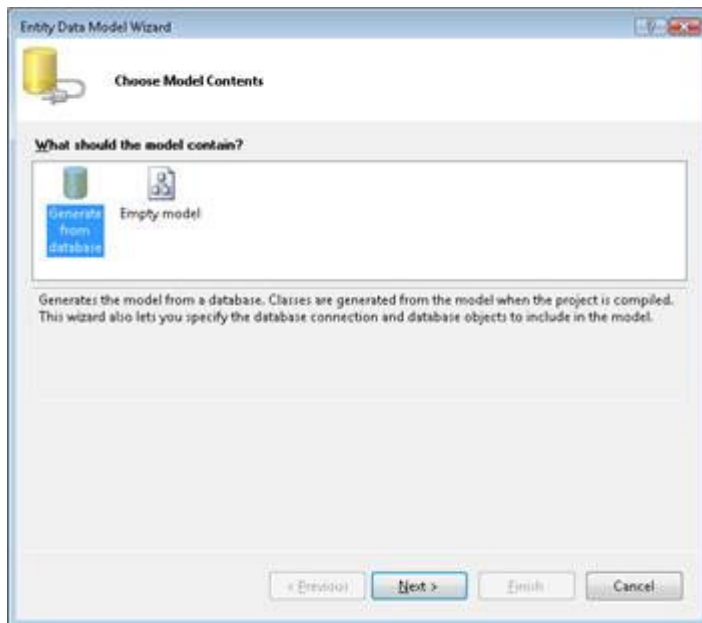


Figure 07: Choisissez le contenu source pour votre modèle

3. A l'étape Choose Model Contents, retenez l'option Generate from database (Figure 7)
4. A l'étape Choose Your Data Connection, choisissez la base ContactManagerDB.mdf et entrez le nom ContactManagerDBEntities pour le paramètre à sauvegarder dans le Web.config (Figure 8)

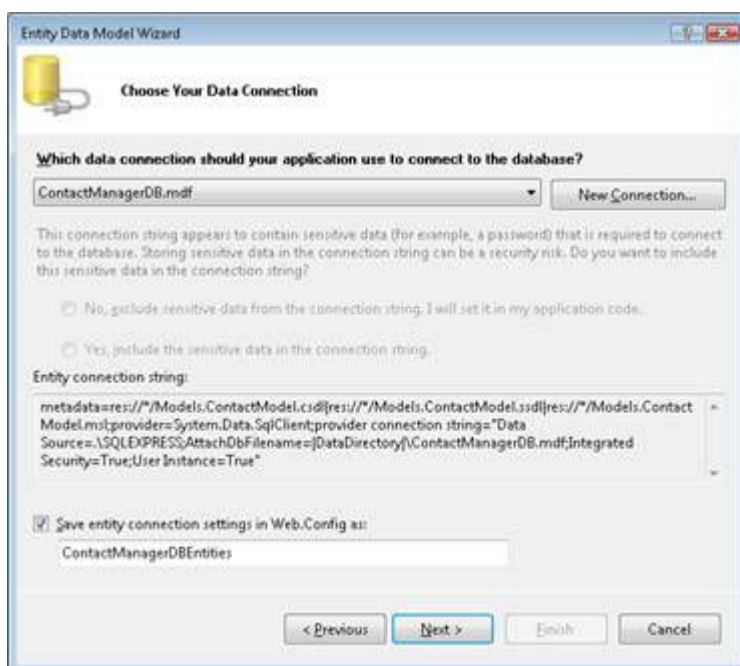


Figure 08: Choisissez le modèle de connexion vers vos données

5. A l'étape Choose Your Database Objects, cochez la case Tables (Figure 9). Le modèle de données inclura toutes les tables présentes dans votre base de données (dans notre cas, il y en a qu'une : Contacts). Entrez Models comme nom pour notre espace de nommage. Cliquez sur le bouton Finish pour terminer l'assistant.

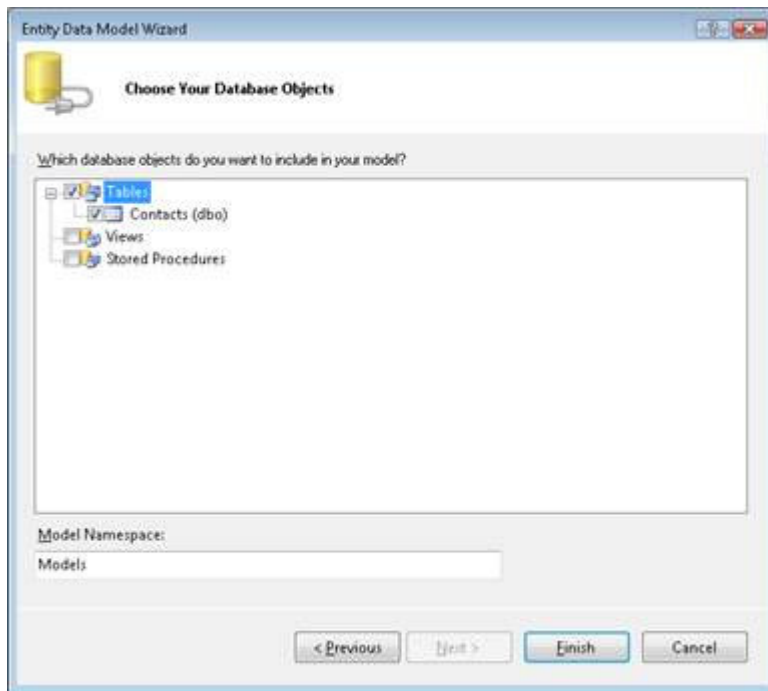


Figure 09: Choisissez les objets à ajouter au modèle

Une fois l'assistant rempli jusqu'au bout, le concepteur Entity Data Model apparaît. Le concepteur affiche une classe correspondant à chaque table présente dans notre base. Vous devriez ainsi voir une classe nommée Contacts.

L'assistant d'Entity Data Model génère des noms de classes basées sur celui des tables. Vous aurez cependant presque toujours besoin de changer le nom des classes générées par l'assistant. Pour cela, cliquez-droit sur la classe Contacts dans le concepteur et choisissez l'option Rename. Changez le nom de la classe de Contacts (au pluriel) vers Contact (au singulier). Après ce changement, la classe devrait ressembler à celle de la Figure 10.

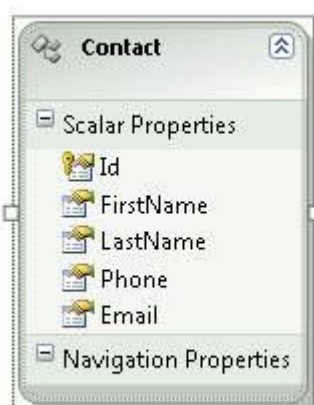


Figure 10: La classe Contact

A ce stage, nous avons créé notre modèle de données. Nous pouvons utiliser la classe Contact pour représenter un enregistrement particulier de notre base de données dans notre application.

Création du contrôleur par défaut: Home

La prochaine étape est de créer notre contrôleur Home. Le « Home controller » sera celui invoqué par défaut par une application ASP.NET MVC.

Créez la classe de ce contrôleur en cliquant-droit sur le répertoire Controllers dans l'explorateur de solution et en choisissant l'option Add, Controller (Figure 11). Notez la case à cocher « ajoutez des méthodes d'actions pour les scénarios de création, mise à jour et détails » (Add action methods for Create, Update, and Details scenarios). Cochez bien cette case avant de cliquer sur le bouton Add.

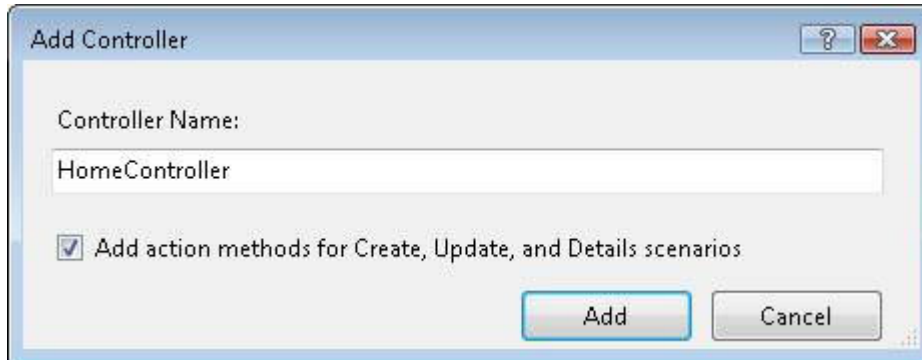


Figure 11: Ajout du contrôleur par défaut (Cliquez pour agrandir l'image)

Lorsque vous créez le contrôleur Home, vous obtenez alors le code présent dans Listing 1.

Listing 1 – Controllers\HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace ContactManager.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

        public ActionResult Index()
        {
            return View();
        }

        //
        // GET: /Home/Details/5

        public ActionResult Details(int id)
        {
            return View();
        }

        //
    }
}
```

```

// GET: /Home/Create

public ActionResult Create()
{
    return View();
}

//
// POST: /Home/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(FormCollection collection)
{
    try
    {
        // TODO: Add insert logic here

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

//
// GET: /Home/Edit/5

public ActionResult Edit(int id)
{
    return View();
}

//
// POST: /Home/Edit/5

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection collection)
{
    try
    {
        // TODO: Add update logic here

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
}
}

```

Lister les contacts

Afin de lister les enregistrements depuis la table Contacts de notre base de données, nous devons créer une action Index() et une vue Index.

Le contrôleur Home contient déjà une action Index(). Nous devons alors modifier cette méthode pour qu'elle ressemble à celle du Listing 2.

Listing 2 – Controllers\HomeController.cs

```
public class HomeController : Controller
{
    private ContactManagerDBEntities _entities = new
    ContactManagerDBEntities();

    //
    // GET: /Home/

    public ActionResult Index()
    {
        return View(_entities.ContactSet.ToList());
    }
    ...
}
```

Notez que la classe du contrôleur du Listing 2 contient un champ privé nommé _entities. Le champ _entities représente les entités de notre modèle de données. Nous utilisons le champ _entities pour communiquer avec la base de données.

La méthode Index() retourne une vue représentant tous les contacts de notre table Contacts. L'expression _entities.ContactSet.ToList() retourne la liste des contacts en tant que liste générique.

Maintenant que nous avons créé le contrôleur gérant l'Index, il nous faut créer la vue associée. Avant de créer la vue Index, compilez votre application via l'option Build, Build Solution. Vous devez toujours compiler votre projet avant d'ajouter une nouvelle vue afin de voir apparaître la liste des modèles de classes dans la fenêtre Add View.

Créez la vue Index en cliquant-droit sur la méthode Index() et en choisissant l'option Add View (Figure 12). La fenêtre d'ajout de vue s'ouvre alors (Figure 13).

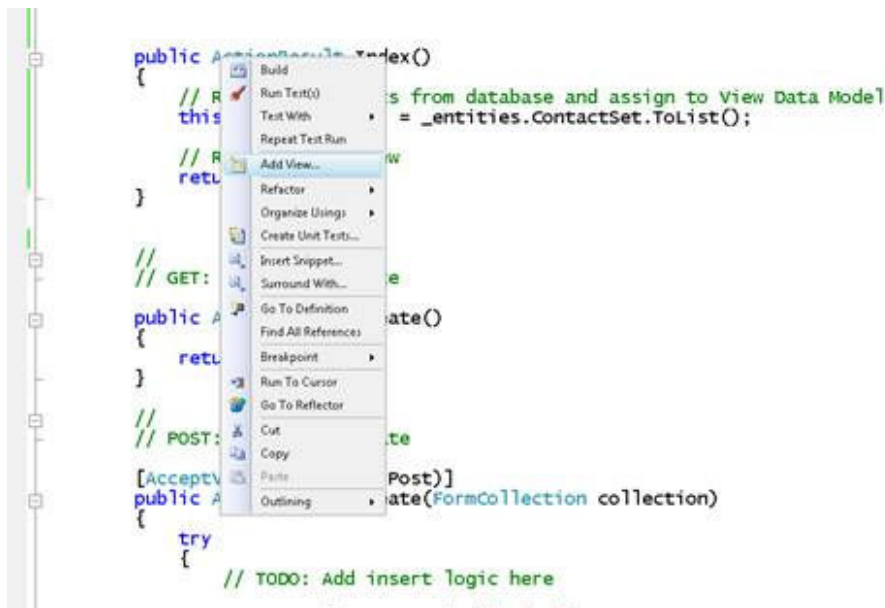


Figure 12: Ajout de la vue Index

Dans la fenêtre Add View, cochez la case intitulée Create a strongly-typed view (créer une vue fortement typée). Sélectionnez la classe de données `ContactManager.Models.Contact` et retenez l'option List pour View Content. En choisissant ces options, cela va générer une vue affichant la liste des enregistrements de la table Contact.

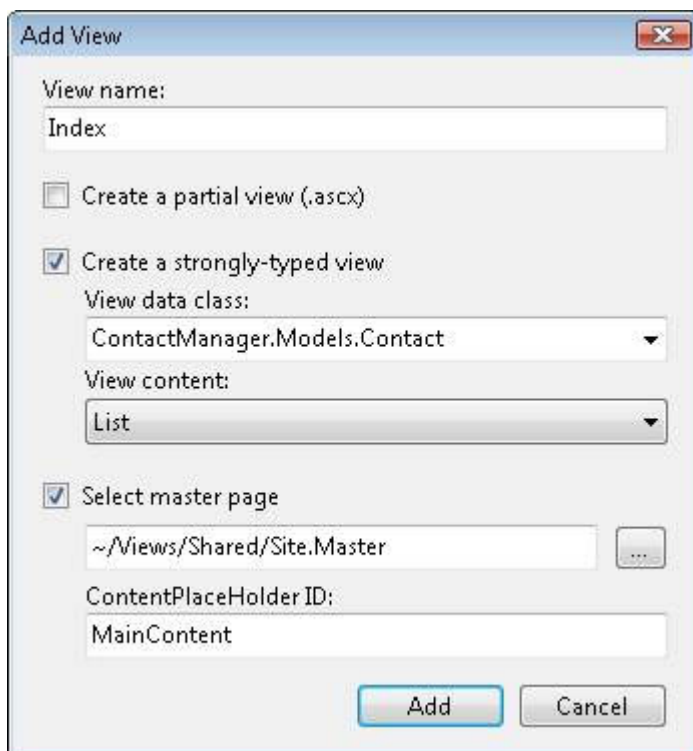


Figure 13: La fenêtre d'ajout d'une vue

Lorsque vous cliquez sur le bouton Add, la vue Index du Listing 3 est générée pour vous. Remarquez la directive `<%@ Page %>` qui apparaît en haut de page. La vue Index hérite de la classe `ViewPage<IEnumerable<ContactManager.Models.Contact>>`. C'est ici que s'effectue notre liaison fortement typée.

Le corps de notre vue Index contient une boucle foreach qui parcourt chacun des contacts représenté par notre modèle de classes. La valeur de chacune des propriétés de la classe Contact est affichée au sein d'une table HTML

Listing 3 – Views\Home\Index.aspx (générée automatiquement)

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<ContactManager.Models.Contact>>"
%>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <table>
        <tr>
            <th></th>
            <th>
                Id
            </th>
            <th>
                FirstName
            </th>
            <th>
                LastName
            </th>
            <th>
                Phone
            </th>
            <th>
                Email
            </th>
        </tr>

        <% foreach (var item in Model) { %>

            <tr>
                <td>
                    <%= Html.ActionLink("Edit", "Edit", new { id=item.Id }) %> |
                    <%= Html.ActionLink("Details", "Details", new { id=item.Id
                %>
            %>

                </td>
                <td>
                    <%= Html.Encode(item.Id) %>
                </td>
                <td>
                    <%= Html.Encode(item.FirstName) %>
                </td>
                <td>
                    <%= Html.Encode(item.LastName) %>
                </td>
                <td>
                    <%= Html.Encode(item.Phone) %>
                </td>
            </tr>
        } %>
```

```

        </td>
        <td>
            <%= Html.Encode(item.Email) %>
        </td>
    </tr>

<% } %>

</table>

<p>
    <%= Html.ActionLink("Create New", "Create") %>
</p>

</asp:Content>

```

Nous devons effectuer une modification à la vue Index. Comme nous ne créons pas de vue Détails, nous pouvons retirer le lien vers celle-ci. Trouver et retirer le code suivant de la vue Index :

```
<%= Html.ActionLink("Details", "Details", new { id=item.Id })%>
```

Après avoir modifié la vue Index, vous pouvez lancer l'application Contact Manager. Pour cela, rendez-vous dans Debug, Start Debugging ou appuyer simplement sur F5. La première fois que vous lancerez l'application, vous verrez apparaître la fenêtre de la Figure 14. Sélectionnez l'option Modify the Web.config file to enable debugging (Modifier le fichier Web.config pour permettre le débogage) et cliquez sur OK.

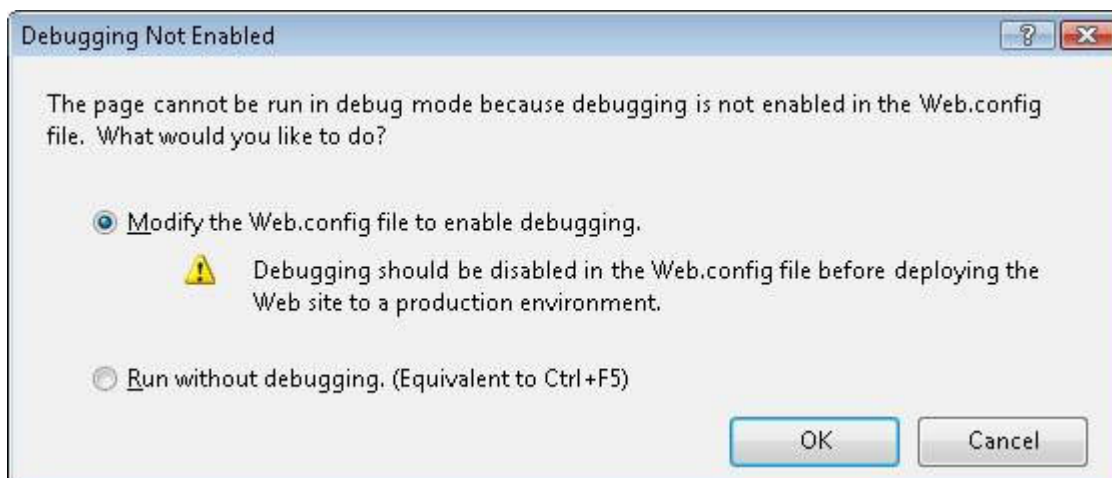


Figure 14: Activation du débogage

La vue Index (/) est retournée par défaut. Cette vue liste bien l'ensemble des données présentes dans la table Contacts (Figure 15).

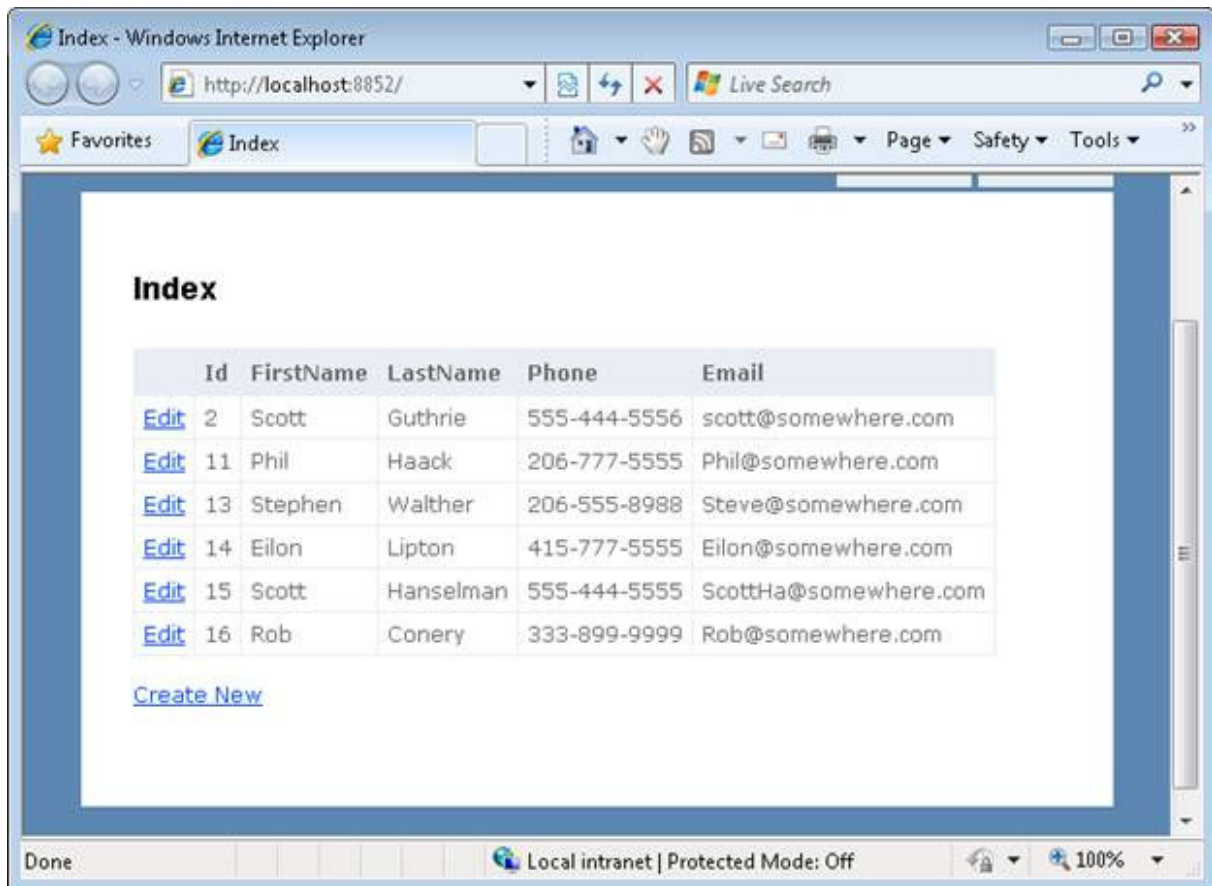


Figure 15: La vue Index en action

Remarquez que la vue Index inclut un lien intitulé Create New (Créer un nouveau contact) au bas de la vue. Dans la section suivante, vous allez apprendre comment créer de nouveaux contacts.

Création de nouveaux contacts

Pour permettre aux utilisateurs de créer de nouveaux contacts, nous devons ajouter 2 actions Create() au contrôleur par défaut (Home). Nous devons en effet créer une première action Create() retournant un formulaire HTML pour nous laisser saisir les informations du nouveau contact à créer. Ensuite, il nous faut une 2ème action Create() qui s'occupera d'insérer ces informations saisies dans la base de données.

La nouvelle méthode Create() que nous devons ajouter au contrôleur Home se trouve dans le Listing 4.

Listing 4 – Controllers\HomeController.cs (avec les méthodes Create)

```
//
// GET: /Home/Create

public ActionResult Create()
{
    return View();
}
```

```
//
// POST: /Home/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)
{
    if (!ModelState.IsValid)
        return View();

    try
    {
        _entities.AddToContactSet(contactToCreate);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

La 1ère méthode Create() peut être appelée avec une requête HTTP GET tant dis que la seconde méthode ne pourra être appelée que via un HTTP POST. En d'autres termes, la seconde méthode Create() ne peut être appelée uniquement lorsque nous allons soumettre le formulaire HTML. La première méthode s'occupe uniquement de retourner une vue contenant le HTML nécessaire à la saisie des informations d'un nouveau contact. C'est donc bien la seconde méthode la plus intéressante : elle s'occupe d'ajouter réellement le contact dans la base.

Vous remarquerez aussi que la 2ème méthode a été modifiée pour accepter une instance de la classe Contact. Grâce à cela, les valeurs du formulaire HTML soumise par le POST sont automatiquement associées à celles de la classe Contact par le framework ASP.NET MVC. Chaque valeur du formulaire HTML sera donc assignée à une des propriétés du Contact passé en paramètre.

Notez également que le paramètre de type Contact est décoré avec l'attribut [Bind]. Cet attribut est utilisé afin d'exclure la propriété Id de la classe Contact du binding automatique. En effet, cette propriété représente une propriété d'identité et nous ne souhaitons pas positionner cette propriété manuellement mais bien laisser la base de données s'en occuper.

Dans le corps de la méthode Create(), Entity Framework est utilisé pour insérer l'instance du nouveau Contact dans la base de données. Le nouveau contact est ajouté dans l'ensemble des contacts existants en mémoire puis la méthode SaveChanges() est appelée pour pousser ces changements vers la base de données sous-jacente.

Vous pouvez générer un formulaire HTML pour créer de nouveaux contacts en cliquant-droit sur l'une des 2 méthodes Create() et en choisissant l'option Add View (Figure 16).

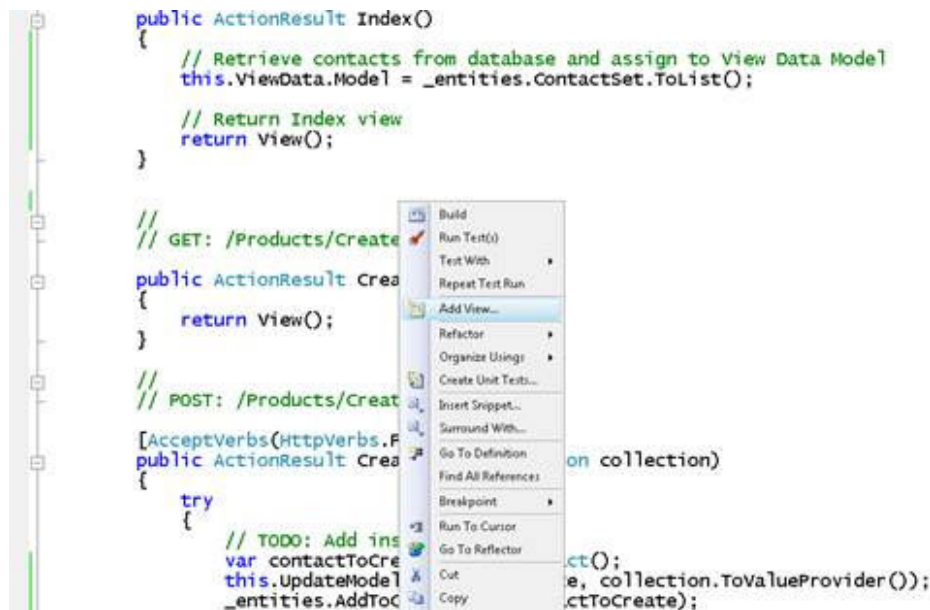


Figure 16: Ajout de la vue de création de contacts

Dans la fenêtre d'ajout d'une vue, choisissez la classe `ContactManager.Models.Contact` et l'option création (Create) comme dans la Figure 17. Lorsque vous cliquez sur le bouton Add, une vue de création est automatiquement générée pour vous.

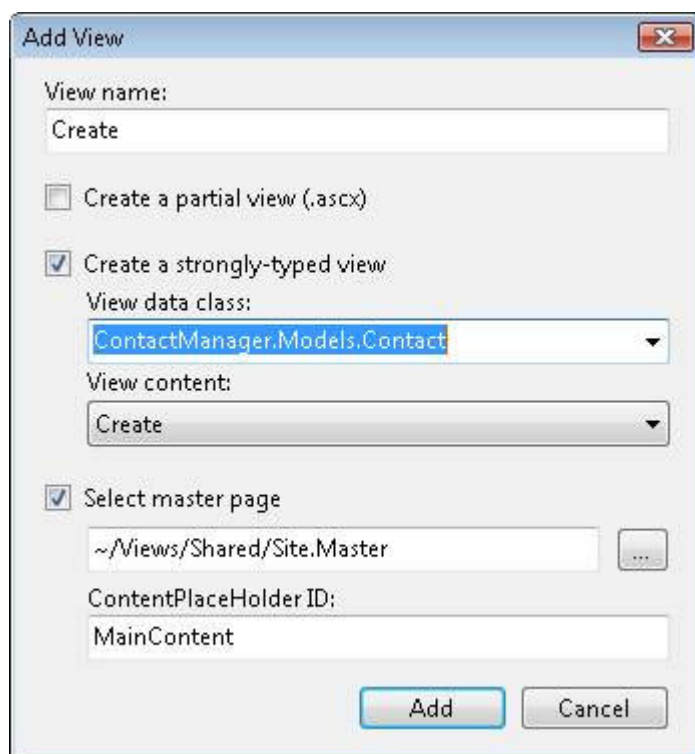


Figure 17: configuration de la vue d'ajout d'un contact

La vue de création contient des champs pour chacune des propriétés de la classe `Contact`. N'oubliez donc pas de retirer la partie liée à la propriété `Id` pour arriver au code présenté dans le Listing 5.

Listing 5 – Views\Home\Create.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary("Create was unsuccessful. Please correct the
errors and try again.") %>

    <% using (Html.BeginForm()) {%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="FirstName">FirstName:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">LastName:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

        <% } %>

        <div>
            <%=Html.ActionLink("Back to List", "Index") %>
        </div>

    </asp:Content>
```

Après avoir modifié les méthodes Create() et ajouter la vue correspondante, vous pouvez désormais lancer l'application Contact Manager et créer de nouveaux contacts. Cliquez sur le lien Create New

en bas de la page Index pour vous rendre sur la vue de création. Vous devriez avoir un résultat similaire à la Figure 18.

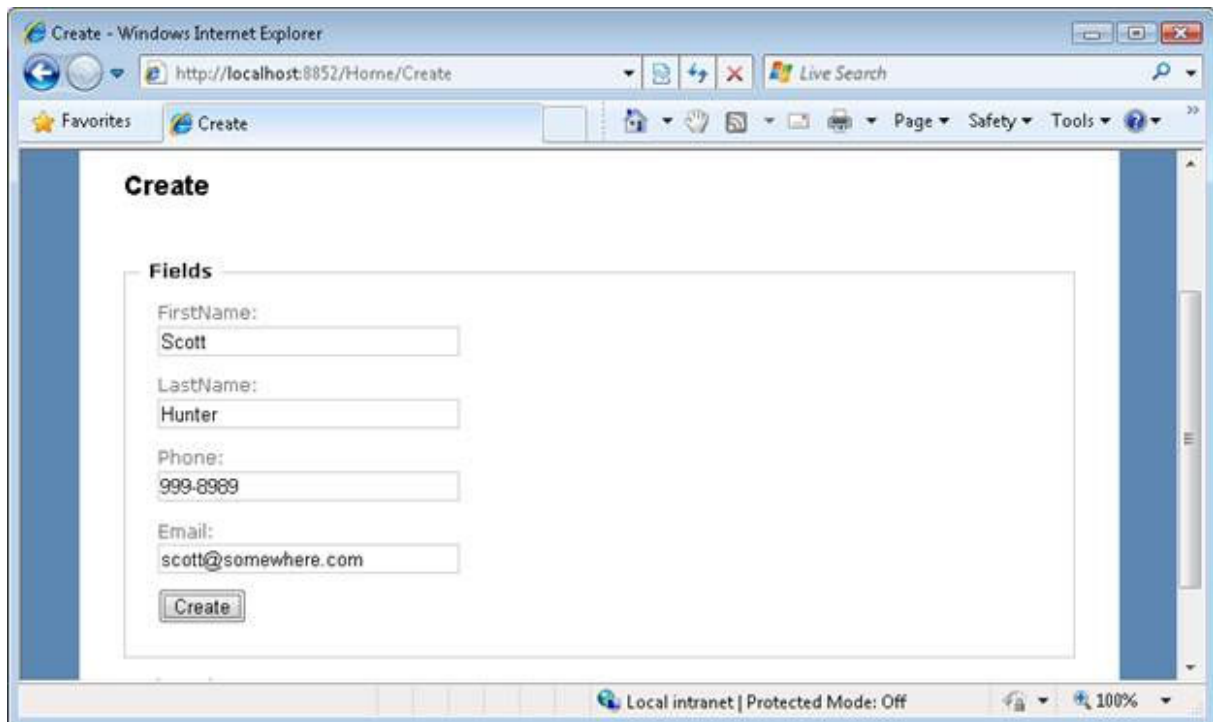


Figure 18: la vue de création de contacts

Editer les contacts

Ajouter la fonctionnalité d'édition d'un enregistrement est très similaire à l'ajout de la création d'un nouveau contact. Tout d'abord, il nous faut ajouter 2 méthodes Edit() au contrôleur Home. Ces 2 méthodes Edit() se trouve dans le Listing 6.

Listing 6 – Controllers\HomeController.cs (avec les méthodes Edit)

```
//  
// GET: /Home/Edit/5  
  
public ActionResult Edit(int id)  
{  
    var contactToEdit = (from c in _entities.ContactSet  
                        where c.Id == id  
                        select c).FirstOrDefault();  
  
    return View(contactToEdit);  
}  
  
//  
// POST: /Home/Edit/5  
  
[AcceptVerbs(HttpVerbs.Post)]  
public ActionResult Edit(Contact contactToEdit)  
{
```

```

        if (!ModelState.IsValid)
            return View();

        try
        {
            var originalContact = (from c in _entities.ContactSet
                                   where c.Id == contactToEdit.Id
                                   select c).FirstOrDefault();

            _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName,
            contactToEdit);
            _entities.SaveChanges();
            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }
}

```

La première méthode `Edit()` est invoquée via une opération de type HTTP GET. Un paramètre `Id` est passé à cette méthode représentant l'identifiant de l'enregistrement devant être édité. Entity Framework est alors utilisé pour retrouver un contact correspondant à cette identifiant. Ensuite, une vue contenant un formulaire HTML pour éditer cet enregistrement est retournée.

La deuxième méthode `Edit()` s'occupe de la mise à jour des données vers la base. Cette méthode accepte une instance de la classe `Contact` comme paramètre. A nouveau, le framework ASP.NET MVC effectue une liaison automatique entre les valeurs des champs saisis dans le formulaire d'édition et les propriétés de cette classe. Notez que cette fois-ci nous n'utilisons pas l'attribut `[Bind(Exclude)]` sur l'édition d'un contact car nous avons bien besoin de cette propriété pour terminer l'opération.

Entity Framework est également utilisé pour sauver le contact modifié dans la base de données. Pour cela, le contact original doit d'abord être retrouvé depuis la base. Ensuite, la méthode `ApplyPropertyChanges()` d'Entity Framework est appelée pour enregistrer les changements du contact en mémoire. Enfin, la méthode `SaveChanges()` d'Entity Framework est appelé pour faire persister ces changements dans la base de données sous-jacente.

De manière analogue à la création d'un contact, vous pouvez générer une vue contenant le formulaire d'édition en cliquant-droit sur la méthode `Edit()` et en choisissant l'option `Add View`. Dans la fenêtre d'ajout d'une vue, sélectionnez la classe `ContactManager.Models.Contact` et la valeur `Edit` pour View content (Figure 19).

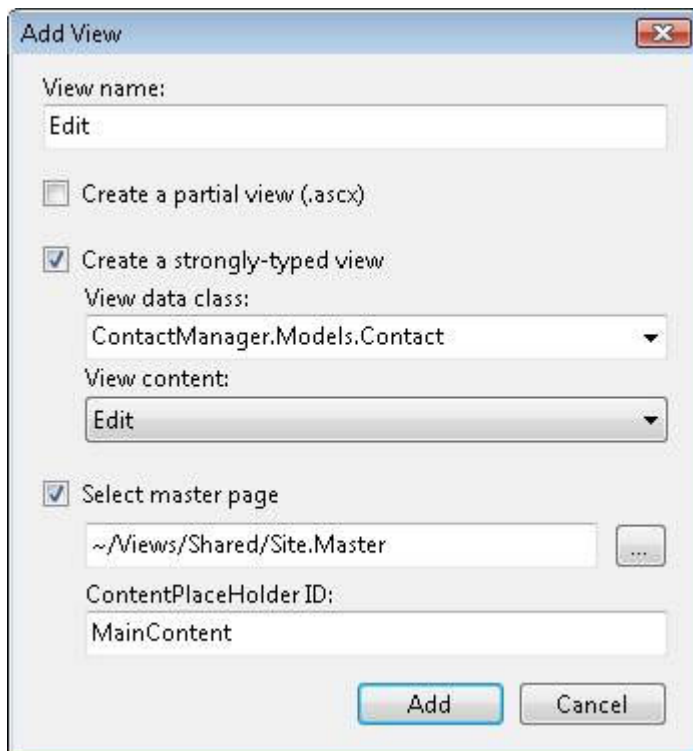


Figure 19: Ajout d'une vue d'édition

Lorsque vous cliquez sur bouton Add, une nouvelle vue d'édition est générée automatiquement pour vous. Le formulaire HTML qui est généré contient des champs correspondant à nouveau à chacune des propriétés de la classe Contact. A nouveau, retirer la partie liée à la propriété Id pour se retrouver avec le code du Listing 7.

Listing 7 – Views\Home\Edit.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Edit</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit</h2>

    <%= Html.ValidationSummary() %>

    <% using (Html.BeginForm()) {%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="FirstName">FirstName:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "") %>
            </p>
            <p>
                <label for="LastName">LastName:</label>
```

```

        <%= Html.TextBox("LastName") %>
        <%= Html.ValidationMessage("LastName", "") %>
    </p>
    <p>
        <label for="Phone">Phone:</label>
        <%= Html.TextBox("Phone") %>
        <%= Html.ValidationMessage("Phone", "") %>
    </p>
    <p>
        <label for="Email">Email:</label>
        <%= Html.TextBox("Email") %>
        <%= Html.ValidationMessage("Email", "") %>
    </p>
    <p>
        <input type="submit" value="Save" />
    </p>
</fieldset>

<% } %>

<div>
    <%=Html.ActionLink("Back to List", "Index") %>
</div>

</asp:Content>

```

Suppression des contacts

Si vous souhaitez supprimer des contacts alors vous devez ajouter 2 méthodes d'actions Delete() au contrôleur Home. La première action Delete() affiche un formulaire de confirmation de suppression. Le 2ème Delete() s'occupe de réellement supprimer l'enregistrement.

Plus tard, dans l'étape #7, nous modifierons l'application Contact Manager pour qu'elle supporte une suppression à l'aide de la technologie Ajax.

Les 2 nouvelles méthodes Delete() se trouve dans le Listing 8

Listing 8 – Controllers\HomeController.cs (méthodes Delete)

```

//
// GET: /Home/Delete/5

public ActionResult Delete(int id)
{
    var contactToDelete = (from c in _entities.ContactSet
                           where c.Id == id
                           select c).FirstOrDefault();

    return View(contactToDelete);
}

//
// POST: /Home/Delete/5

```

```

[AcceptVerbs (HttpVerbs.Post)]
public ActionResult Delete(Contact contactToDelete)
{
    try
    {
        var originalContact = (from c in _entities.ContactSet
                               where c.Id == contactToDelete.Id
                               select c).FirstOrDefault();

        _entities.DeleteObject(originalContact);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

```

La première méthode Delete() retourne un formulaire de confirmation de suppression du contact de la base (Figure 20). La 2ème méthode Delete() s'occupe donc d'effectuer l'opération de suppression de la base. Une fois le contact original retrouvé depuis la base de données, les méthodes DeleteObject() et SaveChanges() d'Entity Framework sont appelées pour effectuer la suppression.

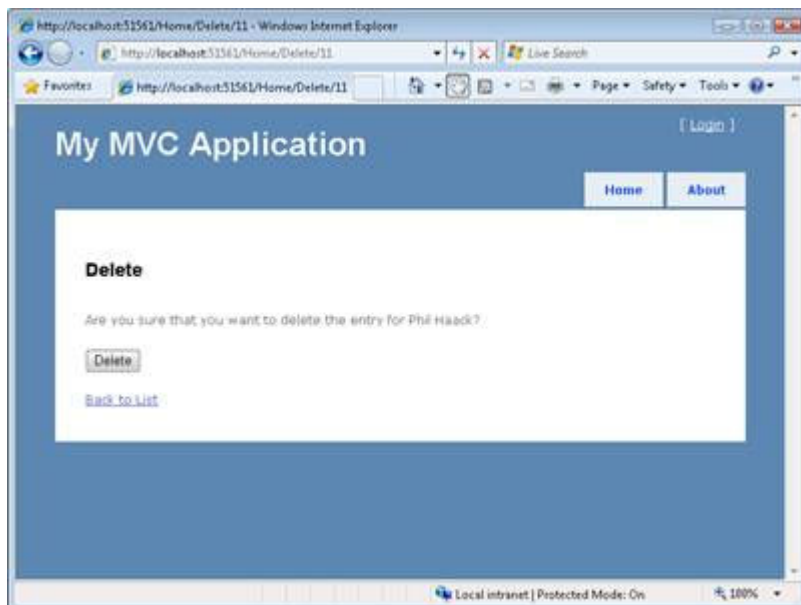


Figure 20: La vue affichant le formulaire de confirmation de suppression

Il nous faut modifier la vue Index pour qu'elle contienne un lien vers la suppression d'un enregistrement (Figure 21). Pour cela, il vous faut ajouter le code suivant dans la même cellule contenant le lien d'édition :

```
<%= Html.ActionLink("Delete", "Delete", new { id=item.Id }) %>
```

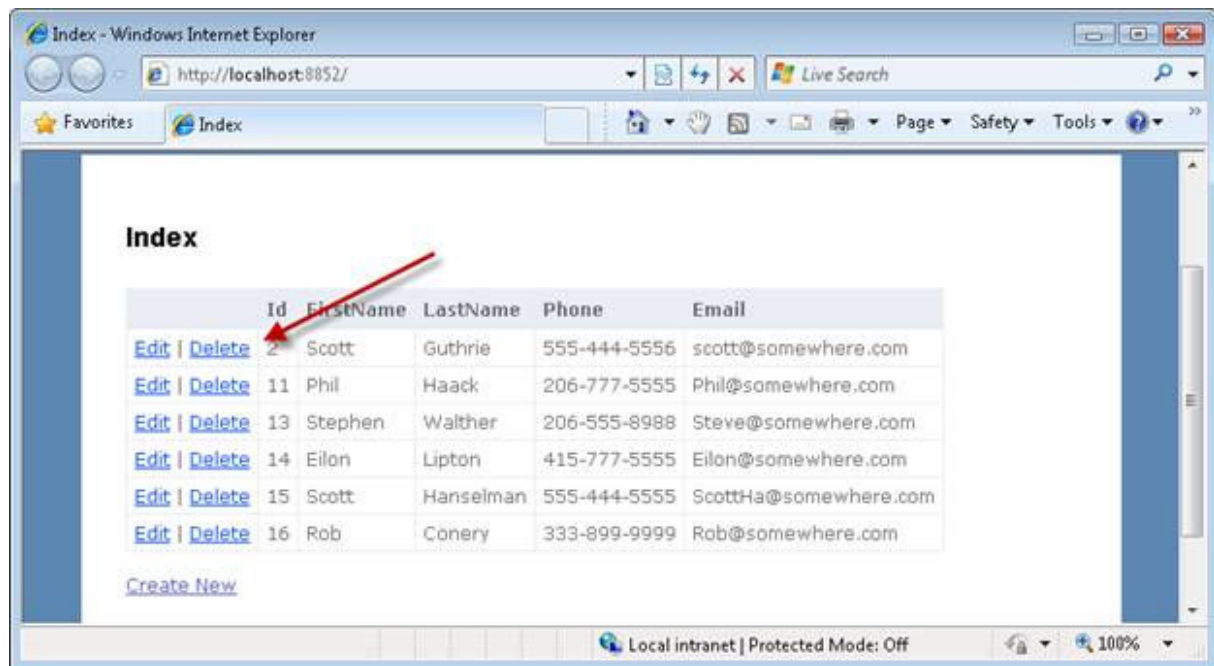


Figure 21: la vue Index avec le lien d'édition

Maintenant, il nous faut créer la vue de confirmation de suppression. Cliquez-droit sur la méthode Delete() dans le contrôleur Home et choisissez d'y ajouter une vue. La fenêtre classique d'ajout de vue apparaît (Figure 22).

Contrairement aux vues pour lister, créer et éditer les vues, nous n'aurons pas ici la possibilité de créer directement une vue de suppression. A la place, choisissez toujours la classe ContactManager.Models.Contact et la valeur Empty (vide) pour View content. Le fait de choisir une vue vide nous obligera simplement à créer la vue nous-même.

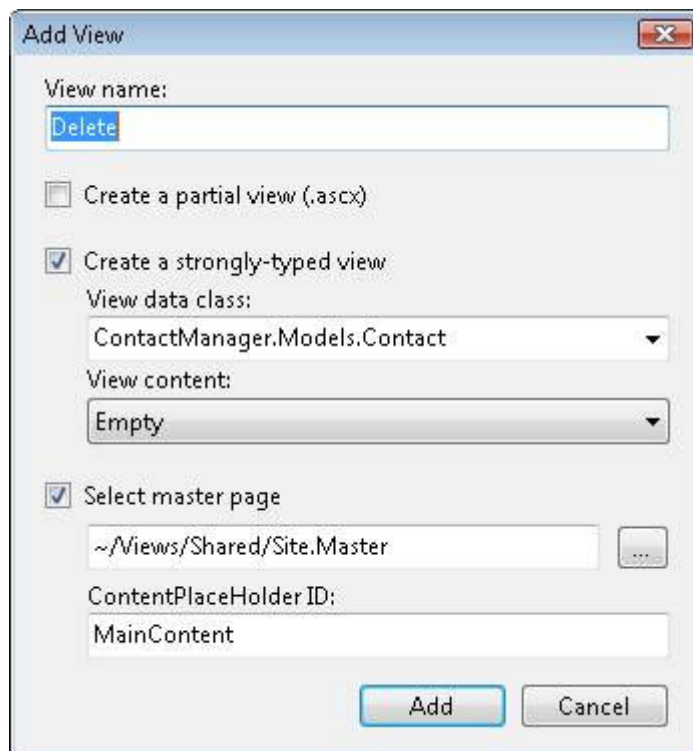


Figure 22: Ajout d'une vue de confirmation de suppression

Le contenu de la vue Delete est présent dans le Listing 9. Cette vue contient un formulaire demandant si oui ou non ce contact doit être supprimé (Figure 20).

Listing 9 – Views\Home\Delete.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Delete</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Delete</h2>

    <p>
        Are you sure that you want to delete the entry for
        <%= Model.FirstName %> <%= Model.LastName %>?
    </p>

    <% using (Html.BeginForm(new { Id = Model.Id }))
        { %>
        <p>
            <input type="submit" value="Delete" />
        </p>
        <% } %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>
</asp:Content>
```

Changer le nom du contrôleur par défaut

Cela peut peut-être vous déranger que le nom de la classe du contrôleur s'occupant des contacts s'appelle HomeController. Ne devrait-il pas plutôt s'appeler ContactController ?

Ce problème est assez simple à corriger. Tout d'abord, il nous faut changer le nom du contrôleur Home via le « Refactoring » de Visual Studio. Pour cela, ouvrez la classe HomeController dans l'éditeur de code de Visual Studio, cliquez-droit sur le nom de la classe et choisissez l'option Refactor, Rename. La fenêtre associée s'ouvre alors.

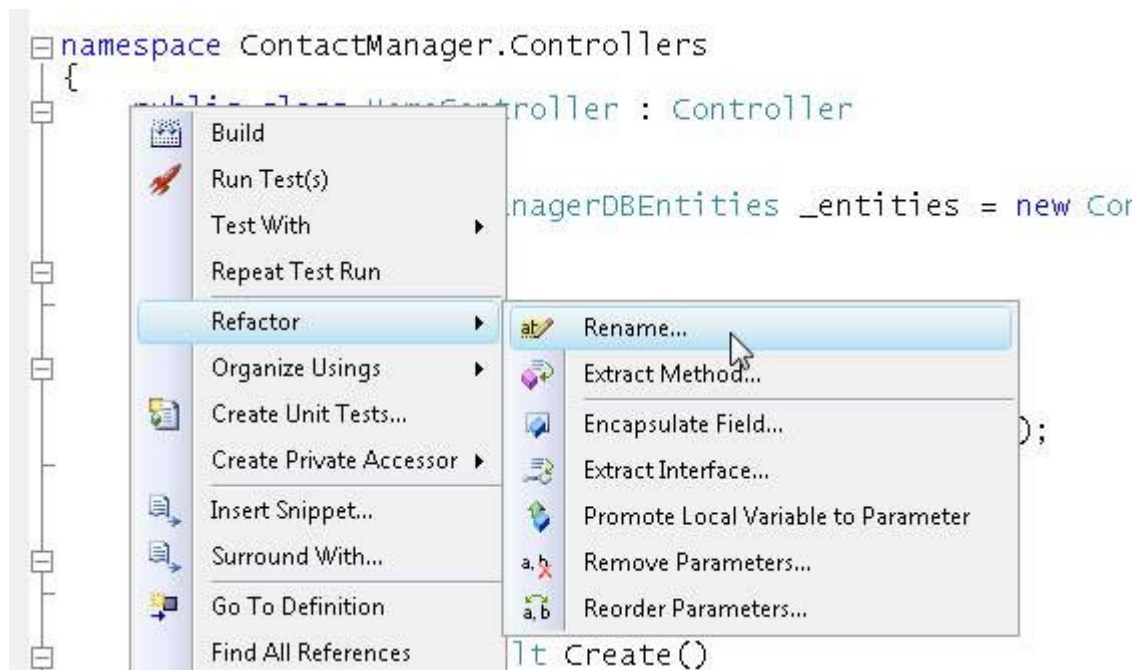


Figure 23: Comment renommer un contrôleur avec Visual Studio

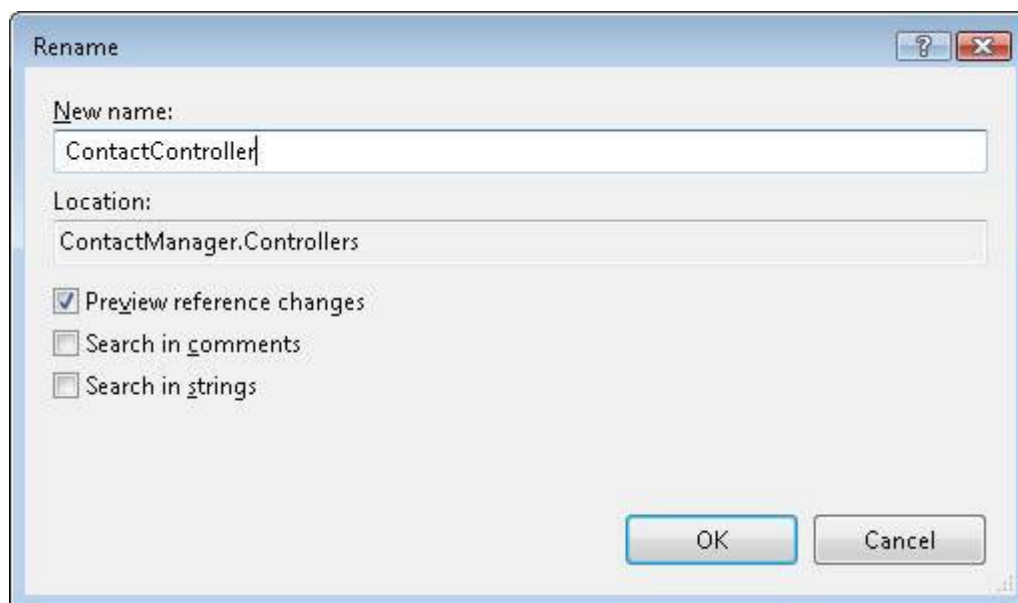


Figure 24: la fenêtre permettant le renommage

Si vous renommez le nom d'une classe d'un contrôleur, Visual Studio s'occupera alors de mettre à jour le nom du dossier dans le répertoire Views également. Ainsi Visual Studio va renommer le répertoire `\Views\Home` en `\Views\Contact`.

Cependant, après ce changement, votre application n'aura donc plus de contrôleur par défaut. Ainsi, si vous relancez votre application, vous obtiendrez la page d'erreur de la Figure 25.

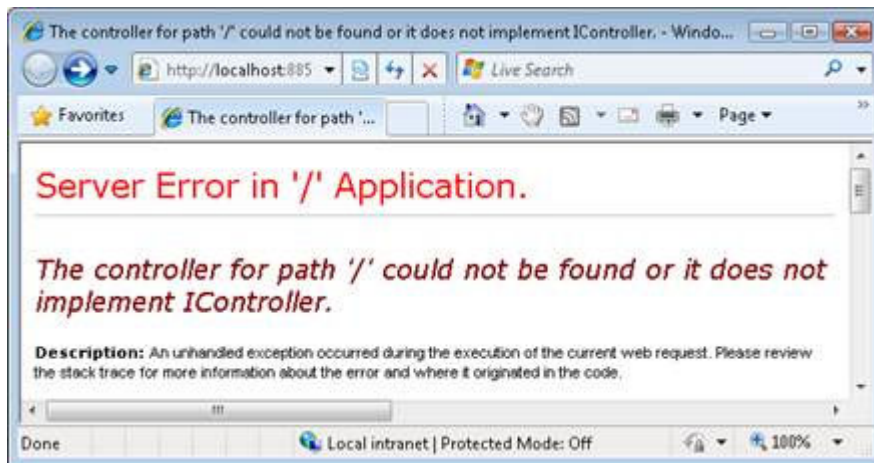


Figure 25: Erreur levée sans contrôleur par défaut

Il nous faut alors mettre à jour la route par défaut dans le fichier Global.asax afin d'utiliser le contrôleur Contact à la place du contrôleur Home. Pour cela, ouvrez le fichier Global.asax et modifiez le contrôleur par défaut pour la route par défaut (Listing 10).

Listing 10 – Global.asax.cs

```
using System.Web.Mvc;
using System.Web.Routing;

namespace ContactManager
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Default",                                // Route name
                "{controller}/{action}/{id}",             // URL with parameters
                new { controller = "Contact", action = "Index", id = "" } // Parameter defaults
            );
        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

Après avoir effectué ces changements, l'application Contact Manager fonctionnera correctement à nouveau. Il utilisera désormais en effet la classe Contact comme contrôleur par défaut.

Conclusion

Dans cette 1ère étape, nous avons créé une application basique de gestion de contacts de manière très rapide. Nous avons profité des bénéfices de Visual Studio pour générer automatiquement le

code initial de nos contrôleurs et les vues associées. Nous avons également profité d'Entity Framework pour générer automatiquement notre modèle de classes à partir d'une base de données.

Désormais, nous pouvons ainsi lister, créer, éditer et supprimer des enregistrements de type contact avec notre application Contact Manager. En d'autres termes, nous pouvons effectuer toutes les opérations de base d'une application web connectée à une base de données.

Cependant, notre application a encore quelques petits soucis. Tout d'abord, avouons que notre application Contact Manager n'est pas l'application la plus attractive qui soit. Elle nécessite un peu de travail de design. Justement, dans la prochaine étape, nous verrons comment on peut changer la page maître par défaut et la feuille de style CSS pour améliorer l'apparence de l'application.

D'autre part, nous n'avons pas encore mis en place la moindre forme de validation. Par exemple, rien ne vous empêche de soumettre le formulaire de création d'un nouveau contact sans entrer la moindre valeur dans les champs. Par ailleurs, vous pouvez également entrer des valeurs non cohérentes pour les numéros de téléphone ou adresses email. Nous commencerons à aborder le problème de la validation de formulaire à l'étape #3.

Enfin, le point le plus important à ce stade là, notre application Contact Manager ne peut pas être modifiée et maintenue de manière simple. Par exemple, la logique d'accès à la base de données est contenue directement au sein des actions contrôleurs. Cela entraîne donc que nous ne pourrions pas modifier notre code d'accès aux données sans modifier nos contrôleurs. Dans une étape ultérieure, nous verrons quelques modèles de développement logiciel (Design Patterns) que nous pourrions implémenter pour rendre l'application plus à même d'être facilement modifiée.

Etape 2 – Rendre l'application plus attrayante

Dans cette étape

Le but de cette étape est de rendre l'application plus jolie. Pour le moment l'application à l'apparence par défaut du model ASP.NET MVC. La PageMaster et la Css (cascading style sheet) attachée propose le design suivant (Figure 1). Ce design n'est pas très joli et surtout c'est le design de toutes les application MVC de démo. On va donc le remplacer pour en faire un personnalisé.

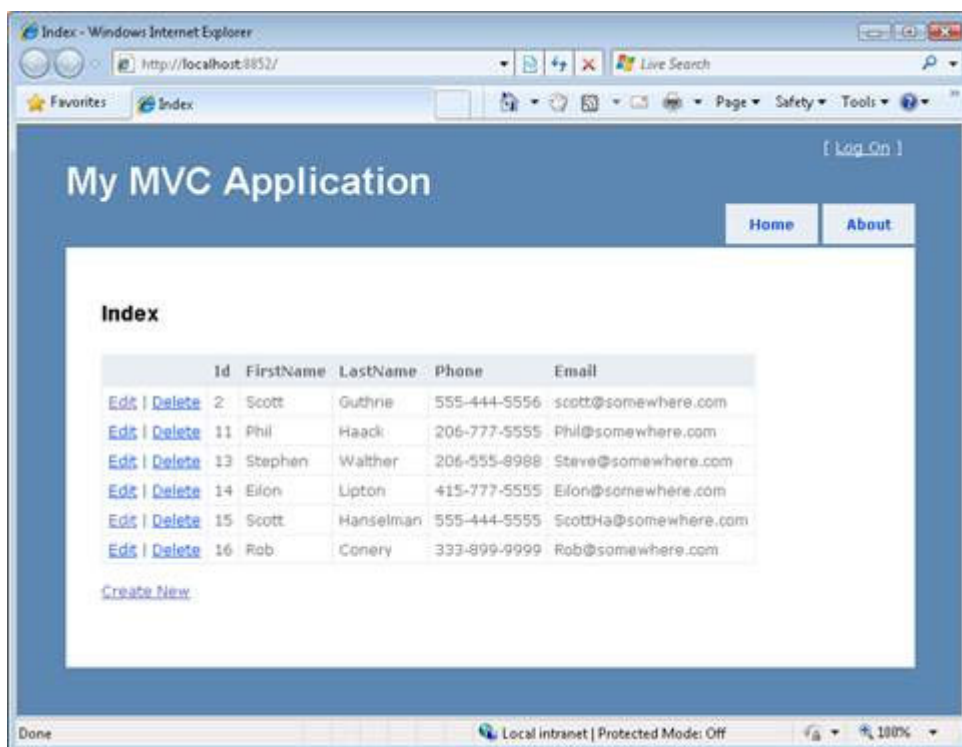


Figure 01: Design par défaut des applications ASP.NET MVC Application

Dans cette étape nous allons voir deux approches pour améliorer le design de l'application.

- Premièrement nous allons voir comment tirer partie du model ASP.NET MVC en téléchargeant simplement de nouveau « Model ». ASP.NET MVC “Design Gallery” permet de créer des applications designer professionnels sans rien faire.
- Deuxièmement comment créer son propre model avec l'aide d'un designer professionnel.

Utilisation de ASP.NET MVC “Design Gallery »

Pour utiliser les ressources gratuites de ASP.NET MVC « Design Gallery » de Microsoft il suffit d'aller à cette Url :

<http://www.ASP.net/mvc/gallery>

ASP.NET MVC « Design Gallery» héberge des collections de sites gratuits déjà designés. C'est sites sont uploadés par les personnes de la communauté. Les visiteurs de ASP.NET MVC « Design Gallery » peuvent voter sur les sites pour choisir leur favorite. (Voir Figure 2).



Figure 02: ASP.NET MVC « Design Gallery»

A l'écriture de ce tutorial le site Web le mieux noté était « October by David Hauser ». Nous allons utiliser celui là pour la suite de cette étape.

1. Cliquer sur le bouton « Download » et sauvegardez le ZIP sur votre ordinateur.
2. Faire bouton droit sur le zip et cliquez « Unblock » (voir Figure 3)
3. Dé-ziper le fichier dans le répertoire « October »
4. Sélectionner tous les fichiers contenus dans le répertoire "Octobre" et faire "Copier"
5. Allez dans Visual Studio dans « ContactManager » et faire « Coller » (Voir figure 4)
6. Allez dans le menu Edit de Visual Studio et faire « Rechercher / Remplacer »
 - a. Remplacer [YourProjectName]
 - b. Par ContactManager (Voir figure 5)

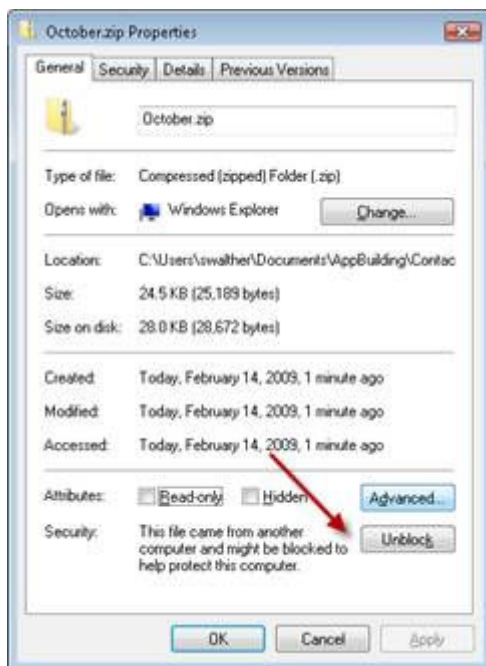


Figure 03: Débloquer le fichier Zip téléchargé du Web



Figure 04: Coller les fichiers dans Visual Studio (Solution explorer)

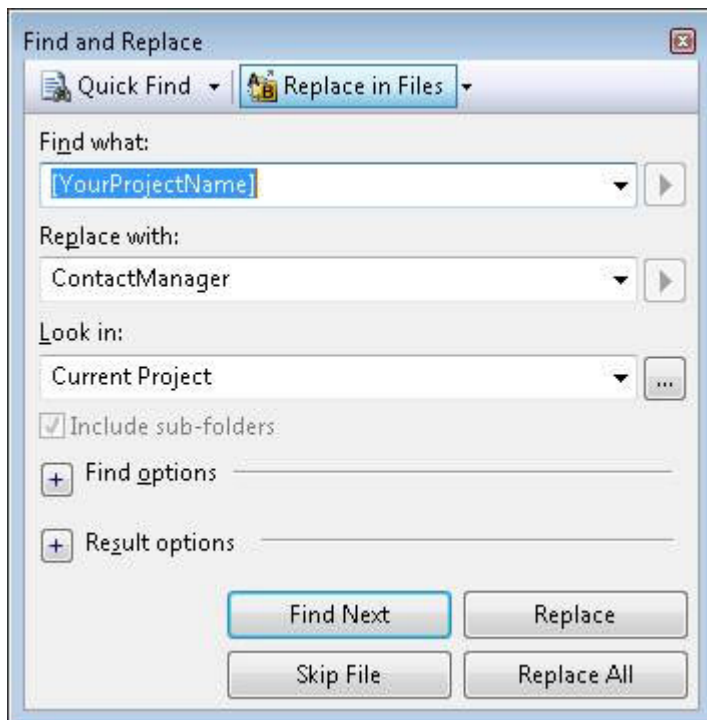


Figure 05: Replace [ProjectName] par ContactManager

Après avoir fini ces étapes, votre application devrait avoir ce nouvel aspect (voir Figure 6) avec l'application du design (October)

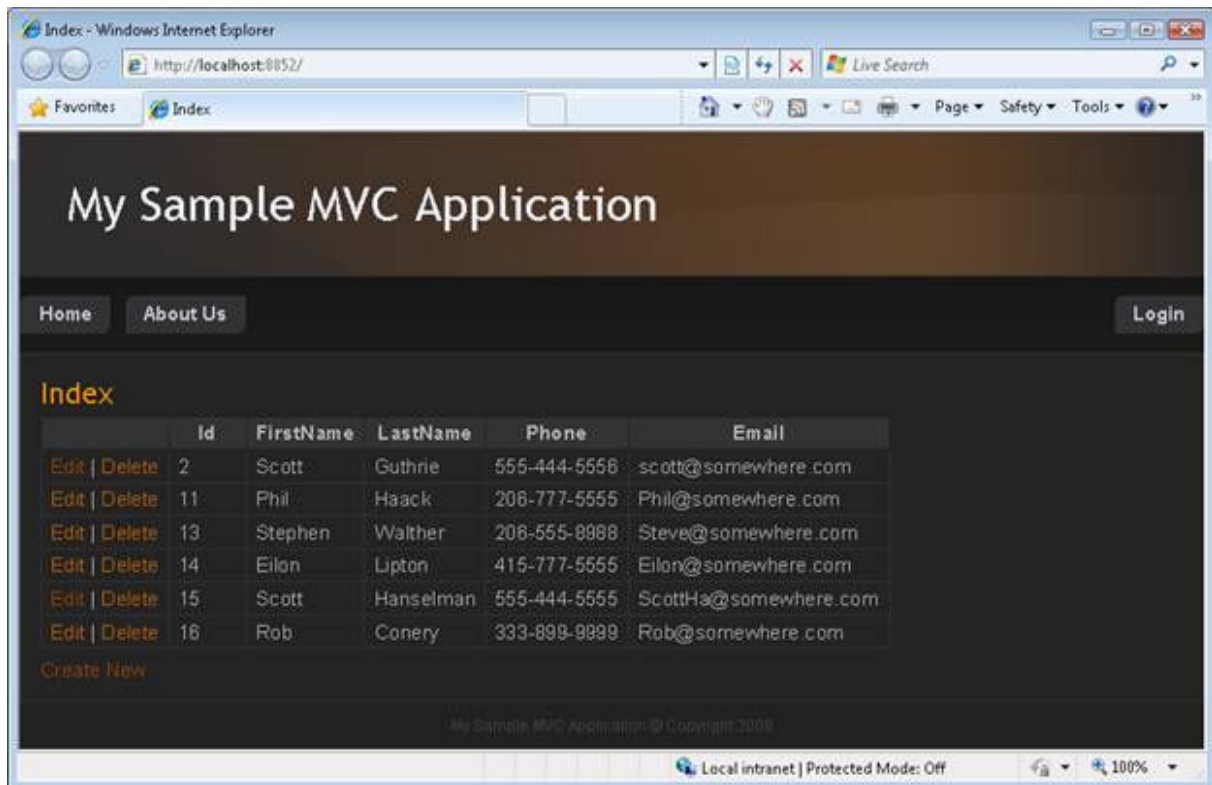


Figure 06: ContactManager avec le design de October

Création d'un design personnalisé

L'ASP.NET MVC Design Gallery est un bon début pour trouver différents styles. La gallery offre une bonne palette de choix de couleur et a pour avantage d'être totalement gratuit.

Mais parfois le design ne répond pas exactement à la demande. A ce moment il faut faire appel à un designer. Nous allons voir cette nouvelles maintenant pour venir personnaliser l'application de Gestion de Contacts.

Nous allons nous mettre dans une situation réel où l'application est finaliser et elle va être transmis aux équipes de design sous forme d'un zip. L'équipe de design n'a pas Visual Studio. Il faudra donc qu'il se munisse de la version gratuite de Visual Studio « Visual Web Développeur » accessible depuis le site <http://www.asp.net>. Le designer pourra donc ouvrir la solution, la lancer et il obtiendra ceci (Voir figure 7).

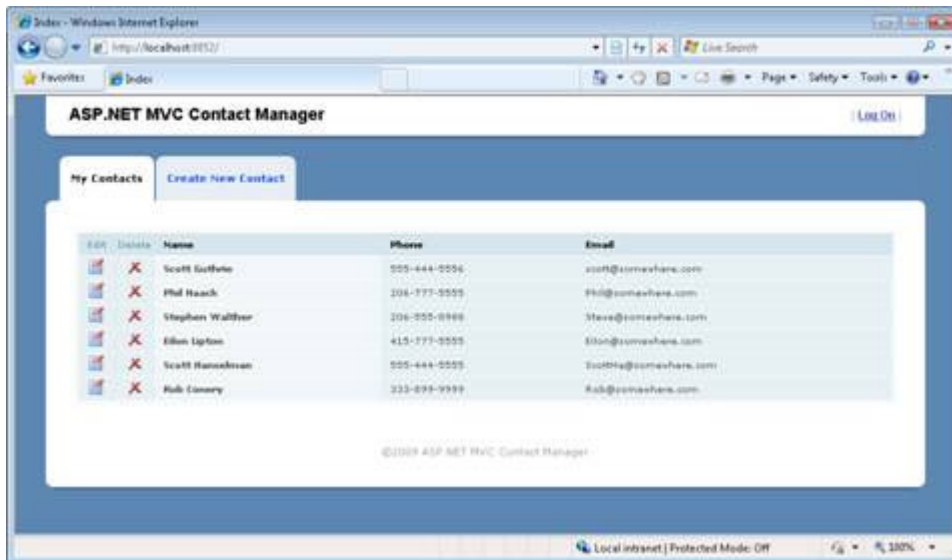


Figure 07: Le design de l'application Gestion de Contact ASP.NET MVC

La création d'un nouveau design se résume à créer deux fichiers : Un fichier CSS et une vue de type MasterPage. La vue MasterPage contient le système de positionnement et le contenu partagé pour les vues dans l'application ASP.NET MVC. Par exemple la vue de la MasterPage contient l'entête, la navigation par onglet et le pied de page (voir figure 7). On va écraser la vue existante Site.Master qui est dans le répertoire « Views\Shared » par la nouvelle générée par la société de design.

La société de design va de son côté créer une nouvelle CSS avec ses Images. On va placer ces nouveaux fichiers dans le répertoire « Content » et écraser le fichier existant « Site.css ». Il faudra mettre l'ensemble des ressources statiques dans le répertoire « Content »

Dans le nouveau design de l'application « Gestion de contact » le designer a utilisé des images pour les commandes Edit et Delete. Alors qu'elles étaient en HTML.

A l'origine les liens étaient de type texte :

```
<td>
    <%= Html.ActionLink("Edit", "Edit", new { id=item.Id }) %> |
    <%= Html.ActionLink("Delete", "Delete", new { id=item.Id }) %>
</td>
```

La méthode `Html.ActionLink` ne supporte pas les images (La méthode HTML encode les liens test pour des raisons de sécurité. Par conséquent nous allons remplacer `Html.ActionLink()` par `Url.Action` comme ceci :

```
<td class="actions edit">
    <a href='<%= Url.Action("Edit", new {id=item.Id}) %>'></a>
</td>
<td class="actions delete">
    <a href='<%= Url.Action("Delete", new {id=item.Id}) %>'></a>
</td>
```

La méthode `Html.ActionLink()` a un rendu HTML d'hyperlien alors que la méthode `Url.Action()` a un rendu HTML d'une URL simplement sans le tag `<a>`

On note aussi que dans le design il y a une notion d'onglet « sélectionné » et « non-sélectionné ». (Voir figure 8)



Figure 08: Tabulation “sélectionné” et “non-sélectionné”

Pour implémenter ces deux notions de sélection et de non-sélection, nous allons créer un Helper HTML « MenuItemHelper ». Cet helper va avoir pour rendu : un tag non-sélectionné et un tag sélectionné <li class= "selected"> et ce en fonction du contexte qui sera fourni par le controller. Le code de la classe MenuItemHelper sera comme ceci (voir Listing 1).

Listing 1 – Helpers\MenuItemHelper.cs

```
using System;
using System.Web.Mvc;
using System.Web.Mvc.Html;

namespace Helpers
{
    /// <summary>
    /// This helper method renders a link within an HTML LI tag.
    /// A class="selected" attribute is added to the tag when
    /// the link being rendered corresponds to the current
    /// controller and action.
    ///
    /// This helper method is used in the Site.Master View
    /// Master Page to display the website menu.
    /// </summary>
    public static class MenuItemHelper
    {
        public static string MenuItem(this HtmlHelper helper, string linkText,
            string actionName, string controllerName)
        {
            string currentControllerName =
            (string)helper.ViewContext.RouteData.Values["controller"];
            string currentActionName =
            (string)helper.ViewContext.RouteData.Values["action"];

            var builder = new TagBuilder("li");

            // Add selected class
            if (currentControllerName.Equals(controllerName,
                StringComparison.CurrentCultureIgnoreCase) &&
                currentActionName.Equals(actionName,
                StringComparison.CurrentCultureIgnoreCase))
                builder.AddCssClass("selected");

            // Add link
            builder.InnerHtml = helper.ActionLink(linkText, actionName,
                controllerName);

            // Render Tag Builder
        }
    }
}
```

```
        return builder.ToString(TagRenderMode.Normal);  
    }  
  
    }  
}
```

La class MenuItemHelper utilise l'object TagBuilder pour simplifier l'écriture des tag pour le rendu HTML. Cette classe est vraiment très utile pour créé des tags, elle a des méthodes pour ajouter des attributs, ajouter des class CSS, générer des Ids et même modifier des « innerHTML » dans des tags.

Conclusion

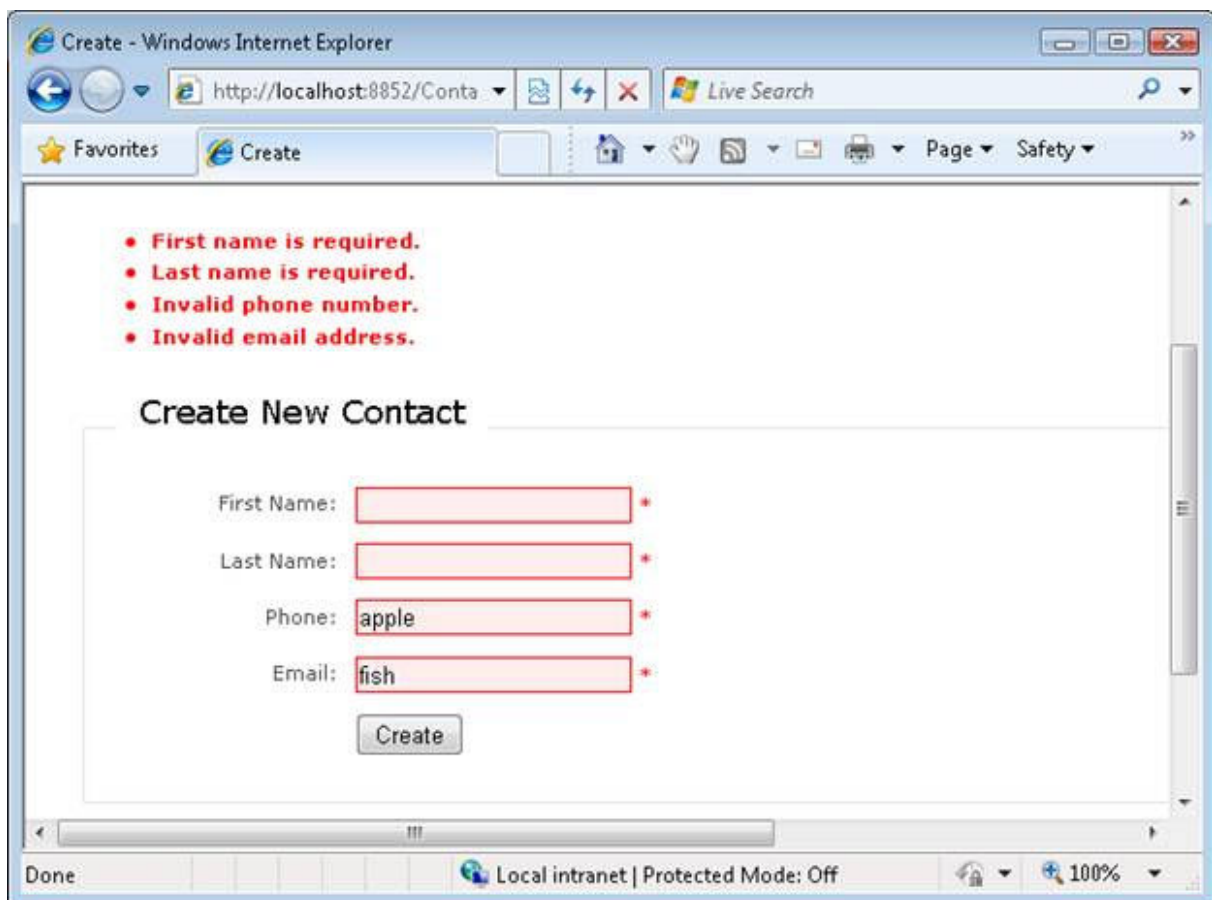
Dans cette étape, nous avons amélioré le design de l'application MVC. Dans un premier temps en remplaçant le design par un autre qui provenait de ASP.NET MVC "Design Gallery" et dans un deuxième temps nous avons appris à créer un nouveau design personnalisé en modifiant le site.master et la CSS. La modification du design peut être très simple mais elle peut aller aussi jusqu'à la création de helper pour des besoins spécifiques comme nous l'avons réalisé avec MenuItemHelper.

Dans la prochaine session nous allons voir un sujet important : la validation de formulaire. Nous allons empêcher les utilisateurs de soumettre un formulaire sans remplir certains champs obligatoires. Nous allons également valider les adresses email et numéros de téléphone.

Etape 3 – Ajout de la validation de formulaires

Dans cette étape

Dans cette 3ème étape dans la fabrication de notre application Contact Manager, nous allons ajouter une logique simple de validation de formulaires. Nous souhaitons éviter de laisser les utilisateurs soumettre un nouveau contact sans rentrer d’abord des valeurs dans les champs obligatoires. Nous souhaitons également valider la bonne constitution des numéros de téléphone et des adresses emails proposés (Figure 1).



The screenshot shows a Windows Internet Explorer window titled 'Create - Windows Internet Explorer'. The address bar displays 'http://localhost:8852/Conta'. The page content includes a list of validation errors in red text:

- First name is required.
- Last name is required.
- Invalid phone number.
- Invalid email address.

Below the errors is a section titled 'Create New Contact' containing a form with the following fields:

- First Name: *
- Last Name: *
- Phone: *
- Email: *

A 'Create' button is located below the form fields. The browser's status bar at the bottom indicates 'Done', 'Local intranet | Protected Mode: Off', and a zoom level of '100%'.

Figure 01: Un formulaire avec une logique de validation en place

Dans cette étape, nous allons ajouter la logique de validation directement au sein des actions contrôleurs. En général, cela n’est pas la méthode recommandée pour ajouter une forme de validation dans une application ASP.NET MVC. Une meilleure approche serait de mettre en place une logique de validation applicative dans une couche de service séparée. Justement, dans l’étape suivante, nous nous occuperons de revoir l’architecture de notre application pour faciliter les opérations de maintenance.

Par ailleurs, toujours afin de rester pour l’instant assez simple, nous allons écrire l’ensemble du code de validation à la main. Au lieu de l’écrire nous-mêmes, on pourrait envisager de bénéficier d’un framework de validation. Par exemple, on peut utiliser Microsoft Entreprise Library Validation

Application Block (VAB) pour implémenter la logique de validation au sein d'ASP.NET MVC. Pour en savoir davantage sur Validation Application Block, rendez-vous ici : <http://msdn.microsoft.com/en-us/library/dd203099.aspx>

Ajout de validations à la vue de création

Commençons par ajouter de la logique de validation à la vue de création de contacts. Heureusement, comme nous avons généré la vue avec Visual Studio, elle contient déjà toute la logique nécessaire à l'affichage de messages de validation au niveau de l'interface graphique. La vue de création est présentée dans le Listing 1.

Listing 1 – \Views\Contact\Create.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <%= Html.ValidationSummary() %>

    <% using (Html.BeginForm()) {%>

        <fieldset class="fields">
            <legend>Create New Contact</legend>
            <p>
                <label for="FirstName">First Name:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">Last Name:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p class="submit">
                <input type="submit" value="Create" />
            </p>
        </fieldset>
```

```
<% } %>

</asp:Content>
```

Noter l'appel à la méthode `Html.ValidationSummary()` juste au dessus du formulaire HTML. S'il y a la moindre erreur de validation levée, alors cette méthode affiche l'ensemble des messages dans une liste à puce.

Notez également l'appel à la méthode `Html.ValidationMessage()` qui apparaît prêt de chaque champs du formulaire. La méthode `ValidationMessage()` affiche un message particulier d'erreur de validation. Dans le cas du Listing 1, une étoile est affichée lorsqu'une erreur de validation est rencontrée.

Pour terminer, la méthode `Html.TextBox()` génère automatiquement une feuille de style CSS lorsqu'il y a une erreur de validation associée à la propriété affichée. `Html.TextBox()` génère une classe appelée `input-validation-error`.

Lorsque vous créez une nouvelle application ASP.NET MVC, une feuille de style appelée `Site.css` est créée automatiquement dans le répertoire `Content`. Cette feuille de style contient les définitions suivantes pour les classes CSS liées à l'apparence des messages d'erreur de validation :

```
.field-validation-error
{
    color: #ff0000;
}

.input-validation-error
{
    border: 1px solid #ff0000;
    background-color: #ffebee;
}

.validation-summary-errors
{
    font-weight: bold;
    color: #ff0000;
}
```

La classe `field-validation-error` est utilisée pour contrôler l'apparence de ce qui est produit par la méthode `Html.ValidationMessage()`. La classe `The input-validation-error` est utilisée pour contrôler le style de la zone de saisie de texte rendue par la méthode `Html.TextBox()`. Enfin, la classe `validation-summary-errors` est utilisée pour contrôler le style de la liste non ordonnée rendue par la méthode `the Html.ValidationSummary()`.

Vous pouvez modifier les classes de la feuille de style décrite dans cette section pour personnaliser l'apparence des messages d'erreur de validation.

Ajout d'une logique de validation à l'action de création

Jusqu'à présent, la vue de création n'a jamais affiché de message d'erreur de validation car nous n'avons pas écrit la logique pour générer ces messages. Pour afficher des messages d'erreur de validation, vous devez d'abord les ajouter à l'objet ModelState.

La méthode UpdateModel() ajoute automatiquement des messages d'erreur à l'objet ModelState s'il y a une erreur simple d'assignation d'une valeur du formulaire vers une propriété. Par exemple, si vous essayez d'affecter la chaîne « pomme » vers une propriété DateAnniversaire de type DateTime alors la méthode UpdateModel() sera appelée pour ajouter une erreur à ModelState.

La version modifiée de la méthode Create() dans le Listing 2 contient une nouvelle section qui s'occupe de valider les propriétés de l'objet Contact fourni avant de tenter de l'insérer dans la base de données.

Listing 2 – Controllers\ContactController.cs (création avec validation)

```
//
// POST: /Contact/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)
{
    // Validation logic
    if (contactToCreate.FirstName.Trim().Length == 0)
        ModelState.AddModelError("FirstName", "First name is required.");
    if (contactToCreate.LastName.Trim().Length == 0)
        ModelState.AddModelError("LastName", "Last name is required.");
    if (contactToCreate.Phone.Length > 0 &&
        !Regex.IsMatch(contactToCreate.Phone, @"((\d{3}\d{3})|(\d{3}-))?\d{3}-\d{4}")
    )
        ModelState.AddModelError("Phone", "Invalid phone number.");
    if (contactToCreate.Email.Length > 0 &&
        !Regex.IsMatch(contactToCreate.Email, @"^[\w-\.] + @ ([\w-]+ \. )+ [\w-]{2,4}$")
    )
        ModelState.AddModelError("Email", "Invalid email address.");
    if (!ModelState.IsValid)
        return View();

    // Database logic
    try
    {
        _entities.AddToContactSet(contactToCreate);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

La section de validation applique 4 règles distinctes:

- La propriété FirstName (prénom) doit avoir une taille supérieure à 0 (et ne peut être constituée uniquement de caractères vides).
- La propriété LastName (nom) doit également avoir une taille supérieure à 0 (et ne peut être constituée uniquement de caractères vides).
- Si la propriété Phone (numéro de téléphone) a une valeur (une taille supérieure à 0 donc), alors cette propriété doit être validée par une expression régulière.
- Si la propriété Email a une valeur, alors cette propriété doit également être validée par une expression régulière.

Lorsqu'il y a une violation de l'une des règles de validation, un message d'erreur est alors ajouté à l'objet ModelState à l'aide de la method AddModelError(). Lorsque vous ajoutez un message à l'objet ModelState, vous devez fournir le nom d'une propriété ainsi qu'un texte contenant votre message d'erreur. Ce message est alors affiché dans la vue via les 2 méthodes Html.ValidationSummary() et Html.ValidationMessage() présentées précédemment.

Après que les règles de validation soient exécutées, la propriété IsValid de l'objet ModelState est vérifiée. La valeur booléenne est positionnée à faux lorsque la moindre erreur de validation fut ajoutée à l'objet ModelState. Ainsi, si la validation a échoué, le formulaire de création est affiché à nouveau avec les messages d'erreur correspondant.

Les expressions régulières pour valider le numéro de téléphone et les adresses emails viennent de la bibliothèque suivante: <http://regexlib.com>

Ajout d'une logique de validation dans l'action d'édition

L'action d'édition met à jour un contact. L'action Edit() a besoin d'effectuer exactement la même validation que l'action Create(). Plutôt que de dupliquer le même code de validation, mieux vaut factoriser cette partie de code de manière à ce que chacune des 2 actions Create() et Edit() appelle la même méthode de validation.

La classe du contrôleur Contact modifiée se trouve dans le Listing 3. Cette classe dispose d'une nouvelle méthode nommée ValidateContact() appelée au sein de chacune des 2 actions.

Listing 3 – Controllers\ContactController.cs

```
using System.Linq;
using System.Text.RegularExpressions;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class ContactController : Controller
    {
        private ContactManagerDBEntities _entities = new
            ContactManagerDBEntities();

        protected void ValidateContact(Contact contactToValidate)
        {
```



```

        if (contactToValidate.FirstName.Trim().Length == 0)
            ModelState.AddModelError("FirstName", "First name is
required.");
        if (contactToValidate.LastName.Trim().Length == 0)
            ModelState.AddModelError("LastName", "Last name is
required.");
        if (contactToValidate.Phone.Length > 0 &&
!Regex.IsMatch(contactToValidate.Phone, @"((\d{3}\ )?|(\d{3}-))?\d{3}-
\d{4}")
            ModelState.AddModelError("Phone", "Invalid phone number.");
        if (contactToValidate.Email.Length > 0 &&
!Regex.IsMatch(contactToValidate.Email, @"^[\w-\.] + @ ([\w-]+ \. )+ [\w-]{2,4}$"))
            ModelState.AddModelError("Email", "Invalid email address.");
    }

    public ActionResult Index()
    {
        return View(_entities.ContactSet.ToList());
    }

    public ActionResult Create()
    {
        return View();
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Create([Bind(Exclude = "Id")] Contact
contactToCreate)
    {
        // Validation logic
        ValidateContact(contactToCreate);
        if (!ModelState.IsValid)
            return View();

        // Database logic
        try
        {
            _entities.AddToContactSet(contactToCreate);
            _entities.SaveChanges();
            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }

    public ActionResult Edit(int id)
    {
        var contactToEdit = (from c in _entities.ContactSet
                             where c.Id == id
                             select c).FirstOrDefault();

        return View(contactToEdit);
    }

    [AcceptVerbs(HttpVerbs.Post)]

```

```

public ActionResult Edit(Contact contactToEdit)
{
    ValidateContact(contactToEdit);
    if (!ModelState.IsValid)
        return View();

    try
    {
        var originalContact = (from c in _entities.ContactSet
                               where c.Id == contactToEdit.Id
                               select c).FirstOrDefault();

        _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName,
        contactToEdit);

        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

public ActionResult Delete(int id)
{
    var contactToDelete = (from c in _entities.ContactSet
                           where c.Id == id
                           select c).FirstOrDefault();

    return View(contactToDelete);
}

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Delete(Contact contactToDelete)
{
    try
    {
        var originalContact = (from c in _entities.ContactSet
                               where c.Id == contactToDelete.Id
                               select c).FirstOrDefault();

        _entities.DeleteObject(originalContact);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
}
}

```

Résumé

Dans cette étape, nous avons ajouté une forme basique de validation de formulaires à notre application de gestion de contacts. Notre logique de validation permet d'éviter aux utilisateurs de soumettre de nouveaux contacts ou d'éditer des contacts existant sans préciser de valeur aux propriétés prénom et nom. De plus, l'utilisateur doit obligatoirement soumettre des numéros de téléphone et adresse email un tant soit peu valides.

Nous avons ajouté cette logique de validation de la manière la plus simple qui soit. Cependant, mélanger une logique de validation avec la logique même du contrôleur pourra créer des problèmes de maintenance à long terme.

Dans la prochaine étape, nous allons revoir l'architecture de notre logique de validation et de notre logique d'accès aux données pour les sortir de nos contrôleurs. Nous allons ainsi bénéficier de plusieurs principes d'architecture logicielle pour créer une application plus facile à maintenir et plus découpée.

Etape 4 – Rendre l'application faiblement couplée

Dans cette étape

Dans cette 4ème étape, nous allons profiter de plusieurs modèles de développement logiciel (Design Patterns) pour maintenir et modifier plus facilement notre application. Par exemple, nous allons revoir l'application pour utiliser 2 patterns connus sous le nom de « Repository pattern » et « Dependency Injection pattern »

Pour le moment, toute la partie d'accès aux données et de validation pour l'application de gestion de contacts est contenu dans la classe contrôleur. Ce n'est pas une très bonne idée car si vous décidez de modifier une partie de l'application, vous risquez d'introduire des bugs dans une autre partie de l'application. Par exemple, si vous modifiez une validation de données, vous risquez d'introduire un nouveau bug dans la partie d'accès aux données ou dans le contrôleur.

(SRP), une classe ne doit changer que pour une seule raison. Mixer contrôleur, validation et couche d'accès aux données n'est pas dans la philosophie du pattern « Single Responsibility Principle ». (Une classe par responsabilité)

Il a plusieurs raisons qui amènent à modifier une application. Vous voulez ajouter une nouvelle fonctionnalité, vous voulez résoudre un bug dans votre application ou modifier l'implémentation d'une fonctionnalité de l'application. Les applications sont rarement statiques. Elles ont tendance à grossir et évoluer dans le temps.

Imaginons, par exemple, qu'on veuille changer notre implémentation de notre accès aux données. Pour le moment l'application de Gestion de Contacts utilise Entity Framework pour accéder à la base. Vous pourriez très bien décider de migrer vers une technologie d'accès aux données différente telle que ADO.NET Data Service ou NHibernate. Cependant, parce que le code d'accès aux données entre la validation et le contrôleur n'est pas isolé, il est impossible d'en modifier le code sans modifier les autres parties de l'application qui ne sont pas directement liée à la couche d'accès aux données.

Quand une application est faiblement couplée, vous pouvez faire des modifications sur une partie de l'application sans modifier le reste. Par exemple, vous pouvez changer de couche d'accès aux données sans modifier la partie validation de données ou du contrôleur.

Dans cette étape, nous allons tirer partie des avantages de modèles de développement logiciel (Design Patterns) en faisant du refactoring sur l'application de Gestion de Contact pour rendre cette application faiblement couplée. Quand cela sera fait, l'application de Gestion de Contacts pourra être modifiée plus facilement.

Le "Refactoring" est un processus de réécriture de l'application mais sans perte de fonctionnalités.

Utilisation du modèle de développement (Repository)

La première modification va consister à tirer partie du modèle de développement appelé (Repository). Nous allons utiliser ce modèle pour isoler le code d'accès aux données du reste de l'application.

L'implémentation de ce modèle impose de suivre deux étapes :

1. Créer une interface
2. Créer une classe qui implémente cette interface.

Tout d'abord, vous devez créer une interface qui décrit toutes les méthodes d'accès aux données. C'est l'interface `ICollectionManagerRepository` contenue dans le « Listing 1 ». Cette interface décrit les 5 méthodes `CreateContact()`, `DeleteContact()`, `EditContact()`, `GetContact`, and `ListContacts()`.

Listing 1 – Models\ICollectionManagerRepository.cs

```
using System;
using System.Collections.Generic;

namespace ContactManager.Models
{
    public interface ICollectionRepository
    {
        Contact CreateContact(Contact contactToCreate);
        void DeleteContact(Contact contactToDelete);
        Contact EditContact(Contact contactToUpdate);
        Contact GetContact(int id);
        IEnumerable<Contact> ListContacts();
    }
}
```

Ensuite, vous devez créer une classe qui implémente l'interface `ICollectionManagerRepository`. Comme nous avons utilisé Microsoft Entity Framework pour la couche d'accès aux données, nous devons créer une nouvelle classe `EntityContactManagerRepository`. Cette classe est contenue dans le listing 2.

Listing 2 – Models\EntityContactManagerRepository.cs

```
using System.Collections.Generic;
using System.Linq;

namespace ContactManager.Models
{
    public class EntityContactManagerRepository :
        ContactManager.Models.ICollectionManagerRepository
    {
        private ContactManagerDBEntities _entities = new
            ContactManagerDBEntities();

        public Contact GetContact(int id)
        {
            return (from c in _entities.ContactSet
```

```

        where c.Id == id
        select c).FirstOrDefault();
    }

    public IEnumerable ListContacts()
    {
        return _entities.ContactSet.ToList();
    }

    public Contact CreateContact(Contact contactToCreate)
    {
        _entities.AddToContactSet(contactToCreate);
        _entities.SaveChanges();
        return contactToCreate;
    }

    public Contact EditContact(Contact contactToEdit)
    {
        var originalContact = GetContact(contactToEdit.Id);

        _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName,
        contactToEdit);
        _entities.SaveChanges();
        return contactToEdit;
    }

    public void DeleteContact(Contact contactToDelete)
    {
        var originalContact = GetContact(contactToDelete.Id);
        _entities.DeleteObject(originalContact);
        _entities.SaveChanges();
    }
}

```

On note que la classe `EntityContactManagerRepository` implémente l'interface `IContactManagerRepository`. La classe implémente donc les 5 méthodes d'accès aux données décrites dans l'interface.

Vous pouvez alors vous demander pourquoi avons-nous besoin d'une interface ?

A une exception près, l'application devra maintenant interagir avec l'interface et non avec la classe. Plutôt que d'appeler les méthodes exposées par la classe `EntityContactManagerRepository`, vous devrez appeler les méthodes exposées par l'interface `IContactManagerRepository`.

Comme cela vous pourrez faire une nouvelle implémentation de l'interface sans être obligé de modifier les appels faits dans l'application. Vous pouvez par exemple avoir deux implémentations de l'interface `IContactManagerRepository` sur deux framework d'accès aux données ADO.NET Data Services avec la classe `DataServiceContactManagerRepository` et Microsoft Entity Framework avec la classe `EntityContactManagerRepository`. Il sera donc possible de basculer de l'une à l'autre sans modifier la couche de validation ou la couche du contrôleur.

La programmation par interface par rapport à la programmation par classe rend l'application plus robuste aux changements.

Vous pouvez créer une interface sur une classe depuis Visual Studio en sélectionnant le menu option « Refactor / Extract Interface ». Par exemple vous pouvez créer en premier la classe EntityContactManagerRepository et utiliser l'extraction d'interface de Visual Studio pour créer automatiquement l'interface IContactManagerRepository.

Utilisation du modèle de programmation d'injection de dépendance.

Maintenant que vous avez migré le code d'accès aux données dans une classe « Repository », vous devez modifier le contrôleur Contact pour utiliser cette classe. Vous devez tirer partie du modèle de programmation « Injection de dépendance » afin d'utiliser la classe « Repository » dans le contrôleur.

Les modifications du contrôleur Contact sont contenues dans le listing 3.

Listing 3 – Controllers\ContactController.cs

```
using System.Text.RegularExpressions;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class ContactController : Controller
    {
        private IContactManagerRepository _repository;

        public ContactController()
        {
            : this(new EntityContactManagerRepository())
        }

        public ContactController(IContactManagerRepository repository)
        {
            _repository = repository;
        }

        protected void ValidateContact(Contact contactToValidate)
        {
            if (contactToValidate.FirstName.Trim().Length == 0)
                ModelState.AddModelError("FirstName", "First name is
required.");
            if (contactToValidate.LastName.Trim().Length == 0)
                ModelState.AddModelError("LastName", "Last name is
required.");
            if (contactToValidate.Phone.Length > 0 &&
!Regex.IsMatch(contactToValidate.Phone, @"((\d{3}) ?)(\d{3}-)?\d{3}-
\d{4}")
                ModelState.AddModelError("Phone", "Invalid phone number.");
            if (contactToValidate.Email.Length > 0 &&
!Regex.IsMatch(contactToValidate.Email, @"^[w-\.]@([w-]+\.[w-]{2,4}$"))
                ModelState.AddModelError("Email", "Invalid email address.");
        }

        public ActionResult Index()
```

```

    {
        return View(_repository.ListContacts());
    }

    public ActionResult Create()
    {
        return View();
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Create([Bind(Exclude = "Id")] Contact
contactToCreate)
    {
        // Validation logic
        ValidateContact(contactToCreate);
        if (!ModelState.IsValid)
            return View();

        // Database logic
        try
        {
            _repository.CreateContact(contactToCreate);
            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }

    public ActionResult Edit(int id)
    {
        return View(_repository.GetContact(id));
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(Contact contactToEdit)
    {
        // Validation logic
        ValidateContact(contactToEdit);
        if (!ModelState.IsValid)
            return View();

        // Database logic
        try
        {
            _repository.EditContact(contactToEdit);
            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }

    public ActionResult Delete(int id)
    {

```



```

        return View(_repository.GetContact(id));
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Delete(Contact contactToDelete)
    {
        try
        {
            _repository.DeleteContact(contactToDelete);
            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }
}

```

On notera que le contrôleur du Listing 3 dispose de deux constructeurs. Le premier prend en paramètre une instance alors que le deuxième prend en paramètre une interface `IContactManagerRepository`. Le contrôleur `Contact` utilise le « Constructeur d'injection de dépendance ».

De ce fait, un seul utilise `EntityContactManagerRepository` (seulement le premier). Le reste de la classe utilise l'interface `IContactManagerRepository`. Il est ainsi facile de basculer d'une implémentation à une autre en changeant le premier constructeur. Passer de `EntityContactManagerRepository` à `DataServicesContactRepository` se fait à un seul endroit dans la classe.

Autre avantage : maintenant le contrôleur `Contact` est testable. Dans le cadre d'un test unitaire vous pouvez instancier le contrôleur `Contact` en passant par une fausse implémentation de `IContactManagerRepository`. Cette fonctionnalité est très importante et sera utilisée dans la prochaine étape quand vous allez écrire des tests unitaires sur l'application de Gestion de Contacts.

Création d'une couche de service

Notez que la couche de validation restait mixée avec contrôleur dans le Listing 3. Pour les mêmes bonnes raisons qui nous ont amenées à isoler la couche d'accès aux données, nous allons isoler la couche de validation.

Pour se faire, nous allons créer une nouvelle couche: une couche de service. La couche de service doit se placer entre le contrôleur et la couche de « repository ». La couche de service contient la couche métier et toute la couche de validation.

La classe `ContactManagerService` se trouve dans le Listing 4. Elle contient la couche de validation initialement contenue dans le contrôleur `Contact`.

Listing 4 – Models\ContactManagerService.cs

```
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Web.Mvc;
using ContactManager.Models.Validation;

namespace ContactManager.Models
{
    public class ContactManagerService : IContactManagerService
    {
        private IValidationDictionary _validationDictionary;
        private IContactManagerRepository _repository;

        public ContactManagerService(IValidationDictionary validationDictionary)
            : this(validationDictionary, new EntityContactManagerRepository())
        {}

        public ContactManagerService(IValidationDictionary validationDictionary, IContactManagerRepository repository)
        {
            _validationDictionary = validationDictionary;
            _repository = repository;
        }

        public bool ValidateContact(Contact contactToValidate)
        {
            if (contactToValidate.FirstName.Trim().Length == 0)
                _validationDictionary.AddError("FirstName", "First name is required.");
            if (contactToValidate.LastName.Trim().Length == 0)
                _validationDictionary.AddError("LastName", "Last name is required.");
            if (contactToValidate.Phone.Length > 0 &&
                !Regex.IsMatch(contactToValidate.Phone, @"((\d{3}\ )?|(\d{3}-))?\d{3}-\d{4}"))
                _validationDictionary.AddError("Phone", "Invalid phone number.");
            if (contactToValidate.Email.Length > 0 &&
                !Regex.IsMatch(contactToValidate.Email, @"^[\w-\.]+" + @"([\w-]+\.)+[\w-]{2,4}$"))
                _validationDictionary.AddError("Email", "Invalid email address.");
            return _validationDictionary.IsValid;
        }

        #region IContactManagerService Members

        public bool CreateContact(Contact contactToCreate)
        {
            // Validation logic
            if (!ValidateContact(contactToCreate))
                return false;

            // Database logic
            try
```

```

        {
            _repository.CreateContact(contactToCreate);
        }
        catch
        {
            return false;
        }
        return true;
    }

    public bool EditContact(Contact contactToEdit)
    {
        // Validation logic
        if (!ValidateContact(contactToEdit))
            return false;

        // Database logic
        try
        {
            _repository.EditContact(contactToEdit);
        }
        catch
        {
            return false;
        }
        return true;
    }

    public bool DeleteContact(Contact contactToDelete)
    {
        try
        {
            _repository.DeleteContact(contactToDelete);
        }
        catch
        {
            return false;
        }
        return true;
    }

    public Contact GetContact(int id)
    {
        return _repository.GetContact(id);
    }

    public IEnumerable<Contact> ListContacts()
    {
        return _repository.ListContacts();
    }

    #endregion
}
}

```

Notez que le constructeur de la classe `ContactManagerService` prend en paramètre un `ValidationDictionary` (Dictionnaire de données). La couche de service communique avec la couche contrôleur par ce dictionnaire de données. Nous reverrons ce dictionnaire de données en détails dans la suite de cette étape qui aborde le modèle « Décoration »

Notez de plus que la classe `ContactManagerService` implémente l'interface `IContactManagerService` toujours dans l'optique d'avoir une application faiblement couplée. Les autres classes l'application de Gestion de Contacts n'interagit pas directement avec `ContactManagerService` mais avec l'interface `IContactManagerService`.

L'interface `IContactManagerService` est contenue dans le Listing 5.

Listing 5 – Models\IContactManagerService.cs

```
using System.Collections.Generic;

namespace ContactManager.Models
{
    public interface IContactManagerService
    {
        bool CreateContact(Contact contactToCreate);
        bool DeleteContact(Contact contactToDelete);
        bool EditContact(Contact contactToEdit);
        Contact GetContact(int id);
        IEnumerable ListContacts();
    }
}
```

Les modifications pour le contrôleur `Contact` est contenue dans le Listing 6. Notez que le contrôleur n'interagit plus avec la classe `ContactManager` « Repository » mais bien avec l'interface `IContactManagerService`. Chaque couche est maintenant bien isolée.

Listing 6 – Controllers>ContactController.cs

```
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class ContactController : Controller
    {
        private IContactManagerService _service;

        public ContactController()
        {
            _service = new ContactManagerService(new
            ModelStateWrapper(this.ModelState));
        }

        public ContactController(IContactManagerService service)
        {
            _service = service;
        }
    }
}
```

```

    public ActionResult Index()
    {
        return View(_service.ListContacts());
    }

    public ActionResult Create()
    {
        return View();
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Create([Bind(Exclude = "Id")] Contact
contactToCreate)
    {
        if (_service.CreateContact(contactToCreate))
            return RedirectToAction("Index");
        return View();
    }

    public ActionResult Edit(int id)
    {
        return View(_service.GetContact(id));
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(Contact contactToEdit)
    {
        if (_service.EditContact(contactToEdit))
            return RedirectToAction("Index");
        return View();
    }

    public ActionResult Delete(int id)
    {
        return View(_service.GetContact(id));
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Delete(Contact contactToDelete)
    {
        if (_service.DeleteContact(contactToDelete))
            return RedirectToAction("Index");
        return View();
    }
}
}

```

L'application suit désormais le modèle SRP (Single Responsibility Principe). Le contrôleur du Listing 6 n'a désormais plus que le rôle de contrôle du flux de l'application. La couche de validation lui à été enlevée et déportée dans une couche de service. Toute la couche d'accès aux données à été de la même manière déportée dans la couche « Repository ».

Utilisation du modèle “Décoration”

Nous allons maintenant complètement découper la couche de service et la couche contrôleur. Logiquement, vous devriez être capable de compiler la couche de service dans une assembly séparée de la couche contrôleur sans avoir besoin d’y faire référence dans l’application MVC.

Cependant, la couche de service a besoin de passer les messages d’erreurs de validation à la couche contrôleur. Comment la couche de service va-t-elle communiquer les messages d’erreurs de validation sans coupler le contrôleur et la couche de service ? Pour cela, vous allez tirer avantage du modèle de programmation appelé « Modèle de décoration »

Un contrôleur utilise ModelStateDictionary appelé ModelState pour représenter les erreurs de validation. Nous sommes alors tentés de passer des « ModelState » entre la couche contrôleur et la couche de service. Cependant l’utilisation de ModelState dans la couche de service la rend dépendante aux fonctionnalités ASP.NET MVC. Ce n’est donc pas une bonne idée car cette couche de service pourrait être utilisée demain par une application WPF par exemple. Nous ne voulons pas dans notre cas rendre la couche de service dépendante d’ASP.NET MVC à travers l’utilisation de la classe ModelStateDictionary.

Le modèle de Décoration encapsule une classe existante dans une nouvelle classe afin d’implémenter une interface. Dans l’application de Gestion de Contacts, nous allons ajouter la classe ModelStateWrapper contenue dans le listing 7. La classe ModelStateWrapper implémente l’interface du Listing 8.

Listing 7 – Models\Validation\ModelStateWrapper.cs

```
using System.Web.Mvc;

namespace ContactManager.Models.Validation
{
    public class ModelStateWrapper : IValidationDictionary
    {
        private ModelStateDictionary _ModelState;

        public ModelStateWrapper(ModelStateDictionary ModelState)
        {
            _ModelState = ModelState;
        }

        public void AddError(string key, string ErrorMessage)
        {
            _ModelState.AddModelError(key, ErrorMessage);
        }

        public bool IsValid
        {
            get { return _ModelState.IsValid; }
        }
    }
}
```

Listing 8 – Models\Validation\IValidationDictionary.cs

```
namespace ContactManager.Models.Validation
{
    public interface IValidationDictionary
    {
        void AddError(string key, string errorMessage);
        bool IsValid {get;}
    }
}
```

Si vous regardez en arrière dans le Listing 5, vous pourrez voir que la couche de service ContactManager utilise exclusivement l'interface IValidationDictionary. La couche ContactManagerService n'est donc pas dépendante de la classe ModelStateDictionary. Quand le Contact contrôleur crée la couche de Service, le contrôleur encapsule ModelState comme ceci :

```
_service = new ContactManagerService(new ModelStateWrapper(this.ModelState));
```

Conclusion

Dans cette étape nous n'avons pas ajouté de fonctionnalités à l'application de Gestion de Contacts. Le but de cette étape était bien de faire du "refactoring" (refonte) sur l'application pour la rendre plus facilement modifiable et maintenable.

Dans un premier temps nous avons implémenté le modèle de développement "Repository" et migré ainsi le code d'accès aux données dans une classe séparée.

Ensuite nous avons isolé la partie validation et logique dans une couche de service. La couche contrôleur interagit avec la couche de service et la couche de service interagissait avec la couche « Repository »

Par ailleurs, quand vous avez créé la couche de service, vous avez tiré avantage du modèle de Décoration pour isoler ModelState de la couche de service. Pour se faire, vous avez manipulé l'interface IValidationDictionary plutôt que ModelState.

Finalement, vous avez tiré avantage du modèle de programmation appelé "Injection de dépendance". Ce modèle fait manipuler des interfaces par le développeur plutôt que des classes. L'implémentation du mode d'injection de dépendance rend aussi l'application plus testable. Dans la prochaine étape, vous allez ajouter des tests unitaires à votre application.

Etape 5 – Créer des tests unitaires

Dans cette étape

Dans l'étape précédente de l'application de gestion de contacts, nous avons repris le code pour le rendre faiblement couplé. Nous avons séparé l'application en plusieurs morceaux : le contrôleur, le service et la couche d'accès aux données. Chaque couche interagit avec les autres à travers des interfaces.

Ces modifications ont permis de rendre l'application plus maintenable et mieux modifiable. Par exemple, si l'on a besoin de changer la façon d'accéder aux données, il suffit de changer la couche d'accès associée (Repository Layer) et ce, sans toucher ni au contrôleur ni au service. En rendant l'application de gestion de contacts faiblement couplée, nous rendons l'application plus agile au changement.

Mais, que se passe-t-il lorsque l'on voudra ajouter une fonctionnalité à l'application de gestion de contacts ? Ou lorsque l'on souhaitera corriger un bug ? A chaque modification, on risque de générer un nouveau bug.

Par exemple si un jour le chef de projet demande de rajouter une fonctionnalité comme la notion de groupes de contacts pour que les utilisateurs puissent organiser leurs contacts et les associer à des groupes tels que « Amis », « Travail », ou autre.

Afin d'implémenter cette nouvelle fonctionnalité, il faudra modifier les trois couches de l'application de gestion de contacts. Ajouter une nouvelle fonctionnalité dans le contrôleur, à la couche de service et bien sûr à la couche d'accès aux données. Dès que le code commencera à être modifié, les fonctionnalités qui fonctionnaient avant risqueront d'être touchées et cassées.

Le fait d'avoir repris le code pour lui donner une architecture en couche est une bonne chose car cela permet de changer une couche sans toucher au reste de l'application. Cependant, pour rendre le code plus facile à maintenir et à modifier dans une couche donnée, il va falloir créer des tests unitaires.

Il faut donc utiliser les tests unitaires pour tester une petite partie de code. Les tests unitaires doivent être plus petits qu'une couche complète d'une application. Par exemple, on peut utiliser les tests unitaires pour tester une méthode. On pourrait prendre de manière unitaire `CreateContact()` de la classe `ContactManagerService`.

Les tests unitaires pour une application doivent être autonomes. Quelques soient les modifications apportées dans l'application, vous devez pouvoir lancer les tests et vérifier si les modifications n'entraînent pas de problèmes dans les fonctionnalités existantes. Les tests rendent votre code plus robuste aux modifications.

Dans cette étape, nous allons ajouter des tests à l'application de gestion de contacts. C'est grâce à ces tests que, dans la prochaine étape, nous allons pouvoir ajouter la fonctionnalité de groupes de contacts sans crainte de casser des fonctionnalités existantes.

Il existe de nombreux frameworks de tests tel que NUnit, xUnit.net and MbUnit. Dans ce tutorial, nous allons utiliser le frameworks de tests inclus dans Visual Studio. Cependant, vous pouvez facilement utiliser d'autre frameworks de tests.

Qu'apportent les tests unitaires ?

Dans un monde parfait, tout notre code devrait être couvert par les tests. Chaque modification d'une ligne de code dans votre application devrait pouvoir être testée instantanément par l'exécution du test unitaire.

Cependant, nous ne vivons pas dans un monde parfait. En pratique, quand on écrit des tests, on se focalise à tester la couche métier. En général, on n'écrit pas de tests pour la couche d'accès aux données, ni pour la couche de présentation.

Pour être utiles, les tests doivent pouvoir être exécutés rapidement. Vous pouvez sans problème créer des centaines (voir des milliers) de tests pour une application. En effet, si l'exécution d'un test unitaire est trop longue, il ne sera pas exécuté. En d'autres termes, les tests unitaires trop longs sont inutiles au développement de tous les jours.

C'est pour cette raison que les tests unitaires ne doivent pas interagir avec la base de données. Faire tourner des centaines de tests unitaires sur une base de données peut être très long. Par contre, abstraire la base et écrire du code qui interagit avec cette abstraction est beaucoup plus intéressant.

C'est exactement pour cette même raison qu'on n'écrit pas de tests unitaires sur les vues. En effet, pour tester une vue, il est nécessaire d'avoir un serveur Web. Cela peut être aussi très lent à mettre en place et n'est donc pas recommandé pour les tests unitaires.

Si votre vue contient de la logique complexe, vous devez déplacer cette logique dans des méthodes de type Helper. Vous pourrez alors écrire des tests unitaires sur ces méthodes Helper sans avoir à monter un serveur Web.

Alors que l'écriture des tests unitaires sur la couche d'accès aux données ou sur les vues n'est pas une bonne idée, ces tests peuvent être d'une grande importance pour évaluer votre application à des fins fonctionnelles ou d'intégration.

Note : ASP.NET MVC repose sur le moteur ASP.NET WebForms. Alors que le moteur des WebForms est dépendant du serveur Web, d'autres moteurs de vues MVC pourrait ne pas l'être.

Utilisation de Mock Object Framework

Quand vous écrivez des tests unitaires, vous aurez presque toujours besoin de tirer avantage de Mock Object Framework. Un Mock Object Framework permet la création de méthodes et de « fausses » classes dans votre application.

Par exemple, vous pouvez utiliser un Mock Object Framework pour générer une version fictive de la couche d'accès aux données. Vous pouvez ensuite utiliser ces classes fictives à la place des vraies classes dans vos tests unitaires. Utiliser la couche d'accès aux données fictives vous permettra d'éviter d'exécuter la vraie couche d'accès pour l'exécution des tests.

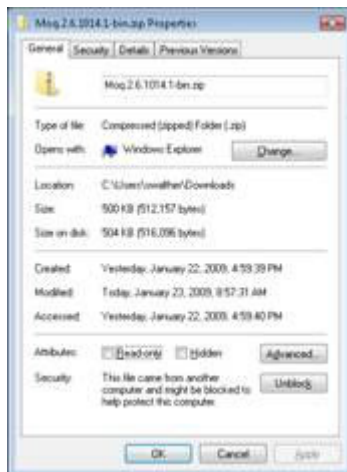
Visual Studio n'intègre pas de Mock Object Framework. Cependant, il existe de nombreux frameworks payants ou open source pour le framework .NET :

1. Moq – Ce framework est sous licence BSD. Vous pouvez le télécharger ici : <http://code.google.com/p/moq/>.
2. Rhino Mocks – Ce framework est sous licence BSD. Vous pouvez le télécharger ici : <http://ayende.com/projects/rhino-mocks.aspx>.
3. Typemock Isolator – Ce framework est payant. Vous pouvez télécharger une version d'évaluation ici : <http://www.typemock.com/>.

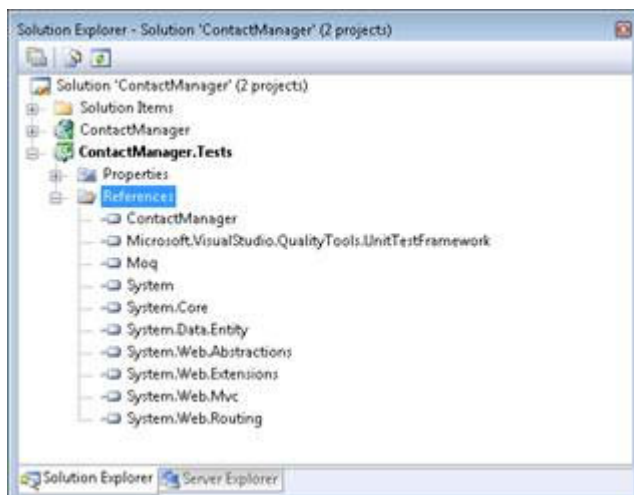
Dans ce tutoriel, nous utiliserons Moq. Cependant, vous pouvez facilement utiliser Rhino Mocks ou Typemock Isolator sur le projet de gestion de contacts.

Avant d'utiliser Moq, vous devez suivre ces étapes :

1. Débloquez le fichier Zip



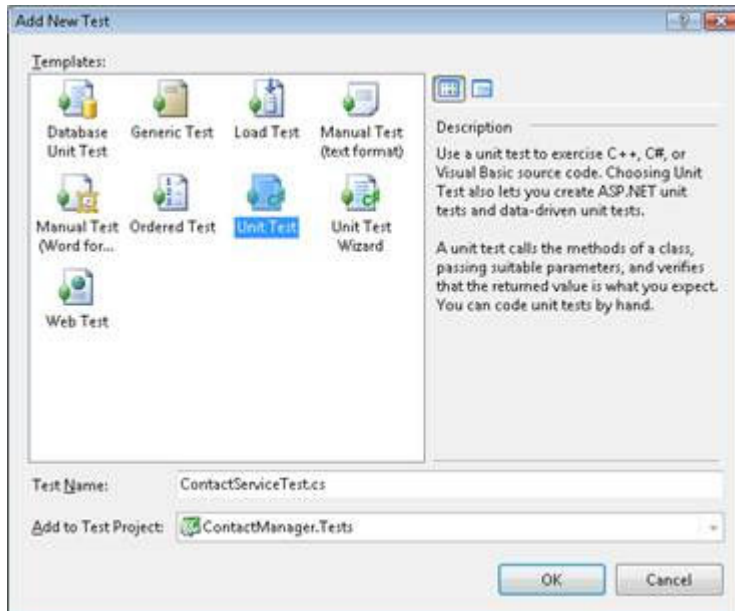
2. Décompressez le fichier téléchargé.
3. Ajoutez une référence à l'assembly Moq en faisant un bouton droit sur « References » sur le projet « ContactManager.Tests ». Sélectionnez « Add Reference », parcourez le répertoire qui vient d'être décompressé et sélectionnez l'assembly Moq.dll.
4. Après cette étape vous devriez obtenir ceci



Création des tests unitaires pour la couche de service

Nous allons maintenant créer un jeu de tests unitaires pour la couche de service de l'application de gestion de contacts. Nous utiliserons ces tests afin de vérifier notre logique de validation.

1. Créez un nouveau répertoire « Models » dans le projet « ContactManager.Tests ». Puis cliquez droit sur ce répertoire et sélectionnez « Add, New Test ».



2. Sélectionnez le template « Unit Test » et donnez lui comme nom : « ContactManagerServiceTest.cs » puis cliquez « Ok »

En général, on fait correspondre le projet de tests avec la structure du projet ASP.NET MVC. Par exemple, on mettra le test du contrôleur dans un répertoire « Controller », le test du modèle dans un répertoire « Model » etc...

Pour commencer, on va tester la méthode CreateContact() de la classe ContactManagerService. Nous allons donc créer ces 5 tests :

3. CreateContact() - Test si CreateContact() retourne la valeur vrai quand un contact « valide » est passé à la méthode.
4. CreateContactRequiredFirstName() – Test si un message d’erreur est généré quand on envoi un Contact sans la propriété « FirstName » remplie (prénom) à la méthode CreateContact()
5. CreateContactRequiredLastName() – Test si un message d’erreur est généré quand on envoi un Contact sans la propriété « LastName » remplie (nom de famille) à la méthode CreateContact()
6. CreateContactInvalidPhone() – Test si un message d’erreur est généré quand on envoi un Contact avec un numéro de téléphone non valide à la méthode CreateContact()
7. CreateContactInvalidEmail() – Test si un message d’erreur est généré quand on envoi un Contact avec une adresse email non valide à la méthode CreateContact()

Le premier test vérifie qu'un contact valide ne génère pas de messages d'erreur. Les tests suivant vérifient chaque règle de saisie.

Code des tests dans le Source 1.

Listing 1 – Models\ContactManagerServiceTest.cs

```
using System.Web.Mvc;
using ContactManager.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

namespace ContactManager.Tests.Models
{
    [TestClass]
    public class ContactManagerServiceTest
    {
        private Mock<IContactManagerRepository> _mockRepository;
        private ModelStateDictionary _modelState;
        private IContactManagerService _service;

        [TestInitialize]
        public void Initialize()
        {
            _mockRepository = new Mock<IContactManagerRepository>();
            _modelState = new ModelStateDictionary();
            _service = new ContactManagerService(new
            ModelStateWrapper(_modelState), _mockRepository.Object);
        }

        [TestMethod]
        public void CreateContact()
        {
            // Arrange
            var contact = Contact.CreateContact(-1, "Stephen", "Walther",
            "555-5555", "steve@somewhere.com");

            // Act
            var result = _service.CreateContact(contact);

            // Assert
            Assert.IsTrue(result);
        }

        [TestMethod]
        public void CreateContactRequiredFirstName()
        {
            // Arrange
            var contact = Contact.CreateContact(-1, string.Empty, "Walther",
            "555-5555", "steve@somewhere.com");

            // Act
            var result = _service.CreateContact(contact);

            // Assert
            Assert.IsFalse(result);
        }
    }
}
```

```

        var error = _modelState["FirstName"].Errors[0];
        Assert.AreEqual("First name is required.", error.ErrorMessage);
    }

    [TestMethod]
    public void CreateContactRequiredLastName()
    {
        // Arrange
        var contact = Contact.CreateContact(-1, "Stephen", string.Empty,
"555-5555", "steve@somewhere.com");

        // Act
        var result = _service.CreateContact(contact);

        // Assert
        Assert.IsFalse(result);
        var error = _modelState["LastName"].Errors[0];
        Assert.AreEqual("Last name is required.", error.ErrorMessage);
    }

    [TestMethod]
    public void CreateContactInvalidPhone()
    {
        // Arrange
        var contact = Contact.CreateContact(-1, "Stephen", "Walther",
"apple", "steve@somewhere.com");

        // Act
        var result = _service.CreateContact(contact);

        // Assert
        Assert.IsFalse(result);
        var error = _modelState["Phone"].Errors[0];
        Assert.AreEqual("Invalid phone number.", error.ErrorMessage);
    }

    [TestMethod]
    public void CreateContactInvalidEmail()
    {
        // Arrange
        var contact = Contact.CreateContact(-1, "Stephen", "Walther",
"555-5555", "apple");

        // Act
        var result = _service.CreateContact(contact);

        // Assert
        Assert.IsFalse(result);
        var error = _modelState["Email"].Errors[0];
        Assert.AreEqual("Invalid email address.", error.ErrorMessage);
    }
}
}

```

Comme on utilise la classe `Contact` dans ce code source, on doit ajouter une référence au Framework « Microsoft Entity ». Ajoutez donc une référence à l'assembly `System.Data.Entity`.

Le code du Listing 1 contient une méthode « `Initialize()` » auquel on a ajouté un attribut `[TestInitialize]`. Cette méthode sera donc appelée automatiquement avant chaque test unitaire (ici elle sera appelée 5 fois, 1 fois par test unitaire). La méthode `Initialize()` crée une couche de simulation de l'accès aux données grâce à cette ligne de code :

```
_mockRepository = new Mock<IContactManagerRepository>();
```

Cette ligne de code utilise le Framework « Moq » pour générer un accès aux données virtuel et ceci depuis l'interface `IContactManagerRepository`. La « fausse » couche d'accès aux données sera utilisée plutôt que celle de type « `EntityContactManagerRepository` » qui aurait besoin de faire un accès physique à la base chaque fois. La « fausse » couche d'accès aux données implémente l'interface `IContactManagerRepository`, mais les méthodes ne font rien.

Quand on utilise le framework « Moq », il y a une distinction à faire entre `_mockRepository` et `_mockRepository.Object`. Le premier fait référence à la classe `Mock<IContactManagerRepository>` qui contient les méthodes pour spécifier le comportement de la « fausse » classe. Tant dis que l'autre implémente simplement l'interface `IContactManagerRepository`.

La « fausse » couche d'accès aux données utilise la méthode `Initialize()` quand une instance de la classe `ContactManagerService` est créé. Tous les tests unitaires utilisent cette instance de la classe `ContactManagerService`.

Le code du Listing 1 contient 5 méthodes. Chacune correspond à un test unitaire. Chaque méthode est décorée de l'attribut `[TestMethod]`. Quand les tests unitaires sont lancés, chaque méthode décorée de cet attribut est appelée. En d'autres termes, chaque méthode qui est décorée de cet attribut `[TestMethod]` devient un test unitaire.

Le premier test unitaire, nommé `CreateContact()`, vérifie que l'appel à `CreateContact()` retourne la valeur « vrai » dans le cas où on lui passe un contact « valide ».

Les autres tests vérifient que la méthode `CreateContact()` renvoie faux quand on lui passe un `Contact` qui n'est pas valide et qu'une exception est bien soulevée. Par exemple, le test `CreateContactRequiredFirstName()` s'occupe de tester la création de la classe `Contact` avec une chaîne vide dans la propriété `FirstName`. Ensuite, il appelle la méthode `CreateContact()` avec ce `Contact` non valide. Finalement, le test vérifie bien que la méthode `CreateContact()` renvoie « faux » et que le modèle lève une exception en conséquence avec comme message « First Name is required ».

Vous pouvez lancer le test du code Listing 1 en allant dans le menu « Test / Run / All Tests in Solution » ou (CTRL+R, A). Le résultat des tests doit s'afficher comme ceci (Figure 4):

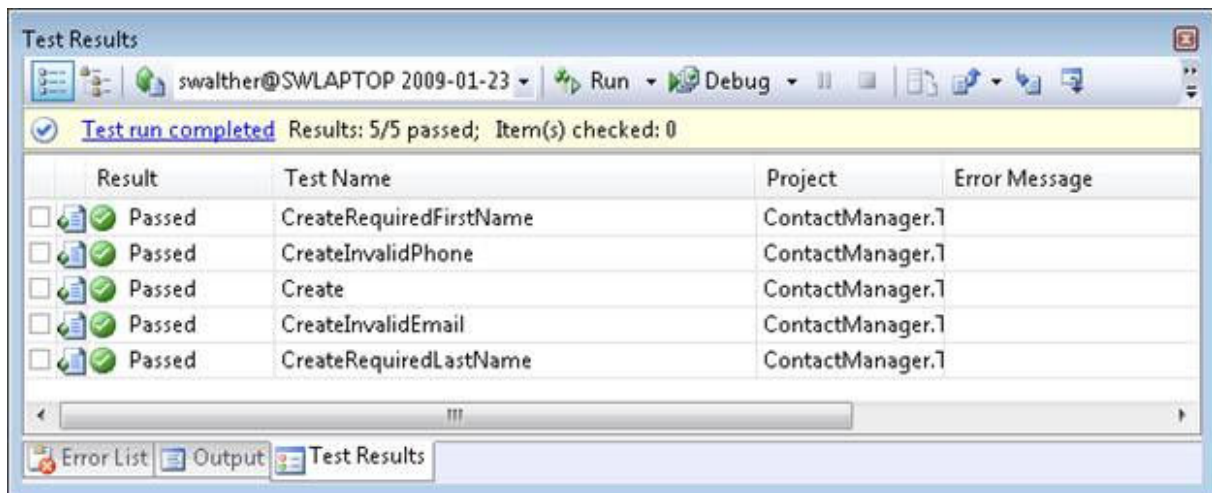


Figure 04: Résultat des tests.

Création des tests unitaires pour les contrôleurs.

Une application ASP.NET MVC contrôle les enchainements des interactions utilisateur. En testant un contrôleur, vous voulez savoir si ce dernier retourne la bonne action ou les bonnes données vis-à-vis du comportement attendu. Vous voulez aussi peut-être vérifier si le contrôleur interagit bien avec les classes du modèle de la manière attendue.

Par exemple, le Listing 2 contient deux tests unitaires sur le contrôleur Contact et sur la méthode Create(). Le premier test vérifie que lorsque l'on passe un Contact « valide » vers la méthode Create(), cette méthode redirige bien sur l'action « Index ». En d'autres termes, quand on lui passe un Contact « valide », la méthode Create() doit retourner une action « Index » de type RedirectToRouteResult.

Il ne faut pas tester la couche de service de l'application de gestion de contacts quand on test la couche contrôleur. Par conséquent, on va utiliser une « fausse » couche de service. On mettra ceci dans la méthode Initialize().

```
_service = new Mock();
```

Dans le test unitaire CreateValidContact(), on appellera la « fausse » couche de service CreateContact() de cette manière :

```
_service.Expect(s => s.CreateContact(contact)).Returns(true);
```

Cette ligne de code permet de toujours renvoyer « vrai » quand la méthode CreateContact() est appelée. En créant cette fausse couche de service, on va pouvoir tester le contrôleur sans avoir besoin d'exécuter la couche de service.

Le second test unitaire vérifie que l'action Create() retourne à nouveau la vue de création de contact quand on lui passe un contact « non valide ». La couche de service doit retourner « faux » à l'appel de la méthode CreateContact().

```
_service.Expect(s => s.CreateContact(contact)).Returns(false);
```

Si la méthode s'exécute comme prévue, elle doit retourner une vue de type « Create » même quand la couche de service renvoie faux. Dans ce cas, le contrôleur doit afficher les messages d'erreur de saisie dans la vue « Create » et l'utilisateur pourra donc corriger les erreurs qu'il a commises.

Si vous avez pour but de créer des tests unitaires sur les contrôleurs, vous devez donc renvoyer explicitement le nom de la vue. Par exemple, vous ne devez pas retourner ceci :

```
return View();
```

Mais plutôt ceci :

```
return View("Create");
```

En renvoyant le nom de la vue de façon explicite, la propriété `ViewResult.ViewName` sera renseignée.

Code Listing 2 – Controllers\ContactControllerTest.cs

```
using System.Web.Mvc;
using ContactManager.Controllers;
using ContactManager.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

namespace ContactManager.Tests.Controllers
{
    [TestClass]
    public class ContactControllerTest
    {
        private Mock<IContactManagerService> _service;

        [TestInitialize]
        public void Initialize()
        {
            _service = new Mock<IContactManagerService>();
        }

        [TestMethod]
        public void CreateValidContact()
        {
            // Arrange
            var contact = new Contact();
            _service.Expect(s => s.CreateContact(contact)).Returns(true);
            var controller = new ContactController(_service.Object);

            // Act
            var result = (RedirectToRouteResult)controller.Create(contact);

            // Assert
            Assert.AreEqual("Index", result.RouteValues["action"]);
        }
    }
}
```



```

[TestMethod]
public void CreateInvalidContact()
{
    // Arrange
    var contact = new Contact();
    _service.Expect(s => s.CreateContact(contact)).Returns(false);
    var controller = new ContactController(_service.Object);

    // Act
    var result = (ViewResult)controller.Create(contact);

    // Assert
    Assert.AreEqual("Create", result.ViewName);
}
}
}

```

Conclusion

Dans cette étape, nous avons créé des tests unitaires sur l'application de gestion de contacts. Les tests unitaires peuvent être lancés de façon autonome à tout moment pour vérifier le comportement de l'application. Les tests unitaires permettent de couvrir les scénarios d'évolution ou de modification sans risque.

Nous avons créé deux types de tests. Le premier a testé la logique de validation de l'application en créant des tests unitaires sur la couche de service. Ensuite, nous avons testé la logique de navigation en créant des tests sur la couche contrôleur. Les tests sur les couches de service et contrôleur sont autonomes grâce à l'utilisation d'un Framework de « Moq permettant de simuler les couches de dépendance.

Dans la prochaine étape, nous allons ajouter de nouvelles fonctionnalités à l'application de gestion de contacts afin d'introduire la notion de groupes de contacts. Nous ajouterons cette fonctionnalité en utilisant le pattern « Test-Driven development » ou programmation pilotée par les tests. Le développement sera donc en permanence contrôlé par les tests unitaires.

Etape 6 – Utiliser la programmation pilotée par les tests

Dans cette étape

Dans l'étape précédente, nous avons créé des tests unitaires afin de construire un cadre sécurisant à l'écriture de notre code. La motivation première de ces tests était de nous permettre de rendre notre code plus souple au changement. Avec les tests unitaires en place, nous pouvons effectuer sereinement n'importe quel changement à notre code et observer immédiatement si nous avons cassé une fonctionnalité existante.

Dans cette étape, nous allons utiliser les tests unitaires dans un but entièrement différent. Ici, nous allons en faire usage au sein d'une philosophie d'architecture applicative appelée « test-driven development » ou programmation pilotée par les tests. Lorsque vous utilisez cette approche, vous écrivez d'abord vos tests et ce n'est qu'après que vous vous lancez dans l'écriture du code fonctionnel qui sera contraint par ces tests.

Plus précisément, avec cette méthode, il y a 3 étapes que vous devez suivre lors de l'écriture de votre code (Rouge/Vert/Refactoring) :

1. Ecrire un test unitaire qui échoue (Rouge)
2. Ecrire un code qui passe le test unitaire avec succès (Vert)
3. Revoir l'architecture de votre code (Refactoring)

Ainsi, première étape, vous devez écrire les tests unitaires. Un test unitaire doit exprimer votre intention sur la manière dont votre code doit se comporter. Lorsque vous écrivez le test unitaire pour la première fois, il doit échouer. Il échoue car vous n'avez pas encore écrit le moindre code fonctionnel satisfaisant les contraintes du test.

Ensuite, vous écrivez le minimum de code possible afin de passer le test unitaire. Le but du jeu étant d'écrire du code de la manière la plus rapide et simple qui soit. Vous ne devez pas à ce stage perdre du temps à penser à l'architecture de votre application. Au lieu de cela, vous devez concentrer vos efforts sur l'écriture minimaliste d'un code satisfaisant l'intention exprimé par le test unitaire.

Finalement, après avoir écrit suffisamment de code, vous pouvez revenir en arrière pour revoir l'architecture globale de votre application. Dans cette étape, vous réécrivez votre code en tirant bénéfice de modèles de développement logiciel – comme le Repository utilisé précédemment – afin de rendre votre code plus facile à maintenir. Vous pouvez réécrire votre code sans crainte ici car il est désormais couvert par les tests unitaires.

Il y a plusieurs avantages à utiliser une programmation pilotée par les tests. Tout d'abord, elle vous impose de vous concentrer uniquement sur le code qui a vraiment besoin d'être écrit. Comme vous êtes monopolisé à écrire uniquement le minimum de code nécessaire au passage du test, cela vous évite de vous disperser en écrivant une quantité importante de code dont vous ne ferez jamais usage.

Par ailleurs, une méthodologie de type « les tests d'abord » vous oblige à développer en ayant en tête la manière dont votre code sera utilisé. En autre termes, en suivant cette méthodologie, vous écrivez de manière constante vos tests depuis une approche utilisateur. Ainsi, une programmation pilotée par les tests peut aboutir en un jeu d'APIs plus propre et facile à comprendre.

Enfin, cette approche apporte l'écriture des tests unitaires comme quelque chose de normal dans le cycle de vie de la conception de votre application. En effet, lorsque les dates de livraison des projets arrivent, les tests représentent typiquement le genre de chose qui est exclus en premier afin de gagner du temps. En pratiquant cette méthode, vous serez donc à priori plus respectueux de l'écriture des tests unitaires tout simplement parce qu'ils feront partis du noyau central de la conception de votre application.

Pour en savoir davantage sur cette méthodologie, nous vous invitons à lire le livre (en Anglais) de Michael Feathers nommé « Working Effectively with Legacy Code ».

Dans cette étape, nous allons donc ajouter une nouvelle fonctionnalité à notre application à travers le support des groupes de contacts. Vous pouvez utiliser les groupes pour organiser vos contacts en catégories comme les groupes Professionnels ou Amis par exemple. Nous allons ajouter cette fonctionnalité en suivant bien évidemment la programmation pilotée par les tests. Nous allons ainsi d'abord écrire nos tests unitaires avant de pouvoir écrire le code fonctionnel fonctionnant au travers de ces tests.

Quelles parties peuvent être testées ?

Comme indiqué dans l'étape précédente, vous n'écrivez pas de tests unitaires pour la logique d'accès aux données ou celle de vos vues. Vous n'écrivez pas de tests unitaires pour la logique d'accès aux données car l'accès à une base de données est une opération relativement lente. Vous n'écrivez pas de tests unitaires pour tester la logique des vues car l'accès à une vue impose de lever une instance d'un serveur web qui est également une opération relativement lente. Or, vous ne devriez pas écrire de tests unitaires à moins que ces derniers puissent être exécutés en boucle de manière extrêmement rapide.

Comme la programmation pilotée par les tests repose, comme son nom l'indique, sur les tests unitaires, nous allons d'abord nous concentrer sur l'écriture de la logique du contrôleur et de la logique métier. Nous allons éviter de toucher à la base de données ou aux vues. Nous reviendrons sur ces 2 composants qu'à la fin de ce tutoriel. Nous allons commencer par ce qui peut être testé.

Créer les scénarios d'usage utilisateur

Lorsque l'on met en pratique la programmation pilotée par les tests, on commence forcément par écrire un test. Cela lève alors immédiatement la question suivante : quels tests devons-nous écrire en premier ? Pour répondre à cette question, vous devez écrire une série de scénarios d'usage utilisateur ou user stories en Anglais.

Ce scénario est une description très brève (souvent d'une seule phrase) précisant un besoin logiciel. Cela doit être une description non technique d'un besoin selon le point de vue d'un utilisateur.

Voici la série de scénarios d'usage qui décrivent les fonctionnalités requises pour supporter la notion de groupe de contacts :

1. L'utilisateur peut voir une liste de groupes de contacts
2. L'utilisateur peut créer un nouveau groupe de contacts
3. L'utilisateur peut supprimer un groupe de contacts existant
4. L'utilisateur peut sélectionner un groupe lorsqu'il crée un nouveau contact
5. L'utilisateur peut sélectionner un groupe lorsqu'il édite un contact existant
6. La liste des groupes de contacts est affichée dans la vue Index
7. Lorsqu'un utilisateur clique sur un groupe, la liste de contacts associée est affichée

Vous noterez que cette liste de scénarios est complètement compréhensible par un client. Nous n'y faisons pas mention de détails d'implémentation technique.

Pendant la conception de votre application, vous pouvez être amenés à affiner ces scénarios. Vous pouvez par exemple séparer un scénario d'usage en plusieurs. Par exemple, on pourrait considérer que la création d'un nouveau groupe de contacts soit soumise à une forme de validation. Ainsi, la demande de création d'un groupe avec un nom vide devrait retourner un message d'erreur de validation.

Après avoir créé cette liste de scénarios, vous êtes prêts à créer votre premier test unitaire. Nous allons commencer par créer un test unitaire permettant de voir la liste des groupes de contacts.

Lister les groupes de contacts

Notre premier scénario nous indique qu'un utilisateur doit pouvoir voir une liste de groupes de contacts. Il nous faut exprimer ce scénario à travers un test.

Créez un nouveau test unitaire en cliquant-droit sur le répertoire « Controllers » dans le projet « ContactManager.Tests » de votre solution puis choisissez Add, New Test, et le type Unit Test (voir Figure 1). Nommez ce test unitaire GroupControllerTest.cs et cliquez sur le bouton OK.

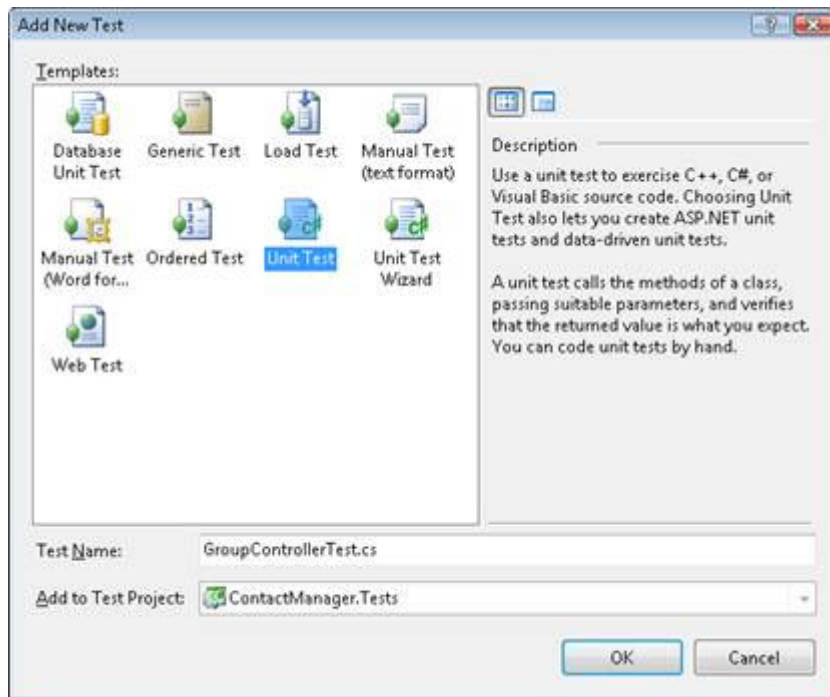


Figure 01: L'ajout du test unitaire GroupControllerTest

Notre premier test unitaire est présent dans le Listing 1. Ce test vérifie que la méthode Index() du contrôleur Group retourne bien un ensemble de groupes. En effet, ce test vérifie qu'une collection est retournée vers les données de la vue.

Listing 1 – Controllers\GroupControllerTest.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Tests.Controllers
{
    [TestClass]
    public class GroupControllerTest
    {
        [TestMethod]
        public void Index()
        {
            // Arrange
            var controller = new GroupController();

            // Act
            var result = (ViewResult)controller.Index();

            // Assert
            Assert.IsInstanceOfType(result.ViewData.Model,
            typeof(IEnumerable));
        }
    }
}
```

```
}  
}
```

Lorsque vous tapez pour la première fois le code du Listing 1 dans Visual Studio, vous aurez pas mal de lignes rouges soulignant le code. En effet, nous n'avons ni créé le contrôleur `GroupController` ni les classes gérant la notion de groupe.

Ainsi, à ce stade, nous ne pouvons même pas compiler notre application et nous ne pouvons ainsi pas non plus exécuter notre premier test unitaire. C'est parfait. Cela compte comme un test qui échoue ! Ainsi, nous avons la permission de commencer à écrire le code fonctionnel. Il nous faut écrire suffisamment de code afin de passer ce premier test.

La classe du contrôleur `Group` du Listing 2 contient le code minimum requis pour passer le test unitaire. L'action `Index()` retourne une liste de groupes codée statiquement (la classe `Group` est définie dans le Listing 3).

Listing 2 – Controllers\GroupController.cs

```
using System.Collections.Generic;  
using System.Web.Mvc;  
using ContactManager.Models;  
  
namespace ContactManager.Controllers  
{  
    public class GroupController : Controller  
    {  
        public ActionResult Index()  
        {  
            var groups = new List();  
            return View(groups);  
        }  
    }  
}
```

Listing 3 – Models\Group.cs

```
namespace ContactManager.Models  
{  
    public class Group  
    {  
    }  
}
```

Après avoir ajouté le contrôleur `Group` et la classe pour gérer les groupes à notre projet, notre premier test unitaire s'exécute avec succès (voir Figure 2). Nous avons ainsi fait le travail minimum pour passer le test. Il faut fêter ça !

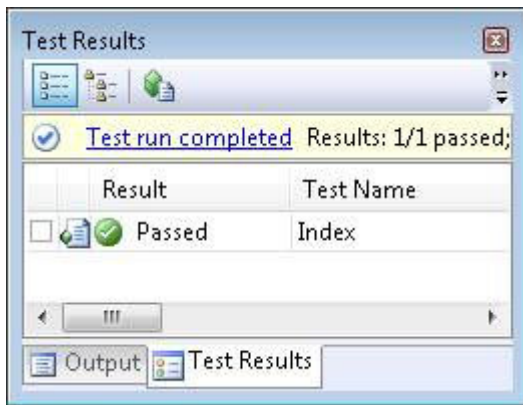


Figure 02: Félicitations!

Création des groupes de contacts

Nous pouvons maintenant passer au 2ème scénario. Nous devons être capables de créer de nouveaux groupes de contacts. Nous devons donc à nouveau exprimer cette intention à travers un test.

Le test dans le Listing 4 vérifie que l'appel à la méthode `Create()` avec un nouveau groupe en paramètre ajoute bien le groupe en question à la liste des groupes retournés par la méthode `Index()`. En d'autres termes, si je crée un nouveau groupe alors je devrais pouvoir le retrouver immédiatement dans la liste des groupes retournée par la méthode `Index()`.

Listing 4 – Controllers\GroupControllerTest.cs

```
[TestMethod]
public void Create()
{
    // Arrange
    var controller = new GroupController();

    // Act
    var groupToCreate = new Group();
    controller.Create(groupToCreate);

    // Assert
    var result = (ViewResult)controller.Index();
    var groups = (IEnumerable<Group>)result.ViewData.Model;
    CollectionAssert.Contains(groups.ToList(), groupToCreate);
}
```

Le test du Listing 4 appelle la méthode `Create()` du contrôleur `Group` avec un nouvel objet de type `Group` en paramètre. Ensuite le test vérifie que l'appel à la méthode `Index()` du contrôleur `Group` retourne bien le groupe fraîchement créé dans les données de la vue.

Le contrôleur `GroupController` contenu dans le Listing 5 présente le minimum de changement requis pour passer le nouveau test.

Listing 5 – Controllers\GroupController.cs

```
using System.Collections.Generic;
using System.Web.Mvc;
using ContactManager.Models;
using System.Collections;

namespace ContactManager.Controllers
{
    public class GroupController : Controller
    {
        private IList<Group> _groups = new List<Group>();

        public ActionResult Index()
        {
            return View(_groups);
        }

        public ActionResult Create(Group groupToCreate)
        {
            _groups.Add(groupToCreate);
            return RedirectToAction("Index");
        }
    }
}
```

Le contrôleur du Listing 5 propose une nouvelle action `Create()`. Cette action ajoute un nouveau groupe à la collection existante de groupes. Notez également que l'action `Index()` a été modifiée pour retourner le contenu de la collection.

Une fois de plus, nous avons effectué le travail minimum requis pour passer le test unitaire. En effet, après avoir mis en place ces petits changements au contrôleur `Group`, l'ensemble de nos tests unitaires passent bien avec succès.

Ajoutez de la validation

Ce besoin n'a pas été clairement établi par les scénarios d'usage. Cependant, il est raisonnable de penser qu'un groupe doit au moins disposer d'un nom. Sinon, l'organisation de nos contacts en groupe ne serait pas très pratique.

Le Listing 6 contient un nouveau test exprimant cette intention. Ce test vérifie qu'une tentative de création sans avoir précisé un nom au groupe génère bien un message d'erreur de validation dans l'objet `ModelState`.

Listing 6 – Controllers\GroupControllerTest.cs

```
[TestMethod]
public void CreateRequiredName()
{
    // Arrange
    var controller = new GroupController();
```



```

// Act
var groupToCreate = new Group();
groupToCreate.Name = String.Empty;
var result = (ViewResult)controller.Create(groupToCreate);

// Assert
var error = result.ViewData.ModelState["Name"].Errors[0];
Assert.AreEqual("Name is required.", error.ErrorMessage);
}

```

Ainsi, afin de satisfaire ce test, il nous faut ajouter une propriété Name à notre classe Group (voir Listing 7). De plus, il nous faut également ajouter une légère couche de validation au sein de l'action Create() de notre contrôleur Group (voir Listing 8).

Listing 7 – Models\Group.cs

```

namespace ContactManager.Models
{
    public class Group
    {
        public string Name { get; set; }
    }
}

```

Listing 8 – Controllers\GroupController.cs

```

public ActionResult Create(Group groupToCreate)
{
    // Validation logic
    if (groupToCreate.Name.Trim().Length == 0)
    {
        ModelState.AddModelError("Name", "Name is required.");
        return View("Create");
    }

    // Database logic
    _groups.Add(groupToCreate);
    return RedirectToAction("Index");
}

```

Vous noterez que l'action contrôleur Create() contient désormais à la fois la logique de validation et d'accès aux données. Pour l'instant, la base de données utilisée par le contrôleur n'est rien d'autre qu'une collection en mémoire.

Le temps est venu de revoir l'architecture

La 3ème étape dans notre cycle Rouge/Vert/Refactoring est donc la revue de notre architecture. A ce stade, nous devons prendre un peu de recul sur notre code et analyser la manière dont on peut revoir son architecture pour en améliorer la conception. C'est donc à ce niveau que l'on réfléchit correctement sur la meilleure façon d'implémenter les principes et méthodes de conceptions logiciels.

Nous sommes libres de modifier notre code par tous les moyens afin d'en améliorer sa conception. Nous avons en effet une couche de protection à travers nos tests unitaires nous empêchant de caser des fonctionnalités existantes.

Pour l'instant, notre contrôleur est un peu chaotique sous l'angle des bonnes pratiques de conceptions logicielles. Il contient un subtil mélange de code dédié à la validation et à l'accès aux données. Afin d'éviter de violer le SRP (Single Responsibility Principle), il nous faut séparer ces notions dans différentes classes.

Notre contrôleur Group revue pour cela est présenté dans le Listing 9. Le contrôleur a été modifié pour utiliser la couche de service ContactManager. C'est la même couche de service que nous avons utilisé avec le contrôleur Contact.

Le Listing 10 contient les nouvelles méthodes ajoutées à la couche de service ContactManager afin de supporter la validation, le listing et la création de groupes. L'interface IContactManagerService fut donc également mise à jour pour inclure ces mêmes nouvelles méthodes.

Le Listing 11 contient une nouvelle classe FakeContactManagerRepository implémentant l'interface IContactManagerRepository. Contrairement à la classe EntityContactManagerRepository qui implémente également l'interface IContactManagerRepository, notre classe FakeContactManagerRepository ne communique pas avec la base de données. Cette classe utilise plutôt une collection en mémoire comme intermédiaire vers une base de données. Nous utiliserons cette classe dans nos tests unitaires en tant que couche de stockage de simulation.

Listing 9 – Controllers\GroupController.cs

```
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class GroupController : Controller
    {
        private IContactManagerService _service;

        public GroupController()
        {
            _service = new ContactManagerService(new
            ModelStateWrapper(this.ModelState));
        }

        public GroupController(IContactManagerService service)
        {
            _service = service;
        }

        public ActionResult Index()
        {
            return View(_service.ListGroups());
        }

        public ActionResult Create(Group groupToCreate)
```

```

        {
            if (_service.CreateGroup(groupToCreate))
                return RedirectToAction("Index");
            return View("Create");
        }
    }
}

```

Listing 10 – Controllers\ContactManagerService.cs

```

public bool ValidateGroup(Group groupToValidate)
{
    if (groupToValidate.Name.Trim().Length == 0)
        _validationDictionary.AddError("Name", "Name is required.");
    return _validationDictionary.IsValid;
}

public bool CreateGroup(Group groupToCreate)
{
    // Validation logic
    if (!ValidateGroup(groupToCreate))
        return false;

    // Database logic
    try
    {
        _repository.CreateGroup(groupToCreate);
    }
    catch
    {
        return false;
    }
    return true;
}

public IEnumerable<Group> ListGroups()
{
    return _repository.ListGroups();
}

```

Listing 11 – Controllers\FakeContactManagerRepository.cs

```

using System;
using System.Collections.Generic;
using ContactManager.Models;

namespace ContactManager.Tests.Models
{
    public class FakeContactManagerRepository : IContactManagerRepository
    {
        private IList<Group> _groups = new List<Group>();

        #region IContactManagerRepository Members

```

```

        // Group methods

        public Group CreateGroup(Group groupToCreate)
        {
            _groups.Add(groupToCreate);
            return groupToCreate;
        }

        public IEnumerable<Group> ListGroups()
        {
            return _groups;
        }

        // Contact methods

        public Contact CreateContact(Contact contactToCreate)
        {
            throw new NotImplementedException();
        }

        public void DeleteContact(Contact contactToDelete)
        {
            throw new NotImplementedException();
        }

        public Contact EditContact(Contact contactToEdit)
        {
            throw new NotImplementedException();
        }

        public Contact GetContact(int id)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<Contact> ListContacts()
        {
            throw new NotImplementedException();
        }

        #endregion
    }
}

```

Modifier l'interface `IContactManagerRepository` impose d'implémenter les méthodes `CreateGroup()` et `ListGroups()` présentes dans la classe `EntityContactManagerRepository`. La manière la plus rapide et simple de faire cela est via l'ajout de méthodes simplissimes comme :

```

public Group CreateGroup(Group groupToCreate)
{
    throw new NotImplementedException();
}

public IEnumerable<Group> ListGroups()
{
    throw new NotImplementedException();
}

```

```
}
```

Pour terminer, ces changements à l'architecture de notre application nous imposent de faire quelques modifications à nos tests unitaires. Il faut désormais faire usage de la classe `FakeContactManagerRepository` lors de l'exécution de nos tests unitaires. La classe `GroupControllerTest` mise à jour se trouve dans le Listing 12.

Listing 12 – Controllers\GroupControllerTest.cs

```
using System.Collections.Generic;
using System.Web.Mvc;
using ContactManager.Controllers;
using ContactManager.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections;
using System.Linq;
using System;
using ContactManager.Tests.Models;

namespace ContactManager.Tests.Controllers
{
    [TestClass]
    public class GroupControllerTest
    {
        private IContactManagerRepository _repository;
        private ModelStateDictionary _ModelState;
        private IContactManagerService _service;

        [TestInitialize]
        public void Initialize()
        {
            _repository = new FakeContactManagerRepository();
            _ModelState = new ModelStateDictionary();
            _service = new ContactManagerService(new
            ModelStateWrapper(_ModelState), _repository);
        }

        [TestMethod]
        public void Index()
        {
            // Arrange
            var controller = new GroupController(_service);

            // Act
            var result = (ViewResult)controller.Index();

            // Assert
            Assert.IsInstanceOfType(result.ViewData.Model,
            typeof(IEnumerable));
        }

        [TestMethod]
        public void Create()
        {
            // Arrange
```

```

        var controller = new GroupController(_service);

        // Act
        var groupToCreate = new Group();
        groupToCreate.Name = "Business";
        controller.Create(groupToCreate);

        // Assert
        var result = (ViewResult)controller.Index();
        var groups = (IEnumerable)result.ViewData.Model;
        CollectionAssert.Contains(groups.ToList(), groupToCreate);
    }

    [TestMethod]
    public void CreateRequiredName()
    {
        // Arrange
        var controller = new GroupController(_service);

        // Act
        var groupToCreate = new Group();
        groupToCreate.Name = String.Empty;
        var result = (ViewResult)controller.Create(groupToCreate);

        // Assert
        var error = _ModelState["Name"].Errors[0];
        Assert.AreEqual("Name is required.", error.ErrorMessage);
    }
}

```

Après avoir effectué tous ces changements, une fois encore, l'ensemble de nos tests unitaires s'exécute avec succès. Nous avons donc terminé le cycle Rouge/Vert/Refactoring. Nous avons implémenté les 2 premiers scénarios d'usage. Nous avons maintenant également des tests unitaires supportant les besoins exprimés par les scénarios d'usage. Implémenter le reste des scénarios va consister à répéter ce même cycle Rouge/Vert/Refactoring.

Modification de notre base de données

Malheureusement, bien que nous ayons satisfait l'ensemble des besoins exprimés par nos tests unitaires, il nous reste encore du travail. Il nous faut encore modifier notre base de données.

Nous devons créer une nouvelle table pour les groupes dans notre base de données. Pour cela, suivez ces étapes :

1. Dans la fenêtre d'exploration des serveurs (Server Explorer), cliquez-droit sur le répertoire « Tables » et choisissez l'option Add New Table
2. Entrez les 2 colonnes dans le concepteur de tables comme indiqué ci-dessous
3. Marquez la colonne Id comme clé primaire et comme colonne d'identité

4. Sauvez cette nouvelle table en la nommant « Groups » en cliquant sur l'icône en forme de disquette.

Column Name	Data Type	Allow Nulls
Id	int	False
Name	nvarchar(50)	False

Ensuite, il faut supprimer l'ensemble des données présentes dans la table Contacts (sinon, nous ne pourrions pas mettre en place un lien de relation entre les tables « Contacts » et « Groups »). Suivez ces étapes :

1. Cliquez-droit sur la table « Contacts » et choisissez l'option Show Table Data
2. Supprimer l'ensemble des lignes de la table.

Etape suivante, il nous faut définir une relation entre notre table contenant les groupes et la table déjà existante contenant les contacts. Pour cela, suivez ces étapes :

1. Double-cliquez sur la table « Contacts » dans l'explorateur de serveurs pour ouvrir le concepteur de tables
2. Ajoutez une nouvelle colonne de type entier (Integer) à la table « Contacts » et nommez la GroupId
3. Cliquez sur le bouton « Relationships » à côté de l'icône en forme de clé pour ouvrir la boîte de dialogue « Foreign Key Relationships » (voir Figure 3).
4. Cliquez sur le bouton « Add » (Ajouter)
5. Cliquez sur le bouton en forme d'ellipse (avec 3 petits points) à droite de « Table and Columns Specification »
6. Dans la fenêtre « Tables and Columns », choisissez « Groups » comme table pour la clé primaire et Id comme colonne comme clé primaire. Choisissez « Contacts » comme table de clé étrangère et GroupID comme colonne pour la clé étrangère (voir Figure 4). Cliquez sur OK
7. En dessous de la partie « INSERT and UPDATE Specification », retenez la valeur Cascade pour la propriété Delete Rule
8. Cliquez sur le bouton « Close » pour fermer la fenêtre « Foreign Key Relationships ».
9. Cliquez sur bouton « Save » pour sauvegarder les changements dans la table « Contacts ».

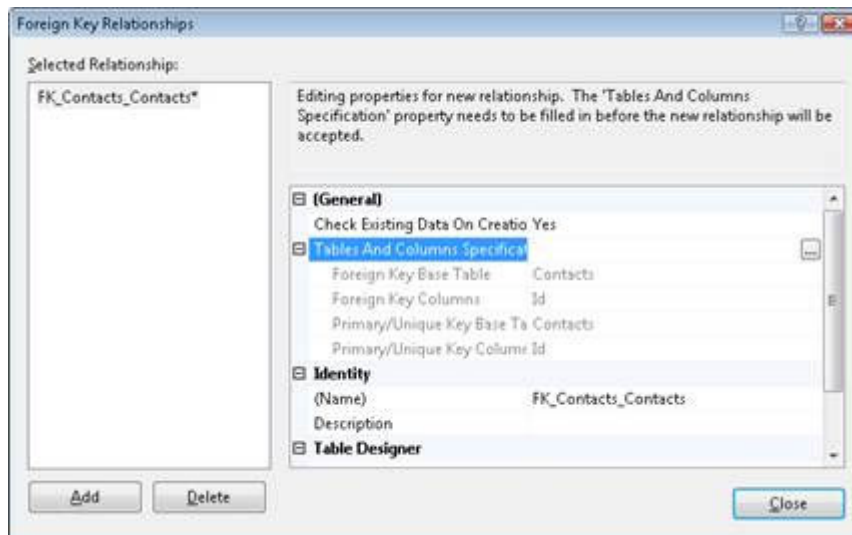


Figure 03: Mise en place d'un lien de relation entre les 2 tables

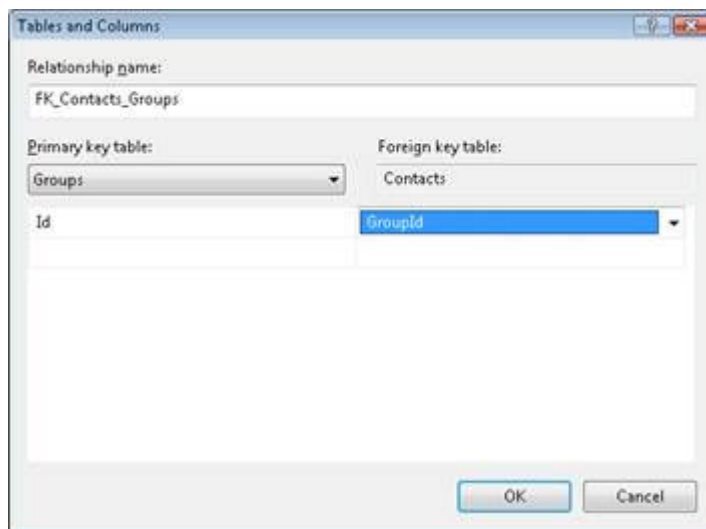


Figure 04: Spécification du type de relation

Mise à jour de notre modèle de données

Il faut maintenant mettre à jour la couche d'accès aux données pour prendre en compte les modifications que nous avons effectuées dans la base. Pour cela, suivez ces étapes :

1. Double-cliquez sur le fichier ContactManagerModel.edmx dans le répertoire « Models » afin d'ouvrir le concepteur d'Entity Framework.
2. Cliquez droit sur la surface de conception et choisissez l'option « Update Model from Database » (Mettre à jour le modèle depuis la base de données).
3. Dans l'assistant de mise à jour, choisissez la table « Groups » et cliquez sur le bouton « Finish » (voir Figure 5).
4. Cliquez droit sur l'entité Groups générée et choisissez l'option « Rename ». Renommez l'entité de Groups (pluriel) vers Group (singulier).

5. Cliquez droit sur la propriété de navigation « Groups » présente en bas de l'entité Contact. Renommez cette propriété de navigation vers Group (singulier).

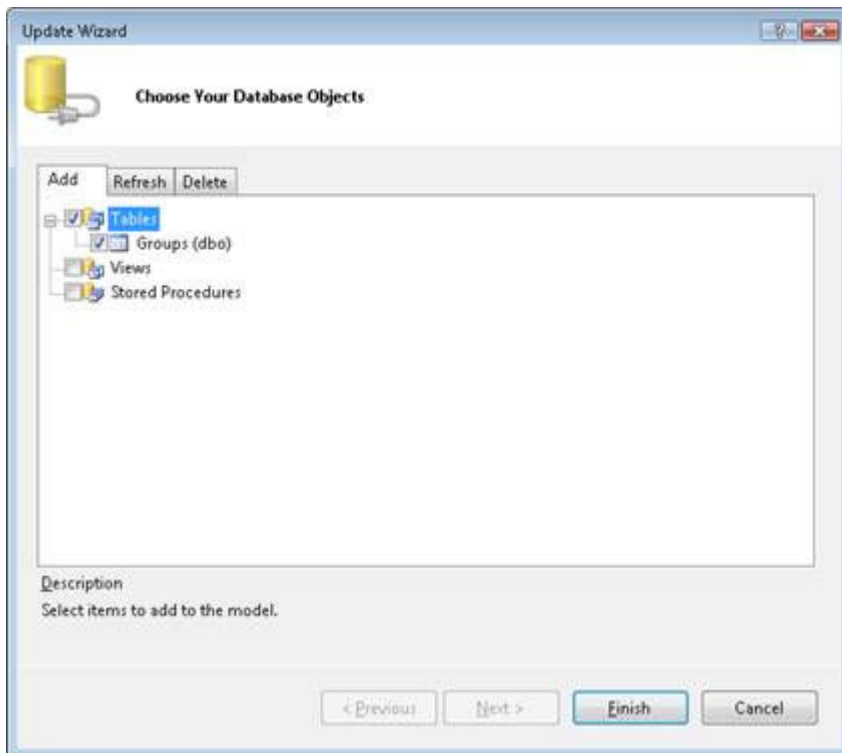


Figure 05: Mise à jour du modèle Entity Framework à partir de la base de données

Après avoir effectué ces étapes, votre modèle de données présentera à la fois les tables Contacts et Groups. Le concepteur d'Entity Framework devrait ressembler à la Figure 6.

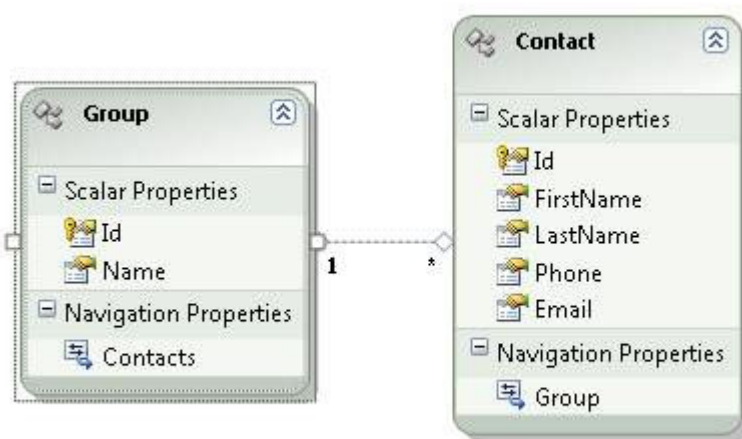


Figure 06: Le concepteur d'Entity Framework affichant les classes Group et Contact liées

Création de nos classes Repository

Nous allons maintenant implémenter nos classes Repository. Au fur et à mesure de cette étape, nous avons ajouté plusieurs nouvelles méthodes à l'interface `IContactManagerRepository` pendant que nous écrivions du code pour satisfaire nos tests unitaires. La version finale de l'interface `IContactManagerRepository` est présentée dans le Listing 14.

Listing 14 – Models\IContactManagerRepository.cs

```
using System.Collections.Generic;

namespace ContactManager.Models
{
    public interface IContactManagerRepository
    {
        // Contact methods
        Contact CreateContact(int groupId, Contact contactToCreate);
        void DeleteContact(Contact contactToDelete);
        Contact EditContact(int groupId, Contact contactToEdit);
        Contact GetContact(int id);

        // Group methods
        Group CreateGroup(Group groupToCreate);
        IEnumerable<Group> ListGroups();
        Group GetGroup(int groupId);
        Group GetFirstGroup();
        void DeleteGroup(Group groupToDelete);
    }
}
```

Pour l'instant, nous n'avons pas encore implémenté la moindre méthode en relation avec la gestion des groupes de contacts. Actuellement, la classe `EntityContactManagerRepository` dispose de méthodes vides pour chacune des méthodes de gestion des groupes de contacts listée dans l'interface `IContactManagerRepository`. Par exemple, la méthode `ListGroups()` ressemble actuellement à cela :

```
public IEnumerable<Group> ListGroups()
{
    throw new NotImplementedException();
}
```

Cette coquille vide nous permet de compiler notre application et de passer nos tests unitaires avec succès. Il est désormais temps de réellement implémenter ces méthodes. La version finale de la classe `EntityContactManagerRepository` est contenue dans le Listing 13.

Listing 13 – Models\EntityContactManagerRepository.cs

```
using System.Collections.Generic;
using System.Linq;
using System;

namespace ContactManager.Models
{

```

```

public class EntityContactManagerRepository :
ContactManager.Models.IContactManagerRepository
{
    private ContactManagerDBEntities _entities = new
ContactManagerDBEntities();

    // Contact methods

    public Contact GetContact(int id)
    {
        return (from c in _entities.ContactSet.Include("Group")
                where c.Id == id
                select c).FirstOrDefault();
    }

    public Contact CreateContact(int groupId, Contact contactToCreate)
    {
        // Associate group with contact
        contactToCreate.Group = GetGroup(groupId);

        // Save new contact
        _entities.AddToContactSet(contactToCreate);
        _entities.SaveChanges();
        return contactToCreate;
    }

    public Contact EditContact(int groupId, Contact contactToEdit)
    {
        // Get original contact
        var originalContact = GetContact(contactToEdit.Id);

        // Update with new group
        originalContact.Group = GetGroup(groupId);

        // Save changes
        _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName,
contactToEdit);
        _entities.SaveChanges();
        return contactToEdit;
    }

    public void DeleteContact(Contact contactToDelete)
    {
        var originalContact = GetContact(contactToDelete.Id);
        _entities.DeleteObject(originalContact);
        _entities.SaveChanges();
    }

    public Group CreateGroup(Group groupToCreate)
    {
        _entities.AddToGroupSet(groupToCreate);
        _entities.SaveChanges();
        return groupToCreate;
    }

    // Group Methods

```

```

public IEnumerable<Group> ListGroups()
{
    return _entities.GroupSet.ToList();
}

public Group GetFirstGroup()
{
    return _entities.GroupSet.Include("Contacts").FirstOrDefault();
}

public Group GetGroup(int id)
{
    return (from g in _entities.GroupSet.Include("Contacts")
            where g.Id == id
            select g).FirstOrDefault();
}

public void DeleteGroup(Group groupToDelete)
{
    var originalGroup = GetGroup(groupToDelete.Id);
    _entities.DeleteObject(originalGroup);
    _entities.SaveChanges();
}
}
}

```

Création des vues

On ne crée pas des vues en réponse à un test unitaire. En effet, comme nous le disions en début de ce tutoriel, les tests unitaires ne visent pas le rendu des vues. Cependant, ce n'est pas par ce que nous utilisons actuellement une programmation pilotée par les tests qu'il faut considérer qu'une application n'a pas besoin d'interface graphique. Il nous faut donc bien évidemment ajouter de nouvelles vues et en modifier certaines dans notre application pour pouvoir exposer à l'utilisateur ces nouvelles fonctionnalités.

Voici les nouvelles vues que l'on doit créer pour gérer nos groupes de contacts (voir Figure 7)

- Views\Group\Index.aspx – Affiche la liste des groupes de contacts disponibles
- Views\Group\Delete.aspx – Affiche un formulaire de confirmation de suppression d'un groupe spécifique de contacts

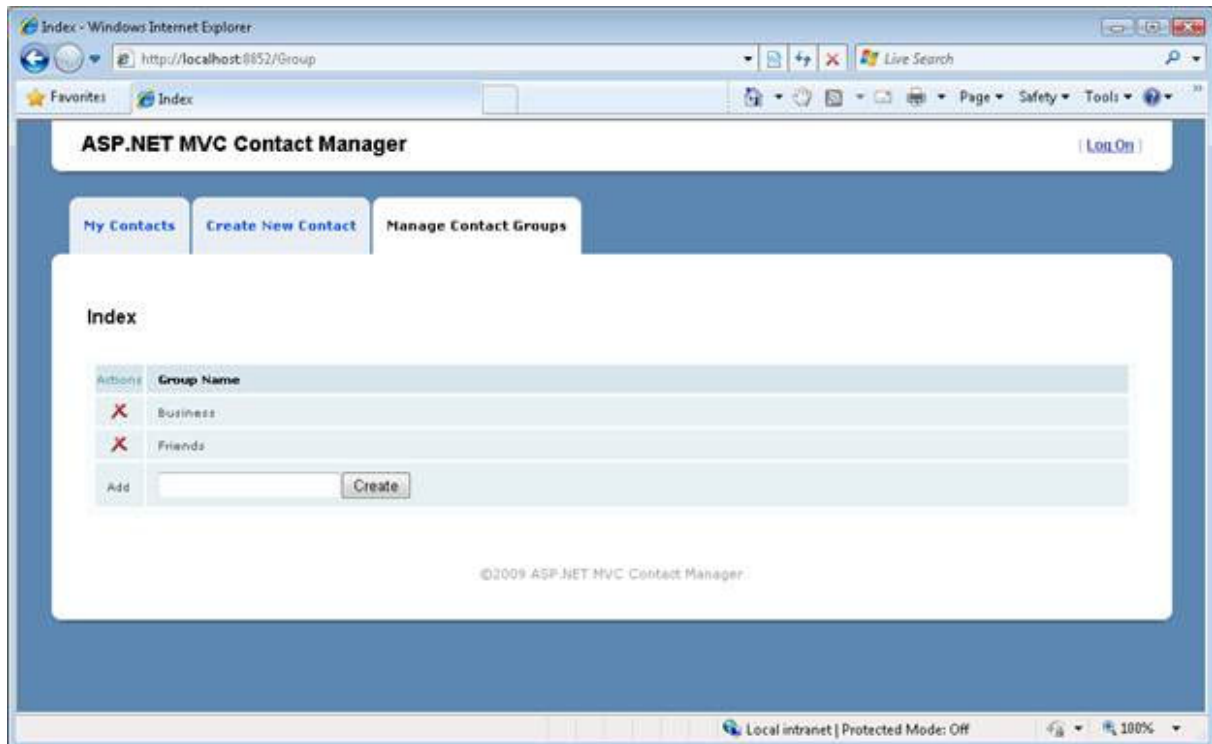


Figure 07: La vue Index pour les groupes

Il faut également modifier les vues suivantes afin d'y ajouter la notion de groupes:

- Views\Home\Create.aspx
- Views\Home\Edit.aspx
- Views\Home\Index.aspx

Afin de trouver ces nouvelles vues, il vous faut les récupérer dans l'application Visual Studio fournie avec ce tutoriel. La Figure 8 présente ainsi la nouvelle vue Index pour les contacts.

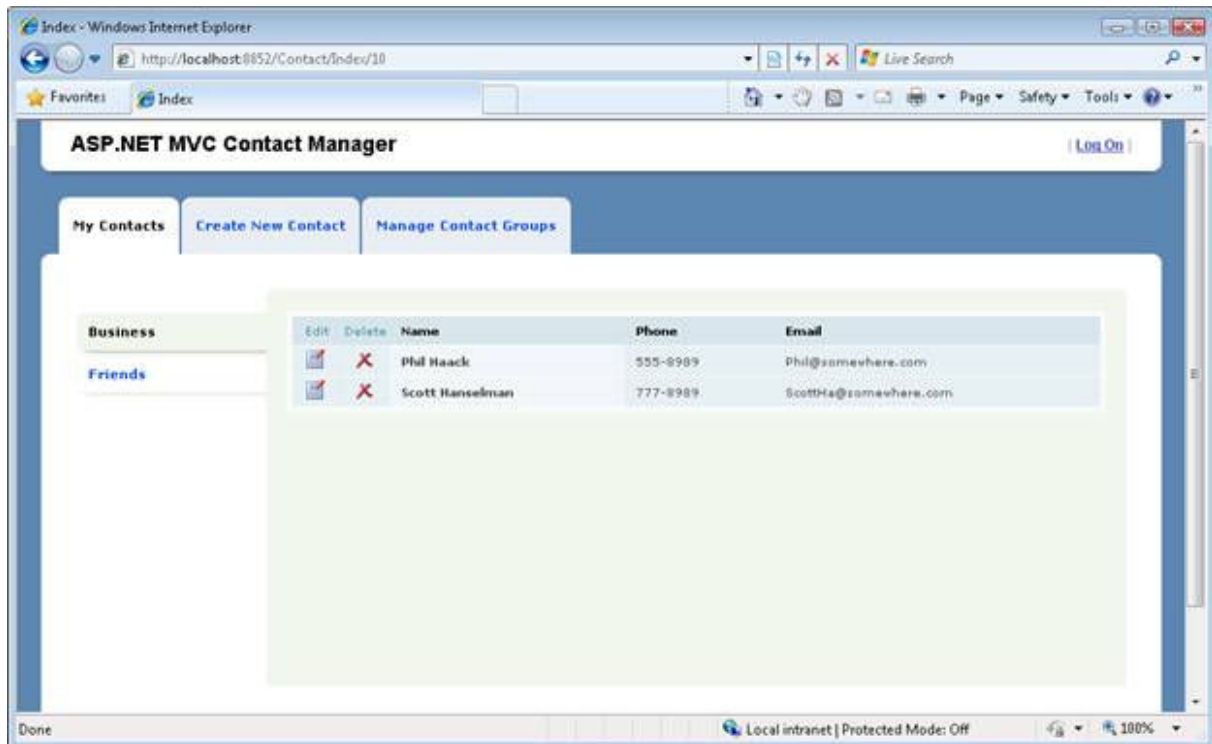


Figure 08: La vue Index pour les contacts

Conclusion

Dans cette étape, nous avons ajouté de nouvelles fonctionnalités à notre application en suivant une méthodologie de programmation pilotée par les tests. Nous avons commencé par créer une série de scénarios d'usage utilisateur. Nous avons ensuite réalisé un ensemble de tests unitaires à partir de ces scénarios d'usage. Cela nous a donc permis d'écrire juste suffisamment de code afin de satisfaire les besoins exprimés par ces tests unitaires avant d'entreprendre sereinement une revue complète de l'architecture du code.

Après cela, nous avons mis à jour notre base de données et nos vues. Nous avons ainsi ajouté une nouvelle table pour les groupes dans notre base puis nous avons mis à jour notre modèle de données Entity à partir de cela. Nous avons également créé et modifier une série de vues associées.

Dans l'étape suivante, la dernière, nous allons revoir notre application afin de tirer parti des avantages de la technologie Ajax. Grâce à Ajax, nous allons améliorer la réponse et les performances de notre application de gestion de contacts.

Etape 7 – Ajout du support d'Ajax

Dans cette étape

Nous allons ici revoir l'architecture de l'application afin d'utiliser la technologie Ajax. Grâce à Ajax, nous allons rendre notre application plus performante. Nous allons en effet éviter de générer à nouveau entièrement la page lorsque nous avons uniquement besoin de mettre à jour une certaine région de celle-ci.

Ainsi, nous allons revoir la vue Index pour ne pas avoir à réafficher la page entière lorsque quelqu'un sélectionne un nouveau groupe de contacts. Au lieu de cela, lorsqu'un utilisateur cliquera sur un groupe de contacts, nous nous occuperons juste de mettre à jour la liste des contacts associés et de laisser le reste de la page telle quelle.

Nous allons également changer la manière dont nos liens de suppression fonctionnent. Au lieu d'afficher une page de confirmation séparée, nous allons afficher une boîte de dialogue en JavaScript. Si vous confirmez que vous souhaitez réellement supprimer un contact, une opération HTTP DELETE sera effectuée sur le serveur afin de le supprimer de la base de données.

De plus, nous allons tirer partie de la technologie jQuery pour ajouter des effets d'animation à notre vue Index. Nous afficherons une animation lorsque qu'une nouvelle liste de contacts sera récupérée depuis le serveur par exemple.

Pour terminer, nous bénéficierons du framework AJAX d'ASP.NET pour gérer correctement l'historique du navigateur. Nous créerons des points d'historique dès que nous effectuerons des appels Ajax pour mettre à jour la liste des contacts. Ainsi, les boutons d'historique retour/avant du navigateur fonctionneront comme attendu.

Pourquoi utiliser Ajax ?

L'utilisation d'Ajax apporte de nombreux bénéfices. Tout d'abord, l'ajout d'Ajax à une application apporte une meilleure expérience utilisateur. Dans une application Web classique, la page entière doit être renvoyée au serveur à chaque fois que l'utilisateur effectue la moindre action. Ainsi, quoique vous fassiez, le navigateur se bloque et l'utilisateur doit attendre que la page soit retournée par le serveur et affichée à nouveau.

Cela constituerait une expérience inacceptable dans le cas d'une application de bureau. Cependant, nous avons fini par nous habituer à cette mauvaise expérience dans le cas d'une application Web car nous ne savions pas que nous pouvions mieux faire. Nous pensions que cela était lié aux limitations intrinsèques des applications Web. En fait, c'était juste une limitation de notre propre imagination.

Avec une application Ajax, vous n'avez pas besoin de bloquer le navigateur et l'expérience utilisateur juste pour mettre à jour une page. Au contraire, vous pouvez effectuer une requête asynchrone en arrière-plan pour effectuer cette mise à jour. Vous ne forcez ainsi pas l'utilisateur à patienter pendant qu'une partie de la page est en cours de mise à jour.

En tirant partie d'Ajax, vous pouvez également améliorer les performances de votre application. Regardez comment l'application Contact Manager fonctionne actuellement sans la présence de fonctionnalités Ajax. Lorsque vous cliquez sur un groupe de contacts, la vue Index entière doit être réaffichée. La liste des contacts et la liste des groupes de contacts doivent être rechargées depuis la base de données. Toutes ces données doivent ainsi être transférées sur le réseau depuis le serveur Web vers le navigateur.

Après l'ajout des fonctionnalités Ajax à notre application, nous pourrions ainsi éviter de recharger la page entière lorsqu'un utilisateur cliquera sur un groupe de contacts. Nous n'aurons plus besoin de charger les groupes de contacts depuis la base de données. Nous n'aurons plus besoin non plus de pousser la vue Index entière sur le réseau. En bénéficiant d'Ajax, nous réduisons ainsi la quantité de travail effectuée par notre base de données et nous réduisons également le trafic réseau nécessaire pour notre application.

N'ayez pas peur d'Ajax

Certains développeurs évitent d'utiliser Ajax car ils ont peur du comportement de leurs applications au sein de certains navigateurs. Ils veulent être sûrs que leurs applications Web fonctionnent encore lorsqu'elles seront accédées depuis un navigateur qui ne supporte pas JavaScript. Or, comme Ajax est fondé sur JavaScript, certains développeurs ne souhaitent par conséquent pas utiliser Ajax.

Cependant, si vous faites attention à la manière d'implémenter Ajax, vous pouvez développer des applications qui pourront fonctionner avec les 2 niveaux de navigateurs. Ainsi notre application Contact Manager pourra fonctionner avec les navigateurs supportant JavaScript et les navigateurs ne le supportant pas.

Si vous utilisez notre application de gestion de contacts avec un navigateur supportant JavaScript alors vous aurez une meilleure expérience utilisateur. Par exemple, lorsque vous cliquerez sur un groupe de contacts, uniquement la partie de la page correspondante sera mise à jour.

Si, d'un autre côté, vous utilisez l'application avec un navigateur ne supportant pas JavaScript (ou si JavaScript a été désactivé) alors vous aurez une expérience un peu moins attrayante. Ainsi, lorsque vous cliquerez sur un groupe de contacts, la page Index entière sera renvoyée vers le navigateur afin d'afficher la liste correspondante de contacts.

Ajouter les fichiers JavaScript requis

Nous aurons besoin d'utiliser 3 fichiers JavaScript pour ajouter les fonctionnalités Ajax à notre application. L'ensemble de ces 3 fichiers sont inclus dans le répertoire Scripts d'une nouvelle application ASP.NET MVC.

Si vous prévoyez d'utiliser Ajax dans plusieurs pages de votre application alors il apparaît logique d'inclure ces fichiers au sein de la page maître de votre application. Ainsi, les fichiers JavaScript seront automatiquement inclus dans l'ensemble des pages de votre application.

Ajoutez les inclusions JavaScript suivantes au sein du tag <head> de votre page maître :


```
<script src="../../../Scripts/MicrosoftAjax.js" type="text/javascript"></script>
<script src="../../../Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
<script src="../../../Scripts/jquery-1.2.6.min.js" type="text/javascript"></script>
```

Refonte de la vue Index pour utiliser Ajax

Commençons par modifier la vue Index de manière à mettre à jour uniquement une partie de la page lorsque l'on clique sur un groupe de contacts. Le rectangle rouge de la Figure 1 présente la région que l'on souhaite mettre à jour.

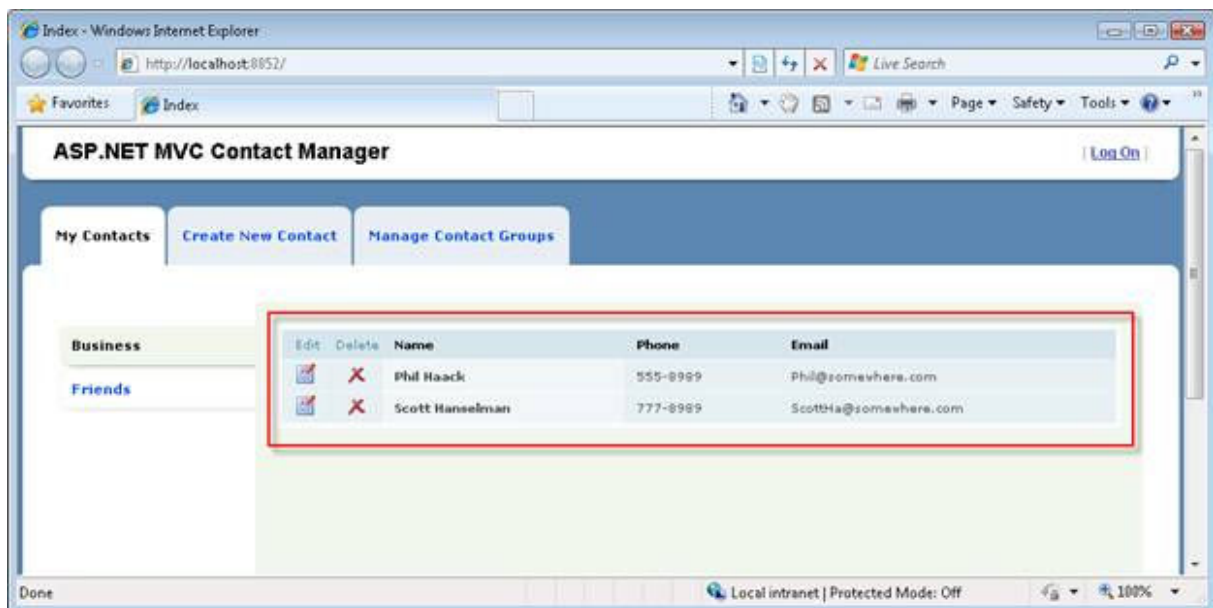


Figure 01: Mise à jour uniquement des contacts

La première étape est de séparer la partie de la vue que l'on souhaite mettre à jour de manière asynchrone dans une zone séparée avec l'utilisation d'un user control (.ascx). Ainsi la section de la vue Index s'occupant d'afficher la table des contacts a été transférée dans le contrôle ASCX du Listing 1.

Listing 1 – Views\Contact\ContactList.ascx

```
<%@ Control Language="C#"
Inherits="System.Web.Mvc.ViewUserControl<ContactManager.Models.Group>" %>
<%@ Import Namespace="Helpers" %>
<table class="data-table" cellpadding="0" cellspacing="0">
  <thead>
    <tr>
      <th class="actions edit">
        Edit
      </th>
      <th class="actions delete">
        Delete
      </th>
      <th>
```

```

        Name
    </th>
    <th>
        Phone
    </th>
    <th>
        Email
    </th>
</tr>
</thead>
<tbody>
    <% foreach (var item in Model.Contacts)
    { %>
    <tr>
        <td class="actions edit">
            <a href='<%= Url.Action("Edit", new {id=item.Id}) %>'></a>
        </td>
        <td class="actions delete">
            <a href='<%= Url.Action("Delete", new {id=item.Id}) %>'></a>
        </td>
        <th>
            <%= Html.Encode(item.FirstName) %>
            <%= Html.Encode(item.LastName) %>
        </th>
        <td>
            <%= Html.Encode(item.Phone) %>
        </td>
        <td>
            <%= Html.Encode(item.Email) %>
        </td>
    </tr>
    <% } %>
</tbody>
</table>

```

Notez que la vue partielle du Listing 1 utilise un modèle différent de celui de la vue principale. L'attribut `Inherits` de la directive `<%@ Page %>` spécifie que la vue partielle hérite de la classe `ViewUserControl<Group>`.

La vue Index principale mise à jour est contenue dans le Listing 2.

Listing 2 – Views\Contact\Index.aspx

```

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.ViewData.IndexModel>"
%>
<%@ Import Namespace="Helpers" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<ul id="leftColumn">
<% foreach (var item in Model.Groups) { %>

```

```

        <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
            <%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new
AjaxOptions { UpdateTargetId = "divContactList"})%>
        </li>
    <% } %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup); %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>

```

Il y a 2 choses à noter dans la vue mise à jour du Listing 2. Tout d'abord, notez que tout le contenu déplacé dans la vue partielle a été remplacé par un appel à `Html.RenderPartial()`. La méthode `Html.RenderPartial()` est appelée lorsque la vue Index est demandée pour la 1ère fois afin d'afficher la liste des contacts.

Par ailleurs, notez que la méthode `Html.ActionLink()` utilisée pour afficher les groupes de contacts a été remplacée par un appel à `Ajax.ActionLink()`. `Ajax.ActionLink()` est appelée avec les paramètres suivants :

```

<%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new AjaxOptions
{ UpdateTargetId = "divContactList"})%>

```

Le 1er paramètre indique le texte à afficher pour le lien, le 2ème paramètre les valeurs à passer et le 3ème paramètre les options Ajax. Dans notre cas, nous utilisons l'option `UpdateTargetId` pour viser le tag HTML `<div>` que nous souhaitons mettre à jour une fois la requête Ajax terminée. Nous voulons ainsi mettre à jour le tag `<div>` avec la nouvelle liste de contacts.

La méthode `Index()` mise à jour pour le contrôleur Contact se trouve dans le Listing 3.

Listing 3 – Controllers\ContactController.cs (méthode Index)

```

public ActionResult Index(int? id)
{
    // Get selected group
    var selectedGroup = _service.GetGroup(id);
    if (selectedGroup == null)
        return RedirectToAction("Index", "Group");

    // Normal Request
    if (!Request.IsAjaxRequest())
    {
        var model = new IndexModel
        {
            Groups = _service.ListGroups(),
            SelectedGroup = selectedGroup
        };
        return View("Index", model);
    }

    // Ajax Request
    return PartialView("ContactList", selectedGroup);
}

```

```
}
```

L'action Index() mise à jour retourne 2 choses possibles en fonction du contexte de l'appelant. Si l'action Index() est appelée depuis une requête Ajax alors le contrôleur retourne la vue partielle. Sinon, l'action Index() retourne la vue entière.

Notez que l'action Index() n'a pas besoin de renvoyer beaucoup de données lorsqu'elle est appelée depuis une requête Ajax. Dans le cas d'une requête normale, l'action Index retourne la liste de l'ensemble des groupes de contacts ainsi que le groupe sélectionné. Dans le cas d'une requête Ajax, l'action Index ne retourne uniquement que le groupe sélectionné. Ajax implique ainsi moins de charge sur votre serveur de base de données.

Par ailleurs, notre nouvelle vue fonctionne avec les 2 types de navigateurs. Si vous cliquez sur un groupe de contacts et que votre navigateur supporte JavaScript alors uniquement la région qui contient la liste des contacts sera mise à jour. Si votre navigateur ne supporte pas JavaScript, la vue entière sera bien retournée.

Malgré tout, notre nouvelle vue Index a un petit problème. Lorsque vous cliquez sur un groupe de contacts, le groupe sélectionné n'est pas surligné. Comme la liste des groupes est affichée en dehors de la région qui est mise à jour par la requête Ajax, le bon groupe n'est pas correctement surligné. Nous allons fixer ce problème dans la section suivante.

Ajout des effets d'animation jQuery

Normalement, lorsque vous cliquez sur un lien vers une page web, vous pouvez vérifier que le navigateur télécharge des données à travers la barre de progression. Cependant, lorsque vous effectuez une requête Ajax, la barre de progression du navigateur n'est pas du tout utilisée. Les utilisateurs peuvent ainsi se demander ce qu'il se passe. Comment en effet savoir si le navigateur est figé ou non ?

Il y a plusieurs façons d'indiquer à l'utilisateur qu'une opération est en cours pendant l'exécution d'une requête Ajax. Une des approches consiste à afficher une petite animation. Par exemple, vous pouvez faire disparaître avec un effet de fondu une région lorsqu'une requête Ajax commence et la faire réapparaître lorsque la requête est terminée.

Nous allons utiliser la librairie jQuery qui est incluse dans le framework Microsoft ASP.NET MVC afin de créer les effets d'animation. La vue Index mise à jour se trouve dans le Listing 4.

Listing 4 – Views\Contact\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.ViewData.IndexModel>"
%>
<%@ Import Namespace="Helpers" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
```

```

<script type="text/javascript">

    function beginContactList(args) {
        // Highlight selected group
        $('#leftColumn li').removeClass('selected');
        $(this).parent().addClass('selected');

        // Animate
        $('#divContactList').fadeOut('normal');
    }

    function successContactList() {
        // Animate
        $('#divContactList').fadeIn('normal');
    }

    function failureContactList() {
        alert("Could not retrieve contacts.");
    }

</script>

<ul id="leftColumn">
<% foreach (var item in Model.Groups) { %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
        <%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new
AjaxOptions { UpdateTargetId = "divContactList", OnBegin = "beginContactList",
OnSuccess = "successContactList", OnFailure = "failureContactList" })%>
    </li>
<% } %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup); %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>

```

Notez que la nouvelle vue Index contient 3 nouvelles fonctions JavaScript. Les 2 premières fonctions utilisent jQuery pour faire disparaître puis apparaître la liste des contacts lorsque vous cliquez sur un nouveau groupe de contacts. La 3ème fonction affiche un message d’erreur lorsqu’une requête Ajax part elle-même en erreur (par exemple lors d’une erreur réseau).

La 1ère fonction s’occupe également de surligner le groupe sélectionné. Un attribut de classe ‘selected’ est ajouté à l’élément parent (l’élément LI) de l’élément cliqué. A nouveau, jQuery rend aisé la sélection du bon élément et l’ajout de la classe CSS.

Ces scripts sont liés au groupe de liens à l’aide du paramètre AjaxOptions de la méthode Ajax.ActionLink(). La méthode Ajax.ActionLink() mise à jour ressemble alors à cela :

```
<%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new
AjaxOptions { UpdateTargetId = "divContactList", OnBegin = "beginContactList",
OnSuccess = "successContactList", OnFailure = "failureContactList" })%>
```

Ajout du support de l'historique du navigateur

Normalement, lorsque vous cliquez sur un lien pour mettre à jour votre page, l'historique du navigateur est mis à jour. Ainsi, vous pouvez cliquer sur le bouton « Précédent » pour revenir en arrière sur l'état précédent de la page. Par exemple, lorsque vous cliquez sur le groupe de contacts « Amis » et que vous cliquez ensuite sur le groupe « Travail », vous pouvez utiliser le bouton « Précédent » du navigateur pour revenir en arrière à nouveau sur le groupe « Amis ».

Malheureusement, l'exécution d'une requête Ajax ne met pas à jour l'historique du navigateur automatiquement. Si vous cliquez sur un groupe de contacts et que le groupe associé est retourné grâce à une requête Ajax alors l'historique n'est pas mis à jour. Vous ne pouvez ainsi pas utiliser le bouton « Précédent » pour revenir sur un groupe de contacts précédemment sélectionné.

Si vous souhaitez permettre aux utilisateurs de naviguer dans l'historique après l'exécution de requêtes Ajax, vous devez mettre en place un petit peu de code. Vous devez en effet vous servir du gestionnaire d'historique du navigateur inclus dans le framework ASP.NET AJAX.

Pour mettre en place le support de l'historique ASP.NET AJAX, vous devez faire 3 choses :

1. Activer la propriété `enableBrowserHistory` à vrai.
2. Sauvegarder les points d'historique lorsque l'état d'une vue change en appelant la méthode `addHistoryPoint()`.
3. Reconstruire l'état de la vue lorsque l'évènement de navigation est levé.

La vue Index mise à jour est présentée dans le Listing 5.

Listing 5 – Views\Contact\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.ViewData.IndexModel>"
%>
<%@ Import Namespace="Helpers" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
    <title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<script type="text/javascript">

    var _currentGroupId = -1;

    Sys.Application.add_init(pageInit);

    function pageInit() {
        // Enable history
```

```

        Sys.Application.set_enableHistory(true);

        // Add Handler for history
        Sys.Application.add_navigate(navigate);
    }

    function navigate(sender, e) {
        // Get groupId from address bar
        var groupId = e.get_state().groupId;

        // If groupId != currentGroupId then navigate
        if (groupId != _currentGroupId) {
            _currentGroupId = groupId;
            $("#divContactList").load("/Contact/Index/" + groupId);
            selectGroup(groupId);
        }
    }

    function selectGroup(groupId) {
        $('#leftColumn li').removeClass('selected');
        if (groupId)
            $('a[groupid=' + groupId + ']').parent().addClass('selected');
        else
            $('#leftColumn li:first').addClass('selected');
    }

    function beginContactList(args) {
        // Highlight selected group
        _currentGroupId = this.getAttribute("groupid");
        selectGroup(_currentGroupId);

        // Add history point
        Sys.Application.addHistoryPoint({ "groupId": _currentGroupId });

        // Animate
        $('#divContactList').fadeOut('normal');
    }

    function successContactList() {
        // Animate
        $('#divContactList').fadeIn('normal');
    }

    function failureContactList() {
        alert("Could not retrieve contacts.");
    }
}

</script>

<ul id="leftColumn">
<% foreach (var item in Model.Groups) { %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
        <%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new
AjaxOptions { UpdateTargetId = "divContactList", OnBegin = "beginContactList",
OnSuccess = "successContactList", OnFailure = "failureContactList" }, new {
groupid = item.Id })%>

```

```

        </li>
    <% } %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup); %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>

```

Dans le Listing 5, le support de l'historique est activé dans la fonction `pageInit()`. Cette fonction est également utilisée pour mettre en place le gestionnaire d'évènement associé à l'évènement de navigation. L'évènement de navigation est levé dès que les boutons « Précédent » ou « Suivant » sont pressés pour changer l'état de la page.

La méthode `beginContactList()` est appelée lorsque vous cliquez sur un groupe de contacts. Cette méthode crée un nouveau point de sauvegarde de l'historique en appelant la méthode `addHistoryPoint()`. L'identifiant du groupe de contacts cliqué est alors ajouté dans l'historique.

L'identifiant du groupe est retrouvé à partir d'un attribut étendu positionné sur le lien du groupe de contacts. Le lien est ainsi généré avec l'appel suivant à la méthode `Ajax.ActionLink()` :

```

<%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new
AjaxOptions { UpdateTargetId = "divContactList", OnBegin = "beginContactList",
OnSuccess = "successContactList", OnFailure = "failureContactList" }, new
{groupid=item.Id})%>

```

Le dernier paramètre passé à la méthode `Ajax.ActionLink()` ajoute le nouvel attribut appelé `groupid` au lien (en minuscule pour une compatibilité XHTML).

Lorsqu'un utilisateur clique sur « Précédent » ou « Suivant », l'évènement de navigation est levé et la méthode `navigate()` est appelée. Cette méthode met à jour les contacts affichés dans la page de manière à faire correspondre l'état de la page au point de sauvegarde de l'historique qui lui est passé en paramètre.

Effectuer des suppressions avec Ajax

Actuellement, pour supprimer un contact, vous devez cliquer sur le lien de suppression puis cliquez sur le bouton « Delete » affiché dans la page de confirmation (Figure 2). Cela implique pas mal d'aller/retours serveurs pour faire quelque chose d'aussi simple que de supprimer un enregistrement de la base de données.

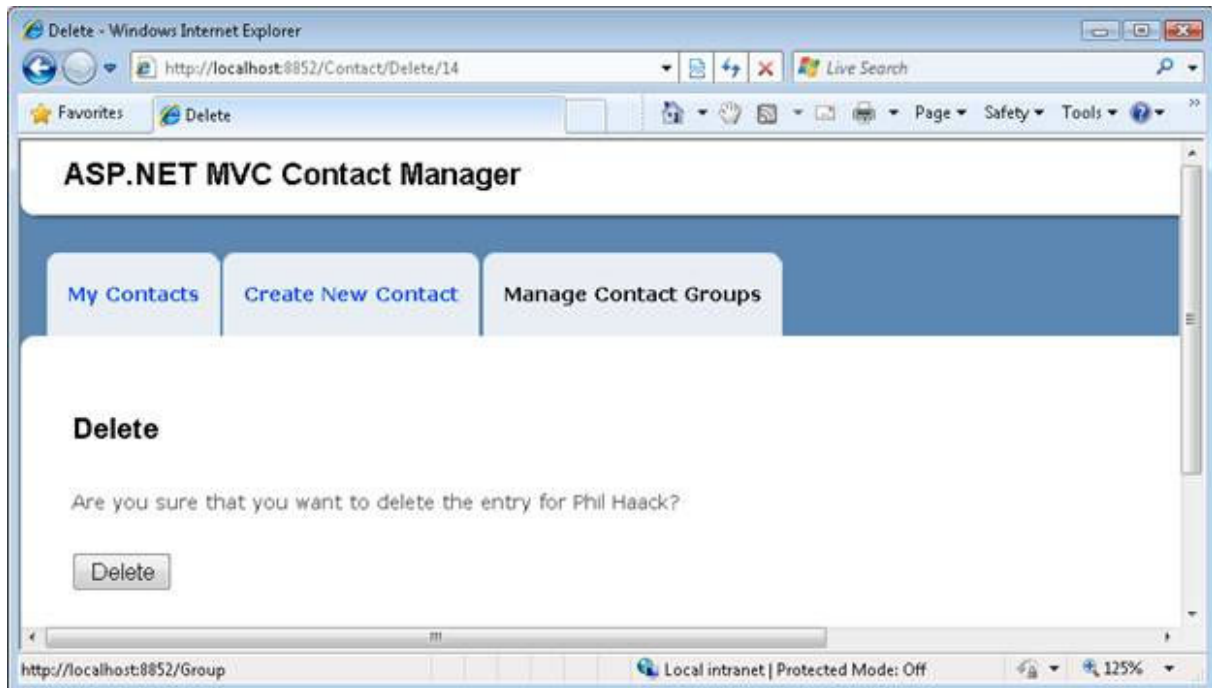


Figure 02: La page de confirmation de suppression

Il est donc tentant de vouloir éviter la page de confirmation et de supprimer un contact directement depuis la vue Index. Vous devriez cependant éviter cette tentation car cette approche amène potentiellement vers des trous de sécurité de votre application. En général, vous ne souhaitez pas qu'une opération de type HTTP GET modifie l'état de votre application Web. Lorsque vous effectuez une suppression, vous souhaitez plutôt effectuer un HTTP POST ou mieux encore une opération de type HTTP DELETE.

Le lien de suppression est présent dans la vue partielle ContactList. Une version mise à jour de la vue partielle est présentée dans le Listing 6.

Listing 6 – Views\Contact\ContactList.ascx

```
<%@ Control Language="C#"
Inherits="System.Web.Mvc.ViewUserControl<ContactManager.Models.Group>" %>
<%@ Import Namespace="Helpers" %>
<table class="data-table" cellpadding="0" cellspacing="0">
  <thead>
    <tr>
      <th class="actions edit">
        Edit
      </th>
      <th class="actions delete">
        Delete
      </th>
      <th>
        Name
      </th>
      <th>
        Phone
      </th>
    </tr>
  </thead>
</table>
```

```

        Email
    </th>
</tr>
</thead>
<tbody>
    <% foreach (var item in Model.Contacts)
    { %>
    <tr>
        <td class="actions edit">
            <a href='<%= Url.Action("Edit", new {id=item.Id}) %>'></a>
        </td>
        <td class="actions delete">
            <%= Ajax.ImageActionLink("../Content/Delete.png", "Delete",
"Delete", new { id = item.Id }, new AjaxOptions { Confirm = "Delete contact?",
HttpMethod = "Delete", UpdateTargetId = "divContactList" })%>
        </td>
        <th>
            <%= Html.Encode(item.FirstName) %>
            <%= Html.Encode(item.LastName) %>
        </th>
        <td>
            <%= Html.Encode(item.Phone) %>
        </td>
        <td>
            <%= Html.Encode(item.Email) %>
        </td>
    </tr>
    <% } %>
</tbody>
</table>

```

Le lien de suppression est généré avec l'appel suivant à la méthode `Ajax.ImageActionLink()` :

```

<%= Ajax.ImageActionLink("../Content/Delete.png", "Delete", "Delete", new {
id = item.Id }, new AjaxOptions { Confirm = "Delete contact?", HttpMethod =
"Delete", UpdateTargetId = "divContactList" })%>

```

La méthode `Ajax.ImageActionLink()` n'est pas incluse en standard dans le framework ASP.NET MVC. Cette méthode est un helper inclus dans le projet Contact Manager.

Pour disposer du helper, ajoutez une nouvelle classe sous le répertoire « Helpers » et nommez la : `ImageActionLinkHelper.cs`. Voici le code associé :

Listing 7 – Helpers\ImageActionLinkHelper.cs (ImageActionLink)

```

using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace Helpers
{
    public static class ImageActionLinkHelper
    {
        public static string ImageActionLink(this AjaxHelper helper, string
imageUrl, string altText, string actionName, object routeValues, AjaxOptions
ajaxOptions)

```

```

        {
            var builder = new TagBuilder("img");
            builder.MergeAttribute("src", imageUrl);
            builder.MergeAttribute("alt", altText);
            var link = helper.ActionLink("[replaceme]", actionName,
            routeValues, ajaxOptions);
            return link.Replace("[replaceme]",
            builder.ToString(TagRenderMode.SelfClosing));
        }
    }
}

```

Le paramètre `AjaxOptions` dispose lui-même de 2 paramètres. Tout d'abord, la propriété `Confirm` est utilisée pour afficher une boîte de dialogue de confirmation JavaScript. Ensuite, la propriété `HttpMethod` est utilisée pour effectuer l'opération HTTP DELETE.

Le Listing 7 contient la nouvelle action `AjaxDelete` qui a été ajoutée au contrôleur `Contact`.

Listing 8 – Controllers\ContactController.cs (AjaxDelete)

```

[AcceptVerbs(HttpVerbs.Delete)]
[ActionName("Delete")]
public ActionResult AjaxDelete(int id)
{
    // Get contact and group
    var contactToDelete = _service.GetContact(id);
    var selectedGroup = _service.GetGroup(contactToDelete.Group.Id);

    // Delete from database
    _service.DeleteContact(contactToDelete);

    // Return Contact List
    return PartialView("ContactList", selectedGroup);
}

```

L'action `AjaxDelete()` est décorée avec un attribut `AcceptVerbs`. Cet attribut évite que l'action soit invoquée par toute autre opération qu'un HTTP DELETE. En particulier, vous ne pourrez pas invoquer cette action à travers un HTTP GET.

Après qu'un enregistrement soit supprimé, vous devez afficher la liste mise à jour des contacts ne contenant plus l'élément supprimé. La méthode `AjaxDelete()` retourne ainsi la vue partielle `ContactList` et la liste mise à jour des contacts.

En résumé

Dans cette étape, nous avons ajouté des fonctionnalités Ajax à notre application de gestion de contacts. Nous avons utilisé Ajax pour améliorer les temps de réponse et la performance globale de notre application.

Tout d'abord, nous avons revu la vue Index pour que la sélection d'un groupe de contacts ne mette pas à jour la vue entière. A la place, la sélection d'un groupe de contacts se contente de mettre à jour la liste des contacts associés.

Ensuite, nous avons utilisé des effets d'animations jQuery pour faire apparaître ou disparaître la liste des contacts sous forme de fondus. L'ajout d'animations à une application Ajax peut être fait pour fournir à l'utilisateur l'équivalent d'une barre de progression.

Nous avons également ajouté le support de l'historique du navigateur à notre application. Malgré l'utilisation d'Ajax, nous avons permis aux utilisateurs d'utiliser les boutons de navigation « Précédent » et « Suivant » pour avoir un résultat tel qu'attendu habituellement.

Pour terminer, nous avons créé un lien de suppression qui utilise les opérations de type HTTP DELETE. En effectuant des suppressions avec Ajax, nous permettons aux utilisateurs de supprimer des enregistrements sans avoir besoin de passer par une page de confirmation supplémentaire générant un aller/retour inutile vers le serveur.