1 Introduction

ASP.NET est la partie du Framework Microsoft .NET dédiée au développement des applications web.
Le Framework .NET est constitué de deux parties : une librairie de classes et le CLR (Common Runtime Language).
Il existe 5 versions du Framework .NET : 1.0, 1.1, 2.0, 3.0 et 3.5. Les versions 1.x sont obsolètes. Nous étudierons ici la version 3.5.

1.1 La Librairie de classes

Le Framework .NET contient des milliers de classes que vous pouvez utiliser pour développer vos applications. Parmi ces classes, on peut citer par exemple la classe **File** utilisée pour manipuler des fichiers, la classe **Graphics** utilisée pour manipuler des images de différents formats ou la classe **SmtpClient** utilisée pour envoyer du courrier avec le protocole SMTP.
Ce ne sont que quelques exemples, le Framework .NET contient environ 13000 de ces classes !

Vous pouvez obtenir une liste détaillée de ces classes avec leur documentation à cette adresse :
http://msdn2.microsoft.com/en-us/library/ms229335.aspx

Pour information, le Framework 2.0 contient 18619 types, 12909 classes, 401759 méthodes et 93105 propriétés publiques. Les Frameworks suivants en contiennent beaucoup plus. Evidemment, il faut connaître parfaitement toutes ces classes.

1.2 Rappel

Chaque classe contient des propriétés, des méthodes et des événements. Les propriétés, méthodes et événements de la classe sont appelés les membres de la classe.
Par exemple, voici quelques membres de la classe SmtpClient :

Propriétés :
Host : Le nom ou l'adresse IP du serveur mail
Port : Le numéro du port sur le serveur

Méthodes :
Send : Envoie un mail de façon synchrone
SendAsync : Envoie un mail de façon asynchrone

Évènements
SendCompleted : Se produit quand l'envoi d'un mail de façon asynchrone est termine.

1.3 Les Namespaces

Fort heureusement, Microsoft a divisé les classes du Framework en entités nommes « Namespaces » (espaces de nom en français). Un namespace est une sorte de « boîte » dans laquelle des classes ayant (plus ou moins) rapport entre elles sont regroupées. Par exemple, les classes utilisées pour accéder a un serveur SQL Server sont regroupées dans un namespace nomme « System.Data.SqlClient ». Autre exemple, toutes les classes utilisées par ASP.NET sont regroupes dans un namespace System.Web.

1.4 Les Assemblies

Une Assembly (assemblée en français) est un fichier DLL dans lequel les classes sont stockées. Par exemple, les classes utilisées par le namespace System.Web sont stockées dans un fichier System.Web.dll.

Pour faire l'analogie (en gros) avec Java, on pourrait dire que les classes du Framework .NET sont regroupées en namespaces, (les packages), eux-mêmes regroupés en assemblies (les .jar).

Tout comme en Java, pour pouvoir utiliser une classe, il faut configurer l'application (nous verrons comment plus tard) pour qu'elle référence la bonne assembly, puis importer le namespace de la classe.

1.5 CLR

La seconde partie du Framework .NET est le Common Language Runtime. Le Common Language Runtime (CLR pour les intimes) se charge de l'exécution de votre code.

Quand vous écrivez une application pour le Framework .NET avec un langage comme C# ou Visual Basic .NET, votre source n'est jamais compilé directement en binaire. Le compilateur C# ou VB convertit votre code dans un langage spécial nomme MSIL (Microsoft Intermediate Language, Langage intermédiaire Microsoft en français). Les familiers de Java sont ici en terrain connu. MSIL est une sorte d'assembleur orienté objet (!), mais à la différence d'un assembleur, il n'est pas spécifique à un CPU, il est indépendant de la plate forme sur laquelle il tourne.

Lorsque l'application est exécutée, le MSIL est compile alors (en temps réel) en binaire (donc ici spécifique a la plateforme) par le JITTER (Just-In-Time compilER).

En fait, toute l'application n'est pas compilée en binaire. Seules les méthodes appelées durant l'exécution sont compilées.

En réalité, le Framework .NET ne comprend qu'un seul langage : MSIL. Cependant, Microsoft va vous épargner cette tache ingrate en vous permettant de développer vos applications en Visual Basic .NET ou C#, les compilateurs MSIL de ces langages étant inclus dans le Framework .NET.

D'autres langages existent cependant (soit libres, soit commerciaux), on peut citer : Cobol, Eiffel, Fortran, Pascal, Python, etc.… (Développer un site "Web 2.0" en Cobol, avouez que ce n'est pas banal). Les premières versions du Framework étaient également livrées avec un compilateur Script (Le JavaScript Microsoft) et J# (Langage Java), mais il semble que ces langages ne soient plus soutenus dans les versions récentes.

Dans les faits, les développeurs utiliseront C# ou Visual Basic.NET. Les vrais développeurs utiliseront C#.

Un petit mot sur la compilation ASP.NET

Lorsque vous créez une page web ASP.NET, vous créez en fait le code source d'une classe .NET (Une instance de la classe System.Web.UI.Page). Le contenu entier de la page ASP.NET (incluant script ET contenu HTML) sont compilés dans une classe .NET

Quand vous appelez cette page via un browser, le Framework vérifie si une classe correspondent à cette page existe. Si elle n'existe pas, il compile automatiquement cette page dans une nouvelle classe et sauve la classe compilée (l'assembly) dans un dossier temporaire situé (pour le Framework 2.0) à cet emplacement :

`\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files`

Le 2.0.5xxx correspond à la version du Framework. Plusieurs frameworks peuvent coexister et être utilisés en même temps sur la même machine. Chaque Framework aura son propre dossier.

Lors de la requête suivante sur cette page, la classe ne sera pas recompilée. Même si vous arrêtez le serveur web et le relancez plus tard, le Framework utilisera cette classe compilée (sauf si vous modifiez le fichier source, dans ce cas, elle est recompilée).

Quand la classe est ajoutée au dossier temporaire ASP.NET, une dépendance est créée entre la classe et la page source ASP.NET. Si la page ASP.NET est modifiée, la classe correspondante est automatiquement effacée pour être recompilée la prochaine fois que la page sera appelée.

Ce procédé est nommé *compilation dynamique* et permet des performances qui vont bien au delà de ce que permettait ASP qui interprétait les pages à chaque requête.

1.6 Installation du framework ASP.NET

Il y a plusieurs façons d'installer et de développer avec le framework.

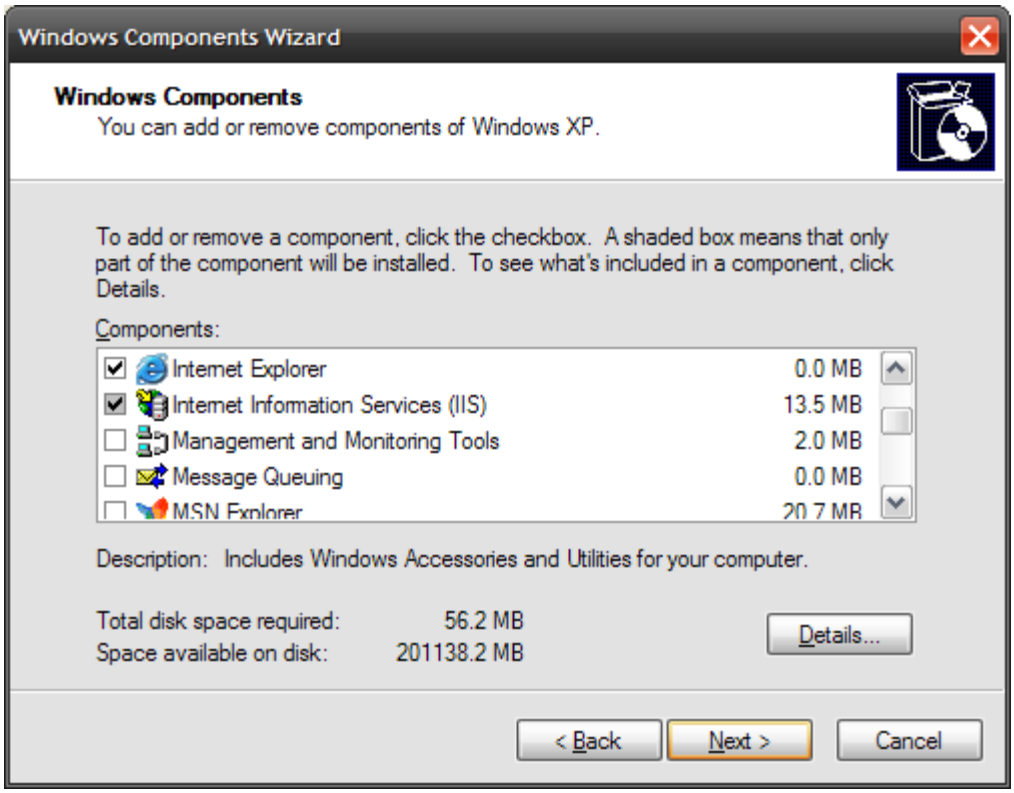
1. Le système d'exploitation

Nous utiliserons pour nos développements une machine fonctionnant sous Windows (2000 SP4, XP SP2-3, Vista ou server), Les exemples ici utilisent XP SP3. Je préconise l'utilisation de XP pro, cette version disposant en standard du serveur web IIS.

2. Activation d'IIS

IIS (Internet Information Server) est le serveur Web de Microsoft. Une version limitée (un seul site, une seule application) est livrée avec Windows XP Pro / Vista. Vous utiliserez IIS si vous n'utilisez pas l'environnement de développement de Microsoft qui intègre son propre serveur web ASP.NET (Cassini).
(Notez que si vous utilisez Windows XP ou Vista Home Edition, vous n'aurez pas le choix, IIS n'étant pas intégré au système.)

Si vous utilisez XP Pro, pour activer IIS : Panneau de Contrôle, Ajouter / Supprimer des programmes, cliquez sur Ajouter / Supprimer des composants Windows, puis cochez la case « Internet Information Services (IIS) »



(Note : Tous les exemples et les copies d'écrans ont été réalises sur une version US de Windows XP Pro).

3. Installez le SDK du Framework .NET

Le SDK (Software Development Kit) du Framework .NET installe toutes les assemblies sur la machine, la documentation, des exemples ainsi que les compilateurs en ligne de commande. Cette étape n'est pas indispensable si on utilise l'environnement de développement de Microsoft, mais je la recommande car elle contient des informations et des outils qu'on ne retrouve pas ailleurs.

On peut télécharger ce SDK à cette adresse :

http://www.microsoft.com/downloads/details.aspx?FamilyId=F26B1AA4-741A-433A-9BE5-FA919850BDBF&displaylang=en

4) L'environnement de développement

On peut utiliser les outils en ligne de commande et Notepad (pour les courageux ou les masochistes), ou un IDE libre comme SharpDevelop (**http://www.icsharpcode.net/OpenSource/SD/**) , mais je préconise l'installation de l'IDE de Microsoft : **Visual Studio Web Express 2008**.

Visual Studio Express 2008 est un excellent IDE gratuit qui existe en 4 versions : C#, Visual Basic, C++ et Web. Les 3 premières versions sont utilisées pour développer des applications Windows Desktop dans ces langages respectifs. La dernière version peut être utilisée en C# ou Visual Basic et sert à construire des applications web ASP.NET. C'est celle-ci qui nous intéresse.

Téléchargez-la à cette adresse et installez-la :

http://download.microsoft.com/download/0/a/c/0ac9dadd-0107-497e-a275-87fc2106941b/vnssetup.exe

(Les exemples donnés ici utilisent la version US de Visual Studio)

Apres l'avoir installé, vous devriez avoir quelque chose qui ressemble à ça :

2 La structure d'une application asp.net

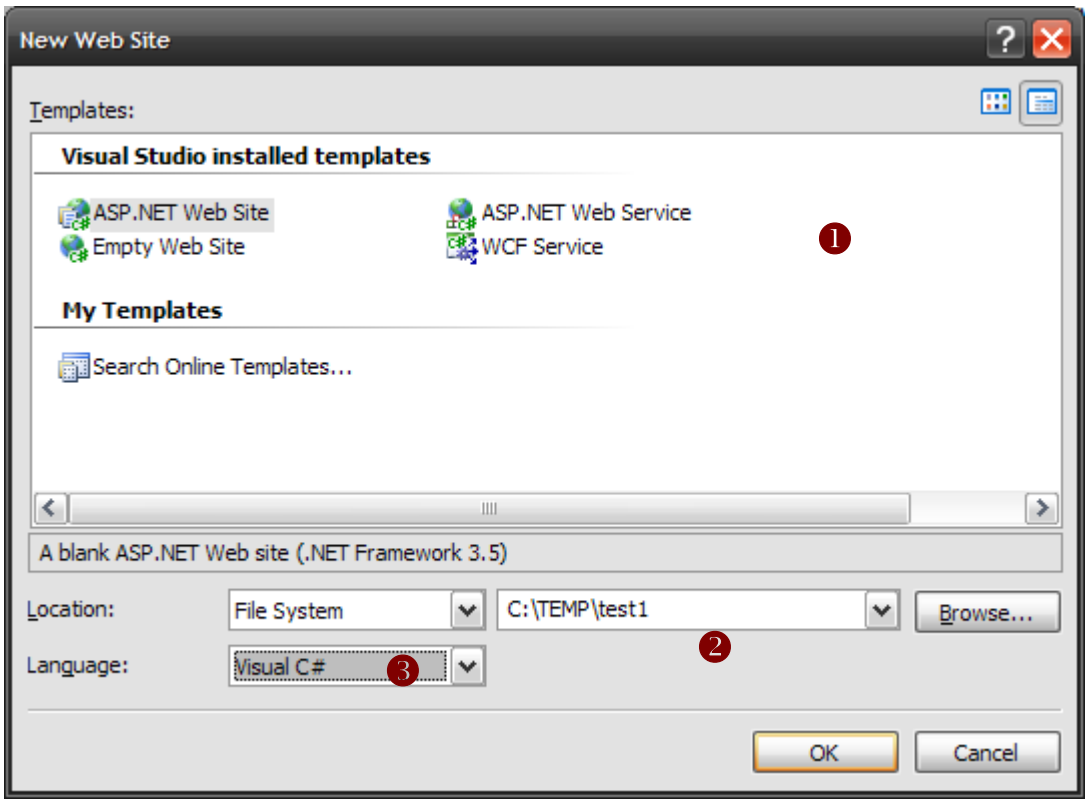
Nous allons commencer par un exemple complexe : Afficher une page web dans laquelle se trouvent un champ de saisie et un bouton. L'utilisateur saisit son nom dans le champ de saisie, clique sur le bouton et le message « Bonjour » suivi de son nom est affiche en retour.

Pour cet exemple nous allons utiliser des contrôles ASP.NET. Nous étudierons en détail ces contrôles plus loin, pour l'instant contentons-nous de les utiliser « bêtement »

Profitons-en pour faire un petit tour du propriétaire de Visual Studio Express Web 2008 (VS 2008 pour les inities).

Commençons par créer un nouveau projet :

`File -> New Web Site`

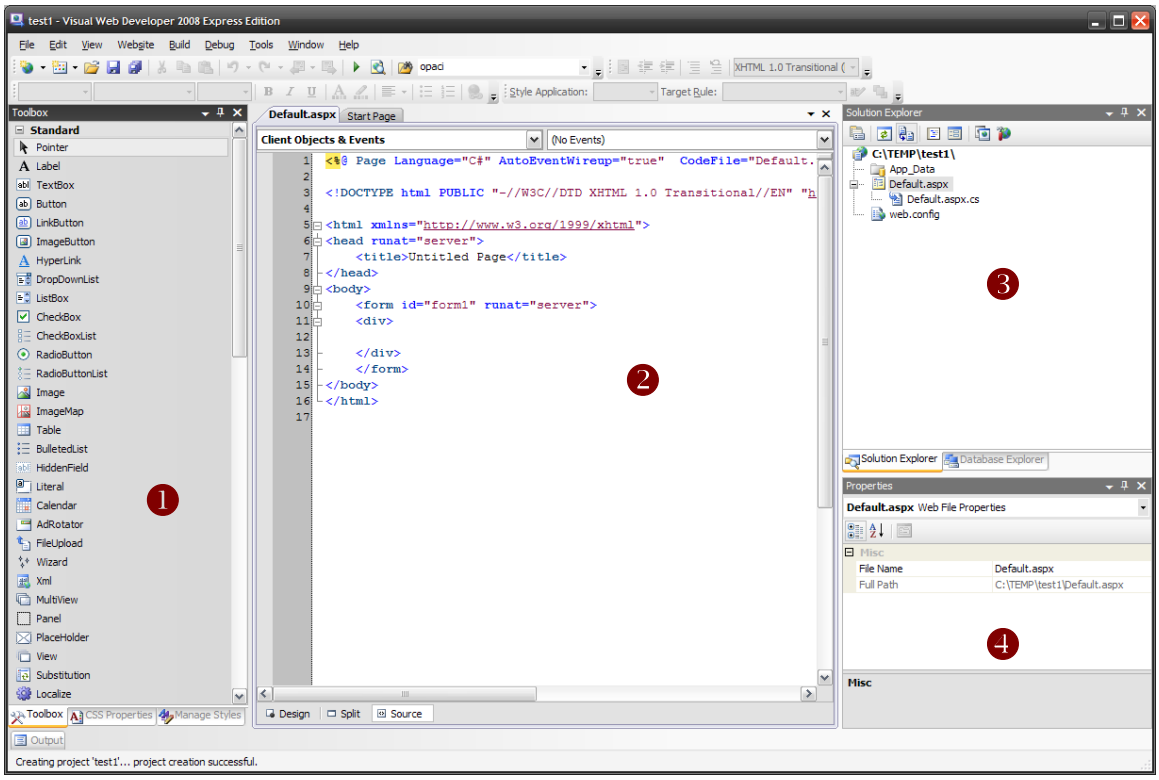


Qu’avons-nous ici ?

1) **Le type de projet** : En ce qui nous concerne, nous sélectionnerons ASP.NET Web Site

2) **L’emplacement** : on peut choisir soit le file system (un emplacement sur le disque dur, le serveur web utilisé sera alors Cassini, le serveur web intégré à VS 2008), soit HTTP (à choisir si on veut utiliser un site web avec IIS, nous verrons comment faire plus tard), soit FTP.
   Pour notre exemple, choisissez « file system » et un dossier de travail.

3) **On choisit le langage** : les gens sérieux choisirons donc Visual C#.



Le projet est crée et nous arrivons dans l’environnement de travail VS 2008. La page par défaut créée est **`Default.aspx`**.

4 zones importantes :

1) **La Toolbox** [Boîte à outils] : ici sont affichés tous les contrôles utilisables dans une page ASP.NET. C'est également dans la toolbox que seront affichés les contrôles provenant d'autres éditeurs que Microsoft. C'est également dans cet emplacement qu'on pourra éditer les propriétés CSS en cliquant sur l'onglet « CSS Properties »

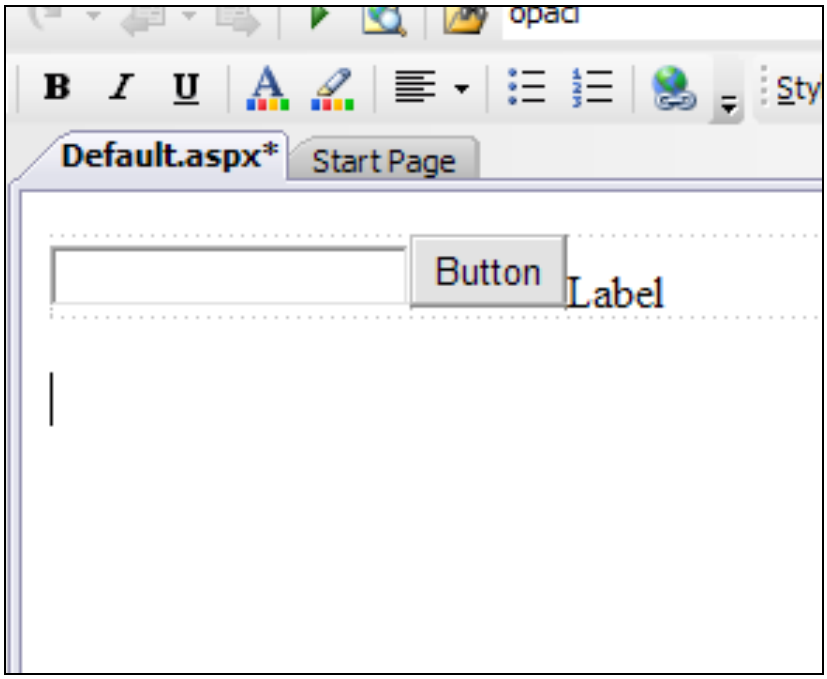
2) **L'éditeur de code**. Cet éditeur est wysiwyg. Vous pouvez y travailler en 3 modes : texte (vous voyez le source), Design (vous travaillez directement sur la page web) ou Split (vous voyez le source en haut et le rendu en bas)

3) **La structure du projet** : Les fichiers sont affichés ici sous forme arborescente. Un deuxième onglet (« Database Explorer ») est disponible, il permet de se connecter à une base de données et d'en afficher le contenu directement.

4) **Le panneau properties** : affiche en permanence les propriétés et événements du contrôle sélectionné. On peut évidemment modifier les propriétés ici. Toute modification est automatiquement reportée dans le source et dans l'écran Design et réciproquement.

Allons-y !

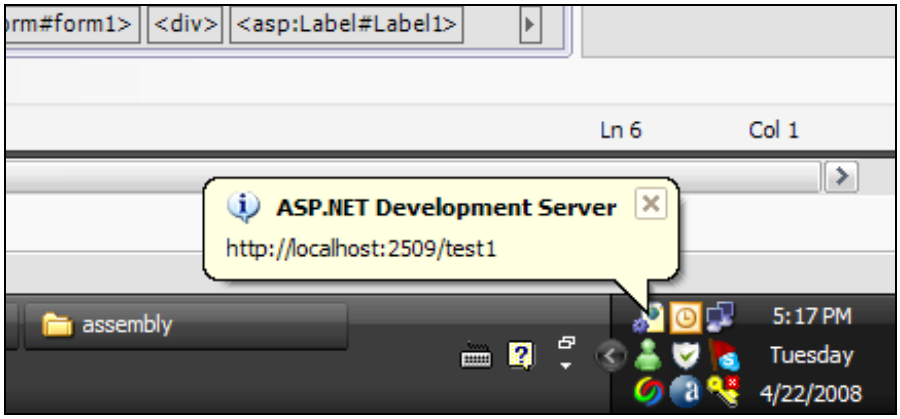
Mettons nous en mode « Design » et déplaçons dans la fenêtre un contrôle « TextBox », un contrôle « Button » et un contrôle « Label »



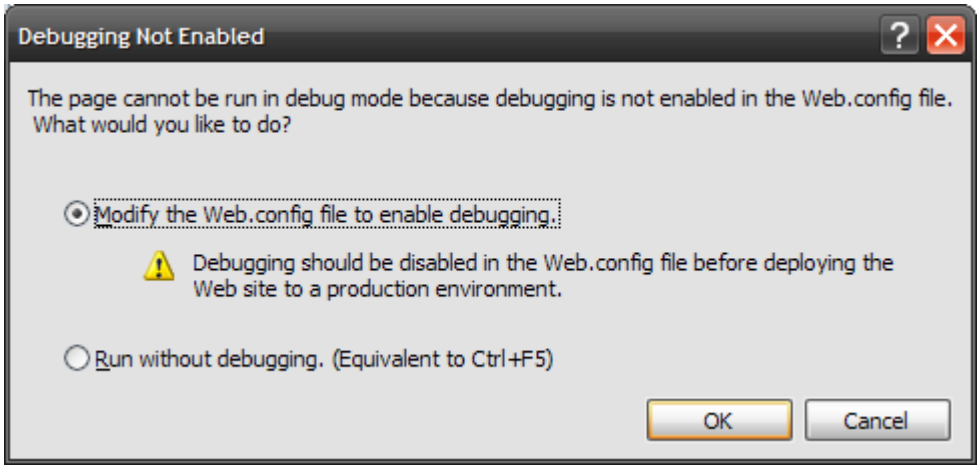
Belle page n'est-ce pas ?

On peut la visualiser tout de suite via le serveur web en cliquant sur l'icône « **`Start Debugging`** » (flèche verte)

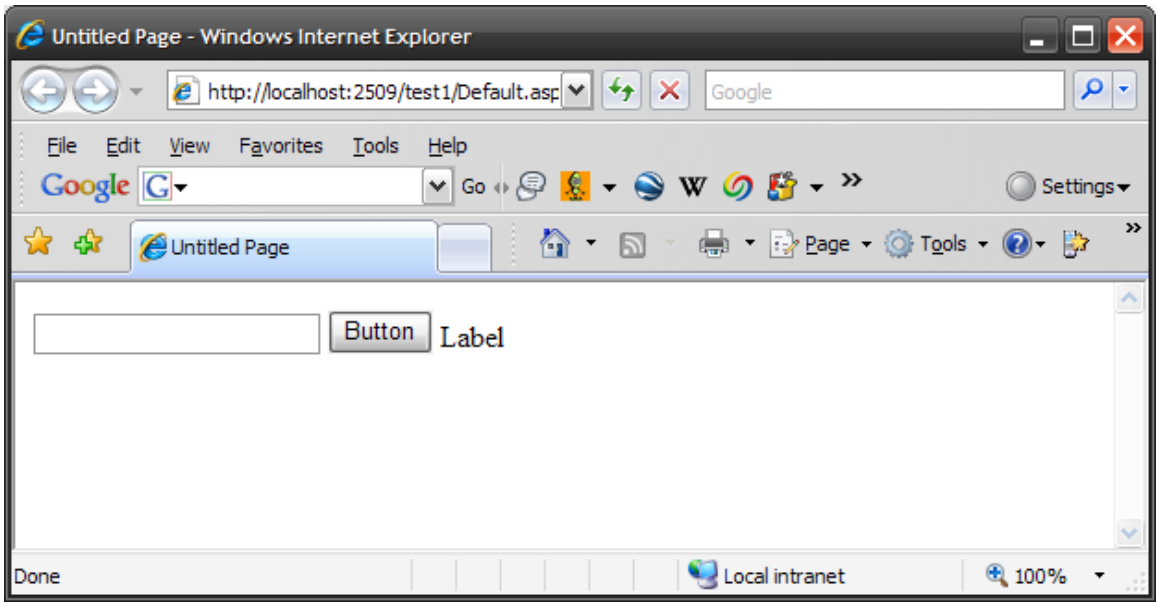
Première chose visible : Cassini, le serveur web ASP.NET intégré à VS 2008 est lance. Son activité est visible dans la barre de taches :



Deuxième chose visible : une alerte demande si on autorise le debugging. Par défaut il est désactivé, en cliquant sur OK, on va autoriser le debugging pour ce site (nous verrons comment debugger un peu plus loin).



La page s'affiche dans le navigateur par défaut du système (ici IE 7) :



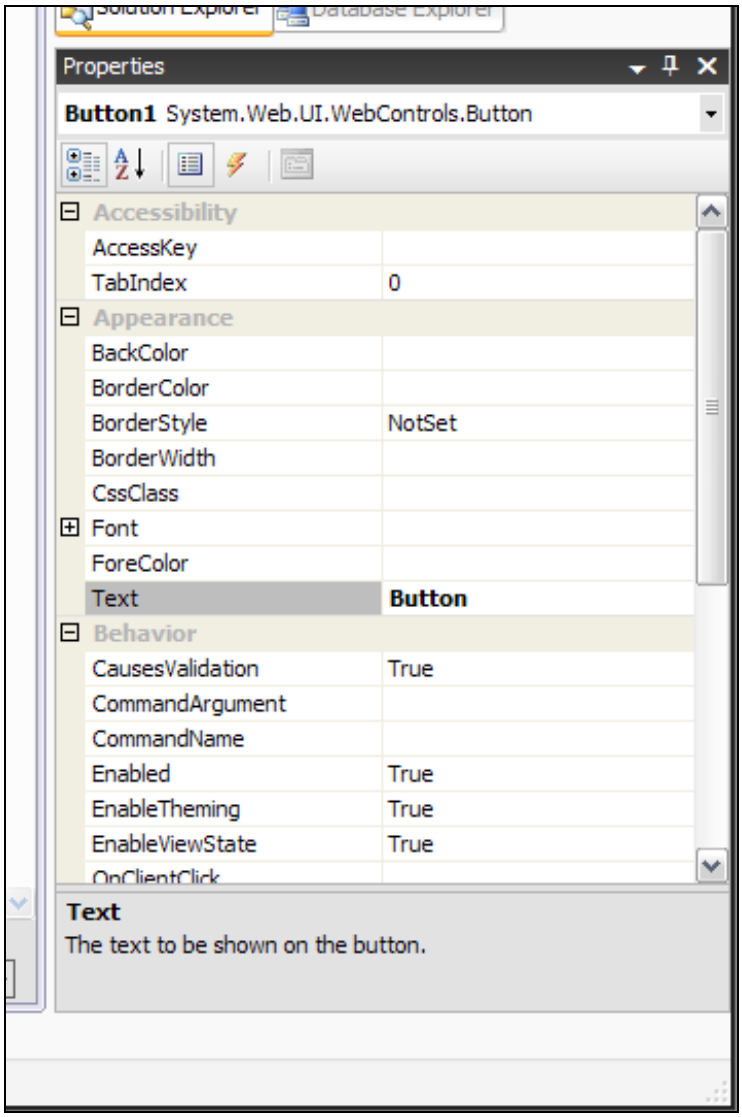
Evidemment, cliquer sur le bouton ne fait rien, va falloir bosser un peu.

Retour a VS.

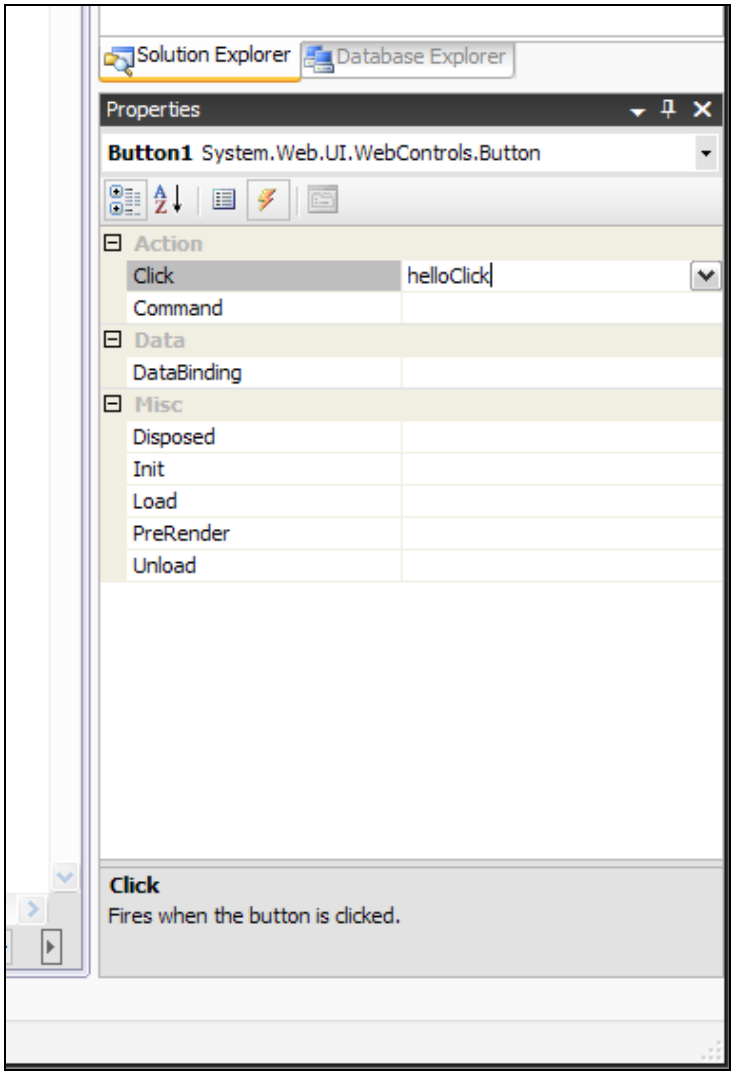
On arrête le debug (bouton carre bleu « Stop Debugging »).

Nous allons pour cet exemple travailler uniquement en mode design. Les pros (dont nous sommes évidemment) utilisent uniquement le mode Source, mais il faut bien commencer par quelque chose.

Cliquez donc sur le bouton. Ses propriétés s'affichent à droite



Si l'envie vous prend, vous pouvez modifier ici sa couleur ou sa police, intéressons nous a l'icône « Events » (l'éclair)



Sans entrer dans les détails pour l'instant, saisissons le nom d'une fonction (helloClick) pour la propriété Click, après avoir clique sur Entrée, le source de cette fonction est affichée automatiquement.

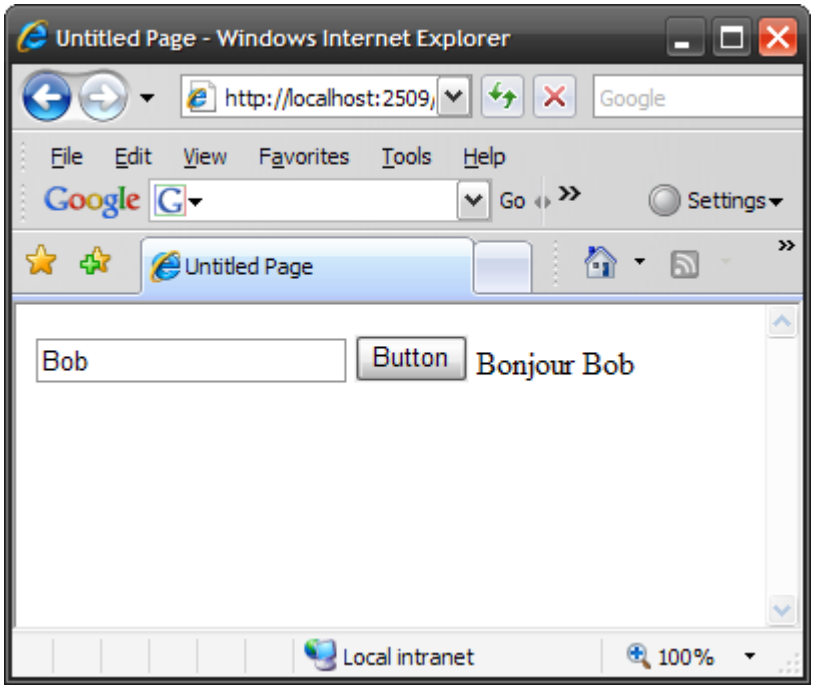
Tapons juste la ligne suivante :

Default.aspx.cs* | Start Page | Default.aspx*

_Default | helloClick(object sender, EventArgs e)

```
5   using System.Web;
6   using System.Web.Security;
7   using System.Web.UI;
8   using System.Web.UI.HtmlControls;
9   using System.Web.UI.WebControls;
10  using System.Web.UI.WebControls.WebParts;
11  using System.Xml.Linq;
12
13  public partial class _Default : System.Web.UI.Page
14  {
15      protected void Page_Load(object sender, EventArgs e)
16      {
17
18      }
19      protected void helloClick(object sender, EventArgs e)
20      {
21          Label1.Text = "Bonjour " + TextBox1.Text;
22      }
23  }
24
```

Et relançons l'application (bouton flèche verte « Debug »)



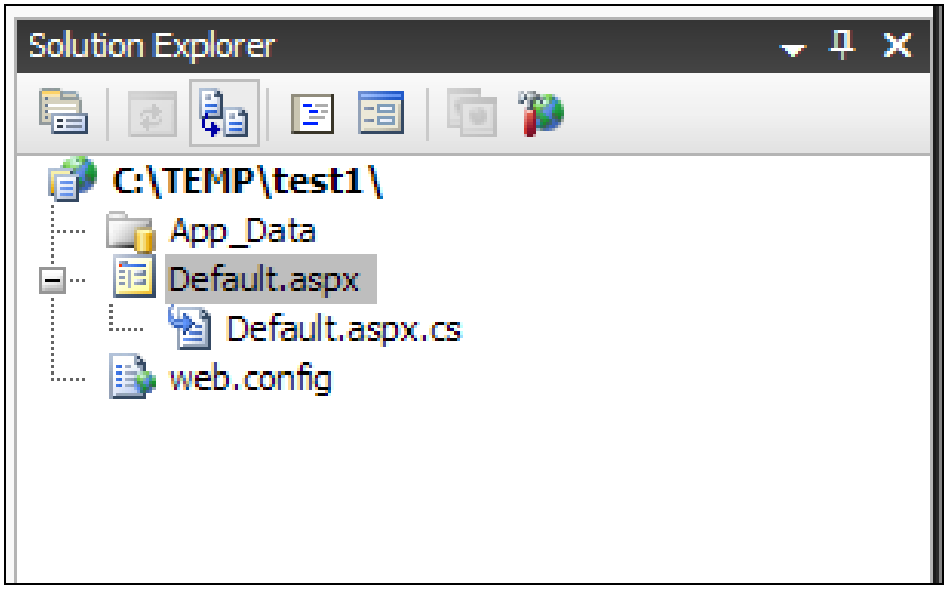
Saisissons un nom dans le champ de saisie et cliquons sur le bouton.

Ô Magie, le message « Bonjour » suivi du nom s'affiche.

Je vous laisse à votre émotion cinq minutes et je reviens pour voir un peu en détail ce qui s'est passe.

Regardons déjà dans le « solution explorer » à quoi ressemble notre projet

Ce projet ASP.NET est constitué de 3 fichiers (oublions le dossier App_Data pour l'instant).



Un **`fichier Default.aspx`** qui est le nom de la page web (c'est ce nom qui sera dans l'url tapée dans le browser). Cette page contient les contrôles ASP.NET, le code HTML, en gros toute la partie « présentation » de la page (ce n'est pas entièrement vrai, nous verrons pourquoi plus tard).

Un fichier **`Default.aspx.cs`** associé à la page asp.net. Ce fichier appelé fichier « Code Behind » contient le code associé à la page. C'est ici qu'on codera la logique nécessaire à la génération du contenu de la page. Ce code peut également être directement intégré dans le fichier ASPX (plus de fichier code behind dans ce cas). Ce sera souvent le cas dans nos exemples pour rendre les sources plus concis. Je préconise cependant de mettre la partie présentation et la partie code dans deux fichiers séparés.
Notez que l'extension du fichier est .CS (le contenu est en C#), ce sera .VB pour du code en Visual BASIC.

Enfin le fichier **`web.config`**. Ce fichier est un fichier XML qui va décrire comment doit se comporter le site web et quel est son paramétrage. Par défaut, VS 2008 préconfigure une partie de ce fichier. (Par exemple, il n'autorise pas le debugging par défaut, d'où le message affiché précédemment, le fait d'avoir autorisé le debugging à modifié un paramètre dans ce fichier).

On peut également créer des dossiers et y placer les pages ASPX et ASPX.CS associées pour créer une arborescence dans le site.

OK, allons voir un peu le contenu de ces fichiers :

Le fichier Default.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
      <asp:Button ID="Button1" runat="server" onclick="helloClick" Text="Button" />
            <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>

    </div>
    </form>
</body>
</html>
```

A première vue, ca ressemble beaucoup a un fichier HTML classique, deuxième vue aussi d'ailleurs.

Où est la différence ?

D'abord sur la 1ere ligne.

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>
```

Cette ligne configure les propriétés de la page. Il y en a d'autres que nous verrons plus loin, retenons `CodeFile` qui va indiquer le nom du fichier .cs associé et `Inherits` le nom de la classe associée (on la retrouvera dans le fichier .cs)

Suit le formulaire ASP.NET

```
<form id="form1" runat="server">
```

Vous noterez qu'une page ASP.NET est ***TOUJOURS*** insérée dans un formulaire. C'est le contenu de ce formulaire (tag `<FORM>`) qui est envoyée au serveur, les contrôles ASP.NET doivent donc être dans ce formulaire quoiqu'il arrive.

Ceci implique donc que vous ne pouvez en aucun cas placer un formulaire dans votre page ASP.NET. Avec ASP.NET oubliez donc l'existence même du tag `<FORM>`, vous n'y avez plus droit (et d'ailleurs, je ne vois pas ce que vous feriez avec…)

Suivent enfin les contrôles ASP.NET

```
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<asp:Button ID="Button1" runat="server" onclick="helloClick" Text="Button" />
<asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
```

Trois contrôles ASP.NET :

- un `TextBox` (champ de saisie) dont l'identifiant est « TextBox1 »
- un `Button` (Bouton) dont l'identifiant est « Button1 » et dont l'événement « onclick » appelle la fonction « helloClick »
- un `Label` (une zone de texte) dont l'identifiant est « Label1 »

Nous allons voir en détail tous ces contrôles et leur fonctionnement. Pour juste anticiper et dégrossir un peu, disons que les contrôles ASP.NET ont tous un attribut « `runat= "server"` » qui signifie que la génération de ce contrôle et ses événements se produiront sur le serveur. Le `onclick` du bouton ici n'est pas un événement onclick JavaScript qui se déclenche sur le client, c'est un événement qui va exécuter une fonction sur le serveur.

Puis le fichier Default.aspx.cs

```
using System;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
```

```
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void helloClick(object sender, EventArgs e)
    {
        Label1.Text = "Bonjour " + TextBox1.Text;
    }
}
```

Qu'y trouvons-nous ?

Une série de directives **`using`**. Cette instruction (l'équivalent du **`import`** en java) importe les namespaces utilises dans la page. En l'occurrence nous n'utilisons pas tous ces namespaces ici, mais VS 2008 importe par défaut tous les namespaces de l'assembly ASP.NET et Linq.

Suit la déclaration de la classe **`_Default`** (nom donne par défaut par VS). **`_Default`** est une page ASP.NET, elle hérite donc de la classe **`Page`**. (Souvenez vous que dans le fichier ASPX de la propriété **`Inherits="_Default"`** indiquant que la page ASPX allait utiliser cette classe).

Enfin, c'est une classe partielle, une partie de son code est automatiquement généré par VS : vous noterez qu'on peut utiliser directement les contrôles définis dans la page ASPX sans les avoir redeclarés ici (ce qu'on était obligé de faire avec le Framework 1.x), c'est parce qu'ils sont déclarés dans une autre partie de la classe, dans une dll non visible ici.

```
protected void helloClick(object sender, EventArgs e)
    {
        Label1.Text = "Bonjour " + TextBox1.Text;
    }
```

On y trouve la définition de la fonction décrite dans le fichier ASPX ici :

```
<asp:Button ID="Button1" runat="server" onclick="helloClick" Text="Button" />
```

Que fait cette fonction ? Elle modifie la propriété « Text » du contrôle **Label1** (le contenu de ce label) en y plaçant "Bonjour" + le contenu de la propriété Text du contrôle **TextBox1** (le champ de saisie).

Puis une autre fonction :

```
protected void Page_Load(object sender, EventArgs e)
{

}
```

Qu'est ce que c'est que cette fonction ?

Une page ASPX, lorsqu'elle est générée sur le serveur passe par plusieurs étapes (chargement en mémoire, traitement des évènements des contrôles, transformation des contrôles en contrôles html standards, etc.…). A chaque étape de la génération de cette page, un évènement se produit, **Page_Load** est l'un d'eux.

Il y a beaucoup d'évènements. On peut affecter du code à chacun de ces évènements.
Dans 95% (voire plus), on n'utilisera que l'évènement **Page_Load** qui se produit quand la page est chargée, avant que les évènements des contrôles aient eu lieu. C'est dans cette fonction qu'on va placer généralement tout le code qui va être utilisé dans la page et c'est pour ca que VS 2008 propose par défaut cette fonction vide, libre au développeur de la remplir.

Voyons un peu quelles fonctions sont appelées lors de la génération de la page :

- **Page_Init** : Appelé lorsque l'évènement Init se déclenche. Elle est appelée avant toute opération sur la page.
- **Page_Load** : Pendant cet événement, vous pouvez effectuer une série d'actions pour créer votre page ASP.NET pour la première fois ou répondre aux événements côté clients qui résultent d'une publication.
- **Page_DataBind** : L'événement **DataBind** se produit quand on veut lier des données à la page (nous verrons cela plus tard).
- **Page_PreRender** : L'événement **PreRender** est déclenché juste avant que l'état d'affichage soit enregistré et que les contrôles soient affichés. Vous pouvez utiliser cet événement pour effectuer toute opération de dernière minute sur vos contrôles.
- **Page_Unload** : Une fois qu'une page a été affichée, l'événement **Page_Unload** se déclenche. Cet événement est approprié à la réalisation d'un travail final de nettoyage, tel que le nettoyage des connexions ouvertes de base de données, la suppression d'objets ou la fermeture de fichiers ouverts.

Il y en a d'autres mais leur utilisation est très spécifique.

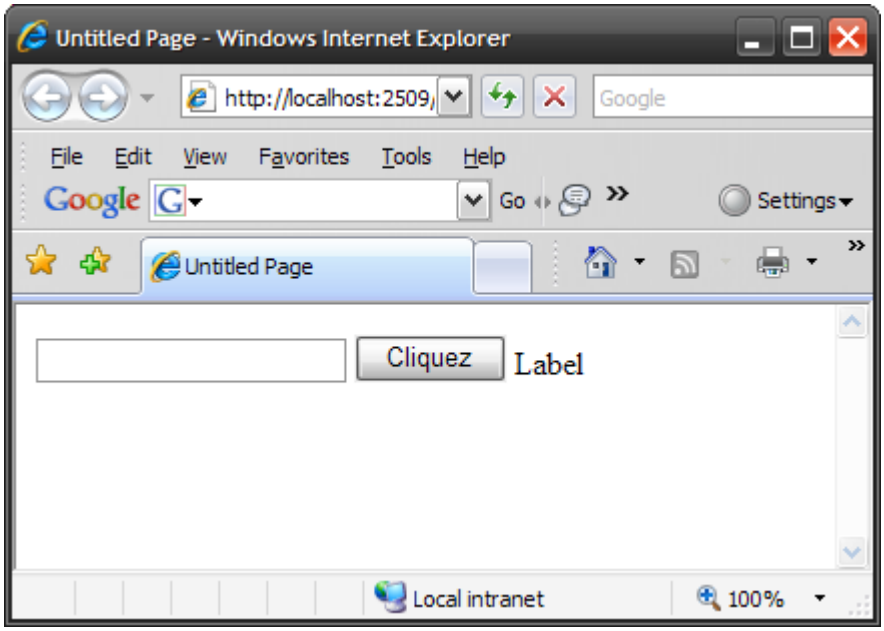
Note sur Page_Load

Cette fonction est donc appelée à chaque fois que la page est générée. Que ce soit la première fois qu'elle est appelée par le browser ou suite à un évènement sur la page (un click sur le bouton va rappeler la page pour exécuter le code associé au click).

La propriété `IsPostBack` de la classe `Page` (dont dérive notre page) permet de savoir si la page est générée suite a sa création initiale ou suite a un évènement sur cette page (on dit qu'elle est rappelée ou qu'un postback a eu lieu).

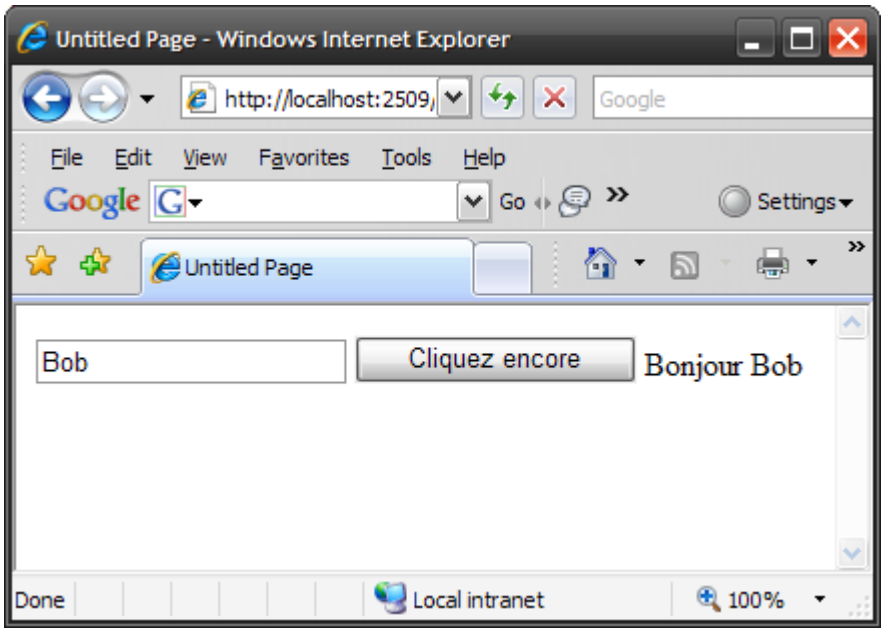
Modifions donc notre code :

```
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack) { Button1.Text = " Cliquez "; }
        else { Button1.Text = " Cliquez encore "; }
    }
```



La première fois qu'on appelle cette page, le texte du bouton est « Cliquez »

Puis, a chaque fois que l'utilisateur clique sur le bouton, la fonction « **`helloClick`** » est appelée sur le serveur, donc l'évènement **`Load`** est déclenché, donc la fonction **`Page_Load`** est appelée, en testant la propriété **`IsPostBack`**, on saura si la fonction est appelée lors de la création initiale de la page ou si c'est suite a un évènement sur la page.



Détail sur les évènements :

Lorsqu'une fonction est appelée par un évènement, elle passe deux paramètres :

```
protected void Page_Load(object sender, EventArgs e)
```

Le premier, **`sender`** est l'objet sur lequel l'évènement s'est produit, ici l'objet **`Page`**, dans notre fonction **`helloClick`**, se serait le bouton **`Button1`**. Ce paramètre est utile si on fait pointer plusieurs évènements sur la même fonction. Le deuxième paramètre va varier en fonction de l'évènement. Ici il n'y a pas d'information supplémentaire, le paramètre est donc un objet **`EventArgs`** (paramètres standards d'un évènement).

Un click sur un contrôle **`ImageButton`** (un bouton qui est en fait une image) passera en paramètre a la place d'EventArgs un objet **`ImageClickEventArgs`** qui contient des informations supplémentaires (par exemple les coordonnées x, y de la souris lors du click).

En fait, la page ASP.NET est un contrôle ASP.NET comme un autre, et à ce titre, tous les contrôles ASP.NET ont le même cycle de vie et les mêmes évènements (Init, Load, Prerender, Dispose, …). Vous pouvez donc associer une fonction à un de ces évènements pour un contrôle précis de la page. Dans la pratique, pour éviter qu'on ne sache plus trop où et quand s'exécute le code, on se limitera aux évènements de la page.

Un peu confus ? Ca va aller en s'améliorant, ne vous inquiétez pas…

Avant de passer aux choses sérieuses, étudions notre dernier fichier, web.xml.

Pour les habitues des serveurs d'applications J2EE, on peut dire que web.xml est « l'équivalent » du fichier server.xml.

Il contient la configuration de l'application web, on peut y placer ses propres paramètres (plutôt que de les éparpiller dans des fichiers textes, ou dans la base de registres), on peut y placer également les chaines de connexions sur les bases de données, etc.… Nous n'allons pas passer en revue l'intégralité du paramétrage de ce fichier, nous y reviendrons au coup par coup selon les besoins. Regardons juste ce que nous aurons éventuellement à modifier dans ce fichier.

AppSettings

Dans cette section, nous pourrons (éventuellement) placer des paramètres utilises par notre application :

```
<appSettings>
      <add key="cle" value="value"/>
</appSettings>
```

Et les lire avec la une méthode de la classe ConfigurationSettings.

Ex: `String val = ConfigurationSettings.AppSettings("cle");`

connectionStrings

```
<connectionStrings>
   <add name="cnx" connectionString=" localhost;Initial
Catalog=appservicesdb;Integrated
   Security=True" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Dans cet exemple, nous définissons une connexion sur une base de données SQL Server. Nous verrons plus loin comment utiliser ces connexions.

Assemblies

```
<assemblies>
   <add assembly="System.Core, Version=3.5.0.0, Culture=neutral,
    PublicKeyToken=B77A5C561934E089"/>
   <add assembly="System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31BF3856AD364E35"/>
   <add assembly="System.Data.DataSetExtensions, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=B77A5C561934E089"/>
   <add assembly="System.Xml.Linq, Version=3.5.0.0, Culture=neutral,
    PublicKeyToken=B77A5C561934E089"/>
</assemblies>
```

Ici sont définies les assemblies qui vont être utilisées dans l'application web. On pourra ensuite importer avec la directive using les namespaces de ces assemblies dont nous avons besoin. Notez que si l'assembly n'est pas référencée ici, l'import dans la page aspx provoquera une erreur. Toujours pour faire une analogie avec Java, on peut dire que cette section est en fait le classpath.

Notez également que l'assembly est définie par son nom suivis de paramètres optionnels comme la version (il est possible de référencer plusieurs versions de la même assembly et d'utiliser des namespaces provenant de plusieurs versions), la culture (ici neutral, une assembly peut être liée a une culture particulière, par exemple en-US ou fr, elle sera utilisée en fonction de la culture sélectionnée dans la page) et enfin une clef publique unique qui permet de vérifier que la dll correspondante n'a pas été remplacée par une autre a l'insu de l'administrateur du site.

A priori, vous ne devriez pas, pour commencer, avoir besoin de bricoler d'autres sections de ce fichier.

3 Les contrôles ASP.NET

Entrons donc dans le vif du sujet. Les contrôles ASP.NET (ceux disponibles dans la toolbox et dont nous avons vu trois spécimen) sont la clef de voute d'ASP.NET. Un contrôle ASP.NET est une classe qui s'exécute sur le serveur et renvoie en retour un contenu à afficher dans le navigateur.

Le Framework ASP.NET inclut en standard environ 70 contrôles, et des sociétés tierces en proposent des centaines d'autres ! Ces contrôles vous permettent quasiment tout, de la simple ligne de texte a la bannière de publicité en passant par le traitement de texte intégré ou le tableau affichant les données d'une base.

Ces contrôles sont divisés en 8 groupes :

- **Les contrôles standards** : Ils vous permettent d'utiliser les éléments de base des formulaires comme les boutons, les champs de saisie, les labels.
- **Les contrôles de validation** : Ils vous permettent de valider un formulaire avant de soumettre ses données au serveur.
- **Les contrôles "riches"** : Ils affichent des objets plus complexes comme les calendriers, les wizards, les boutons pour uploader des fichiers, les bannières de publicités ou les éditeurs de texte
- **Les contrôles de données** : Ils vous permettent de travailler avec des données comme par exemples des données issues d'une base de données. Vous pouvez utiliser ces contrôles pour afficher, modifier, supprimer ou ajouter des données.
- **Les contrôles de navigation** : Ils vous permettent d'afficher des éléments de navigation dans le site comme les menus, les arbres ou les breadcrumbs
- **Les contrôles de login** : Ils affichent des boites de login avec nom d'utilisateur et mot de passe, vous permettent de modifier les mots de passe ou d'afficher des formulaire d'enregistrement
- **Enfin, les contrôles HTML**. Ils vous permettent de transformer n'importe quel tag HTML standard en control ASP.NET

3.1 Viewstate

Avant de voir les contrôles en détail, abordons une caractéristique très importante et intéressante d'ASP.NET : le viewstate.

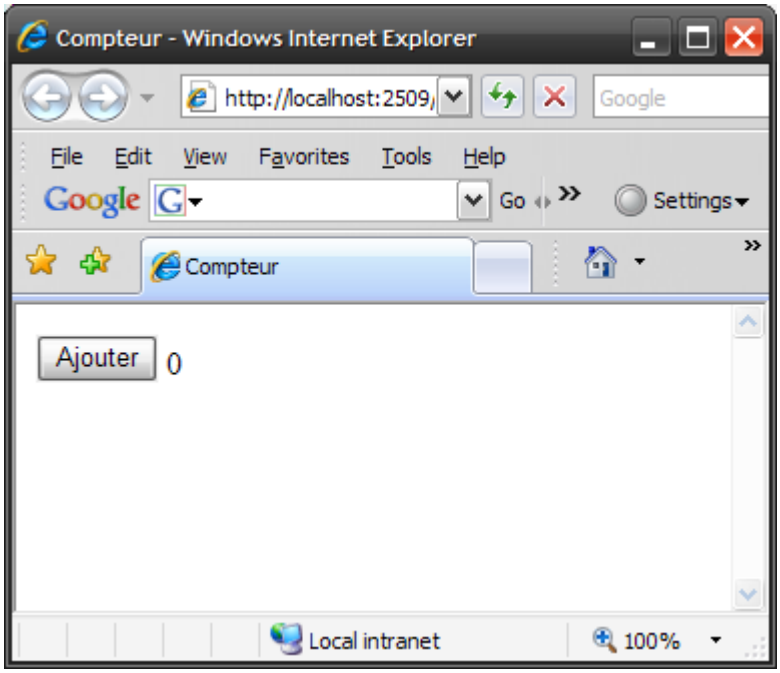
Le protocole http, comme vous devez le savoir et un protocole « stateless » (sans état) : Chaque fois que vous demandez l'affichage d'une page d'un site, pour le serveur web, vous êtes une personne complètement différente et nouvelle.

Le Framework ASP.NET cependant permet de dépasser cette limitation du protocole http. Par exemple, si vous assignez une valeur à la propriété Text d'un label, comme nous l'avons fait, le contrôle conservera cette valeur durant toutes les requêtes qui seront faites sur cette page.

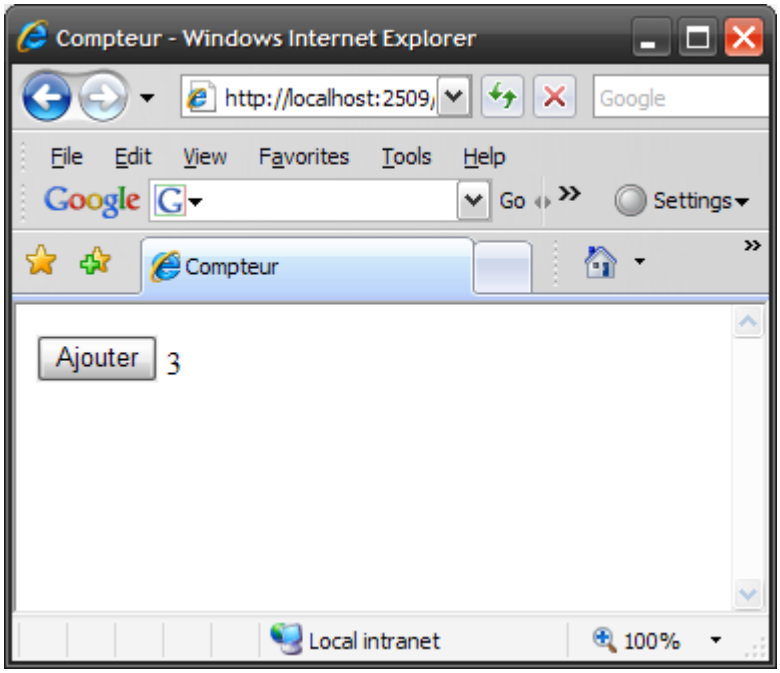
Pour mieux comprendre, prenons cet exemple : (Attention, pour des raisons de concision, le code qui se trouve normalement dans la page aspx.cs est ici directement dans la page aspx. Pour cela, il est juste place dans un tag <script runat= »server »> …
</script>

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

    protected void btnAjout_Click(object sender, EventArgs e)
    {
    lblCompteur.Text = (Int32.Parse(lblCompteur.Text) + 1).ToString();
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Compteur</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
       <asp:Button id="btnAjout" Text="Ajouter" OnClick="btnAjout_Click"
Runat="server"/>
       <asp:Label id="lblCompteur" Text="0" Runat="server" />
    </div>
    </form>
</body>
</html>
```

Que fait cette page ? Elle est composée d'un bouton et d'un label initialise au départ avec le contenu « 0 ». Une fonction btnAjout_Click est associée a l'évènement « onclick » sur le bouton. Cette fonction récupère le contenu du label, le convertit en nombre, puis ajoute 1 et finalement le resauve dans le label.



Au bout de 3 clicks :



Normalement, du moins avec des technologies web " classiques " , le contenu du label devrait rester figé sur « 0 », ou alors, il faudrait envisager quelque usine à gaz de bon aloi à base de variable de session ou de cookie pour aller conserver la valeur qu'on aurait été récupérer préalablement dans un champ caché ou dans un paramètre de l'url, bref du bricolage comme il y a 15 ans.

Pour permettre de conserver la configuration de tous les contrôles ASP.NET de la page entre deux postbacks, le Framework ASP.NET utilise une astuce appelée ViewState. Si vous affichez la source de la page dans le browser (Menu View -> Source)

Au milieu du code html, on trouve un champ cache qui ressemble a ca :

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKLTc2ODE1OTYxNw9kFgICBA9kFgICAw8PFgIeBFRleHQFATNkZGSgjDnu3yfn1lTqTxMVQdG0
hRMx1w==" />
```

Ce champ caché contient la valeur de la propriété Text du contrôle Label (et les valeurs de toutes les autres propriétés de tous les contrôles ASP.NET de la page). Quand une page est renvoyée au serveur, le Framework ASP.NET récupère cette chaine, la décode et réinitialise toutes les propriétés des contrôles ASP.NET de la page. De cette façon, ASP.NET conserve l'état des contrôles entre deux postbacks.

Par défaut, le viewstate est activé pour chaque contrôle. Si par exemple, vous changez la couleur de fond d'un calendrier, la nouvelle couleur est conservée durant les postbacks suivants. Si vous sélectionnez une ligne dans une liste déroulante, la ligne sélectionnée le restera car toutes les valeurs de ces propriétés sont stockées dans le viewstate.

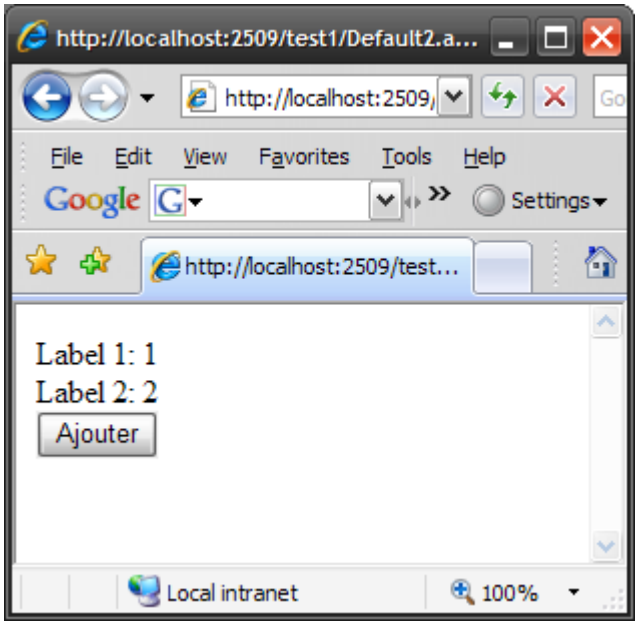
Le viewstate est un bon truc, mais finalement, parfois ca peut devenir embêtant. Le champ caché __VIEWSTATE peut devenir très gros. Mettre beaucoup de données dans le viewstate peut ralentir l'affichage de la page et générer un trafic important (le viewstate fait des aller retour entre le client et le serveur a chaque postback).

Mais les concepteurs ont pensé à tout. Chaque contrôle possède une propriété nommée EnableViewState. Par défaut est elle initialisée a true. Si vous mettez la valeur de cette propriété a false, le viewstate est désactive pour ce contrôle. L'état de ce contrôle ne sera pas conserve entre deux postbacks.

Un petit exemple pour voir ce que ça donne :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

protected void btnAjouter_Click(object sender, EventArgs e)
    {
    Label1.Text = (Int32.Parse(Label1.Text) + 1).ToString();
    Label2.Text = (Int32.Parse(Label2.Text) + 1).ToString();
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
   <body>
    <form id="form1" runat="server">
      Label 1:
      <asp:Label id="Label1" EnableViewState="false" Text="0" Runat="server" />
      <br />
      Label 2:
      <asp:Label id="Label2" Text="0" Runat="server" />
      <br />
      <asp:Button id="btnAdd" Text="Ajouter" OnClick="btnAjouter_Click" Runat="server"
/>
     </form>
   </body>
</html>
```

Le premier label, sur lequel on a désactivé le viewstate, n'est pas incrémenté, la valeur précédente est perdue entre deux postbacks.



Maintenant, a vous de voir quand vous pouvez vous permettre de désactiver le viewstate (si par exemple, vous régénérez à chaque postback le contenu d'un contrôle, il est inutile de générer son viewstate qui ne sera, de toute façon, pas utilisé).

Note : Le Framework 2.0 a introduit une nouvelle fonctionnalité appelée *control state*. Le control state est similaire au viewstate excepté qu'il ne conserve que les informations importantes. Par exemple, le contrôle GridView (qui est un tableau de données, nous l'étudierons plus tard) utilise le control state pour stocker la ligne sélectionnée. Même si vous désactivez le viewstate, ce contrôle se souviendra de la ligne sélectionnée.

Mais il y a mieux. En plus de l'état des contrôles, vous pouvez stocker vos propres valeurs dans le viewstate. Un objet « ViewState » est accessible dans la page. C'est une sorte de hashmap (table de hachage en français)

Exemple : `ViewState["foo"] = "bar"`

Vous venez de stocker « `bar` » dans le champ cache __VIEWSTATE de la page. Lors du prochain postback, vous pourrez le récupérer en faisant :

```
String fooValue = ViewState["foo"].toString() ;
```

Vous pouvez, à priori, stocker n'importe quoi dans le viewstate, les objets étant sérialisés, mais attention, vous augmentez au passage la taille de vos pages et le volume de données qui transitent sur le réseau. Une alternative à ca peut être d'utiliser des variables de session.

3.2 Session, Application

La variable de session s'utilise de la même façon que la hashmap ViewState. Sauf qu'au lieu d'utiliser "ViewState", vous utilisez "Session" :

Exemple : `Session["foo"] = "bar"`

Puis, plus tard :

```
String fooValue = Session["foo"].toString() ;
```

Quelle différence avec ViewState ? Dans ce cas, la variable "foo" est stockée en mémoire sur le serveur et non plus dans la page (au lieu d'encombrer votre page, vous encombrez la mémoire du serveur). L'avantage de la variable session, c'est qu'elle est accessible sur toutes les pages que visite l'utilisateur. Vous pouvez ainsi conserver des informations de page en page sans avoir à les repasser en paramètre à chaque fois.

Mais qu'est ce qu'une session ? En gros, c'est le temps entre le moment ou le visiteur arrive sur une page du site et 20 minutes après qu'il n'ait plus donné signe de vie. A chaque fois que le visiteur fait une action, un compteur est remis à zéro sur le serveur. Si le compteur atteint 20 minutes, ses variables de session sont effacées de la mémoire du serveur.

Vous aurez donc compris que les variables de sessions ne sont donc accessibles qu'aux pages du site visitées par un seul utilisateur. Si néanmoins, vous voulez faire partager le contenu d'une variable à tous les utilisateurs du site, vous utiliserez la hashmap "Application" (qui s'utilise de la même façon) :

Exemple : `Application["foo"] = "bar"`

Puis, plus tard :

```
String fooValue = Application["foo"].toString() ;
```

Pour résumer :

- Application : commun à toutes les pages vues par tous les utilisateurs (disparaît après l'arrêt de l'application, c'est-à-dire, quand on arrête le site web dans IIS)

- Session : commun à toutes les pages vues par un utilisateur (disparaît après 20 minute d'inactivité)

- ViewState : présent sur une seule page pour un seul utilisateur

4 Les contrôles en détail

Bien, maintenant, nous allons pouvoir étudier un peu plus en détail les différents contrôles qu'ASP.NET propose en standard. N'oubliez pas qu'il existe des centaines d'autres contrôles (gratuits ou payants) proposes par des sociétés tierces.

4.1 Afficher des informations

ASP.NET possède deux contrôles que vous pouvez utiliser pour afficher du texte sur une page. Le contrôle Label et le contrôle Literal. Le contrôle Literal affiche du texte brut, alors que le contrôle Label propose des possibilités de formatage plus évoluées.

4.1.1 Le contrôle Label

Si vous voulez modifier dynamiquement du texte affiche dans une page, Label est le contrôle qu'il vous faut. Toute chaine de caractère assignée à la propriété Text du contrôle sera affichée par le label quand la page sera générée. Vous pouvez afficher du texte simple mais également du code HTML. Il y a deux façons d'assigner du texte à ce contrôle :

```
<asp:Label id="Label1" runat="server" Text="texte à afficher" />
```

Ou

```
<asp:Label id="Label1" runat="server"><b>texte à afficher</b></asp:Label>
```

Dans le deuxième cas, vous pouvez utiliser, comme vous le voyez, du code HTML.

Dans votre code, vous pouvez assigner ce que vous voulez :

```
Label1.Text = "<b>texte à afficher</b>";
```

Par défaut, le contenu à affiche est généré dans un tag `<span>`. Si vous regardez le source de la page générée, vous trouverez :

```
<span id="Label1"><b>texte à afficher</b></span>
```

Ce contrôle possède d'autres propriétés (toutes modifiables, comme n'importe quel contrôle ASP.NET dans votre code behind). Je ne cite ici que les propriétés les plus importantes :

BackColor	Couleur du fond du texte
BorderColor	Couleur de la bordure
BorderStyle	Style css de la bordure (None, Dotted, Solid, …)
BorderWidth	Epaisseur de la bordure

CssClass	Classe CSS associée
Font	Police utilisée
ForeColor	Couleur du texte
Style	Style CSS associe
TooTip	Contenu de l'attribut title

Comme avec tous les contrôles ASP.NET, ces propriétés sont converties en style css associe au contrôle. C'est pour cela que je préconise, plutôt que d'initialiser toutes ces propriétés pour chaque contrôle, d'utiliser plutôt une classe CSS définie dans un fichier de style séparé.

4.1.2 Le contrôle Literal

Le contrôle Literal est similaire au contrôle Label. Vous pouvez utiliser le contrôle Literal pour afficher du texte ou du HTML. Cependant, contrairement au contrôle Label, le contrôle Literal ne génère pas son contenu dans un tag , il est affiche brut.

Ex :

```
<asp:Literal id="Literal1" runat="server"><b>ceci est du texte</b></asp:Literal>
```

Générera

```
<b>ceci est du texte</b>
```

Si vous modifiez la propriété Mode avec la valeur encode :

```
<asp:Literal id="Literal1" runat="server" Mode="encode"> <b>ceci est du
texte</b></asp:Literal>
```

Générera

```
&lt;b&gt;tagada&lt;/b&gt;
```

Les caractères spéciaux sont encodés en html, au lieu d'afficher du texte en gras, le texte "ceci est du texte" sera affiché.

4.2 Saisir des données

ASP.NET propose plusieurs contrôles pour saisir des données. Nous allons étudier dans cette section les contrôles TextBox, CheckBox et RadioButton. Ces contrôles correspondent aux types INPUT standard du HTML.

4.2.1 Le contrôle TextBox

Le contrôle TextBox est utilise pour afficher 3 types de champs de saisie en fonction de la valeur de la propriété TextMode :

SingleLine	Affiche un champ de saisie sur une ligne
MultiLine	Affiche une zone de saisie multi ligne (TextArea en HTML)
Password	Affiche un champ de saisie sur une ligne dans lequel le texte tape est caché

En plus de cette propriété, le contrôle TextBox dispose des propriétés suivantes (comme d'habitude, je ne cite pas toutes les propriétés. Je vous laisse le soin de les découvrir vous-même, je ne cite que celles que je juge les plus utiles).

AutoPostBack	Déclenche automatiquement un postback quand le texte du contrôle est modifié
Columns	Spécifie le nombre de colonnes à afficher
Enabled	Active / Désactive le contrôle
MaxLength	Nombre de caractères maximum que l'utilisateur peut saisir
ReadOnly	Empêche l'utilisateur de modifier le contenu du champ
Rows	Nombre de lignes à afficher
Wrap	Autorise le retour à la ligne automatique quand le TextMode est initialise a MultiLine

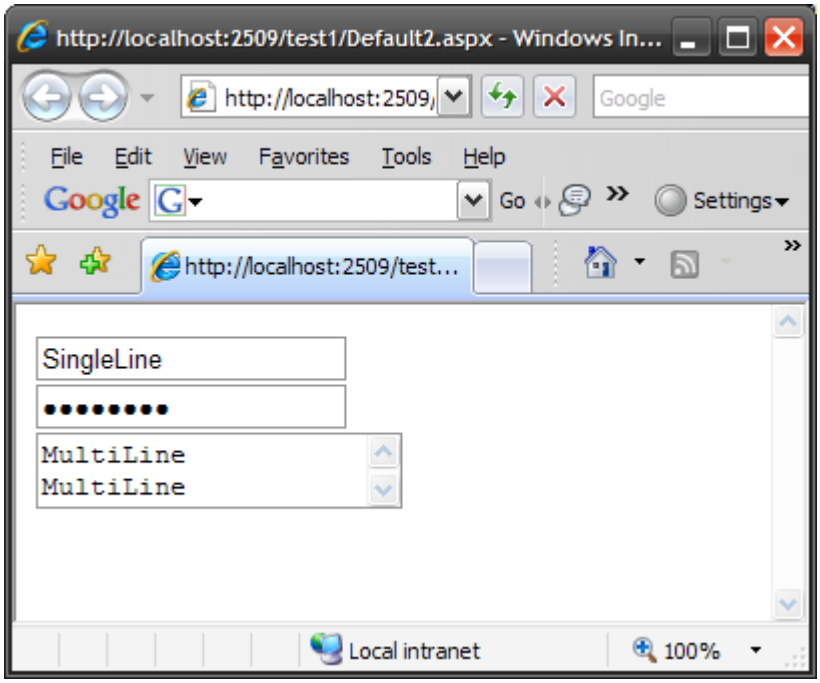
Nous l'avons vu précédemment, un contrôle ASP.NET est une classe, a ce titre, en plus des propriétés, il dispose de méthodes et d'évènements. Parmi les méthodes intéressantes, citons :

Focus	Place le focus par défaut sur ce contrôle

Et parmi les évènements :

TextChanged	Déclenche un événement sur le serveur quand le contenu du champ a change

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
    <form id="form1" runat="server">
        <asp:TextBox id="txtUtilisateur" TextMode="SingleLine" Runat="server" />
        <br />
        <asp:TextBox id="txtPassword" TextMode="Password" Runat="server" />
        <br />
        <asp:TextBox id="txtCommentaires" TextMode="MultiLine" Runat="server" />
    </form>
</body>
</html>
```



4.2.2 Le contrôle CheckBox

Le contrôle CheckBox affiche une case à cocher. Notez que le contrôle CheckBox a une propriété Text qui permet d'associer un texte à la case a cocher. Si vous utilisez cette propriété, le texte génère sera place dans un tag **`<label>`**

Quelques propriétés intéressantes du CheckBox

AutoPostBack	Déclenche automatiquement un postback quand le texte du contrôle est modifie
Checked	Coche par défaut la case
Enabled	Active ou désactive le contrôle
Text	Affiche un texte associe a la case a cocher
TextAlign	Indique la position du texte (Left ou Right : à gauche ou à droite de la case)

Comme le TextBox, le CheckBox a une méthode Focus et un évènement CheckedChanged, déclenche sur le serveur quand le case a été cochée ou décochée.

4.2.3 Le contrôle RadioButton

Le contrôle RadioButton s'utilise toujours dans un groupe. Un seul bouton radio peut être sélectionné à la fois dans ce groupe (Le fait de sélectionner un bouton désélectionne les autres).

Checked	Sélectionne par défaut le bouton
Enabled	Active ou désactive le contrôle
GroupName	Nom du groupe du bouton (tous les boutons du groupe ont la même valeur)
Text	Texte associé au bouton
TextAlign	Position de ce texte

Ce contrôle possède les mêmes méthodes et évènements que la case à cocher (CheckBox)

4.3 Envoyer un formulaire

Une fois les informations saisies, il s'agit de les envoyer sur le serveur (faire le postback). Nous avons vu plus haut qu'une page ASP.NET est en fait un grand formulaire. Ce formulaire est posté sur le serveur soit lorsqu'un événement se déclenche sur la page web, soit lorsque l'utilisateur l'envoie lui-même via un bouton de soumission (submit button in english).

Il existe trois sortes de submit buttons : Le bouton (contrôle **Button**), le bouton Image (**ImageButton**) et le bouton lien (**LinkButton**). Ces trois boutons font exactement la même chose (Envoyer le formulaire vers le serveur), mais ont un aspect différent.

4.3.1 Le contrôle Button

Le contrôle button génère un objet html du type `<INPUT TYPE="BUTTON">`, un bouton classique.

Parmi les propriétés intéressantes :

Enabled	Active / désactive le contrôle
OnClientClick	Indiquez ici le nom d'une fonction JavaScript qui sera exécutée sur le client lorsque le bouton sera cliqué. Attention, il s'agit bien ici d'une fonction client. Si la fonction renvoie false, l'envoi du formulaire sur le serveur sera annulé.
CommandName	Voir plus bas
CommandArgument	Voir plus bas
PostBackUrl	Indique l'url d'une page ASP.NET vers laquelle doit se faire le postback (nous allons voir ca en détail plus bas)
Text	Texte du bouton

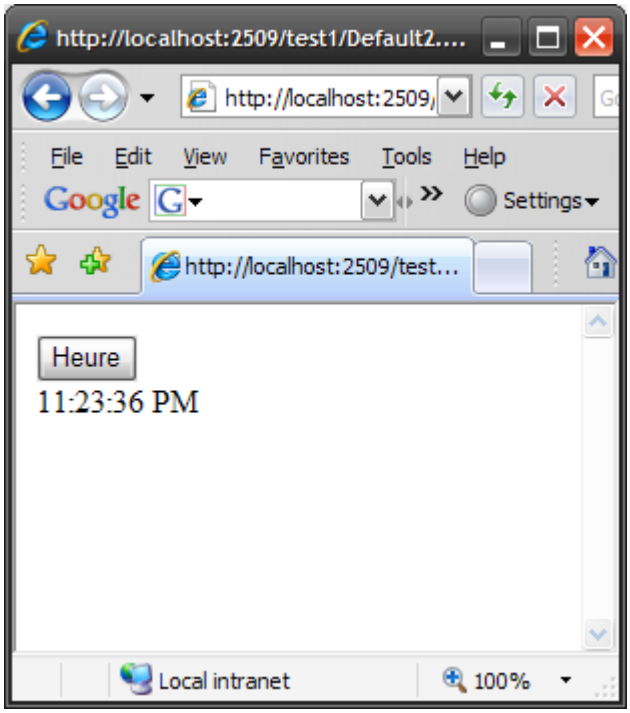
Parmi les évènements intéressants

Click	Exécute la fonction associée sur le serveur quand le bouton est clique
Command	Idem, sauf que passe en paramètre le contenu des propriétés CommandName et CommandArgument. Cet événement est utilise a la place de Click quand par exemple plusieurs boutons utilisent la même fonction lors du click. Ces valeurs peuvent être utilisées alors pour déterminer un traitement particulier.

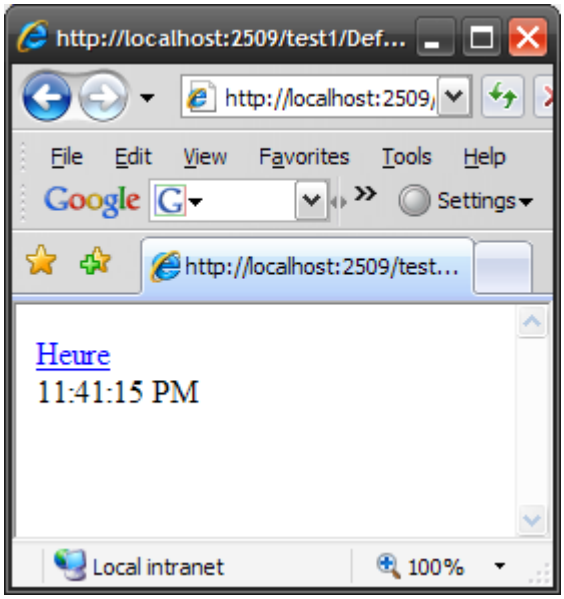
Ex : Affichage de l'heure

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        lblTime.Text = DateTime.Now.ToString("T");
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
  <form id="form1" runat="server">
     <asp:Button id="btnSubmit" Text="Heure" OnClick="btnSubmit_Click" Runat="server"
/>
     <br />
     <asp:Label id="lblTime" Runat="server" />
  </form>
</body>

</html>
```

Notez bien que le `<script runat="server">` s'exécute sur le serveur et que le `<script type="text/javascript">` s'execute sur le client



Le LinkButton est exactement comme un Button, la seule différence c'est qu'au lieu d'être affiche sous la forme d'un bouton, il est affiche sous la forme d'un lien.



4.3.2 Le contrôle ImageButton

Comme les deux contrôles précédents, le contrôle ImageButton permet d'envoyer le formulaire sur le serveur, cependant, le contrôle ImageButton affiche toujours une image.

On retrouve les mêmes propriétés que précédemment, avec, en plus :

AlternateText	Texte a afficher si l'image n'est pas visible (navigateur texte, pour non voyant, etc…)
ImageUrl	URL de l'image a afficher
DescriptionUrl	Utilisé par les navigateurs pour non voyants : entrez ici l'url d'une page qui va décrire l'image
ImageAlign	Alignement de l'image par rapport aux autres éléments HTML (are AbsBottom, AbsMiddle, Baseline, Bottom, Left, Middle, NotSet, Right, TextTop, et Top)

On retrouve les mêmes méthodes et évènements que les contrôles précédents.

Notez que, contrairement aux deux contrôles précédents, la fonction appelée par l'événement Click ou Command ne reçoit pas un paramètre **`EventArgs`** mais un paramètre **`ImageClickEventArgs`** qui indique la coordonnée (x, y) a laquelle s'est produit le click dans l'image (propriétés X et Y de lmageClickEventArgs).
Vous pouvez utiliser cette coordonnée pour faire une sorte d'image map sur le serveur.

4.3.3 Utilisation des scripts client avec les contrôles boutons

Ces trois contrôles ont donc une propriété OnClientClick. Vous pouvez utiliser cette propriété pour exécuter du code sur le client quand le bouton est clique.

Exemple :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
    protected void btnDetruit_Click(object sender, EventArgs e)
    {
        lbl.Text = "Le site a ete detruit !";
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
  <form id="form1" runat="server">
    <asp:Button id="btnDetruit" Text="Detruire le site"
     OnClick="btnDetruit_Click"
     OnClientClick="return confirm('Etes-vous sur ?');" Runat="server" />
    <asp:Label id="lbl" Runat="server" />
  </form>
</body>

</html>
```

Si la fonction javascript appelle cote client renvoie false, le postback est annule et le formulaire n'est pas renvoyé sur le serveur.

4.4 Postback "croisés"

Par défaut, quand vous cliquez sur un contrôle Button, le postback est fait sur la page contenant le bouton (ce qui devrait toujours être le cas, ainsi, tout le code associé à une page se trouve associé à cette page). Cependant, vous pouvez parfois avoir besoin que le postback se fasse vers une autre page. Pour ce faire, vous pouvez utiliser la propriété PostBackUrl (propriété qu'on retrouve dans les trois contrôles). Cette propriété contient l'URL d'une page ASP.NET, page sur laquelle se fera le postback.

Dans l'exemple qui suit, la première page contient un champ de saisie (un TextBox), l'utilisateur saisit du texte et valide, le postback se fait alors sur la deuxième page, on affiche sur cette deuxième page le texte saisi dans la première :

1ere page

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
    <form id="form1" runat="server">
      <asp:Label id="lbl" Text="Texte:" Runat="server" />
      <asp:TextBox id="texte" Runat="server" />
      <asp:Button id="btn" Text="OK" PostBackUrl="page2.aspx" Runat="server" />
    </form>
</body>
</html>
```

2eme page

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
    void Page_Load()
    {
        if (PreviousPage != null)
        {
            TextBox txt = (TextBox)PreviousPage.FindControl("texte");
            lbl.Text = String.Format("Texte : {0}", txt.Text);
        }
    }
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
    <form id="form1" runat="server">
        <asp:Label id="lbl" Runat="server" />
    </form>
</body>
</html>
```

Que fait-on dans cette deuxième page ? La propriété **`PreviousPage`** de l'objet Page (notre page donc) est utilise pour savoir si le **`Page_Load`** est consécutif a un postback effectue sur une autre page.

Si c'est le cas, on utilise la méthode **`FindControl`** de la page. Cette méthode cherche un objet dans la page a partir de son identifiant. L'objet générique est alors caste sur un TextBox d'où on extrait le contenu de la propriété Text.

Pas très simple n'est-ce pas ? On peut faire plus simple et plus propre. En exposant dans une propriété de la page précédente le texte saisi par l'utilisateur :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<script runat="server">
public string Texte
{
    get { return texte.Text; }
}
</script>

<body>
    <form id="form1" runat="server">
      <asp:Label id="lbl" Text="Texte:" Runat="server" />
      <asp:TextBox id="texte" Runat="server" />
      <asp:Button id="btn" Text="OK" PostBackUrl="page2.aspx" Runat="server" />
    </form>
</body>
</html>
```

Sur l'autre page, le code de la méthode Page_Load s'en trouve simplifie :

```
<script runat="server">
void Page_Load()
{
    if (Page.PreviousPage != null)
    {
        lbl.Text = String.Format("Texte : {0}", PreviousPage.Texte);
    }
}
</script>
```

Mieux non ?

4.4.1 Spécifier le bouton par défaut

Vous pouvez, si vous avez plusieurs boutons dans la page, spécifier lequel sera appelé par défaut si l'utilisateur appuie sur la touche "Entrée". Pour cela, renseignez la propriété "**`DefaultButton`**" du formulaire de la page ASP.NET avec le nom du bouton en question.

4.5 Afficher des images

Le Framework ASP.NET dispose de deux contrôles pour afficher des images : le contrôle Image et le contrôle ImageMap. Le contrôle Image affiche bêtement une image, le contrôle ImageMap vous permet de créer des images map (désolé, mais je ne sais pas trop comment traduire ca en français)

4.5.1 Le contrôle Image.

Il possède les mêmes propriétés du contrôle ImageButton.

4.5.2 Le contrôle ImageMap

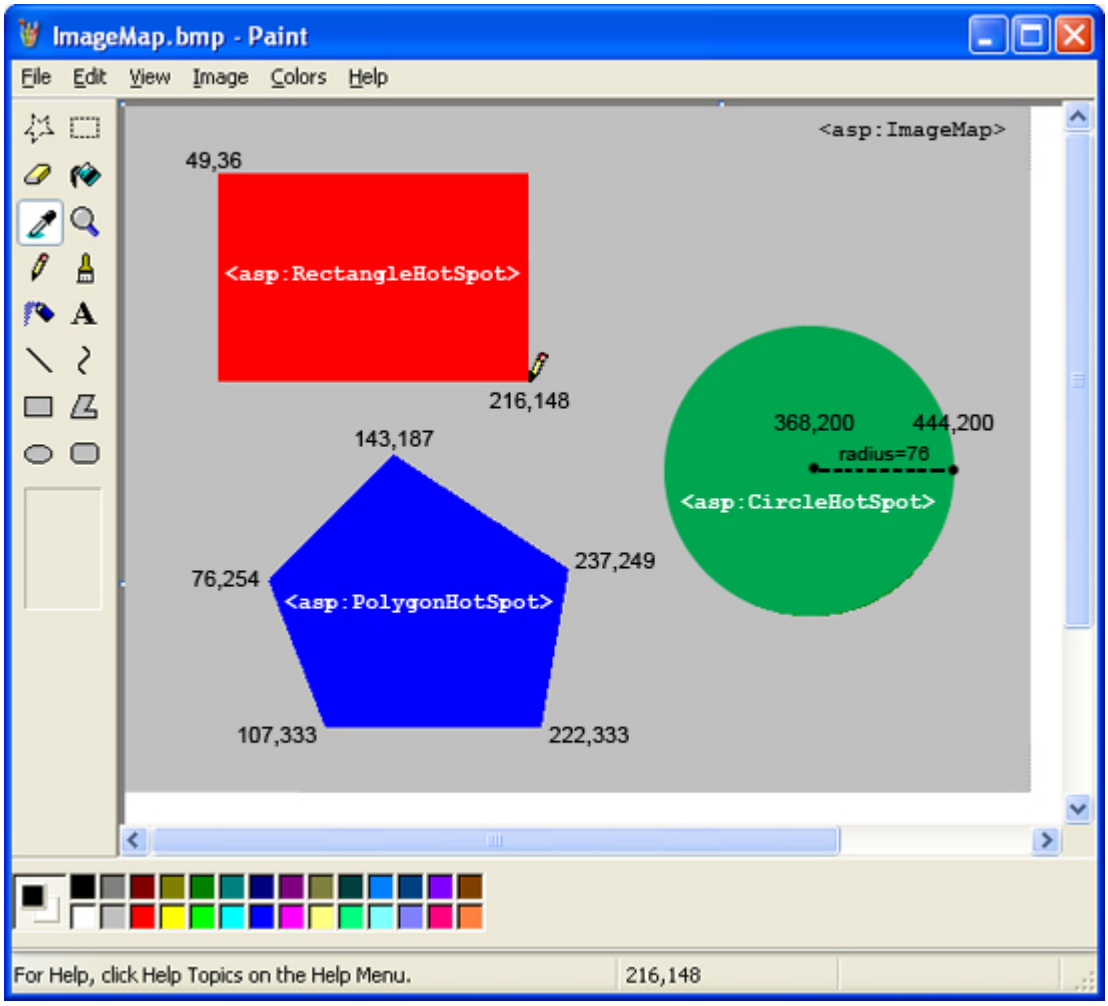
Ce contrôle vous permet de créer des images map. Les images map sont des images sur lesquelles on a défini plusieurs zones. Vous pouvez alors effectuer des traitements particuliers en fonction de la zone cliquée. Par exemple, vous pouvez utiliser une image map pour faire une barre de navigation. Dans ce type d'utilisation, cliquer sur une des zones enverra l'utilisateur vers une partie ou une autre du site.

Un contrôle ImageMap est composé d'instances de classes HotSpot.
Un HotSpot définit une zone cliquable dans une image map. Le Framework ASP.NET propose trois types de HotSpots :

- **CircleHotSpot :** Définit une zone circulaire dans une image
- **PolygonHotSpot**—Definit une zone irrégulière
- **RectangleHotSpot**—Définit une zone rectangulaire.

Exemple : une page permettant d'appeler des moteurs de recherche :

Voici les coordonnées des différents éléments de la page :



Le contrôle ASP.NET sera défini :

```
<asp:ImageMap Runat="Server"
  ImageUrl="ImageMap.jpg"
  AlternateText="Web search image map"
  HotSpotMode="Navigate">

  <asp:RectangleHotSpot
    AlternateText="Go to Google"
    Left="49"
    Top="36"
    Right="216"
    Bottom="148"
    NavigateUrl="http://www.google.com"
    Target="_blank"/>

  <asp:CircleHotSpot
    AlternateText="Go to AltaVista"
    X="368"
    Y="200"
    Radius="76"
    NavigateUrl="http://www.altavista.com"
    Target="_blank"/>

  <asp:PolygonHotSpot
    AlternateText="Go to Yahoo"
    Coordinates="143,187, 237,249, 222,333, 107,333, 76,254"
    NavigateUrl="http://www.yahoo.com"
    Target="_blank"/>

</asp:ImageMap>
```

4.6 Le container

Le contrôle Panel vous permet de regrouper plusieurs contrôles. Notez donc que si vous cachez le panel (en mettant la propriété Visible a false), vous cachez du même coup les contrôles se trouvant dedans.

Parmi les propriétés intéressantes du Panel :

Direction	Vous permet de définir dans quel sens le texte va être affiche (**LeftToRight** : de gauche à droite, **RightToLeft** : de droite à gauche)
GroupingText	Par défaut, ASP.NET convertit le panel en un tag html `<DIV>`. Si vous mettez une chaine de caractères dans cette propriété, ASP.NET générera a la place un `<fieldset>`. Le fieldset est une sorte de div encadré par un cadre au bords arrondis. Il peut contenir un titre (affiche avec le tag html `<legend>`), c'est le contenu de cette propriété qui est le titre du fieldset.
HorizontalAlign	Indique comment est aligné le contenu du panel (Auto : centre, Both : texte justifié à droite et à gauche, Left: justifié à gauche, Right : justifié à droite)
ScrollBars	Affiche des barres de défilement autour du contenu du panel (**Auto** : n'affiche des barres de défilement que si le contenu du panel est plus grand que le panel lui-même, **Both** : affiche deux barres de defilement (vertical et horizontal) , même si le contenu du panel est plus petit que le panel lui-même, **Horizontal** et **Vertical** : affiche toujours une barre de défilement horizontale et verticale quelque soit la taille du contenu et **None** : n'affiche jamais de barre de defilement : si le contenu du panel est plus grand que le panel lui-même, il ne sera jamais visible.

4.7 Le contrôle Hyperlink

Le contrôle Hyperlink est en fait un lien `<a>` html. Contrairement au contrôle LinkButton qui, rappelons le est un bouton ayant un aspect de lien, le contrôle Hyperlink ne fait aucun postback vers le serveur.

Enabled	Active / désactive le contrôle
ImageUrl	Spécifie une image pour le lien (le lien est en fait une image)
NavigateUrl	URL du lien
Target	Spécifie la fenêtre de destination, le cas échéant, la frame ou iframe.

5 Les contrôles de validation

OK, avec les contrôles que vous avez vus précédemment, vous êtes capable de couvrir de façon basique la plupart des besoins des sites web. Mais ASP.NET propose d'autres contrôles. Ceux-ci, plus complexes et plus riches, vous permettent de vous affranchir de développements fastidieux.

Parmi ces contrôles, on trouve les contrôles de validation. Les contrôles de validation vérifient les données saisies par l'utilisateur et si ces données comportent une erreur, ne correspondent pas a un format précis, comportent des valeurs erronées ou en contradictions entre elles, affichent un message d'erreur.

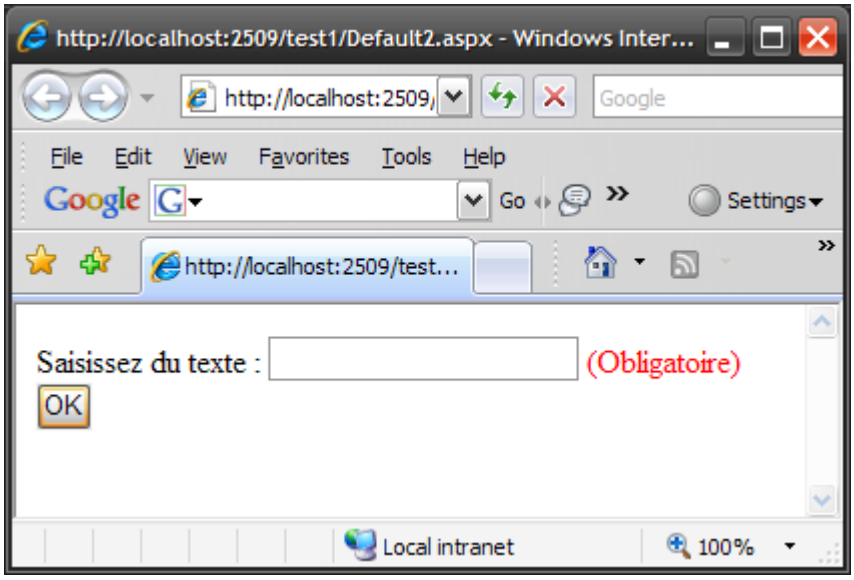
Le Framework ASP.NET 3.5 inclut six contrôles de validation :

- **RequiredFieldValidator :** Oblige l'utilisateur à saisir une donnée dans un champ de saisie
- **RangeValidator** : Vérifie que la donnée saisie se situe bien dans une certaine fourchette
- **CompareValidator** : Compare la donnée saisie avec une valeur ou vérifie le type de la donnée saisie
- **RegularExpressionValidator** : Vérifie que la donnée saisie est valide en fonction en l'évaluant avec une expression régulière
- **CustomValidator** : Valide la donnée saisie grâce a une fonction que vous pouvez définir vous-mêmes
- **ValidationSummary** : Enfin, affiche un résume de toutes les erreurs de validation de la page

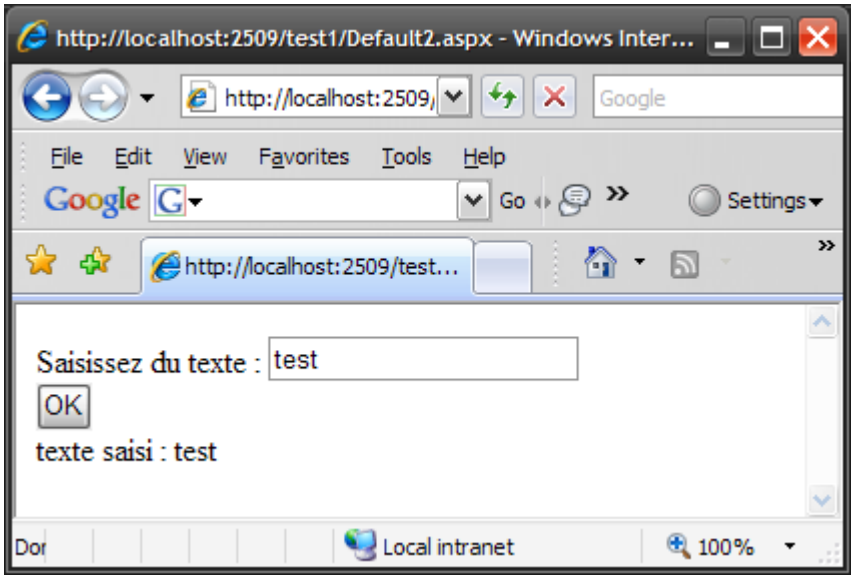
Vous pouvez associer un contrôle de validation a n'importe quel contrôle de formulaire. Par exemple, si vous voulez obliger l'utilisateur a saisir une valeur dans un contrôle TextBox, vous pouvez associer un contrôle RequiredFieldValidator a ce contrôle TextBox.

Petit exemple simple :

Si l'utilisateur clique sur OK sans avoir saisi de texte, le message "(Obligatoire)" s'affiche a droite du champ ou la saisie est obligatoire.



Si l'utilisateur clique sur OK après avoir saisi du texte, aucun message d'erreur ne s'affiche et le texte saisie est réaffiché sur la page :



```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

void btnSubmit_Click(Object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        lbl.Text = "texte saisi : " + txtBox.Text;
    }
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
   <form id="form1" runat="server">
      <asp:Label id="lbl1" Text="Saisissez du texte :"
       AssociatedControlID="txtBox"  Runat="server" />
      <asp:TextBox id="txtBox" Runat="server" />
      <asp:RequiredFieldValidator id="val" ControlToValidate="txtBox"
       Text="(Obligatoire)" Runat="server" />
      <br />
     <asp:Button id="btn" Text="OK" OnClick="btnSubmit_Click" Runat="server" />
     <br />
     <asp:Label id="lbl" Runat="server" />
  </form>
</body>
</html>
```

Chaque contrôle **RequiredFieldValidator** est associé avec un contrôle spécifique en utilisant sa propriété **ControlToValidate** (on indique dans cette propriété quel contrôle il va valider).
Vous noterez au passage que vous pouvez créer plusieurs contrôles de validation et associer à tous ces contrôles le même contrôle a valider.

Vous noterez aussi (notez bien, c'est important) que dans le page load, il y a un test :

```
    if (Page.IsValid)
```

Lorsque la validation se fait cote serveur (nous allons voir ca juste après), un postback est donc déclenché pour revenir sur le serveur faire cette validation. Le code de la fonction Page_Load sera donc toujours exécuté, qu'il y ait eu une erreur ou non. Ce qui est logique, Page Load est un événement déclenché toujours lors de la génération de la page quoi qu'il se soit passe sur la page.
On teste donc la propriété **`IsValid`** de la page avant de faire un traitement particulier.
Si la page ne contient aucun contrôle de validation, la page sera toujours valide (Page.IsValid est toujours a **`true`**), sinon, si une erreur de validation s'est produite, il sera a **`false`**.

5.1 Utilisation de Javascript

Par défaut, les contrôles de validation effectuent leurs validations à la fois sur le client (dans le navigateur) et sur le serveur. Sur le client, la validation se fait avec JavaScript. C'est tout de même plus agréable, du point de vue utilisateur, d'avoir un message d'erreur affiché immédiatement en cas de mauvaise saisie plutôt que d'être obligé d'attendre un aller-retour entre le navigateur et le serveur pour faire cette vérification.
ASP.NET 3.5 supporte la validation côté client sur les navigateurs Internet Explorer, Firefox et Opera.

Cependant, même si la validation se fait côté client, lorsque les données seront envoyées au serveur, une nouvelle validation sera refaite côté serveur. Ceci est fait pour des raisons de sécurité : une personne mal intentionnée pourrait recréer un faux formulaire, supprimer les validations et envoyer les données au serveur non validées. C'est pour ca qu'il faut, si une validation est faite sur le client, toujours vérifier la valeur de la propriété **`IsValid`** de la Page.

5.2 La propriété Display

Tous les contrôles de validations ont une propriété Display qui détermine comment le message d'erreur sera affiche. Cette propriété accepte une de ces trois valeurs :

- **Static**
- **Dynamic**
- **None**

Par défaut, la propriété Display a la valeur "**Static**" : dans ce cas, le message d'erreur sera généré de la façon suivante :

```
<span id="val" style="color:Red;visibility:hidden;">(Obligatoire)</span>
```

Notez que le message d'erreur est généré dans un **** qui a un style css "**visibility**" mis a **hidden**.

Si vous utilisez la valeur "Dynamic", le message est généré de la façon suivante :

```
<span id="val" style="color:Red;display:none;">(Obligatoire)</span>
```

Dans ce cas, un style CSS **display** cache le message d'erreur.

Si vous vous rappelez votre cours sur les CSS, les styles CSS **visibility:hidden** et **display:none** peuvent être utilisés pour cacher un élément de la page. Cependant, un élément dont le style **visibility** est mis a **hidden** occupera toujours sa place dans la page même si il n'est pas affiché, alors que l'élément dont le style **display** est mis à **none** n'occupera aucune place dans le page.

La troisième possibilité est **None**. Si vous la choisissez, aucun message n'est affiché. Pour afficher le message d'erreur, on utilisera le contrôle **ValidationSummary** qu'on va voir plus bas.

5.3 Les autres propriétés intéressantes

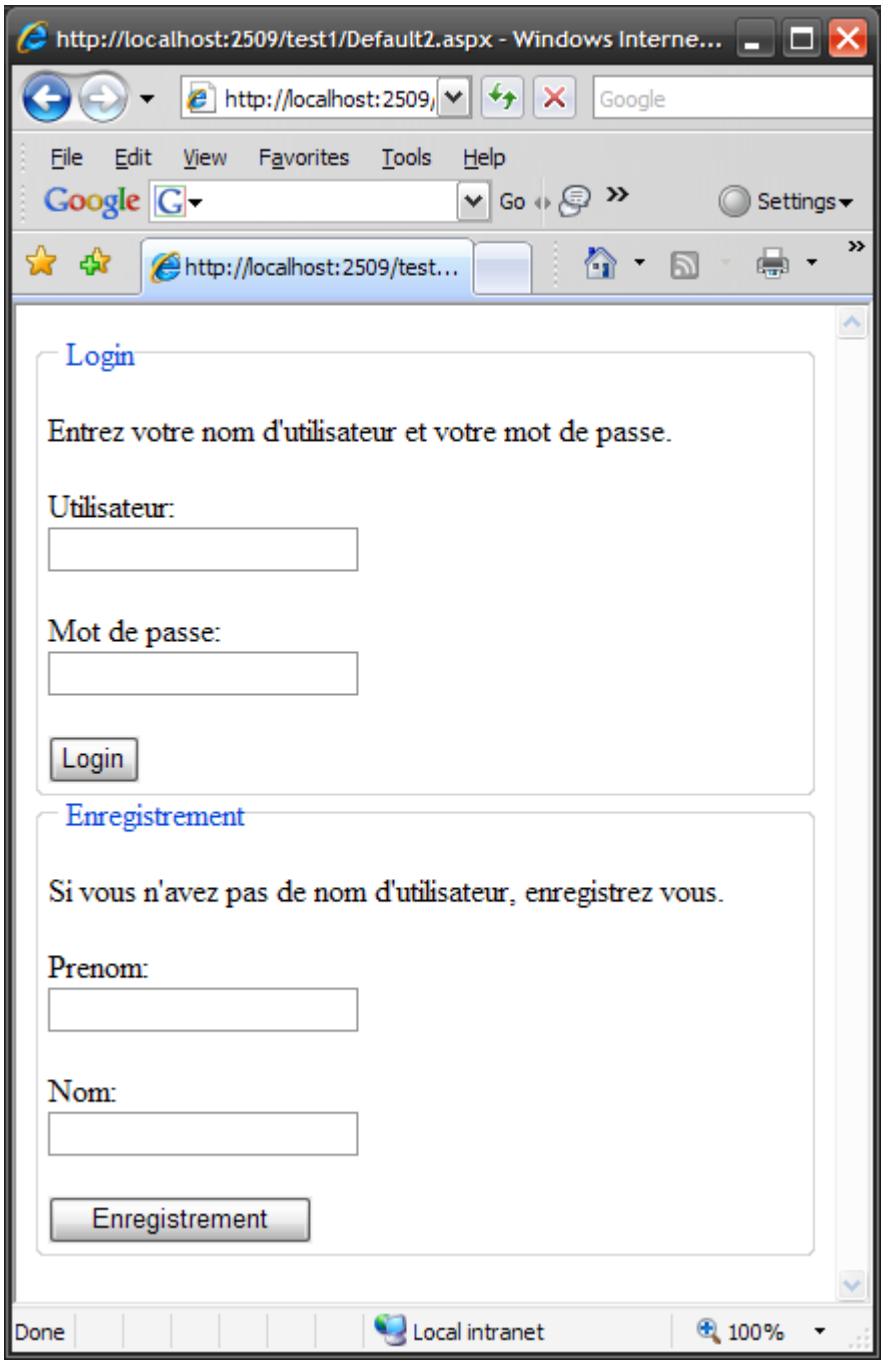
Un truc vite fait, la propriété **Text** contient le message d'erreur à afficher. Par défaut, c'est du texte, mais vous pouvez y insérer des tags HTML si vous voulez afficher des messages d'erreur plus élabores.

Un autre truc pour aider l'utilisateur a corriger son erreur, mettez la propriété **SetFocusOnError** à **true**, lorsqu'une erreur se produira sur le champ à valider, le curseur sera automatiquement positionné sur ce contrôle.

5.4 Les groupes de validation

Un groupe de validation (ValidationGroup) vous permet de grouper des champs d'un formulaire.

Supposons que vous ayez plusieurs petits formulaires dans une page :



Ici nous avons deux petits formulaires : un formulaire avec deux champs pour saisir son nom d'utilisateur et son mot de passe et un formulaire pour saisir son nom et son prénom.

On veut obliger l'utilisateur à saisir soit les champs "Utilisateur" et "Mot de passe" soit les champs "Prenom" et "Nom". Tels que nous les avons vus maintenant, si l'utilisateur renseigne prénom et nom et clique sur "Enregistrement", un message d'erreur va s'afficher en demandant de renseigner les champs "Utilisateur" et "Mot de passe".
Pourquoi ? Parce que lorsqu'on fait un postback (avec le bouton "Enregistrement" par exemple), les validations vont être faites sur tous les champs, peu importe qu'ils soient dans le même formulaire ou non.

Bien embêtant, et pour la petite histoire, c'était ce qui se produisait dans le Framework 1.x, ce qui, en pratique, interdisait l'utilisateur des validateurs.

Heureusement, il y a une solution. Chaque contrôle de validation possède une propriété nommée "**ValidationGroup**". Dans notre exemple, les contrôles de validation du formulaire "Login" auront le même groupe de validation :

```
<fieldset>
<legend>Login</legend>
   <p>Entrez votre nom d'utilisateur et votre mot de passe.</p>
   <asp:Label id="lblUserName" Text="Utilisateur:"
    AssociatedControlID="txtUserName" Runat="server" />
   <br />
   <asp:TextBox id="txtUserName" Runat="server" />
   <asp:RequiredFieldValidator id="reqUserName" ControlToValidate="txtUserName"
    Text="(Required)" ValidationGroup="LoginGroup" Runat="server" />
   <br /><br />
   <asp:Label id="lblPassword" Text="Mot de passe:"
    AssociatedControlID="txtPassword" Runat="server" />
   <br />
   <asp:TextBox id="txtPassword" TextMode="Password" Runat="server" />
   <asp:RequiredFieldValidator id="reqPassword" ControlToValidate="txtPassword"
    Text="(Required)" ValidationGroup="LoginGroup" Runat="server" />
   <br /><br />
   <asp:Button id="btnLogin" Text="Login" ValidationGroup="LoginGroup"
    Runat="server" OnClick="btnLogin_Click" />
 </fieldset>
```

Les contrôles de validation reqUserName (utilisé pour valider le champ de saisie "Nom d'utilisateur") et le contrôle de validation reqPassword (utilisé pour valider le champ de saisie "Mot de Passe") ont tous les deux leur propriété "ValidationGroup" qui vaut "LoginGroup". Ils appartiennent donc au même groupe de validation.

Que voit-on plus bas avec le contrôle Button, une nouvelle propriété "**ValidationGroup**" également ! A quoi sert-elle ? Quand le bouton sera pressé, il ne lancera la validation que sur les contrôles de validation de son propre groupe et oubliera les autres contrôles de la page.

5.5 Désactiver la validation

Nous avons vu que des qu'on clique sur un contrôle Bouton (Button, LinkButton ou ImageButton), la page fait un postback sur le serveur, et juste avant, si des validateurs sont présents dans la page, va les exécuter. Imaginons qu'on ait un bouton "Abandonner" sur la page. L'utilisateur cliquera dessus si il veut quitter cette page et son formulaire. Si il clique dessus, comme le bouton "Abandonner" est un contrôle de type Button, les validations vont être faites. Un comble ! Pour pouvoir quitter la page du formulaire, il va falloir qu'il le remplisse.

Pour éviter ce comportement, tous les contrôles Button (Button, LinkButton, ImageButton) ont une propriété "CauseValidation", si vous la mettez a false, le postback est fait sans que la validation soit faite préalablement.

5.6 Utiliser le contrôle RequiredFieldValidator pour vérifier autre chose qu'une chaine de caractère vide

Par défaut, le contrôle **`RequiredFieldValidator`** vérifie qu'une chaine de caractère non vide a été saisie (les espaces ne comptent pas). Si vous entrez n'importe quoi dans le champ (autre que des espaces), le message d'erreur associé à ce contrôle n'est pas affiché.

Mais vous pouvez utiliser la propriété "InitialValue" pour spécifier une valeur par défaut autre qu'une chaine de caractère vide. Par exemple :

```
<asp:TextBox id="txt" runat="server" Text="Entrez du texte" />
<asp:RequiredFieldValidator id="val" runat="server" ControlToValidate="txt"
InitialValue="Entrez du texte" />
```

Le champ de saisie affichera par défaut "Entrez du texte", si vous ne tapez pas autre chose que ce texte, une erreur de validation se produira. Mais on peut utiliser cette propriété pour vérifier qu'une option a bien été sélectionnée dans une liste déroulante.

Petit exemple :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
void okClick(Object sender, EventArgs e)
{
    if (Page.IsValid)
    lbl.Text = listeCouleurs.SelectedValue;
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<body>

<form id="form1" runat="server">
    <asp:Label id="couleurs" Text="Couleur preferee :"
    AssociatedControlID="listeCouleurs" Runat="server" />
    <asp:DropDownList id="listeCouleurs" Runat="server">
        <asp:ListItem Text="Choisissez une couleur" Value="aucune" />
        <asp:ListItem Text="Rouge" Value="Rouge" />
        <asp:ListItem Text="Vert" Value="Vert" />
        <asp:ListItem Text="Bleu" Value="Bleu" />
    </asp:DropDownList>
    <asp:RequiredFieldValidator id="req" Text="(Obligatoire)"
     InitialValue="aucune"
     ControlToValidate="listeCouleurs" Runat="server" />
    <br />
    <asp:Button id="btn" Text="OK" Runat="server" OnClick="okClick" />
    <hr />
    <asp:Label id="lbl" Runat="server" />
</form>
</body>
</html>
```

Dans ce cas, le validateur vérifiera que le contrôle valide (ici une liste déroulante) a une valeur différente de la valeur proposée par défaut. Si c'est le cas, la valeur choisie est affichée, sinon, c'est un message d'erreur qui est affiche.

5.7 Le contrôle RangeValidator

Voila un autre contrôle de validation bien utile. Ce contrôle vous permet de vérifier si la valeur d'un champ d'un formulaire est comprise entre une valeur minimale et une valeur maximale. Pour cela, vous devez initialiser 5 propriétés de ce contrôle :

ControlToValidate	Identifiant du champ à valider
Text	Texte a afficher si la validation échoue
MinimumValue	Valeur minimale
MaximumValue	Valeur maximale
Type	Type de comparaison : **String** (chaine de caractère), **Integer** (nombre entier), **Double** (nombre décimal), **Date** et **Currency** (Monétaire)

Exemple :

Un champ texte ou l'utilisateur doit entrer un âge compris entre 19 et 60. Si la valeur entrée est hors de cette fourchette, un message d'erreur est affiché.

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
<form id="form1" runat="server">
    <asp:Label id="lblAge" Text="Age:" AssociatedControlID="txtAge"
    Runat="server" />
    <asp:TextBox id="txtAge" Runat="server" />
    <asp:RangeValidator id="reqAge" ControlToValidate="txtAge" Text="(Age incorrect)"
MinimumValue="19" MaximumValue="60" Type="Integer" Runat="server" />
    <br />
    <asp:Button id="btnSubmit" Text="OK" Runat="server" />
</form>

</body>
</html>
```

5.8 Le contrôle CompareValidator

Le contrôle CompareValidator vous permet de faire deux types de validation.
Vous pouvez utiliser ce validateur pour vérifier que les données entrées correspondent a un type de données particulier, comme par exemple une date dans un champ "date de naissance".

Vous pouvez utiliser aussi ce validateur pour comparer la valeur saisie dans un champ du formulaire avec la valeur d'un autre contrôle. Par exemple, dans un site de réservation de billet d'avion, on utilisera ce validateur pour vérifier que la date de retour sera après la date de départ.

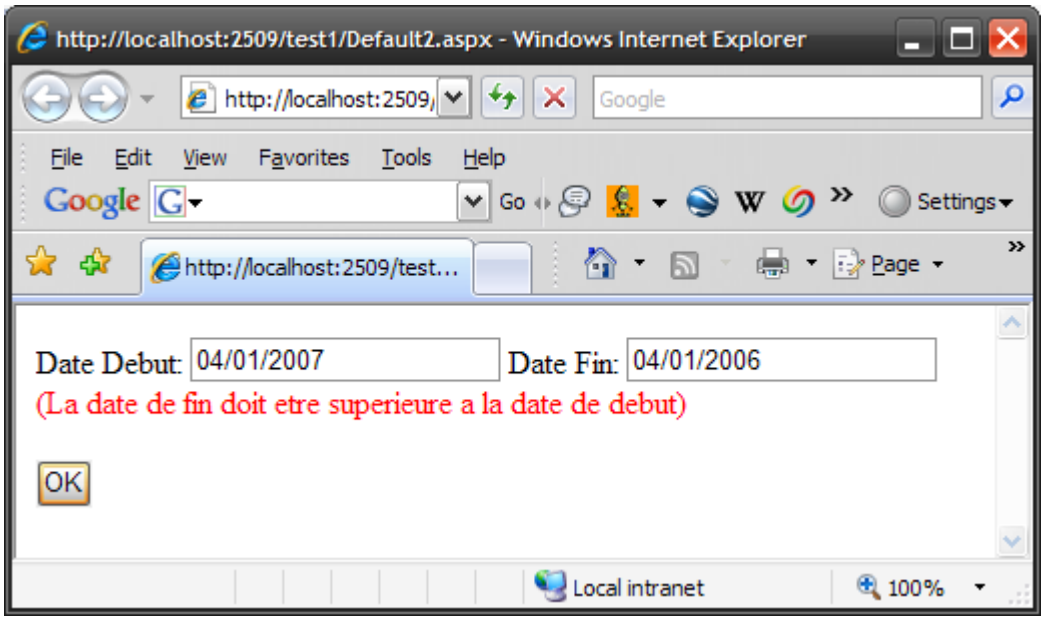
Ce comparateur a six propriétés importantes :

ControlToValidate	Le contrôle a valider
Text	Le message d'erreur à afficher si la validation échoue
Type	Le type de donnée à comparer (comme précédemment sont String, Integer, Double, Date, Currency)
Operator	Le type de comparaison a faire : • DataTypeCheck : la donnée saisie doit être du bon type • Equal : égal • GreaterThan : supérieur a • GreaterThanEqual : supérieur ou égal a • LessThan : inferieur a • LessThan Equal : inferieur ou égal a • NotEqual : différent de
ValueToCompare	Valeur fixe avec laquelle comparer
ControlToCompare	Identifiant du contrôle avec lequel on veut faire la comparaison

Dans l'exemple suivant, nous allons vérifier que la date saisie dans un champ "date de fin" est supérieur a la date saisie dans un champ "date de début"

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
<form id="form1" runat="server">
 <asp:Label id="lbldébut" Text="Date Début:" Runat="server" />
 <asp:TextBox id="début" Runat="server" />
 <asp:Label id="lblfin" Text="Date Fin:" Runat="server" />
 <asp:TextBox id="fin" Runat="server" />
 <asp:CompareValidator id="cmpDate" Text="(La date de fin doit être
  superieure a la date de début)" ControlToValidate="fin"
  ControlToCompare="début" Type="Date" Operator="GreaterThan" Runat="server" />
 <br /><br />
 <asp:Button id="btnSubmit" Text="OK" Runat="server" />
</form>
</body>
</html>
```

Et ca donne ca :



(C'est une date au format US)

5.9 Le contrôle RegularExpressionValidator

Le contrôle **RegularExpressionValidator** vous permet de valider la valeur saisie dans un champ de formulaire avec une expression régulière. Je subodore que vous savez ce qu'est une expression régulière ? C'est une chaine de caractères contenant une suite de caractères ésotériques décrivant un format particulier. On valide la saisie d'un utilisateur avec une expression régulière, si l'évaluation échoue, ca signifie que les données saisies par l'utilisateur n'ont pas le format attendu.

On utilisera typiquement une expression régulière pour valider qu'un utilisateur a saisi correctement une adresse email, un numéro de sécurité sociale, un numéro de téléphone, une date, une somme, un code produit, etc…

Ecrire une expression régulière qui va servir à valider un champ de saisi n'est pas a priori un exercice simple. Si vous utilisez Visual Studio Express 2008, lorsque vous créerez un tel contrôle, il vous proposera une liste d'expressions régulières déjà toutes faites.
Si vous voulez savoir tout sur les expressions régulières, je vous conseille le site : **http://www.regular-expressions.info/** (en anglais, mais vous maitrisez bien sur cette langue).

Pour tester vos expressions régulières, j'utilise Expresso, un petit outil bien sympathique (et gratuit en plus) :
http://www.ultrapico.com/Expresso.htm

Petit exemple : vérifions que l'utilisateur a bien entre une adresse email :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
<form id="form1" runat="server">
 <asp:Label id="lblEmail" Text="Email :" AssociatedControlID="txtEmail" Runat="server"
/>
 <asp:TextBox id="txtEmail" Runat="server" />
 <asp:RegularExpressionValidator id="regEmail"
  ControlToValidate="txtEmail"
  Text="(Adresse Email Incorrecte)"
  ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"
  Runat="server" />
 <br /><br />
 <asp:Button id="btnSubmit" Text="OK" Runat="server" />
</form>
</body>
</html>
```

Notez bien ce machin : "`\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*`"

C'est l'expression régulière qui vérifie que la chaine de caractères est un email. Ca fait peur (il y a de quoi), renseignez-vous donc un peu sur le sujet avant.

5.10 Le contrôle CustomValidator

Si finalement aucun contrôle de validation ne fait ce que vous voulez (vous êtes bien difficile), tout espoir n'est pas perdu. Le CustomValidator vous permet de créer votre propre fonction de validation.

Les trois propriétés importantes du contrôle sont :

ControlToValidate	L'identifiant du contrôle a valider
Text	Le message d'erreur à afficher si la validation échoue
ClientValidationFunction	Le nom d'une fonction (cote client, donc JavaScript) a appeler pour valider la saisie

La dernière propriété vous intrigue ? En fait, avec CustomValidator, vous pouvez choisir de faire la validation côté client ou côté serveur, si vous la faites côté client, indiquez donc dans cette propriété le nom de la fonction JavaScript à appeler pour valider, si vous voulez valider cote serveur, il faut indiquerez nom d'une fonction (cote serveur cette fois-ci) dans la propriété OnServerValidate.

Illustrons ça, nous allons créer un formulaire avec un champ texte multilignes, l'utilisateur saisit du texte, si ce texte fait plus de 10 caractères, un message d'erreur est affiché.

Commençons la validation côté serveur

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

void validation_serveur(Object source, ServerValidateEventArgs args)
{
    if (args.Value.Length > 10) args.IsValid = false;
    else args.IsValid = true;
}

</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>

<form id="form1" runat="server">
```

```
  <asp:Label id="lbl" Text="Commentaire :" AssociatedControlID="txt" Runat="server" />
  <br />
  <asp:TextBox id="txt" TextMode="MultiLine" Columns="30" Rows="5" Runat="server" />
  <asp:CustomValidator id="val" ControlToValidate="txt" Text="(Le commentaire doit
faire moins de 10 caracteres)" OnServerValidate="validation_serveur"
  Runat="server" />
  <br /><br />
  <asp:Button id="btn" Text="OK" Runat="server" />
</form>
</body>
</html>
```

La partie intéressante est bien entendu la fonction "validation_serveur" qui va vérifier que le texte saisi fait moins de 10 caractères.

Cette fonction récupère 2 paramètres : le 1er paramètre (source) est le paramètre habituel d'un évènement, c'est le contrôle qui a généré l'évènement, en l'occurrence ici le champ texte. Le plus intéressant est le paramètre "ServerValidateEventArgs".

Les propriétés de ce paramètre sont :

Value	Récupérez la dedans la valeur a valider
IsValid	Mettez cette valeur a true ou false selon que la validation a réussi/ échoué

Simple non, et c'est aussi simple côté client :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
   <script type="text/javascript">
   function validation_client(source, args)
   {
      if (args.Value.length > 10) args.IsValid = false;
      else args.IsValid = true;
   }
   </script>
</head>

<body>
<form id="form1" runat="server">

    <asp:Label id="lbl" Text="Commentaire :"
     AssociatedControlID="txtComments"
     Runat="server" />
    <br />
    <asp:TextBox id="txt" TextMode="MultiLine" Columns="30" Rows="5"
     Runat="server" />
    <asp:CustomValidator id="val" ControlToValidate="txt" Text="(Le
     commentaire doit faire moins de 10 caracteres)"
     ClientValidationFunction="validation_client"
     Runat="server" />
    <br /><br />
    <asp:Button id="btn" Text="OK" Runat="server" />

</form>
</body>
</html>
```

C'est la même chose ! La fonction JavaScript accepte les mêmes paramètres et les mêmes propriétés. Faites votre choix !

A savoir : quand il y a plusieurs validators sur une page (un de ceux vu plus haut) et des customValidator, ces derniers seront toujours validés en dernier.

5.11 Le contrôle ValidationSummary

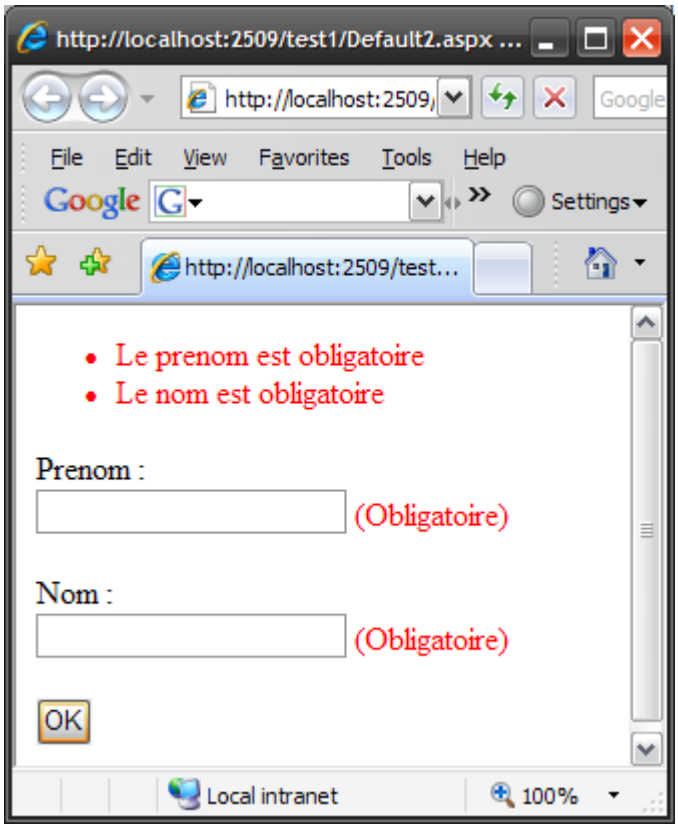
Ce contrôle affiche la liste de toutes les erreurs de validation de la page au même endroit. Il est particulièrement utile lorsque la page contient des grands formulaires. Si l'erreur s'est produit tout en bas de la page, l'utilisateur ne verra pas le message. Si vous utilisez un contrôle **ValidationSummary**, vous pourrez afficher en haut de la page la liste de toutes les erreurs du formulaire.

Vous avez peut être noté que chaque contrôle de validation à une propriété **ErrorMessage**. A quoi sert-il puisqu'on utilise la propriété **Text** pour afficher le message d'erreur ?
Text est utilise pour afficher le message d'erreur à cote du champ en erreur (par exemple "obligatoire") alors que **ErrorMessage** est utilisé pour afficher dans le résumé des erreurs de la page (par exemple "Le champ nom est obligatoire").

Petit Exemple : on doit saisir le nom et le prénom dans ce formulaire. Si un des deux champs n'est pas renseigne, un message ("obligatoire") est affiche a cote du champ en erreur et un résumé de l'erreur est affiche en haut de la page :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
<form id="form1" runat="server">
  <asp:ValidationSummary id="ValidationSummary1" Runat="server" />
  <asp:Label id="lblprenom" Text="Prenom :"
   AssociatedControlID="txtPrenom"
   Runat="server" />
  <br />
  <asp:TextBox id="txtPrenom" Runat="server" />
  <asp:RequiredFieldValidator id="reqPrenom" Text="(Obligatoire)"
   ErrorMessage="Le prenom est obligatoire"
   ControlToValidate="txtPrenom"
   Runat="server" />
  <br /><br />
  <asp:Label id="lblnom" Text="Nom :" AssociatedControlID="txtNom"
   Runat="server" />
  <br />
  <asp:TextBox id="txtNom" Runat="server" />
  <asp:RequiredFieldValidator id="reqNom" Text="(Obligatoire)" ErrorMessage="Le nom est
obligatoire" ControlToValidate="txtNom" Runat="server" />
  <br /><br />
  <asp:Button id="btn" Text="OK" Runat="server" />
```

```
</form>


</body>
</html>
```



Ce contrôle a quelques propriétés intéressantes :

DisplayMode	Indique la façon dont le résumé va être affiché : **BulletList** (liste avec des points, comme sur l'exemple), **List** (une liste simple), **SingleParagraph** (messages affiches les un a la suite des autres)
HeaderText	Vous pouvez ajouter un petit texte en haut du résumé
ShowMessageBox	Si = true, affichera une boite d'alerte au lieu de l'afficher dans la page web
ShowSummary	Si = false, cache le résumé

6 Les contrôles "riches"

ASP.NET met a notre disposition des contrôles très évolués, parmi ceux-ci :

6.1 Le contrôle FileUpload

Le contrôle FileUpload vous permet d'envoyer des fichiers sur le serveur web via une page web. Après que le fichier ait été envoyé sur le serveur, vous pouvez le sauver ou vous voulez, le plus généralement sur le disque ou dans une base de données.

Le contrôle FileUpload dispose de ces propriétés (il y en a d'autres, découvrez les vous-mêmes !)

Enabled	Active / Désactive le contrôle
FileBytes	Permet de récupérer le fichier transféré sous forme d'un tableau d'octets
FileContent	Permet de récupérer le fichier transfere sous forme d'un flux
FileName	Nom du fichier transféré
HasFile	Renvoie true si un fichier a bien été transfere
PostedFile	Récupère le fichier dans une classe HttpPostedFile (bien pratique)

Ce contrôle a également deux méthodes très intéressantes :

Focus	Place le focus par défaut sur ce contrôle dans la page
SaveAs	Sauve le fichier récupère sur un disque sur le serveur

Je disais que récupérer le fichier dans une classe HttpPostedFile est bien pratique car cette classe possède quelques propriétés et méthodes qui facilitent bien la vie, parmi celles-ci :

ContentLength	Indique la taille en octets du fichier téléchargé
ContentType	Indique le type MIME du fichier
FileName	Nom du fichier
InputStream	Récupère le fichier sous forme d'un flux
SaveAs	Sauve le fichier sur un disque du serveur

Vous noterez qu'il y a un peu redondance dans tout ca, vous pouvez récupérer le nom du fichier soit par la propriété FileName du contrôle, soit via la propriété FileName de la propriété PostedFile, idem pour la sauvegarde, soit directement, soit via PostedFile. (Il vaut mieux avoir trop de choix que pas assez)

Suivez bien l'exemple suivant, peu de lignes mais il montre toute la puissance d'ASP.NET. Ca mord un peu sur la suite des cours, ne vous inquiétez pas, on détaillera ca dans le chapitre suivant :

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.IO" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

protected void btn_Click(object sender, EventArgs e)
{
    if  (upImage.HasFile)
        {
        if  (verifieType(upImage.FileName))
            {
            String filePath = "~/dossierUpload/" + upImage.FileName;
            upImage.SaveAs(MapPath(filePath));
            }
        }
}

bool verifieType(string fichier)
{
    string ext = Path.GetExtension(fichier);
    switch (ext.ToLower())
        {
        case ".gif": return true;
        case ".png": return true;
        case ".jpg": return true;
        case ".jpeg": return true;
        default: return false;
        }
}

void Page_PreRender()
{
    string upFolder = MapPath("~/dossierUpload /");
    DirectoryInfo dir = new DirectoryInfo(upFolder);
    listImages.DataSource = dir.GetFiles();
    listImages.DataBind();
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
```

```
<body>
<form id="form1" runat="server">
    <asp:Label id="lbl" Text="Image :" AssociatedControlID="upImage" Runat="server" />
    <asp:FileUpload id="upImage" Runat="server" />
    <br /><br />
    <asp:Button id="btn" Text=" Ajoute Image " OnClick="btn_Click" Runat="server" />
    <hr />
    <asp:DataList id="listImages" RepeatColumns="3" runat="server">
        <ItemTemplate>
        <asp:Image ID="Image1"
         ImageUrl='<%# Eval("Name", "~/ dossierUpload /{0}") %>'
         style="width:200px" Runat="server" />
        <br />
        <%# Eval("Name") %>
        </ItemTemplate>
    </asp:DataList>
</form>
</body>
</html>
```

Que fait ce programme ? Il vous permet de choisir une image, de l'envoyer sur le serveur, puis d'afficher sous forme d'un tableau de vignette la liste des images présentes sur le serveur, tout ca en moins de 50 lignes.

Détaillons d'abord la partie contrôles :

```
<asp:Label id="lbl" Text="Image :" AssociatedControlID="upImage" Runat="server" />
<asp:FileUpload id="upImage" Runat="server" />
<asp:Button id="btn" Text=" Ajoute Image " OnClick="btn_Click" Runat="server" />
```

On a ici un **label** qui affiche le texte, un contrôle **FileUpload** et un bouton pour valider l'envoi. Quand l'utilisateur clique sur le bouton, un postback est fait et la fonction btn_Click est appelée sur le serveur

```
<asp:DataList id="listImages" RepeatColumns="3" runat="server">
<ItemTemplate>
 <asp:Image ID="Image1"
  ImageUrl='<%# Eval("Name", "~/ dossierUpload /{0}") %>'
  style="width:200px" Runat="server" />
 <br />
 <%# Eval("Name") %>
 </ItemTemplate>
</asp:DataList>
```

La, c'est un peu plus complique, on n'a pas encore vu ce contrôle, il s'agit d'un DataList, il permet d'afficher dans le format qu'on veut (défini par le tag ItemTemplate) des données qu'il ira chercher dans une source de données qu'on lui assignera. Dans notre cas, on lui demande d'afficher sur 3 colonnes des données sous la forme d'une image et d'un texte.

Jetons un œil au code behind :

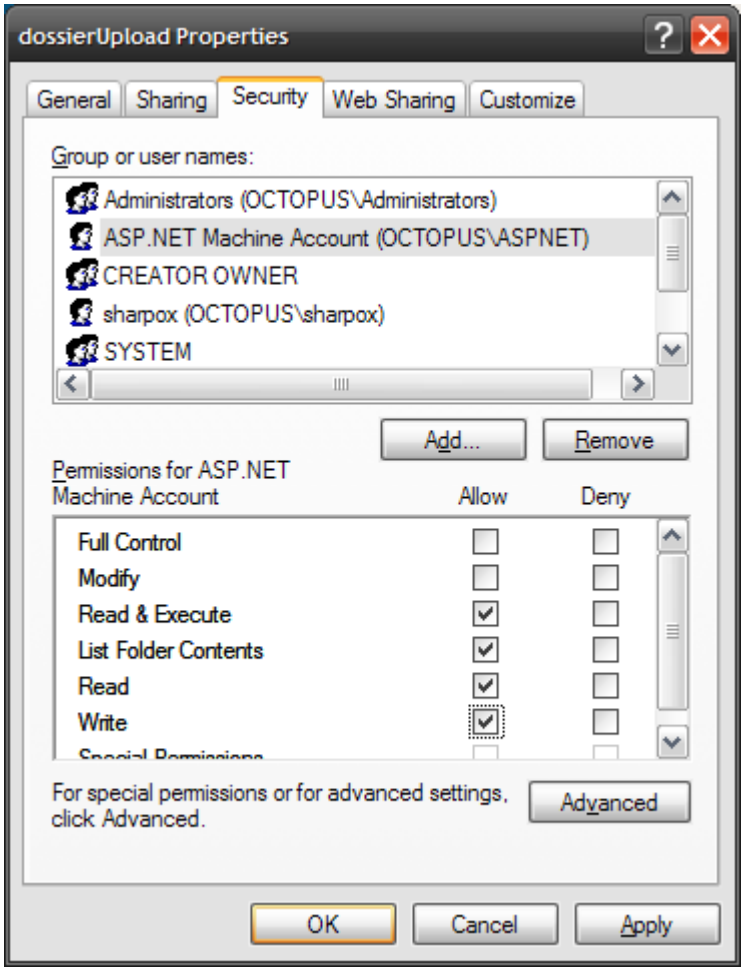
```
protected void btn_Click(object sender, EventArgs e)
{
    if  (upImage.HasFile)
        {
        if  (verifieType(upImage.FileName))
            {
            String filePath = "~/dossierUpload/" + upImage.FileName;
            upImage.SaveAs(MapPath(filePath));
            }
        }
}
```

Quand l'utilisateur clique sur le bouton ok donc, on arrive ici. Le premier test sert a vérifier que le contrôle **FileUpload** a bien récupéré un fichier, on vérifie ensuite que le format du fichier est un format accepté (la vérification est sommaire, a partir du nom du fichier, rien ne vous empêche de faire quelque chose de plus élaboré), puis on sauve le nom du fichier sur le disque du serveur.

Notez ici l'utilisation de la fonction **MapPath**. Cette fonction convertit un chemin relatif à la racine du site en un chemin complet sur le disque dur du serveur. Si votre site est situe `dans C:\inetpub\site` et que vous faites `MapPath("~/dossier/fichier")`, vous `obtiendrez C:\inetpub\site\dossier\fichier`.
N'oubliez pas de faire cette manipulation car la méthode SaveAs ne connait pas le chemin relatif du site et échouera lors de la sauvegarde.

Notez également, très important, que vous devez donner des droits en écritures sur le dossier de sauvegarde, ici "dossierUpload" à l'utilisateur ASP.NET.

L'utilisateur ASP.NET diffère selon le système. Si votre site tourne sous Windows 2003 server (toutes versions), l'utilisateur est le compte "NETWORK SERVICE" (en anglais) ou "SERVICE RESEAU" sur un système en français. Pour les autres systèmes (XP ou Vista), l'utilisateur est le compte "ASPNET".



(Exemple pour Windows XP Pro : OCTOPUS est le nom de la machine)

```
void Page_PreRender()
{
    string upFolder = MapPath("~/dossierUpload /");

    DirectoryInfo dir = new DirectoryInfo(upFolder);
    listImages.DataSource = dir.GetFiles();
    listImages.DataBind();
}
```

On retrouve ici un des évènements produits lors de la génération de la page. Cette fonction récupère un tableau contenant tous les fichiers d'un dossier et se sert de ce tableau comme source de données pour la DataList que nous avons vu plus haut.

Pourquoi ne pas avoir utilise un évènement Page_Load comme d'habitude ? Souvenez-vous de l'ordre d'exécution des évènements, le Page_Load se produit en premier, puis les évènements liés aux contrôles sur la page.
Lors du Page_Load, la fonction associée a l'évènement du bouton "Ajoute Image" n'a pas encore été exécutée (l'image uploadée n'a pas encore été sauvée dans le bon répertoire), donc générer la liste à ce moment n'aurait pas de sens, elle ne contiendrait pas l'image que l'utilisateur vient d'envoyer. On utilisera donc l'évènement PreRender qui se déclenche après les évènements des contrôles, là, on est sur que l'image aura été sauvée.

6.1.1 Uploader des gros fichiers

Par défaut, vous ne pouvez pas uploader des fichiers de plus de 4 Mo. Si vous voulez dépasser cette limite, il faut modifier un paramètre dans le fichier web.xml :

```
<configuration>
   <system.web>
   <httpRuntime  maxRequestLength="10240" requestLengthDiskThreshold="100" />
   </system.web>
</configuration>
```

Le paramètre "**maxRequestLength**" indique la taille maximale autorisée. Si vous dépassez cette taille, une exception est générée.

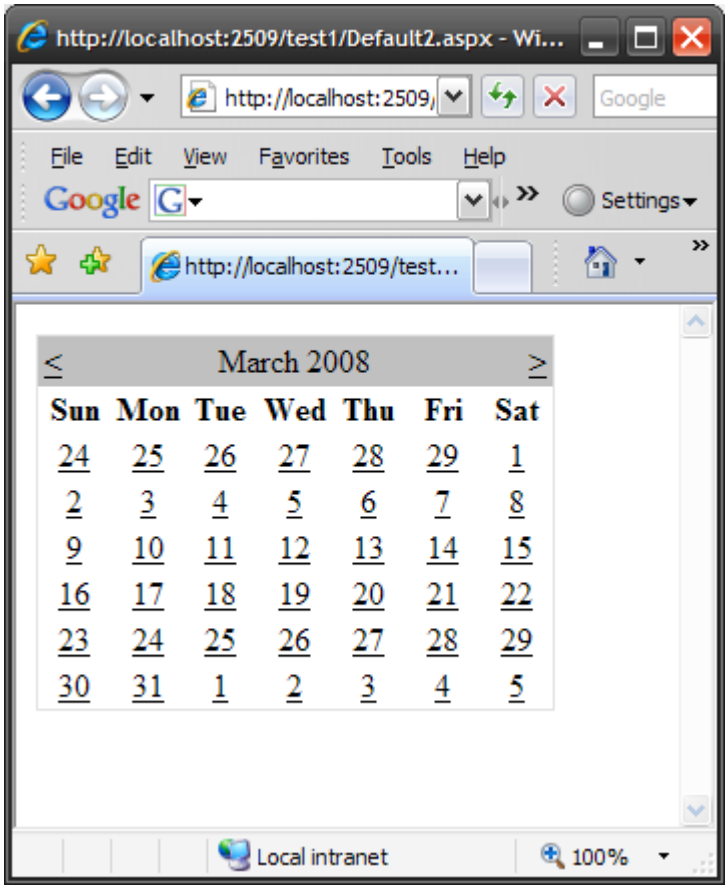
Au passage, vous pouvez également modifier le paramètre " **requestLengthDiskThreshold**" qui indique, en octets, la taille du buffer en mémoire où est stocké le fichier avant de le transférer dans un fichier temporaire sur le disque dur du serveur, vous pouvez augmenter cette taille pour améliorer les performances du transfert au détriment de la taille occupée en mémoire.

6.2 Afficher un calendrier

Le contrôle Calendar vous permet d'afficher un calendrier. Vous pouvez utiliser ce calendrier pour saisir une date ou pour afficher par exemple un calendrier d'évènements a venir.

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
<form id="form1" runat="server">
    <asp:Calendar id="Calendar1" Runat="server" />
</form>
</body>
</html>
```

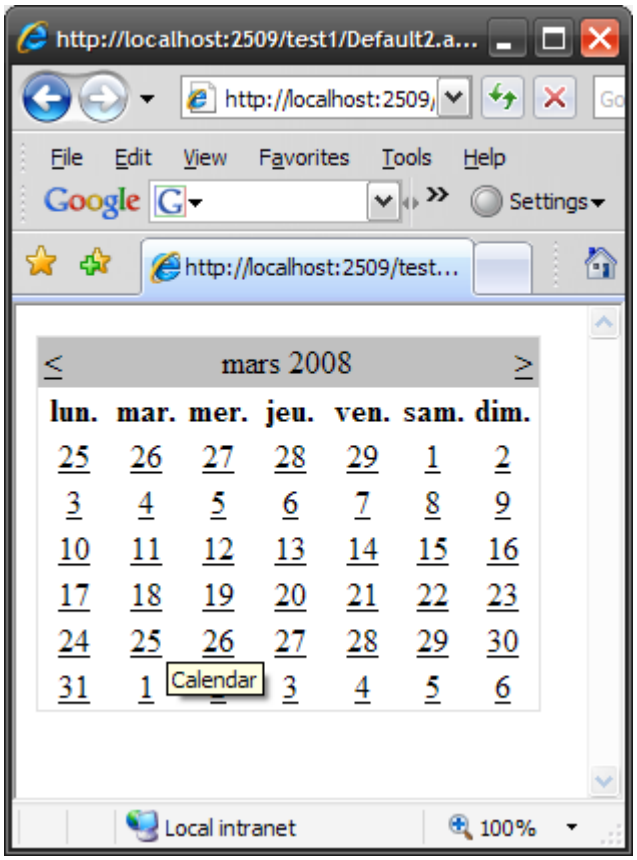
Difficile de faire plus simple ?



Il affiche le calendrier dans le format et la langue du serveur

Si vous modifiez une des propriétés de la page (la propriété Culture) :

```
<%@ Page Language="C#" Culture="fr-FR" %>
```



Magie, le calendrier s'affiche en français !

Ce contrôle propose les propriétés suivantes (entre autres) :

DayNameFormat	Format des jours de la semaine : • FirstLetter : 1ere lettre • FirstTwoLetters : deux premières lettres • Full : nom complet • Short : nom abrégé (lun. Mar.) • Shortest : nom très abrégé (lu ma me)
NextMonthText	Spécifie le texte qui apparait pour le mois suivant
NextPrevFormat	Spécifie le format du lien pour le mois suivant : • CustomText : texte libre • FullMonth : nom complet du mois • ShortMonth : nom abrégé du mois

PrevMonthText	Spécifie le texte qui apparait pour le mois précédent
PrevMonthText	Idem a NextPrevFormat (pour le mois précédent)
SelectedDate	Renvoie ou initialise la date actuellement sélectionnée
SelectedDates	Si plusieurs dates sont sélectionnables simultanément, renvoie ou initialise la liste de ces dates
SelectionMode	Indiquez ici comment va se faire la sélection : • Day : un seul jour sélectionnable • DayWeek : ajoute un bouton en début de chaque semaine pour pouvoir sélectionner une semaine complète • DayWeekMonth : ajoute un bouton en début de mois pour pouvoir sélectionner le mois complet • None : aucune sélection possible : le calendrier sert juste a afficher
SelectMonthText	Le texte (ou code html) du lien pour sélectionner le mois (par défaut, c'est >)
SelectWeekText	Le texte (ou code html) du lien pour sélectionner la semaine
ShowDayHeader	Affiche / cache le nom des jours en haut du calendrier
ShowNextPrevMonth	Affiche / cache les liens pour passer aux mois suivants / précédents
ShowTitle	Affiche / cache le titre du calendrier
TitleFormat	Format du titre : • Month : affiche le mois • MonthYear : affiche le mois et l'année
TodaysDate	Vous permet de modifier la date du jour (uniquement pour ce contrôle), par défaut, c'est la date du serveur
VisibleDate	Le calendrier sera positionné par défaut sur le mois de cette date (sinon, il est positionné par défaut a la date du jour)
DayRender	Evénement déclenché à chaque fois qu'un jour va être affiché dans le calendrier, spécifiez ici une fonction à appeler pour afficher un contenu spécifique dans chaque journée)
SelectionChanged	Evénement déclenché quand un jour, une semaine ou un mois a été sélectionné
VisibleMonthChanged	Evénement déclenché quand le lien sur le mois suivant ou précédent est cliqué

6.3 Afficher des morceaux de pages (vues) : le contrôle MultiView

Le contrôle MultiView vous permet d'afficher et de cacher différentes parties de la page. Il est particulièrement utile si vous voulez créer une page avec des onglets.

Le contrôle MultiView contient un ou plusieurs contrôles View. Vous choisissez, via le contrôle MultiView quel contrôle View va être affiche (les autres étant donc caches, car un seul contrôle View peut être affiche a la fois).

Qu'est-ce qu'un contrôle View ? c'est un container simple qui peut contenir n'importe quoi : du HTML standard, des contrôles ASP.NET, du texte, etc…

Le contrôle MultiView dispose des propriétés suivantes :

ActiveViewIndex	Indique ou sélectionne le numéro du contrôle View visible
Views	Renvoie une collection des contrôles View
GetActiveView	Renvoie le contrôle View actuellement affiche
SetActiveView	Sélectionne le contrôle View qui va être affiche
ActiveViewChanged	Evénement déclenché lorsqu'un contrôle View est sélectionné

Le contrôle View dispose de deux évènements

Activate	Déclenché lorsqu'il est sélectionné
Deactivate	Déclenché lorsqu'il est désactivé

Petit exemple :

Une liste déroulante dans laquelle on choisit une vue, la vue correspondante est affichée lors de la sélection.

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
    MultiView1.ActiveViewIndex = int.Parse(DropDownList1.SelectedValue);
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
```

```
<body>
<form id="form1" runat="server">

  <asp:DropDownList ID="DropDownList1" runat="server"
   onselectedindexchanged="DropDownList1_SelectedIndexChanged" AutoPostBack="True">
      <asp:ListItem Text="View 1" Value="0"></asp:ListItem>
      <asp:ListItem Text="View 2" Value="1"></asp:ListItem>
      <asp:ListItem Text="View 3" Value="2"></asp:ListItem>
  </asp:DropDownList>

  <br /><br />

  <asp:MultiView id="MultiView1" ActiveViewIndex="0" Runat="server">
      <asp:View ID="View1" runat="server">
          <br />Vue 1
            <br />Vue 1
      </asp:View>

      <asp:View ID="View2" runat="server">
            <br />Vue 2
            <br />Vue 2
      </asp:View>

      <asp:View ID="View3" runat="server">
            <br />Vue 3
            <br />Vue 3
      </asp:View>
   </asp:MultiView>
</form>
</body>
</html>
```

Notez juste la propriété "AutoPostBack=true" de la liste déroulante, petit rappel, ca signifie qu'un postback aura lieu lorsqu'on sélectionnera un élément de la liste, sinon, il ne se passera rien, l'évènement cote serveur ne sera pas déclenché.

6.4 Afficher un formulaire en plusieurs parties

Le contrôle **MultiView** peut également être utilise pour diviser un grand formulaire en plusieurs petits sous formulaires. Vous pouvez associer des commandes particulières a des boutons (en initialisant la propriété CommandName du bouton) contenus dans un MultiView, quand les boutons sont cliqués, le **Multiview** change seul la vue active.

Vous pouvez utiliser une des quatre commandes suivantes :

NextView	Le MultiView sélectionnera la vue suivante
PrevView	Le MultiView selectionnera la vue précédente
SwitchViewByID	Le multiview selectionnera la vue dont l'identifiant est spécifié dans la propriété CommandArgument du bouton
SwitchViewByIndex	Le multiview selectionnera la vue dont le numéro est spécifié dans la propriété CommandArgument du bouton

Exemple :

Trois vues, avec en bas de chaque vue, des boutons de navigation vers la vue suivante ou précédente. Sur la dernière vue, un bouton pour revenir au début.
Tout cela sans une seule ligne de code.

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
<body>
<form id="form1" runat="server">

 <asp:MultiView id="MultiView1" ActiveViewIndex="0" Runat="server">
  <asp:View ID="View1" runat="server">
    <br />Vue 1<br /><br />
    <asp:Button ID="Button1" runat="server" Text="Vue suivante"
     CommandName="NextView" />
  </asp:View>
  <asp:View ID="View2" runat="server">
    <br />Vue 2<br /><br />
    <asp:Button ID="Button2" runat="server" Text="Vue precedente"
     CommandName="PrevView" />
    <asp:Button ID="Button3" runat="server" Text="Vue suivante" CommandName="NextView"
/>
  </asp:View>
  <asp:View ID="View3" runat="server">
    <br />Vue 3<br /><br />
    <asp:Button ID="Button5" runat="server" Text="Vue precedente"
     CommandName="PrevView" />
    <asp:Button ID="Button4" runat="server" Text="1ere vue"
CommandName="SwitchViewByID"
     CommandArgument="View1" />
  </asp:View>
</asp:MultiView>
</form>
</body>
</html>
```

7 Les masterpages

Bien remis du chapitre sur les contrôles ? Attaquons maintenant les master pages (pages maitres en français). Une **MasterPage** vous permet de partager le même contenu entre plusieurs pages du site sans avoir à le recopier sur chaque page.

Pour prendre un exemple simple, supposons que toutes les pages de votre site aient une entête et un pied de page identique, ce serait un peu bête de recopier cette entête et ce pied sur chaque page, vous vous rendrez vite compte que c'était bête le jour ou vous devrez modifier une information dans le pied de page et que vous devrez répéter l'opération 300 fois.

D'où l'intérêt de la **MasterPage**. Vous créez une sorte de page ASP.NET qui contient l'entête et le pied et vous indiquez où, dans cette page, le contenu va changer. Puis, lorsque vous créerez une nouvelle page ASP.NET, vous indiquerez juste que vous utiliserez cette MasterPage et vous vous préoccuperez uniquement du contenu de la page. Quand elle sera générée, l'entête et le pied seront automatiquement ajoutés.

Ainsi, le jour où vous voulez modifier l'entête de votre site, vous n'aurez qu'à modifier votre MasterPage, toutes les pages l'utilisant seront automatiquement modifiées.

La MasterPage étant une page ASP.NET normale, vous pouvez y inclure tous les contrôles que vous désirez, et même, si besoin est, communiquer entre la MasterPage et la page incluse.

7.1 Créer une MasterPage

Vous créez une MasterPage en créant un fichier dont l'extension est .**master**. Vous pouvez placer cette MasterPage n'importe où dans le site, vous pouvez utiliser plusieurs MasterPage dans le même site. Une fois encore, vous pouvez placer dans une MasterPage tout ce que vous placez habituellement dans une page ASP.NET (Contrôles, HTML, texte…)

Si vous utilisez VS2008, vous créez encore plus simplement une page en cliquant avec le bouton droit sur l'emplacement ou vous voulez la créer dans le site, puis en cliquant sur Add (Ajouter) et en choisissant "**MasterPage**".

Il y a deux choses a savoir concernant la MasterPage. Notez qu'au lieu de l'habituelle directive `<%@ Page @%>` qu'on trouve en haut d'une page ASP.NET et qui définit les propriétés de la page, on trouve la directive `<%@ Master @%>`, notez ensuite que la MasterPage contient un nouveau contrôle, le contrôle **PlaceHolder**.

Lorsque la MasterPage est fusionnée avec la page contenu, le contenu est inséré a l'emplacement des contrôles PlaceHolder.

Petit Exemple :

```
<%@ Master Language="C#" %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
    <asp:ContentPlaceHolder id="head" runat="server">
    </asp:ContentPlaceHolder>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <div>CECI EST L'ENTETE</div>

        <asp:ContentPlaceHolder id="ContentPlaceHolder1"
         runat="server">
        </asp:ContentPlaceHolder>

        <div>CECI EST LE PIED DE PAGE</div>
    </div>
    </form>
</body>
</html>
```

Ceci est une MasterPage générée par VS2008, j'y ai juste ajoute une entête et un pied de page. Qu'y trouve-t-on d'intéressant ?

```
<%@ Master Language="C#" %>
```

Qui indique que c'est une MasterPage

```
<asp:ContentPlaceHolder id="head" runat="server">
</asp:ContentPlaceHolder>
```

Un contrôle PlaceHolder placé dans le **<HEAD>** de la page

```
<div>CECI EST L'ENTETE</div>
<asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
</asp:ContentPlaceHolder>
```

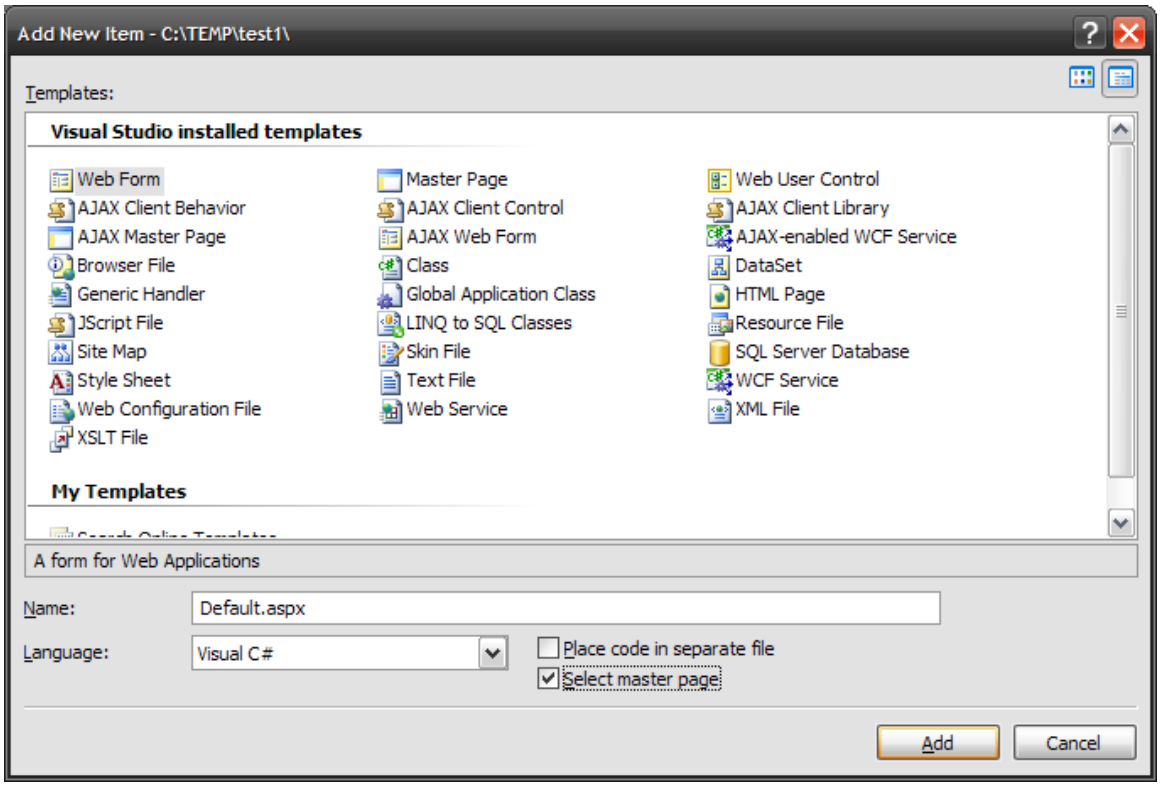
```
<div>CECI EST LE PIED DE PAGE</div>
```

Puis la page elle-même : l'entête, un contrôle PlaceHolder qui contiendra le contenu de la page et le pied de page.

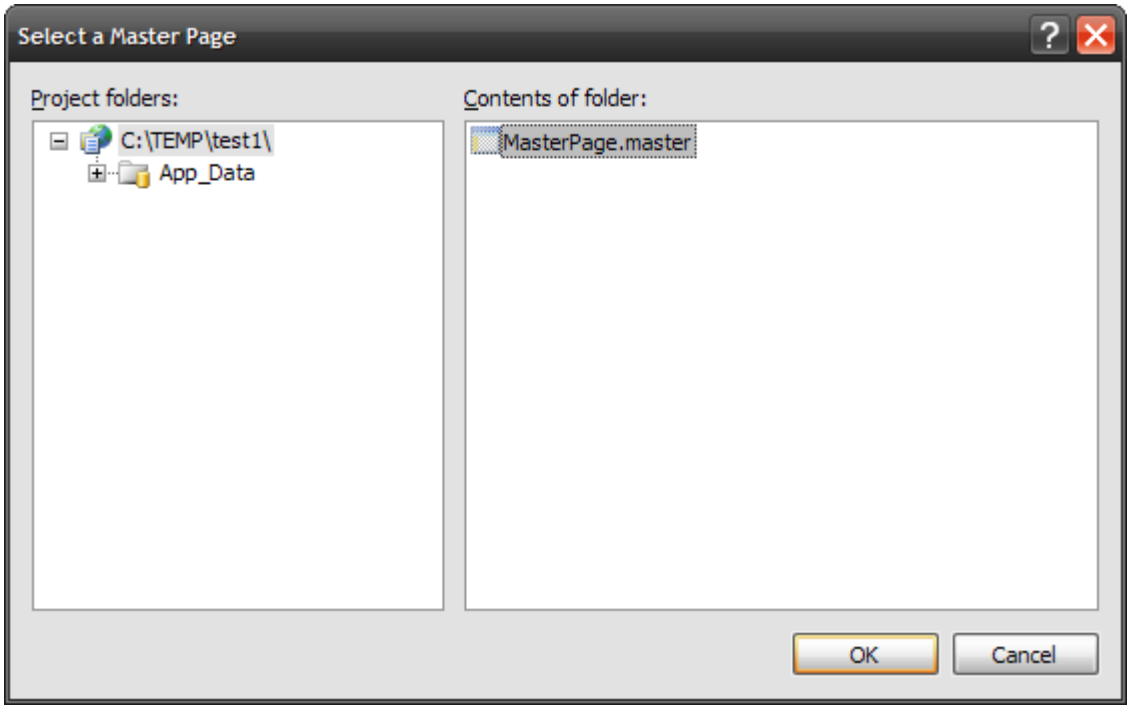
Pourquoi avoir un autre contrôle PlaceHolder dans l'entête ? Au cas ou on voudrait que la page qui va utiliser cette MasterPage veuille insérer des éléments particuliers dans le <HEAD> de la page : script, fichier css, méta tags, etc… Nous allons voir ca.

7.2 Utilisation de la MasterPage

Une fois la MasterPage créée, nous allons l'utiliser avec une page ASP.NET. Si vous utilisez VS2008 (ce que je vous conseille une fois encore), lors de l'ajout d'une nouvelle page ASP.NET (WebForm), cochez la case "Select MasterPage"



L'arborescence du site est affichée et sélectionnez une MasterPage :



Et tout est fait ! La page générée ressemble a ca :

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master" Title="Untitled Page" %>

<script runat="server">

</script>

<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
</asp:Content>
```

Plutôt simple non ? Qu'y trouve-t-on ?

Notre directive Page est un peu différente :

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master" Title="Untitled Page" %>
```

La propriété **MasterPageFile** indique le chemin dans le site de la MasterPage qui va être utilisée

Ensuite, on ne trouve plus de tags **<HTML>**, **<HEAD>**, **<BODY>**, etc… puisque tout est dans la MasterPage, on trouve juste un nouveau contrôle utilisé deux fois : **Content**. Le contrôle Content va contenir le contenu de votre page. La propriété importante de ce contrôle est ContentPlaceHolderID. Vous y indiquez l'identifiant de contrôle **PlaceHolder** correspondant dans la MasterPage.

Ce que vous placerez dans le contrôle Content "Content1" ira se placer dans le contrôle PlaceHolder "ContentPlaceHolder1" de la MasterPage.

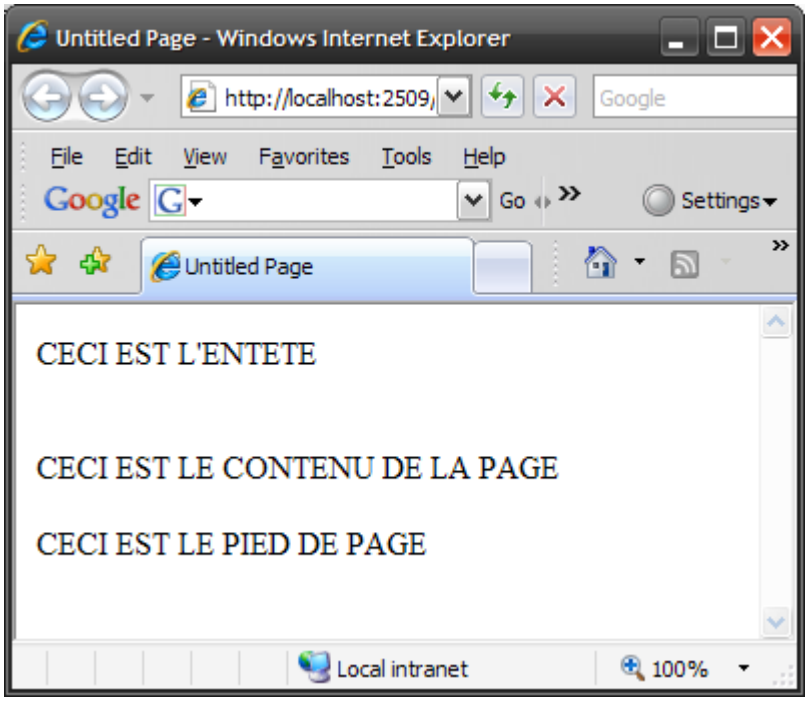
Exemple :

Modifions la page ASP.NET :

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master" Title="Untitled Page" %>

<script runat="server">

</script>

<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
   <script type="text/javascript" src="/script.js"></script>
   </asp:Content>
   <asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
Runat="Server">
   <br /><br />CECI EST LE CONTENU DE LA PAGE<br /><br />
</asp:Content>
```

Dans le PlaceHolder "HEAD" nous plaçons le chargement d'un fichier de script JavaScript, dans le placeholder "ContentPlaceHolder1", nous plaçons le contenu de notre page.

ET voici ce qui est affiché dans le browser : l'entête et le pied viennent de la MasterPage, le contenu de la page elle-même (dont l'URL est utilisée)



Et si on regarde le source de la page généré :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Untitled Page</title>
    <script type="text/javascript" src="/script.js"></script>
</head>

<body>
    <form name="aspnetForm" method="post" action="Default.aspx"
     id="aspnetForm">
    <div>
      <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
       value="/wEPDwUKMTY1NDU2MTA1MmRkEKzBQflqIWrdTcuftE54F4be7vE=" />
    </div>

    <div>
        <div>CECI EST L'ENTETE</div>

        <br /><br />CECI EST LE CONTENU DE LA PAGE<br /><br />

        <div>CECI EST LE PIED DE PAGE</div>
    </div>
    </form>
</body>
</html>
```

On trouve dans le tag <HEAD> de la page notre chargement de script :

```
<head>
    <title>Untitled Page</title>
    <script type="text/javascript" src="/script.js"></script>
</head>
```

7.3 Créer un contenu par défaut

Vous avez surement remarque que vous n'êtes pas obligé de remplir les contrôles Content dans les pages ASP.NET utilisant une MasterPage. Si vous n'y mettez pas de contenu, rien ne sera affiché dans la page générée (ce qui est un peu normal).

Mais vous pouvez créer un contenu par défaut a un PlaceHolder de la MasterPage. Dans le contrôle PlaceHolder de la MasterPage, vous créez un contenu :

```
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
    <br /><br />CECI EST LE CONTENU PAR DEFAUT DE LA PAGE<br /><br />
</asp:Content>
```

Si, dans la page ASP.NET utilisant cette MasterPage, vous ne créez pas de contenu pour le PlaceHolder "ContentPlaceHolder1", c'est celui par défaut qui sera utilise. Si vous créez un contenu pour ce PlaceHolder, le contenu par défaut de la MasterPage sera automatiquement remplace par celui de la page ASP.NET

7.4 Imbriquer des MasterPage

Bien sur, une MasterPage étant une sorte de page ASP.NET, rien n'interdit à une MasterPage d'avoir elle-même une MasterPage. Vous pouvez ainsi profiter d'une des nouvelles fonctionnalités de VS2008,: vous pouvez travailler en mode design avec des MasterPage imbriquées, ce qui, jusqu'alors était impossible.

7.5 Utilisation d'images et de liens dans les MasterPage

Un point important a noter : l'utilisation des images et des liens HyperText dans les MasterPage est un peu tordue lorsque vous utilisez des chemins relatifs pour vos URL. Les chemins relatifs sont interprétés différemment selon que vous utilisez ce chemin dans un tag HTML ou dans un contrôle ASP.NET.

Si vous utilisez un chemin relatif dans un contrôle ASP.NET, l'URL est interprétée relativement a la MasterPage. Par exemple, si la MasterPage est dans un dossier "MesMasterPages", et que vous avez un contrôle Image dans la page :

```
<asp:Image ImageUrl="image.gif" runat="server" />
```

Le serveur ira chercher l'image `"/MesMasterPages/image.gif"`.

La situation est différente si vous utilisez un élément HTML standard. Si, toujours dans cette MasterPage, vous avez un élément <IMG> :

```
<img src="image.gif" />
```

Et que cette MasterPage est utilisée par une page ASP.NET placée dans le dossier "MesPages", le serveur ira chercher l'image "**`/MesPages/image.gif`**".

La règle est : les chemins relatifs dans les URL utilisées par des contrôles ASP.NET sont relatifs à la MasterPage, les chemins relatifs dans les URL utilisées par des éléments HTML sont relatifs à la page utilisant cette MasterPage. Un peu complexe certes, mais une fois qu'on le sait…

7.6 Accéder a des propriétés de la MasterPage a partir des pages de contenu

Pour pouvoir facilement accéder a des propriétés de la MasterPage à partir des pages de contenu, je vous conseille d'exposer ces propriétés via une propriété C# :

Supposons que vous ayez un contrôle Label dans la MasterPage que vous vouliez modifiez à partir de la page contenu :

```
<%@ Master Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<script runat="server">
public string labelModifiable
{
    get { return label.Text; }
    set { label.Text = value; }
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >

<body>
<form id="form1" runat="server">
    <h1><asp:Label ID="label" runat="server" /></h1>
    <asp:contentplaceholder id="ContentPlaceHolder1" runat="server" />
</form>

</body>
</html>
```

La propriété Text du label que l'on veut rendre modifiable est exposée via la propriété "labelModifiable" de la MasterPage.

Dans la page contenu, on trouvera :

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master" Title="Untitled Page" %>
<%@ MasterType VirtualPath="~/MasterPage.master" %>

<script runat="server">

void Page_Load()
{
    Master.labelModifiable = "Label modifie";
}

</script>

<asp:Content ID="Content" ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
<br /><br />CECI EST LE CONTENU DE LA PAGE<br /><br />
</asp:Content>
```

Dans la page contenu, on accède donc simplement a la propriété labelModifiable de la façon suivante :

```
Master.labelModifiable = "Label modifie";
```

Il faut cependant noter quelque chose d'important. Par défaut, la propriété **Master** de la page retourne la MasterPage de la page. La MasterPage en question est de type Page, c'est une page ASP.NET normale. Sauf que dans une page ASP.NET normale, il n'y a pas de propriété "labelModifiable", ce qui va provoquer une erreur de compilation. Pour éviter cela, on a ajoute une ligne au début du fichier :

```
<%@ MasterType VirtualPath="~/MasterPage.master" %>
```

Que signifie-t-elle ? qu'on va typer le type Master retourné par la propriété Master avec la MasterPage indiquée par la propriété VirtualPath. Les propriétés exposées par cette page deviennent donc visibles (et compilable par la même occasion).

Si vous trouvez la gymnastique un peu trop pénible (créer une propriété dans la MasterPage, puis ajouter un typage dans la page contenu), vous pouvez utiliser la fonction FindControl.

7.7 Utilisation de la fonction FindControl

Cette méthode demande moins de travail en amont mais je la considère comme moins propre… A vous de voir. Dans ce cas, plus besoin de créer une propriété dans la MasterPage, plus besoin d'ajouter un typage dans la page contenu. On va accéder directement au contrôle "label" de la MasterPage a partir de la page contenu :

Dans le Page_Load, on fera :

```
Label l = (Label)Master.FindControl("label");
l.Text = "label modifie";
```

FindControl va aller chercher dans la MasterPage un contrôle identifie par "label". La fonction renvoie un Contrôle générique, il faut donc le caster sur un Label, puis une fois récupère, on peut accéder a toutes les propriétés de ce contrôle, dont la propriété Text qui nous intéresse ici.
A vous de voir ce que vous préférez.

7.8 Charger des MasterPages dynamiquement

Avant de vous laisser a vos méditations sur les MasterPage, sachez que vous pouvez charger des MasterPages dynamiquement (et donc modifier a la volée la MasterPage d'une page, ce qui peut vous permettre d'avoir plusieurs habillages pour une même page).

Simplement en utilisant la propriété MasterPageFile de la page :

```
protected void Page_PreInit(object sender, EventArgs e)
{
      MasterPageFile = "maMasterPage.master";
}
```

Vous noterez que la MasterPage doit être initialisée tout au début de la génération de la page. Donc l'utilisation de Page_Load n'est pas possible, il est alors trop tard et la page a déjà commencée a être générée. C'est pour ca que la sélection de la MasterPage se fera lors de l'évènement **PreInit** qui est le tout premier évènement déclenché lors de la génération de la page, avant d'avoir fait quoi que ce soit.

8 Créer des sites web avec des thèmes

Grace aux thèmes d'ASP.NET vous pouvez associer un style spécifique a contrôle, ce style sera utilise de la même façon sur tout le site.

8.1 Créer un thème

Pour créer un thème, il suffit de créer un dossier particulier a la racine de votre site : le dossier "App_Thèmes". Chaque sous-dossier crée dans ce dossier représente un thème.
Le dossier contiendra les fichiers définissant le thème : des fichiers skin et des fichiers CSS.

Un thème peut contenir un ou plusieurs fichiers Skin. Un fichier Skin définit les propriétés d'un contrôle ASP.NET dont vous voulez modifier l'apparence. Par exemple, si vous voulez que tous les contrôles TextBox de votre site aient un fond jaune et une bordure en pointilles, vous aller créer un Thème "MonTheme" (donc créer un dossier "MonTheme" dans le dossier "App_Themes") et dans ce dossier créer un fichier Skin "TextBox.skin". (vous pouvez nommer votre fichier Skin comme il vous plait, mais tant qu'a faire, pour s'y retrouver facilement, autant le nommer avec le nom du contrôle qu'il va modifier).

Dans ce fichier, on trouvera :

```
<asp:TextBox
BackColor="Yellow"
BorderStyle="Dotted"
Runat="Server" />
```

qui a défini les propriétés **BackColor** (couleur de fond) a jaune et **BorderStyle** (style de la bordure) en pointilles pour le contrôle de type **TextBox**. Notez que pour définir le skin de ce contrôle, vous devez toujours mettre l'attribut "**runat=server**" mais vous ne devez pas mettre d'attribut ID. Notez également que seules les propriétés ayant rapport avec l'affichage sont modifiables dans un skin (vous ne pouvez pas initialiser ici une fonction appelée lors d'un évènement ou une propriété du type AutoPostBack qui fera un postback automatique si le contenu du contrôle est modifié).

Vous avez bien compris qu'il ne s'agit pas ici d'une page ASP.NET mais juste de la définition de la façon dont doit être affiche le contrôle TextBox si il se trouve dans une page qui utilise le thème "MonTheme".

Vous pouvez définir dans ce skin d'autres contrôles ASP.NET. En fait, vous pouvez vous organiser comme vous voulez : tous vos contrôles dans le même fichier skin, un fichier skin par contrôle, c'est vous qui voyez…

De toute façon peut importe, lors de la compilation, tous les skin seront compilés et regroupés dans une seule classe.

Comment l'utiliser ?

```
<%@ Page Language="C#" Theme="MonTheme" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
<form id="form1" runat="server">
    <asp:TextBox Runat="server" />
</form>
</body>
</html>
```

Simplement en initialisant la propriété "Thème" de la page : le contrôle TextBox de cette page aura un fond jaune et une bordure pointillée.

Que se passe-t-il si vous voulez, dans votre page, que malgré tout, un contrôle TextBox n'ait pas un fond jaune et une bordure pointillée mais ait une bordure pleine et un fond rouge ?

Pour cela, on va ajouter dans notre fichier TextBox.skin :

```
<asp:TextBox
SkinID=”TextBoxRouge”
BorderStyle=”Solid”
BorderWidth=”5px”
BackColor="red"
Runat=”Server” />
```

Notez la propriété "SkinID", nous avons nommé notre skin. Comme ca, dans la page ASP.NET, tous les contrôles TextBox auront un fond jaune et une bordure pointillée sauf ceux défini comme ceci :

```
<asp:TextBox Runat="server" SkinID="TextBoxRouge" />
```

Celui-ci aura une bordure pleine et un fond rouge.

Enfin, si vous décidez de ne pas avoir de thème pour un TextBox particulier dans la page, vous utiliserez la propriété "EnableTheming" :

```
<asp:TextBox Runat="server" EnableTheming="false" />
```

Et la, votre TextBox n'aura aucun thème. Cette propriété (EnableTheming) existe pour tous les contrôles ASP.NET

8.2 Configurer les thèmes dans le fichier web.xml

Plutôt que de configurer le thème sur chaque page, vous pouvez le faire de façon globale dans le fichier web.xml :

```
<configuration>
<system.web>
<pages theme="MonTheme" />
</system.web>
</configuration>
```

Toutes les pages du sites utiliseront donc le thème "MonTheme". Si vous voulez cependant qu'une page en particulier n'utilise pas ce thème, vous pouvez modifier la propriété EnableTheming de la page :

```
<%@ Page Language="C#" EnableTheming="false" %>
```

8.3 Ajouter des feuilles de style CSS aux thèmes

Vous pouvez utiliser les feuilles de styles CSS comme alternatives aux thèmes. L'avantage étant que les feuilles de styles permettent de modifier l'apparence non seulement des contrôles ASP.NET mais également des éléments HTML standards. Si vous placez une feuille de style dans un dossier de thème, cette feuille de style est automatiquement appliquée sur la page qui utilise ce style.

Si par exemple, vous placez trois feuilles Style1.css, Style2.css et Style3.css dans le dossier "MonTheme" et que vous créez une page avec la propriété Theme = "MonTheme"

```
<%@ Page Language="C#" Theme="StyleTheme" %>
```

La page générée aura dans son entête :

```
<link href="App_Themes/MonTheme/Style1.css" type="text/css" rel="stylesheet" />
<link href="App_Themes/MonTheme/Style2.css" type="text/css" rel="stylesheet" />
<link href="App_Themes/MonTheme/Style3.css" type="text/css" rel="stylesheet" />
```

Je préfère cet usage des thèmes plutôt que la création de fichiers skin. N'oubliez pas que chaque propriété d'affichage d'un contrôle ASP.NET est en fait un style CSS. Par exemple, la propriété "**BackColor=red**" sera convertie **en style="background-color:red;",** et ceci sur CHAQUE TextBox de la page et du site, ce qui, in fine, risque de générer des pages très lourdes dans lesquelles chaque élément aura une propriété "style" longue comme le bras.

Je le dis ici, je le répéterai plus tard, plutôt que d'utiliser toutes les propriétés d'affichage des contrôles ASP.NET, essayez a chaque fois que c'est possible d'utiliser un style défini dans une feuille de style a part (et qui ne sera charge qu'une fois).

8.4 Appliquer un thème dynamiquement

On peut, comme on l'a fait avec les MasterPage, affecter un thème a la page dynamiquement :

```
protected void Page_PreInit(object sender, EventArgs e)
{
      Theme = "MonTheme"
```

```
}
```

On initialise dynamiquement la propriété Thème de la page. Une fois encore, cette initialisation doit être faite au tout début de la génération de la page, donc lors de l'évènement PreInit.

9 Créer des contrôles personnalises avec les Web User Controls.

Un Web User Control (Contrôle Utilisateur Web) vous permet de créer un nouveau contrôle a partir de contrôles existants. Vous n'êtes donc plus limités aux seuls contrôles proposés par le Framework ASP,NET.

Imaginez que vous deviez, par exemple, afficher le même formulaire pour saisir les coordonnées d'un visiteur dans de nombreux endroits de votre site. Le formulaire en question est constitué de contrôles TextBox, de Labels, de contrôles de validation pour vérifier que l'adresse est correctement saisie, de boutons d'envoi, etc…

Vous pourriez très bien recopier ce bloc sur chaque page dans laquelle vous voulez insérer un formulaire d'adresse. Outre le fait que vous allez passer du temps à le faire, vous allez être bien embêté le jour où il faudra ajouter un nouveau champ ou en supprimer un.

D'où l'intérêt des **Web User Control** : vous allez placer votre formulaire d'adresse dans un Web User Control "adresse" et utiliser ce user control sur n'importe quelle page de votre site, comme si vous utilisiez un contrôle ASP.NET standard.

9.1 Créer un Web User Control

Commençons par créer un simple User Control qui va juste afficher trois champs de saisie nom, prénom, email. (nous oublierons ici les contrôles de validation pour ne pas alourdir l'exemple).

Pour les utilisateurs de VS2008, cliquez avec le bouton droit sur le dossier dans lequel vous voulez créer votre User Control, cliquez sur Add New Item et choisissez Web User Control (nommons le "Adresse").

Vous noterez au passage qu'au lieu d'avoir une extension .ASPX comme les pages ASP.NET, le contrôle a une extension ASCX. Ce fichier ASCX ne sera jamais accessible directement, il devra être appelé par une page ASPX (nous allons voir ca plus bas).

Code du contrôle :

```
<%@ Control Language="C#" ClassName="adresse" %>

<script runat="server">

</script>

<fieldset>
 <asp:Label ID="Label1" runat="server" Text="Nom : " AssociatedControlID="tbNom" />
 <asp:TextBox ID="tbNom" runat="server" />
 <asp:Label ID="Label2" runat="server" Text="Prenom : " AssociatedControlID="tbPrenom"
/>
 <asp:TextBox ID="tbPrenom" runat="server" />
 <asp:Label ID="Label3" runat="server" Text="Email : " AssociatedControlID="tbEmail" />
 <asp:TextBox ID="tbEmail" runat="server" />
 <asp:Button ID="Button1" runat="server" Text=" OK " />
</fieldset>
```

Ca ressemble beaucoup a une page ASP.NET en plus simple : La première ligne indique les propriétés du contrôle la ou on trouvait les propriétés d'une page.

Suivent des déclarations classiques de contrôles ASP.NET et d'éléments HTML. On ne trouve pas ici tous les éléments html d'une page (HTML, HEAD, BODY, …), ceux-ci se trouveront dans la page dans laquelle ce contrôle sera inclus.

Créons donc maintenant une page:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    </div>
    </form>
</body>
</html>
```

Incluons-y ce contrôle. Petite astuce pour commencer simplement, une fois la page créée dans VSNET 2008, placez-vous en mode design, puis faites un drag and drop du fichier ASCX présent dans l'arborescence de votre site directement sur la page. Vous voyez instantanément le contrôle s'afficher dans la page !

Mieux, revenez en mode "Source" et toutes les initialisations ont été faites pour vous :

```
<%@ Page Language="C#" %>

<%@ Register src="adresse.ascx" tagname="adresse" tagprefix="uc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <uc1:adresse ID="adresse1" runat="server" />

    </div>
    </form>
</body>
</html>
```

Que trouvons-nous de nouveau dans notre page ?

Une nouvelle ligne est apparue :

```
<%@ Register src="adresse.ascx" tagname="adresse" tagprefix="uc1" %>
```

Cette ligne enregistre le User Control dans la page. Trois propriétés sont initialisées :

src	Le chemin du contrôle ASCX
tagname	Le nom avec lequel nous allons utiliser ce contrôle dans la page. Vous pouvez utiliser n'importe quoi, tant qu'à faire autant utiliser le nom du fichier.
tagprefix	Le namespace XML qui va être utilisé pour ce contrôle. Encore une fois on peut utiliser n'importe quelle chaine de caractère. VS2008 utilise par défaut uc1, si vous ajoutez un autre user control dans la page, son tagprefix sera uc2, puis uc3 et ainsi de suite (on va voir comment l'utiliser juste en dessous)

Plus bas dans la page, on trouve la déclaration du user control :

```
<uc1:adresse ID="adresse1" runat="server" />
```

Il est déclaré comme un contrôle ASP.NET standard : ID est sont identifiant, on retrouve la propriété runat. Notez que le tagprefix et le tagname sont utilises ici pour déclarer le contrôle.

9.2 Déclarer un user control dans web.xml

Si vous utilisez un usercontrol souvent, plutôt que de le déclarer dans chaque page avec la directive Register, vous pouvez le faire au niveau du fichier web.xml.

```
<configuration>
<system.web>
<pages>
<controls>

     <add tagPrefix="uc1" tagName="adresse" src="adresse.ascx"/>

</controls>
</pages>
</system.web>
</configuration>
```

Vous pourrez alors utiliser ce contrôle dans n'importe quelle page sans avoir besoin de le déclarer explicitement en début de page.

9.3 Exposer les propriétés d'un user control

En exposant certaines propriétés du UserControl, vous pourrez, à partir de la page qui l'héberge, le configurer soit par programme, soit directement lors de sa déclaration.

Créons une propriété a notre user control permettant de preremplir le champ adresse email :

```
<%@ Control Language="C#" ClassName="adresse" %>

<script runat="server">
    public String adresseEmail
    {
        get { return tbEmail.Text; }
        set { tbEmail.Text = value; }
    }

</script>

<fieldset>
<asp:Label ID="Label1" runat="server" Text="Nom : " AssociatedControlID="tbNom" />
<asp:TextBox ID="tbNom" runat="server" />
<asp:Label ID="Label2" runat="server" Text="Prenom : " AssociatedControlID="tbPrenom"
/>
<asp:TextBox ID="tbPrenom" runat="server" />
<asp:Label ID="Label3" runat="server" Text="Email : " AssociatedControlID="tbEmail" />
<asp:TextBox ID="tbEmail" runat="server" />
<asp:Button ID="Button1" runat="server" Text=" OK " />
</fieldset>
```

Nous avons crée une propriété "adresseEmail" qui permet d'accéder a la propriété Text du contrôle "tbEmail"

```
public String adresseEmail
{
    get { return tbEmail.Text; }
    set { tbEmail.Text = value; }
}
```

Nous pouvons maintenant y accéder de deux façons dans notre page ASP.NET

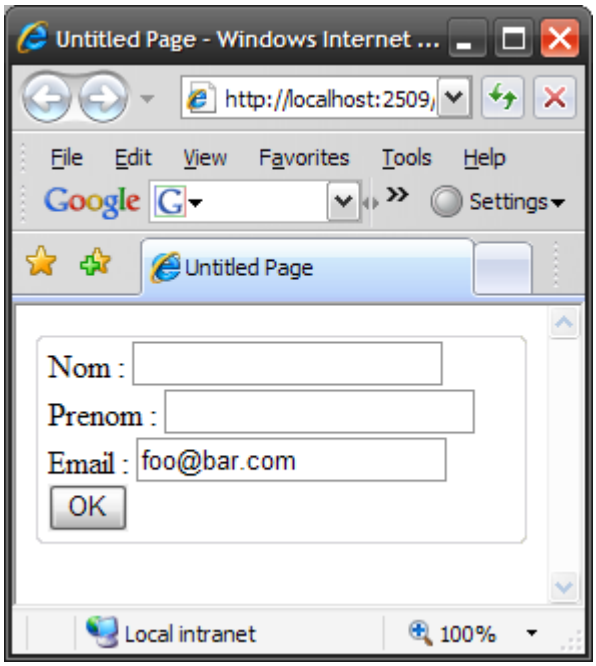
1) sous forme déclarative, en modifiant la déclaration du contrôle :

```
<uc1:adresse ID="adresse1" runat="server" adresseEmail="foo@bar.com" />
```

2) par code :

```
<script runat="server">

void Page_Load()
{
    Adresse1.adresseEmail = "foo@bar.com";
}

</script>
```

Dans les deux cas, le résultat sera :



9.4 Exposer un évènement d'un UserControl

En plus d'exposer des propriétés d'un user control, vous pouvez également exposer des évènements. Lorsque quelque chose se produira dans le User Control (vous déterminerez quoi), un évènement se déclenchera, cet évènement étant expose, une fonction spécifique pourra être appelée dans la page principale (celle qui héberge le user control).

Dans notre exemple, créons un évènement "envoyerAdresse" qui se déclenchera lorsque l'utilisateur cliquera sur le bouton OK du formulaire d'adresse.

```
<%@ Control Language="C#" ClassName="adresse" %>

<script runat="server">
    public event EventHandler envoyerAdresse;

    public String adresseEmail
    {
        get { return tbEmail.Text; }
        set { tbEmail.Text = value; }
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        envoyerAdresse(this, EventArgs.Empty);
    }
</script>

<fieldset>
<asp:Label ID="Label1" runat="server" Text="Nom : " AssociatedControlID="tbNom" />
<asp:TextBox ID="tbNom" runat="server" />
<asp:Label ID="Label2" runat="server" Text="Prenom : " AssociatedControlID="tbPrenom"
/>
<asp:TextBox ID="tbPrenom" runat="server" />
<asp:Label ID="Label3" runat="server" Text="Email : " AssociatedControlID="tbEmail" />
<asp:TextBox ID="tbEmail" runat="server" />
<asp:Button ID="Button1" runat="server" Text=" OK " onclick="Button1_Click" />
</fieldset>
```

Qu'avons-nous de nouveau ?

```
public event EventHandler envoyerAdresse;
```

On a défini et expose ici un évènement nomme "envoyerAdresse" (avec le mot clef "event")

```
protected void Button1_Click(object sender, EventArgs e)
{
    envoyerAdresse(this, EventArgs.Empty);
}
```

Et créé cette fonction lorsqu'on clique sur le bouton OK du formulaire. Cette fonction appelle la fonction associée a l'évènement "envoyerAdresse".

Comment va-t-on l'utiliser dans la page ?

```
<%@ Page Language="C#" %>
<%@ Register src="adresse.ascx" tagname="adresse" tagprefix="uc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

    protected void envoi(Object sender, EventArgs args)
    {
        ...
    }

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<body>
    <form id="form1" runat="server">
      <uc1:adresse ID="adresse1" runat="server" adresseEmail="foo@bar.com"
       OnenvoyerAdresse="envoi" />
    </form>
</body>
</html>
```

Simplement en initialisant une nouvelle propriété lors de la déclaration du User Control :

```
<uc1:adresse ID="adresse1" runat="server" adresseEmail="foo@bar.com"
OnenvoyerAdresse="envoi" />
```

Une nouvelle propriété "OnenvoyerAdresse" a été automatiquement créée, elle définit la fonction qui sera appelle lorsque l'évènement "envoyerAdresse" se produira. La fonction en question est définie plus haut :

```
protected void envoi(Object sender, EventArgs args)
{
    ...
}
```

Remplacez "…" par votre code ;-)

9.5 Charger dynamiquement des User Controls

Vous n'êtes pas obligé de passer nécessairement par la déclaration du User Control dans votre page pour l'utiliser. Vous pouvez charger le User Control durant la génération de la page. Supposons par exemple qu'en fonction d'un paramètre envoyé a la page, vous vouliez charger un User control qui va afficher un formulaire demandant à l'utilisateur de saisir son adresse ou si il est déjà enregistre, un autre User Control qui affichera un formulaire demandant a l'utilisateur d'entrer son nom et son mot de passe. (Nous verrons plus tard qu'il y a moyen de faire ca bien plus simplement, mais c'est juste un exemple).

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

    protected void Page_Load(object sender, EventArgs e)
    {
        Control userControl;
        if (Request["typeUtilisateur"] == "nonEnregistre")
          userControl =  Page.LoadControl("adresse.ascx");
        else userControl = Page.LoadControl("motDePasse.ascx");
        PlaceHolder1.Controls.Add(userControl);
    }

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<body>
    <form id="form1" runat="server">
        <asp:PlaceHolder ID="PlaceHolder1" runat="server"></asp:PlaceHolder>
    </form>
</body>
</html>
```

Que fait le programme ?

```
if (Request["typeUtilisateur"] == "nonEnregistre")
    userControl = Page.LoadControl("adresse.ascx");
else userControl = Page.LoadControl("motDePasse.ascx");
```

Si le paramètre "TypeUtilisateur" de la page = "nonEnregistre" , le user control "adresse.ascx" est chargé sinon, c'est le user control "motDePasse.ascx" (vous devez indiquez ici le chemin complet du user control).
(le paramètre de la page est passe dans l'url : http://mapage?typeUtilisateur=nonEnregistre).

Puis, on ajoute ce contrôle au contrôle PlaceHolder "placeHolder1", la page est donc affichée avec le bon user control.

Au passage, vous avez noté qu'on à utilisé un nouveau contrôle ASP.NET : le contrôle **PlaceHolder**. Ce contrôle ne fait rien et n'affiche rien par défaut. On s'en sert comme d'une marque dans la page, si on veut, comme dans notre cas, charger dynamiquement un contrôle, on utilisera ce contrôle pour savoir ou placer sur la page le contrôle charge dynamiquement. C'est ce que fait la ligne :

```
PlaceHolder1.Controls.Add(userControl);
```

On ajoute (méthode Add) à la liste des contrôles contenus dans le placeHolder (propriété **Controls**) le nouveau contrôle fraichement charge.

Vous noterez que la fonction LoadControl renvoie un Object. Si vous voulez initialiser une des propriétés du User Control charge, il faudra préalablement le caster vers le type d'objet avec lequel vous voulez travailler :

```
adresse userControlAdresse = (adresse)Page.LoadControl("adresse.ascx");
userControlAdresse.adresseEmail = "foo@bar.com";
```

Cependant, vous ne pourrez pas faire ce casting si le contrôle "adresse" n'a pas été préalablement enregistre sur la page, soit par :

```
<%@ Register src="adresse.ascx" tagname="adresse" tagprefix="uc1" %>
```

En haut de la page, ou, comme nous l'avons vu plus haut, dans le fichier web.xml.

10 ADO.NET et le DataBinding

ASP.NET vous permet de lier de façon très simple des données provenant de sources diverses dans des contrôles. Ces contrôles afficheront les données sous différents formats, vous permettrons de les modifier, les supprimer, etc…

Ces contrôles, capables de manipuler des données pour les afficher et / ou les modifier sont divises en 3 grandes familles :

- Les contrôles "Liste" : listes déroulantes, listes de cases a cochets, listes de boutons radio, …
- Les contrôles "Tabulaires" : listes affichées sous formes de tableaux, des plus simples au plus élabores
- Les contrôles "Hiérarchiques" : typiquement des arbres

Les données peuvent être donc récupérées dans ce qu'on appelle des sources de données (Data Sources). Les data sources utilisables sont :

- SqlDataSource : une base de données : Le framework propose par défaut 3 types de Data Sources SQL : SQL Server, OLEDB et ODBC. De façon générale, chaque fournisseur de base de données propose gratuitement un connecteur .NET (MySQL, Oracle, DB2, …)
- LinqDataSource : une source de données Linq (nous verrons cela plus loin)
- AccessDataSource : une source de données Microsoft Access.
- ObjectDataSource : une source de données provenant d'un objet métier (une classe)
- XmlDataSource : une source de données provenant d'un document XML
- SiteMapDataSource : la source de données provient d'un Site Map Provider. Un Site Map Provider est un objet représentant la structure d'un site web (particulièrement utilise avec un contrôle de type hiérarchique)

Nous allons commencer par les contrôles de type Liste, les plus simples, ces contrôles serviront a introduire la façon dont se fait le DataBinding (la liaison des données entre la source et le contrôle servant a les manipuler).

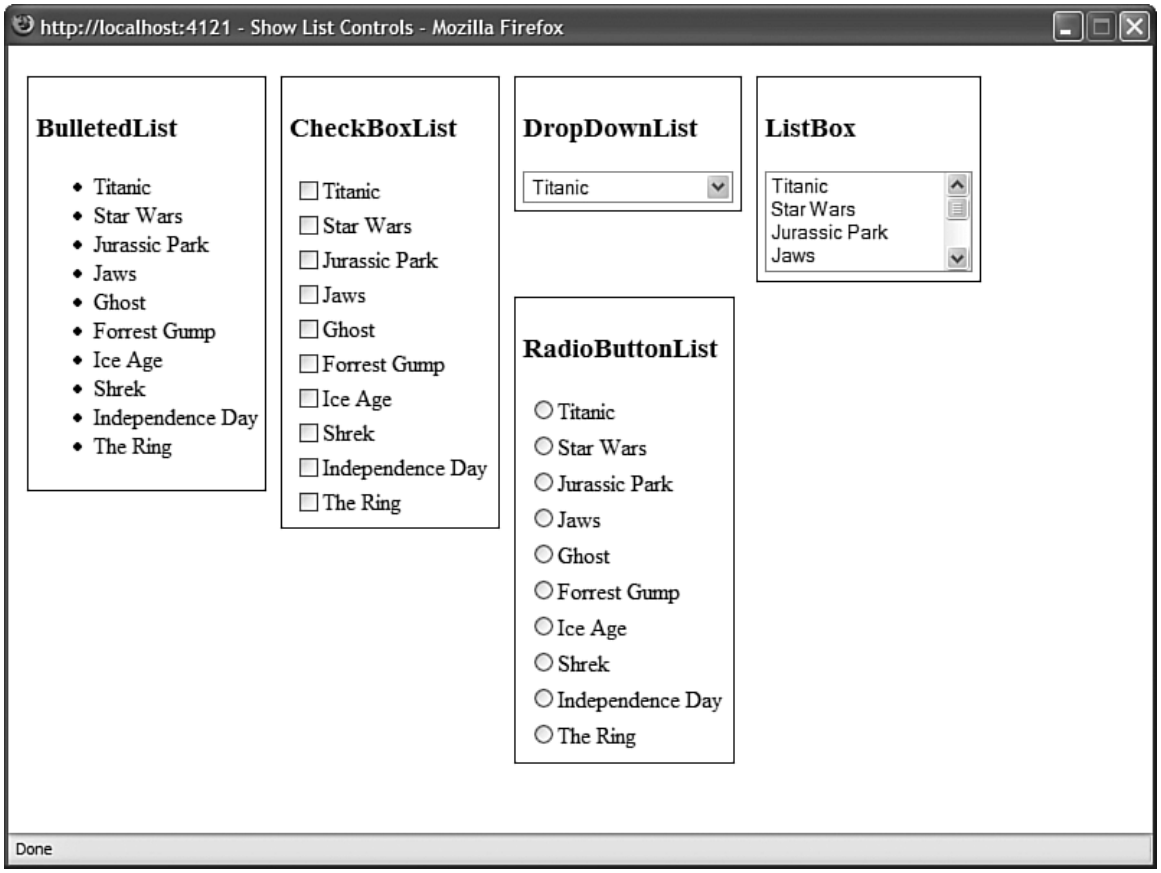
10.1 Les contrôles Liste

Les contrôles de type Liste sont utilises pour afficher simplement des données. Le Framework 3.5 inclus les types de liste suivantes (ces listes fonctionnent quasiment toutes sur le même modèle) :

- BulletedList : Une liste ou chaque élément est précédé d'un point. Chaque élément peut être affiché comme du texte, un bouton ou un lien.
- CheckBoxList : Une liste de cases à cocher. Plusieurs cases peuvent être cochées.
- DropDownList : Une liste déroulante. Un seul élément de la liste peut être sélectionné.
- List Box : Une liste (qui peut avoir un ascenseur si son contenu est important). Vous pouvez configurer ce contrôle pour que l'utilisateur ne puisse sélection qu'un élément ou, au contraire, plusieurs.
- RadioButtonList : Affiche une liste de boutons radio. Seul un bouton radio peut être sélectionné a la fois.

Ces 5 contrôles dérivent tous d'une classe de base "ListControl", ce qui signifie qu'ils partagent la plupart de leurs propriétés et méthodes.

Une petite image présentant ces contrôles affichant les données provenant de la même source :

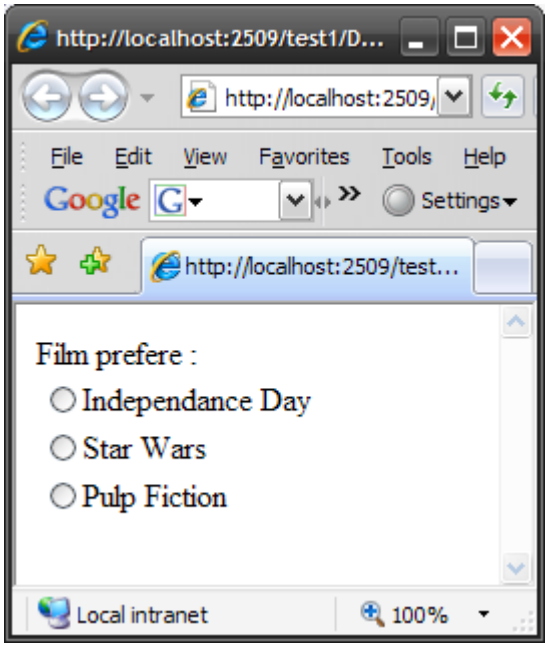


10.2 Création d'une Liste

Dans l'exemple qui suit, nous allons créer une liste affichant une liste de films, on demande a l'utilisateur de choisir son film préféré. Pour l'instant, la liste n'est pas liée a une source de données, les films sont codés "en dur" dans la page :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
<form id="form1" runat="server">
<asp:Label id="lbl" Text="Film prefere : " AssociatedControlID="listeFilms"
Runat="server" />
<asp:RadioButtonList id="listeFilms" Runat="server">
    <asp:ListItem Text="Independance Day" Value="film1" />
    <asp:ListItem Text="Star Wars" Value="film2" />
    <asp:ListItem Text="Pulp Fiction" Value="film3" />
</asp:RadioButtonList>
</form>
</body>
</html>
```

Ce qui donne :



Chaque liste est composée d'une liste de contrôles ListItem.

Le contrôle ListItem ne peut être présent que dans une liste. Il possède les propriétés suivantes :

Attributes	Vous permet d'ajouter des attributs HTML a l'élément
Enabled	Active / Désactive le contrôle
Selected	Sélectionne l'élément
Text	Texte de l'élément
Value	Valeur de l'élément

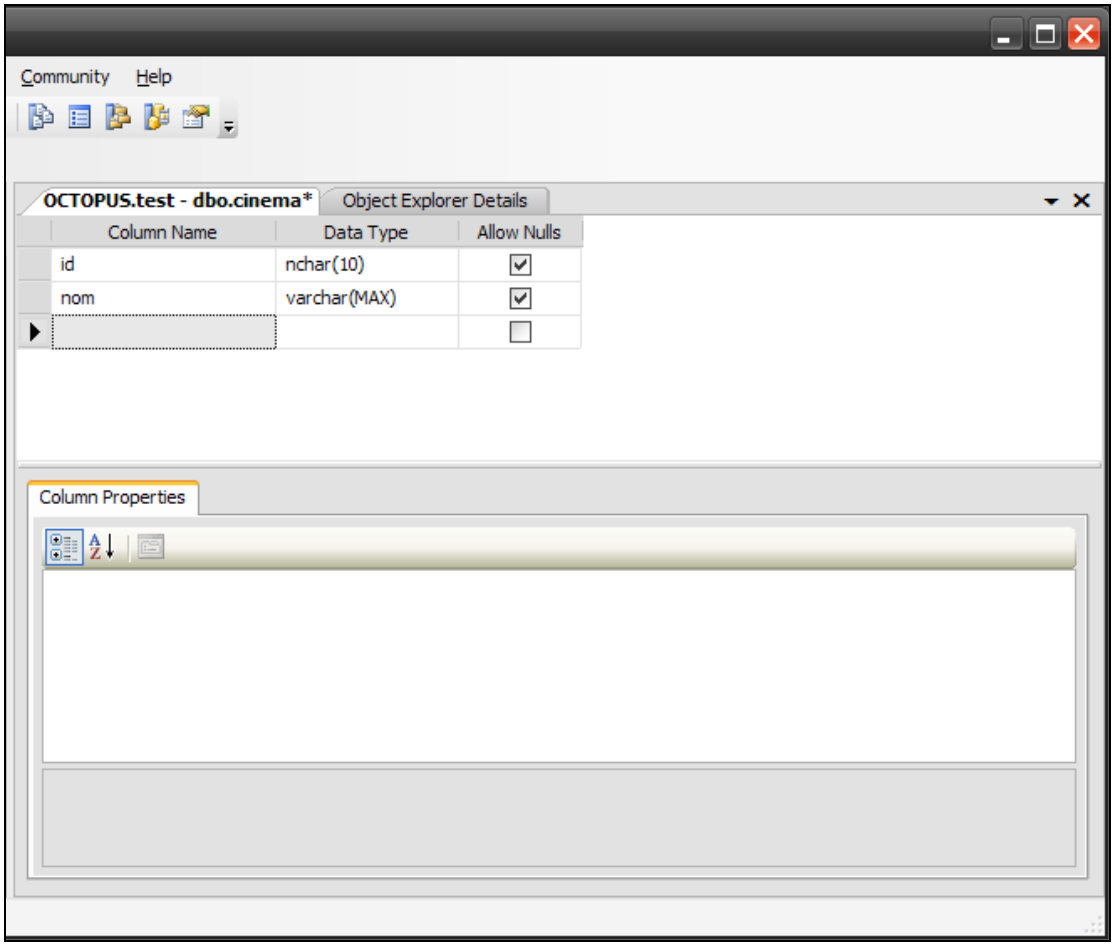
On va corser l'affaire et aller chercher cette liste dans une base de données. Bon, petite digression. Il va falloir que vous installiez sur votre machine un serveur de base de données. On pourrait installer une base open source comme MySQL mais pour éviter toute éventuelle incompatibilité avec le connecteur .NET, votre Framework ou je ne sais quoi, nous allons utiliser pour nos essais SQL Server 2005.
Vous pouvez télécharger SQL Server Express 2005 gratuitement sur le site de Microsoft a cette adresse :

http://msdn.microsoft.com/fr-fr/express/aa975050.aspx

Je ne vais pas entrer dans le détail de l'installation et de l'utilisation de SQL Server 2005 ici. Vous trouverez les détails sur l'installation ici :

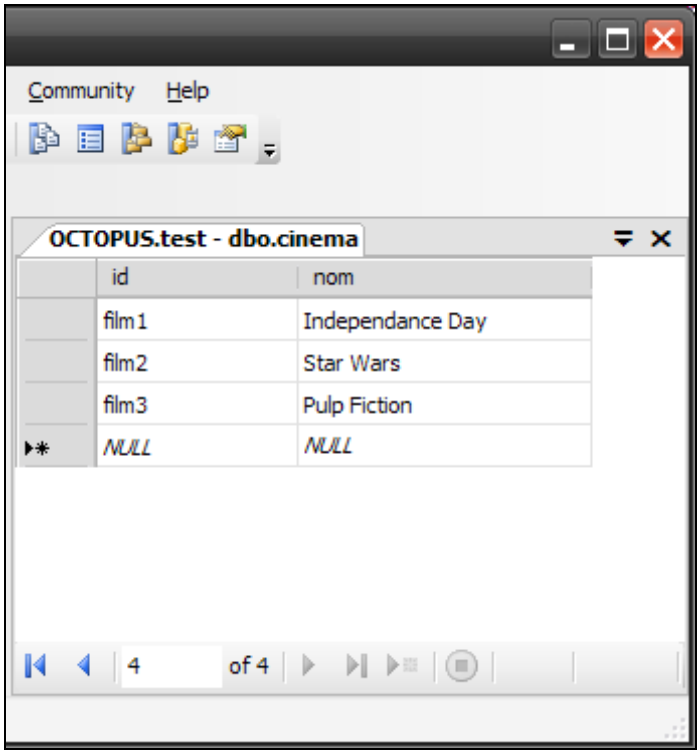
http://fromelard.free.fr/Scripts/SQL_Express/SQL_Express_Install_FR.pdf

Créez donc juste une base "Test" et une table "cinema" ayant la structure suivante :



Recommençons donc avec cette fois-ci une liaison a une base de données pour aller lire la liste des films.

Commencez par renseigner la table :



Modifions donc la page, elle va devenir plus simple (et plus compliquée) :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
<form id="form2" runat="server">
   <asp:Label id="lbl" Text="Film prefere : "  AssociatedControlID="listeFilms"
    Runat="server" />
   <asp:RadioButtonList id="listeFilms" Runat="server"
    DataSourceID="sourceFilms"
    DataTextField="nom" DataValueField="id" />

    <asp:SqlDataSource id="sourceFilms"
     SelectCommand="SELECT id, nom FROM cinema"
     ConnectionString="Data Source=octopus;Initial Catalog=test;UserId=user;
     Password=password;" Runat="server" />
</form>
</body>
</html>
```

Bien, prenons les choses dans l'ordre :

Nous avons d'abord modifié le contrôle RadioButtonList :

```
<asp:RadioButtonList id="listeFilms" Runat="server" DataSourceID="sourceFilms"
DataTextField="nom" DataValueField="id" >
</asp:RadioButtonList>
```

Il n'y a plus de ListItem (un peu normal, nous allons générer le contenu a partir de la base), et trois nouvelles propriétés sont apparues :

```
DataSourceID="sourceFilms"
```

Cette propriété va indiquer l'identifiant de la source de données qui va remplir cette liste (on va la voir après)

```
DataTextField="nom"
DataValueField="id"
```

La source de données va renvoyer des données (en l'occurrence ici des données venant d'une table), la propriété DataTextField va indiquer quel est le champ de la table qui va renseigner la propriété Text de l'élément de la liste (ce qui sera affiche après le bouton radio) et la propriété "DataFieldValue" va indiquer quelle est le champ qui contiendra la valeur de cet élément de la liste.

Plus bas, on trouve la DataSource, il s'agit d'un contrôle SqlDataSource, c'est-à-dire que la source de données est une base de données Microsoft SQL Server.

```
<asp:SqlDataSource
id="sourceFilms"
SelectCommand="SELECT id, nom FROM cinema"
ConnectionString="Data Source=octopus;Initial Catalog=test;User
Id=user;Password=password;"
Runat="server" />
```

La data source contient les propriétés suivantes :

SelectCommand	La requête SQL
ConnectionString	La chaine de connexion sur la base de données

La chaine de connexion sur la base est une chaine de caractères qui décrit comment se connecter sur la base (le nom du serveur, de la base, de l'utilisateur, du mot de passe).
La chaine de connexion sur les bases varie en fonction du type de serveur de base de données. Vu le nombre de SGBD, je préfère vous envoyer sur le site connectionstrings.com qui indique, pour chaque utilisation et chaque SGBD, le format de la chaine de connexion.

http://www.connectionstrings.com

Notez que pour simplifier, vous pouvez stocker la chaine de connexion dans le fichier web.xml, vous n'aurez pas a la répéter partout dans le site :

Vous vous souvenez que nous avions vu que le fichier web.xml contient une section <connectionstrings>
Dans cette section, vous allez pouvoir définir vos chaines de connexion :

```
<configuration>
<connectionStrings>
<add name=”films” connectionString=” Data Source=octopus;Initial Catalog=test;User
Id=user;Password=password;"” />
</connectionStrings>
</configuration>
```

Et pour l'utiliser dans votre contrôle SqlDataSource :

```
<asp:SqlDataSource
id="sourceFilms"
SelectCommand="SELECT id, nom FROM cinema"
ConnectionString="<%$ ConnectionStrings: films %>"
Runat="server" />
```

Il y a un autre avantage à stocker la chaine de connexion à un seul endroit. Le Framework va placer les connexions dans un pool. A chaque fois qu'une connexion est demandée, plutôt que d'en recréer une autre, il va aller chercher dans le pool si la connexion n'a pas déjà été préalablement créée et ouverte et si c'est le cas, l'utilisera. La clef de recherche dans le pool est la chaine de connexion, si vous accédez a la même base avec une chaine variant légèrement (un caractère en plus ou en moins, un espace supplémentaire), une nouvelle connexion sera créée. Centraliser les chaines de connexion assure donc leur unicité.

10.3 La même chose par programmation

Nous venons de voir comment faire le data binding de façon déclarative directement dans la page ASP.NET. Nous pouvons faire la même chose par programme :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head>
<script runat="server">

protected void Page_Load(object sender, EventArgs e)
{
    String cnx =
    ConfigurationManager.ConnectionStrings["films"].ConnectionString;
    SqlDataSource ds = new SqlDataSource(cnx, "select id, nom from cinema");
    listeFilms.DataTextField = "nom";
    listeFilms.DataValueField = "id";
    listeFilms.DataSource = ds;
    listeFilms.DataBind();
}

</script>

</head>
<body>
<form id="form2" runat="server">
<asp:Label id="lbl" Text="Film prefere : " AssociatedControlID="listeFilms"
Runat="server" />
<asp:RadioButtonList id="listeFilms" Runat="server"></asp:RadioButtonList>
</form>
</body>
</html>
```

Nous avons enlevé le contrôle SqdDataSource de la page ASP.NET, et supprimé toute les références a cet objet dans le contrôle RadioButtonList, puis, dans le source associe a la page :

Ici nous allons chercher la chaine de connexion dans la section ConnectionStrings du fichier web.config. On accède aux chaines en utilisant leur nom comme index du tableau.

```
String cnx = ConfigurationManager.ConnectionStrings["films"].ConnectionString;
```

Ici on crée la datasource, au passage on indique la requête select associée a cette data source :

```
SqlDataSource ds = new SqlDataSource(cnx, "select id, nom from cinema");
```

Ici on initialise notre radiobuttonlist, en indiquant quel champ de la table va être le champ texte, quel champ va être le champ valeur et quelle est la data source

```
listeFilms.DataTextField = "nom";
listeFilms.DataValueField = "id";
listeFilms.DataSource = ds;
```

Enfin, on initialise les données de la liste en allant aller les chercher via la connexion. En faisant un data bind, la requête select de la connexion est appelée, et le résultat initialise le contrôle associe.

```
listeFilms.DataBind();
```

Facile !

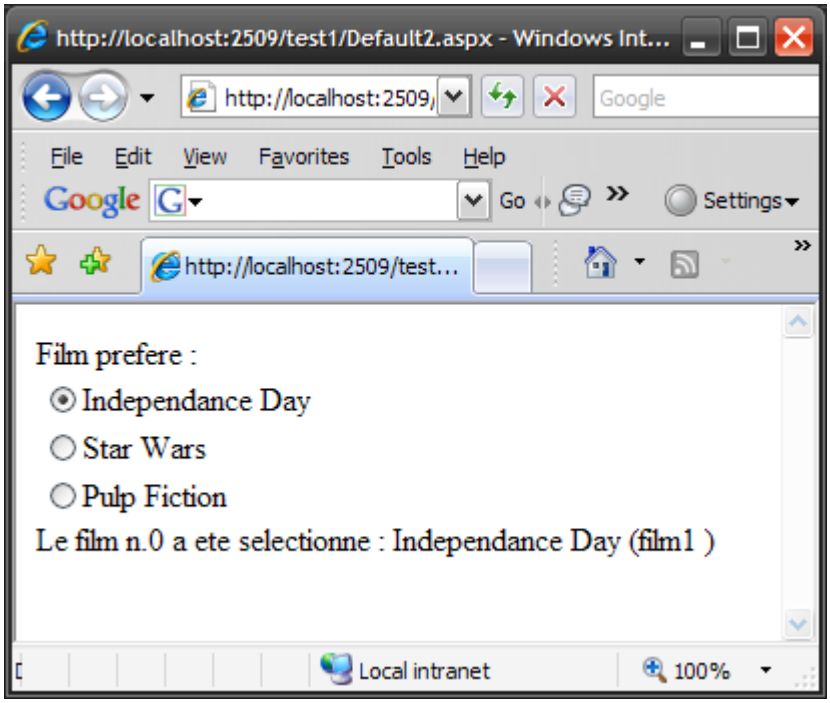
Une fois affichée, il faut récupérer les données sélectionnées par l'utilisateur :

SelectedIndex	Récupère l'index de la ligne sélectionnée
SelectedItem	Recupere l'item complet
SelectedValue	Recupere la valeur sélectionnée

(Notez que **selectedindex** et **selectedvalue** fonctionnent en lecture et en écriture, vous sélectionnez un item particulier de la liste en initialisant la propriété selectedindex ou selectedvalue).

Petit exemple :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head>
<script runat="server">

protected void listeFilms_SelectedIndexChanged(object sender, EventArgs e)
{
Label1.Text = "Le film n." + listeFilms.SelectedIndex + " a ete selectionne : " +
listeFilms.SelectedItem.Text + " (" + listeFilms.SelectedValue + ")";
}
</script>

</head>
<body>
<form id="form2" runat="server">
<asp:Label id="lbl" Text="Film prefere : " AssociatedControlID="listeFilms"
Runat="server" />

<asp:SqlDataSource
id="sourceFilms"
SelectCommand="SELECT id, nom FROM cinema"
ConnectionString="Data Source=octopus;Initial Catalog=test;User
Id=sa;Password=password;"
Runat="server" />
<asp:Label ID="Label1" runat="server"></asp:Label>
</form>
</body>
</html>
```



Quelles sont les changements ?

La liste d'abord, on a initialisé deux nouvelles propriétés :

```
onselectedindexchanged="listeFilms_SelectedIndexChanged"
```

Une fonction est affectée a l'évènement **selectedindexchanged**. Cet évènement se produit lorsqu'on clique sur un des boutons de la liste (en fait lorsqu'on change l'index de la sélection).

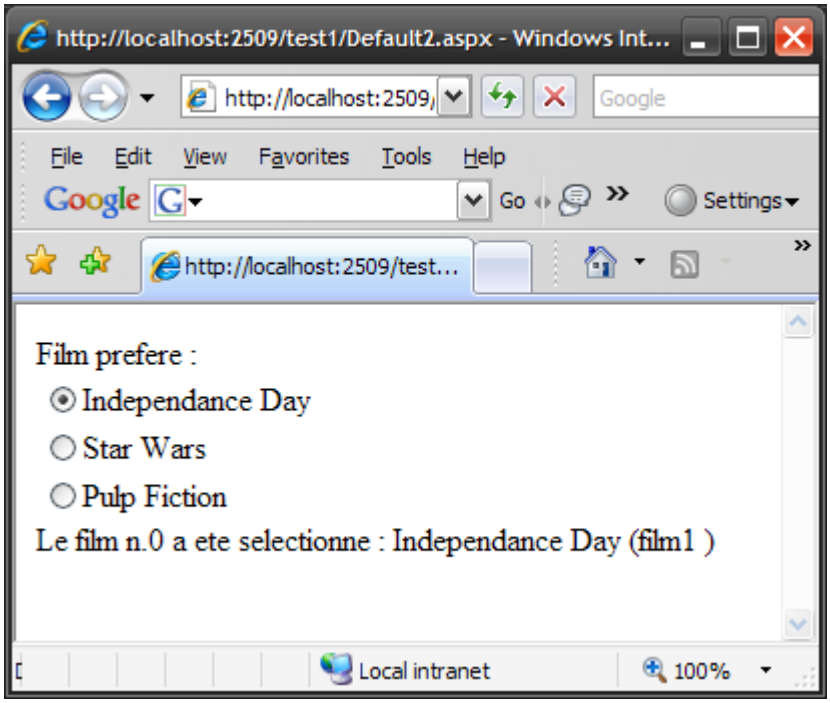
```
AutoPostBack="True"
```

Que nous avons déjà rencontré plus haut : le fait de cliquer sur un des radio buttons va provoquer un postback (nécessaire pour retourner sur le serveur pour exécuter notre fonction associée au click)

Et enfin notre code C#

```
protected void listeFilms_SelectedIndexChanged(object sender, EventArgs e)
{
    Label1.Text = "Le film n." + listeFilms.SelectedIndex + " a ete selectionne : " +
listeFilms.SelectedItem.Text + " (" + listeFilms.SelectedValue + ")";
}
```

La fonction se contente de récupérer l'index de la sélection (commence toujours a 0), le texte de l'élément sélectionné et sa valeur :



Nous allons donc maintenant faire le tour de tous les contrôles ASP.NET supportant le data binding, ce premier tour d'horizon n'utilisera que la commande SQL Select (nous lirons donc uniquement les données pour les afficher). Nous les modifierons par la suite.

Commençons avec les autres contrôles ListControl.

Nous avons vu que la RadioButtonList fait partie de cette famille. Il y a d'autres membres qui fonctionnent exactement de la même façon, il suffit juste de remplacer le nom du contrôle. Exemple, vous voulez une liste déroulante a la place de la liste de boutons radio ?

Remplacez juste :

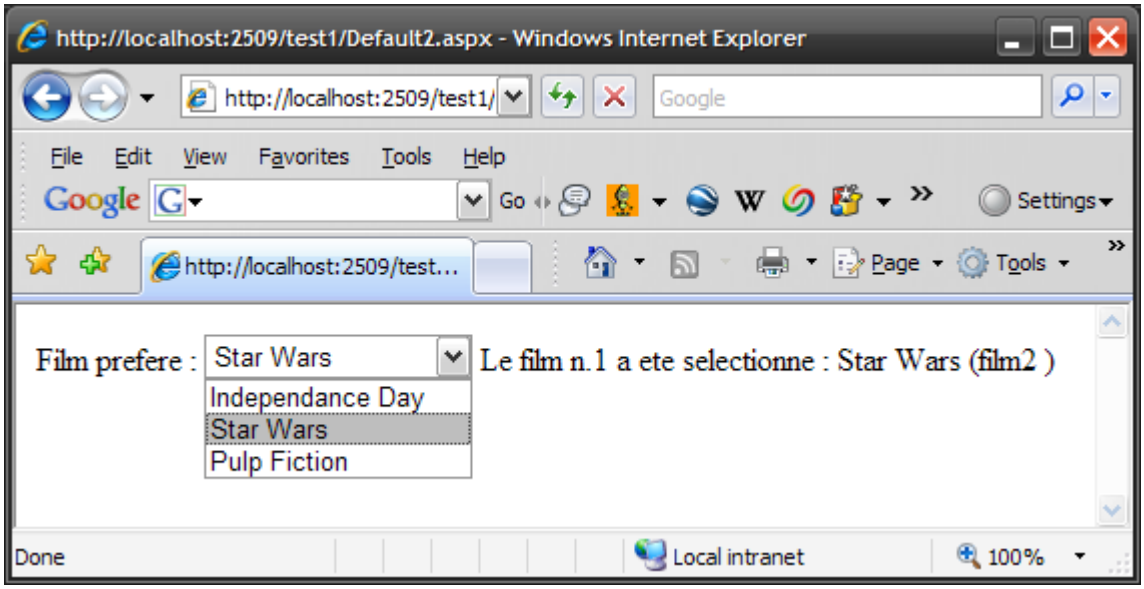
```
<asp:RadioButtonList id="listeFilms" Runat="server" DataSourceID="sourceFilms"
DataTextField="nom" DataValueField="id" >
```

```
</asp:RadioButtonList>
```

Par

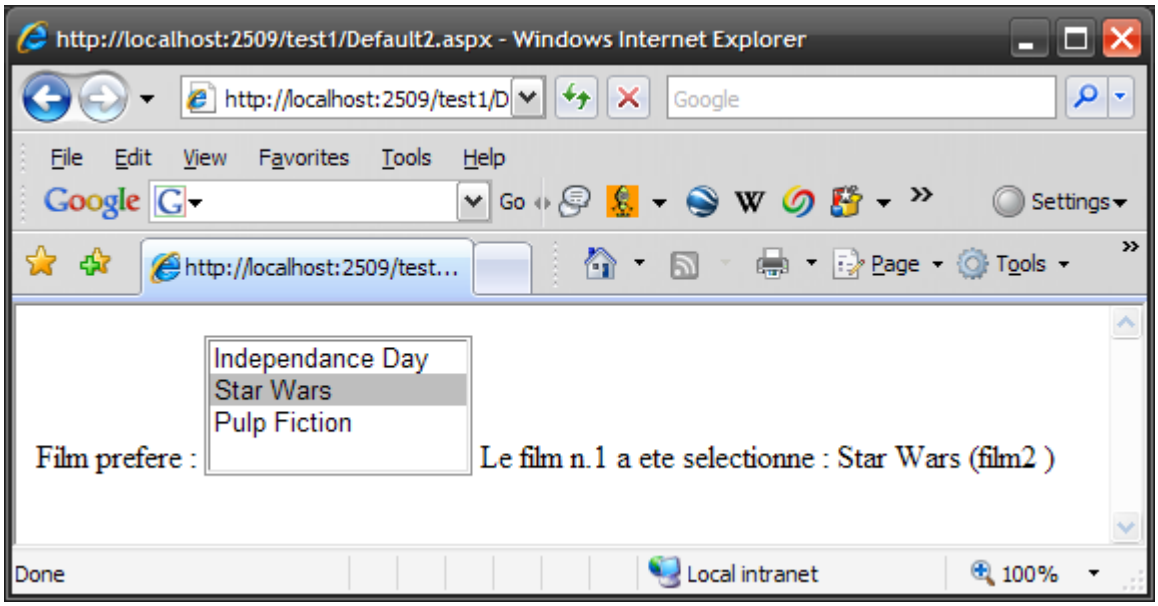
```
<asp:DropDownList id="listeFilms" Runat="server" DataSourceID="sourceFilms"
DataTextField="nom" DataValueField="id" >
</asp:DropDownList>
```

Et vous obtenez :



Vous pouvez également utiliser une liste de sélection avec le contrôle ListBox :

```
<asp:ListBox id="listeFilms" Runat="server" DataSourceID="sourceFilms"
DataTextField="nom" DataValueField="id" >
</asp:ListBox>
```



Notez cependant que le contrôle ListBox a deux propriétés supplémentaires intéressantes :

Rows	Indique le nombre de lignes affichables dans la liste, si par exemple, vous avez une liste de 50 films et que Rows = 4, seuls 4 films seront visibles a la fois, il faudra utiliser l'ascenseur de la liste pour faire défiler la liste.
SelectionMode	Permet de déterminer si l'utilisateur peut sélectionner une seule ligne ou plusieurs : Single (une seule) ou Multiple (plusieurs)

Essayons avec SelectionMode = Multiple

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head>
<script runat="server">

protected void Button1_Click(object sender, EventArgs e)
{
    Literal1.Text = "";
    foreach (ListItem item in listeFilms.Items )
    {
    if (item.Selected) Literal1.Text += "Le film : " + item.Text + " est
selectionne.<br>";
    }
}
</script>

</head>
<body>
<form id="form2" runat="server">
  <asp:Label id="lbl" Text="Film prefere : "
   AssociatedControlID="listeFilms"
   Runat="server" />
  <asp:ListBox id="listeFilms" Runat="server" DataTextField="nom"
   DataValueField="id" SelectionMode="Multiple"
   DataSourceId="sourceFilms" />
  <asp:Button ID="Button1" runat="server" Text="valider"
   onclick="Button1_Click"  />
  <asp:SqlDataSource id="sourceFilms"
   SelectCommand="SELECT id, nom FROM cinema"
   ConnectionString="Data Source=octopus;Initial Catalog=test;User
   Id=sa; Password=password;" Runat="server" />
  <br /><br />
  <asp:Literal ID="Literal1" runat="server" />
</form>
</body>
```

```
</html>
```

Les changements :

```
<asp:ListBox id="listeFilms" Runat="server" DataTextField="nom"
    DataValueField="id" SelectionMode="Multiple" DataSourceId="sourceFilms">
```

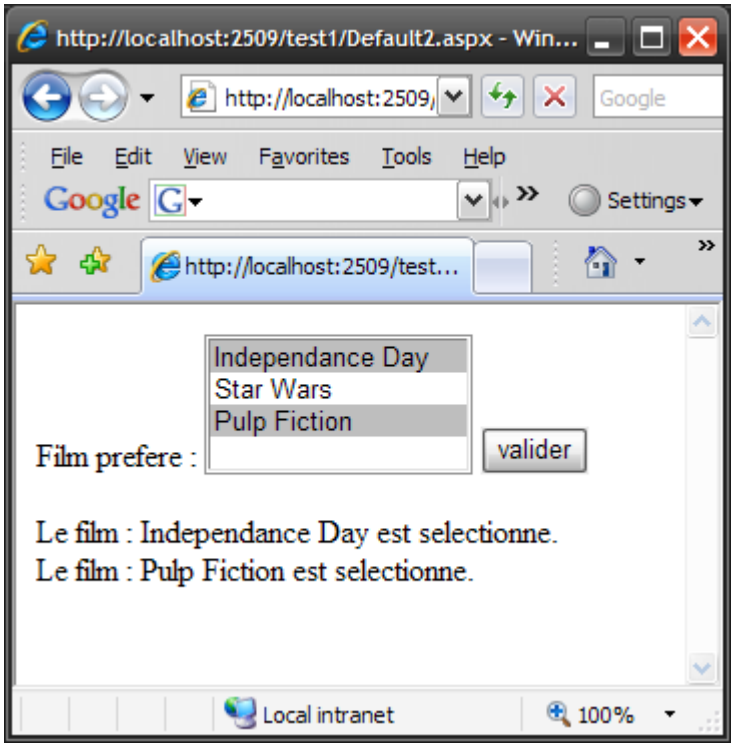
La propriété **SelectionMode** est initialisée, vous notez qu'on a supprimé les propriétés **AutoPostBack** et la fonction associée a l'évènement **selectedIndex**. Pourquoi ? Parce que cet évènement se déclenche dès que l'utilisateur clique sur une ligne, si nous le laissons, le postback se fera dès la première sélection même si l'utilisateur veut en sélectionner une autre, ce qui n'est pas spécialement pratique.

```
<asp:Button ID="Button1" runat="server" Text="valider" onclick="Button1_Click"  />
```

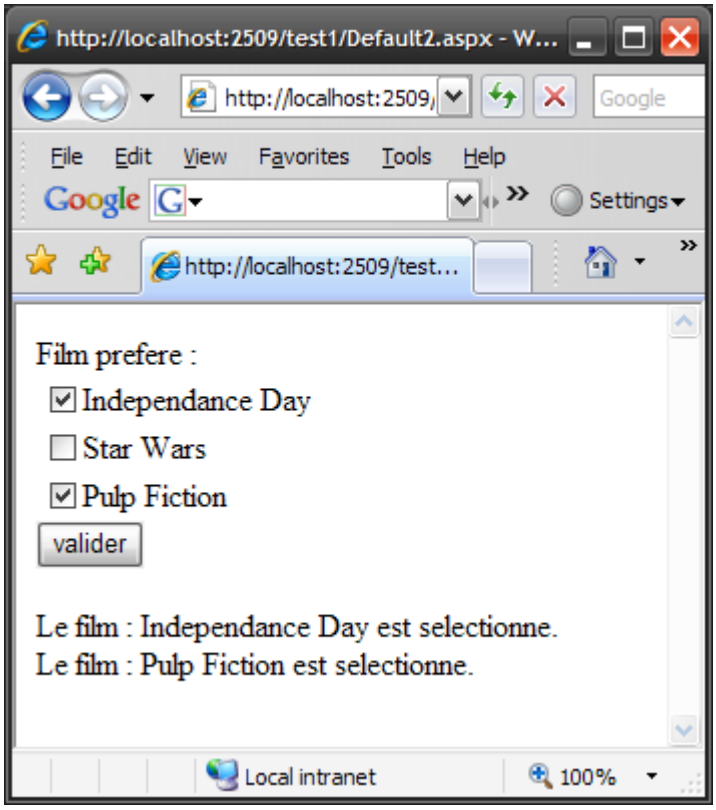
Nous avons ajoute un bouton pour faire le postback : l'utilisateur sélectionné les lignes et clique sur ce bouton.

```
protected void Button1_Click(object sender, EventArgs e)
{
    Literal1.Text = "";
    foreach (ListItem item in listeFilms.Items )
    {
        if (item.Selected) Literal1.Text += "Le film : " + item.Text + " est
selectionne.<br>";
    }
}
```

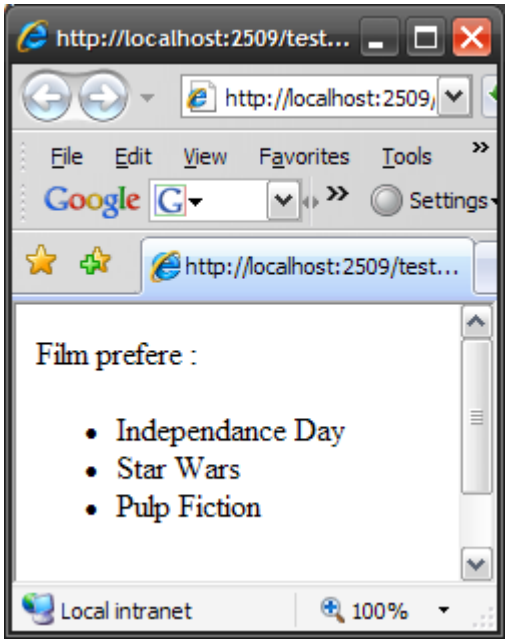
L'évènement associé au click sur le bouton parcours la items de la liste et lorsque l'un d'entre eux est sélectionné, affiche le film en question :



On peut remplacer la listBox par une CheckBoxList :



Le dernier de la famille est la BulletedList. Elle se contente d'afficher la liste, aucune interaction n'est possible :



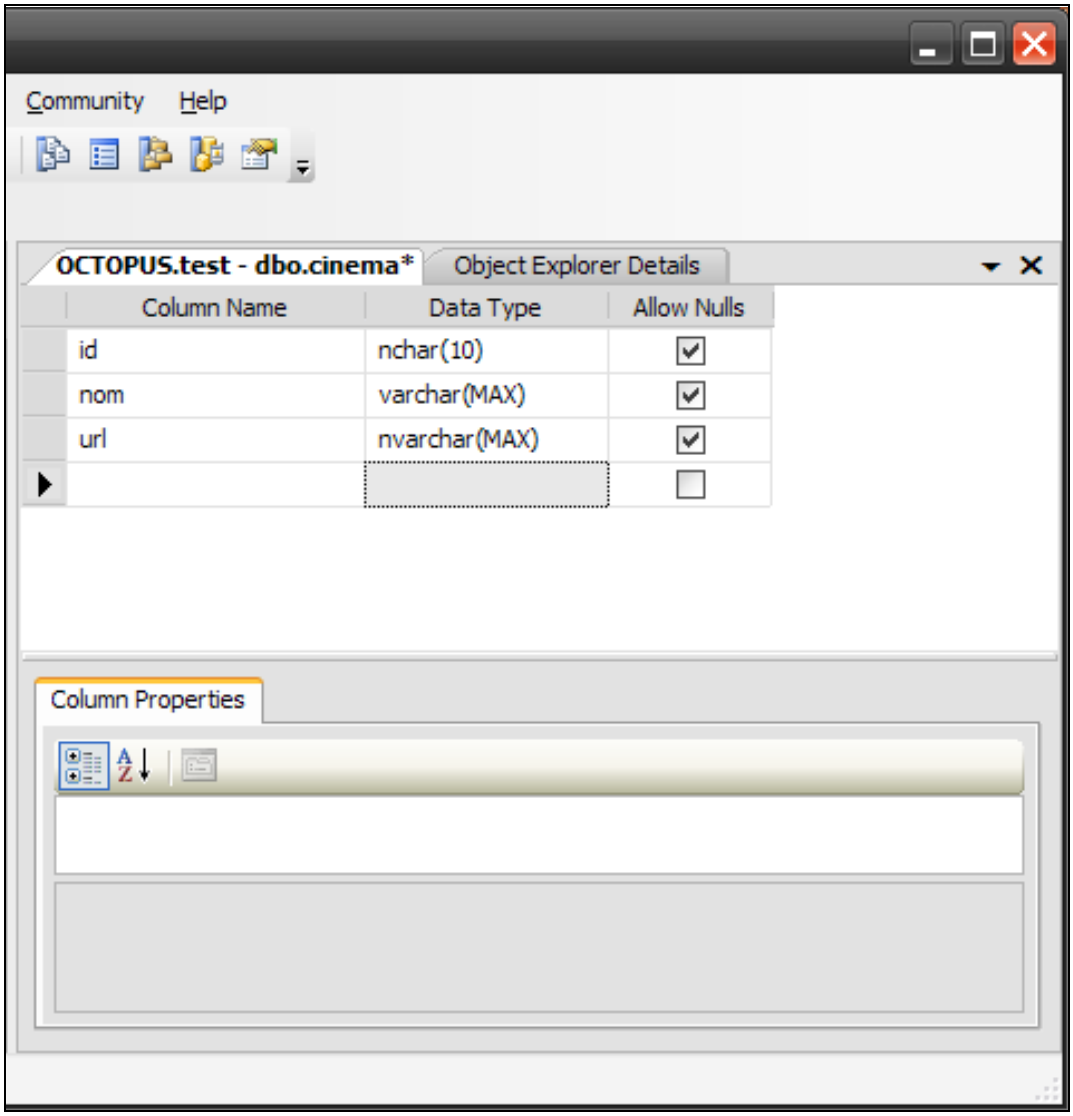
On peut néanmoins choisir la forme du point précédant chaque item grâce a la propriété **BulletStyle** :

Circle	Un cercle
Disc	Un disque (par défaut)
LowerAlpha	Une lettre de l'alphabet en minuscule : a, puis b, c, d …
LowerRoman	Un chiffre romain en minuscule : i, ii, iii, iv etc…
Numbered	Un nombre : 1, 2, 3, 4…
Square	Un carre noir
UpperAlpha	Une lettre de l'alphabet en majuscule : A, B, C, D…
UpperRoman	Un chiffre romain en majuscule : I, II, III, IV
CustomImage	Une image de votre choix dont l'url est spécifiée dans la propriété "BulletImageUrl"

En fait, on peut rendre cette liste interactive grâce a une autre propriété : BulletedListDisplayMode :

HyperLink	Chaque item de la liste va être un lien sur une autre page. Le lien sera la valeur de chaque item
LinkButton	Chaque item va être un bouton de type link
Text	Du texte brut

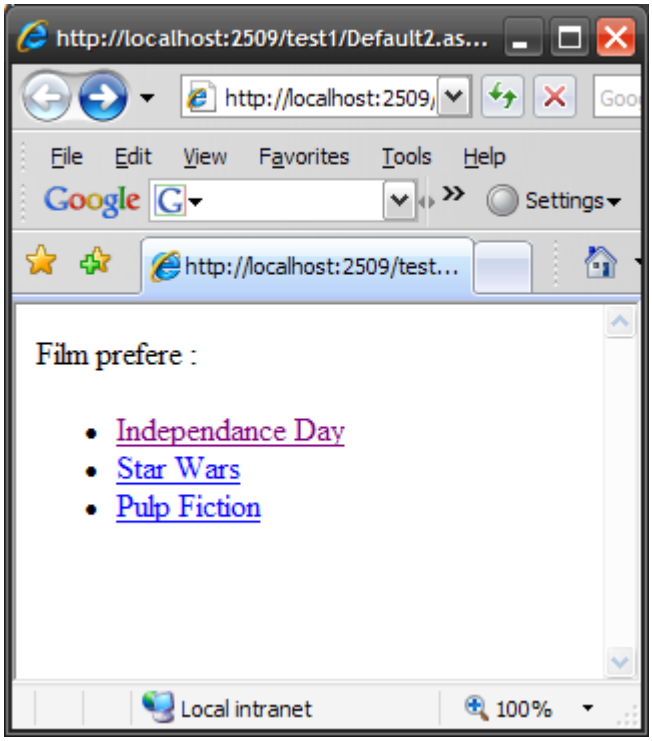
Petit exemple : pour cela ajoutons un champ URL dans notre table :



Un champ URL (qu'on renseigne),

Modifions notre contrôle :

```
<asp:BulletedList id="listeFilms" Runat="server" DataTextField="nom"
    DataValueField="url" SelectionMode="Multiple"  DataSourceId="sourceFilms"
DisplayMode="HyperLink">
</asp:BulletedList>
```



Et si on regarde le source :

```
<ul id="listeFilms" SelectionMode="Multiple">
<li><a href="http://www.imdb.com/title/tt0116629/">Independance Day</a></li>
<li><a href="http://www.imdb.com/title/tt0076759/">Star Wars</a></li>
<li><a href="http://www.imdb.com/title/tt0110912/">Pulp Fiction</a></li>
</ul>
```

La liste a été générée sous forme de lien à partir de la valeur de chaque item.

10.4 LIER LES DONNEES AVEC DES TABLEAUX

Juste une petite aparté, nous avons vu pour l'instant que les DataSource des contrôles étaient lies a des DataSources allant chercher leurs données dans une base de données. Vous n'êtes pas obligés d'utiliser une base de données, vous pourriez très bien a la place utiliser un tableau ou une liste en mémoire.

Petit exemple, supposons que nous ayons une classe "Personnes" :

```
public class Personne
    {
        String _nom;
        String _prenom;
        int _age;

        public Personne(String nom, String prenom, int age)
        {
            _nom = nom;
            _prenom = prenom;
            _age = age;
        }

        public String nom
        {
            get { return _nom; }
        }

        public String prenom
        {
            get { return _prenom; }
        }

        public int age
        {
            get { return _age; }
        }
    }
```

Et un contrôle BulletedList dans notre page :

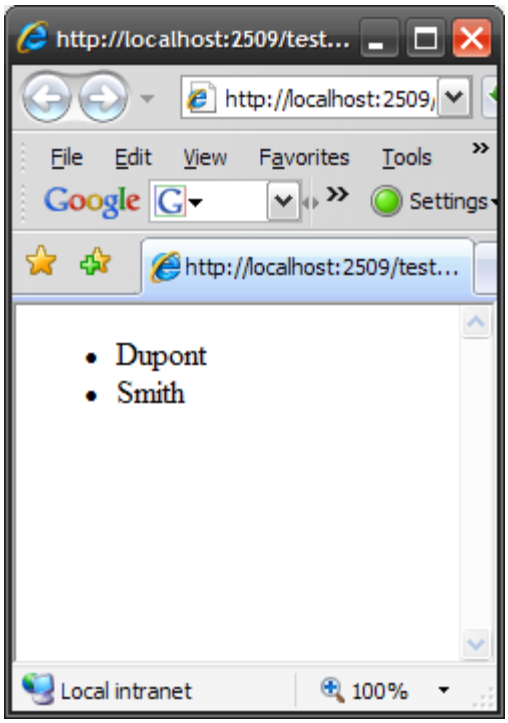
```
<asp:BulletedList ID="liste" runat="server" DataTextField="nom">
</asp:BulletedList>
```

Il suffira de faire lors du page load :

```
Personne p1 = new Personne("Dupont", "Jean", 25);
Personne p2 = new Personne("Smith", "John", 42);
List<Personne> personnes = new List<Personne>();
personnes.Add(p1);
personnes.Add(p2);
liste.DataSource = personnes;
liste.DataBind();
```

Notez bien qu'on bind la data source sur une List en mémoire cette fois.

Et on obtiendra :



Notez que le champ de la classe indique dans DataTextField (idem pour DataValueField) doit être une propriété de la classe, c'est-à-dire que son contenu doit pouvoir être accessible via un getter. Voila qui étend de façon intéressante les possibilités d'utilisation de ces contrôles.

J'espère que cette petite mise en bouche sur les listes vous a plu, nous allons attaquer maintenant les contrôles avec template.

10.5 Les contrôles Repeater et DataList

Un peu plus évolués que les listes vues précédemment, les contrôles **Repeater** et **DataList** permettent également d'afficher le résultat d'une requête sur une base de données mais de façon plus souple que les listes. En effet, ces deux contrôles utilisent ce qu'on appelle des templates (ou modèles). Les templates permettent de définir un modèle décrivant comment chaque ligne renvoyée par la requête va être affichée, le contrôle appliquant ensuite ce modèle sur chaque ligne renvoyée.

Les deux contrôles font en gros la même chose, la différence réside dans le fait que le Repeater affiche des données de façon complètement libres alors que la DataList les affiche dans une table HTML.

10.6 Le Repeater

Le repeater est le contrôle le plus souple pour afficher le résultat d'une requête d'une base de données. Il est en fait souple car il vous laisse entièrement déterminer comment les données vont être affichées.
Pour illustrer son fonctionnement, nous allons ajouter un nouveau champ a notre base, un champ texte nomme "realisateur" :

OCTOPUS.test - dbo.cinema | Object Explorer Details

id	nom	url	realisateur
film1	Independance Day	http://www.imdb.com/title/tt0116629/	Roland Emmerich
film2	Star Wars	http://www.imdb.com/title/tt0076759/	Georges Lucas
film3	Pulp Fiction	http://www.imdb.com/title/tt0110912/	Quentin Tarentino
NULL	NULL	NULL	NULL

4 of 4

Notre repeater va nous permettre d'afficher cette page :

Cette page est composée d'un titre, d'une liste d'enregistrements affichés selon un modèle et un pied de page.

Regardons le source de cette page :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head>
<script runat="server"></script>

</head>
<body>
<form id="form2" runat="server">
<asp:Repeater ID="RepeaterFilms" runat="server" DataSourceID="sourceFilms">
<HeaderTemplate><h1>LES FILMS</h1></HeaderTemplate>
<ItemTemplate>
    <h2><%#Eval("nom")%></h2>
    <h3>Realisateur : <%#Eval("realisateur")%></h3>
    <asp:HyperLink ID="HyperLink1" runat="server"
     NavigateUrl='<%#Eval("url")%>'>Informations sur le film
    </asp:HyperLink>
</ItemTemplate>
<SeparatorTemplate>
    <hr />
</SeparatorTemplate>
<FooterTemplate>
<hr />
Nombre de films : <%# RepeaterFilms.Items.Count%>
</FooterTemplate>
</asp:Repeater>

<asp:SqlDataSource
id="sourceFilms"
SelectCommand="SELECT url, nom, realisateur FROM cinema"
ConnectionString="Data Source=octopus;Initial Catalog=test;User
Id=sa;Password=password;"
Runat="server" />
<br /><br />
<asp:Literal ID="Literal1" runat="server"></asp:Literal>
</form>
```

```
</body>
</html>
```

Le contrôle Repeater est défini ainsi :

```
<asp:Repeater ID="RepeaterFilms" runat="server" DataSourceID="sourceFilms">
```

Le repeater est bindé sur la data source "sourceFilms". Comme avec nos exemple sur les contrôles **List**, le contrôle Repeater peut également, de la même façon, être bindé par programme et non uniquement par déclaration.

Le Repeater est constitue de 4 templates. Ces templates vont définir la façon dont l'entête des données, les données elles-mêmes, la séparation entre chaque donnée et le pied vont être affichées.

Le template <**HeaderTemplate**> défini comment le titre va être affiche. Vous pouvez placer dans ce template (comme dans tous les templates) tout ce que vous voulez : texte brut, HTML, contrôles ASP.NET. Ici c'est du HTML standard (<H1>) et du texte brut.

```
<HeaderTemplate><h1>LES FILMS</h1></HeaderTemplate>
```

Le template **ItemTemplate** est le plus important. Il défini comment chaque ligne renvoyée par la requête SQL va être affichée. C'est un template, donc comme précédemment, on peut y placer n'importe quoi.

```
<ItemTemplate>
    <h2><%#Eval("nom")%></h2>
    <h3>Realisateur : <%#Eval("realisateur")%></h3>
    <asp:HyperLink ID="HyperLink1" runat="server"
NavigateUrl='<%#Eval("url")%>'>Informations sur le fim</asp:HyperLink>
</ItemTemplate>
```

Qu'y trouvons-nous ?

```
<h2><%#Eval("nom")%></h2>
```

Cette ligne va afficher, pour chaque ligne le nom du film encadré par un tag HTML `<h2>`. Les tags spéciaux `<%# et %>` (hérite du vieil ASP) permettent d'appeler dans une page ASP.NET du code C#. La fonction **Eval** appelée ici permet de récupérer le contenu du champ passe en paramètre. Ici, on récupérera pour chaque ligne, le contenu du champ "nom".

```
<h3>Realisateur : <%#Eval("realisateur")%></h3>
```

On fait la même chose avec le champ "realisateur".

```
<asp:HyperLink ID="HyperLink1" runat="server"
NavigateUrl='<%#Eval("url")%>'>Informations sur le fim</asp:HyperLink>
```

Et enfin, on crée un lien. Un template pouvant contenir un contrôle ASP.NET, on crée ici un contrôle HyperLink dont l'url est le contenu du champ "url" de l'enregistrement affiche.

Le template suivant est le **SeparatorTemplate**. Il définit comment chaque enregistrement va être séparé. Vous pouvez placer ce que vous voulez ici, une image, un texte. J'ai pense qu'un tag HTML <HR> serait approprie.

```
<SeparatorTemplate>
    <hr />
</SeparatorTemplate>
```

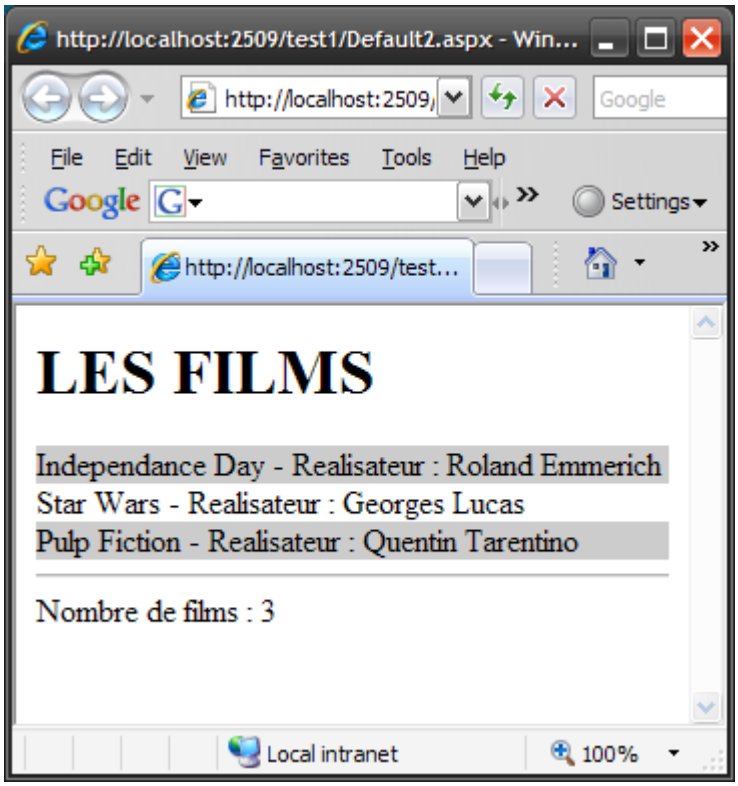
Et enfin le template <FooterTemplate> qui défini comment le pied va être affiche. Ici on affiche le nombre de films. On appelle donc encore du code C# avec <%# %> et dans ce code que fait-on ?

```
<FooterTemplate>
<hr />
Nombre de films : <%# RepeaterFilms.Items.Count%>
</FooterTemplate>
</asp:Repeater>
```

RepeaterFilms.Items.Count

RepeaterFilms est le nom de notre repeater. Il possède beaucoup de propriétés. Entre autre la propriété Items (un tableau contenant chaque ligne du repeater), ce tableau ayant lui-même une propriété count indiquant sa taille.

Un petit truc avant d'en finir avec le Repeater :



Que fait ce repeater ? il affiche un listing avec la liste des films. Un film sur deux a une couleur de fond différente.

```
<ItemTemplate>
    <div style="background-color:#CCC">
    <%#Eval("nom")%> - Realisateur : <%#Eval("realisateur")%>
    </div>
</ItemTemplate>
```

J'ai modifié notre ItemTemplate, affiche les données dans un div de couleur grise.

Et J'ai utilise un autre template dont je n'avais pas encore parle :

```
<AlternatingItemTemplate>
    <div>
    <%#Eval("nom")%> - Realisateur : <%#Eval("realisateur")%>
    </div>
</AlternatingItemTemplate>
```

Le template <**AlternativeItemTemplate**> contient les mêmes informations que le template <**ItemTemplate**> a ce détail près que le div n'a pas de couleur de fond.

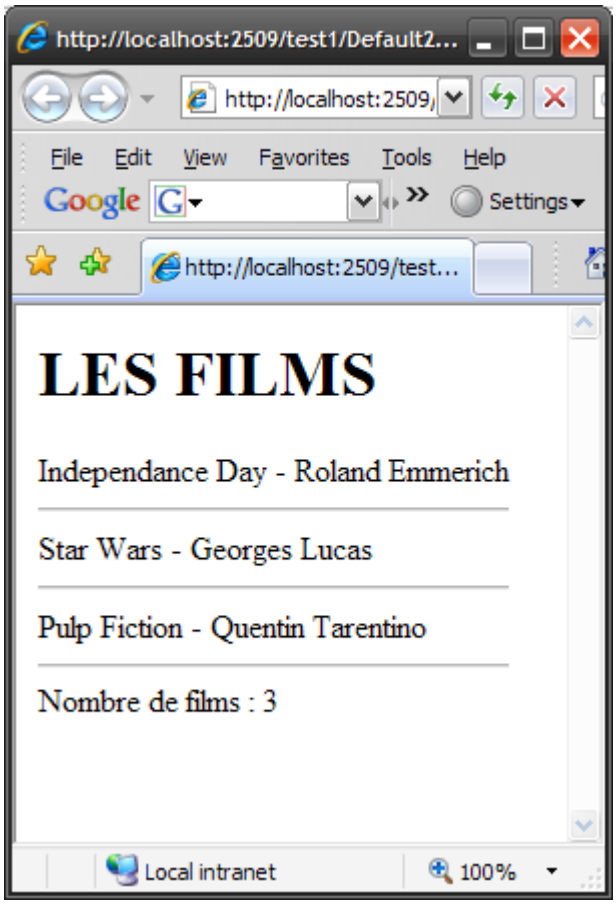
Lorsque les enregistrements vont être affichés, le Repeater va utiliser le format de **l'ItemTemplate** pour le 1er, puis celui de **l'AlternateItemTemplate** pour le second, puis celui de **l'ItemTemplate** pour le 3eme et ainsi de suite. Il alterne entre les deux template. Ca peut être très pratique pour afficher un long listing en alternant les couleurs de fond sur chaque ligne pour améliorer sa lisibilité.

Tous ces templates ne sont pas obligatoires (si vous ne voulez pas de pied ou d'entête, n'utilisez pas de template <HeaderTemplate> ou <FooterTemplate>). Aucun n'est obligatoire, y compris ItemTemplate, mais bon, la, ca n'a plus trop d'utilité…

10.7 Le contrôle DataList

Le contrôle **DataList**, tout comme le contrôle Repeater utilise des templates, et ça tombe bien, ce sont les mêmes que le Repeater. Mais le DataList est plus intéressant car il permet, outre l'affichage, de modifier, supprimer ou ajouter des données. Voila qui est intéressant !

Modifions donc notre liste en remplaçant juste le Repeater par un DataList :



Jetons un œil au source :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head>
<script runat="server">
</script>

</head>
<body>
<form id="form2" runat="server">
<asp:DataList ID="liste" runat="server" DataSourceID="sourceFilms" >
<HeaderTemplate><h1>LES FILMS</h1></HeaderTemplate>
<ItemTemplate>
    <div>
    <%#Eval("nom")%> - <%#Eval("realisateur")%>
    </div>
</ItemTemplate>
<SeparatorTemplate>
<hr />
</SeparatorTemplate>
<FooterTemplate>
<hr />
Nombre de films : <%# liste.Items.Count%>
</FooterTemplate>
</asp:DataList>

<asp:SqlDataSource
id="sourceFilms"
SelectCommand="SELECT url, nom, realisateur FROM cinema"
ConnectionString="Data Source=octopus;Initial Catalog=test;User
Id=sa;Password=password;"
Runat="server" />
<br /><br />
<asp:Literal ID="Literal1" runat="server"></asp:Literal>
</form>
</body>
</html>
```

C'est la même chose qu'un repeater, on a juste change le nom du contrôle.

La différence, c'est que le repeater se contente d'utiliser les templates que vous avez renseigné sans rien ajouter. La DataList crée une grande table et insère chaque enregistrement dans une ligne de la table. Si vous voulez éviter l'affichage dans une table, vous pouvez utiliser la propriété "RepeatLayout" du contrôle. Cette propriété peut avoir l'une des deux valeurs suivantes :

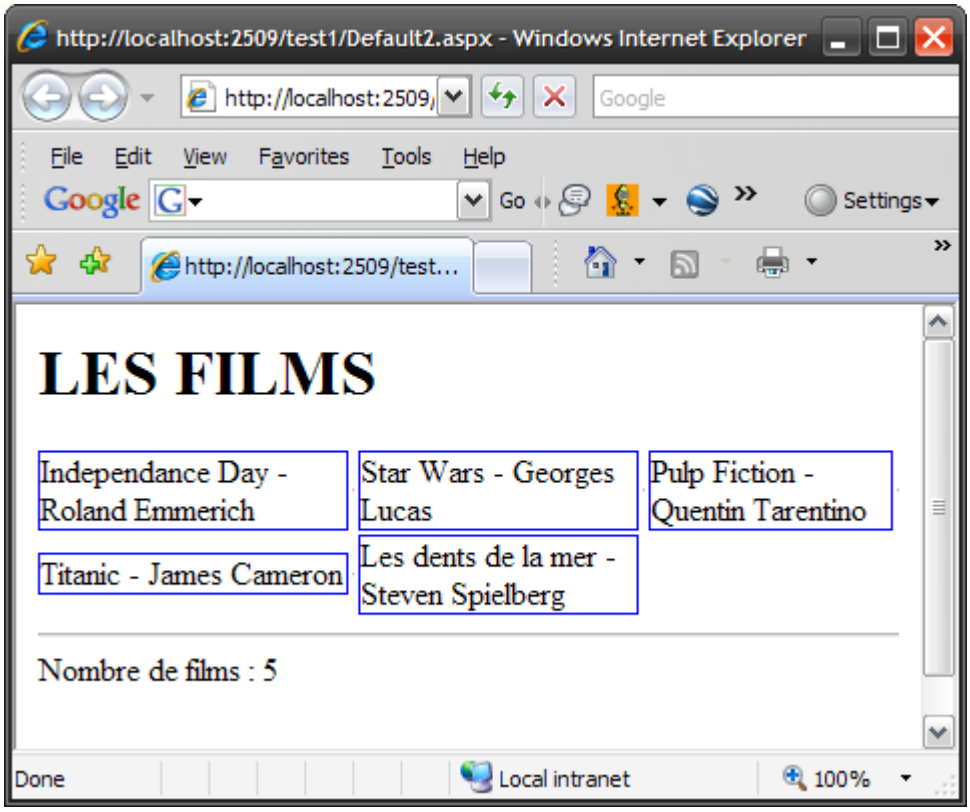
Table	Les enregistrements sont affichés dans une table (par défaut)
Flow	Les enregistrement sont affiches à l'intérieur d'un tag `<span>`

Vous pouvez également afficher les données en colonne au lieu de les afficher par défaut en ligne, pour cela, modifiez les propriétés RepeatColumns et RepeatDirections

RepeatColumn	Nombre de colonnes à afficher
RepeatDirection	Sens de l'affichage (Vertical ou Horizontal)

Modifions le DataList pour afficher sur 3 colonnes :

```
<asp:DataList ID="liste" runat="server" DataSourceID="sourceFilms" RepeatColumns="3"
RepeatDirection="Horizontal" >
<HeaderTemplate><h1>LES FILMS</h1></HeaderTemplate>
<ItemTemplate>
    <div style="border:solid 1px blue">
    <%#Eval("nom")%> - <%#Eval("realisateur")%>
    </div>
</ItemTemplate>
<SeparatorTemplate>
<hr />
</SeparatorTemplate>
<FooterTemplate>
<hr />
Nombre de films : <%# liste.Items.Count%>
</FooterTemplate>
</asp:DataList>
```

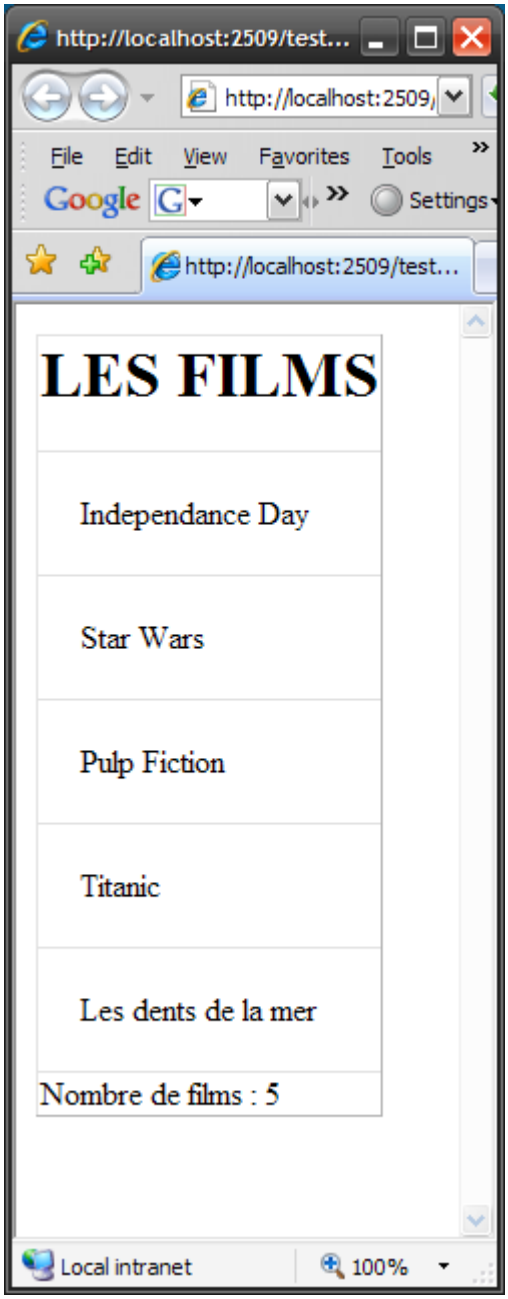


Les enregistrements sont affichés sur 3 colonnes.
Le template SeparatorTemplate ne peut plus être utilisé.

Jusque là, rien d'extraordinaire. On pourrait, avec un peu de réflexion a arriver à la même chose avec un repeater. Nous allons voir maintenant ou le DataList est supérieur au Repeater. Le DataList fonctionne en fait comme une liste normale, a savoir qu'on peut sélectionner un enregistrement, le modifier, le supprimer, etc…

10.7.1 Sélectionner un enregistrement

Commençons donc par sélectionner un enregistrement, nous allons repasser en ligne plutôt qu'en colonnes.



```
<asp:DataList ID="liste" runat="server" DataSourceID="sourceFilms" GridLines="Both"  >
<HeaderTemplate><h1>LES FILMS</h1></HeaderTemplate>
<ItemTemplate>
    <div style="padding: 20px">
    <%#Eval("nom")%>
    </div>
</ItemTemplate>
<FooterTemplate>
Nombre de films : <%# liste.Items.Count%>
</FooterTemplate>
</asp:DataList>
```

Notez au passage la nouvelle propriété **GridLine** qui peut prendre une de ces valeurs :

Vertical	N'affiche que les lignes verticales
Horizontal	N'affiche que les lignes horizontales
Both	Affiche les lignes horizontales et verticales
None	N'en n'affiche pas

Nous allons donc modifier un peu notre DataList pour que, lorsqu'on sélectionnera une ligne de la liste, un message s'affiche en bas de la liste indiquant l'identifiant du film sélectionné. A partir de cet identifiant, libre à nous d'interroger la base pour récupérer, par exemple, des informations supplémentaires.

J'ai cliqué sur "Pulp Fiction", le message indiquant l'id de ce film (film3) s'affiche à l'écran.

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head>
<script runat="server">

protected void liste_SelectedIndexChanged(object sender, EventArgs e)
{
   lbl.Text = "l'id du film selectionne est : " + liste.SelectedValue;
}
</script>

</head>
<body>
<form id="form2" runat="server">
<asp:DataList ID="liste" runat="server" DataSourceID="sourceFilms"
    GridLines="Both" DataKeyField="id"
    onselectedindexchanged="liste_SelectedIndexChanged" >
    <HeaderTemplate><h1>LES FILMS</h1></HeaderTemplate>
    <ItemTemplate>
        <div style="padding: 20px">
        <asp:LinkButton ID="Link"  CommandName="select" runat="server">
         <%#Eval("nom")%></asp:LinkButton>
        </div>
   </ItemTemplate>
   <FooterTemplate>
       Nombre de films : <%# liste.Items.Count%>
   </FooterTemplate>
</asp:DataList>
<br />
<asp:Label ID="lbl" runat="server" ></asp:Label>

<asp:SqlDataSource
   id="sourceFilms"
   SelectCommand="SELECT id, url, nom, realisateur FROM cinema"
   ConnectionString="Data Source=octopus;Initial Catalog=test;
   UserId=sa; Password=password;" Runat="server" />
```

```
   <br /><br />
<asp:Literal ID="Literal1" runat="server"></asp:Literal>
</form>
</body>
</html>
```

Qu'avons-nous modifié ?

```
<asp:DataList ID="liste" runat="server" DataSourceID="sourceFilms"
    GridLines="Both" DataKeyField="id"
    onselectedindexchanged="liste_SelectedIndexChanged" >
```

Deux choses : la propriété **DataKeyField** est apparue. Elle va déterminer quel est le champ de la table qui va être utilise comme valeur (comme clef) pour la ligne.
La propriété **onselecteindexchanged** a été initialisée, cette propriété indique le nom d'une fonction qui va être appelée lorsqu'on sélectionne une ligne de la DataTable.

Comment selectionne-t-on une ligne ?

```
<asp:LinkButton ID="Link" CommandName="select" runat="server">
<%#Eval("nom")%></asp:LinkButton>
```

Grâce à ce bouton. C'est un LinkButton qui va donc faire un postback. On donne comme nom de commande à ce bouton "Select" et ça suffit pour en faire un bouton qui va sélectionner la ligne dans laquelle il se trouve.

On ajoute ensuite une ligne de code sur le serveur :

```
protected void liste_SelectedIndexChanged(object sender, EventArgs e)
{
   lbl.Text = "l'id du film selectionne est : " + liste.SelectedValue;
}
```

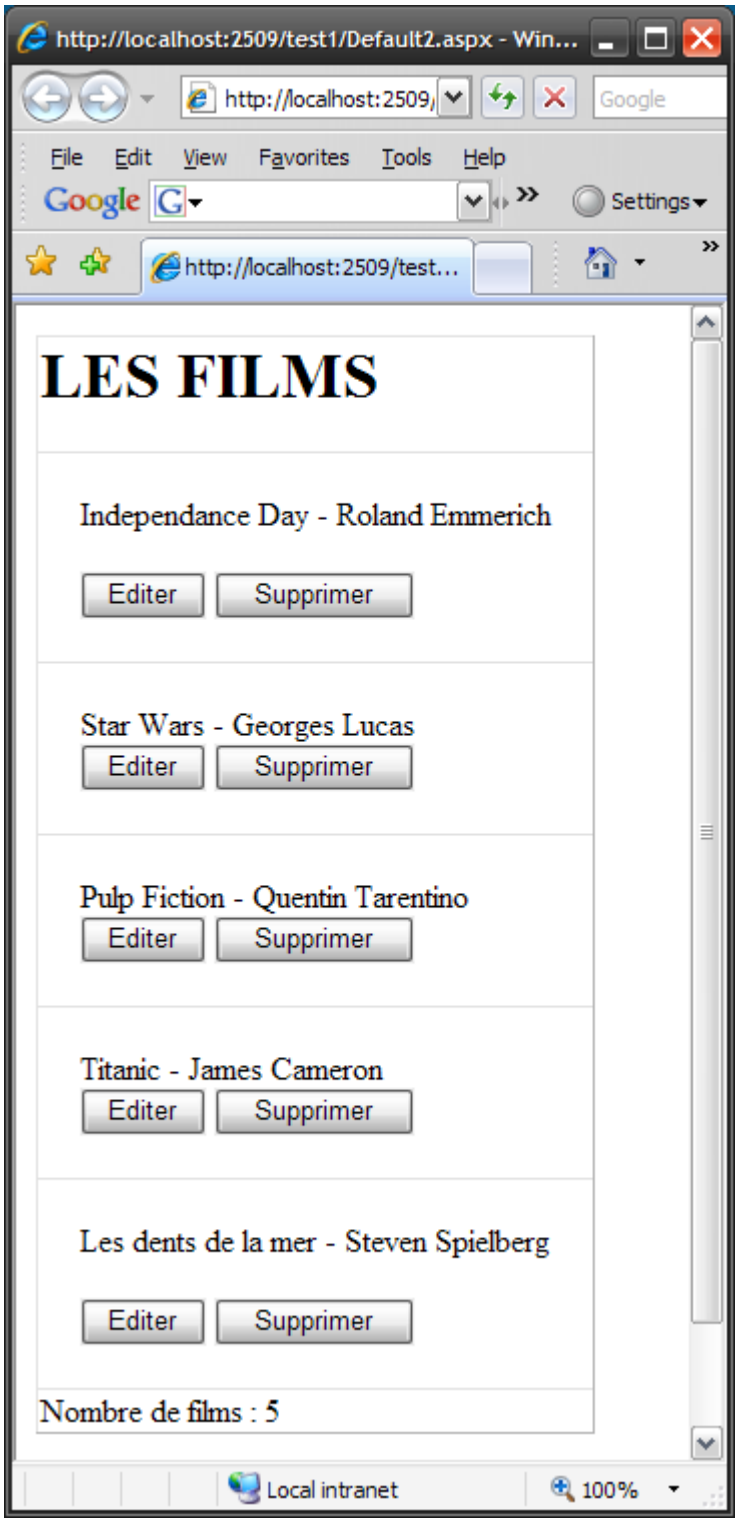
Cette ligne affiche dans le label la valeur de la ligne sélectionnée de la liste. C'est tout ! épatant non ?

Nous allons maintenant faire mieux, nous allons éditer le contenu de la liste !

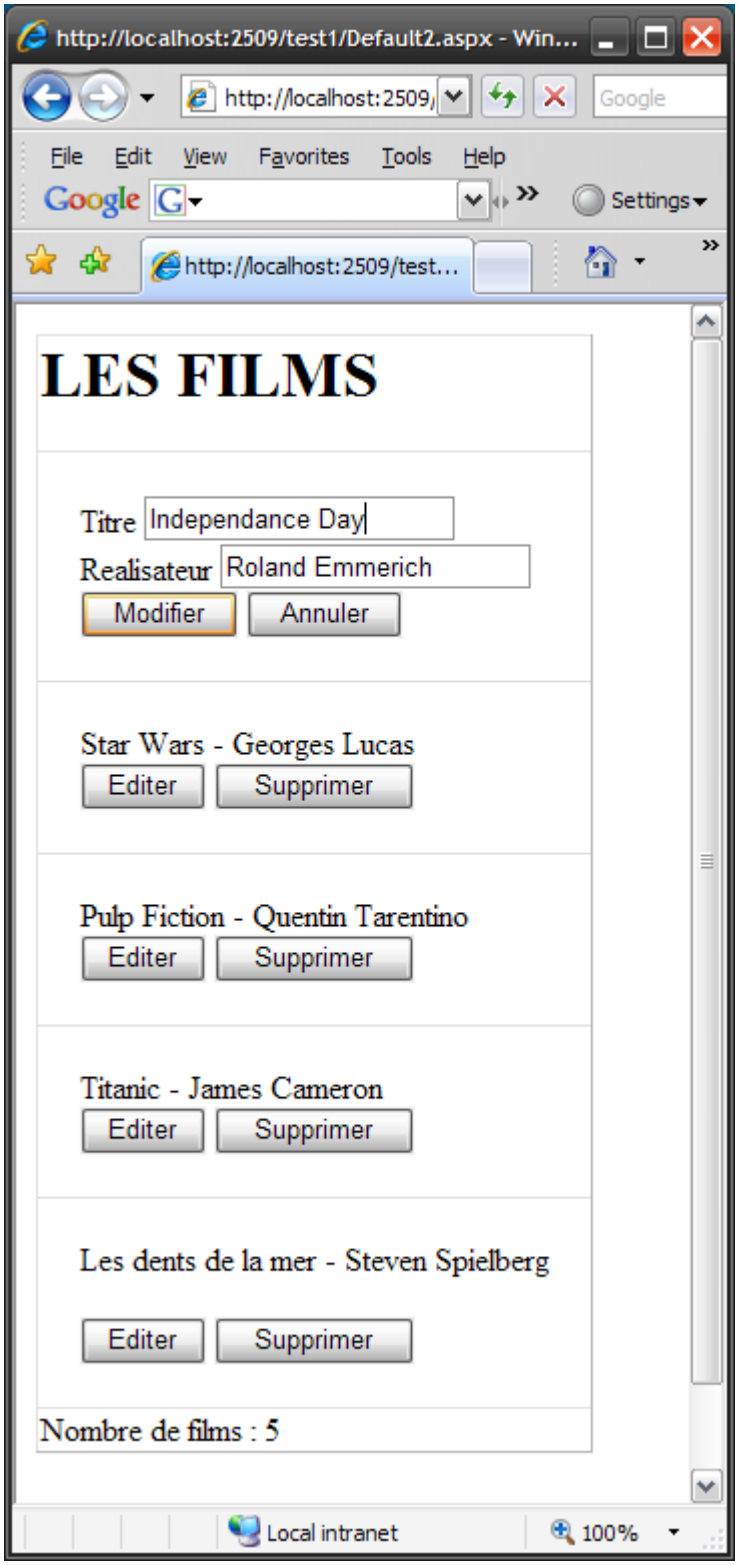
10.7.2 Editer le contenu de la DataList

Comme vous le verrez plus tard en découvrant les contrôles GridView, FormView et DetailsView, le contrôle DataList n'est pas le plus pratique pour éditer des contenus, car il oblige à taper du code C#, allons-y tout de même !

Nous allons afficher la liste et placer à côté de chaque ligne un bouton "Editer" et un bouton "Supprimer". Lorsqu'on cliquera sur "editer", un zone d'édition s'affichera sous la ligne pour nous permettre d'en modifier le contenu, et en cliquant sur "supprimer" … nous supprimerons l'enregistrement.

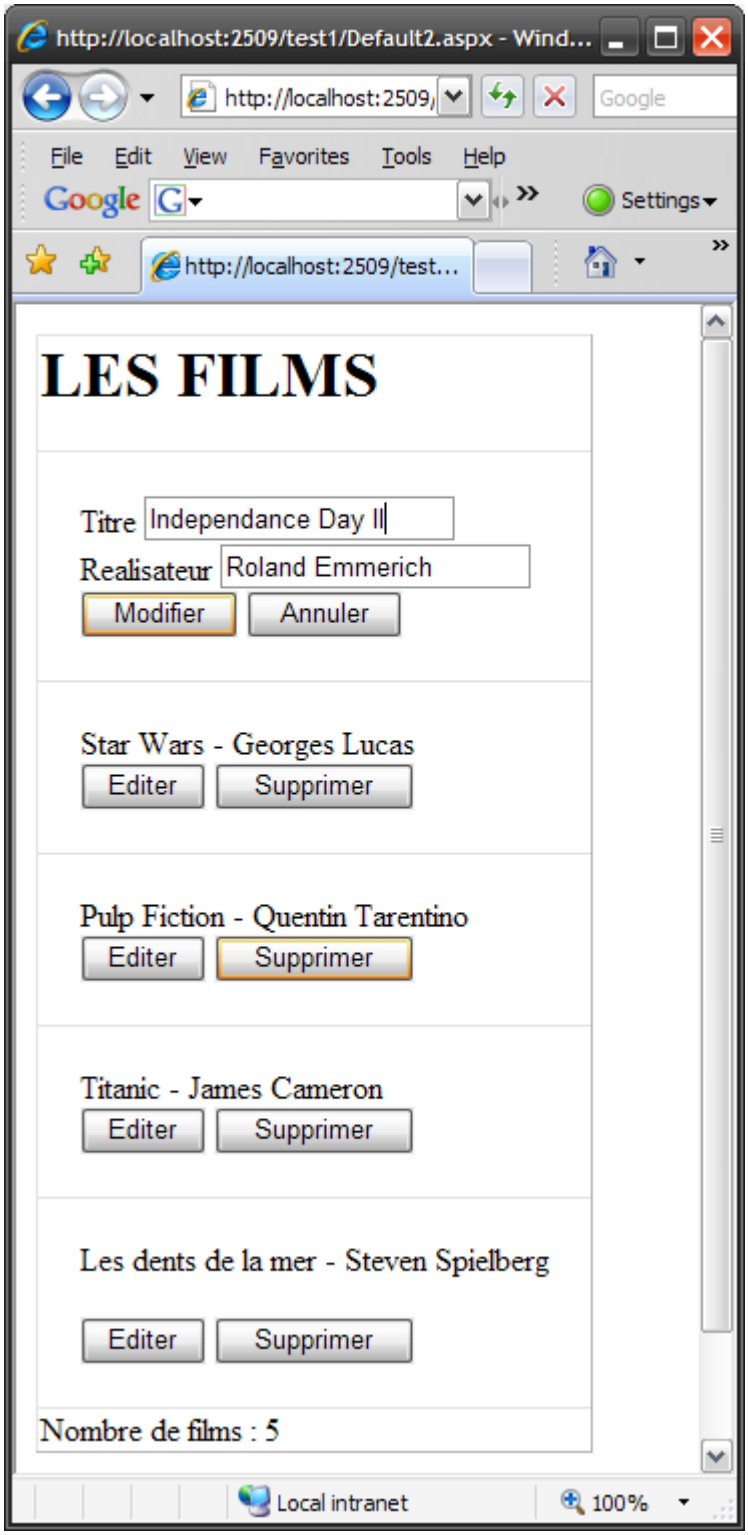


On clique sur "Editer" Independance Day :



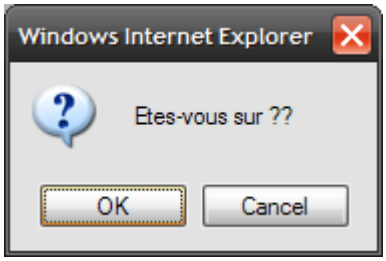
La ligne est remplacée par deux champs éditables.

Sous ces champs, on trouve des boutons "Modifier" pour accepter la modification ou "annuler" pour sortir de l'édition sans modification.



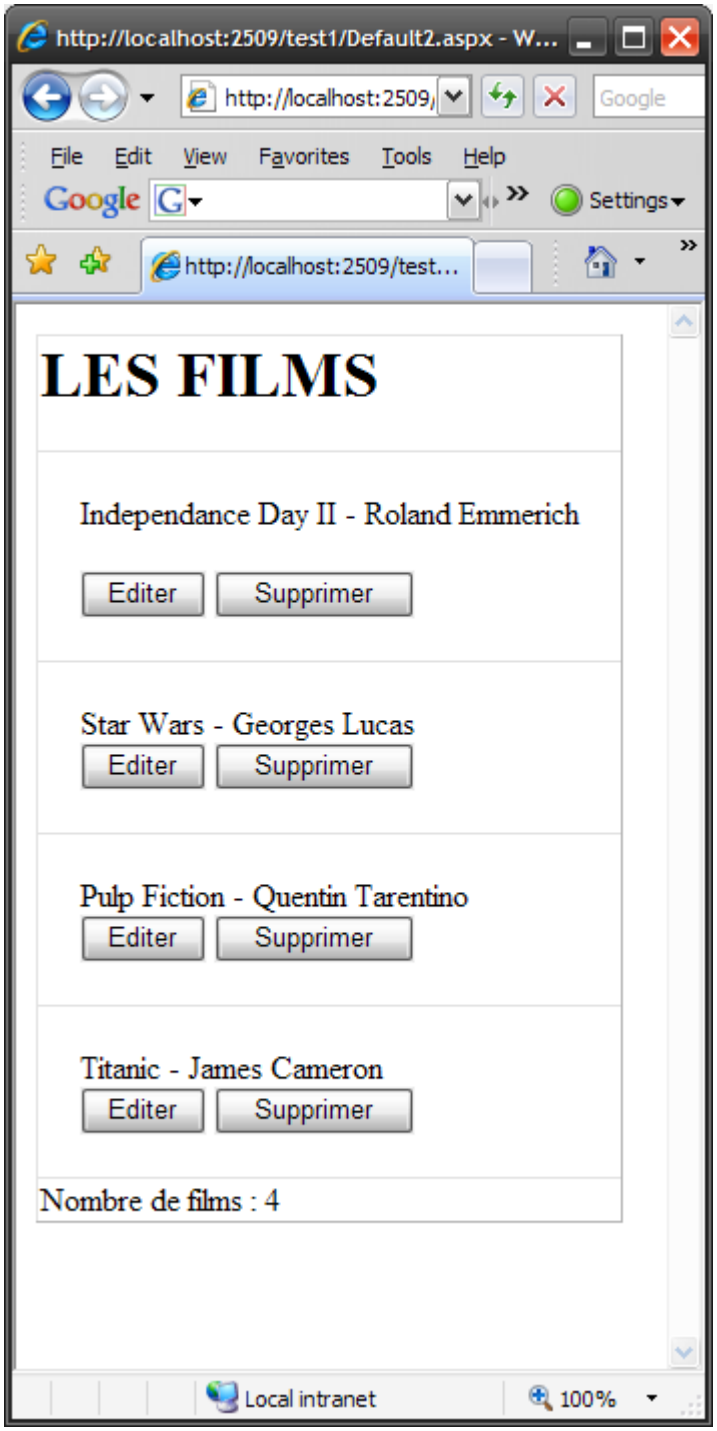
On modifie, on clique sur "Modifier" et c'est sauve dans la base. Si on avait clique sur "Annuler" on serait repasse dans l'affichage par défaut sans que rien ne soit sauve.

Supprimons maintenant "Les dents de la mer" en cliquant sur le bouton "supprimer"



Un petit message nous demande quand même de confirmer.

On confirme, et hop :



Les dents de la mer ont disparu, la liste ne contient plus que 4 films.

Comment avons-nous fait ?

Tout d'abord, notre DataList a un peu change, détaillons-la :

```
<asp:DataList ID="liste" runat="server" DataSourceID="sourceFilms"
    GridLines="Both" DataKeyField="id"
    oneditcommand="liste_EditCommand"
    onupdatecommand="liste_UpdateCommand"
    ondeletecommand="liste_DeleteCommand"
    oncancelcommand="liste_CancelCommand" >
```

La définition de la DataList elle-même inclut 4 nouvelles propriétés :

Oneditcommand	La fonction qui sera appelée quand on cliquera sur le bouton "Editer"
Onupdatecommand	La fonction qui sera appelée quand on cliquera sur le bouton "Modifier"
Ondeletecommand	La fonction qui sera appelée quand on cliquera sur le bouton "Supprimer"
oncancelcommand	La fonction qui sera appelée quand on cliquera sur le bouton "Annuler"

Plus une autre, DataKeyField. Cette dernière indique le nom du champ dans la table qui est la clef de la table, c'est un champ dont on doit être certain que le contenu sera unique pour chaque enregistrement. Nous en aurons besoin pour faire les suppressions et les mises a jour.

Le template du header ne change pas :

```
<HeaderTemplate><h1>LES FILMS</h1></HeaderTemplate>
```

Le template de l'item change, puisqu'a chaque ligne nous avons deux boutons a afficher, le bouton "Editer" et le bouton "Supprimer".

```
<ItemTemplate>
<div style="padding: 20px">
  <%#Eval("nom")%> - <%#Eval("realisateur")%> <br />
  <asp:Button ID="bt_edit" runat="server" Text=" Editer " CommandName="Edit" />
  <asp:Button ID="bt_suppr" runat="server" Text=" Supprimer " CommandName="Delete"
OnClientClick="return confirm('Etes-vous sur ??');" />
</div>
</ItemTemplate>
```

Ce sont deux boutons ASP.NET (ils auraient pu être remplacés par des LinkButton ou des ImageButton). Le bouton appelant l'édition aura TOUJOURS sa propriété CommandName égale a "Edit" et le bouton appelant la suppression aura toujours sa propriété CommandName égale a "Delete".

Vous noterez au passage que le bouton de suppression a une propriété OnClientClick : on définit dans cette propriété une fonction javascript (donc cote client) qui sera appelée quand on cliquera sur ce bouton. Si la fonction renvoie false, le click sur ce bouton est annule et aucun évènement ne se déclenche sur le serveur.

Voici du nouveau ! On connaissait le template <ItemTemplate>, voici le template <EditItemTemplate>. Quand la ligne passe en mode édition (nous allons voir comment ensuite), le template affiche est remplace par le EditItemTemplate.

```
<EditItemTemplate>
<div style="padding: 20px">
  <asp:Label ID="Label1" runat="server" Text="Titre" AssociatedControlID="txt_Titre" />
  <asp:TextBox ID="txt_Titre" runat="server" Text='<%#Eval("nom")%>' />
  <br />
  <asp:Label ID="Label2" runat="server" Text="Realisateur"
  AssociatedControlID="txt_Realisateur" />
  <asp:TextBox ID="txt_Realisateur" runat="server" Text='<%#Eval("realisateur")%>' />
  <br />
  <asp:Button ID="bt_update" runat="server" Text=" Modifier " CommandName="Update" />
  <asp:Button ID="bt_cancel" runat="server" Text=" Annuler " CommandName="Cancel"/>
</div>
</EditItemTemplate>
```

Que trouve-t-on dans ce template ? Le nom et le réalisateur du film qui sont affichés ici dans des **TextBox** au lieu de l'être dans des Label (on va pouvoir les modifier) et deux boutons, un qui va servir à valider la modification et un qui va permettre d'annuler la modification.

Le bouton servant a valider la modification doit obligatoirement avoir sa propriété CommandName définie par "Update"
Le bouton servant a annuler la modification doit obligatoirement avoir sa propriété CommandName définie par "Cancel"

Le Foote ne change pas.

```
<FooterTemplate>
Nombre de films : <%# liste.Items.Count%>
</FooterTemplate>
</asp:DataList>
```

La datasource change aussi !

```
<asp:SqlDataSource
id="sourceFilms"
SelectCommand="SELECT id, url, nom, realisateur FROM cinema"
DeleteCommand="DELETE cinema where id=@id"
UpdateCommand="UPDATE cinema set nom=@nom, realisateur=@realisateur WHERE id=@id"
```

```
ConnectionString="Data Source=octopus;Initial Catalog=test;User
Id=sa;Password=password;"
Runat="server">
```

On trouve en plus de la commande select, deux nouvelles commandes définies par les propriétés DeleteCommand (commande SQL qui va faire la suppression) et UpdateCommand (commande SQL qui va faire la mise a jour).

Contrairement a la commande de sélection, ces deux dernières ont besoin de paramètres, ces paramètres vont être définis ainsi. La gymnastique est un peu tordue suivez le mouvement :

commande de mise a jour : UPDATE cinema set nom=@nom, realisateur=@realisateur where id=@id.

Chaque paramètre est défini par @ suivi d'un nom. On retrouve ces noms dans la section UpdateParameters definie plus bas :

```
<UpdateParameters>
<asp:Parameter Name="id" />
<asp:Parameter Name="nom" />
<asp:Parameter Name="realisateur" />
</UpdateParameters>
```

Ce qui va se passer de façon interne, c'est que nous, nous affecterons un contenu à chacun des paramètres définis dans la section UpdateParameters, et quand le Framework exécutera la commande de mise a jour, il remplacera par exemple @realisateur par le contenu du paramètre dont le nom est "realisateur".

Rien n'empêcherait d'avoir "update cinema set nom=@toto, realisateur=@zorro …." A condition de 1) avoir un <asp:Parameter Name="toto"> et un <asp:Parameter Name="zorro"> dans la section UpdateParameters et 2) d'affecter (nous verrons ca plus bas) le bon contenu a toto et zorro.
C'est pour ca que, pour simplifier, on donne aux paramètres les mêmes noms que les champs.

Enfin, on a une section DeleteParameters pour les paramètres de la commande DELETE (en l'occurrence ici un seul paramètre)

```
<DeleteParameters>
<asp:Parameter Name="id" />
</DeleteParameters>
```

Côté code C#, il y a du nouveau aussi. Nous avons vu plus haut que nous avions 4 fonctions appelées quand on clique sur un des quatre boutons. Il faut donc définir ces fonctions :

```
protected void liste_EditCommand(object source, DataListCommandEventArgs e)
{
    liste.EditItemIndex = e.Item.ItemIndex;
    liste.DataBind();
```

```
}
```

La première est celle appelée quand on clique sur le bouton "Editer", on passe un élément de la liste en mode "Edition". Le fait de passer cet élément en mode édition entraine l'utilisation du template EditItemTemplate à la place du template ItemTemplate. Comment fait-on ? Très simplement :

La DataList a une propriété nommée EditItemIndex qui indique quel est l'item en cours d'édition, on indique ici qu'il s'agit de l'item dans lequel on vient de cliquer (on récupère dans e.Item l'item dans lequel on vient de cliquer, la propriété ItemIndex est son index dans la liste). Le fait d'initialiser la propriété EditItemIndex va passer la ligne en mode édition.

```
    liste.EditItemIndex = e.Item.ItemIndex;
```

Les deux TextBox vont donc être affichés, mais vides, l'appel a DataBind va réinitialiser le contenu de la liste et donc des deux boutons.

La deuxième fonction va être appelée quand on cliquera sur le bouton "Modifier" dans le mode édition. Elle va appeler la commande SQL Update. Mais n'oublions pas que cette commande a besoin de 3 paramètres, l'un pour indiquer le nouveau contenu du champ nom, un autre pour indiquer le nouveau contenu du champ realisateur et un troisième qui va donner la valeur de la clef pour cette enregistrement (il faut bien savoir quel va être l'enregistrement qui va être modifié).

```
protected void liste_UpdateCommand(object source, DataListCommandEventArgs e)
    {
    TextBox txt_Titre = (TextBox)e.Item.FindControl("txt_Titre");
    TextBox txt_Realisateur = (TextBox)e.Item.FindControl("txt_Realisateur");
```

Ces deux lignes récupèrent les contrôles dans lesquels l'utilisateur a saisi les nouvelles valeurs : ce sont des TextBox. e.Item est l'item dans la liste, la fonction FindControl va aller chercher dans cet item un contrôle ASP.NET dont le nom est "txt_Titre" pour le nom du film et "txt_Realisateur" pour le nom du realisateur. Je reconnais que sur ce coup là, le framework aurait pu proposer une solution plus élégante mais bon, faisons avec.

Ici on va initialiser le contenu de nos paramêtres :

```
sourceFilms.UpdateParameters["id"].DefaultValue =
liste.DataKeys[e.Item.ItemIndex].ToString();
sourceFilms.UpdateParameters["nom"].DefaultValue = txt_Titre.Text;
sourceFilms.UpdateParameters["realisateur"].DefaultValue = txt_Realisateur.Text;
sourceFilms.Update();
```

```
sourceFilms.UpdateParameters["nom"].DefaultValue = txt_Titre.Text;
```

par exemple, la valeur (propriété DefaultValue) du paramètre "nom" est initialisée avec la propriété Text du TextBox.

```
sourceFilms.UpdateParameters["id"].DefaultValue =
liste.DataKeys[e.Item.ItemIndex].ToString();
```

La ligne la est plus tordue. Le paramètre id contient la valeur de la clef pour cet enregistrement (on fera un update …. Where id = cette valeur) . Ou aller la trouver ? Souvenez-vous de la propriété DataKeyField de la liste. Le fait d'initialiser cette propriété va lors du databinding, renseigner un tableau de la DataList nomme "DataKeys". Ce tableau va contenir, pour chaque item, la valeur de sa clef. On récupère donc la clef de l'item en cours de modification en faisant

```
liste.DataKeys[e.Item.ItemIndex].ToString();
```

La dernière ligne (sourceFilms.Update()) appelle la commande Update avec les paramètres correctement renseignes.

La fonction appelée pour supprimer un item est très simple, elle initialise le paramètre id de la commande Delete avec la clef de l'item, puis appelle la fonction Delete.

```
protected void liste_DeleteCommand(object source, DataListCommandEventArgs e)
    {
    sourceFilms.DeleteParameters["id"].DefaultValue =
liste.DataKeys[e.Item.ItemIndex].ToString();
    sourceFilms.Delete();
    }
```

Enfin, cette fonction (appelée quand l'utilisateur clique sur Annuler pendant l'édition) quitte le mode édition (en mettant -1 comme n. item en cours d'édition, soit aucun) et rebind la DataList.

```
protected void liste_CancelCommand(object source, DataListCommandEventArgs e)
    {
    liste.EditItemIndex = -1;
    liste.DataBind();
    }
```

Vous avez vu qu'on peut facilement editer une liste avec ASP.NET. On peut le faire plus simplement encore avec d'autres contrôles plus évolués que nous allons étudier.

10.8 Le contrôle GridView

Le contrôle GridView est le contrôle le plus puissant et le plus complexe du Framework ASP.NET. Il permet d'afficher, de sélectionner, de trier, de paginer et d'editer des données (provenant d'une base de données par exemple).

Notez que ce contrôle n'existait pas dans les versions 1.x, on utilisait a la place un contrôle nomme DataGrid (qui est une version simplifiée du GridView). Le DataGrid est toujours présent dans les derniers Framework pour des raisons de compatibilité, mais il ne devrait pas en principe être utilisé. Nous ne l'étudierons donc pas.

Pour nos exemples, nous allons utiliser des données provenant d'une base de données. Ce sera plus simple. Histoire de travailler sur une même base, nous allons utiliser la base nommée "AdventureWorks", c'est une base d'exemple pour SQL Server disponible sur CodePlex (CodePlex est un site de Microsoft sur lequel des projets libres sont disponibles, une sorte de SourceForge dédiée aux technologies Microsoft).

Téléchargez donc cette base sur :
http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=MSFTDBProdSamples&DownloadId=11757

Installez la et attachez la a votre serveur SQL server 2005.

Ok, commençons par afficher bêtement le contenu de la table "customer"

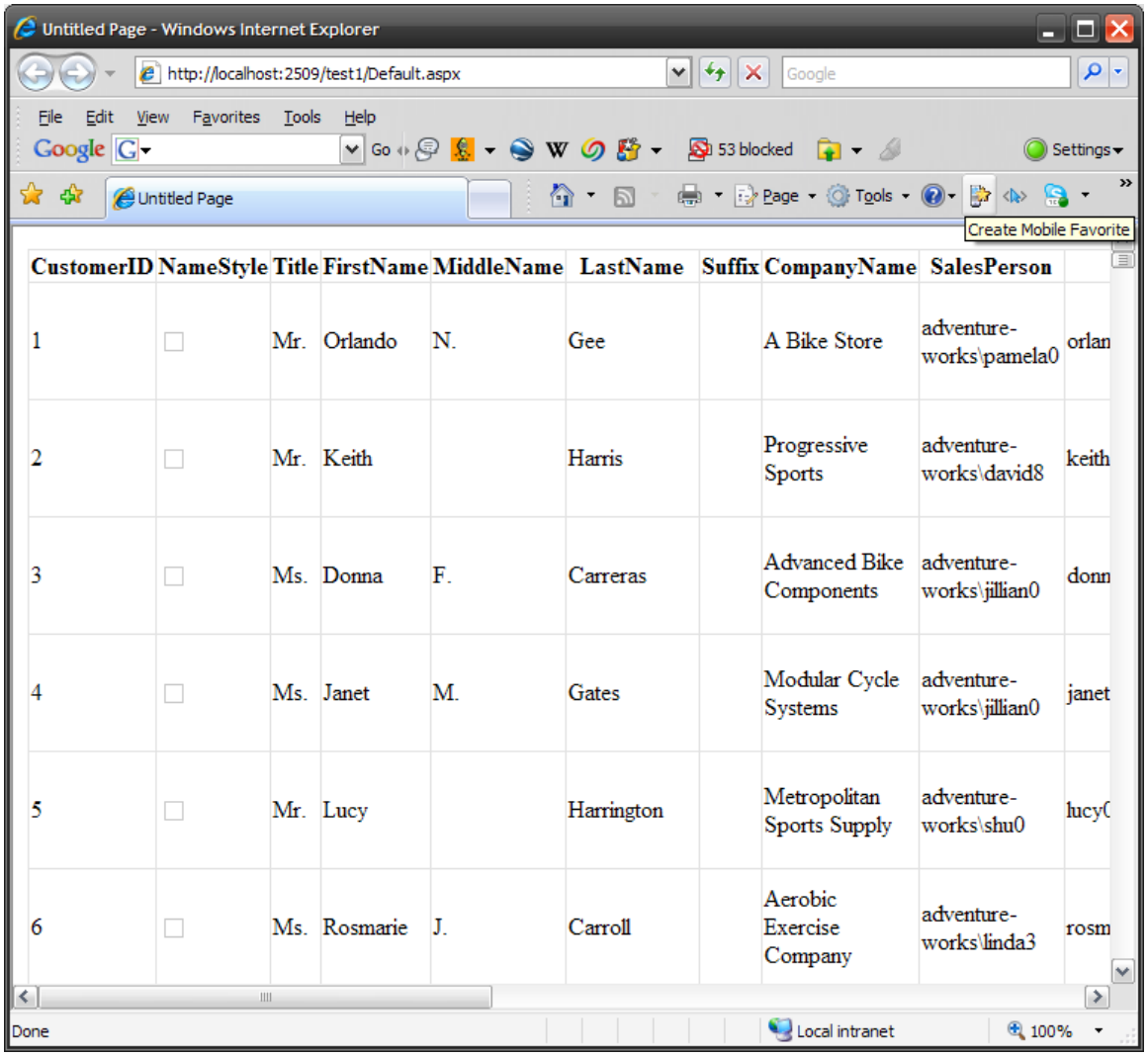
10.8.1 Afficher les données

Le code est réduit a son strict minimum :

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <asp:GridView ID="gridview" runat="server"
     DataSourceID="datasource">
    </asp:GridView>

    <asp:SqlDataSource id="datasource"
    SelectCommand="SELECT * FROM SalesLT.Customer"
    ConnectionString="Data Source=octopus;
    Initial Catalog=AdventureWorksLT; User Id=sa; Password=password;"
    Runat="server">
    </asp:SqlDataSource>

    </form>
</body>
</html>
```

Juste une datasource avec une commande select et un GridView qui utilise cette datasource.

Le résultat est brut, inesthétique, mais efficace :



CustomerID	NameStyle	Title	FirstName	MiddleName	LastName	Suffix	CompanyName	SalesPerson	
1	☐	Mr.	Orlando	N.	Gee		A Bike Store	adventure-works\pamela0	orlan
2	☐	Mr.	Keith		Harris		Progressive Sports	adventure-works\david8	keith
3	☐	Ms.	Donna	F.	Carreras		Advanced Bike Components	adventure-works\jillian0	donn
4	☐	Ms.	Janet	M.	Gates		Modular Cycle Systems	adventure-works\jillian0	janet
5	☐	Mr.	Lucy		Harrington		Metropolitan Sports Supply	adventure-works\shu0	lucy(
6	☐	Ms.	Rosmarie	J.	Carroll		Aerobic Exercise Company	adventure-works\linda3	rosm

Notez que le GridView a vu que le champ "NameStyle" est un champ booléen, et a donc affiché une case à cocher par défaut.

La grille est en fait une table HTML. Chaque enregistrement est affiche dans une ligne (`TR`) et chaque champ dans une cellule (`TD`). L'entête (ici les noms des champs de la table sont affiches dans un tag `<TH>`.

10.8.2 Sélectionner une ligne

Nous allons améliorer notre gridview et en profiter pour introduire de nouvelles notions. Que diriez-vous d'afficher une GridView avec le n., nom et prénom des clients, et en sélectionnant un client, on afficherait dans une autre gridview la liste des commandes qu'il a passe, le tout, évidemment sans taper une seule ligne de code ?

Nous avons modifié le gridview et ajouté une propriété "**AutoGenerateSelectButton**", initialisée à **True**. Sur chaque ligne du gridview, un bouton "Select" va être ajouté. Nous avons aussi initialisé la propriété "**DataKeyNames**". Cette propriété indique quelle est le nom du champ clef. Quand la ligne de la DataGrid sera sélectionnée, la propriété SelectedValue de la DataGrid aura la valeur de ce champ. Nous verrons plus bas ou l'utiliser.

```
<asp:GridView ID="gdClients" runat="server" DataSourceID="datasourceClients"
DataKeyNames="CustomerID" AutoGenerateSelectButton="true" Width="500" >
</asp:GridView>
```

Nous créons une deuxième gridview simple avec juste une datasource :

```
<asp:GridView ID="gdCommandes" runat="server" DataSourceID="datasourceCommandes"
Width="500" >
</asp:GridView>
```

Nous allons modifier notre commande select pour ne conserver que le n., nom et prénom et nous créons une deuxième datasource pour aller interroger la table SalesLT.SalesOrderHeader (qui est la table des commandes).

La 1ere datasource ne change pas, j'ai juste ajoute une commande "TOP 10" (syntaxe propre a SQL Server) pour limiter les résultats a 10 et ne pas récupérer toute la table :

```
    <asp:SqlDataSource id="datasourceClients"
    SelectCommand="SELECT top 10 CustomerID, LastName, FirstName FROM SalesLT.Customer"
    ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User
Id=sa;Password=password;"
    Runat="server">
    </asp:SqlDataSource>
```

La deuxième datasource est intéressante, elle va afficher dans un GridView la liste des commandes passées par le client sélectionné dans la 1ere DataGrid sans une seule ligne de code !

La commande Select a besoin d'un paramètre, l'id du client, on fait ici comme on l'a fait précédemment, la commande SQL a une clause WHERE et on va utiliser un paramètre @customerId défini plus bas.

```
<asp:SqlDataSource id="dataSourceCommandes"
SelectCommand="SELECT SalesOrderID, OrderDate, TotalDue FROM SalesLT.SalesOrderHeader
where CustomerID = @customerId "
ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User Id=sa;
Password=password;" Runat="server">
    <SelectParameters>
    <asp:ControlParameter name="customerId" ControlID="gdClients"
    PropertyName="selectedValue" />
    </SelectParameters>
</asp:SqlDataSource>
```

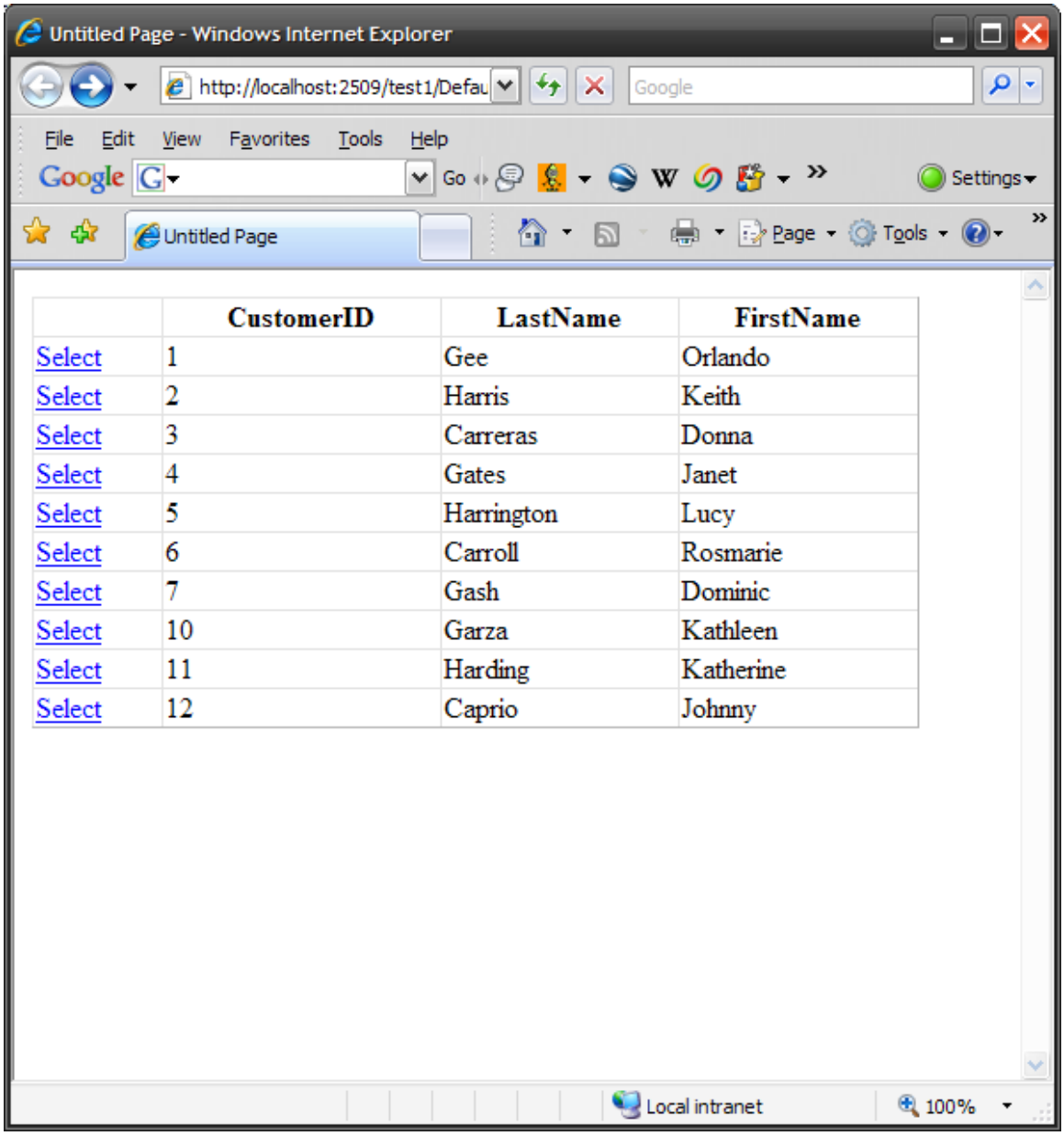
Mais, a la différence de ce qu'on avait vu avec la DataList, au lieu d'utiliser un paramètre de type <asp:Parameter>, nous allons utiliser un paramètre de type **`<asp:ControlParameter>`**.

Quelle est la différence ? Le paramètre <asp:Parameter> doit avoir sa valeur initialisée par du code C#, c'est le résultat d'un calcul, d'une donnée récupérée "manuellement" dans une table ou ailleurs,
Le paramètre <asp:Parameter> va aller chercher la valeur du paramètre dans une propriété d'un autre contrôle ASP.NET présent sur la page!

```
<asp:ControlParameter name="customerId" ControlID="gdClients"
PropertyName="selectedValue" />
```

Qu'avons-nous ici ? un paramètre (customerId) qui va aller chercher sa valeur dans la propriété "**selectedValue**" du contrôle gdClients. Le contrôle gdClients, c'est notre gridview client, sa propriété, c'est la valeur de la ligne sélectionnée dans ce gridview. Dès que la propriété "**SelectedValue**" sera modifiée dans le contrôle gdClient (c'est-à-dire quand l'utilisateur aura cliqué sur le bouton "Sélectionner"), la data source sera régénérée et le contrôle associé a cette datasource sera repeuplé sans aucune ligne de code (du moins de notre part !).

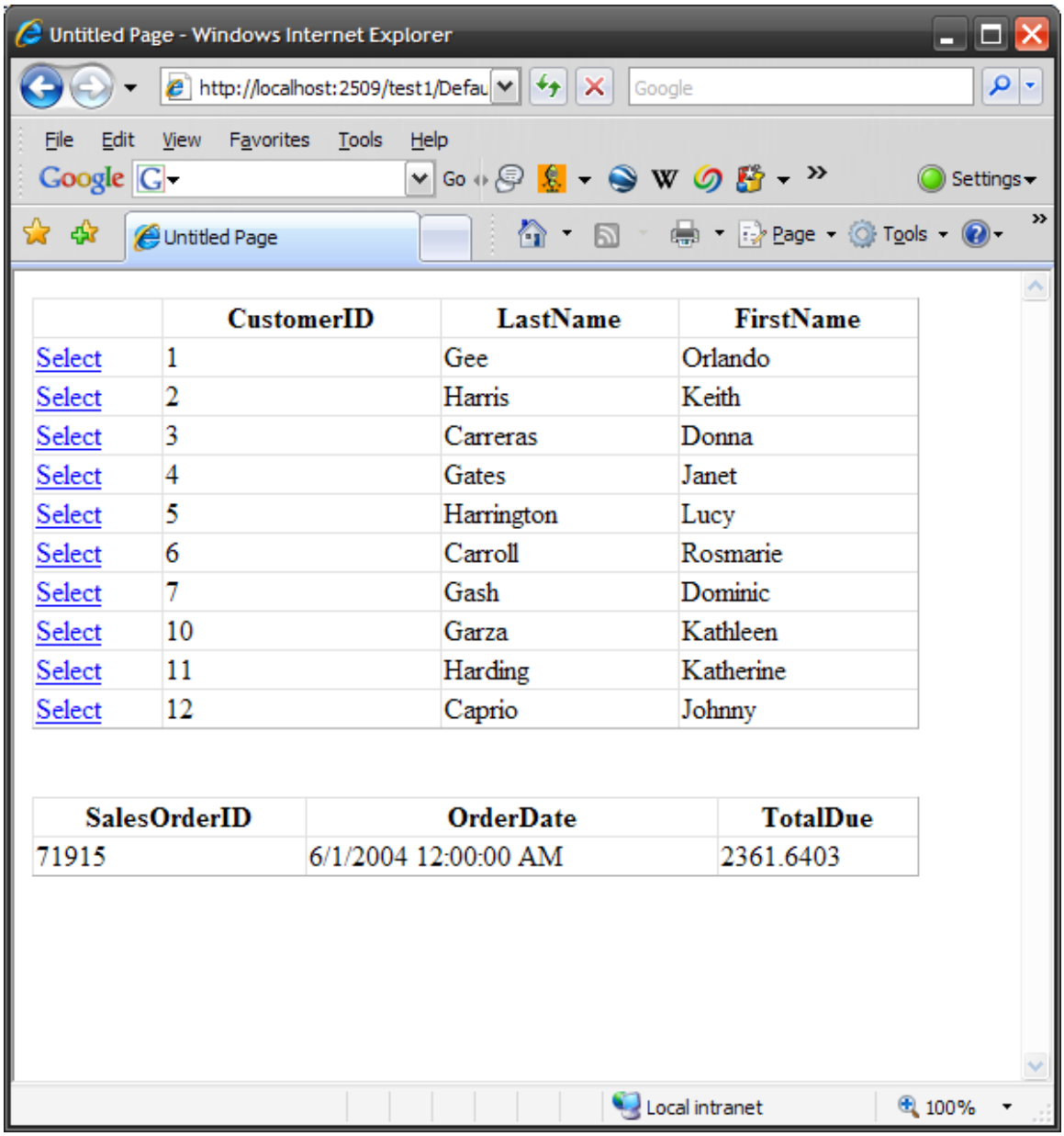
Qu'est ce que ca donne ?



	CustomerID	LastName	FirstName
Select	1	Gee	Orlando
Select	2	Harris	Keith
Select	3	Carreras	Donna
Select	4	Gates	Janet
Select	5	Harrington	Lucy
Select	6	Carroll	Rosmarie
Select	7	Gash	Dominic
Select	10	Garza	Kathleen
Select	11	Harding	Katherine
Select	12	Caprio	Johnny

La liste s'affiche, chaque ligne est précédée d'un bouton select.

Si on clique sur un bouton select (par exemple celui du client 6), la ligne est sélectionnée, la valeur de la propriété "**SelectedValue**" du gridView est donc celle du champ CustomerID (6). La valeur étant modifiée, la datasource "dataSourceCommand" va relancer sa commande select et remplir le gridview associé :



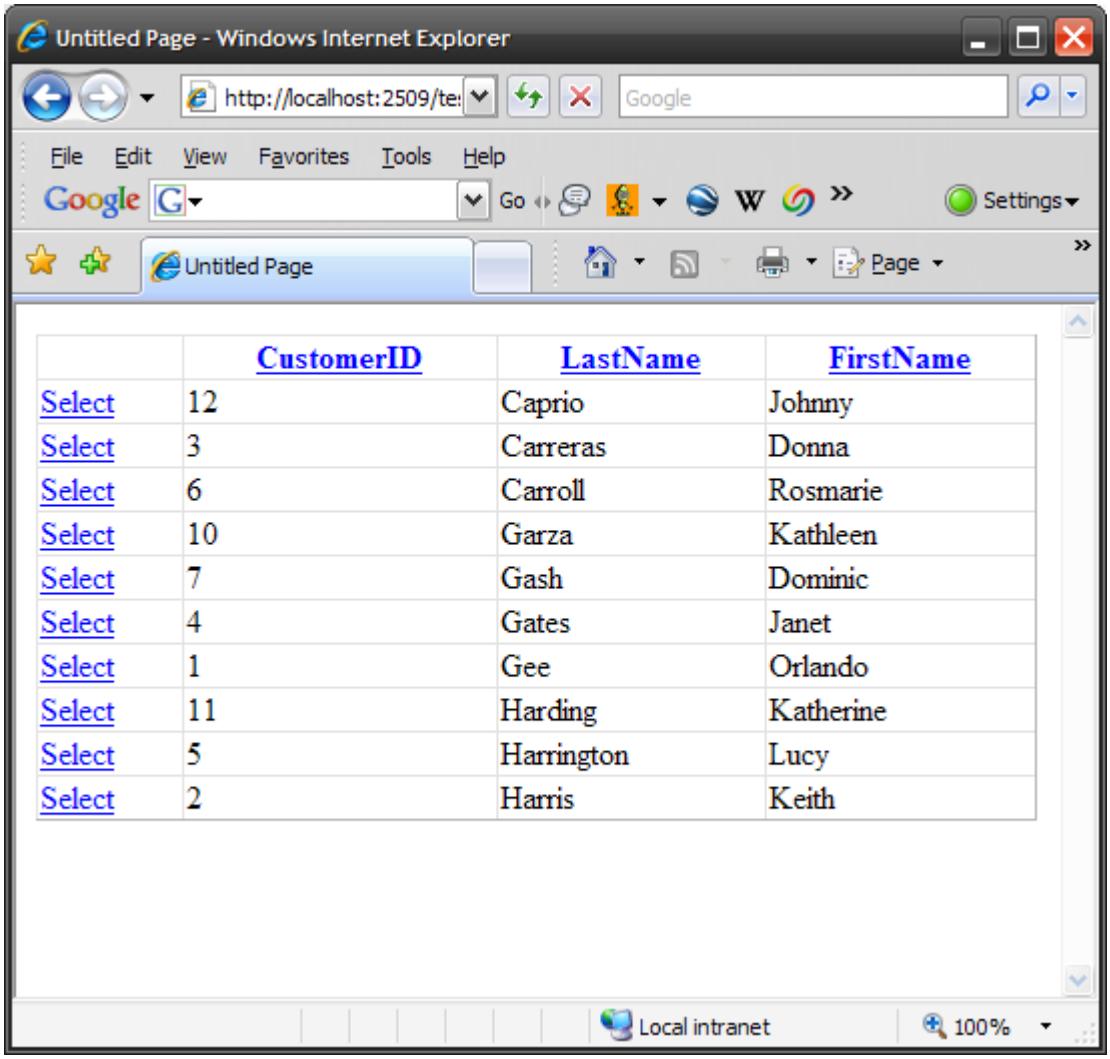
	CustomerID	LastName	FirstName
Select	1	Gee	Orlando
Select	2	Harris	Keith
Select	3	Carreras	Donna
Select	4	Gates	Janet
Select	5	Harrington	Lucy
Select	6	Carroll	Rosmarie
Select	7	Gash	Dominic
Select	10	Garza	Kathleen
Select	11	Harding	Katherine
Select	12	Caprio	Johnny

SalesOrderID	OrderDate	TotalDue
71915	6/1/2004 12:00:00 AM	2361.6403

A vous de jouer pour ajouter un troisième gridview qui va afficher le détail de la commande (table SalesOrderDetail) quand l'utilisateur cliquera sur une commande.

10.8.3 Trier les colonnes

Vous voulez trier les colonnes ? Ajoutez juste a votre gridview la propriété "AllowSorting = true" :



	CustomerID	LastName	FirstName
Select	12	Caprio	Johnny
Select	3	Carreras	Donna
Select	6	Carroll	Rosmarie
Select	10	Garza	Kathleen
Select	7	Gash	Dominic
Select	4	Gates	Janet
Select	1	Gee	Orlando
Select	11	Harding	Katherine
Select	5	Harrington	Lucy
Select	2	Harris	Keith

Chaque entête est devenue un lien, le fait de cliquer dessus (j'ai clique ici sur "LastName") trie la gridview sur la colonne dans l'ordre ascendant, un second click dessus le fait en ordre descendant SANS UNE SEULE LIGNE DE CODE !

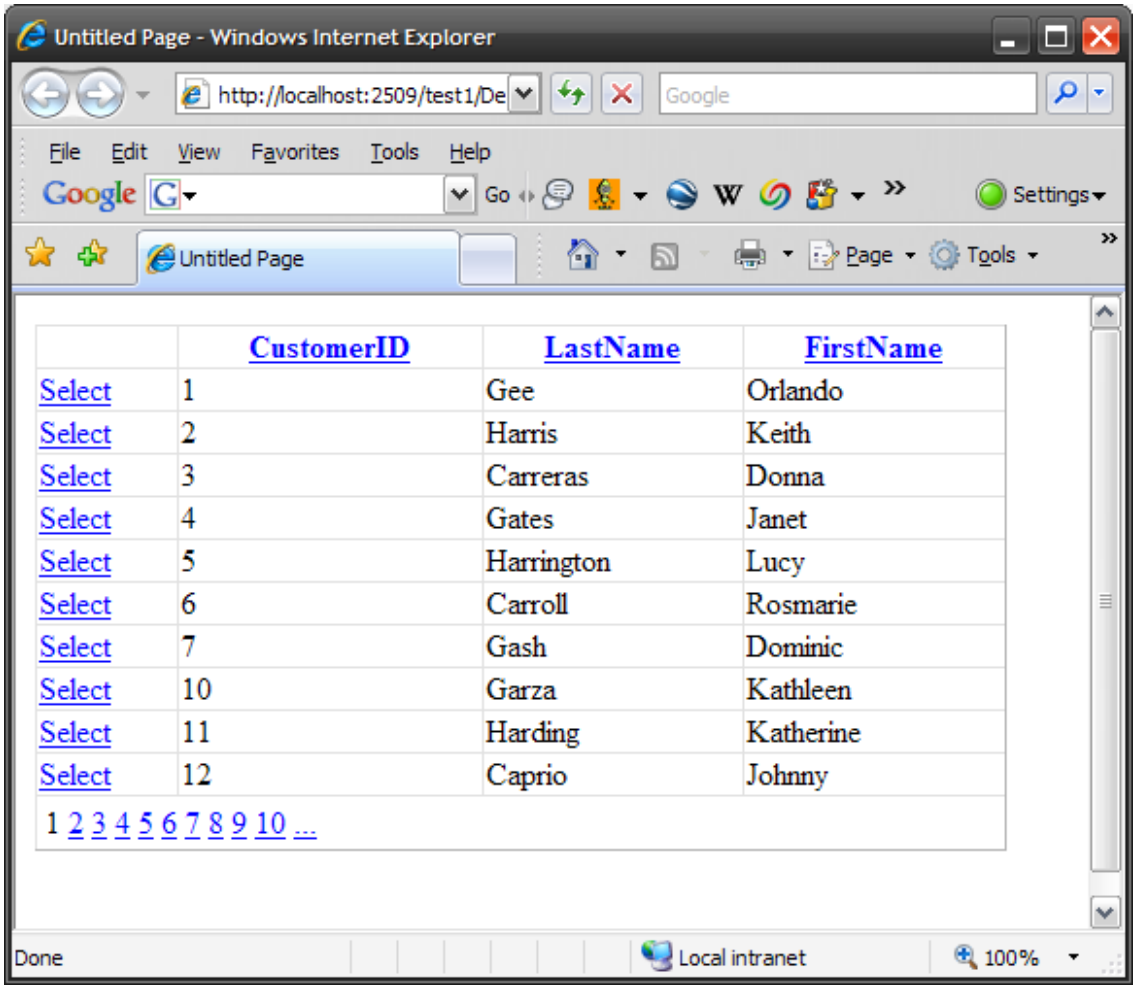
Vous pouvez également trier la gridview par programme en appelant la méthode Sort de l'object GridView.

Nous ferions : `gdCustomers.Sort(nom_du_champ, direction);`

Direction étant `SortDirection.Ascending` ou `SortDirection.Descending`;

10.8.4 Pagination

Vous avez remarque que j'avais limité l'affichage à 10 lignes pour éviter d'avoir des pages trop longues. En effet, par défaut, la GridView affiche tous les enregistrements de la table. On peut limiter le nombre de lignes affichées simultanément dans la GridView en initialisant la propriété "AllowPaging" (permettre la pagination) à true. Retirons donc notre TOP 10 de la commande select et ajoutons cette propriété :



Et instantanément, sans aucune ligne de code, la pagination est faite automatiquement, on peut passer de page en page juste en cliquant sur un des numéros dans le pied de la gridview.
Par défaut, il y a 10 lignes affichées, si vous voulez en afficher plus ou moins, vous pouvez modifier la propriété "PageSize" de la GridView.

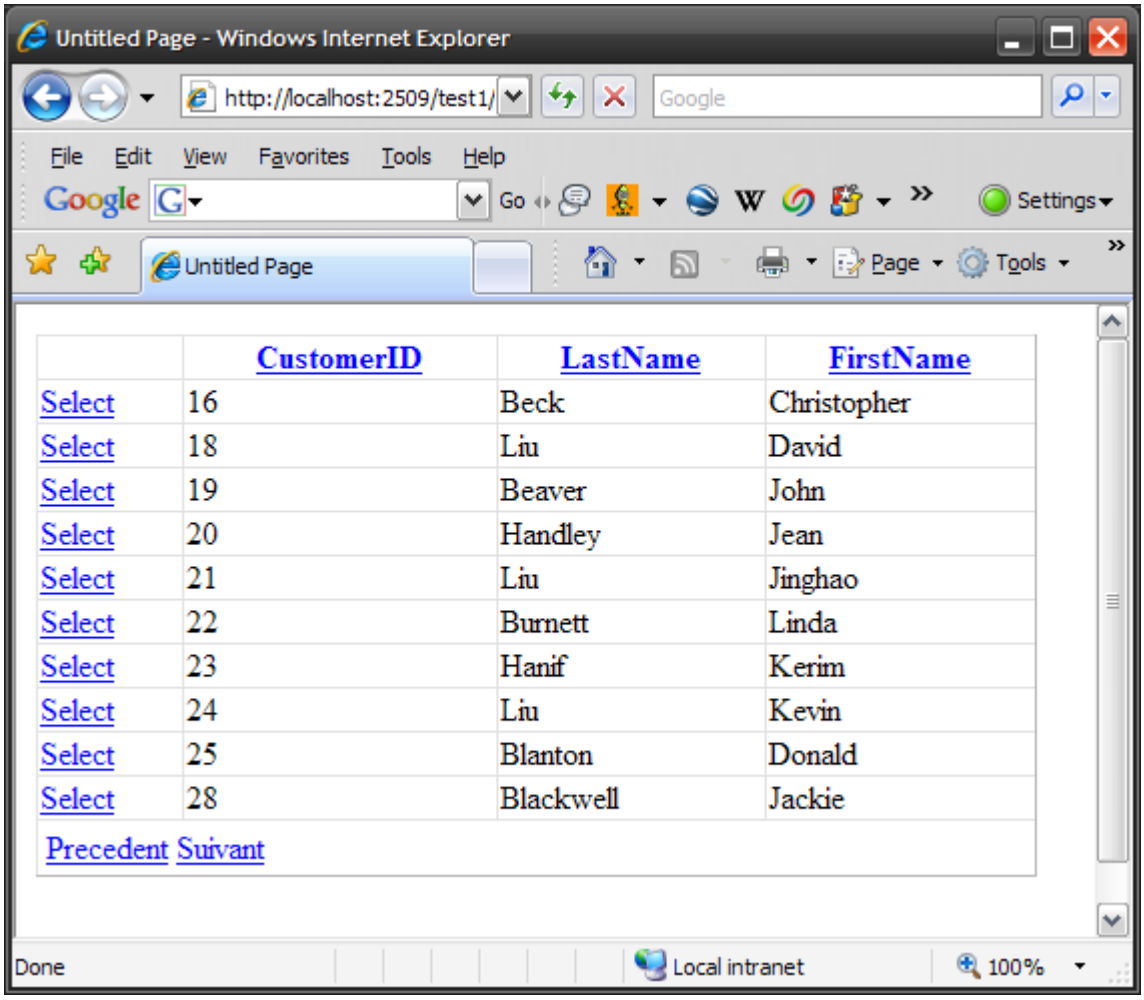
10.8.5 Personnaliser la pagination

Par défaut, le gridView affiche une suite de nombres, vous pouvez personnaliser cet affichage en utilisant les propriétés suivantes :

PagerSettings-FirstPageImageUrl	URL d'une image pour le lien sur la 1ère page
PagerSettings-FirstPageText	Texte pour le lien sur la 1ère page
PagerSettings-LastPageImageUrl	URL d'une image pour le lien sur la dernière page
PagerSettings-LastPageText	Texte pour le lien sur la dernière page
PagerSettings-Mode	Type de pagination: • **NextPrevious** : affiche un lien sur la page suivante et la page précédente • **NextPreviousFirstLast** : idem mais affiche en plus un lien sur la première et la dernière page • **Numeric** : Affiche le numéro des pages • **NumericFirstLast** : Affiche le numéro des pages avec un lien sur la première et un lien sur la dernière
PagerSettings-NextPageImageUrl	URL d'une image pour le lien sur la dernière page
PagerSettings-PageButtonCount	Indique le nombre de numéros de pages maximum a afficher (10 par défaut)
PagerSettings-Position	Positionnement de la pagination : • **Bottom** : sous la GridView • **Top** : au dessus de la GridView • **TopAndBottom** : les deux
PagerSettings-PreviousPageImageUrl	URL d'une image pour le lien sur la page précédente
PagerSettings-PreviousPageText	Texte pour le lien sur la page précédente
PagerSettings-Visible	Affiche / cache la pagination

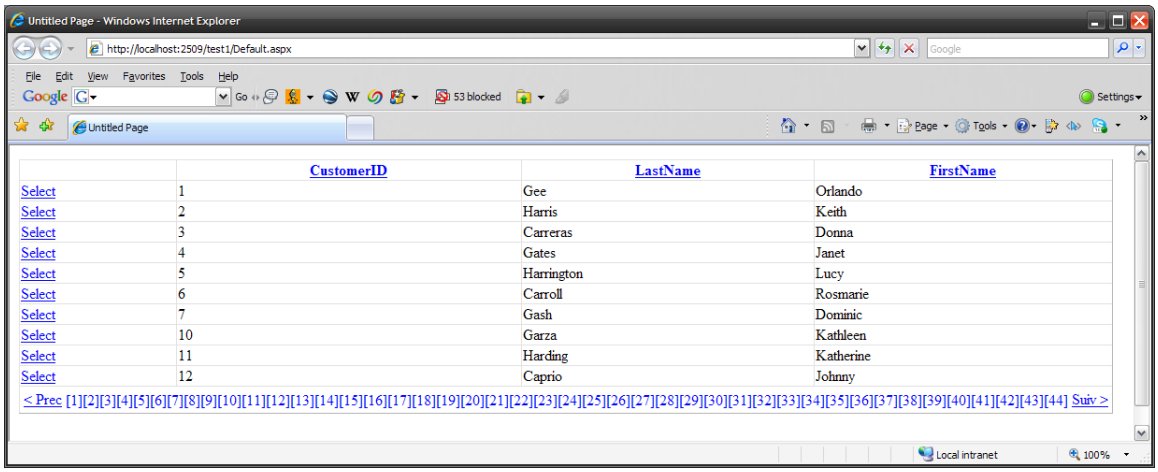
Juste pour voir, modifions le texte du lien vers les pages suivantes et précédentes :

```
<asp:GridView AllowPaging="true" AllowSorting="true" ID="gdClients" runat="server"
DataSourceID="datasourceClients" DataKeyNames="CustomerID"
AutoGenerateSelectButton="true" Width="500"
PagerSettings-NextPageText="Suivant" PagerSettings-PreviousPageText="Precedent"
PagerSettings-Mode="NextPrevious">
```



Mais le contrôle GridView vous permet de personnaliser complètement votre pagination grâce au template nomme "PagerTemplate".

Vous avez vu que, par défaut, vous ne pouvez afficher a la fois les numéros des pages et des liens vers les pages précédentes et suivantes, nous allons pouvoir le faire avec un template de pagination :



Bon, dans notre cas, ce n'est pas très pratique vu le nombre de pages, à vous de le modifier pour n'afficher qu'une partie des pages.

Comment avons-nous fait ?

```
<asp:GridView OnDataBound="gdClients_DataBound" AllowPaging="true" AllowSorting="true"
ID="gdClients" runat="server" DataSourceID="datasourceClients"
DataKeyNames="CustomerID" AutoGenerateSelectButton="true" Width="500" >
  <PagerTemplate>
     <table>
     <tr>
      <td>
         <asp:LinkButton id="lnkPrevious" Text="&lt; Prec"
          CommandName="Page"
          CommandArgument="Prev" Runat="server" />
      </td>
      <td>
         <asp:Menu id="menuPager" Orientation="Horizontal"
          OnMenuItemClick="menuPager_MenuItemClick" Runat="server" />
      </td>
      <td>
        <asp:LinkButton id="lnkNext" Text="Suiv &gt;"
         CommandName="Page"
         CommandArgument="Next" Runat="server" />
      </td>
    </tr>
    </table>
    </PagerTemplate>
    </asp:GridView>

    <br /><br />
    <asp:GridView ID="gdCommandes" runat="server"
    DataSourceID="datasourceCommandes"
    Width="500" >
    </asp:GridView>
```

Nous avons donc un template de type PagerTemplate. Dans ce template, comme dans tous les templates, vous pouvez placer des contrôles ASP.NET, des tags HTML ou du texte. Notre paginateur (si j'ose dire) est en fait une table, elle a trois cellules : le bouton allant sur la page précédente (qui est un contrôle LinkButton), un contrôle Menu (que nous n'avons pas encore vu), qui affiche des menus soit horizontalement, soit verticalement, ici, nous allons utiliser ce contrôle pour afficher la liste des pages et un autre contrôle LinkButton pour passer a la page suivante.

Regardez bien comment ASP.NET détermine comment ces boutons doivent fonctionner : les boutons "précédent" et "suivant" ont leur propriété **CommandName** initialisée avec "**Page**", cela signifie que ce contrôle va permettre de se déplacer dans les pages. La nature du déplacement est indiquée dans la propriété **CommandArgument**. Elle peut prendre une des valeurs suivantes :

Next	Page suivante
Prev	Page précédente
First	Première page
Last	Dernière page
x	Un nombre entier indiquant une page

Pour la liste des pages, nous procédons ainsi dans le code C#

```
protected void gdClients_DataBound(object sender, EventArgs e)
{
 Menu menuPager = (Menu)gdClients.BottomPagerRow.FindControl("menuPager");
    for (int i = 0; i < gdClients.PageCount; i++)
    {
        MenuItem item = new MenuItem();
        item.Text = "[" + (i + 1) + "]";
        item.Value = i.ToString();
        menuPager.Items.Add(item);
    }
}
```

Cette fonction est appelée lorsque l'évènement DataBound se déclenche sur le gridview (voir la propriété `OnDataBound` dans la déclaration du data grid). On va construire ici la liste des pages à afficher. Pourquoi cet évènement ? Parce qu'il se déclenche une fois les données liées au gridview. On connait donc ici combien de pages il va y avoir dans le contrôle. On obtiendra ce nombre via la propriété "**PageCount**" du gridview.

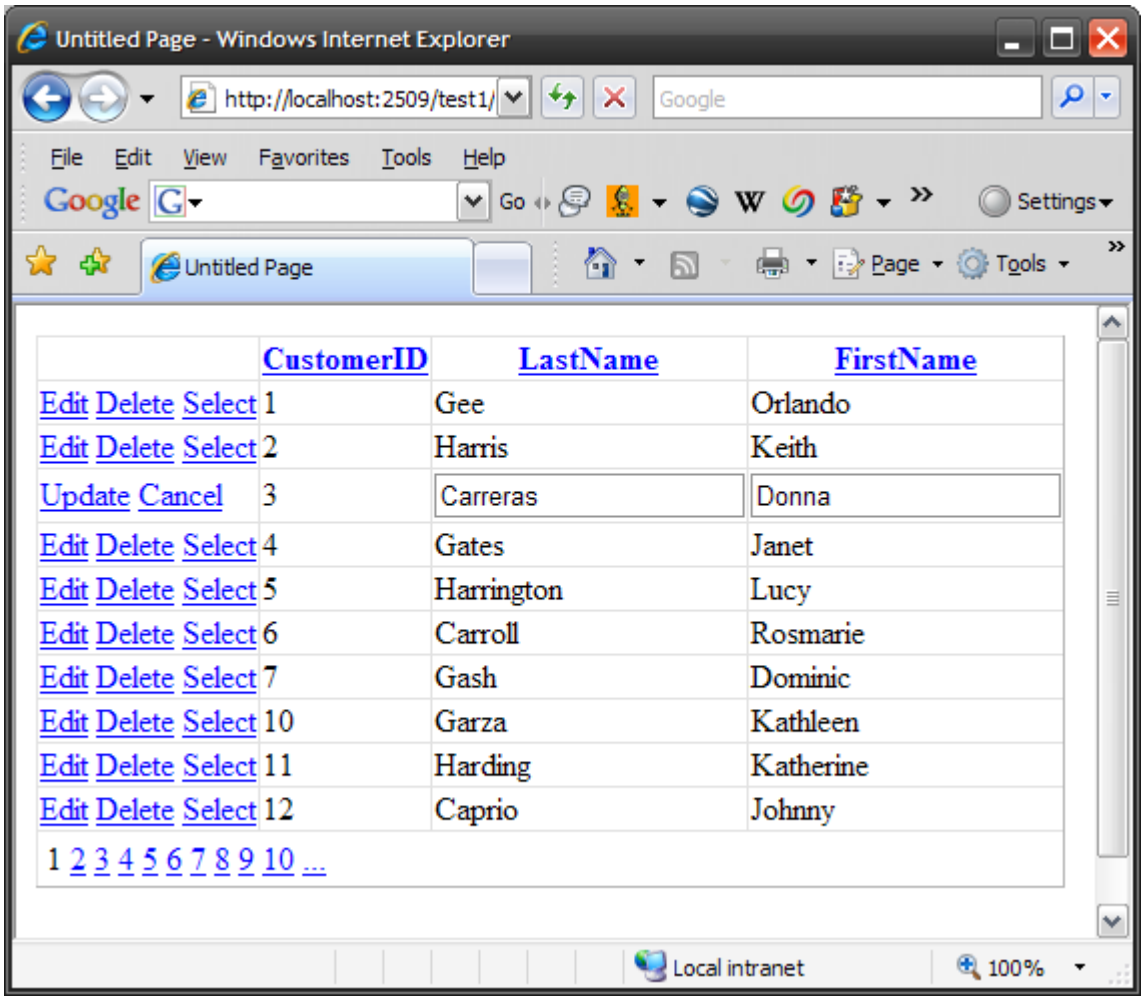
Que fait cette fonction ? Elle récupère le paginateur situé en bas du contrôle (BottonPagerClient) et récupère dans ce paginateur le contrôle Menu, puis ajoute la liste des pages dans le menu.

Enfin, cette fonction est appelée quand on clique sur un des items du menu, on se contente de changer la page courante du gridview (propriété PageIndex) en fonction de l'item clique.

```
protected void menuPager_MenuItemClick(object sender, MenuEventArgs e)
{
    gdClients.PageIndex = Int32.Parse(e.Item.Value);
}
```

10.8.6 Editer avec un gridview

Nous avions vu qu'on pouvait éditer des enregistrements avec une Data List. Même si c'était assez simple, il fallait quand même (c'est bien dommage) taper du code C#. Au XXIème siècle, moins on tape de code, mieux on se porte. Le GridView va donc nous permettre d'éditer des enregistrements sans taper une seule ligne de code.



Regardez bien cet écran, sans *aucune* ligne de code, nous avons des données provenant d'une base de données, des colonnes triables en ordre ascendant ou descendant, une pagination des résultats et maintenant deux liens : Edit et Delete pour supprimer l'enregistrement choisi (en cliquant sur "Delete") ou l'éditer en cliquant sur "Edit" : dans ce cas, les champs deviennent des TextBox, le contenu est éditable et vous pouvez le mettre à jour en cliquant sur le lien "Update" ou annuler la modification en cliquant sur le lien "Cancel".

Comment-a-t-on fait ?

Simple, dans les propriétés du grid view, on a ajoute les propriétés suivantes :

```
AutoGenerateEditButton="true"
AutoGenerateDeleteButton="true"
```

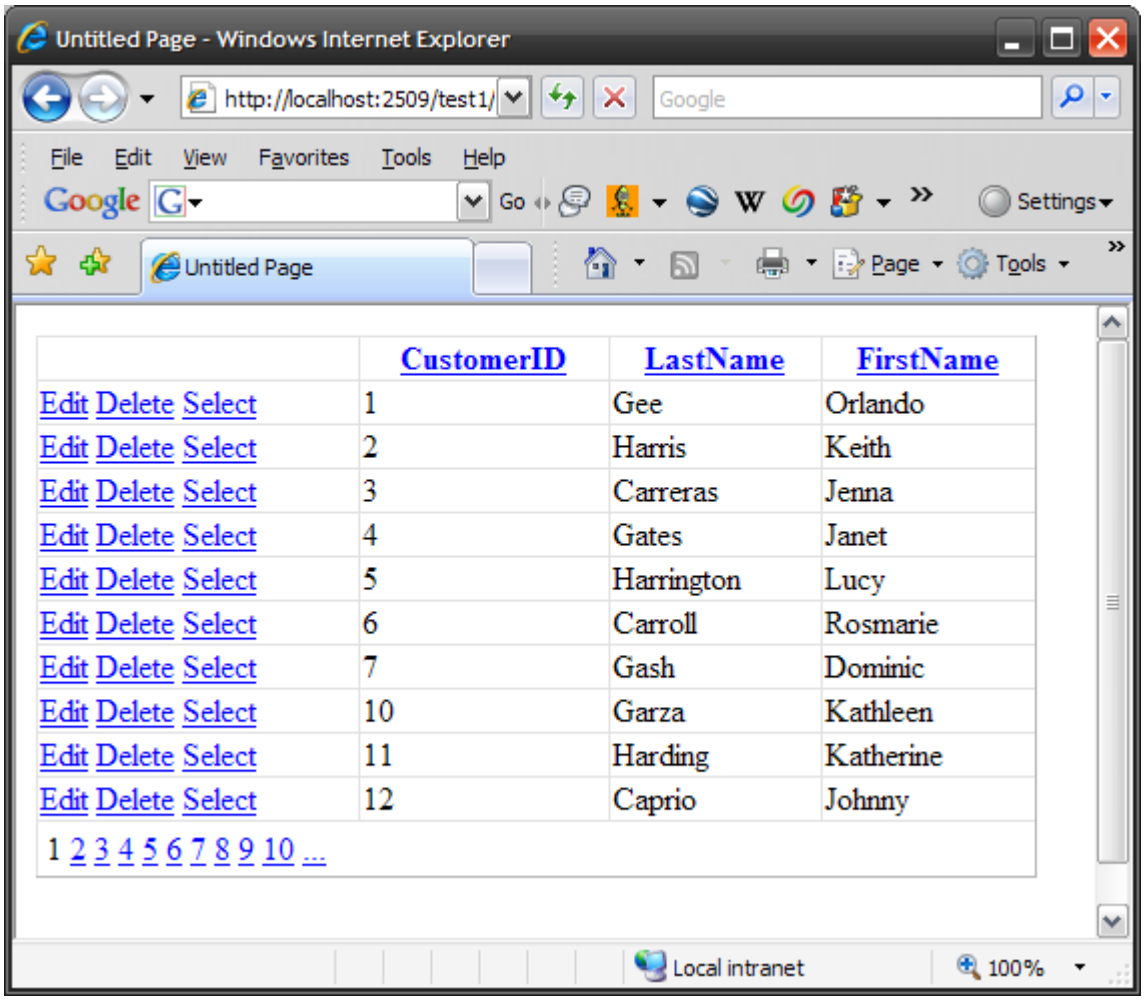
Et dans la data source, on a ajoute les requêtes nécessaires a la modification et a la suppression :

```
UpdateCommand="UPDATE SalesLT.Customer SET LastName=@LastName, FirstName=@FirstName
WHERE CustomerID=@CustomerID"
DeleteCommand="DELETE SalesLT.Customer WHERE CustomerID=@CustomerID"
```

Les deux commandes seront appelées lors de la modification et de la suppression. Contrairement au Data List ou il fallait, à la main, initialiser les paramètres, ici, vous ne faites rien. @FirstName correspond à ce qui est tapé par l'utilisateur dans le champ FirstName, @LastName à ce qu'il a tapé dans le champ LastName et @CustomerID, ne pouvant pas être modifié par l'utilisateur (car c'est la clef définie par la propriété `DataKeyNames="CustomerID"`) sera la valeur de ce champ pour cet enregistrement.

Le principe est le même pour la suppression.

Je clique donc sur Donna, je remplace par Jenna, je clique sur Update :



	CustomerID	LastName	FirstName
Edit Delete Select	1	Gee	Orlando
Edit Delete Select	2	Harris	Keith
Edit Delete Select	3	Carreras	Jenna
Edit Delete Select	4	Gates	Janet
Edit Delete Select	5	Harrington	Lucy
Edit Delete Select	6	Carroll	Rosmarie
Edit Delete Select	7	Gash	Dominic
Edit Delete Select	10	Garza	Kathleen
Edit Delete Select	11	Harding	Katherine
Edit Delete Select	12	Caprio	Johnny

1 2 3 4 5 6 7 8 9 10 ...

Essayez la même chose en PHP ou Java (ah mince, je m'étais promis de ne pas critiquer les autres technos)

Un truc en passant : lorsque vous cliquez sur "Update", un postback est fait et la requête est effectuée sur le serveur. La page est donc rechargée en retour.

Ce rechargement peut être un peu gênant : si l'utilisateur modifie un enregistrement en bas de la page, après rechargement, il se retrouve en haut de la page, alors qu'il s'attend à rester la où il était.

Pour pallier ce problème, il suffit d'ajouter dans la directive `<%@ Page %>` la propriété "

MaintainScrollPositionOnPostback="true" , le navigateur repositionnera la page la où elle était avant le postback. Cette directive fonctionne avec IE, Firefox et Opera.

10.8.7 Pas d'enregistrements

Lors du databinding, aucun résultat peut être renvoyé (requête suite a une recherche qui ne retourne rien par exemple). Dans ce cas, plutôt que d'afficher un gridview vide, on prévoira de renseigner la propriété "**EmptyDataText**" avec un message a afficher en cas de résultat vide.

Si on considère que un seul message ne suffit pas, on peut utiliser le template **EmptyDataTemplate**. Ce template vous permet d'afficher n'importe quel type de contenu (comme tous les templates) : contrôles ASP.NET, HTML ou texte. Le contenu de ce template ne sera affiché que si le résultat de la commande Select ne renvoie aucun résultat.

10.8.8 Styler le gridview

Le GridView possède toute une liste de propriétés pour styler son contenu. Je ne préconise pas d'utiliser ces propriétés pour styler le contrôle (d'ailleurs, je ne préconise pas d'utiliser ces propriétés pour styler n'importe quel control ASP.NET). Ces styles sont en fait des "raccourcis" vers des styles css et on se retrouve avec des pages très lourdes car chaque cellule du gridview a une propriété style de 10 kilomètres.
Autant utiliser les classes css et affecter a chaque élément du gridview une classe css.

Les éléments stylables sont les suivants :

RowStyle-CssClass	Classe css pour chaque ligne de la gridview
AlternatingRowStyle-CssClass	Quand utilisé, cette classe est utilisée une ligne sur deux
HeaderStyle-CssClass	Classe css de l'entête
FooterStyle-CssClass	Classe css du pied
PagerStyle-CssClass	Classe css du pager
SelectedRowStyle-CssClass	Classe css de la ligne sélectionnée

En plus de ces propriétés, les propriétés suivantes sont utiles :

GridLines	Affichage des lignes du tableau : • **Both** : verticales et horizontales • **Vertical** : lignes verticales • **Horizontal** : lignes horizontales • **None** : pas d'affichage
ShowFooter	Affiche une ligne supplémentaire en pied du tableau
Show Header	Affiche une ligne en entête du tableau

10.8.9 Personnaliser les champs du gridview

Par défaut, le GridView affiche chaque champ comme un champ texte avec l'information "brute", c'est-à-dire telle qu'elle est lue dans la base. Ca pourrait être intéressant de pouvoir soit formater le texte affiché (un total affiché avec des décimales, une date au bon format) ou d'utiliser autre chose que du texte pour l'affichage (case à cocher, liste déroulante ou lien HyperText).

Une solution existe, elle consiste a spécifier explicitement quels sont les champs à afficher dans le gridview et comment les afficher. Le gridview supporte 7 types de champs :

BoundField	Un champ texte qui peut être formaté
CheckBoxField	La valeur du champ est affichée comme une case à cocher
CommandField	Affiche des liens pour éditer, supprimer ou sélectionner une ligne
ButtonField	Affiche le contenu comme un bouton (Button, LinkButton ou ImageButton)
HyperLinkField	Affiche le contenu du champ sous forme d'un lien hypertexte.
ImageField	Affiche un champ comme une image
TemplateField	Le plus puissant : vous pouvez afficher le contenu du champ grâce à un template, vous pouvez donc afficher ici n'importe quel type de données.

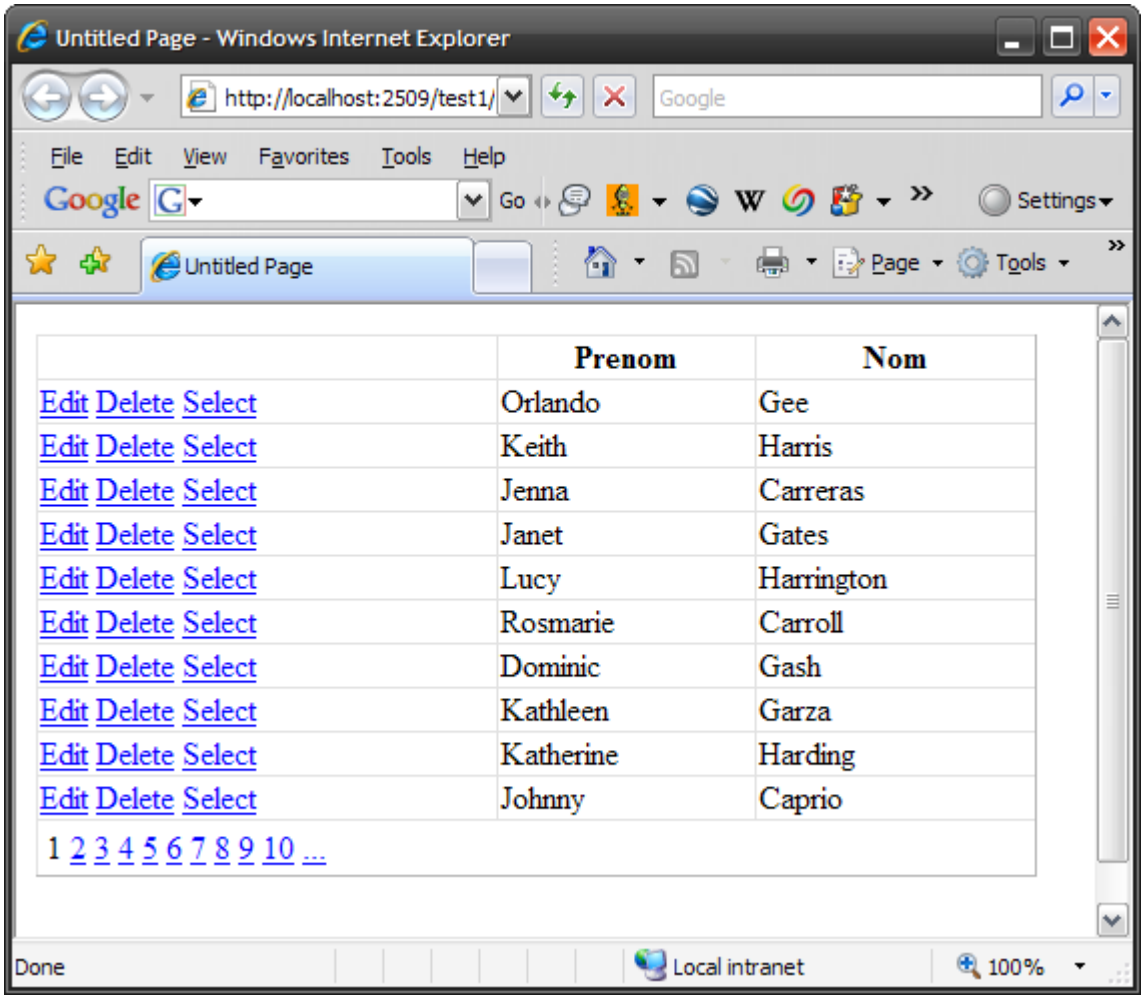
Comment va-t-on les utiliser ?

```
<asp:GridView>
   <Columns>
      <asp:BoundField … />
      <asp:CommandField … />
   </Columns>
</asp:GridView>
```

On place les champs dans un tag **<column> </column>**

10.8.10 Le BoundField

Le Hounsfield affiche le contenu du champ comme du texte. Il possède de plus des propriétés permettant de personnaliser cet affichage.



Dans cet exemple, nous avons deux BoundFields, un qui utilise le champ "FirstName" et un qui utilise le champ "LastName" :

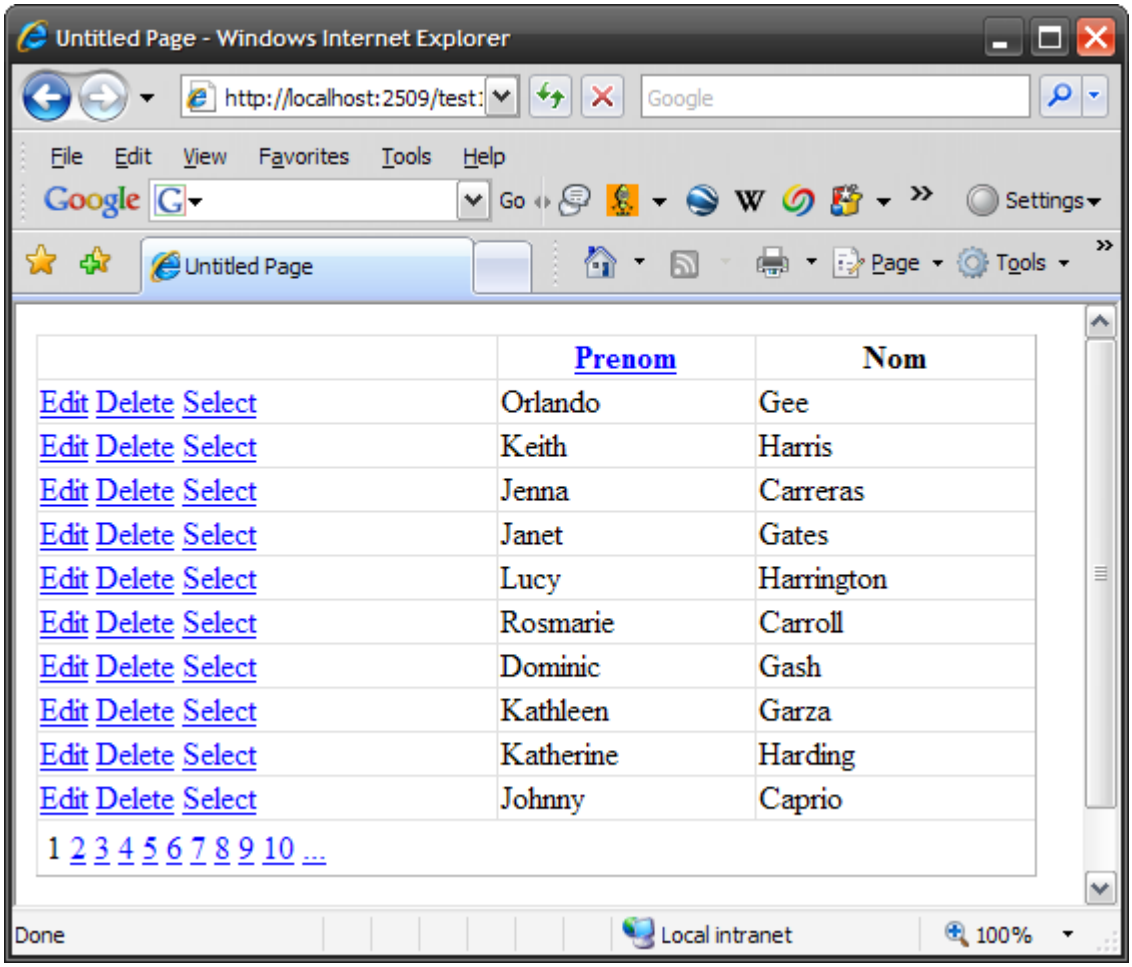
```
<asp:GridView AllowPaging="true" AllowSorting="true" ID="gdClients" runat="server"
 DataSourceID="datasourceClients" DataKeyNames="CustomerID"
 AutoGenerateSelectButton="true"  AutoGenerateEditButton="true"
 AutoGenerateDeleteButton="true" AutoGenerateColumns="false" Width="500" >
 <Columns>
     <asp:BoundField DataField="FirstName" HeaderText="Prenom" />
     <asp:BoundField DataField="LastName" HeaderText="Nom" />
 </Columns>
</asp:GridView>
```

Avant tout, une nouvelle propriété du GridView est déclarée : `AutoGenerateColumns="false"`. Par défaut, en effet, le Grid View génère automatiquement une colonne par champ renvoyé par la commande Select. Si vous omettez cette déclaration, vos colonnes personnalisées seront affichées *en plus* de tous les champs renvoyées par la requête.

Notez que la propriété du BoundField utilisée pour lier le champ au champ de la table est "**DataField**". Vous noterez au passage que la propriété "**HeaderText**" permet de modifier l'entête de la colonne.
Les plus observateurs auront noté que le lien dans l'entête de la colonne a disparu. Cela signifie-t-il qu'on ne peut plus trier la colonne ? Avec cette déclaration, c'est le cas. Par défaut, le GridView ne permet pas le tri sur les colonnes quand on utilise des colonnes personnalisées (ce qui est un peu normal, il ne sait pas ce que vous allez afficher dans la colonne).

Comment faire alors ? Il va falloir indiquer au GridView la façon de trier la colonne, on le fait en utilisant la propriété "**SortExpression**" :

```
<asp:BoundField DataField="FirstName" HeaderText="Prenom" SortExpression="FirstName" />
```



On a indiqué ici pour le 1er champ que l'expression utilisée pour le tri est le champ "FirstName".

Les autres propriétés du BoundField sont :

DataField	Le champ de la table utilisé
DataFormatString	Vous pouvez utiliser un formatage spécifique pour l'affichage. Ce formatage est défini par une chaine de caractère (un peu comme les chaines de formatage du printf en C). Vous trouverez la définition de ces caractères ici : **http://msdn.microsoft.com/fr-fr/library/system.web.ui.webcontrols.boundfield.dataformatstring.aspx**
ApplyFormatInEditMode	Par défaut, lorsqu'on repasse en mode édition, on perd le formatage défini juste avant et on repasse en mode "donnée brute". Pour conserver le formatage durant l'édition, on met cette propriété a true.
FooterText	Texte a afficher dans le pied de la colonne
FooterStyle	Style du pied de la colonne
HeaderImageUrl	URL d'une image a afficher dans l'entête de la colonne
HeaderStyle	Style de l'entête de la colonne
HtmlEncode	Encode en html la valeur à afficher (ex : si la valeur est gras, le texte affiche sera gras, sinon, ce sera **gras.**
ItemStyle	Style du contenu du champ
NullDisplayText	Texte a afficher si le contenu du champ est null

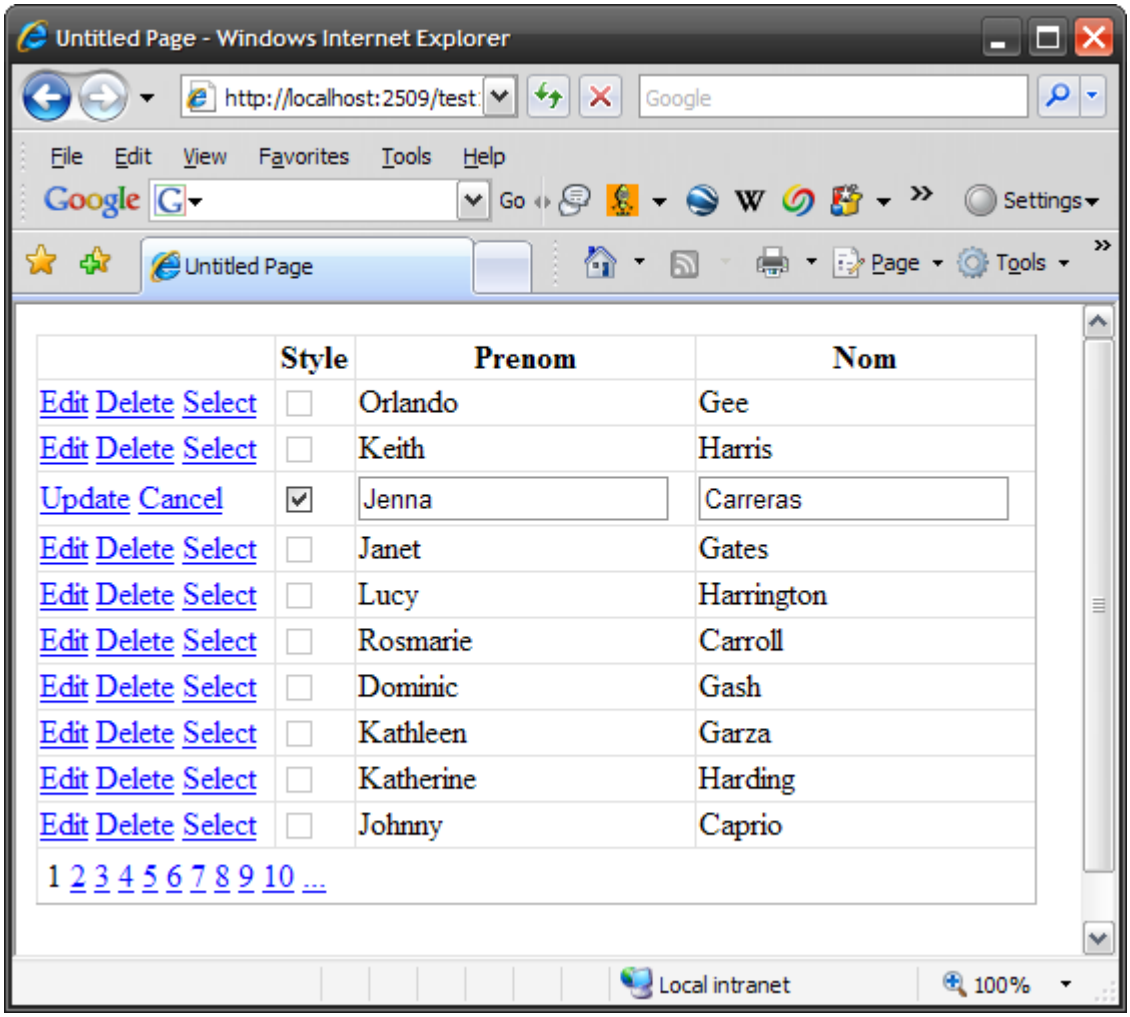
ReadOnly	True si le champ ne peut être édité
Visible	Affiche / Cache la colonne

10.8.11 Le CheckBoxField

Le CheckboxField affiche une boite à cocher a la place du contenu du champ. Ca n'a pas tellement de sens de l'utiliser avec autre chose qu'un champ binaire (bit, booléen, selon votre base de données). Vous ne pouvez bien sur n'éditer le champ que si vous êtes en mode édition, autrement, la case sera affichée mais désactivée.

Voyons ce que ca donne avec le champ "NameStyle" de notre table qui, justement, est un champ binaire :

```
<asp:CheckBoxField DataField="NameStyle" HeaderText="Style"  />
```



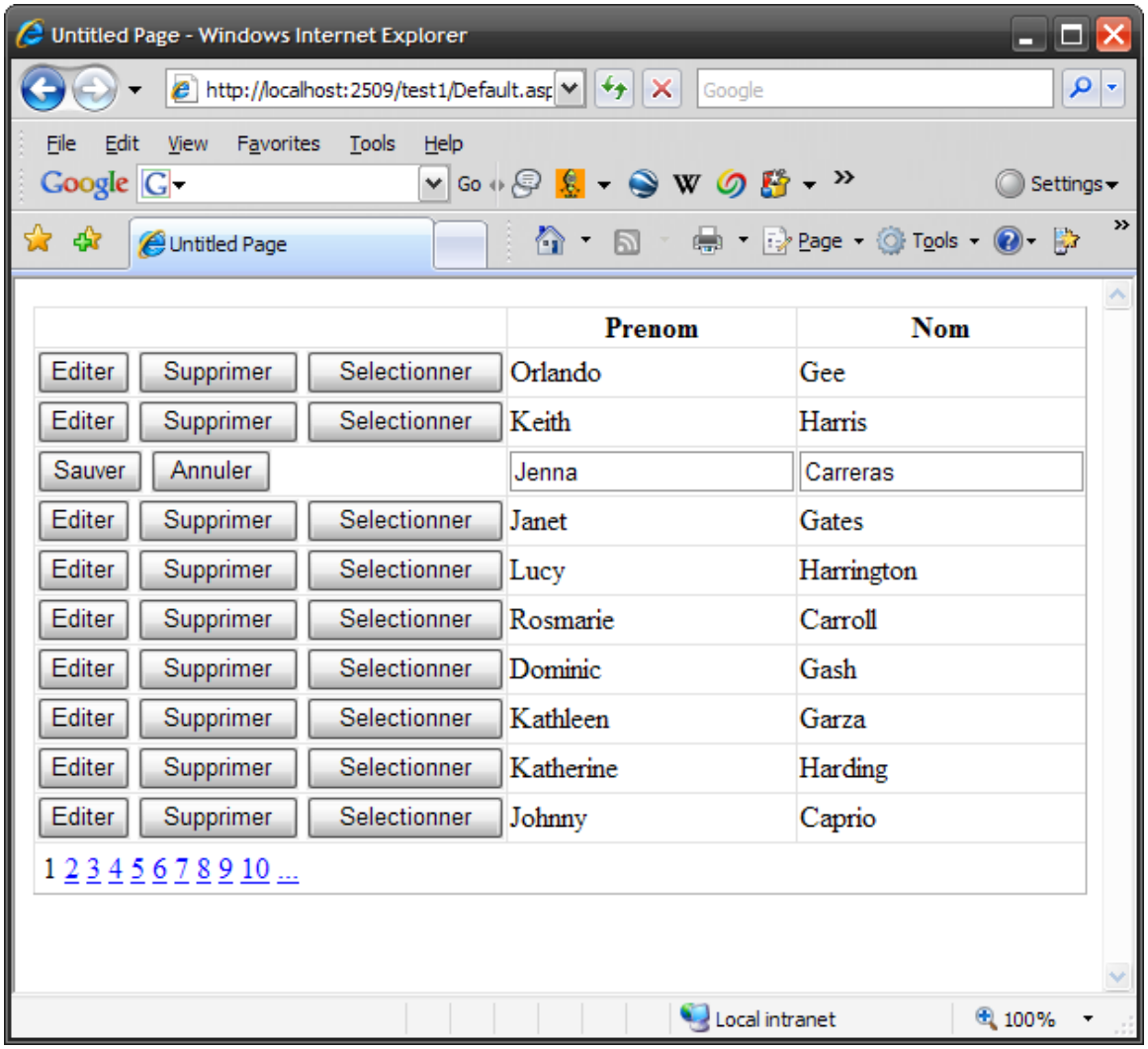
Toutes les cases a cocher reflètent le contenu de la table (les champs sont à false dans la table), seule la case a cocher de la ligne en cours d'édition est modifiable.

Le CheckBoxField a également une propriété "**Text**" qui permet, si vous le désirez, d'afficher un texte a cote de la cas à cocher.

10.8.12 Le CommandField

Le CommandField vous permet de modifier l'apparence des boutons ou liens d'édition, de suppression, de sélection, de mise a jour et d'abandon de mise a jour (ce qui n'est pas du luxe).

Modifions donc notre grid view pour afficher, a la place de nos liens, des boutons avec un texte différent :



```
<asp:GridView AllowPaging="true" AllowSorting="true" ID="gdClients" runat="server"
 DataSourceID="datasourceClients" DataKeyNames="CustomerID" AutoGenerateColumns="false"
 Width="500" >
    <Columns>
        <asp:CommandField
            ButtonType="Button"
            ShowEditButton="true"
            EditText="Editer"
            ShowDeleteButton="true"
            DeleteText="Supprimer"
            ShowSelectButton="true"
            SelectText="Sélectionner"
            UpdateText="Sauver"
            CancelText="Annuler"
            />
        <asp:BoundField DataField="FirstName" HeaderText="Prenom" />
        <asp:BoundField DataField="LastName" HeaderText="Nom" />
    </Columns>
</asp:GridView>
```

Nous avons supprimé les déclarations qui permettent la génération automatique des boutons (`AutoGenerateSelectButton="true" AutoGenerateEditButton="true" AutoGenerateDeleteButton="true"`) du gridview et ajoute un contrôle CommandField.

Les propriétés du CommandField sont les suivantes :

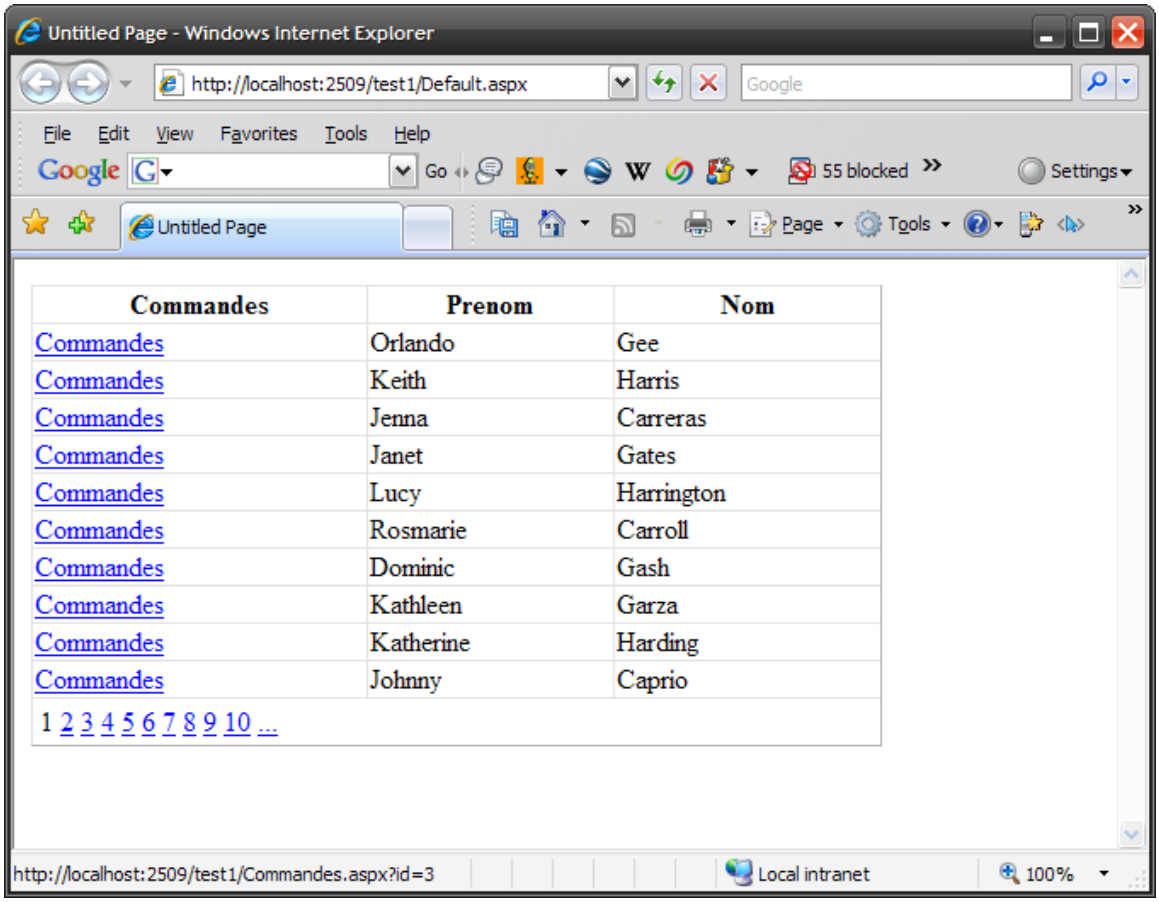
ButtonType	Type de bouton : • **Button** (ce que nous avons utilise) • **Image** : une image • **Link** : un lien (par défaut)
CancelText	Texte à afficher pour le bouton d'annulation de la modification
CancelImageUrl	URL d'une image à utiliser si ce bouton est une image
ShowCancelButton	True/false si il faut afficher ou non ce bouton
DeleteText	Texte à afficher pour le bouton de suppression
DeleteImageUrl	URL d'une image si ce bouton est une image
ShowDeleteButton	Affiche ou non ce bouton
EditText	Texte à afficher pour le bouton d'édition
EditImageUrl	URL d'une image a afficher si ce bouton est une image
ShowEditButton	Affiche ou non ce bouton
UpdateText	Texte a afficher pour le bouton de validation de la modification
UpdateImageUrl	URL d'une image si ce bouton est une image

SelectText	Texte a afficher pour le bouton de sélection de la ligne
SelectImageUrl	URL d'une image si ce bouton est une image
ShowSelectButton	Affiche ou non ce bouton
CausesValidation	True / false : active / désactive la validation quand les champs sont édités
ValidationGroup	Groupe de validation

10.8.13 L' HyperLinkField

L'hyperlinkField est utilise pour créer un lien vers une autre page. Nous allons profiter de cet exemple pour voir une façon intéressante de paramétrer une commande select pour un grid view.

Nous allons créer un gridview avec nos clients (nous allons a peine modifier le gridview existant) mais nous ajouterons un HyperlinkField sur chaque ligne qui nous enverra sur une autre page. Cette autre page affichera un autre grid view avec la liste des commandes passées par le client sélectionné.



Sur chaque ligne nous avons un lien, chaque lien pointe sur l'url :

`Commandes.aspx?id=…  (l'identifiant client pour la ligne).`

Voici les colonnes de notre GridView :

```
<Columns>
```

```
  <asp:HyperLinkField HeaderText="Commandes" Text="Commandes"
DataNavigateUrlFields="CustomerId" DataNavigateUrlFormatString="Commandes.aspx?id={0}"
/>
  <asp:BoundField DataField="FirstName" HeaderText="Prenom" />
  <asp:BoundField DataField="LastName" HeaderText="Nom" />
</Columns>
```

Je passe sur les deux dernières colonnes, ce sont les mêmes que précédemment. Notre première colonne est intéressante.

La propriété **Text** de l'HyperLinkField permet de définir un texte qui sera affiche par défaut. Si au lieu d'utiliser un texte figé, vous voulez afficher le contenu d'un champ, vous utiliserez la propriété **DataTextField**. La propriété **DataNavigateUrlFields** vous permet d'indiquer une liste de champs de la table (sépares par des virgules). Les valeurs de ces champs vont être passés en paramètres a l'url pointée par le lien.

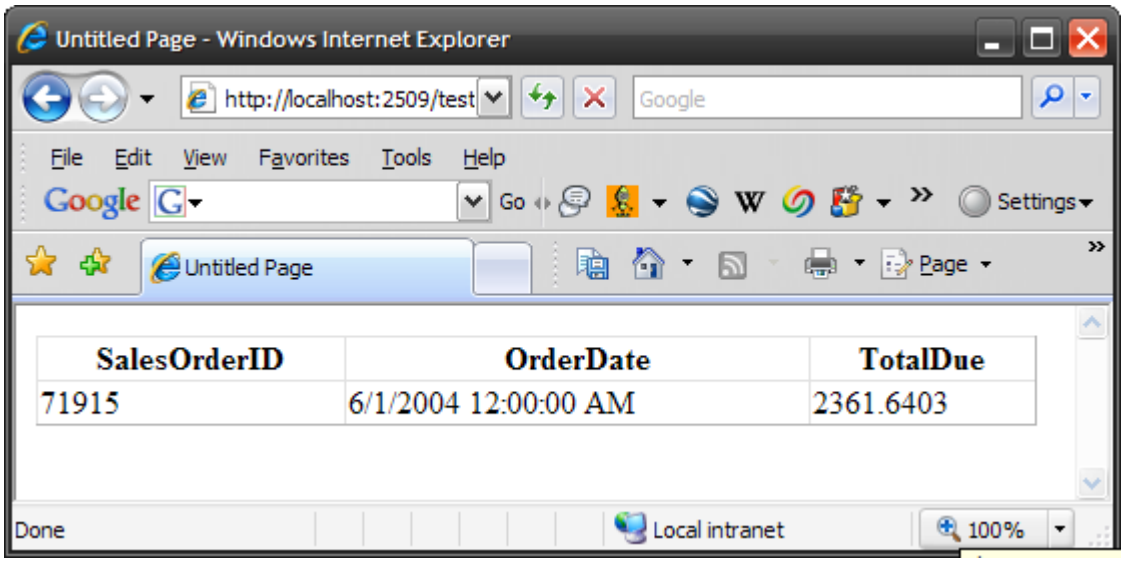
Enfin la propriété DataNavigateUrlFormatString indique l'url a appeler et l'endroit ou on place les paramètres :

```
Commandes.aspx?id={0}
```

Lorsque l'url du lien sera générée pour chaque ligne, la chaine "{0}" sera remplacée par la valeur du premier champ défini dans la propriété DataNavigateUrlFields, la chaine "{1}" sera remplacée par la valeur du deuxième champ, etc…

Le lien pourrait également être un lien fixe vers une page, dans ce cas, soit on ne passe aucun paramètre, soit on utilise la propriété "**NavigateUrl**". De même, si vous voulez ouvrir le lien dans une autre page (ou frame), vous pouvez utiliser la propriété "**Target**".

Dans notre exemple, j'ai clique sur le 6eme lien (Rosmarie Carroll), la page suivante s'affiche :



Que contient cette page ?

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server"></script>

<html xmlns="http://www.w3.org/1999/xhtml">

<body>
    <form id="form1" runat="server">
    <asp:GridView ID="gdCommandes" runat="server"
     DataSourceID="datasourceCommandes"
     Width="500" >
    </asp:GridView>

   <asp:SqlDataSource id="dataSourceCommandes"
    SelectCommand="SELECT SalesOrderID, OrderDate, TotalDue FROM
    SalesLT.SalesOrderHeader
    where CustomerID = @customerId "
    ConnectionString="Data Source=octopus;Initial
    Catalog=AdventureWorksLT; User Id=sa;
    Password=password;" Runat="server">
    <SelectParameters>
        <asp:QueryStringParameter  Name="customerId" QueryStringField="id" />
    </SelectParameters>
    </asp:SqlDataSource>

    </form>
</body>
</html>
```

Elle contient un Gridview (sans aucun paramétrage) qui utilise une datasource nomme "datasourceCommandes".

Tout est dans la dataSource. La commande select contient un paramètre :

```
SelectCommand="SELECT SalesOrderID, OrderDate, TotalDue FROM SalesLT.SalesOrderHeader
where CustomerID = @customerId "
```

Le paramètre est @customerId.

Ou ce paramètre va-t-il être initialisé ? On pourrait très bien partir dans une usine a gaz qui irait récupérer le paramètre dans la querystring, puis qui ferait une requête, puis binderait le résultat de la requête sur le gridview. Le tout à la main évidemment. Sauf qu'on est au XXIe siècle, donc on évite le code.

Il suffit de déclarer un nouveau type de paramètre :

```
<SelectParameters>
<asp:QueryStringParameter  Name="customerId" QueryStringField="id" />
</SelectParameters>
```

On indique ici que le paramètre nomme customerId est un paramètre de type QueryStringParameter, ce qui signifie que la valeur de ce paramètre se trouve dans la querystring, et où se trouve-t-il dans la query string ? Dans le paramètre défini par la propriété "queryStringField".

Pour résumer, au moment du binding, la commande select du gridview va être appelée, le Framework voit qu'il y a un paramètre nomme @customerId. Il va chercher dans les SelectParameters un paramètre dont le nom est "customerId". Il va le trouver dans un QueryStringParameter, ce qui signifie que la valeur de ce paramètre est dans la query string, plus précisément dans le champ "id" de la query String.

Et qu'avons-nous dans le paramètre id de la querystring ? le n. du client qui avait été génère par le HyperLinkField dans notre page precedente.

10.8.14 L' ImageField

L'image Field va nous permettre d'afficher une image en fonction du contenu d'un champ de la table. Ce n'est pas un contrôle qui permet d'afficher une image elle-même stockée dans la base de données. La base de données pourra par exemple contenir un chemin, et le contrôle ImageField servira a afficher une image a partir de ce chemin.

Les principales propriétés de ce contrôle sont les suivantes :

AlternateText	Texte à afficher dans la propriété "Alt" de l'image
DataAlternateTextField	Indique le champ de la commande Select qui contient le contenu de la propriété alt de l'image
DataAlternateTextFormatString	Chaine de formatage du texte a afficher dans la propriété alt.
DataImageUrlField	Indique le champ de la commande Select qui contient le chemin de l'image
DataImageUrlFormatString	Chaine de formatage de l'url de l'image
NullImageUrl	Chemin de l'image a afficher dans le cas ou le champ contenant le chemin est null.

Petite précision au sujet des chaines de formatage. Supposons que le champ "image" de la base de données contienne le nom d'une image a afficher (et seulement son nom, pas son chemin), et que les images se trouvent dans "dossier/images". Nous allons paramétrer le contrôle ainsi :

```
<asp:ImageField
      DataImageUrlField = "image"
      DataImageUrlFormatString = "dossier/images/{0}"
/>
```

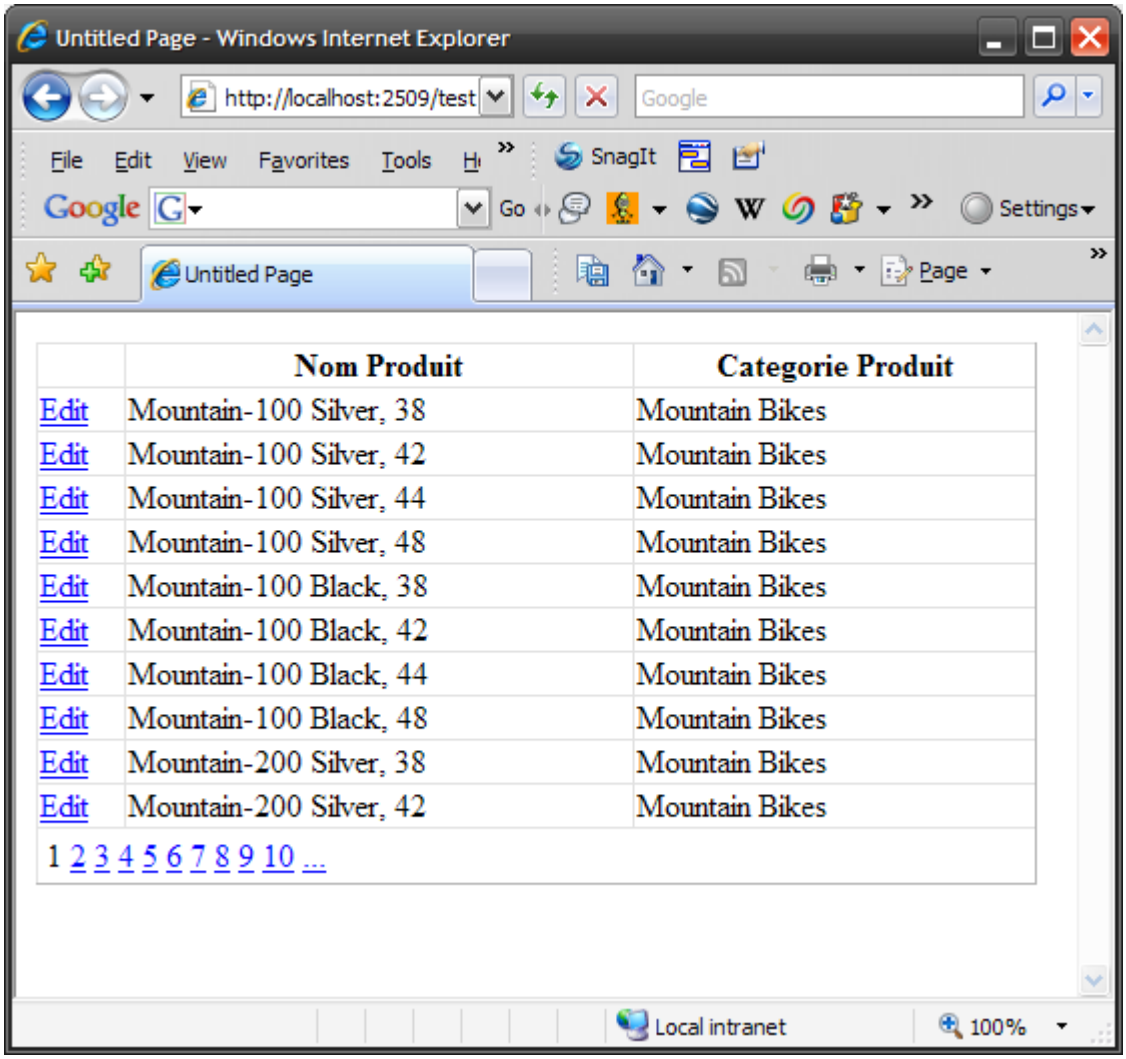
La chaine "{0}" sera remplacée par le contenu du champ "image". L'url de l'image "img.gif" sera donc "dossier/images/img.gif".

Il en est de même pour le contenu de la propriété alt de l'image.

10.8.15 Le TemplateField

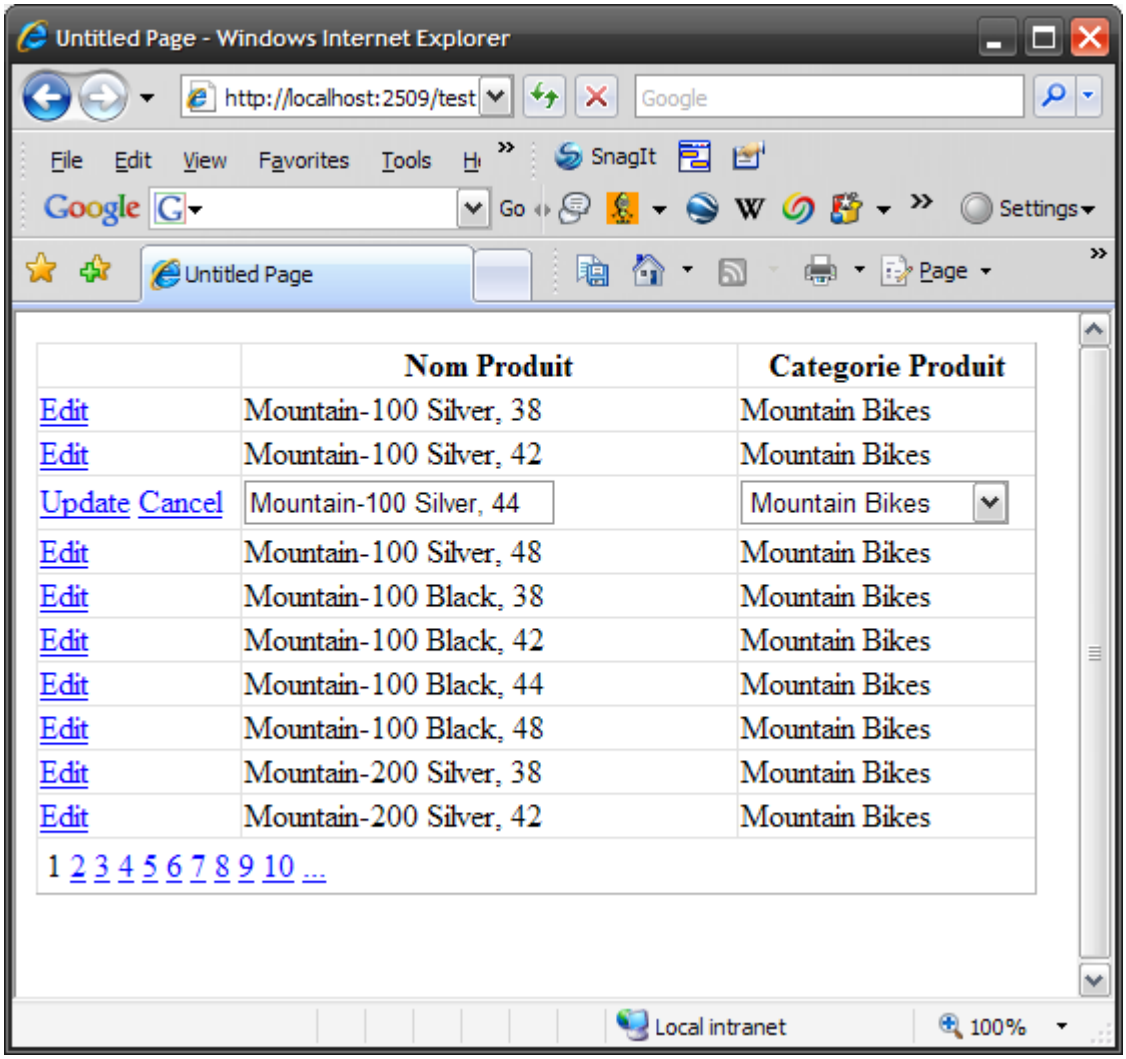
Le TemplateField vous permet d'ajouter une colonne dans le gridview que vous pouvez personnaliser entièrement. Les TemplateFields sont particulièrement utiles pour éditer des enregistrements dans la base de données en y ajoutant par exemple des validateurs.

Dans notre exemple, nous allons utiliser la table "**Product**". Lors de l'édition, nous obligerons l'utilisateur à entrer obligatoirement un nom pour le produit.
Vous noterez que cette table a un champ "ProductCategoryID". C'est l'identifiant de la catégorie du produit. Plutôt que d'obliger l'utilisateur qui modifie un produit à entrer le numéro de la catégorie, il serait plus agréable de le laisser choisir dans une liste déroulante la catégorie qui l'intéresse. Le template Field va nous le permettre.

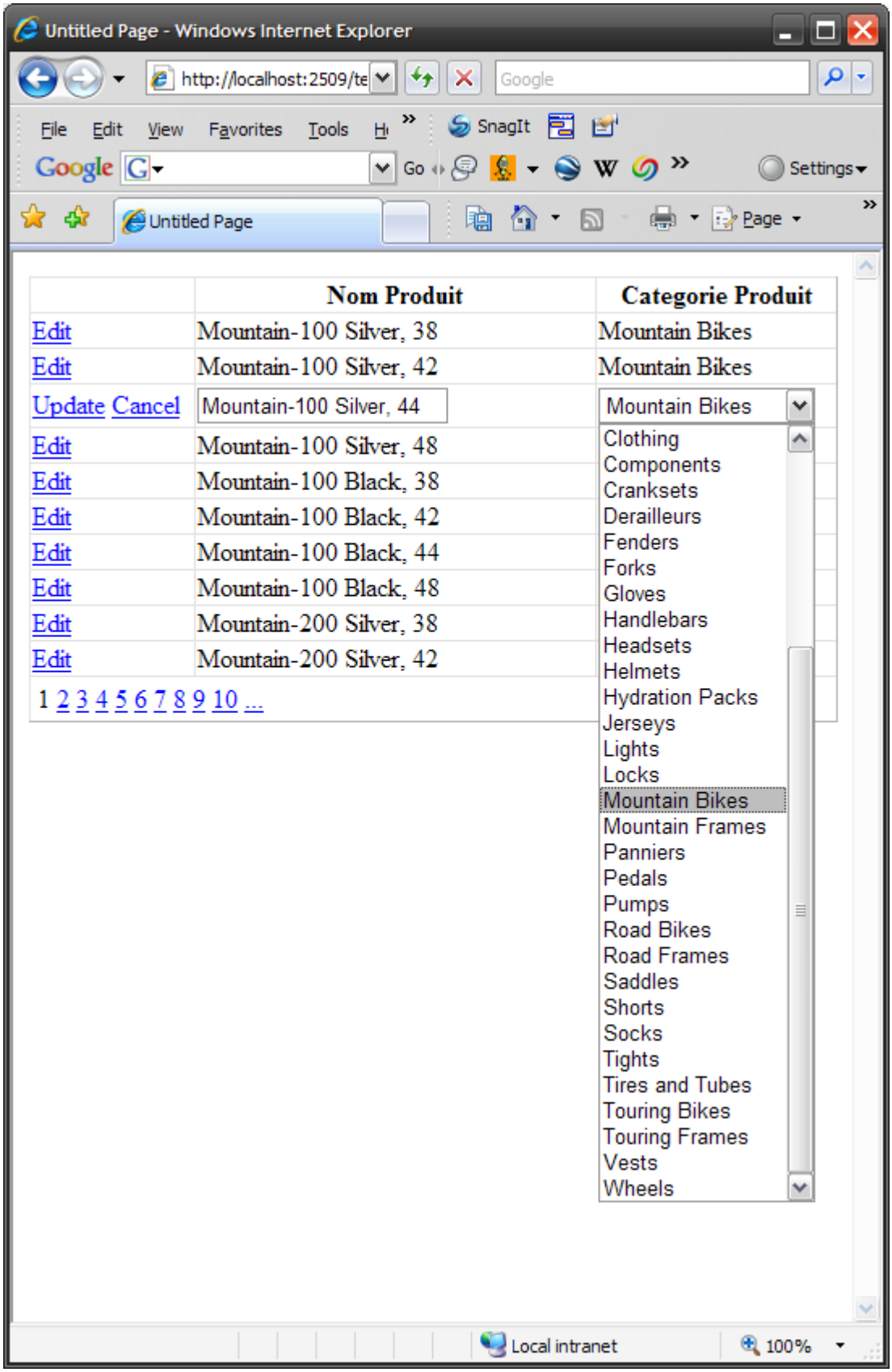


Par défaut, la grid view affiche la liste des produits : le produit, suivi de sa catégorie.

Si on édite une ligne :



Le nom du produit est éditable, la catégorie peut être modifiée, vous noterez qu'on peut la choisir dans une liste déroulante :



Vous noterez également que la ligne sélectionnée est justement la categorie du produit.

Si vous choisissez une autre categorie (dérailleurs par exemple) et que vous cliquez sur "Update" :

Untitled Page - Windows Internet Explorer

http://localhost:2509/test1/De

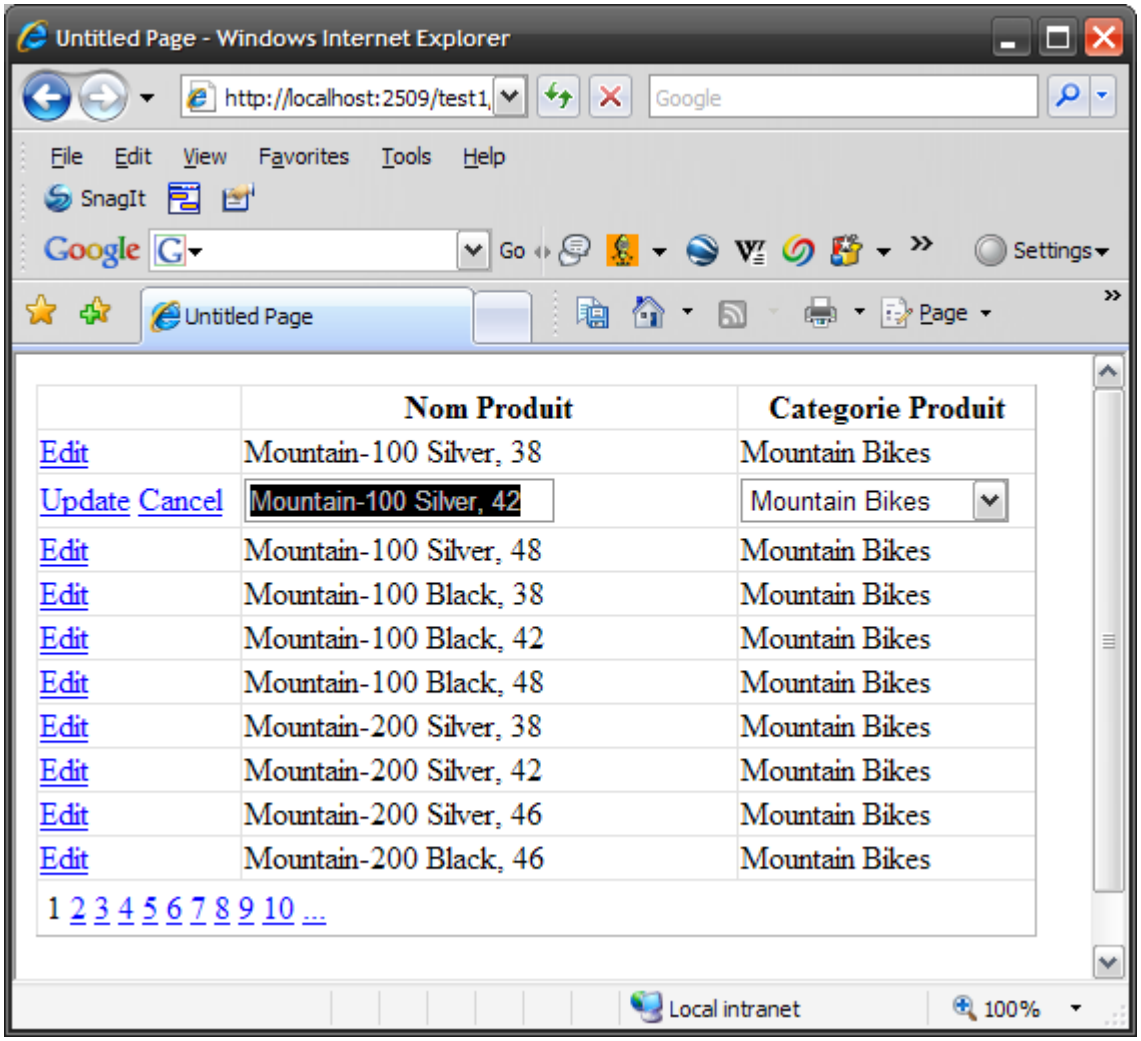
File Edit View Favorites Tools Help

	Nom Produit	Categorie Produit
Edit	HL Crankset	Cranksets
Edit	Front Derailleur	Derailleurs
Edit	Rear Derailleur	Derailleurs
Edit	Mountain-100 Black, 44	Derailleurs
Edit	LL Fork	Forks
Edit	ML Fork	Forks
Edit	HL Fork	Forks
Edit	LL Headset	Headsets
Edit	ML Headset	Headsets
Edit	HL Headset	Headsets

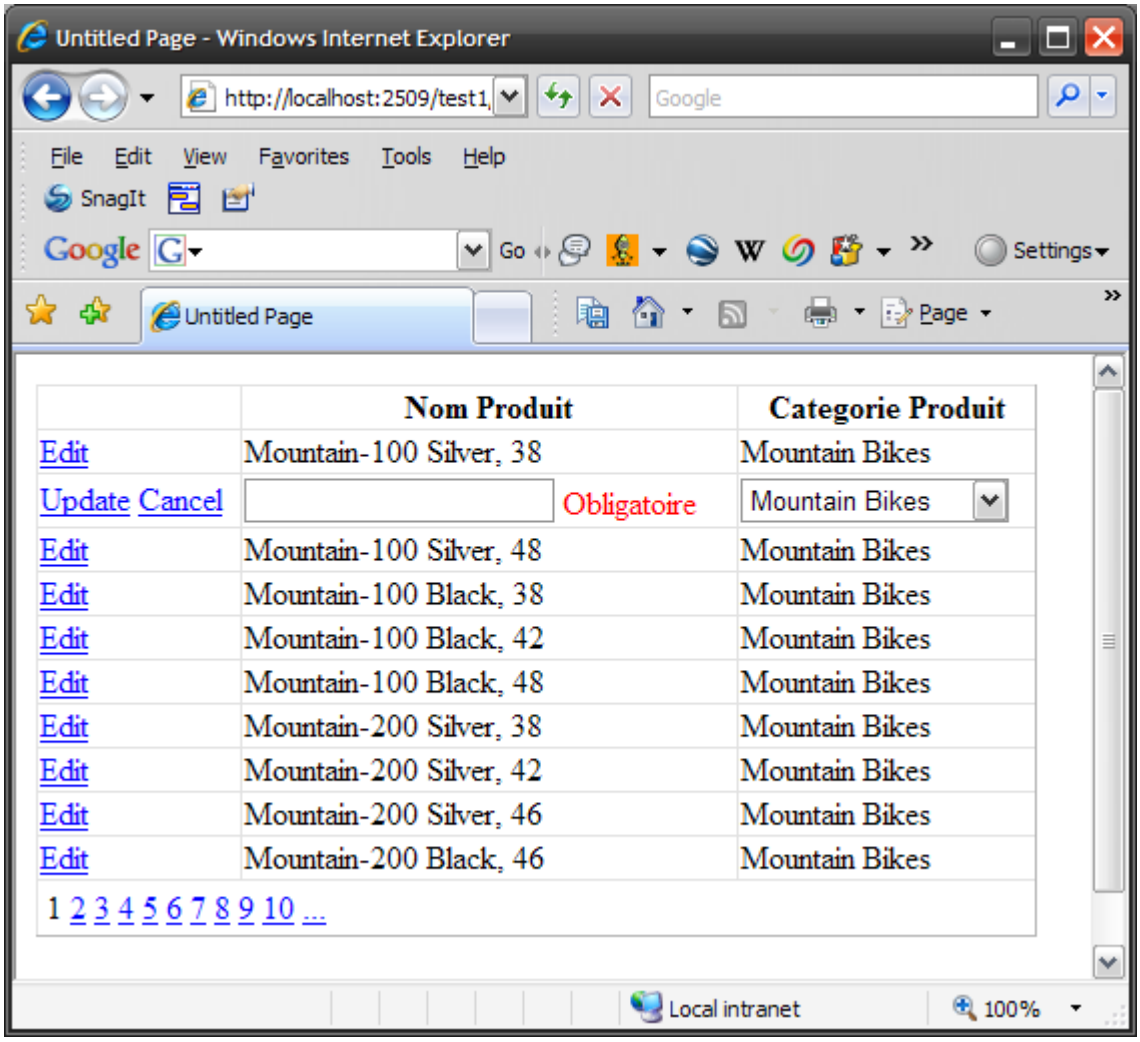
... 11 12 13 14 15 16 17 18 19 20 ...

Local intranet 100%

De même, si on modifie le nom d'un produit :



Qu'on efface tout le champ et qu'on clique sur "Update"



Il est donc impossible de valider une ligne sans remplir le TextBox.

Comment a-t-on procédé ?

Notre GridView a quelque peu change :

```
<asp:GridView AllowPaging="true" AllowSorting="true" ID="gdClients" runat="server"
DataSourceID="datasourceProduits" DataKeyNames="ProductID" AutoGenerateColumns="false"
AutoGenerateEditButton="true"
Width="500" >
<Columns>

<asp:TemplateField HeaderText="Nom Produit">
    <ItemTemplate>
        <%# Eval("nom_produit") %>
    </ItemTemplate>
    <EditItemTemplate>
    <asp:TextBox Text='<%# Bind("nom_produit") %>' id="nomProduit"
     runat="server" />
    <asp:RequiredFieldValidator ControlToValidate="nomProduit"
     Text="Obligatoire"
     ID="valNom" runat="server" />
    </EditItemTemplate>
 </asp:TemplateField>

<asp:TemplateField HeaderText="Categorie Produit">
  <ItemTemplate>
      <%# Eval("nom_categorie") %>
  </ItemTemplate>
  <EditItemTemplate>
     <asp:DropDownList ID="liste_categories"
      DataSourceID="datasourceCategories"
      DataTextField="Name" DataValueField="ProductCategoryID"
      SelectedValue='<%# Bind("ProductCategoryID") %>' runat="server" />
  </EditItemTemplate>
</asp:TemplateField>

</Columns>
</asp:GridView>
```

Nos colonnes sont composées de deux TemplateFields. Un pour le nom du produit, un pour la categorie. Examinons le premier :

Il est compose de deux "sous-templates" : un utilise pour l'affichage,

Dans celui-ci on affiche le nom du produit, en utilisant la fonction Eval que nous avons déjà rencontrée qui va afficher, pour l'enregistrement courant le champ indique en paramètre.

```
<ItemTemplate>
    <%# Eval("nom_produit") %>
</ItemTemplate>
```

L'autre pour l'édition :

```
<EditItemTemplate>
    <asp:TextBox Text='<%# Bind("nom_produit") %>' id="nomProduit"
    runat="server" />
    <asp:RequiredFieldValidator ControlToValidate="nomProduit" \
    Text="Obligatoire"
    ID="valNom" runat="server" />
    </EditItemTemplate>
</asp:TemplateField>
```

Dans celui-ci, nous avons deux composants ASP.NET que nous connaissons : un TextBox et un validateur, je ne reviens pas sur l'utilisation de ces deux contrôles. Notez comment la propriété "**Text**" du TextBox est initialisée. Elle utilise la fonction Bind.

Petite précision. Vous ne le voyez pas mais lors du data binding, les données sont lues dans la base, puis renseignent une table en mémoire, cette table est composée de lignes, le contenu de chaque ligne est utilisé ensuite pour renseigner le DataGrid. En appelant la fonction Eval("champ"), le champ de la ligne est affiché. En appelant la fonction Bind("champ"), le champ est affiché, mais, si vous modifiez la propriété pour laquelle la fonction Bind est appelée, ce que vous avez modifié sera répercuté dans la table en mémoire. Comme c'est cette table qui est utilisée pour remettre a jour la base, ca tombe bien, ce que vous aurez modifié sera répercuté dans la base.

Le deuxième ItemTemplate est plus complexe :

En cas d'édition, rien de particulier, on affiche juste le nom de la catégorie:

```
<ItemTemplate>
    <%# Eval("nom_categorie") %>
</ItemTemplate>
```

En cas de modification, on affiche une DropDownList (que nous avons vu précédemment). La seule astuce est d'utiliser ici la fonction Bind pour la propriété SelectedValue. Elle initialisera la liste en la positionnant sur la bonne catégorie et modifiera la table sous jacente quand une autre valeur sera sélectionnée.

```
<EditItemTemplate>
   <asp:DropDownList ID="liste_categories" DataSourceID="datasourceCategories"
```

```
   DataTextField="Name" DataValueField="ProductCategoryID"
   SelectedValue='<%# Bind("ProductCategoryID") %>' runat="server" />
</EditItemTemplate>
```

Enfin les deux DataSources :

Cette première datasource est utilisée pour le DataGrid

```
<asp:SqlDataSource id="datasourceProduits"

SelectCommand="SELECT SalesLT.Product.Name as nom_produit, ProductID,
SalesLT.Product.ProductCategoryID , SalesLT.ProductCategory.Name as nom_categorie FROM
SalesLT.Product, SalesLT.ProductCategory where
SalesLT.ProductCategory.ProductCategoryID = SalesLT.Product.ProductCategoryID "

UpdateCommand="UPDATE SalesLT.Product SET Name=@nom_produit,
ProductCategoryID=@ProductCategoryID WHERE ProductID=@ProductID"

ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User
Id=sa;Password=password;"
Runat="server">

</asp:SqlDataSource>
```

Celle-ci sert a renseigner la DropDownList.

```
<asp:SqlDataSource id="datasourceCategories"
SelectCommand="SELECT ProductCategoryID, Name FROM SalesLT.ProductCategory order by
Name"

ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User
Id=sa;Password=password;"
Runat="server">
</asp:SqlDataSource>
```

10.8.16 Les autres templates

En plus des templates que nous avons vu, citons deux autres templates qui fonctionnent de la même façon (c'est-à-dire pour chaque colonne) :

Le **HeaderTemplate** permet d'afficher des informations dans l'entête de la colonne, et le **FooterTemplate** permet d'afficher des informations dans le pied de la colonne.

10.8.17 Les évènements du GridView

Le GridView est le plus riche des contrôles disponible par défaut dans ASP.NET. Il possède un grand nombre d'évènements auxquels vous pouvez attacher une fonction.

Ces évènements se déclenchent durant la création du DataGrid

DataBinding	Déclenché immédiatement avant que les données soient bindees au DataGrid
DataBound	Déclenché immédiatement après que les données aient été bindees au DataGrid
RowCreated	Déclenché après que chaque ligne ait été créée
RowDataBound	Déclenché après que chaque ligne ait été bindee a sa source de données

Ces évènements se déclenchent lors de l'édition d'une ligne

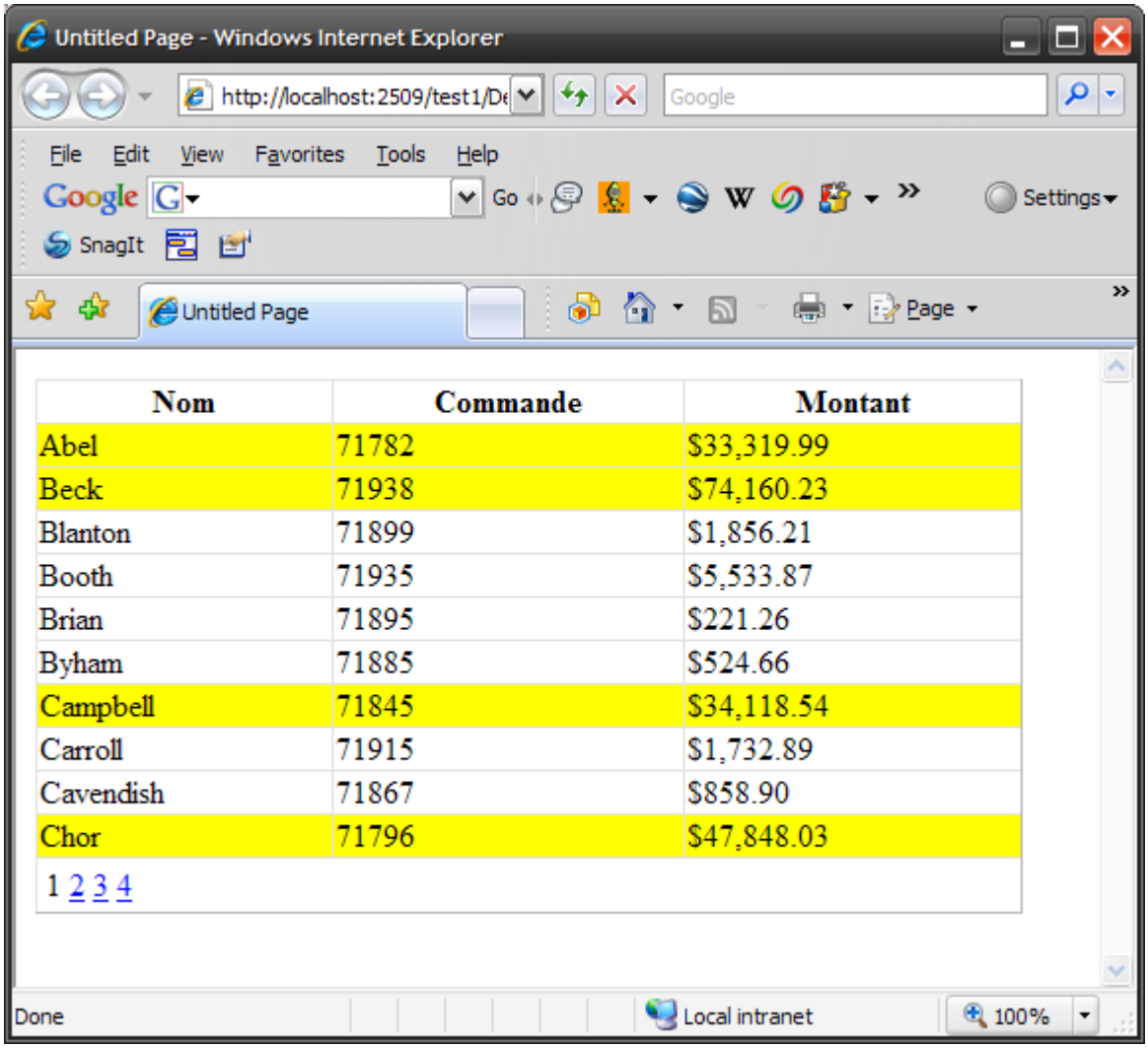
RowCommamd	Déclenché quand un bouton d'édition est cliqué
RowUpdating	Déclenché juste avant que la ligne soit mise a jour dans la base
RowUpdated	Déclenché après que la ligne ait été mise à jour dans la base
RowDeleting	Déclenché avant que la ligne ne soit supprimée de la base
RowDeleted	Déclenché après que la ligne ait été supprimée de la base
RowCancelingEdit	Déclenché après avoir clique sur le bouton "Annuler" lors de l'édition

Enfin, une série d'évènements sont déclenchés durant l'utilisation du DataGrid

PageIndexChanging	Déclenché juste avant que la page ait été changée
PageIndexChanged	Déclenché juste après que la page ait été changée
Sorting	Déclenché juste avant un tri
Sorted	Déclenché juste après un tri
SelectedIndexChanging	Déclenché juste avant qu'une ligne soit sélectionnée
SelectedIndexChanged	Déclenché juste après qu'une ligne ait été sélectionnée

Comment utiliser ces évènements ?

Imaginons un gridView dans lequel nous allons afficher toutes les commandes passées par client et dans lequel nous allons "surligner" les commandes dont le montant dépasse 10000.



Nom	Commande	Montant
Abel	71782	$33,319.99
Beck	71938	$74,160.23
Blanton	71899	$1,856.21
Booth	71935	$5,533.87
Brian	71895	$221.26
Byham	71885	$524.66
Campbell	71845	$34,118.54
Carroll	71915	$1,732.89
Cavendish	71867	$858.90
Chor	71796	$47,848.03
1 2 3 4		

Comment avons-nous procédé ?

Très simplement :

Notre DataGrid est très classique :

```
<asp:GridView AllowPaging="true" AllowSorting="true" ID="gdCommandes" runat="server"
 DataSourceID="datasourceProduits" AutoGenerateColumns="false"
OnRowDataBound="surligne"
 Width="500" >
    <Columns>
      <asp:BoundField DataField="LastName" HeaderText="Nom" />
      <asp:BoundField DataField="SalesOrderID" HeaderText="Commande" />
      <asp:BoundField DataField="total" HeaderText="Montant" DataFormatString="{0:c}"
/>
    </Columns>
</asp:GridView>
```

3 colonnes : une bindée sur le champ LastName de la commande Select, une sur le numéro de commande et une sur la colonne "Total". Vous noterez au passage que le format de la colonne total est "{0:c}", ce qui signifie afficher le contenu au format monétaire (en l'occurrence au format monétaire du serveur, qui, dans notre exemple est un Windows xp version US. Si il tournait sur une version française, nous aurions eu pour la première ligne : 33 319,99 euros).

La data source est très classique aussi :

```
<asp:SqlDataSource id="datasourceProduits"
SelectCommand=" SELECT  SalesLT.Customer.LastName, detail.SalesOrderID,
SUM(detail.LineTotal) AS total
FROM    SalesLT.Customer
INNER JOIN SalesLT.SalesOrderDetail AS detail
INNER JOIN SalesLT.SalesOrderHeader AS header ON detail.SalesOrderID =
header.SalesOrderID ON SalesLT.Customer.CustomerID = header.CustomerID
GROUP BY detail.SalesOrderID, SalesLT.Customer.LastName
ORDER BY SalesLT.Customer.LastName"
ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User Id=sa;
Password=password;"
Runat="server">
</asp:SqlDataSource>
```

Reste juste une chose dont je n'avais pas parle dans le DataGrid : la propriété `OnRowDataBound="surligne",` elle signifie que lorsque l'évènement RowDataBound se déclenchera (je vous rappelle que cet évènement se d'éclanche juste avant que les donnes de la ligne soient bindées), on va appeler la fonction "surligne".

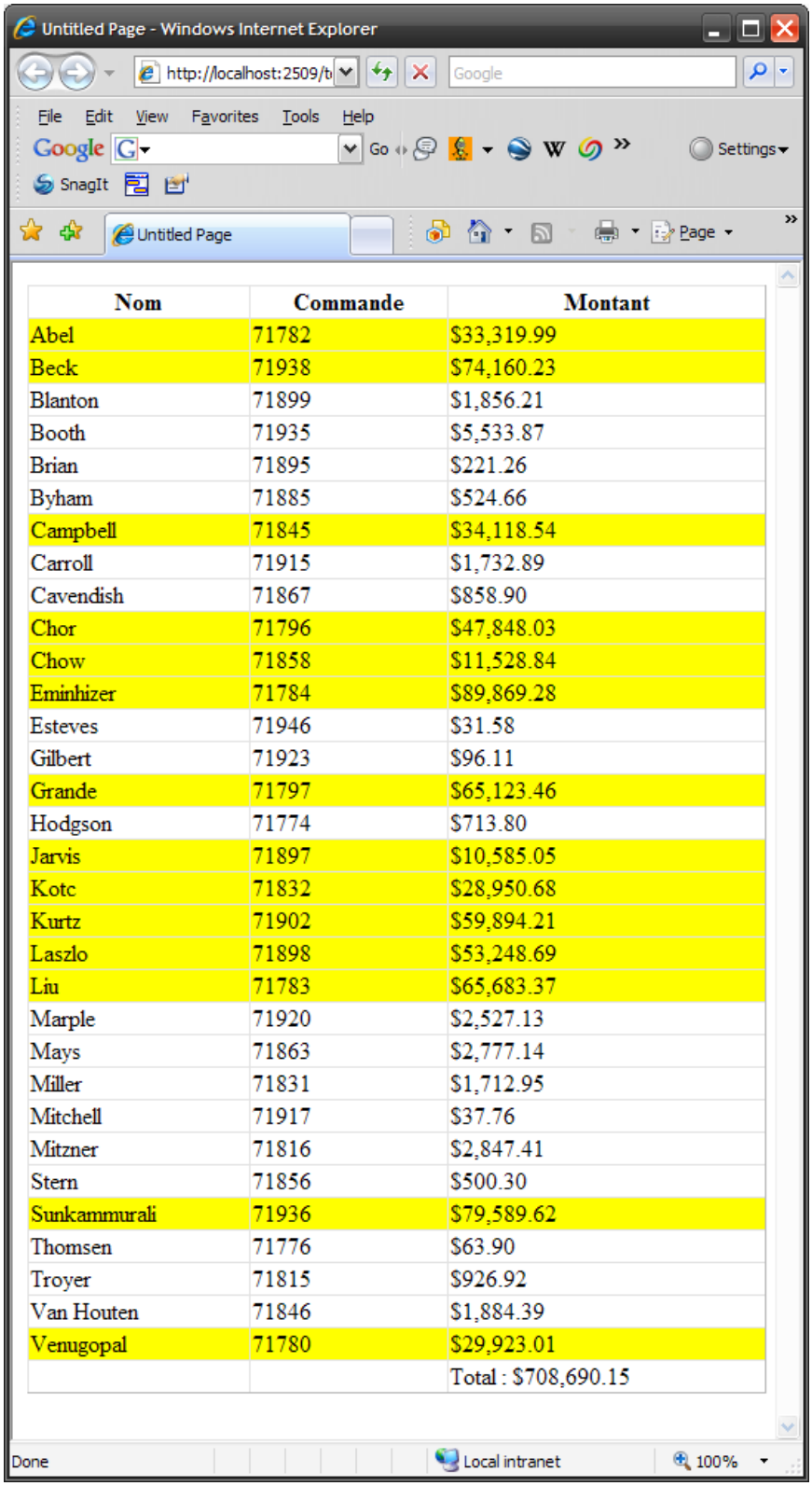
Cette fonction est définie ainsi :

```
<script runat="server">
```

```
protected void surligne(object sender, GridViewRowEventArgs e)
{
  if  (e.Row.RowType == DataControlRowType.DataRow)
  {
    decimal montant = (decimal)DataBinder.Eval(e.Row.DataItem,"total");
    if (montant > 10000) e.Row.BackColor = System.Drawing.Color.Yellow;
  }
}
</script>
```

Que fait-elle ? Si le type de la ligne bindee est une ligne du tableau (ce peut aussi être une ligne d'entête ou du pager), on va récupérer dans "montant" la valeur du champ "total" de la table, puis, si il est supérieur a une valeur fixée, on change la couleur du fond de la ligne. Simple non ?

Nous allons conserver notre DataGrid et lui ajouter un pied, dans la colonne Total, ce pied affichera le montant total de toutes les commandes :



Nom	Commande	Montant
Abel	71782	$33,319.99
Beck	71938	$74,160.23
Blanton	71899	$1,856.21
Booth	71935	$5,533.87
Brian	71895	$221.26
Byham	71885	$524.66
Campbell	71845	$34,118.54
Carroll	71915	$1,732.89
Cavendish	71867	$858.90
Chor	71796	$47,848.03
Chow	71858	$11,528.84
Eminhizer	71784	$89,869.28
Esteves	71946	$31.58
Gilbert	71923	$96.11
Grande	71797	$65,123.46
Hodgson	71774	$713.80
Jarvis	71897	$10,585.05
Kotc	71832	$28,950.68
Kurtz	71902	$59,894.21
Laszlo	71898	$53,248.69
Liu	71783	$65,683.37
Marple	71920	$2,527.13
Mays	71863	$2,777.14
Miller	71831	$1,712.95
Mitchell	71917	$37.76
Mitzner	71816	$2,847.41
Stern	71856	$500.30
Sunkammurali	71936	$79,589.62
Thomsen	71776	$63.90
Troyer	71815	$926.92
Van Houten	71846	$1,884.39
Venugopal	71780	$29,923.01
		Total : $708,690.15

Je n'ai pas modifié la datasource. Le GridView change quelque peu :

```
<asp:GridView  AllowSorting="true" ID="gdCommandes" runat="server"
DataSourceID="datasourceProduits" AutoGenerateColumns="false" OnRowDataBound="surligne"
ShowFooter="true" Width="500" >
```

```
  <Columns>
     <asp:BoundField DataField="LastName" HeaderText="Nom" />
     <asp:BoundField DataField="SalesOrderID" HeaderText="Commande" />
     <asp:TemplateField HeaderText="Montant">
         <ItemTemplate>
              <%# Eval("total", "{0:c}") %>
         </ItemTemplate>
         <FooterTemplate>
              <asp:Label ID="lbTotal" runat="server" ></asp:Label>
         </FooterTemplate>
     </asp:TemplateField>
 </Columns>
</asp:GridView>
```

Notez que la colonne "Montant" a été modifiée. Ce n'est plus un BoundField, car le footer (pied) n'est supporté que pour les TemplateField. J'ai donc créé un template field qui affiche le contenu du champ "total" et ajouté un FooterTemplate qui contient un contrôle Label. C'est dans ce contrôle que nous allons afficher la somme totale.

Notez que la propriété "**ShowFooter**" est mise a "true". Par défaut, le gridview n'affiche pas les footers.

La fonction surligne a été améliorée :

```
decimal total = 0;

protected void surligne(object sender, GridViewRowEventArgs e)
{
 if  (e.Row.RowType == DataControlRowType.DataRow)
 {
    decimal montant = (decimal)DataBinder.Eval(e.Row.DataItem,"total");
    if (montant > 10000) e.Row.BackColor = System.Drawing.Color.Yellow;
    total += montant;
 }

 if  (e.Row.RowType == DataControlRowType.Footer)
 {
    Label lb = (Label)e.Row.FindControl("lbTotal");
    lb.Text = String.Format( "Total : {0:c}", total );
 }
}
```

Nous avons une variable "total" initialisée a 0. A chaque fois qu'une ligne est bindée, notre fonction est appelée. Si c'est une ligne de données (DataRow), on ajoute le valeur du champ "total" à notre variable et si c'est une ligne de footer (Footer), on va chercher notre label, et on initialise le texte du label avec notre variable "total" (formatée au format monétaire).

Vous noterez que j'ai supprimé la pagination pour notre exemple, car chaque page ayant un footer, un total incorrect s'afficherait sur chaque page (la somme des commandes de toutes les pages précédentes).

10.9 Les contrôles DetailView et FormView

Modifier les données directement dans une liste comme nous l'avons fait avec la DataList et le GridView peut être très pratique, mais on peut vouloir travailler avec un seul enregistrement a la fois. C'est à ça que servent les contrôles DetailView et FormView. Ces deux types de contrôles font la même chose : ils vous permettent d'afficher, d'éditer, d'insérer ou de supprimer un enregistrement dans une table. Ils vous permettent également de vous déplacer en avant ou en arrière dans une liste d'enregistrements.

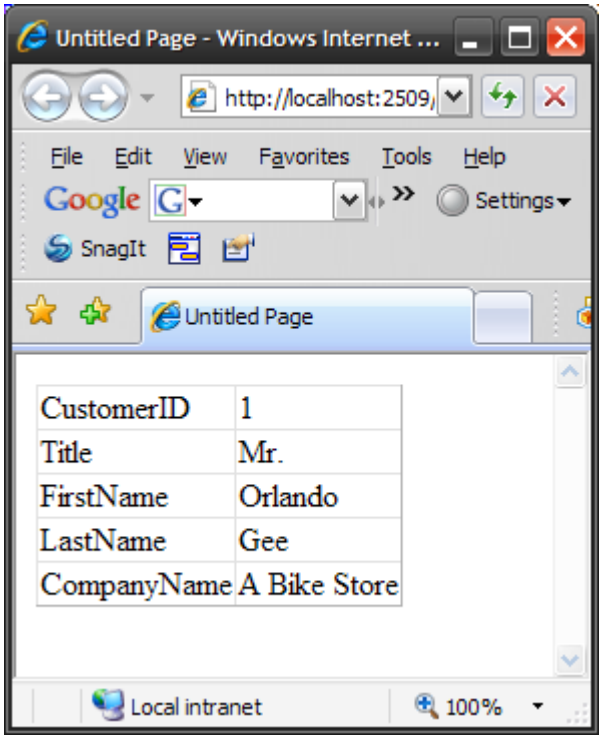
La différence entre ces deux contrôles réside dans la façon dont les données sont affichées. Le contrôle DetailView affiche toujours chaque champ dans une ligne d'une table HTML, le contrôle FormView utilise des templates et vous permet de personnaliser l'affichage génère par le contrôle.

10.10 Le contrôle DetailView

10.10.1 Afficher des donnes

Commençons par le plus simple, affichons des données avec ce contrôle. Tout comme le repeater ou le DataList, le DetailView est un contrôle qui supporte le data binding. Ce databinding peut être fait soit sous forme déclarative dans la page ASP.NET, soit par programme.

Nous allons afficher les données de la table Customer de notre base.



Le code associé est on ne peut plus simple :

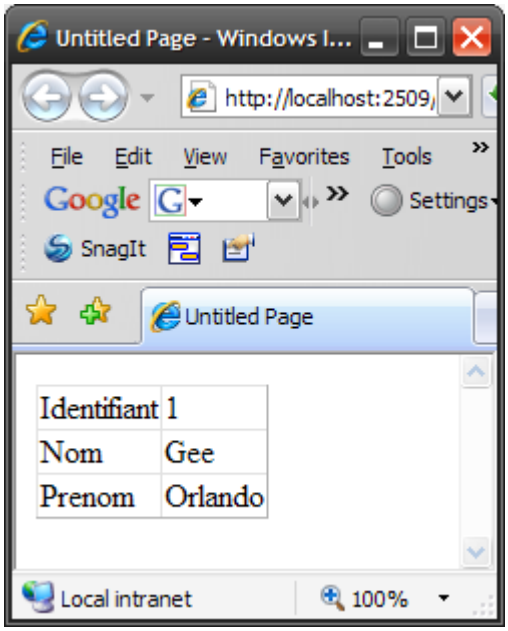
```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<body>
<form id="form1" runat="server">
<asp:DetailsView ID="DetailsView1" runat="server" DataSourceID="datasource" />

<asp:SqlDataSource id="datasource"
SelectCommand=" SELECT CustomerID, Title, FirstName, LastName, CompanyName FROM
SalesLT.Customer" ConnectionString="Data Source=octopus;Initial
Catalog=AdventureWorksLT;UserId=sa; Password=password;"
Runat="server">
</asp:SqlDataSource>
</form>
</body>
</html>
```

Nous retrouvons notre data source (nous avons modifie bien sur la commande Select) et un contrôle DetailView dans lequel on n'initialise aucune propriété en particulier, à l'exception bien sûr de la datasource. Le DataView va donc récupérer tous les champs de la commande Select, afficher leur nom dans une première colonne et leur contenus dans une deuxième.

Tout comme avec le GridView, vous pouvez personnaliser le DetailView en utilisant des contrôles spéciaux. Le DetailView supporte les mêmes contrôles que le GridView :

BoundField	Affiche le contenu d'un champ sous forme de texte
CheckBoxField	Affiche le contenu d'un champ sous forme de case a cocher
CommandField	Affiche des boutons pour permettre l'édition, la suppression et la sélection
ButtonField	Affiche le contenu d'un champ comme un bouton (bouton, lien ou image)
HyperLinkField	Affiche le contenu d'un champ sous forme de lien
ImageField	Affiche le contenu d'un champ sous forme d'image
TemplateField	Template pour personnaliser complètement l'affichage du champ

Utilisons donc des BoundFields pour afficher l'identifiant, le nom et le prénom du client. Grace au BoundField, nous pouvons spécifier le contenu de la colonne de gauche.



La déclaration du DetailView reprend des éléments connus :

```
<asp:DetailsView ID="DetailsView1" runat="server" DataSourceID="datasource"
AutoGenerateRows="false" >
    <Fields>
        <asp:BoundField HeaderText="Identifiant" DataField="CustomerID" />
        <asp:BoundField HeaderText="Nom" DataField="LastName" />
        <asp:BoundField HeaderText="Prenom" DataField="FirstName" />
    </Fields>
</asp:DetailsView>
```

On retrouve le même type de déclaration qu'avec le GridView. Sauf qu'au lieu de placer les contrôles dans un tag <**Columns**>, on le met dans un tag <**Fields**>.

Notez que pour utiliser ces contrôles, il convient de mettre la propriété "**AutoGenerateRows**" a "**false**".

10.10.2 Afficher un message si le résultat de la requête ne renvoie aucun résultat

Si la commande Select ne retourne aucun résultat, il y a deux façons d'afficher un message indiquant cet état :

1. En utilisant la propriété EmptyDataText du DetailView.

Vous indiquez ici un texte qui sera affiche à la place du DetailView. Cette chaine de caractères supporte les tags HTML. Vous pouvez donc initialiser cette propriété avec par exemple : "`<span style="color: red, font-weight: bold, font-size:larger">Aucun enregistrement</span>`"

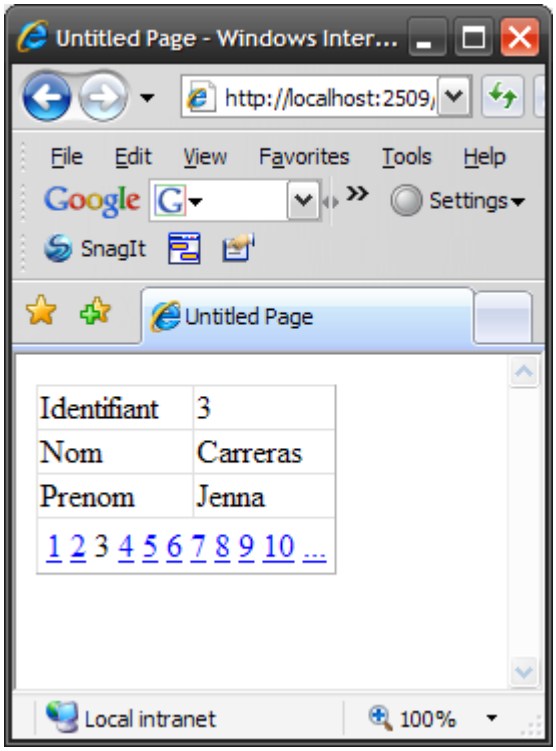
2. En utilisant un EmptyDataTemplate

Le template EmptyDataTemplate vous permet d'afficher des informations plus complexes. Comme il s'agit d'un template, vous pouvez y placer des tags HTML, des contrôles ASP.NET ou du texte brut :

```
<EmptyDataTemplate>
     <div style="Border: solid 2px red; font-size: 20px; font-weight: bold">
      Aucun enregistrement !
     </div>
</EmptyDataTemplate>
```

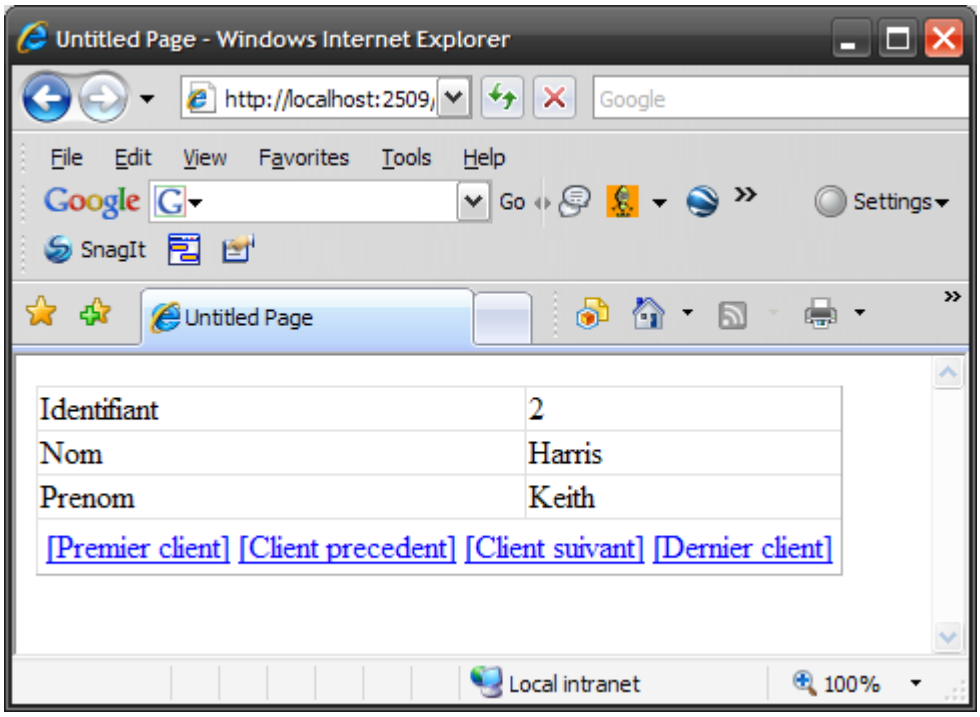
10.10.3 Se déplacer dans les données avec le contrôle DetailView

Tel quel, le contrôle n'affiche que les données du premier enregistrement renvoyé par la commande Select. En initialisant la propriété "**AllowPaging**" a "**true**", le déplacement devient possible dans le résultat de la commande :



Vous remarquerez en bas la liste des enregistrements. Il suffit de se cliquer sur l'un d'entre eux pour y accéder.

On peut bien entendu, personnaliser l'aspect de la barre de pagination :




```
<asp:DetailsView ID="DetailsView1" runat="server" DataSourceID="datasource"
AutoGenerateRows="false" AllowPaging="true" >
    <PagerSettings
        Mode = "NextPreviousFirstLast"
        FirstPageText = "[Premier client]"
        PreviousPageText = "[Client precedent]"
        NextPageText = "[Client suivant]"
        LastPageText = "[Dernier client]"
    />

   <Fields>
     <asp:BoundField HeaderText="Identifiant" DataField="CustomerID" />
     <asp:BoundField HeaderText="Nom" DataField="LastName" />
     <asp:BoundField HeaderText="Prenom" DataField="FirstName" />
  </Fields>
</asp:DetailsView>
```

Le PagerSettings vous semble familier ? Il possède les mêmes propriétés que le pager du GridView. Je vous invite donc a retourner au chapitre traitant du pager pour le GridView si vous voulez en savoir plus.

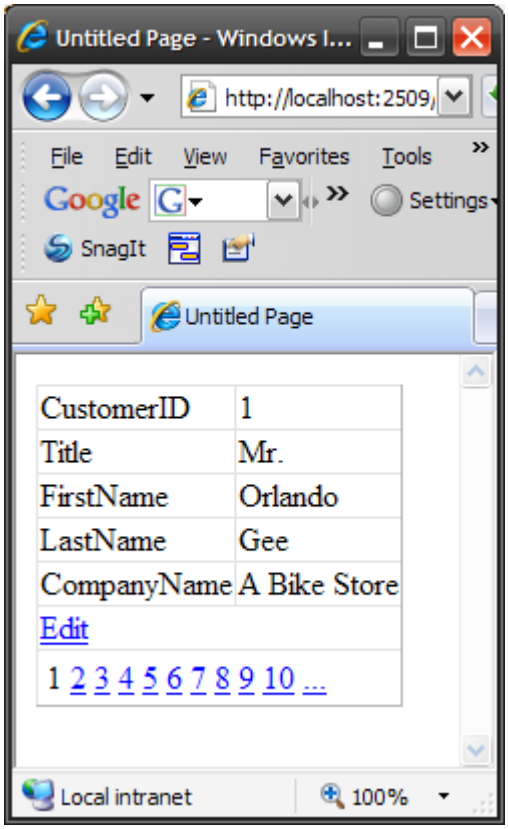
10.10.4 Modifier les données

Afficher les données, c'est bien, pouvoir les modifier, c'est mieux ! Pour pouvoir modifier les données, on retrouve un fonctionnement très similaire à ce que nous avons rencontre avec le GridView.

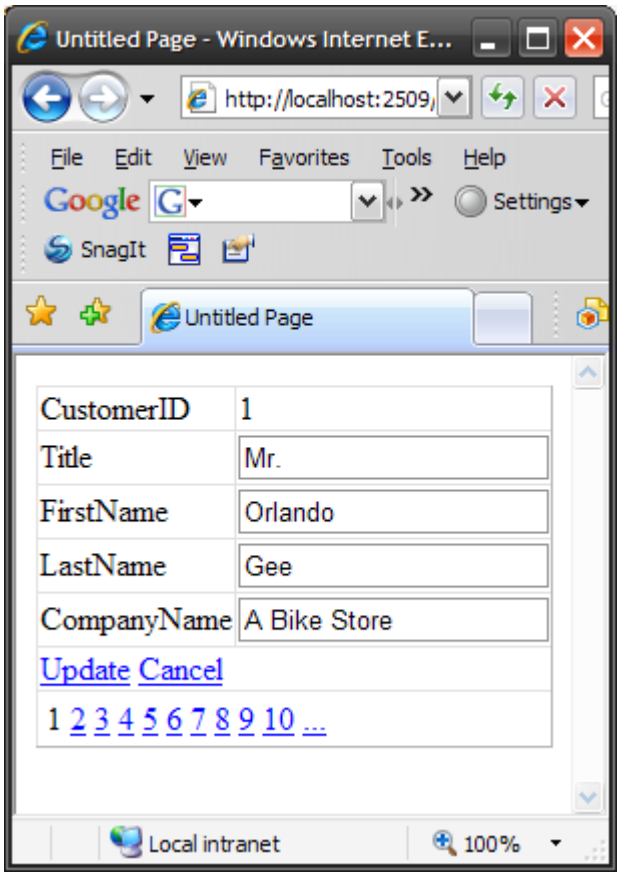
Pour cela remodifions notre GridView :

```
<asp:DetailsView ID="DetailsView1" runat="server" DataSourceID="datasource"
AutoGenerateRows="true" AllowPaging="true"
DataKeyNames="CustomerID" AutoGenerateEditButton="true" >
</asp:DetailsView>
```

Nous avons fait au plus simple : Il suffit d'initialiser la propriété "**AutoGenerateEditButton**" a "**true**" et d'indiquer quel est la clef de l'enregistrement (ici CustomerID) pour que le bouton d'édition
apparaisse :



En cliquant dessus, on retrouve le mode de fonctionnement déjà aborde avec le GridView :



Les champs deviennent accessibles en écriture (les labels sont remplaces par des TextBox) et les boutons "Update" (mise a jour de la base) et "Cancel" (annulation de la modification) apparaissent.

Vous noterez que le champ CustomerID étant la clef de l'enregistrement, il n'est pas accessible en écriture.

Il ne suffit cependant pas de cliquer sur "Update" pour que la modification se fasse. Lorsqu'on clique sur "Update", la commande Update de la datasource est appelée. Il faut donc déclarer une commande Update pour la datasource :

```
<asp:SqlDataSource id="datasource"
SelectCommand=" SELECT CustomerID, Title, FirstName, LastName, CompanyName FROM
SalesLT.Customer"
UpdateCommand=" UPDATE SalesLT.Customer SET LastName=@LastName, FirstName=@FirstName
WHERE CustomerID=@CustomerID"
ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User
Id=sa;Password=password;"
Runat="server">
</asp:SqlDataSource>
```

Et c'est tout.

Notez également que les mécanismes utilises pour l'édition personnalisée du GridView sont également disponibles ici. Vous pouvez donc utiliser un EditItemTemplate pour utiliser des contrôles ASP.NET comme les validateurs ou des listes déroulantes. Je vous incite fortement a les expérimenter ici.

10.10.5 Accès concurrents à la base

Vous pouvez vous demander ce qui se passe si deux personnes accèdent en même temps au même enregistrement. Supposons qu'un utilisateur A édite l'enregistrement et attende une heure avant de mettre cliquer sur "Update", durant ce temps, un utilisateur B édite et met à jour l'enregistrement. Les modifications de B vont être perdues lorsque A cliquera sur "Update".

Ceci est du au fait que, par défaut, la datasource écrase sans aucune vérification les données modifiées, tant pis si elles ont été modifiées entre le moment ou on à commencé l'édition et le moment de la mise à jour. Parfois, ce n'est pas important (seule une personne peut éditer les données), parfois, c'est gênant.

On peut éviter cela en modifiant le datasource de la façon suivante :

```
<asp:SqlDataSource id="datasource"
SelectCommand=" SELECT CustomerID, Title, FirstName, LastName, CompanyName FROM
SalesLT.Customer"
UpdateCommand=" UPDATE SalesLT.Customer SET LastName=@LastName, FirstName=@FirstName
WHERE CustomerID=@CustomerID"
ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User
Id=sa;Password=password;"
ConflictDetection="CompareAllValues"
OnUpdated="miseAjour"
Runat="server">
```

```
</asp:SqlDataSource>
```

Nous avons initialisé la propriété "**ConflictDetection**" avec "**CompareAllValues**". Par défaut, cette propriété est initialisée avec "OverwriteChanges" (écraser les données). CompareAllValues va sauvegarder en mémoire les valeurs renvoyées par la commande Select lors de l'initialisation et lorsque l'update va être fait, un autre Select va être fait, puis les valeurs vont être comparées, si elles sont toutes égales, l'enregistrement n'a pas été modifie, en revanche, si elles sont différentes, l'update ne pourra se faire car une modification a eu lieu entre temps.

Comment renvoyer cette information a l'utilisateur ?

Vous noterez qu'on appelle la fonction "miseAjour" lorsque l'évènement "Updated" se produit dans le datasource. Cet évènement se produit après l'update. (Il existe ainsi beaucoup d'évènement pour le datasource : Updated (après l'update), Updating (avant l'update), Selected (après le select), Selecting (avant le select), Inserted (après l'insert), inserting (avant l'insert).

La fonction miseAjour :

```
protected void miseAjour(object sender, SqlDataSourceStatusEventArgs e)
    {
    if (e.AffectedRows == 0) {  } // FAITES CE QUE VOUS VOULEZ ICI
    }
```

La fonction mise a jour, étant appelée par l'évènement Updated reçoit un paramètre "**SqlDataSourceStatusEventArgs**" (les autres évènements cites ci-avant font de même).
Parmi les propriétés de cet évènement, la propriété "**AffectedRows**" indiquent le nombre d'enregistrement qui ont été affectes par la commande, si le nombre est 0, c'est que la mise a jour n'a pu se produire, vous pouvez donc afficher un message invitant l'utilisateur a retenter la manipulation.

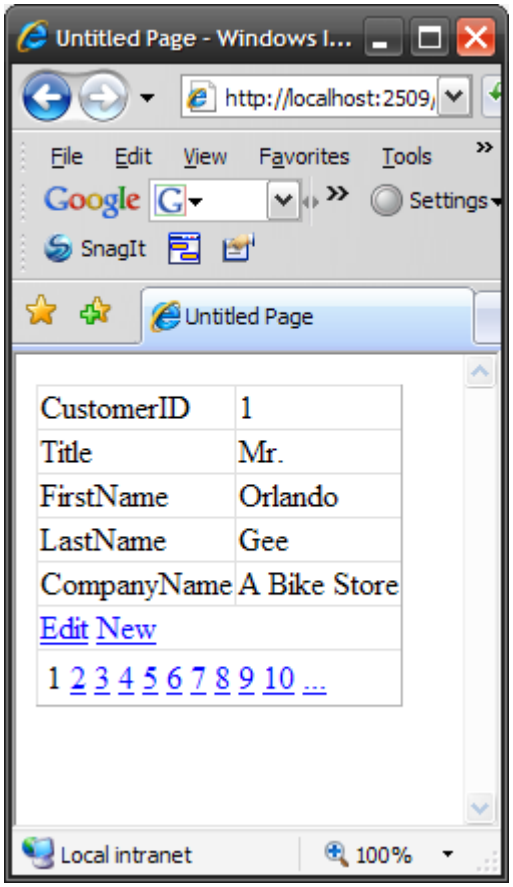
10.10.6 Ajouter des données a la base

On peut bien sur insérer de nouveaux enregistrements dans la base avec le DetailsView.

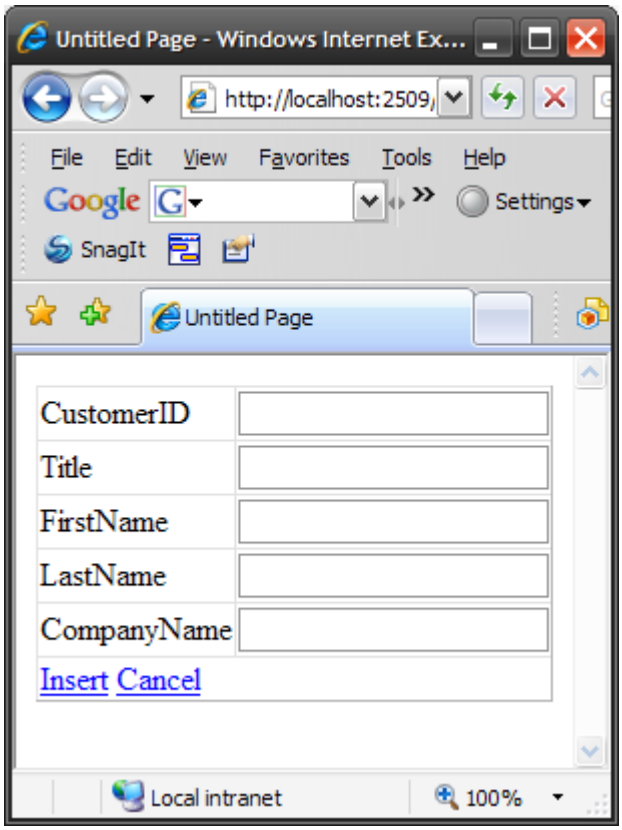
Si vous modifiez le GridView de cette façon :

```
<asp:DetailsView ID="DetailsView1" runat="server" DataSourceID="datasource"
AutoGenerateRows="true" AllowPaging="true"
DataKeyNames="CustomerID" AutoGenerateEditButton="true" AutoGenerateInsertButton="true"
>
</asp:DetailsView>
```

En initialisant la propriété "**AutoGenerateInsertButton**" a "**true**", un bouton "New" apparait dans le formulaire :



En cliquant sur ce bouton, un formulaire vierge apparait :



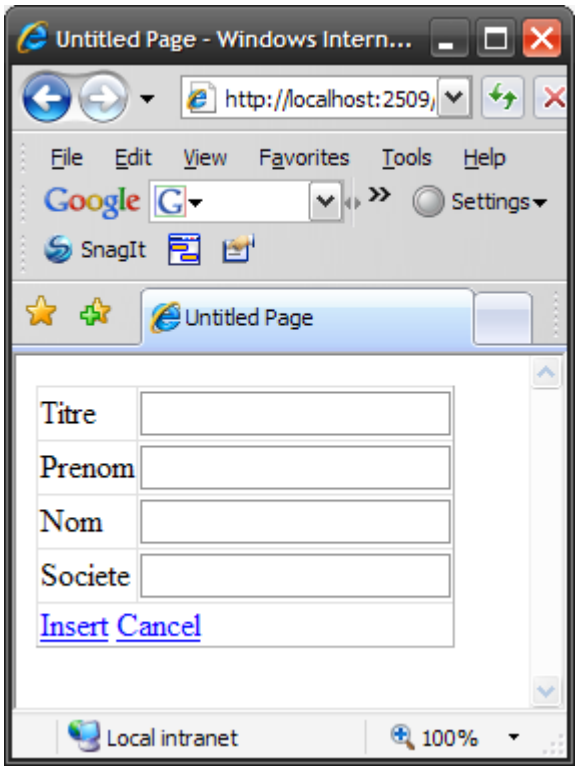
Si vous cliquez sur "Insert", le nouvel enregistrement est ajouté à la base, "Cancel" annule la création.

Sauf que, comme vous le constatez, l'identifiant du client est un champ que l'utilisateur peut saisir. Or, dans la plupart des cas, ce sera un identifiant génère automatiquement lors de l'insert par la base de données. Il ne faudrait donc pas laisser la possibilité a l'utilisateur de le saisir.

Pour cela, il va falloir modifier notre GridView :

```
<asp:DetailsView ID="DetailsView1" runat="server" DataSourceID="datasource"
AutoGenerateRows="false" AllowPaging="true"
DataKeyNames="CustomerID" AutoGenerateEditButton="true" AutoGenerateInsertButton="true"
>
<Fields>
    <asp:BoundField InsertVisible="false" DataField="CustomerID"
    HeaderText="Identifiant" />
    <asp:BoundField DataField="Title" HeaderText="Titre" />
    <asp:BoundField DataField="FirstName" HeaderText="Prenom" />
    <asp:BoundField DataField="LastName" HeaderText="Nom" />
    <asp:BoundField DataField="CompanyName" HeaderText="Societe" />
</Fields>
</asp:DetailsView>
```

Nous avons redéfinit les champs avec des BoundFields et notez la propriété "**InsertVisible**" du BoundField "CustomerID". Elle est initialisée a "false", ca signifie que lorsqu'on passera en insertion (en cliquant sur "New"), ce champ ne sera pas modifiable par l'utilisateur :



Nous n'aurons bien sur pas oublie de mettre a jour la datasource ainsi avec une commande INSERT.

```
<asp:SqlDataSource id="datasource"
SelectCommand=" SELECT CustomerID, Title, FirstName, LastName, CompanyName FROM
SalesLT.Customer"
UpdateCommand=" UPDATE SalesLT.Customer SET LastName=@LastName, FirstName=@FirstName
WHERE CustomerID=@CustomerID"
InsertCommand=" INSERT SalesLT.Customer (Title, FirstName, LastName, CompanyName)
VALUES (@Title, @FirstName, @LastName, @CompanyName)"
ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User
Id=sa;Password=password;"
ConflictDetection="CompareAllValues"
OnUpdated="miseAjour"
Runat="server">
</asp:SqlDataSource>
```

Si vous essayez cet exemple, vous allez avoir une drôle de surprise, en fait, vous allez avoir une belle erreur :

```
Server Error in '/test1' Application.
--------------------------------------------------------------------------------

Cannot insert the value NULL into column 'PasswordHash', table
'AdventureWorksLT.SalesLT.Customer'; column does not allow nulls. INSERT fails.
The statement has been terminated.
```

Pourquoi ?

Parce que, et vous vous trouverez confronté au même problème souvent, lorsque la table a été créée, certaines règles ont été mises en place. Dans notre cas de figure, il a été spécifié que le champ "PasswordHash" ne peut être null (vide). Or notre commande INSERT se contente de renseigner les champs titre, nom, prenom, société (et identifiant de façon automatique), mais nulle part le champ "PasswordHash" n'est renseigné.

Vous pourriez vous dire, qu'a cela ne tienne, nous allons ajouter un BoundField pointant sur ce champ, l'utilisateur le saisissant, le problème sera résolu. Sauf que ce champ est typiquement une donnée qui ne peut être saisie par l'utilisateur (en l'occurrence ici, un calcul fait sur le mot de passe du client (que nous n'avons d'ailleurs pas saisi)).

Lors de l'insertion d'un nouvel enregistrement dans la table, certains champs doivent être saisis par l'utilisateur, d'autres champs doivent avoir leur valeur initialisée par une fonction, une méthode d'une classe, etc… Pour cela, il existe plusieurs façons de procéder. Je vous donne ici celle que je pense être la plus simple :

D'abord modifiez la datasource pour inclure les fameux champs :

```
InsertCommand=" INSERT SalesLT.Customer (Title, FirstName, LastName, CompanyName,
PasswordHash, PasswordSalt) VALUES (@Title, @FirstName, @LastName, @CompanyName,
@PasswordHash, @PasswordSalt)"
```

Nous avons ajoute les champs PasswordHash et PasswordSalt, dans les values nous indiquons @PasswordHash et @PasswordSalt.

Si nous laissons tel quel, une erreur va se produire lors de l'insertion. Aucun champ de saisie existant pour ces deux champs, les paramètres @PasswordHash et @PasswordSalt seront vides et l'insertion sera refusée. Il faut donc renseigner ces deux paramètres. Pour cela, rien de plus simple.

Le DetailsView fonctionne en bien des points comme le GridView. En l'occurrence, il possède les mêmes évènements (voir le paragraphe sur le GridView). Il en possède même deux de plus : **Inserting** (qui va se déclencher juste avant une insertion) et **Inserted** (qui se déclenché juste après l'insertion). Nous allons donc "accrocher" une fonction a l'évènement "Inserting" pour renseigner ces paramètres juste avant de les utiliser. Et les renseigner est très simple :

```
protected void DetailsView1_ItemInserting( object sender, DetailsViewInsertEventArgs e
)
{
    e.Values["PasswordHash"] = "valeur1";
    e.Values["PasswordSalt"] = "valeur2";
}
```

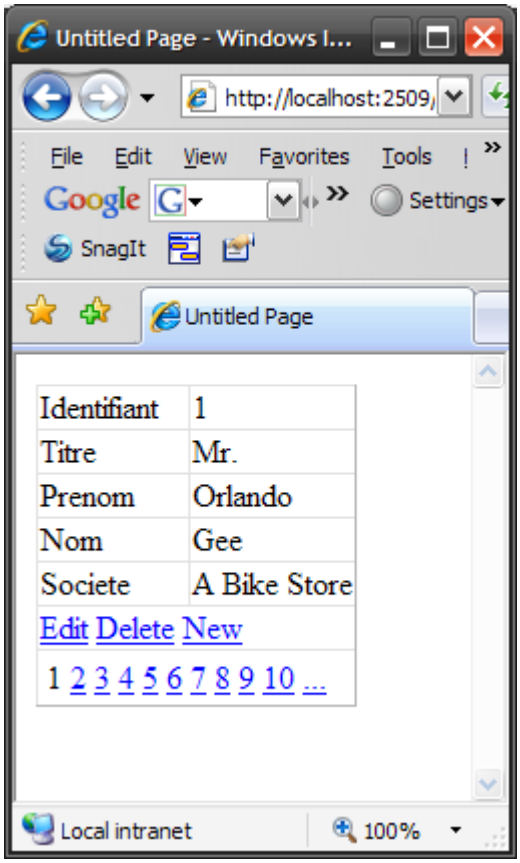
La propriété Values du paramètre e est justement la liste des paramètres et leurs valeurs qui vont être insérées dans la table. Il suffit de renseigner ici les valeurs désirées (au passage on peut modifier les valeurs saisies par l'utilisateur).

10.10.7 Supprimer les donnes du DetailsView

On peut bien sur également supprimer des données de la table a partir du DetailView.

Pour cela, il suffit d'initialiser la propriété "AutoGenerateDeleteButton" du GridView a "true".
Et bien sur d'ajouter une commande DELETE au datasource :

```
DeleteCommand=" DELETE SalesLT.Customer WHERE CustomerId=@CustomerId"
```

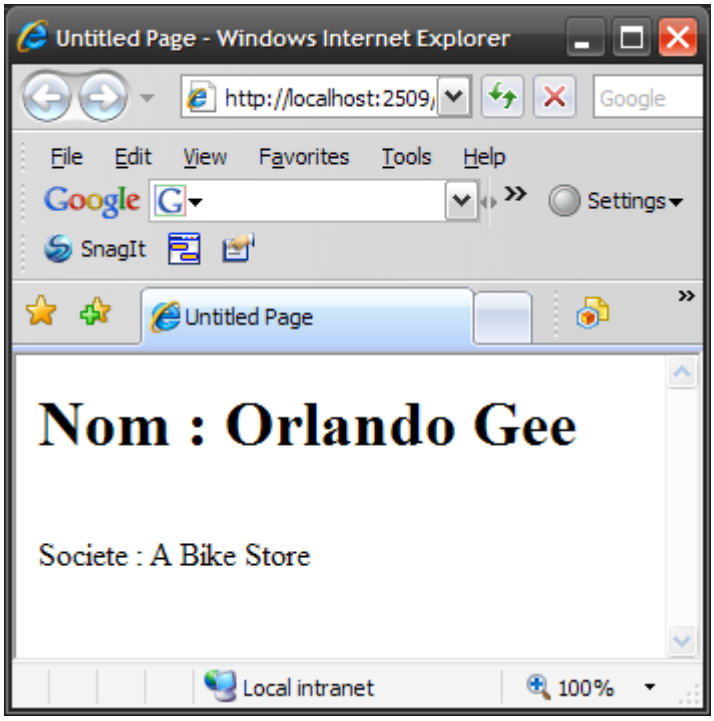


Un bouton "Delete" est apparu. En cliquant dessus, l'enregistrement est supprimé.

10.11 Le contrôle FormView

Vous pouvez utiliser le contrôle FormView pour faire tout ce que vous faites avec le contrôle DetailsView. Vous pouvez donc l'utiliser pour afficher, éditer, supprimer ou insérer des enregistrements. Cependant, contrairement au contrôle DetailsView, le FormView fonctionne uniquement avec des templates.
Vous pouvez donc contrôler d'avantage l'aspect de votre formulaire.

Commençons par un formulaire ultra simple : il va afficher, pour un client, son nom et sa société :

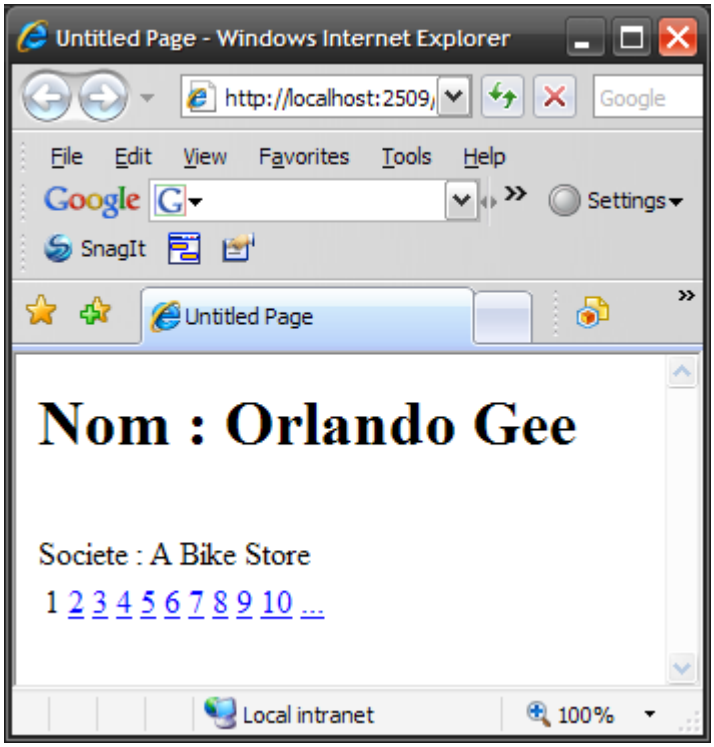


Pas très beau, mais très simple :

```
<asp:FormView id="fvClients" DataSourceID="dataSource" Runat="server">
  <ItemTemplate>
     <h1>Nom : <%# Eval("FirstName") %> <%# Eval("LastName") %></h1>
     <br />
     Societe : <%# Eval("CompanyName") %>
 </ItemTemplate>
```

Comme vous le voyez, l'affichage est fait dans un template, on utilise donc la function Eval vue plus haut pour afficher les champs désirés.

Comme le DetailsView, le FormView supporte la pagination. Initialiser la propriété "**AllowPaging**" a "true"

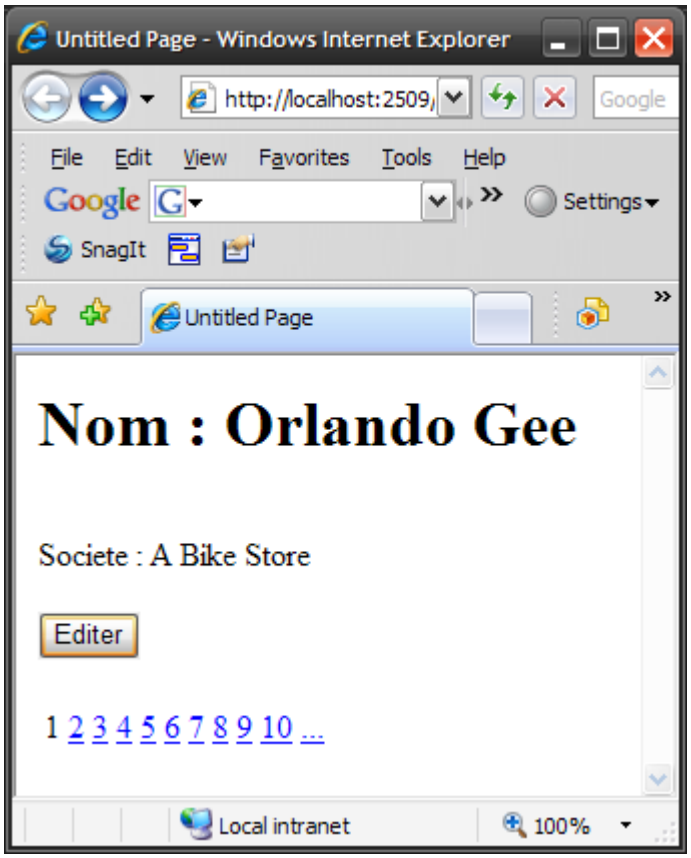


Fait apparaitre le paginateur et permet d'accéder aux autres enregistrements.

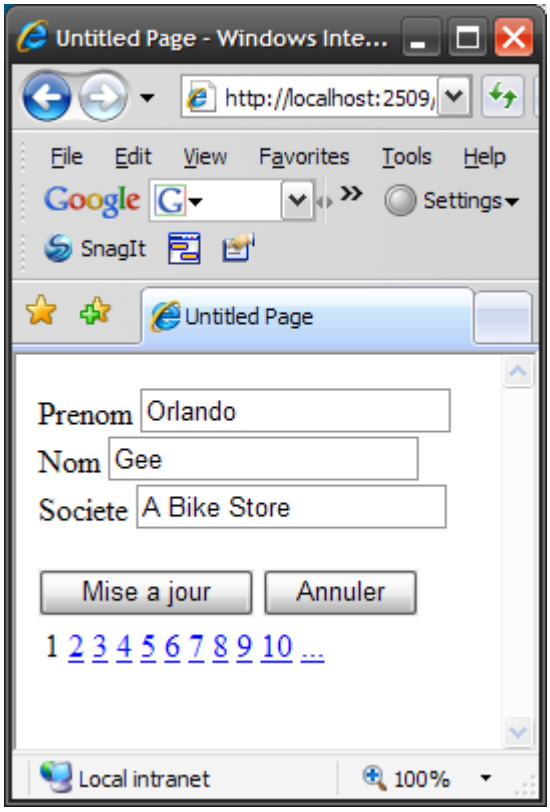
Vous pouvez bien sur personnaliser le paginateur avec un template PagerTemplate exactement de la même façon que le paginateur pour le GridView. Je vous laisse donc expérimenter.

10.11.1 Editer avec le FormView

Comme pour l'affichage, l'édition se fait via des templates. Modifions donc le formulaire :

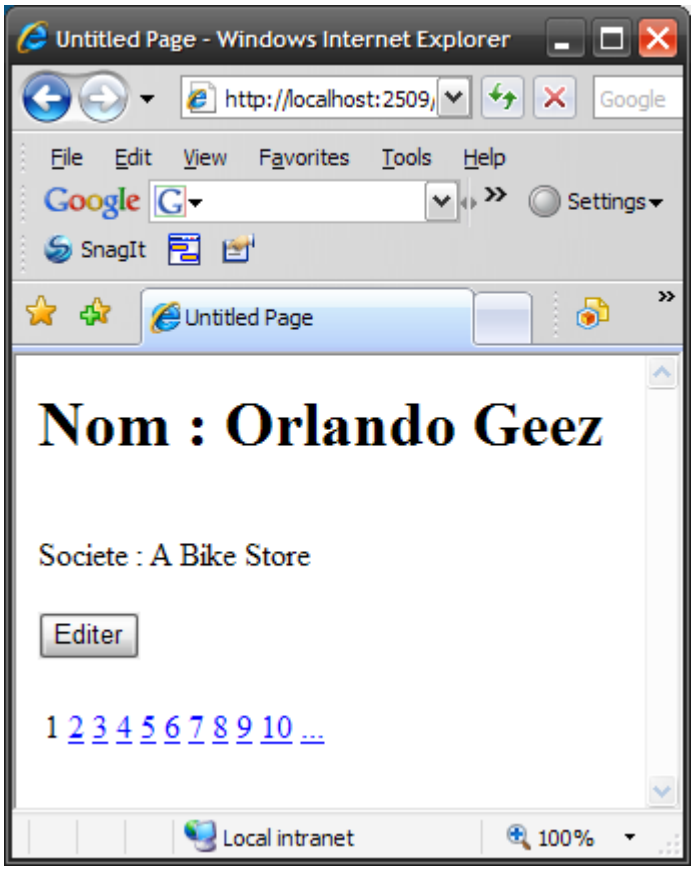


Un bouton "Editer" a été ajouté. Lorsqu'on clique dessus, le formulaire passe en mode édition :



Les champs deviennent des TextBox. Deux boutons sont affiches : un pour mettre a jour l'enregistrement et un pour annuler la modification.

Une fois la modification faite et après avoir clique sur "Mise a jour" :



Le prénom est modifié.

Comment avons-nous fait ? C'est extrêmement simple :

Le FormView est modifie pour utiliser un template pour l'édition :

```
<asp:FormView id="fvClients" DataSourceID="dataSource" Runat="server"
AllowPaging="true" DataKeyNames="CustomerID">
    <ItemTemplate>
        <h1>Nom : <%# Eval("FirstName") %> <%# Eval("LastName") %></h1>
        <br />
        Societe : <%# Eval("CompanyName") %>
        <br /><br />
        <asp:Button ID="Button1" runat="server" Text="Editer"
        CommandName="Edit" />
        <br /><br />
    </ItemTemplate>
    <EditItemTemplate>

      <asp:Label ID="lblPrenom" runat="server"
       AssociatedControlID="txtPrenom">Prenom</asp:Label>
      <asp:TextBox ID="txtPrenom" runat="server"
      Text='<%# Bind("FirstName") %>' /><br />
      <asp:Label ID="lblnom" runat="server"
       AssociatedControlID="txtNom">Nom</asp:Label>
      <asp:TextBox ID="txtNom" runat="server"
      Text='<%# Bind("LastName") %>' />
      <br />
      <asp:Label ID="lblSociete" runat="server"
       AssociatedControlID="txtNom">Societe</asp:Label>
      <asp:TextBox ID="TextBox1" runat="server"
      Text='<%# Bind("CompanyName") %>' />
     <br /><br />
     <asp:Button ID="Button2" runat="server" Text=" Mise a jour "
      CommandName="Update" />
     <asp:Button ID="Button3" runat="server" Text=" Annuler "
      CommandName="Cancel" />

    </EditItemTemplate>
</asp:FormView>
```

L'ItemTemplate est modifié : on y a ajouté un bouton "Edition". C'est ce bouton qui va, lorsqu'on cliquera dessus, passer en mode Edition. Pour cela, il suffit d'initialiser la propriété "**CommandName**" avec la valeur "**Edit**"

Nous avons ajouté un EditTemplate. Ce template contient des labels (pour le texte fixe) et des TextBoxes dans lesquelles l'utilisateur va saisir les nouvelles valeurs. Pour initialiser le contenu des TextBoxes, il suffit d'utiliser la fonction "Bind" que nous avons vu précédemment.

Enfin, nous avons ajouté deux boutons, l'un pour mettre à jour et l'un pour annuler la modification. Comme pour le bouton d'édition, il suffit d'initialiser la propriété CommandName du bouton avec "Update" pour que le bouton fasse la mise a jour, et "Cancel" pour qu'il annule la modification.

Notez que les boutons peuvent être l'un des trois types de boutons ASP.NET (Button, LinkButton ou ImageButton).

Il faut également modifier le datasource :

```
<asp:SqlDataSource id="datasource"
 SelectCommand=" SELECT CustomerID,  FirstName, LastName, CompanyName
                 FROM SalesLT.Customer"
 UpdateCommand=" UPDATE SalesLT.Customer SET LastName=@LastName,
                 FirstName=@FirstName, CompanyName=@CompanyName WHERE
                 CustomerID=@CustomerID"
ConnectionString="Data Source=octopus;
Initial Catalog=AdventureWorksLT;User Id=sa;Password=password;"
Runat="server">
</asp:SqlDataSource>
```

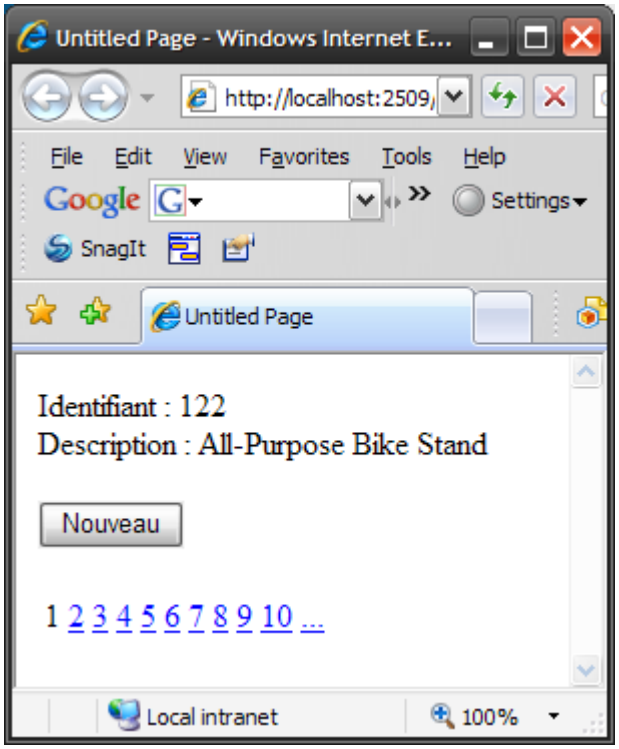
Il y a maintenant une commande Update. Vous noterez que cette commande met à jour l'enregistrement dont le champ CustomerID = @CustomerID (@CustomerID étant un paramètre). Ce champ ne fait pas partie des champs de l'item EditItemTemplate. Il a été initialisé dans la propriété "**DataKeyNames**" du FormView (exactement comme le DetailsView).

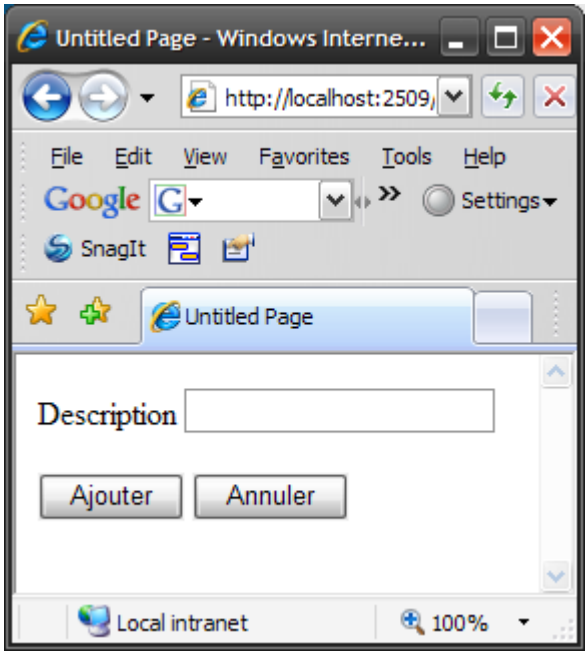
Comme dans le DetailsView, cette propriété contient le nom de la clef de l'enregistrement.

Notez que vous pouvez passer en mode édition par défaut en initialisant la propriété "**DefaultMode**" a "**Edit**"

10.11.2 Ajouter des données avec le FOrmView

On peut bien sur, également ajouter des données dans la table avec le FormView. Modifions notre FormView pour visualiser les enregistrements de la table ProductModel (pour éviter les problèmes lors de l'insertion avec les champs non null).
Nous allons ajouter un bouton "Nouveau" dans le FormView. Ce bouton va afficher un formulaire vierge dans lequel on va saisir un nouvel enregistrement :





En cliquant sur "Ajouter" , l'enregistrement est ajouté à la table.

```
<asp:FormView id="fvProduits" DataSourceID="dataSource" Runat="server"
AllowPaging="true" DataKeyNames="ProductModelID"  >
<ItemTemplate>
   Identifiant : <%# Eval("ProductModelID") %><br />
   Description : <%# Eval("Name") %><br /><br />
   <asp:Button ID="Button1" runat="server" Text="Nouveau"  CommandName="New" />
   <br /><br />
</ItemTemplate>
```

```
<EditItemTemplate>

 <asp:Label ID="lblDescription" runat="server"
 AssociatedControlID="txtDescription">Description</asp:Label>
 <asp:TextBox ID="txtDescription" runat="server"
  Text='<%# Bind("Name") %>' />
 <br /><br />
 <asp:Button ID="Button2" runat="server" Text="Ajouter" CommandName="Insert" />
 <asp:Button ID="Button3" runat="server" Text="Annuler" CommandName="Cancel" />

</EditItemTemplate>
</asp:FormView>
```

Vous noterez que le bouton utilise pour ajouter un nouvel enregistrement possède une propriété "**CommandName**" initialise a "**New**".

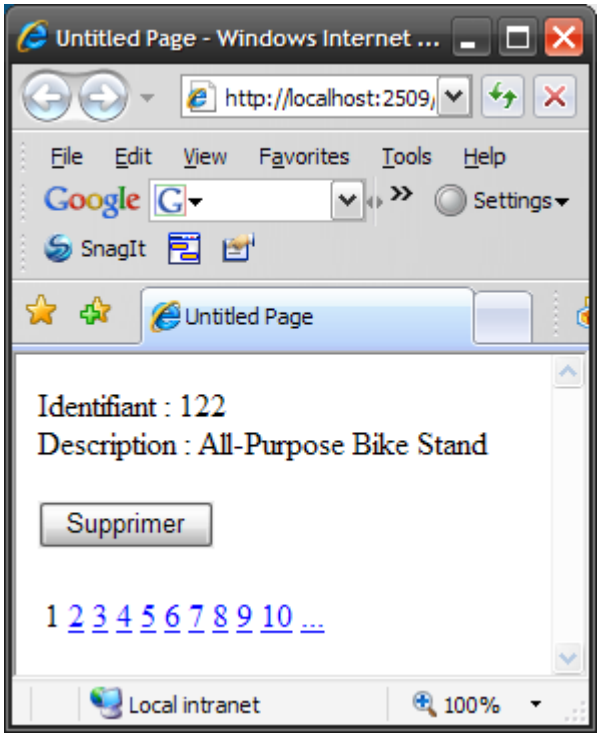
Le bouton utilisé pour insérer l'enregistrement dans la table possède une propriété "**CommandName**" initialisée a "**Insert**".

```
<asp:SqlDataSource id="datasource"
  SelectCommand=" SELECT ProductModelID, Name FROM SalesLT.ProductModel"
  InsertCommand=" INSERT SalesLT.ProductModel (Name) VALUES (@Name)"
  ConnectionString="Data Source=octopus;Initial Catalog=AdventureWorksLT;User
Id=sa;Password=password;"
  Runat="server">
</asp:SqlDataSource>
```

Une commande Insert a été ajoutée au datasource.

10.11.3 Supprimer des données avec le FormView

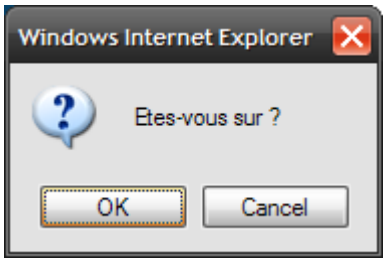
Extrêmement simple : il suffit d'ajouter un bouton dont le propriété **CommandName** est initialisée a "**Delete**". Lorsque ce bouton est clique, il appelle la commande Delete du datasource :




```
<asp:FormView id="fvProduits" DataSourceID="dataSource" Runat="server"
AllowPaging="true" DataKeyNames="ProductModelID"  >
 <ItemTemplate>
   Identifiant : <%# Eval("ProductModelID") %><br />
   Description : <%# Eval("Name") %><br /><br />
   <asp:Button ID="Button1" runat="server" Text="Supprimer"
    CommandName="Delete"
    OnClientClick="return confirm('Etes-vous sur ?')" />
    <br /><br />
    </ItemTemplate>
 </asp:FormView>
```

Le FormView est très simple, nous n'avons pas besoin d'autre template que le template ItemTemplate.

Notez la propriété "OnClientClick" du bouton "supprimer". Elle contient du code JavaScript (qui peut être l'appel d'une fonction). Si le code JavaScript renvoie false, la suppression sera annulée.
Dans notre cas, nous appelons la fonction confirm qui va, lors du click sur le bouton, afficher cette boite d'alerte :



Si l'utilisateur clique sur "Cancel", la fonction renverra false et la suppression sera annulée.

11 Accéder aux donnés par programmation

Nous avons, jusqu'à présent, utilisé les objets DataSource directement dans les pages ASPX pour accéder aux données. Nous pouvons évidemment accéder également aux donnés par programmation dans le code behind associé à la page. Il y a deux façons d'accéder aux données :

- De façon "connectée" : les données sont récupérées dans la base de données au fur et à mesure et envoyés aux composants qui les utilisent, c'est ce que fait le contrôle DataSource.
- De façon "déconnectée": les données sont chargées d'abord dans un DataSet (une sorte de base de données relationnelle en mémoire), puis sont lues a partir de ce DataSet. Elles peuvent également être modifiées dans ce DataSet (ce qui ne touche pas a la base elle-même), il faudra donc renvoyer plus tard les modifications faites a la base.

On peut utiliser pour faire des requêtes dans la base, soit SQL, soit Linq to SQL, une extension au langage C# présente dans .NET 3.5. Linq est un langage permettant de faire des requêtes complexes sur n'importe quel type d'objet de façon standardisée (tableau, liste, vecteur, XML, base de données).

Je n'entrerai pas dans le détail de toutes ces fonctionnalités. Je vous invite a consulter la documentation relative a ADO.NET (la partie "accès aux bases de données" de .NET) et LINQ sur Internet.

11.1 Les acces aux données connectées

Les trois objets utilises pour accéder aux données connectees sont :

- **Connection** : Pour se connecter sur la source de données
- **Command** : Pour exécuter des commandes dans la source de données
- **DataReader** : Pour récupérer et manipuler les données récupérées dans la source

Ces trois objets sont disponibles en différents parfums qui vont dépendre de la base de données utilisée. Les drivers livres avec .NET sont :

- System.Data.SqlClient : driver pour se connecter sur une base SQL Server
- System.Data.OleDb : driver pour se connecter sur une base ayant un pilote OLE-DB (quasiment toutes)
- System.Data.Odbc : driver pour se connecter sur une base ayant un pilote ODBC

Pour info, OLE-DB est un sur ensemble d'ODBC. ODBC est une technologie ancienne permettant d'accéder de façon unifiée aux bases de données relationnelles. OLE-DB permet en plus d'accéder, via SQL, a des données non relationnelles (un tableur par exemple).
Passer par OLE DB ou ODBC est évidemment plus lent qu'appeler le driver de la base de données directement (on dit "nativement" comme par exemple avec System.Data.SqlClient), mais ces drivers ont le mérite de pouvoir être utilises avec toutes les bases de données. De plus en plus de fournisseurs de bases de données proposent désormais des drivers ADO.NET "natif". Vous pouvez par exemple télécharger le namespace System.Data.OracleClient chez Oracle pour accéder aux bases Oracle ou System.Data.MySQLClient pour MySQL.

Une fois que vous avez importe le bon namespace, vous devez utiliser les objets Connection, Command et DataReader correspondants. Par exemple, si vous optez pour un driver SQL Server, ce sera SqlConnection, SqlCommand et SqlDataReader. Si vous utilisez le driver OLE DB, ce sera OleDbConnection, OleDbCommand, OleDbDataReader, et ainsi de suite.

L'approche proposée par .NET n'est pas idéale car elle oblige (si on utilise les drivers natifs) a indiquer explicitement les classes a utiliser, si par la suite on change de base de données, il faudra modifier le code (certes, ce n'est que du rechercher / remplacer, mais sur un gros projet, ca devient vite pénible). Il existe un moyen d'éviter cela en utilisant les classes de bases dont dérivent toutes les classes des drivers natifs (DbConnection, DbCommand et DbDataReader), mais on dépasse ici le sujet de la formation. Notez cependant que ce problème disparait lorsqu'on utilise Linq (nous verrons cela après).

Il y a deux façons de peupler les données d'un contrôle ASP.NET, commençons par la plus brute (que nous avions déjà rencontrée plus haut) :

11.1.1 DataBinding direct

Supposons une page ASPX contenant un seul contrôle : un GridView :

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
    <form id="form1" runat="server">
    <div>
        <asp:GridView ID="GridView1" runat="server" />
    </div>
    </form>
</body>
</html>
```

Le code behind associe va aller lire les 10 premiers enregistrements de la table "Customers" :

```
protected void Page_Load(object sender, EventArgs e)
  {
  using (
SqlConnection cnx = new
SqlConnection(ConfigurationManager.ConnectionStrings["test"].ConnectionString))
     {
     cnx.Open();
     SqlCommand cmd = new SqlCommand("SELECT TOP 10 FirstName, LastName, CompanyName
FROM SalesLT.Customer", cnx);
     SqlDataReader reader = cmd.ExecuteReader();
     GridView1.DataSource = reader;
     GridView1.DataBind();
     cnx.Close();
     }
}
```

Que fait ce code ? Basiquement :

1. il crée une nouvelle connexion vers une base SQL Server (la chaine de connexion est récupérée dans le fichier web.config)
2. Il ouvre la connexion
3. Il crée une nouvelle commande SQL pour SQL Server (SELECT…)
4. Il crée un nouveau DataReader pour SQL Server a partir de cette commande
5. Le DataReader est utilise comme source de données pour le GridView
6. Le GridView récupère les données (GridView1.DataBind())
7. La connexion est fermée.

Notez deux choses a propos de ce code :

D'abord, vous devez importer le namespace utilise pour SQL Server :

```
using System.Data.SqlClient;
```

Ensuite, notez également l'usage du mot clef **using**

Les connexions et les acces aux bases de donnees utilisent des parties du système d'exploitation non managees (cad qui ne font pas partie de .NET). Ces parties du système, la plupart du temps ecrites en C ou en C++ allouent de la mémoire (souvenez vous avec des fonctions comme malloc) et doivent les liberer a la fin de l'utilisation (avec mfree).
C'est le cas des connexions, mais c'est egalement le cas de beaucoups d'objets du système comme les polices (classe Font) ou des fichiers (classe File) par exemple.
Pour permettre de desallouer la mémoire utilisee par ces objets, .NET les a encapsule dans des objets implementant une interface nommee IDisposable. Cette interface a une fonction nommee Dispose qui permet de desallouer la mémoire consommee par les objets.
Normallement, lorsqu'on a fini d'utiliser ces objets, .NET appelle la fonction Dispose de ces objets, mais si une exception se produit pendant leur utilisation, il y a de fortes chances pour que la mémoire allouee reste allouee.
En attendant que tout le système soit en code manage, il y a deux solutions pour pallier a ce probleme :

1. En faisant un try … catch classique :

```
try
{
SqlConnexion cnx = new SqlConnexion(…)
// code utilisant la connexion
}
catch
{
((IDisposable)cnx).Dispose();
}
```

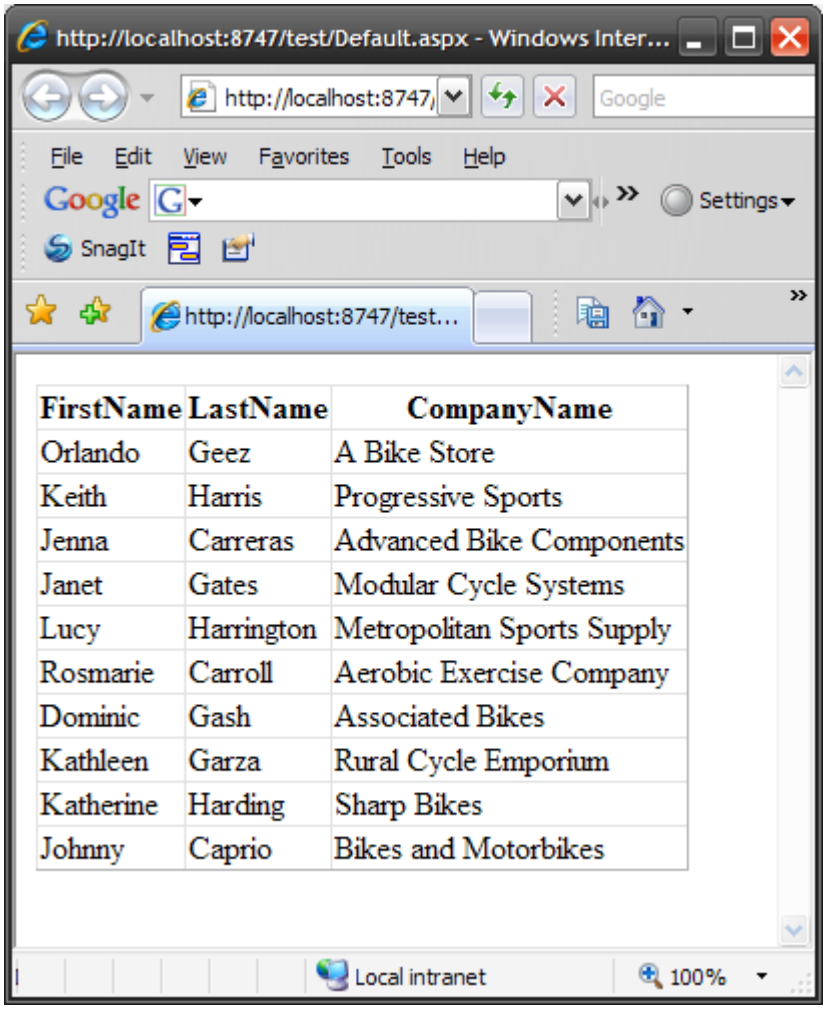
   En cas d'exception, la methode Dispose sera appellee.

2. en utilisant using :

```
using (SqlConnexion = new SqlConnexion(…))
{
// code utilisant la connexion
}
```

   Ce qui revient au meme mais est plus concis.

Le résultat donne ceci :



FirstName	LastName	CompanyName
Orlando	Geez	A Bike Store
Keith	Harris	Progressive Sports
Jenna	Carreras	Advanced Bike Components
Janet	Gates	Modular Cycle Systems
Lucy	Harrington	Metropolitan Sports Supply
Rosmarie	Carroll	Aerobic Exercise Company
Dominic	Gash	Associated Bikes
Kathleen	Garza	Rural Cycle Emporium
Katherine	Harding	Sharp Bikes
Johnny	Caprio	Bikes and Motorbikes

L'affichage est brut, mais rien n'a été configure pour personnaliser le GridView.

11.1.2 Initialisation "manuelle"

OK, maintenant, faisons plus complique (?). Nous allons utiliser le DataReader pour lire tous les enregistrements renvoyes et peupler le DataGrid "a la main".

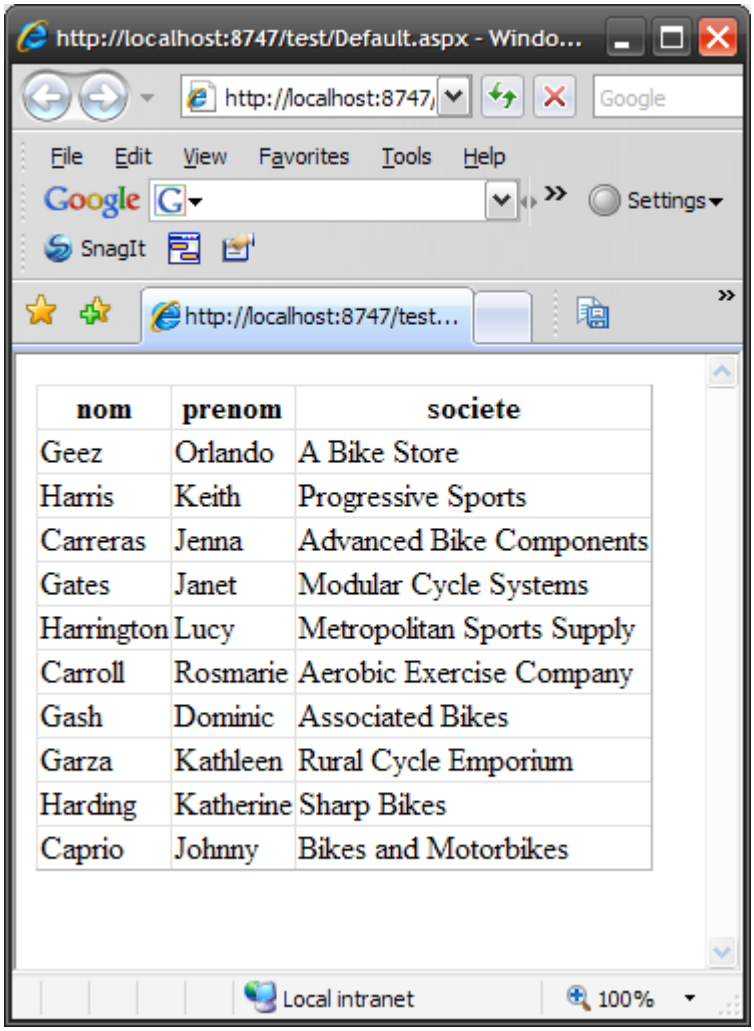
```
using System.Collections.Generic;

public partial class _Default : System.Web.UI.Page
{
   class Client
   {
      String _nom;
      String _prenom;
      String _societe;
      public String nom
      {
             get { return _nom; }
             set { _nom = value; }
      }
      public String prenom
      {
             get { return _prenom; }
             set { _prenom = value; }
      }
      public String societe
      {
             get { return _societe; }
             set { _societe = value; }
      }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
       using (SqlConnection cnx = new
SqlConnection(ConfigurationManager.ConnectionStrings["test"].ConnectionString))
            {
            cnx.Open();
            SqlCommand cmd = new SqlCommand("SELECT TOP 10 FirstName,
            LastName, CompanyName FROM SalesLT.Customer", cnx);
            SqlDataReader reader = cmd.ExecuteReader();
```

```
            List<Client> clients = new List<Client>();
            while (reader.Read())
            {
                Client client = new Client();
                client.prenom = (String)reader["FirstName"];
                client.nom = (String)reader["LastName"];
                client.societe = (String)reader["CompanyName"];
                clients.Add(client);
            }
            GridView1.DataSource = clients;
            GridView1.DataBind();
            cnx.Close();
        }
    }
}
```

Qu'avons-nous fait ?

Je crée au début une classe client avec getters et setters appropries, puis je fais la même chose que précédemment mais au lieu de binder directement le DataReader sur le GridView, je le parcours et j'initialise une liste d'objets "Client", c'est cette liste qui est bindee a la fin sur le DataGrid.
Vous noterez au passage qu'il a fallu que j'importe le namespace System.Collections.Generic pour utiliser les listes (objet Liste).

Le résultat :



nom	prenom	societe
Geez	Orlando	A Bike Store
Harris	Keith	Progressive Sports
Carreras	Jenna	Advanced Bike Components
Gates	Janet	Modular Cycle Systems
Harrington	Lucy	Metropolitan Sports Supply
Carroll	Rosmarie	Aerobic Exercise Company
Gash	Dominic	Associated Bikes
Garza	Kathleen	Rural Cycle Emporium
Harding	Katherine	Sharp Bikes
Caprio	Johnny	Bikes and Motorbikes

Vous noterez que les titres des colonnes sont les noms des propriétés de la classe.

Vous pouvez également continuer d'utiliser des DataSources dans les pages ASPX, mais, au lieu de pointer directement sur des bases de données, vous pouvez leur faire utiliser vos propres classes et y implémenter votre logique :

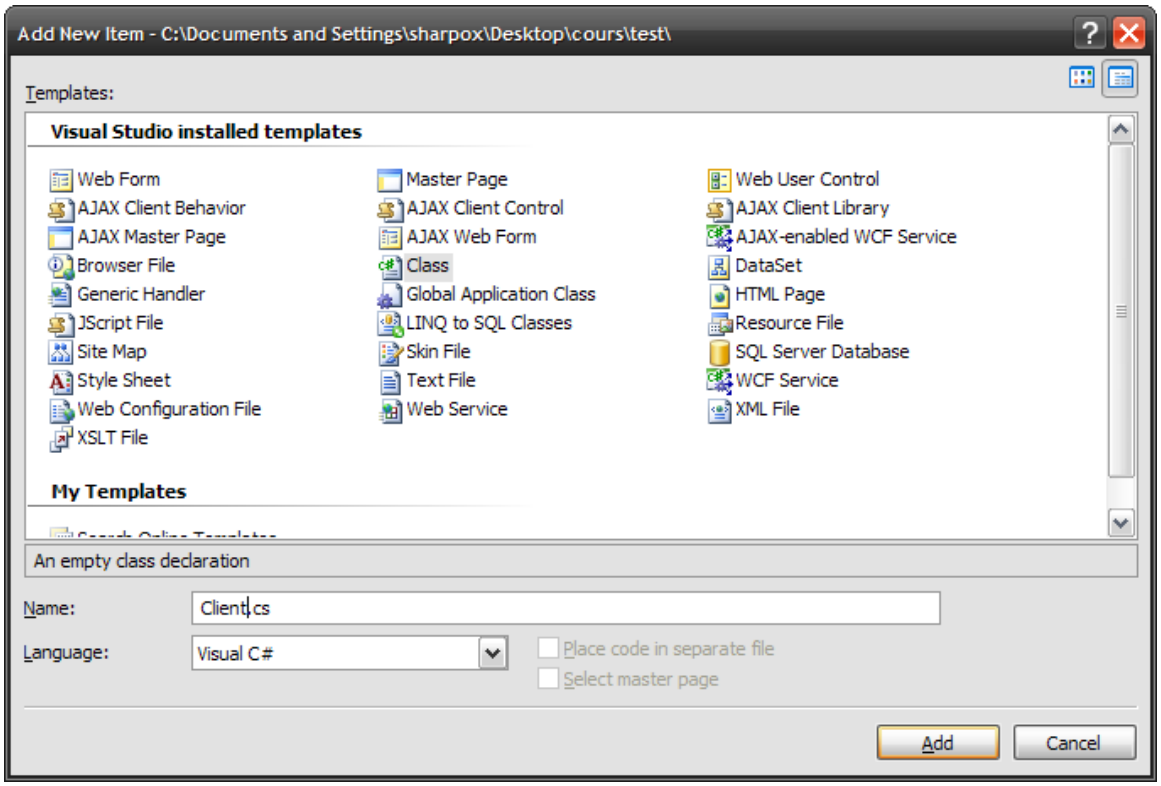
Exemple :

Modifions notre page ASPX :

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
<form id="form1" runat="server">
<div>
   <asp:GridView ID="GridView1" runat="server"
   DataSourceID="maDataSource" />
   <asp:ObjectDataSource ID="maDataSource" runat="server"
   TypeName="Client" SelectMethod="selectClients" />
</div>
</form>
</body>
</html>
```

Notre GridView utilise une DataSource nommée "maDataSource". Cette DataSource est une DataSource d'un nouveau type : c'est une ObjectDataSource : au lieu d'aller chercher ses données dans une base de données, elle va aller les chercher dans un objet dont le nom de la classe est défini par l'attribut "TypeName" (ici Client).
Comme les autres DataSources, elle a des proprietes select, update, delete… de manière a pouvoir être utilisée de la même façon avec tous les objets bindables d'ASP.NET. La propriété SelectMethod indique donc le nom d'une méthode de la classe Client a appeler pour renvoyer les données a binder dans le contrôle.

Il nous a donc fallu créer une classe "Client" a part (en cliquant avec le bouton droit sur la racine du projet, puis sur "Add new Item…") :



La classe (comme toutes les classes "isolées") sera placée dans un dossier nomme "App_Code".

Voici son contenu :

```
using System.Data.SqlClient;
using System.Collections.Generic;

public class Client
{
    String _nom;
    String _prenom;
    String _societe;

    public String nom
      {
      get { return _nom; }
      set { _nom = value; }
      }
    public String prenom
      {
      get { return _prenom; }
      set { _prenom = value; }
      }
    public String societe
      {
      get { return _societe; }
      set { _societe = value; }
      }
```

```
    public List<Client> selectClients()
    {
        List<Client> clients = new List<Client>();

        using (SqlConnection cnx = new
SqlConnection(ConfigurationManager.ConnectionStrings["test"].ConnectionString))
        {
            cnx.Open();
            SqlCommand cmd = new SqlCommand("SELECT TOP 10 FirstName,
            LastName, CompanyName FROM SalesLT.Customer", cnx);
            SqlDataReader reader = cmd.ExecuteReader();
            while (reader.Read())
            {
                Client client = new Client();
                client.prenom = (String)reader["FirstName"];
                client.nom = (String)reader["LastName"];
                client.societe = (String)reader["CompanyName"];
                clients.Add(client);
            }
            cnx.Close();
        }

        return clients;
    }
}
```

Elle reprend quasiment le même code que précédemment sauf qu'aucun binding n'est fait ici. Lorsque la page sera créée, ASP.NET ira chercher la DataSource, puis la classe utilisée par cette DataSource (Client), créera une instance de la classe et appellera la méthode "selectClients" pour récupérer les données a afficher.

Supposons maintenant, qu'en plus d'afficher des données a partir de cet objet, on veuille en supprimer ou en modifier. C'est très simple, on va d'abord modifier la classe Client pour lui ajouter deux méthodes : une pour supprimer, et une pour modifier :

```
using System.Data.SqlClient;
using System.Collections.Generic;

public class Client
{
    String _nom;
    String _prenom;
    String _societe;
    int _id;

    public int id
    {
        get { return _id; }
        set { _id = value; }
    }
    public String nom
    {
        get { return _nom; }
        set { _nom = value; }
    }
    public String prenom
    {
        get { return _prenom; }
        set { _prenom = value; }
    }
    public String societe
    {
        get { return _societe; }
        set { _societe = value; }
    }
```

```
    public List<Client> selectClients()
    {
        List<Client> clients = new List<Client>();

        using (SqlConnection cnx = new
SqlConnection(ConfigurationManager.ConnectionStrings["test"].ConnectionString))
        {
            cnx.Open();
            SqlCommand cmd = new SqlCommand("SELECT TOP 10 CustomerId,
            FirstName, LastName, CompanyName FROM SalesLT.Customer",
            cnx);
            SqlDataReader reader = cmd.ExecuteReader();
            while (reader.Read())
            {
                Client client = new Client();
                client.id = (int)reader["CustomerId"];
                client.prenom = (String)reader["FirstName"];
                client.nom = (String)reader["LastName"];
                client.societe = (String)reader["CompanyName"];
                clients.Add(client);
            }
            cnx.Close();
        }

        return clients;
    }
```

```
    public void updateClients(int id, String prenom, String nom, String societe)
    {
        using (SqlConnection cnx = new
SqlConnection(ConfigurationManager.ConnectionStrings["test"].ConnectionString))
        {
            cnx.Open();
            SqlCommand cmd = new SqlCommand("UPDATE SalesLT.Customer set
FirstName=@FirstName, LastName=@LastName, CompanyName=@CompanyName WHERE CustomerId =
@CustomerId", cnx);
            cmd.Parameters.AddWithValue("@CustomerId", id);
            cmd.Parameters.AddWithValue("@FirstName", prenom);
            cmd.Parameters.AddWithValue("@LastName", nom);
            cmd.Parameters.AddWithValue("@CompanyName", societe);
            cmd.ExecuteNonQuery();
            cnx.Close();
        }
     }

    public void deleteClients(int id)
    {
        using (SqlConnection cnx = new
SqlConnection(ConfigurationManager.ConnectionStrings["test"].ConnectionString))
        {
            cnx.Open();
            SqlCommand cmd = new SqlCommand("DELETE SalesLT.Customer WHERE CustomerId =
@CustomerId", cnx);
            cmd.Parameters.AddWithValue("@CustomerId", id);
            cmd.ExecuteNonQuery();
            cnx.Close();
        }
    }
}
```

Nous avons d'abord modifie un peu les propriété de l'objet pour lui ajouter un champ id, car nous aurons besoin d'un identifiant pour supprimer ou modifier un enregistrement dans la table.
Puis nous avons modifie la méthode selectClient pour lire initialiser la propriété id au passage.

Les deux nouvelles méthodes sont assez simples a comprendre :

La méthode updateClient reçoit 4 paramètres : l'identifiant de l'enregistrement a modifier, la nouvelle valeur du nom, du prenom et de la société.
Pour mettre a jour l'enregistrement, on utilise une commande SQL "UPDATE" classique. La façon d'y insérer les parametres rappellera le preparestatement de jdbc pour les amateurs de Java. On indique la valeur de chaque paramètre par @nom_du_parametre, puis, on initialise la valeur du paramètre en faisant commande.Parameters.addWithValue("@nom_du_parametre", valeur_du_parametre).
Notez qu'on appelle ensuite la méthode ExecuteNonQuery qui lance une commande SQL qui ne retourne aucun enregistrement (UPDATE, DELETE, INSERT…)

La méthode deleteClient procède exactement de la même façon.

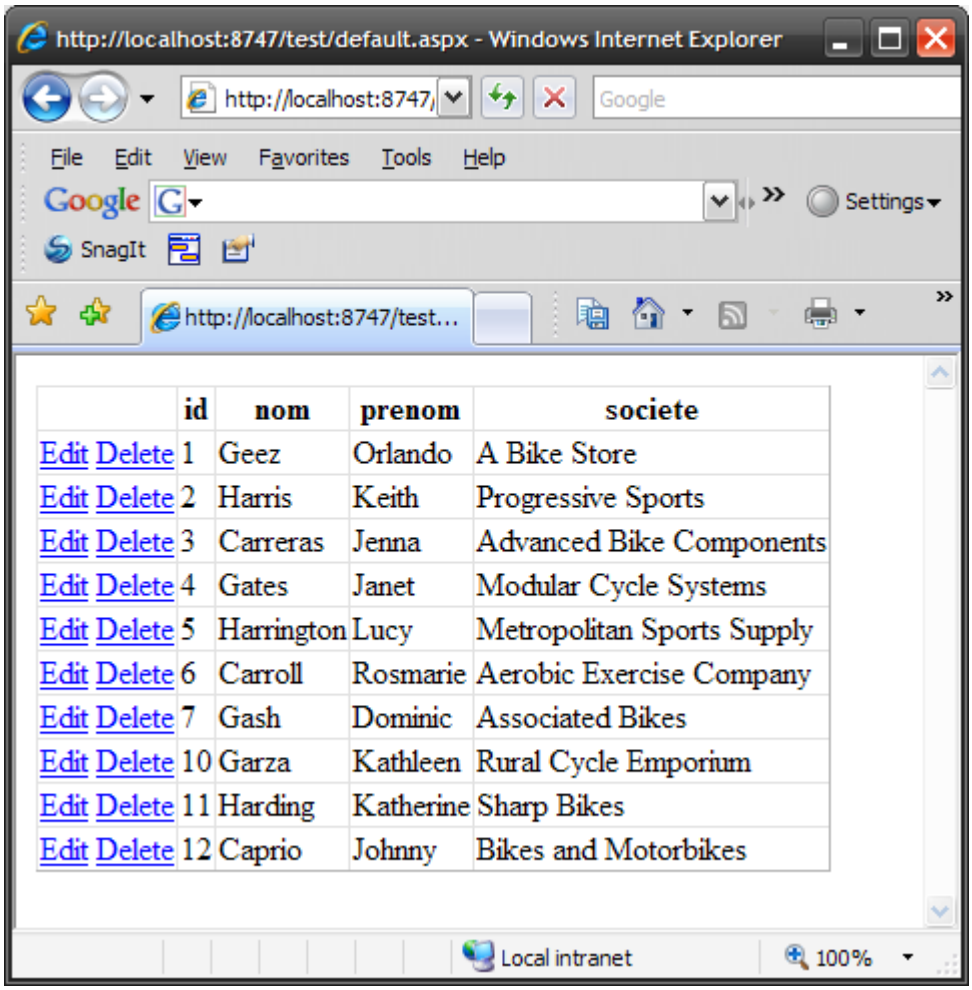
Mais ou et comment appelle-t-on ces méthodes ?

Très simplement, comme on le faisait auparavant, directement dans le GridView :

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
    <form id="form1" runat="server">
    <div>
        <asp:GridView ID="GridView1"
            runat="server"
            DataSourceID="maDataSource"
            DataKeyNames="id"
            AutoGenerateEditButton="true"
            AutoGenerateDeleteButton="true" />

        <asp:ObjectDataSource ID="maDataSource"
            runat="server"
            TypeName="Client"
            SelectMethod="selectClients"
            UpdateMethod="updateClients"
            DeleteMethod="deleteClients"
            />
    </div>
    </form>
</body>
</html>
```

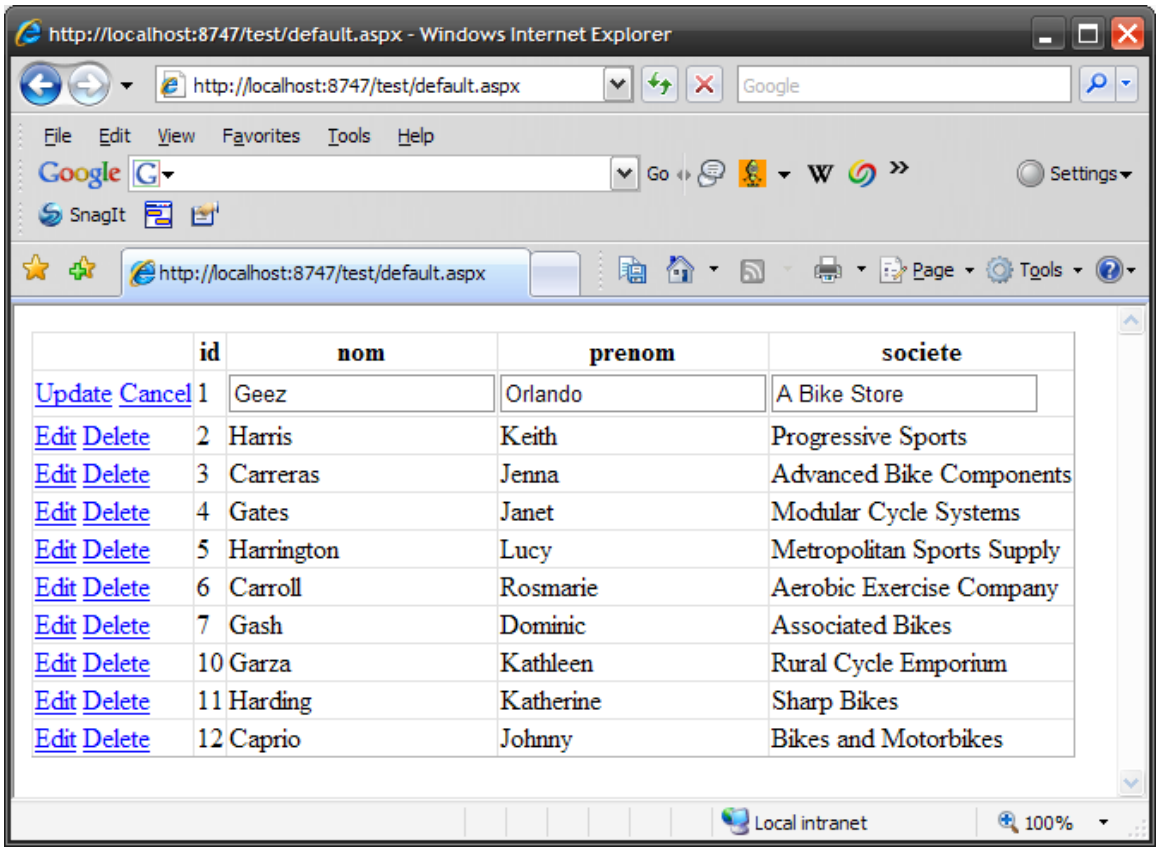
L'ObjectDataSource est modifie pour indiquer quelles sont les méthodes a appeler lors d'un update et lors d'un insert. Le GridView est simplement modifie pour faire apparaitre un bouton "Editer" et "Supprimer" (on a également initialise sa propriété DataKeyNames pour indiquer quel est la clef de chaque enregistrement).

Lorsque la page s'affiche, les liens pour modifier (Edit) et Supprimer (Delete) sont affiches :



	id	nom	prenom	societe
Edit Delete	1	Geez	Orlando	A Bike Store
Edit Delete	2	Harris	Keith	Progressive Sports
Edit Delete	3	Carreras	Jenna	Advanced Bike Components
Edit Delete	4	Gates	Janet	Modular Cycle Systems
Edit Delete	5	Harrington	Lucy	Metropolitan Sports Supply
Edit Delete	6	Carroll	Rosmarie	Aerobic Exercise Company
Edit Delete	7	Gash	Dominic	Associated Bikes
Edit Delete	10	Garza	Kathleen	Rural Cycle Emporium
Edit Delete	11	Harding	Katherine	Sharp Bikes
Edit Delete	12	Caprio	Johnny	Bikes and Motorbikes

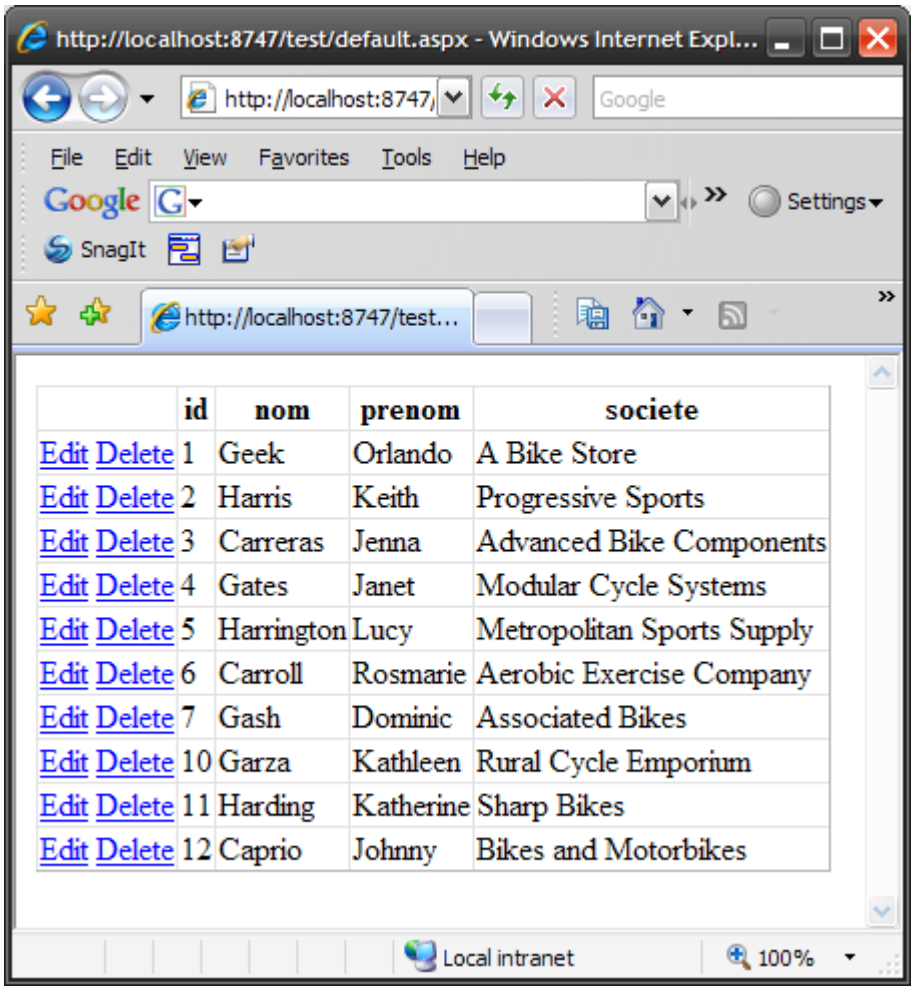
Je vais modifier la première ligne, je clique sur "Edit" :



	id	nom	prenom	societe
Update Cancel	1	Geez	Orlando	A Bike Store
Edit Delete	2	Harris	Keith	Progressive Sports
Edit Delete	3	Carreras	Jenna	Advanced Bike Components
Edit Delete	4	Gates	Janet	Modular Cycle Systems
Edit Delete	5	Harrington	Lucy	Metropolitan Sports Supply
Edit Delete	6	Carroll	Rosmarie	Aerobic Exercise Company
Edit Delete	7	Gash	Dominic	Associated Bikes
Edit Delete	10	Garza	Kathleen	Rural Cycle Emporium
Edit Delete	11	Harding	Katherine	Sharp Bikes
Edit Delete	12	Caprio	Johnny	Bikes and Motorbikes

Comme avec tout GridView, la ligne devient éditable sans une seule ligne de programmation. La colonne id étant la DataKeyName, elle n'est pas editable.

Je modifie le nom du premier client (je remplace Geez par Geek) et je clique sur "Update" pour mettre a jour :



	id	nom	prenom	societe
Edit Delete	1	Geek	Orlando	A Bike Store
Edit Delete	2	Harris	Keith	Progressive Sports
Edit Delete	3	Carreras	Jenna	Advanced Bike Components
Edit Delete	4	Gates	Janet	Modular Cycle Systems
Edit Delete	5	Harrington	Lucy	Metropolitan Sports Supply
Edit Delete	6	Carroll	Rosmarie	Aerobic Exercise Company
Edit Delete	7	Gash	Dominic	Associated Bikes
Edit Delete	10	Garza	Kathleen	Rural Cycle Emporium
Edit Delete	11	Harding	Katherine	Sharp Bikes
Edit Delete	12	Caprio	Johnny	Bikes and Motorbikes

La ligne est automatiquement modifiée et mise a jour dans la base. Comment ? Lorsqu'on a clique sur "Update", la méthode nommée dans la propriété "UpdateMethod" est appelée avec ses paramètres correctement renseignes en fonction du contenu de la ligne. Tout est fait automatiquement sans que vous ayez a programmer la logique qui lie l'objet servant a la présentation web et votre classe.

Cette façon de procéder permet de bien dissocier le code générant le contenu du code des pages elles-mêmes. On peut bien entendu l'utiliser également avec les accès déconnectés que nous allons voir a présent.

11.2 Travailler en mode déconnecté

Nous venons de voir que nous pouvions travailler en mode "connecte" : une requête dans la base de données renvoie des résultats et nous travaillons directement sur ces résultats. Nous modifions les données et les modifications sont instantanément reportées dans la base.
ADO.NET permet également de travailler en mode "déconnecté" : les données sont récupérées dans la base, placées en mémoire dans des structures et on travaille sur ces données de façon indépendante (la base de donnée peut même être stoppée). Une fois les données modifiées, on peut, éventuellement, reporter les modifications dans la base.

Pour travailler en mode déconnecté, nous allons utiliser les objets suivants :

- **DataAdapter** : Il va aller chercher les données dans la base pour les placer en mémoire
- **DataTable :** Représente une table de base de données en mémoire
- **DataView :** Represente une vue de base de données en mémoire
- **DataSet :** Represente une base de données complète en mémoire.

L'utilisation des DataSets est relativement complexe et demanderait une formation a elle seule. Nous n'entrerons donc pas dans les détails. Sachez simplement que ces objets utilisent de façon interne les objets vus précédemment (un DataAdapter va utiliser un DataReader). Ils sont plus puissants mais mobilisent plus de mémoire et sont de façon générale plus lents que les objets connectes.

Commençons avec le DataAdapter

11.2.1 Le DataAdapter

Le DataAdapter va faire la liaison entre la base de donnée "physique" et les objets de la base de données en mémoire. Il va lire les données dans la base et remplir des tables en mémoire. C'est également lui qui va renvoyer les données modifiées vers la base de données.
Comme pour les objets connectes, cet objet existe en plusieurs parfums : pour une utilisation avec SQL Server, on utilisera SqlDataAdapter, pour une utilisation avec OLE DB, on utilisera OleDbDataAdapter, etc…
Commençons un exemple simple, nous allons utiliser un DataAdapter pour remplir une table en mémoire, puis afficher cette table dans notre GridView :

La classe Client est modifiée :

```
public class Client
{
    public DataTable selectClients()
    {
       String cnxString =
       ConfigurationManager.ConnectionStrings["test"].ConnectionString;
       SqlDataAdapter adapter = new SqlDataAdapter("SELECT TOP 10
CustomerId, FirstName, LastName, CompanyName FROM SalesLT.Customer", cnxString);
        DataTable table = new DataTable();
        adapter.Fill(table);
        return table;
    }
}
```

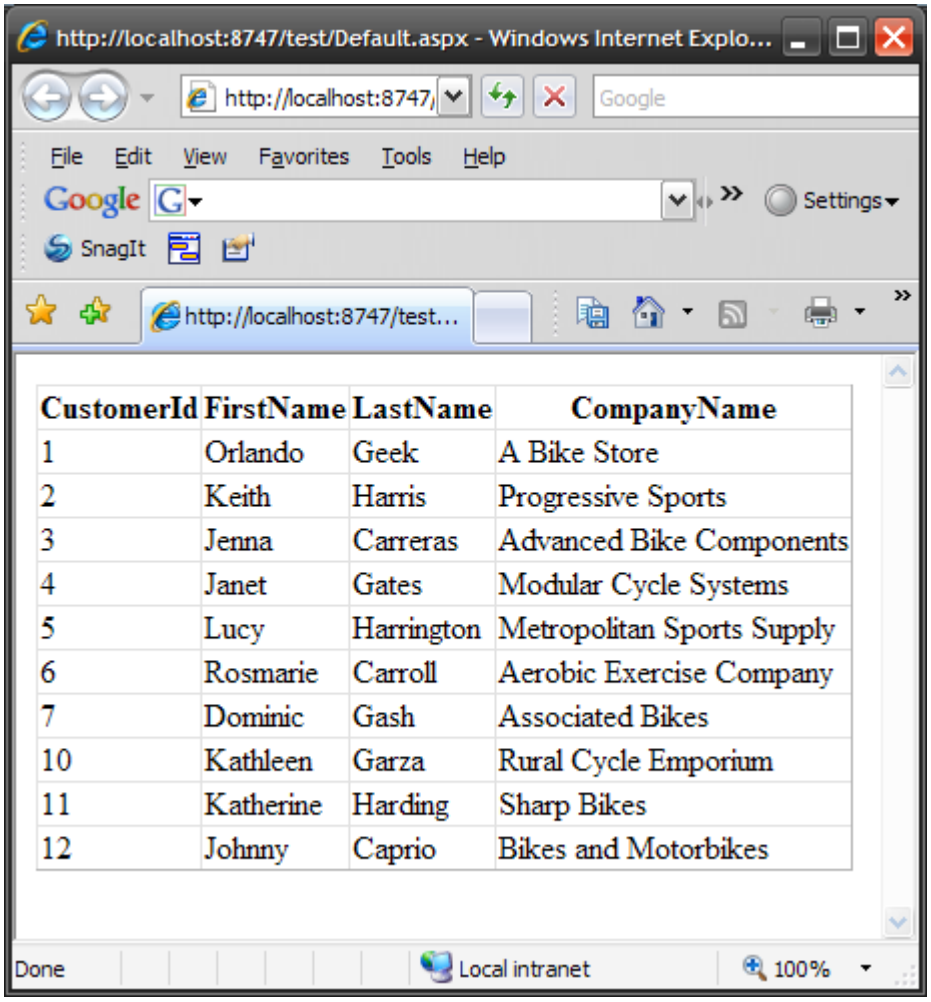
Le code est simple : on crée un DataAdapter vers la base SQL, puis on crée une table en mémoire et le DataAdapter exécute la requête et remplit la table qu'on retourne.

Le code de la page ASPX est simple également :

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
    <form id="form1" runat="server">
    <div>
        <asp:GridView ID="GridView1"
            runat="server"
            DataSourceID="maDataSource"
           />

        <asp:ObjectDataSource ID="maDataSource"
            runat="server"
            TypeName="Client"
            SelectMethod="selectClients"
            />
    </div>
    </form>
</body>
</html>
```

On a juste notre GridView qui utilise notre ObjectDataSource vu précédemment avec les objets connectes.

Le résultat :



CustomerId	FirstName	LastName	CompanyName
1	Orlando	Geek	A Bike Store
2	Keith	Harris	Progressive Sports
3	Jenna	Carreras	Advanced Bike Components
4	Janet	Gates	Modular Cycle Systems
5	Lucy	Harrington	Metropolitan Sports Supply
6	Rosmarie	Carroll	Aerobic Exercise Company
7	Dominic	Gash	Associated Bikes
10	Kathleen	Garza	Rural Cycle Emporium
11	Katherine	Harding	Sharp Bikes
12	Johnny	Caprio	Bikes and Motorbikes

Pour information, la connexion a la base de données se fait lors du "Fill". Le DataAdapter se charge de la vérification que la connexion s'est bien faite, vous n'avez donc pas besoin d'encadrer le Fill avec des try catch. Notez cependant que, une fois le Fill termine, le DataAdapter va refermer la connexion et la reouvrira éventuellement pour une autre opération, ce qui peut se révéler couteux en temps, vous pouvez donc, si vous avez d'autres opérations a faire avec la base utiliser plutôt cette technique :

```
using (SqlConnexion cnx = new SqlConnexion(…)
{
      cnx.Open();
      SqlDataAdapter adapter = new SqlDataAdapter(requete, cnx);
      adapter.Fill(table);
      . . .
      cnx.close();
}
```

Au lieu de passer en paramètre une chaine de connexion au DataAdapter, vous lui passez directement une connexion active. Il l'utilisera au lieu d'en ouvrir une (et évidemment ne la fermera pas a la fin du Fill). Ainsi, avec une seule connexion, vous pouvez faire plusieurs operations.

Comment marche le DataAdapter ? C'est assez simple : il fait la liaison entre la base et les objets en mémoire. Il fait cette liaison par l'intermédiaire de commandes SQL select, update, insert et Delete. Quand on lui demande de lire des données, il utilise sa commande SQL select (qu'on lui a passe en paramètre lors de sa création). Quand on lui demande de mettre a jour la base de données en fonction de la table qui est en mémoire, il va comparer la base de données avec la table et envoyer des séries de commandes insert, Delete ou update pour modifier la base de manière a ce qu'elle soit identique a la table en mémoire.
Les commandes insert, update et delete sont créées automatiquement avec l'objet SqlCommandBuilder.

Exemple :

```
DataTable table = new DataTable();

using (SqlConnexion cnx = new SqlConnexion(ma_chaine_de_connexion)
{
      cnx.Open();
      SqlDataAdapter adapter = new SqlDataAdapter("SELECT col1, col2,
      col3 FROM table", cnx);
      adapter.Fill(table);
      table.Rows[0]["col2"] = "nouvelle valeur";
      SqlCommandBuilder cmdBuilder = new SqlCommandBuilder(adapter);
      adapter.Update(table);
      cnx.close();
}
```

Qu'avons-nous fait ?

1. Creation d'une DataTable
2. Creation d'une connexion sur la base
3. Ouverture de la connexion
4. Creation d'un DataAdapter avec cette connexion. Ce DataAdapter est initialise avec une requête SQL pour lire les données dans la base
5. Remplissage d'une table avec ces données
6. Modification d'une donnée dans la table (ligne 0, colonne 1)
7. Creation des commandes insert, delete, update pour le DataAdapter
8. Le DataAdapter va mettre a jour la base en fonction du contenu de la table
9. Fermeture de la connexion.

On voit bien que ces objets sont plus puissants que les objets connectes, ils vont se charger de la mise a jour automatiquement sans quasiment aucune opération de notre part. Ceci se fait évidemment au détriment de la vitesse d'exécution et de la mémoire utilisée pour ces opérations.

11.2.2 L'objet DataTable

L'objet DataTable que nous venons de découvrir est une table de base de données mais située en mémoire. Vous pouvez ajouter les lignes a cette table automatiquement avec le DataAdapter comme nous venons de le voir (avec sa méthode Fill) ou a partir de données XML, mais également manuellement en parcourant un DataReader. Cet objet est très complet, c'est plus qu'un simple tableau en mémoire.
Commençons par créer a la main une table et remplissons la, puis elle sera renvoyée au GridView qui l'affichera :

```
public class Client
{
    public DataTable selectClients()
    {
        DataTable table = new DataTable();
        table.Columns.Add("Id", typeof(int));
        table.Columns.Add("Nom", typeof(string));
        table.Columns.Add("Prenom", typeof(String));
        table.Columns["Id"].AutoIncrement = true;

        DataRow ligne = table.NewRow();
        ligne["Nom"] = "Bon";
        ligne["Prenom"] = "Jean";
        table.Rows.Add(ligne);

        ligne = table.NewRow();
        ligne["Nom"] = "Aimar";
        ligne["Prenom"] = "Jean";
        table.Rows.Add(ligne);

        return table;
    }
}
```

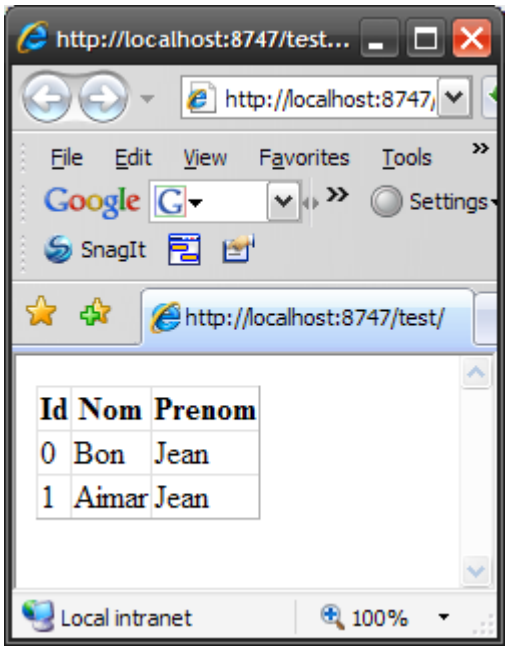
Une DataTable est composée de lignes (DataRow) accessibles par la propriété Rows de la DataTable (table.Row[4] renvoie la 5eme ligne de la table) et de colonnes. Pour accéder a une donnée d'une colonne dans une ligne particulière, on utilisera la syntaxe : table.Rows[numero_ligne]["nom_colonne"]. A la place du nom de la colonne, vous pouvez également utiliser son numéro.

Pour utiliser la table, nous commençons donc par créer des colonnes en indiquant leur nom et leur type, puis nous génèrerons une nouvelle ligne a partir de la table, nous l'initialisons et l'ajoutons a la table, et ce, pour nos deux clients.
La DataTable a des propriétés identiques a ce que peut avoir une table de base de données. On peut donc, comme on l'a fait ici :

```
table.Columns["Id"].AutoIncrement = true;
```

indiquer que la colonne "id" est une colonne dont le contenu va s'incrémenter automatiquement a chaque ligne ajoutée. (La première ligne aura pour valeur 0, puis 1, etc… On peut également spécifier la valeur de départ et le pas d'incrémentation).

Le résultat :



Faire une requête dans la DataTable

Tout comme une "vraie" table de base de données, on peut faire des requêtes dans une DataTable. Malheureusement, on ne peut pas utiliser SQL. A la place, il y a une série de méthodes qui sont, en définitive, moins intuitives et complètes que SQL.
Un select dans une DataTable renvoie un tableau de DataRow.

Si par exemple, on veut renvoyer dans notre GridView la liste des clients dont le nom commence par "A" :

```
public class Client
{
    public DataRow[] selectClients()
    {
      String cnxString =
      ConfigurationManager.ConnectionStrings["test"].ConnectionString;
      SqlDataAdapter adapter = new SqlDataAdapter("SELECT  CustomerId, FirstName,
LastName, CompanyName FROM SalesLT.Customer", cnxString);
        DataTable table = new DataTable();
        adapter.Fill(table);

        DataRow[] select = table.Select("LastName LIKE 'A*'",
                                        "LastName");
        return select;
    }
}
```

On lit tous les clients de la table dans ta table "table", puis on fait un select dessus avec la méthode "Select". Cette méthode retourne un tableau de lignes correspondant au résultat de la requête.
On renvoie ce tableau au DataGrid.

Malheureusement, le DataGrid ne sait pas comment afficher un tableau de DataRow (il sait le faire a partir d'une table, car il peut récupérer le schéma de la table, mais pas a partir de lignes seules). Il nous faut donc modifier (un peu lourdement a mon gout) le GridView :

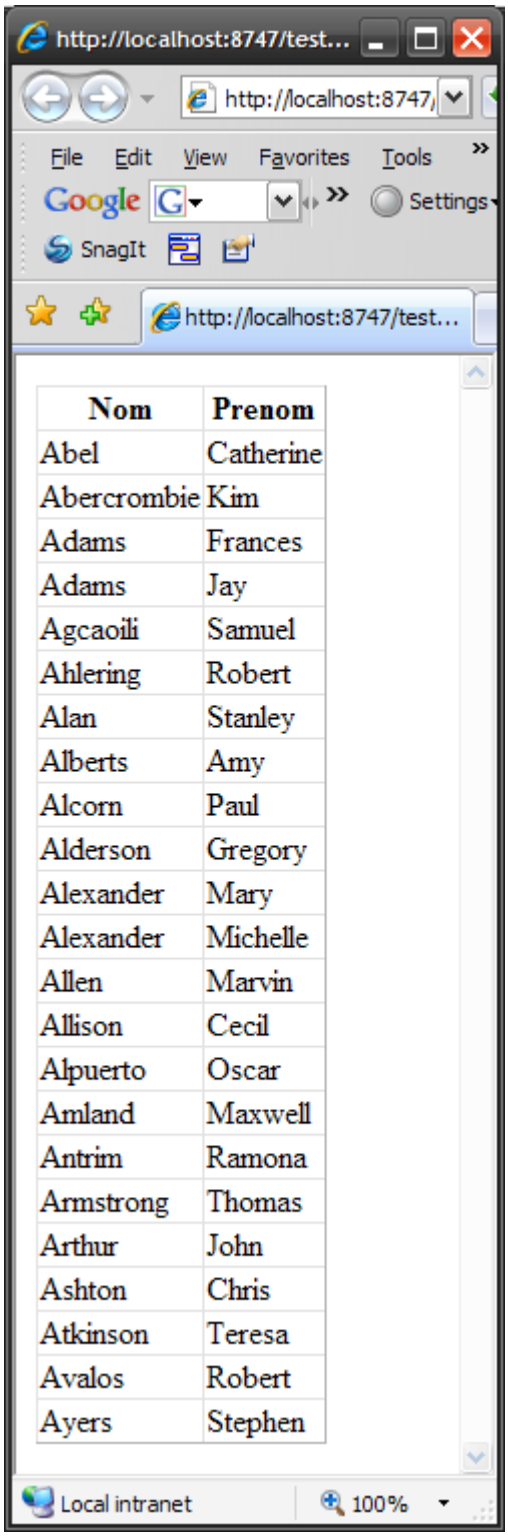
```
<asp:GridView ID="GridView1"
            runat="server"
            DataSourceID="maDataSource"
            AutoGenerateColumns="false">
  <Columns>
    <asp:TemplateField HeaderText="Nom">
      <ItemTemplate>
        <%#((DataRow)Container.DataItem)["LastName"] %>
      </ItemTemplate>
    </asp:TemplateField>

    <asp:TemplateField HeaderText="Prenom">
      <ItemTemplate>
        <%#((DataRow)Container.DataItem)["FirstName"] %>
      </ItemTemplate>
    </asp:TemplateField>
  </Columns>
</asp:GridView>
```

Nous allons devoir afficher le contenu du GridView avec des templates et forcer un cast sur un objet DataRow pour chaque ligne. Quand le GridView reçoit ses données lors du DataBind, il ne sait pas quelle est la nature de chaque élément du tableau. Il faut donc faire ce cast pour le forcer a reconnaitre un DataRow, puis aller chercher la colonne désirée.
Vous devrez en plus ajouter la ligne suivante en début de page :

```
<%@ Import Namespace="System.Data" %>
```

Car par défaut, la page ASPX ne connait pas le type DataRow.

Le résultat :



Nom	Prenom
Abel	Catherine
Abercrombie	Kim
Adams	Frances
Adams	Jay
Agcaoili	Samuel
Ahlering	Robert
Alan	Stanley
Alberts	Amy
Alcorn	Paul
Alderson	Gregory
Alexander	Mary
Alexander	Michelle
Allen	Marvin
Allison	Cecil
Alpuerto	Oscar
Amland	Maxwell
Antrim	Ramona
Armstrong	Thomas
Arthur	John
Ashton	Chris
Atkinson	Teresa
Avalos	Robert
Ayers	Stephen

Plutôt lourd pour arriver a ce résultat, mais c'était pour l'exemple.

Chaque ligne de la table (chaque DataRow donc) possède une propriété nommée RowState qui indique dans quel état est la ligne. A chaque fois qu'une modification est faite dans la ligne, l'etat est modifie. Les états possibles sont :

- **Unchanged :** La ligne n'a pas été modifiée depuis sa création
- **Added :** La ligne a été ajoutée a la table
- **Modified :** La ligne a été modifiée
- **Deleted :** La ligne a été supprimée
- **Detached :** La ligne a été crée a partir d'une table mais non ajoutée a la table

A quoi cela sert-il ? Cela va notamment servir au DataAdapter pour savoir quelles modifications il doit faire dans la base de données. Cela peut également vous servir pour annuler des modifications faites dans la table (une fonction Undo par exemple).

Pour valider les modifications, on appellera la méthode `AcceptChanges()` de la table.
Pour annuler les modifications, on appelera la méthode `RejectChanges()` de la table. L'annulation replacera la table dans l'etat ou elle était lors du dernier appel a "AcceptChanges()" (ou lors de sa création si AcceptChanges() n'a jamais été appelé précédemment).
Surtout ! n'oubliez pas d'appeler la méthode AcceptChanges() après avoir renvoyé les données vers la base de données :

```
adapter.Update(table);
table.AcceptChanges();
```

La première ligne va demander au DataAdapter de modifier la base de données en fonction des modifications présentes dans la table "table". Si vous omettez de faire un AcceptChanges() juste après, le prochain update dans la base risque d'écraser des données modifiées entre temps dans la base avec des données de la table qui seraient considérées a tort comme étant récemment modifiées.

11.2.3 L'objet DataView

La DataView est un objet qui représente une vue. Pour ceux qui l'auraient oublie, une vue est une sorte de table "virtuelle" créée a partir d'une requête sur une table (ou une autre vue). Elle peut ensuite être manipulée comme une vraie table.
Pour créer une vue très rapidement a partir d'une DataTable, il suffit d'utiliser la propriété DefaultView de la table. Elle renvoie une vue de toutes les lignes de la table.
On peut créer une DataView a partir de données filtrées ou triées d'une DataTable. C'est très simple :

```
DataView vue = new DataView(table, "LastName LIKE 'A*'", "LastName ASC",
DataViewRowState.CurrentRows);
```

Cette ligne va créer une vue (DataView) a partir de la DataTable "table", au passage, elle ne va récupérer que les lignes dont la colonne "LastName" commence par "A", puis trier les réponses sur la colonne "LastName" de façon ascendante.

Le dernier paramètre indique le type de données qu'on veut récupérer dans la DataTable : CurrentRows indique de récupérer uniquement les lignes qui sont présentes dans la table. Parmi les valeurs possibles de ce paramètre, on peut citer :

- Added : récupérer uniquement les nouvelles lignes
- Deleted : les lignes supprimées
- Unchanged : celles qui n'ont pas change
- None : toutes

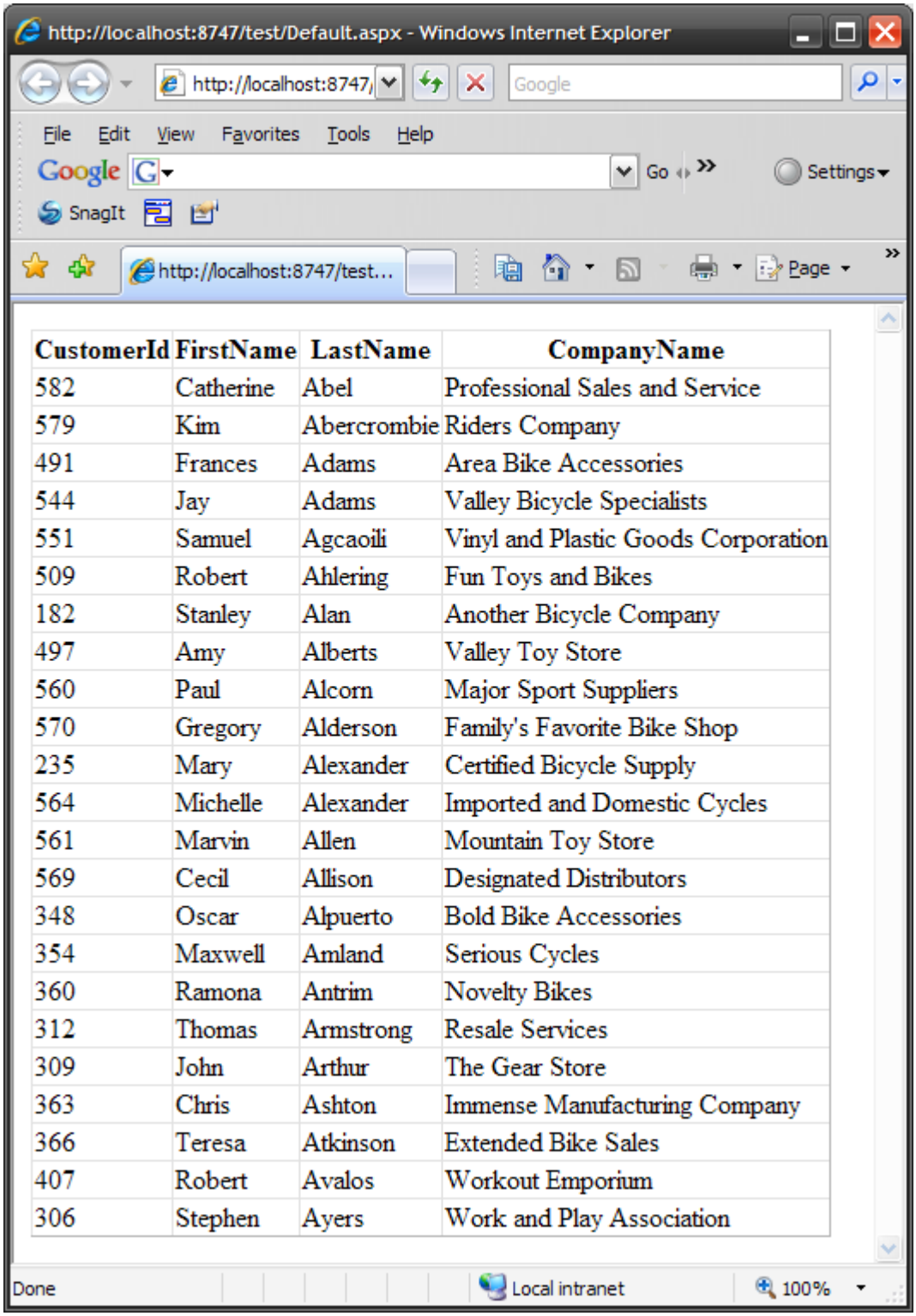
Si on reprend notre classe :

```
    public DataView selectClients()
    {
       String cnxString =
      ConfigurationManager.ConnectionStrings["test"].ConnectionString;
      SqlDataAdapter adapter = new SqlDataAdapter("SELECT  CustomerId, FirstName,
LastName, CompanyName FROM SalesLT.Customer", cnxString);
      DataTable table = new DataTable();
      adapter.Fill(table);
      DataView vue = new DataView(table, "LastName LIKE 'A*'",
      "LastName ASC", DataViewRowState.CurrentRows);
      return vue;
    }
```

On crée donc une vue a partir de la table et on renvoie cette vue au GridView. Heureusement, le GridView sait comment afficher des données provenant d'une vue, on peut donc supprimer tout ce qu'on a mis pour afficher le GridView a partir du tableau de DataRow :

```
   <asp:GridView ID="GridView1"
            runat="server"
            DataSourceID="maDataSource"
   />
```

Le résultat :



CustomerId	FirstName	LastName	CompanyName
582	Catherine	Abel	Professional Sales and Service
579	Kim	Abercrombie	Riders Company
491	Frances	Adams	Area Bike Accessories
544	Jay	Adams	Valley Bicycle Specialists
551	Samuel	Agcaoili	Vinyl and Plastic Goods Corporation
509	Robert	Ahlering	Fun Toys and Bikes
182	Stanley	Alan	Another Bicycle Company
497	Amy	Alberts	Valley Toy Store
560	Paul	Alcorn	Major Sport Suppliers
570	Gregory	Alderson	Family's Favorite Bike Shop
235	Mary	Alexander	Certified Bicycle Supply
564	Michelle	Alexander	Imported and Domestic Cycles
561	Marvin	Allen	Mountain Toy Store
569	Cecil	Allison	Designated Distributors
348	Oscar	Alpuerto	Bold Bike Accessories
354	Maxwell	Amland	Serious Cycles
360	Ramona	Antrim	Novelty Bikes
312	Thomas	Armstrong	Resale Services
309	John	Arthur	The Gear Store
363	Chris	Ashton	Immense Manufacturing Company
366	Teresa	Atkinson	Extended Bike Sales
407	Robert	Avalos	Workout Emporium
306	Stephen	Ayers	Work and Play Association

11.2.4 L'objet DataSet

Un objet DataSet représente une base de données complète en mémoire. Un DataSet peut donc contenir plusieurs DataTables ainsi que des relations parent / enfants entre ces tables.
Sachez seulement que, comme la DataTable, on peut modifier le contenu des tables de ce DataSet puis, demander une mise a jour globale de la base de données en fonction de toutes les modifications qui ont eu lieu dans toutes les tables.
Le DataSet étant un objet très riche (et donc complexe), nous ne nous étendrons pas d'avantage sur son utilisation ici.

12 Accéder aux données via LINQ

Une des principales nouveautés du Framework 3.5 est l'apparition de la technologie LINQ to SQL. Cette technologie, disponible hélas que dans le Framework .NET, révolutionne la façon d'accéder aux bases de données.

Jusqu'à présent, la seule façon d'accéder aux données d'une base passait par l'utilisation du langage SQL, langage dont la syntaxe et les objets manipules sont plutôt éloignes des objets utilises dans les langages modernes. D'un cote on a des tables, des lignes et des colonnes, de l'autre, des classes, des méthodes et des propriétés. L'idéal serait de pouvoir manipuler les données des bases de données directement comme des classes d'objets.

Des librairies et frameworks existent pour tenter de faire le pont entre les deux (Hibernate par exemple). A la différence de ces librairies, LINQ to SQL est directement intégré non seulement au Framework .Net, mais également au langage C# lui-même. Interroger une base de données (ou n'importe quelle structure) via LINQ ne se fait pas en appelant des fonctions mais en utilisant directement des mots clefs du langage.

Avant de plonger dans le vif du sujet, revenons rapidement sur les caractéristiques de C# qui vont être utilisées avec LINQ.

12.1 Rappel sur les nouveautés de C#

12.1.1 Les propriétés automatiques

```
public class Class1
{
    // propriété automatique

    public int PropAuto { get; set; }

    // propriété "normale"

    private decimal _PropNormale;
    public decimal PropNormale
        {
        get { return _PropNormale; }
        set { _PropNormale = value; }
        }
}
```

Regardez bien la différence entre la propriété PropAuto et la propriété PropNormale. PropNormale est définie de la façon dont on le faisait avant C#3.5, c'est-à-dire avec une variable privée qui contient la valeur de la propriété et un getter et un setter pour initialiser cette variable.
Avec C#3.5, il suffit d'indiquer les mots clef get et set et lors de la compilation, la variable privée est créée ainsi que le getter et le setter.

12.1.2 Les initialiseurs

Supposons cette classe :

```
public class Class1
{
    public int Prop1 { get; set; }
    public int Prop2 { get; set; }
    public int Prop3 { get; set; }
    public int Prop4 { get; set; }
}
```

La façon classique d'initialiser une telle classe est comme suit :

```
Class1 cl = new Classe1();
cl.Prop1 = 10;
cl.Prop2 = "string 1";
cl.Prop3 = 20;
cl.Prop4 = "string 2";
```

ce qui est, convenez le un peu long. Vous pouvez également créer un constructeur attendant ces 4 paramètres. Mais pourquoi faire a la main ce que C# fait seul ?

il suffit d'utiliser les initialiseurs !

```
Class1 cl = new Class1 {Prop1 = 10, Prop2 = "string 1", Prop3 = 20, Prop4 = "string
2"};
```

Vous indiquez dans lors du new quelles sont les propriétés initialisées avec quelles valeurs et tout se fait automatiquement.

12.1.3 L'inférence de type

L'inférence de type apporte a C# des caractéristiques que l'on retrouve dans JavaScript.

Le type de la variable va être détermine en fonction de la valeur qu'on lui affecte.

```
var hello = "Hello";
var toto = 10;
```

Hello est une chaine de caractères car on lui a affecte "Hello" et toto est un entier. (Mais une fois affecte, contrairement a JavaScript, le type ne peut plus changer).

12.1.4 Les types anonymes

Les types anonymes vous permettent de créer l'instance d'une classe a la volée et d'en initialiser son contenu :

```
var client = new {nom = "Dupont", prenom = "Jean"};
var nom = client.nom;
```

Vous noterez que client n'a pas de type, vous connaissez juste le nom de cette variable, son type est anonyme.

12.1.5 Les lambda expressions

Les lambda expressions, introduites avec C# 3.5 permette de définir des méthodes de façon très succincte.

Supposons que vous vouliez, par programmation exécuter une fonction lorsqu'on clique sur un bouton. La méthode classique, que je montre ici au cas où vous la rencontreriez dans des exemples, consiste à créer une fonction puis un handler a cette fonction et enfin affecter ce handler à l'évènement :

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
void Page_Init()
    {
    btn.Click += new EventHandler(btn_Click);
    }

void btn_Click(object sender, EventArgs e)
    {
    lblResult.Text = DateTime.Now.ToString();
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<body>
<form id="form1" runat="server">
    <asp:Button id="btn" Text="OK" Runat="server" />
    <asp:Label id="lbl" Runat="server" />
</form>
</body>
</html>
```

Un peu long non ?

C#2.0 a introduit la notion de Delegate, qui permet de créer une fonction anonyme et de la déclarer "au vol" :

```
<script runat="server">
void Page_Init()
    {
    btn.Click += delegate(object sender, EventArgs e)
        {
        lblResult.Text = DateTime.Now.ToString();
        };

</script>
```

Plus de fonction et plus de handler.

Les lambda expression vont encore plus loin dans la concision.

```
void Page_Init()
{
btn.Click += ( sender, e ) => lblResult.Text = DateTime.Now.ToString();
}
```

La lambda expression :

```
( sender, e ) => lblResult.Text = DateTime.Now.ToString();
```

Utilise l'operateur => qui signifie "va dans" pour séparer la liste des paramètres du corps de la fonction. Vous pouvez également (bien que ce ne soit pas obligatoire) indiquer le type des paramètres :

```
(Object sender, EventArgs e) => lblResult.Text = DateTime.Now.ToString();
```

Notez également que s'il n'y a qu'un seul paramètre, les parenthèses ne sont pas obligatoires.

12.1.6 Etendre les classes

Cette fonctionnalité a été reprise de JavaScript ou vous pouvez (comme vous le savez n'est-ce-pas ?) ajouter des méthodes a n'importe quel objet sans être oblige de créer une nouvelle classe dérivant de celle a laquelle vous voulez ajouter des méthodes.

Supposons que vous vouliez ajouter une méthode HtmlEncode à la classe String. Vous pourriez créer une nouvelle classe StringWithHtmlEncode qui dériverait de String, mais avouez que ce n'est pas élégant. Grace a l'extension des méthodes, nous allons ajouter directement cette méthode à la classe String.

Pour cela, il suffit de définir la fonction HtmlEncode comme suit :

```
public static class MonExtension
    {
        public static string HtmlEncode( this string str )
        {
            return System.Web.HttpUtility.HtmlEncode( str );
        }
    }
```

La fonction est définie dans une classe statique. La seule différence avec une méthode "normale" est le mot clef this utilise dans le passage des paramètres qui indique a quel type la méthode va être ajoutée.

Il suffit ensuite d'appeler la nouvelle méthode ainsi :

```
String s = "<b>ceci est du HTML</b>";
var s2 = s.HtmlEncode();
```

12.2 LINQ

LINQ to SQL est une extension de LINQ. LINQ signifie "Language Integrated Query". Il s'agit d'un ensemble de nouveaux mots clefs ajoutes a C# (et aussi a VB.NET, mais je ne me souviens plus de ce qu'est VB.NET).
Voici un petit exemple d'une requête LINQ :

```
var mots = new List<String> { "machin", "truc", "bidule" };
var resultats = from w in mots
                where w.Contains("u")
                select w;
```

Ca a l'air un peu étrange (ca l'est). De quoi s'agit-il ?

La première ligne initialise une liste de chaines de caractères nommée "mots". La seconde ligne est une requête Linq.
A première vue, la requête LINQ ressemble beaucoup a une requête SQL mais a l'envers. Celle-ci récupère tous les mots de la liste qui contiennent la lettre "u".

Vous pouvez faire des requêtes LINQ sur tous les objets qui implémentent l'interface IEnumerable (c'est-à-dire beaucoup de choses : tableaux, listes, tables, etc…)

Les mots clefs utilises dans une requête LINQ sont les suivants :

- **from :** Indique quelle est la source des données et la variable utilisée pour itérer dans la source de données.
- **where :** Permet de filtrer le résultat de la requête.
- **select :** indique quels seront les éléments qui seront utilises dans le résultat de la requête.
- **group :** Permet de regrouper les valeurs selon une clef commune.
- **into :** Sauve les résultats d'un regroupement ou d'une jointure dans une variable temporaire.
- **orderby :** Trie les résultats de la requête sur un critère de façon ascendante ou descendante.
- **join :** Fait une jointure entre deux sources sur une clef commune.
- **let :** Cree une variable temporaire qui contient les résultats de sous-requêtes.

Construire une requête LINQ est un peu comme construire une requête SQL a l'envers. On commence par spécifier la clause from qui indique ou on veut aller chercher les données, puis, si on veut, on peut spécifier une clause where pour filtrer les données. A la fin, on spécifie une clause select qui définit a quoi les données renvoyées vont ressembler.

Pour information, il y a deux façons d'écrire une requête LINQ. La simple et la moins simple. La simple est celle que nous venons de voir. La moins simple utilise les lambda expressions. (De toute façon, si vous utilisez la première façon, elle sera automatiquement traduite en lambda expression par le compilateur).

Par exemple :

```
var resultats = from w in mots
                where w.Contains("u")
```

```
                    select w;
```

est traduite en lambda expression par :

```
var resultats = mots.Where(w => w.Contains("u")).Select(w => w);
```

Ca peut paraitre barbare mais il y a un avantage a utiliser cette notation. A chaque étape de l'évaluation de l'expression, LINQ retourne un objet implémentant IEnumerable. (mots implémente IEnumerable, mots.Where(. . .) est IEnumerable, mots.Where(. . .).Select(. . .) est IEnumerable et ainsi de suite).
Et c'est bien intéressant car l'interface IEnumerable a des méthodes bien utiles pour filtrer, calculer, re sélectionner en cours de route. Parmi ces méthodes :

- **Aggregate() :** Appelle une fonction sur chaque élément de l'objet
- **Average() :** calcule la moyenne des elements
- **Count() :** renvoie le nombre d'elements
- **Distinct() :** renvoie les elements distincts
- **Max() :** renvoie la valeur maximale des elements
- **Min() :** renvoie la valeur minimale des elements
- **Select() :** (utilise dans l'exemple) – Renvoie certains elements ou certaines propriétés des éléments
- **Skip() :** Saute un certains nombre d'elements et renvoie le reste
- **Take() :** Renvoie un certain nombre d'elements de la séquence
- **Where() :** filtre les elements (comme dans l'exemple).

On pourrait donc faire :

```
var resultats = mots.Where(w => w.Contains("u")).Where(w =>
w.Contains("z")).Skip(4).Take(3).Select(w => w);
```

qui signifie : dans la liste des mots prendre ceux qui contiennent la lettre "u", puis dans ces mots, ne conserver que ceux qui contiennent la lettre "z", puis sauter les 4 premiers mots et n'en conserver que les 3 premiers. Ouf ! Essayez de faire ca en une ligne avec un autre langage !!

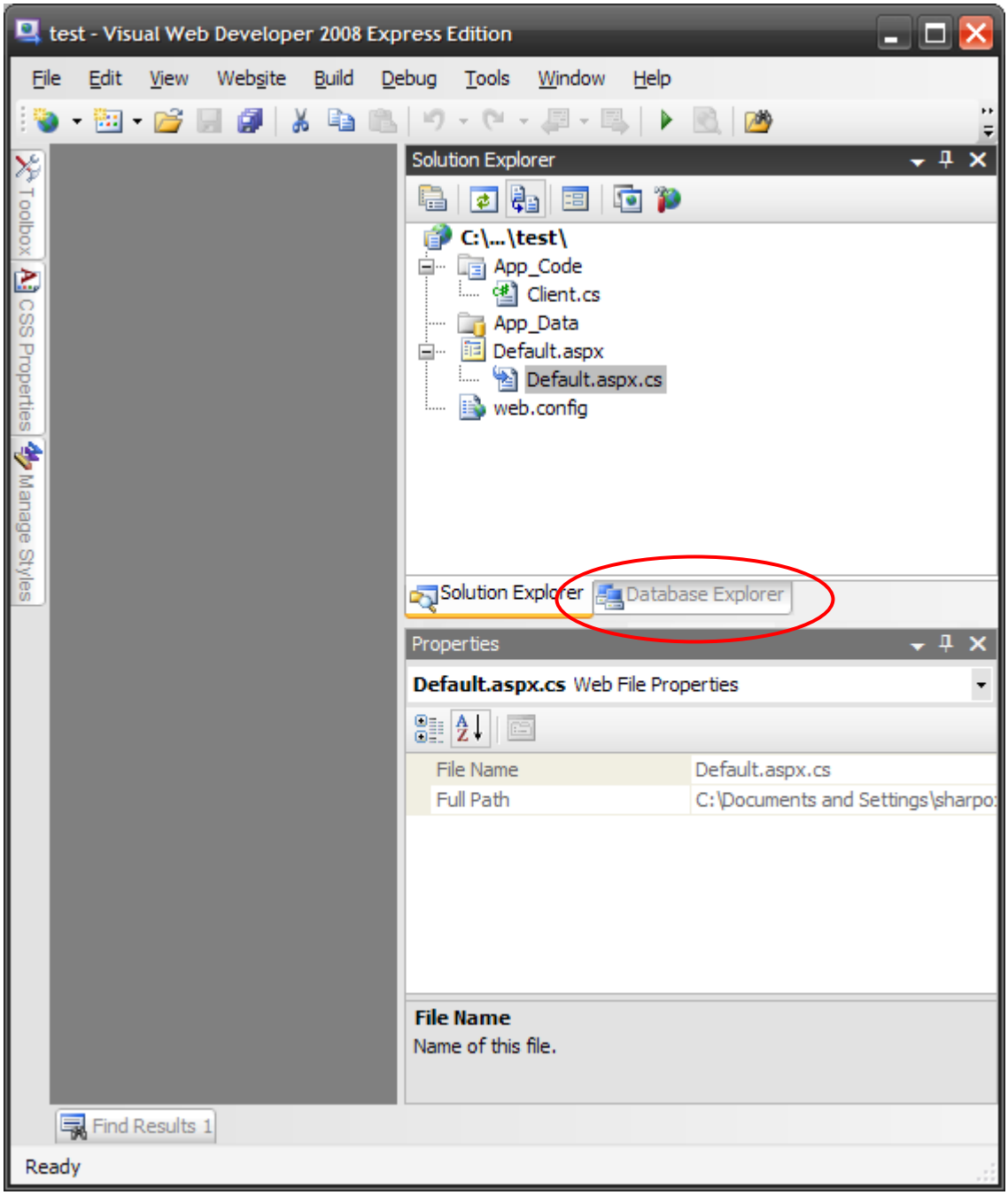
Ceci est la version "générique" de LINQ. Il existe plusieurs versions de LINQ qui permettent d'interroger d'autres types de données que des objets IEnumerables. Il existe notamment LINQ to SQL qui permet d'interroger des bases de données (ce que nous allons voir maintenant). Pour information, il existe aussi LINQ to XML, LINQ to DataSet (pour faire des requêtes dans des DataSets) ou des versions de LINQ faites par d'autres sociétés que Microsoft, comme LINQ to NHibernate.

Petite précision : LINQ to SQL ne fonctionne qu'avec SQL Server (toutes versions : SQL 2000, 2005, 2008 et Express). Des societes tierces proposent cependant des versions de LINQ to SQL pour Oracle, MySQL et PostGres.

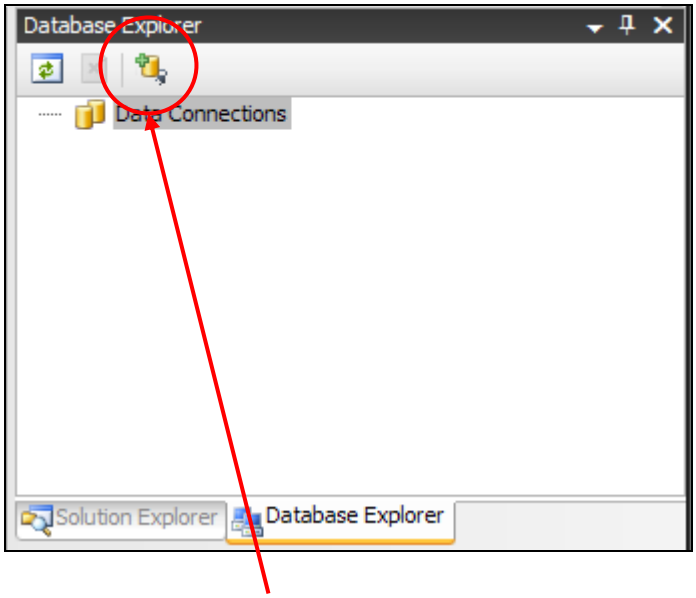
12.3 Utiliser LINQ to SQL

Pour pouvoir utiliser LINQ to SQL avec une base de données, il va falloir créer des "Entities". Ces entities sont des classes d'objets représentant les objets de la base de données. C'est par l'intermédiaire de ces entities qu'on accédera a la base de données.
Chaque classe représente les colonnes d'une table et décrit très précisément comment ces colonnes sont définies dans la base.
On peut bien sur les définir a la main, mais c'est long et pénible et on risque des oublis qui risquent de produire des erreurs bien plus tard dont on ne connaitra pas l'origine.
Fort heureusement, Visual Studio (Express et "normal") proposent des outils pour générer les entities de façon automatique : un petit drag and drop et les classes sont prêtes a être utilisées.

Nous allons voir comment faire :

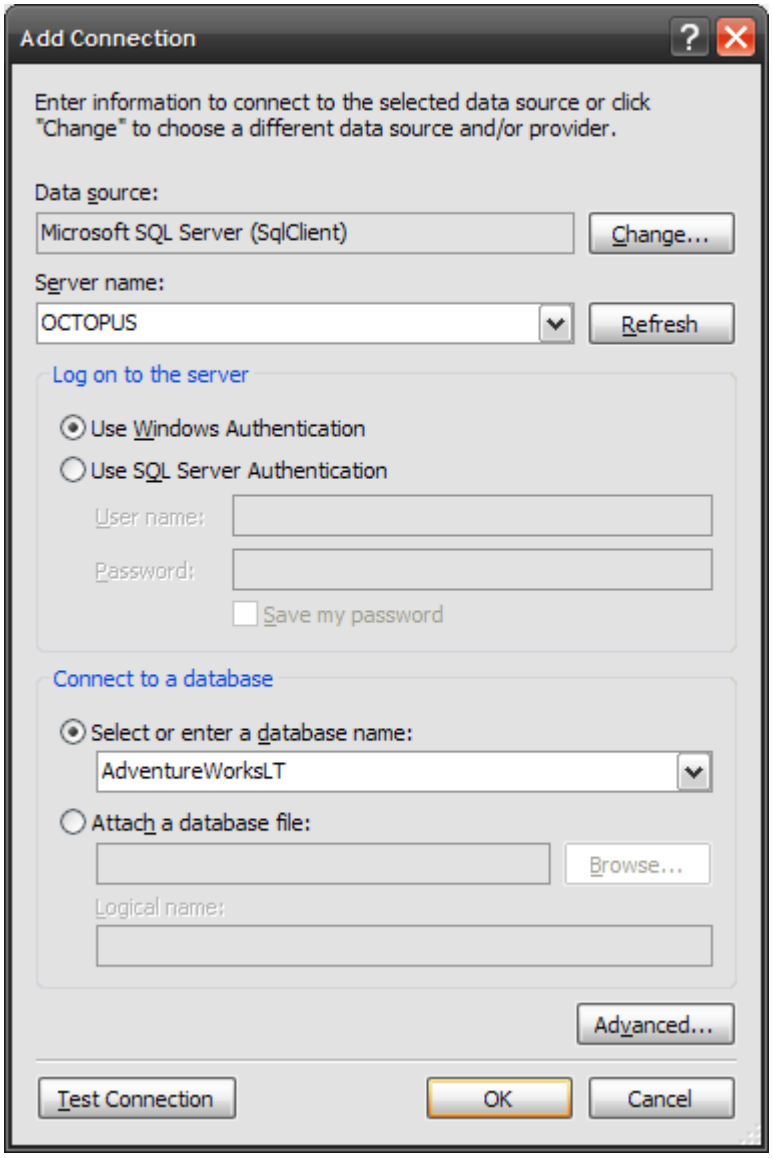

test - Visual Web Developer 2008 Express Edition
File
Edit
View
Website
Build
Debug
Tools
Window
Help
Toolbox
CSS Properties
Manage Styles
Solution Explorer
C:\...\test\
App_Code
Client.cs
App_Data
Default.aspx
Default.aspx.cs
web.config
Solution Explorer
Database Explorer
Properties
Default.aspx.cs Web File Properties
File Name
Default.aspx.cs
Full Path
C:\Documents and Settings\sharpo
File Name
Name of this file.
Find Results 1
Ready

Vous avez remarque que a droite de l'onglet "Solution Explorer", il y a un onglet "Database Explorer". Cet onglet permet de se connecter a un serveur de base de données :

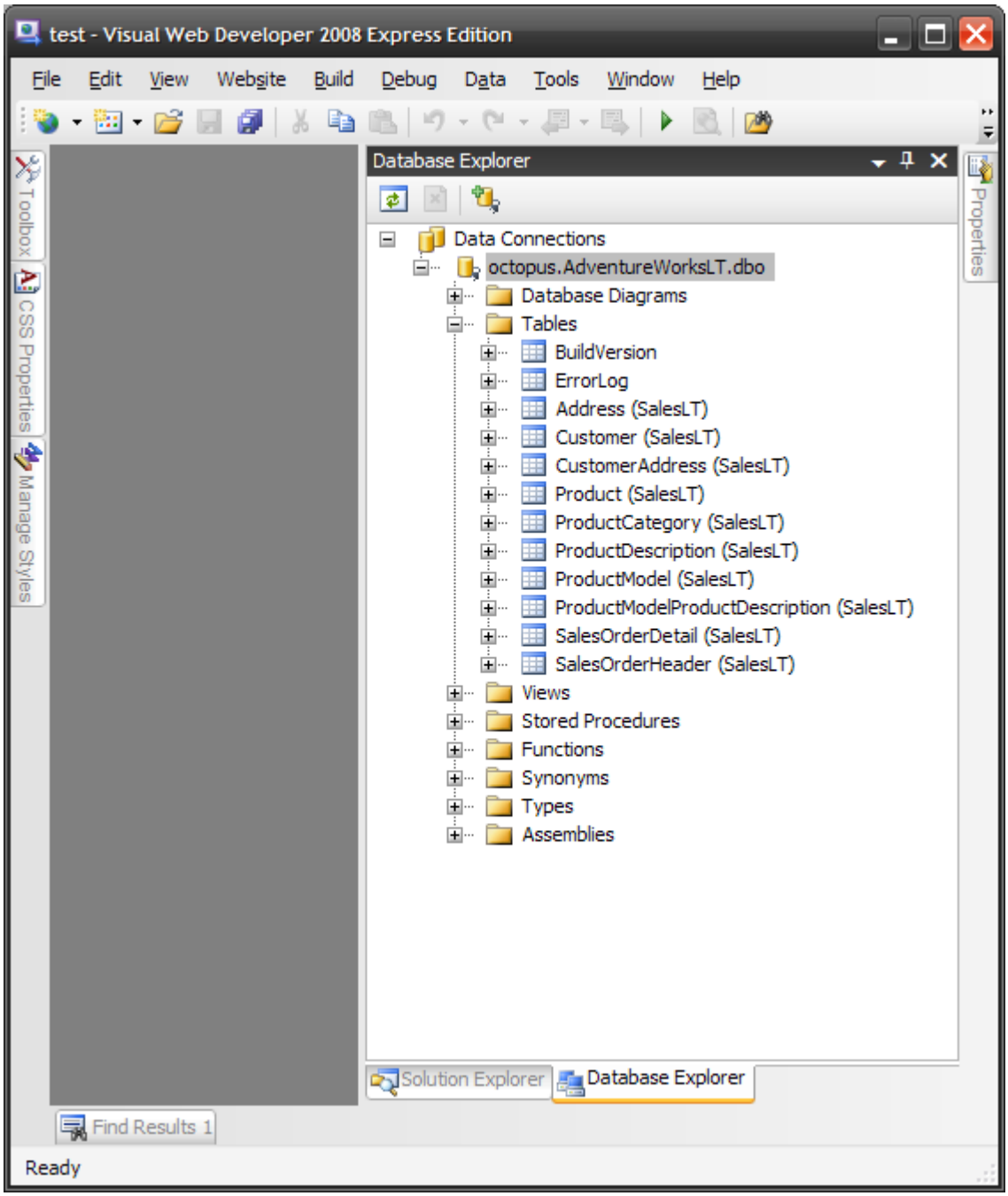


Cliquez sur ce petit icône :

Renseignez la boite de dialogue qui s'affiche (OCTOPUS ici est le nom de ma machine) :

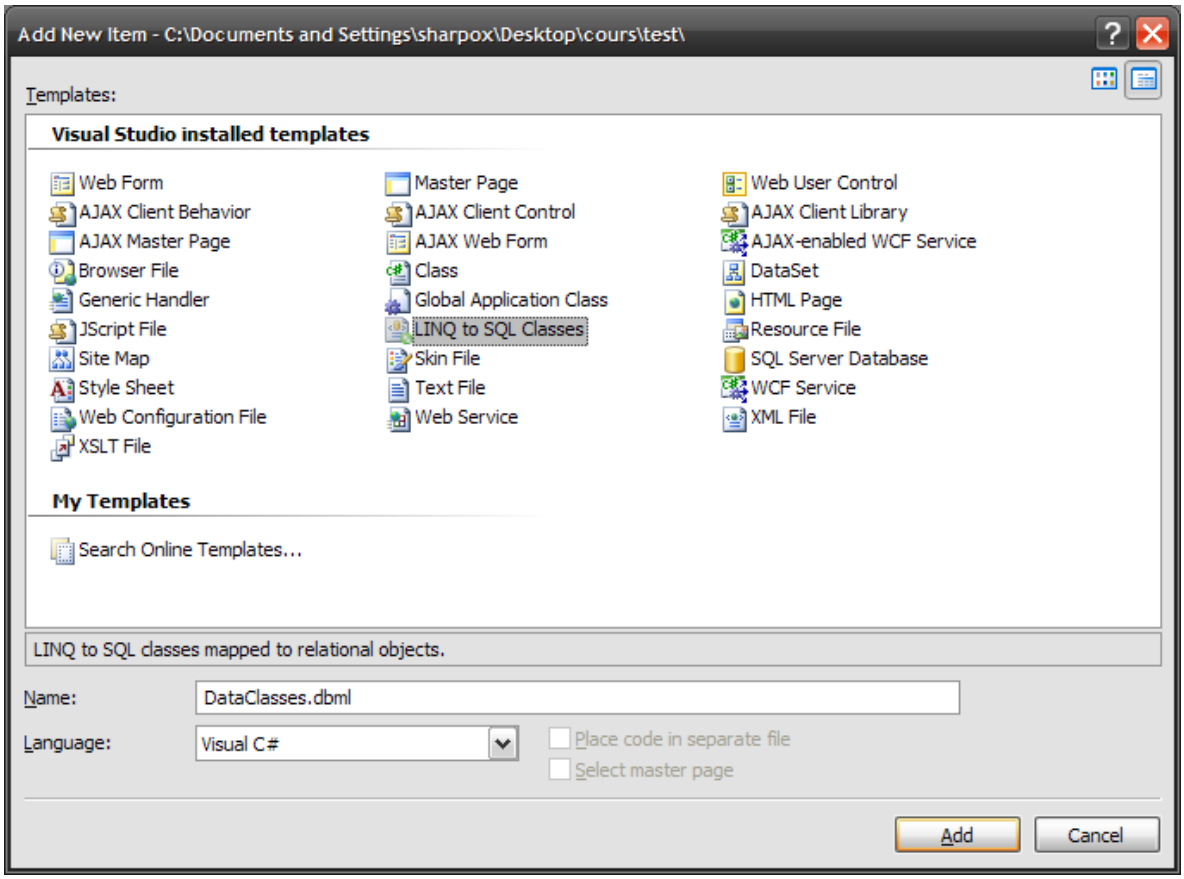


Une fois connecte, le serveur est affiche, vous pouvez afficher la liste des tables de la base sélectionnée :



OK, nos tables sont accessibles, nous allons créer les entities nécessaires a l'utilisation de la table "Customer".

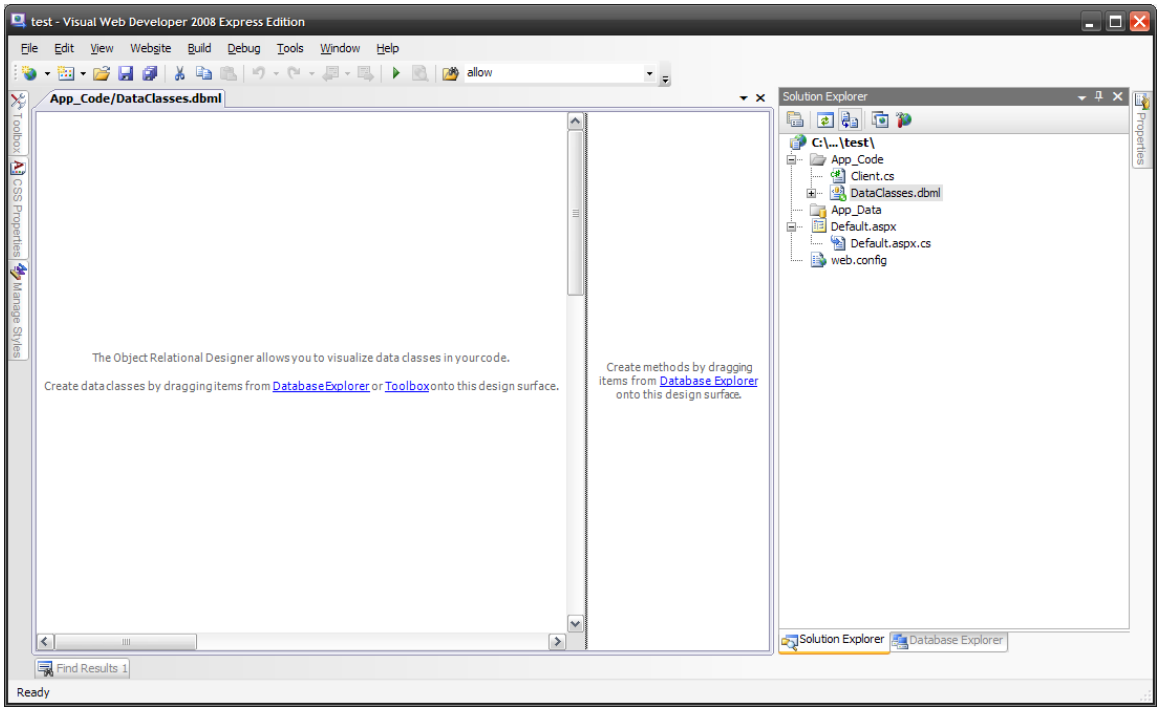
On repasse dans le Solution Explorer, on clique droit sur le projet et on choisit "Add new Item…" (Ajouter) :



On va choisir "LINQ to SQL Classes". Par défaut, la configuration des entities sera sauvée dans le fichier "DataClasses.dbml".

Conservons ce fichier. (Il sera place, comme toutes les classes et librairies "externes" dans le dossier "App_Code".

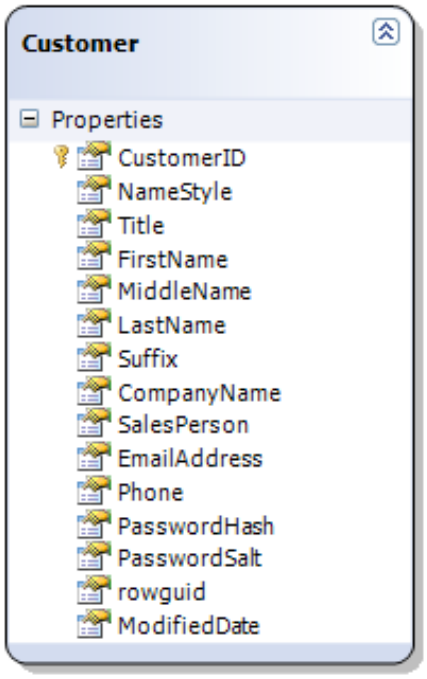
Une fois crée, double cliquez sur ce fichier :



On se retrouve devant une fenêtre divisée en deux parties. La va commencer la séance de drag and drop. Dans la partie gauche de la fenetre, on va déposer des tables : les classes correspondant a ces tables seront créées automatiquement. Dans la partie droite, on va deposer des procédures stockées (des fonctions qui s'exécutent directement sur le serveur de base de données), des méthodes correspondantes seront créées automatiquement.

On présélectionne donc le Database Explorer, puis on drag and drop la table Customer dans la partie gauche.

On se retrouve devant cet écran :



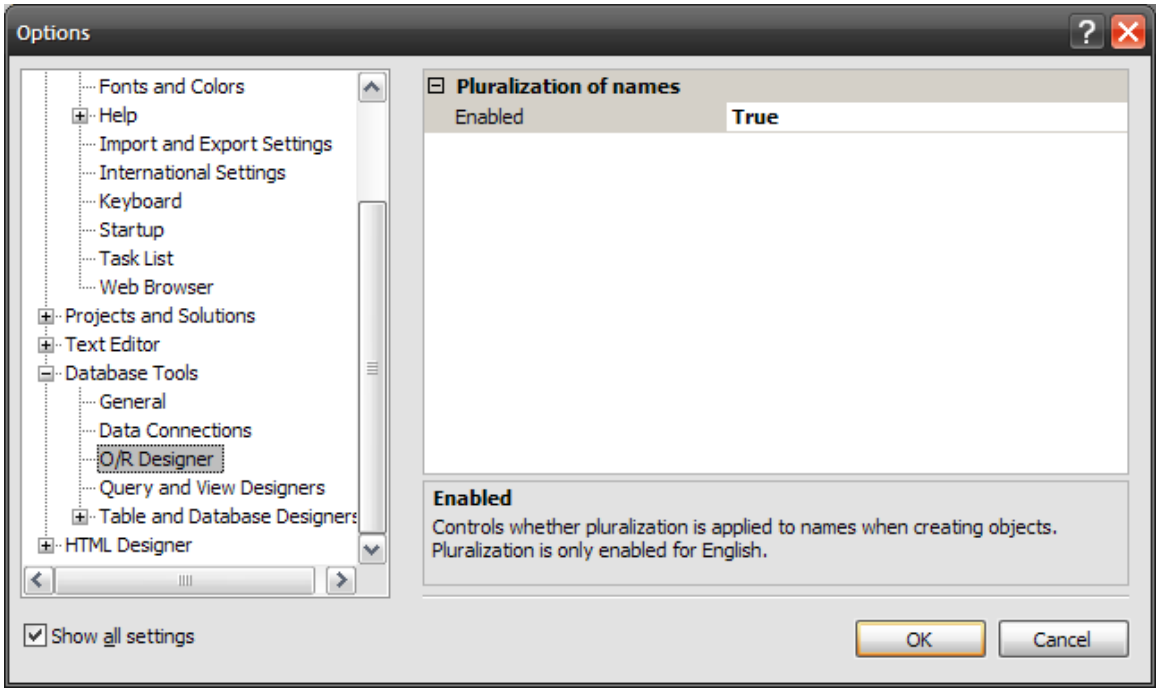
La classe "Customer" a été créée. Elle contient les mêmes champs que la table.

Cette classe peut être utilisée telle quelle, mais elle est déconnectée de la base de données. Pour accéder aux données de la base de données via cette classe, il faut passer par un DataContext.
Le DataContext est une classe qui est créée automatiquement en même temps que nos classes. Elle porte le nom qu'on a donne au fichier .dbml (ici le DataContext de cette base s'appelle DataClassesDataContext).

C'est par les propriétés de cet objet qu'on va accéder aux classes qui elles-mêmes permettent d'accéder aux données de la base.

La Classe DataClassesDataContext est la classe generee par Visual Studio. Elle permet d'acceder a toutes les tables placees dans le data explorer. Une fois qu'on cree une instance de cette classe, la connexion est faite sur la base et les objets sont accessibles via LINQ. Le DataContext possede une propriete par table. Ici la table "Customer" est accessible via une classe "Customer", cet objet "Customer" etant renvoye par la propriete "Customers" du data context.

Notez que pour acceder a la classe "Customer" (mappee sur la table "Customer"), il faut passer par la propriete "Customers" du data context (le nom de la propriete est le nom de la classe au pluriel). Pour une raison qui m'echappe, Visual Studio cree par defaut des proprietes dans le data context a partir du nom des tables au pluriel (et parfois, de facon fantaisiste, votre table "journal" sera accessible via la propriete "journals"). Pour eviter ca, allez dans le menu Tools, puis Options :



Passez le paramètre "Pluralization of names" a false et cliquez sur OK.

Ceci étant fait, nous allons nous intéresser a la façon dont on fait des requêtes LINQ.

12.4 Faire des requêtes LINQ

LINQ to SQL utilise SQL de façon sous-jacente. Vous ne le voyez jamais, mais LINQ convertit toutes ses requêtes en SQL avant de les envoyer au serveur de base de données (ce qui est normal, ce dernier ne connaissant que SQL).
Pour accéder a nos objets manipulables par LINQ, il faut avoir au préalable instancie un objet DataClassesDataContext :

```
DataClassesDataContext db = new DataClassesDataContext();
```

12.4.1 Faire un "SELECT"

Nous avons vu qu'il y a deux façons de faire une requête LINQ, soit en utilisant le mode requête, soit en utilisant les lambda expression. Nous utiliserons ces dernières.

Commençons par un bête :

```
select * from Customer
```

Il sera traduit par :

```
var query = db.Customers;
```

ou si vous voulez utiliser le mode requête :

```
var query = from c in db.Customers select c;
```

Sélectionnons des champs en particulier :

```
select FirstName, LastName, CompanyName from customers
```

ca donne :

```
var query = db.Customers.Select(c => new { c.FirstName, c.LastName, c.CompanyName });
```

ou en mode requête :

```
var query = from c in db.Customers
            select new { c.FirstName, c.LastName, c.CompanyName };
```

Dans ce cas, la valeur de query sera une liste (IEnumerable) d'objets dont les propriétés sont celles définies lors du select. Le new n'est pas la par hasard : la sélection crée un objet anonyme dont les propriétés sont les champs sélectionnes dans l'objet Customer.

12.4.2 Faire un select avec un where

Soit la requête SQL :

select FirstName, LastName from Customers where CompanyName='machin'

Sera traduit par

```
var query = db.Customers
            .Where(c => c.CompanyName == "machin"
            .Select(c => new { c.FirstName, c.LastName });
```

Ou

```
var query = from c in db.Customers
            where ( c.CompanyName == "machin" )
            select ( new { c.FirstName, c.LastName } );
```

12.4.3 Faire un select avec tri

Soit la requête SQL :

select * from Customer order by LastName, FirstName

Sera traduit par :

```
var query = db.Customers.OrderBy(c => c.LastName).ThenBy(c => c.FirstName);
```

ou

```
var query = from c in db.Customers
            orderby LastName, FirstName
            select new { c };
```

Notez que dans le 1er cas (expressions lambda), on utilise deux méthodes de l'interface IEnumerable : OrderBy pour le 1er critère de recherche, puis ThenBy pour les autres critères. Pour trier en ordre descendant, on utilisera les méthodes OrderByDescending et ThenByDescending. En mode requête, on utilisera : orderby LastName descending, FirstName descending.

12.4.4 Paginer les résultats

Paginer les résultats avec SQL est complique. LINQ offre deux méthodes qui rendent la pagination simple :
Skip() saute un certain nombre d'enregistrements renvoyés et Take() n'en récupère que certains d'entre eux.
Supposons que nous voulions récupérer les 10 enregistrements situes a partir du 30eme :

```
var query = db.Customers.Skip(30).Take(10);
```

12.4.5 Faire des jointures

Pour faire la jointure, nous allons avoir besoin d'une nouvelle table dans le DataContext. Reouvrez donc le DataClasses.dbml et ajouter la table "SalesOrderHeader" (les commandes passées par les clients).
Si on veut faire la requête SQL suivante (afficher les commandes et les noms des clients qui les ont passees) :

```
select SalesOrderID, LastName from SalesOrderHeader, Customer where
SalesOrderHeader.CustomerID = Customer.CustomerID order by LastName
```

(ou plus proprement)

```
SELECT SalesOrderHeader.SalesOrderID, Customer.LastName
FROM SalesOrderHeader
INNER JOIN Customer
ON SalesOrderHeader.CustomerID = Customer.CustomerID
ORDER BY Customer.LastName
```

Sera traduit par :

```
var query = db.SalesOrderHeaders
    .Join(db.Customers,
    s => s.CustomerId,
    c = c.CustomerId,
   (c, m) => new { s.SalesOrderID, c.LastName })
   .OrderBy(c.LastName);
```

Ou

```
var query = from s in db.SalesOrderHeaders
            join c in db.Customers on s.CustomerID equals c.CustomerID
            orderby c.LastName
            select new
            {
            s.SalesOrderID,
            c.LastName
            };
```

12.4.6 Insérer avec LINQ (INSERT INTO)

Insérer des données avec LINQ est très simple. Vous devez vous rappeler que vous travaillez ici avec des classes et non plus directement avec la base de données. Insérer un enregistrement revient a créer une instance de l'objet a insérer.
Par exemple, supposons qu'on veuille créer un nouveau client :

```
Customer c = new Customer();
// ici, vous initialisez le contenu de votre objet Customer
db.Customers.InsertOnSubmit(c);
db.SubmitChanges();
```

Notez au passage, que si l'identifiant de votre enregistrement est un champ s'incrémentant automatiquement, vous récupérerez simplement la valeur de ce champ en faisant

```
String idNouveauClient = c.CustomerID;
```

12.4.7 Modifier un enregistrement avec LINQ (UPDATE)

Modifier un enregistrement est egalement tres simple. Supposons qu'on veuille modifier les informations sur le client dont l'identifiant est "100".

```
Customer client = db.Customers.Single(c => c.CustomerID == 100);
client.FirstName = "Jean";
client.LastName = "Bon";
db.SubmitChanges();
```

Au passage, vous noterez l'utilisation d'une fonction dont je n'avais pas encore parle : Single. Single renvoie un enregistrement unique, on peut lui passer en parametre un suite d'expressions a verifier. L'enregistrement repondant a tous ces criteres sera renvoye.

12.4.8 Supprimer un enregistrement avec LINQ

La suppression est egalement tres simple. Supposons qu'on veuille supprimer le client dont l'identifiant est "100" :

```
Customer client = db.Customers.Single(c => c.CustomerID == 100);
db.DeleteOnSubmit( client );
db.SubmitChanges();
```

12.4.9 Commandes SQL via LINQ

Vous pouvez egalement faire des requetes SQL directement via LINQ sans passer par ADO.NET (ca peut etre pratique si vous avez déjà ouvert une connexion avec LINQ et que vous ne vouliez pas reouvrir une connexion specifique a ADO.NET pour faire vos commandes SQL.
Pour cela, le data context dispose de deux methodes, une pour executer une commande SQL qui renvoie un resultat (typiquement une requete SELECT), une autre pour executer une commande qui ne renvoie pas de resultat (un INSERT par exemple).

Exemple de requete SQL avec resultat (SELECT) :

```
var query = db.ExecuteQuery(typeof(Customer),
"SELECT * FROM Customer WHERE CustomerID = @id", new object[] { 100 });
```

(Le typeof au debut sert a indiquer le type de donnees attendues en retour)

Example sans resultat :

```
db.ExecuteCommand(" INSERT Customers ( LastName, FirstName ) VALUES ( @lastname,
@firstname )", new object[] { "Bon", "Jean" });
```

Voila, nous allons passer aux choses serieuses. Avant cela, je vous conseille deux liens interessants :

http://weblogs.asp.net/scottgu/archive/2007/07/31/linq-to-sql-debug-visualizer.aspx

Ce lien vous envoie sur le blog de Scott Guthrie, un ingenieur de chez Microsoft. Il a cree un visualiseur pour Visual Studio assez interessant. Les visualiseurs sont des plug-ins au debugger de Visual Studio, ils vous permettent de visualiser des donnees de facon "native" pendant que vous debuggez. Ce visualiseur vous permet d'afficher l'equivalent SQL de la requete LINQ que vous etes en train de debugger.

http://www.sqltolinq.com/

Ce lien vous envoie sur Linqer. Linqer est un petit outil pratique qui vous permet de convertir du SQL en LINQ. C'est plutôt experimental et ca ne donne pas toujours le resultat attendu mais ca peut aider.

12.5 Utiliser LINQ to SQL avec des contrôles ASP.NET

Notez qu'a la différence des requêtes LINQ faites sur des objets IEnumerable comme un tableau, la requête dans la base de données n'est pas faite lorsque la requête LINQ est créée mais des que vous commencerez a utiliser le résultat de la requête (typiquement en commençant a parcourir les résultats).

Prenons un petit exemple simple : envoyer les données dans un GridView.

La page ASPX est on ne peut plus simple.

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
    <form id="form1" runat="server">
    <div>
        <asp:GridView ID="GridView1"
            runat="server"
        />
    </div>
    </form>
</body>
</html>
```

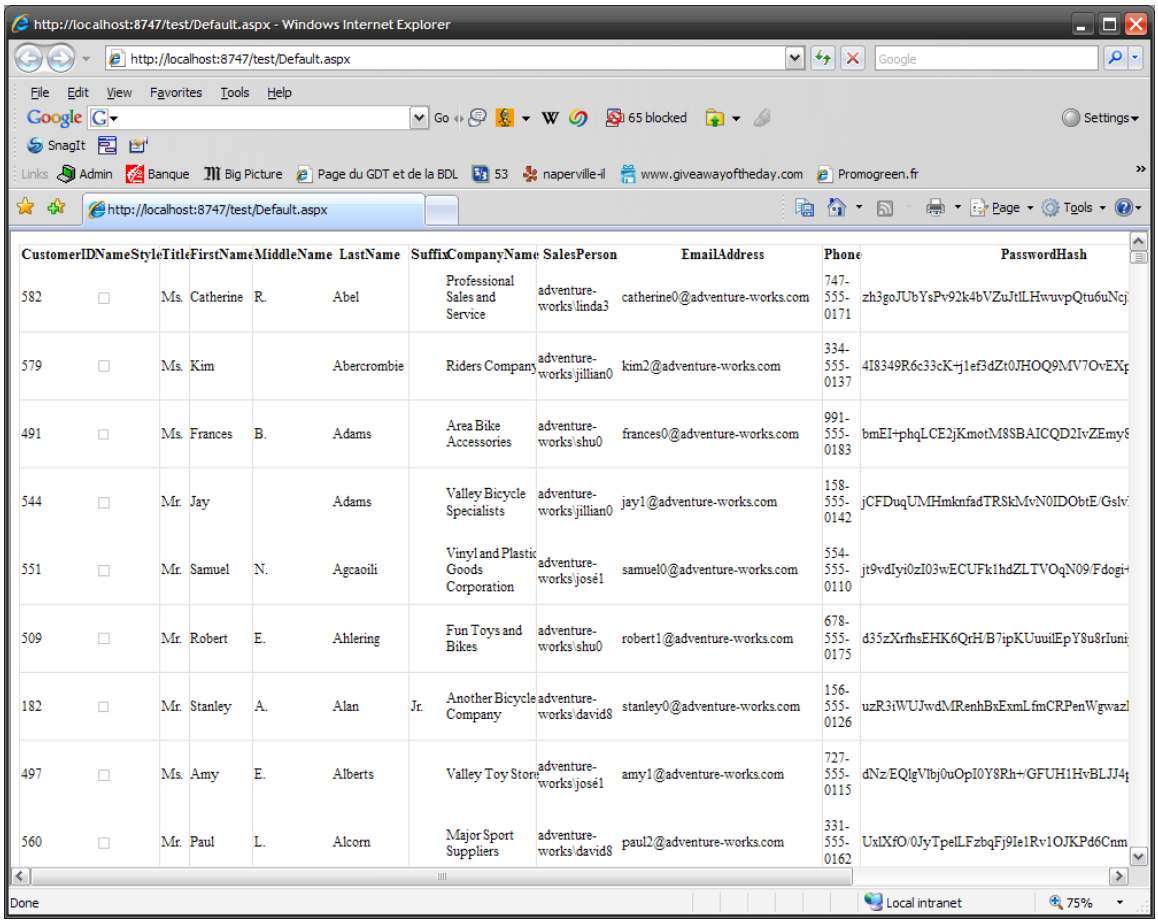
Dans le code behind, nous allons juste lire la table et binder ses données sur le GridView :

```
    protected void Page_Load(object sender, EventArgs e)
    {
        DataClassesDataContext db = new DataClassesDataContext();
        GridView1.DataSource = db.Customers.OrderBy(m => m.LastName);
        GridView1.DataBind();
    }
```

On commence par créer une instance de la Classe DataClassesDataContext, puis on initialise la DataSource.

Cela étant dit, vous avez compris a quoi ressemble la DataSource : c'est la table "Customer" triée sur le nom de famille. (En notation expression lambda).

Ca renvoie donc toute la table (de façon brute) :



CustomerID	NameStyle	Title	FirstName	MiddleName	LastName	Suffix	CompanyName	SalesPerson	EmailAddress	Phone	PasswordHash
582	☐	Ms.	Catherine	R.	Abel		Professional Sales and Service	adventure-works\linda3	catherine0@adventure-works.com	747-555-0171	zh3goJUbYsPv92k4bVZuJtlLHwuvpQtu6uNcj
579	☐	Ms.	Kim		Abercrombie		Riders Company	adventure-works\jillian0	kim2@adventure-works.com	334-555-0137	4I8349R6c33cK+j1ef3dZt0JHOQ9MV7OvEXp
491	☐	Ms.	Frances	B.	Adams		Area Bike Accessories	adventure-works\shu0	frances0@adventure-works.com	991-555-0183	bmEI+phqLCE2jKmotM8SBAICQD2IvZEmy8
544	☐	Mr.	Jay		Adams		Valley Bicycle Specialists	adventure-works\jillian0	jay1@adventure-works.com	158-555-0142	jCFDuqUMHmknfadTRSkMvN0IDObtE/Gslv
551	☐	Mr.	Samuel	N.	Agcaoili		Vinyl and Plastic Goods Corporation	adventure-works\josé1	samuel0@adventure-works.com	554-555-0110	jt9vdIyi0zI03wECUFk1hdZLTVOqN09/Fdogi+
509	☐	Mr.	Robert	E.	Ahlering		Fun Toys and Bikes	adventure-works\shu0	robert1@adventure-works.com	678-555-0175	d35zXrfhsEHK6QrH/B7ipKUuuilEpY8u8rIuni
182	☐	Mr.	Stanley	A.	Alan	Jr.	Another Bicycle Company	adventure-works\david8	stanley0@adventure-works.com	156-555-0126	uzR3iWUJwdMRenhBxExmLfmCRPenWgwazl
497	☐	Ms.	Amy	E.	Alberts		Valley Toy Store	adventure-works\josé1	amy1@adventure-works.com	727-555-0115	dNz/EQIgVlbj0uOpI0Y8Rh+/GFUH1HvBLJJ4
560	☐	Mr.	Paul	L.	Alcorn		Major Sport Suppliers	adventure-works\david8	paul2@adventure-works.com	331-555-0162	UxlXfO/0JyTpelLFzbqFj9Ie1Rv1OJKPd6Cnm

Pour info, chaque classe créée est une classe partielle (on peut lui ajouter des méthodes et des propriétés sans avoir besoin de créer un nouvel objet qui en dérive).

Il suffit de faire quelque chose comme ca :

```
public partial class Customer {

	public static IEnumerable<Customer> Select()
	{
		DataClassesDataContext db = new DataClassesDataContext();
		Return db.Customers;
	}
}
```

La méthode statique Select a été ajoutée a la classe "Customer", comme elle renvoie un IEnumerable, elle peut être utilisée comme DataSource.

12.6 Linq DataSource

Avec LINQ, un nouvel objet .NET est apparu, le LINQ DataSource. Il est utilise comme un DataSource normal, sauf qu'au lieu de parler SQL, il parle LINQ. Vous indiquez donc vos commandes select, update, insert et delete en LINQ.

Petit exemple avec notre table "Customers". Nous allons faire une petite page qui contient un champ texte, on y tapera le début d'un nom, et après avoir clique sur "OK", la liste des clients commencant par ces lettres sera affichée.

D'abord la page ASPX :

```
<form id="form1" runat="server">
<div>
   Entrez le nom : <br />
   <asp:TextBox ID="TextBoxNom" runat="server" Text="" />
   <asp:Button ID="Button1" runat="server" Text="OK" />

   <asp:GridView  ID="GridView1"
                  runat="server"
                  DataSourceID="LinqDataSource1"
                  Visible="false"/>

   <asp:LinqDataSource ID="LinqDataSource1" runat="server"
            ContextTypeName="DataClassesDataContext"
            TableName="Customers"
            Where="LastName.StartsWith(@DebutLastName)"
            OrderBy="LastName"
            Select="new (LastName, FirstName, CompanyName)">
            <WhereParameters>
                <asp:ControlParameter
                    ConvertEmptyStringToNull ="false"
                    ControlID="TextBoxNom"
                    PropertyName="Text"
                    Type="String"
                    Name="DebutLastName" />
            </WhereParameters>
        </asp:LinqDataSource>

    </div>
    </form>
```

Qu'y trouve-t-on ? Au début un classique champ texte et un bouton, ne nous attardons pas la dessus. Puis un GridView basique dont la DataSource est "LinqDataSource1".

Vient ensuite le plus intéressant, la DataSource elle-même. Petites explications:

- **ContextTypeName** indique le ContextData crée par Visual Studio
- **TableName** indique la table sur laquelle va se faire la requête. Attention, il ne s'agit pas du nom de la table mais du nom de la propriété du DataContext qui renvoie une classe mappée sur la table (d'où le Customers au pluriel).
- **Where** est la clause where. Notez que vu qu'on utilise LINQ, on peut aller plus loin que SQL et utiliser ici du code C# directement, LastName étant mappe sur une String, on peut utilise toutes les fonctions de la classe String, ici StartsWith renvoie true si la chaine commence par la valeur passée en paramètre. @DebutLastName est le paramètre, il va contenir ce que l'utilisateur a saisi.
- **OrderBy** permet de trier la réponse.
- **Select** enfin permet de créer l'objet qui sera renvoye, on y choisi les champs qu'on veut y voir apparaitre.

Pour éviter d'avoir a initialiser nous même la valeur du paramètre @DebutLastName, nous allons indiquer directement dans la DataSource ou aller chercher la valeur de ce paramètre. Cela se fait avec la clause <WhereParameter>.

Dans cette clause, nous indiquons ou aller chercher la valeur des paramètres. On retrouve un mécanisme qu'on avait vu précédemment avec les DataSources. On peut aller chercher la valeur de ces paramètres dans plusieurs endroits : la query string dans l'url, une valeur en session, ou, et c'est ce qui nous intéresse, dans une propriété d'un objet présent sur la page. Pour indiquer qu'on est dans ce dernier cas, on va utiliser l'objet ControlParameter (le paramètre est dans un contrôle ASPX).

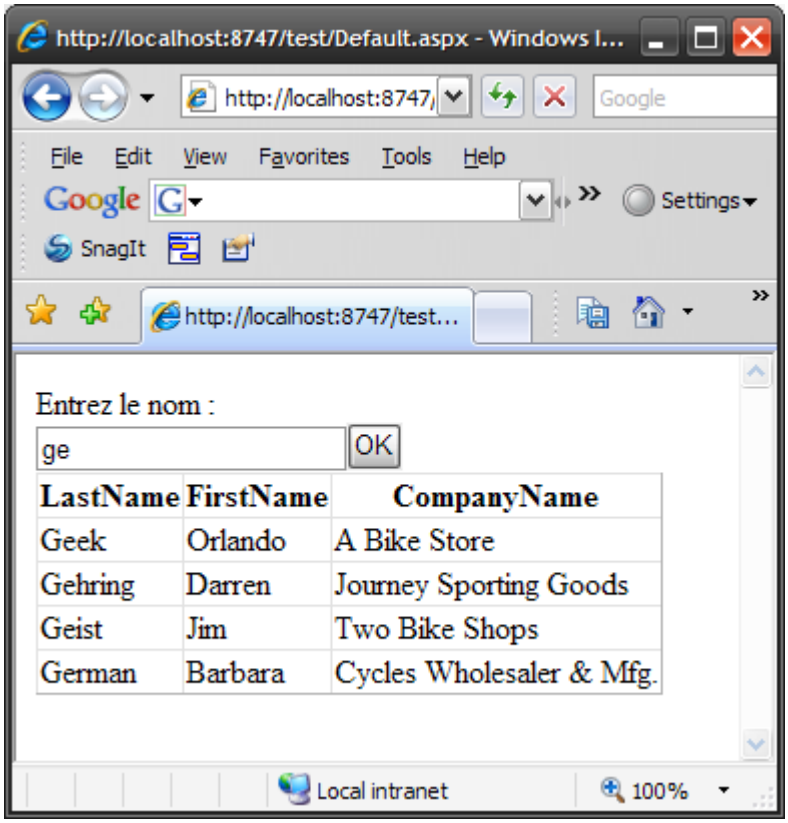
- **ControlID** : indique l'id du contrôle ou aller chercher la valeur du paramètre
- **PropertyName** : indique la propriété du contrôle utilisée (ici la propriété Text qui contient ce que l'utilisateur a tape)
- **Type** : indique le type de donnée du paramètre.
- **Name** : le nom du paramètre (utilise dans la clause where)
- **ConvertEmptyStringToNull** : Par defaut, si la valeur du parametre est une chaine vide, elle est convertie en null, ce qui va provoquer une erreur (StartsWith(null) n'est pas genial). On indique donc ici de laisser la chaine comme elle est.

Que va-t-il se passer ? A chaque fois que la page est chargée (initialement ou lors d'un postback réalise avec le bouton OK), la DataSource va aller voir ce que contient la propriété "Text" du contrôle nomme "TextBoxNom", puis va faire une requête LINQ avec cette valeur et binder le résultat de cette requête sur le DataGrid.

Elle le fait aussi une fois au chargement initial de la page, ce qui affiche une première fois un DataGrid avec tout le contenu de la table, ce qui n'est pas heureux. C'est pour ca que la propriété "Visible" est mise a "false" par défaut : le GridView n'est pas affiche par défaut. Lors d'un postback, il est rendu visible (dans le code behind) :

```
protected void Page_Load(object sender, EventArgs e)
 {
    if (IsPostBack) GridView1.Visible = true;
 }
```

Le résultat :



Assez simple n'est-ce-pas ? On peut également avec ce type de DataSource faire des insertions, modifications ou suppressions très simplement (encore plus simplement qu'avec SQL !).

Reprenons nos clients et créons une page ASPX avec un GridView dans lequel nous permettrons l'édition et la suppression et un DetailView nous permettant l'insertion :

```
<form id="form1" runat="server">
    <div>
        <asp:DetailsView ID="detailCustomer"
            DataSourceID="LinqDataSource1"
            DefaultMode="Insert"
            AutoGenerateRows="false"
            AutoGenerateInsertButton="true"
            runat="server"
            OnItemInserted="nouveauClient">
            <Fields>
             <asp:BoundField DataField="LastName" HeaderText="Nom" />
             <asp:BoundField DataField="FirstName"
                 HeaderText="Prenom" />
             <asp:BoundField DataField="CompanyName"
                 HeaderText="Societe" />
            </Fields>
            </asp:DetailsView>

        <br /><br />

        <asp:GridView ID="GridView1"
            runat="server"
            DataKeyNames="CustomerID"
            DataSourceID="LinqDataSource1"
            AllowPaging="true"
            PageSize="5"
            AutoGenerateEditButton="true"
            AutoGenerateDeleteButton="true"
        />
```

```
        <asp:LinqDataSource ID="LinqDataSource1" runat="server"
            ContextTypeName="DataClassesDataContext"
            TableName="Customers"
            OrderBy="LastName"
            EnableInsert="true"
            EnableUpdate="true"
            EnableDelete="true"
            AutoPage="true" />

    </div>
    </form>
```

Bien. Qu'avons-nous ?

D'abord un DetailView qui va nous permettre d'insérer un nouveau client (le GridView ne permet pas l'insertion directe). Je reviens sur les propriétés déclarées (bien que ce soit du déjà vu) :

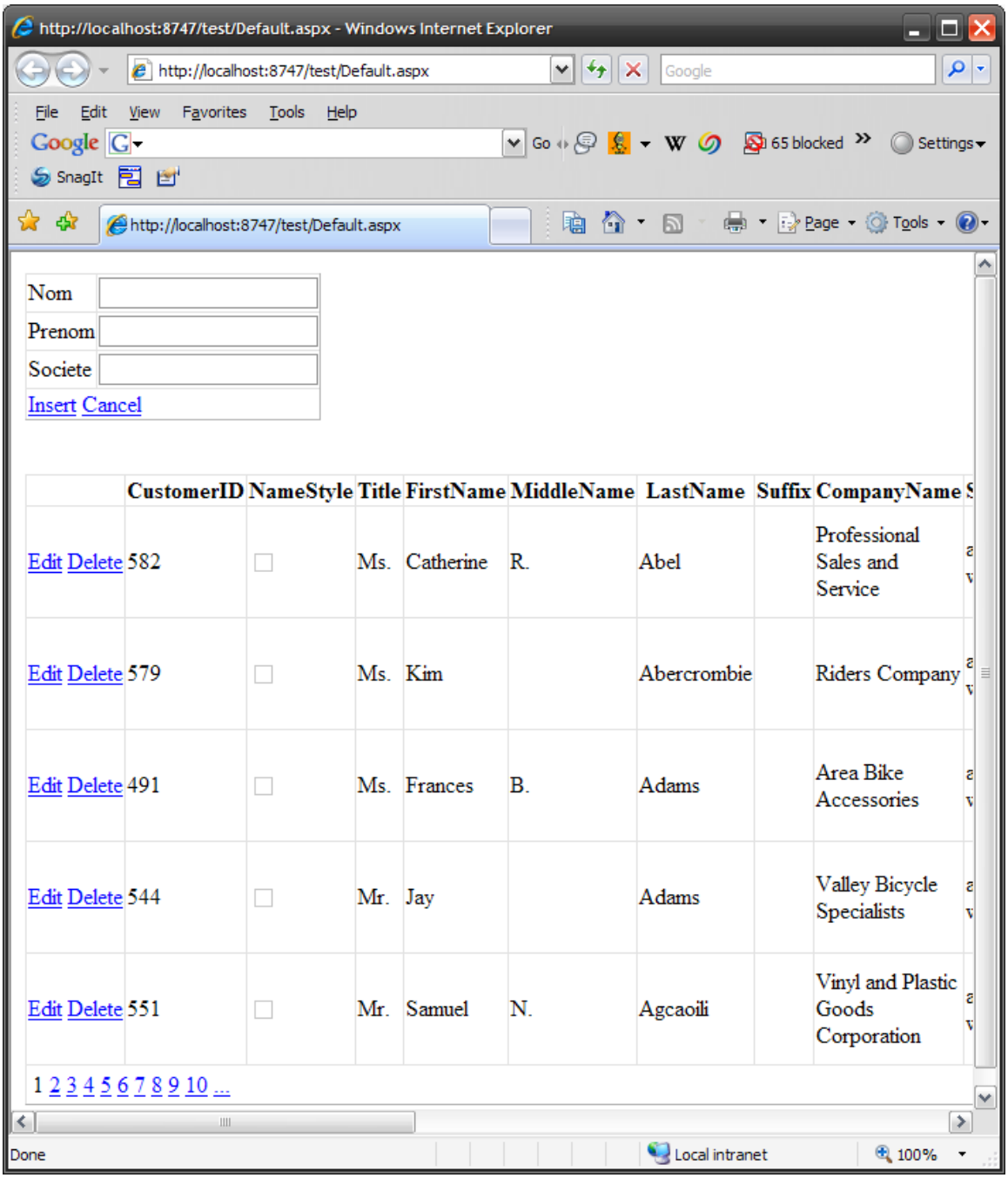
- **DataSourceID** : La DataSource liee au DetailView, ici notre DataSource LINQ
- **DefaultMode** : Le mode par defaut est "Insert", c'est-à-dire l'insertion. Le DetailView n'est pas utilise pour editer des donnees existantes (mode "Edit") ou pour visualiser des donnees existantes (mode "ReadOnly") mais pour ajouter des donnees a la table.
- **AutoGenerateRows** : false, on ne veut pas creer un formulaire avec tous les champs de la table mais seulement ceux qu'on va choisir dans la clause <Fields>
- **AutoGenerateInsertButton** : true – Affiche automatiquement un bouton "Insert" pour inserer l'enregistrement dans la table
- **OnItemInserted** : Le nom d'une fonction dans le code behind qui sera appellee lorsqu'un enregistrement sera insere dans la table (pour rafraichir le contenu du GridView avec le nouvel enregistrement).

Suivent ensuite la liste des champs qu'on veut afficher dans le DetailView (DataField indique le nom du champ dans la table, et HeaderText le texte a afficher a gauche du champ).

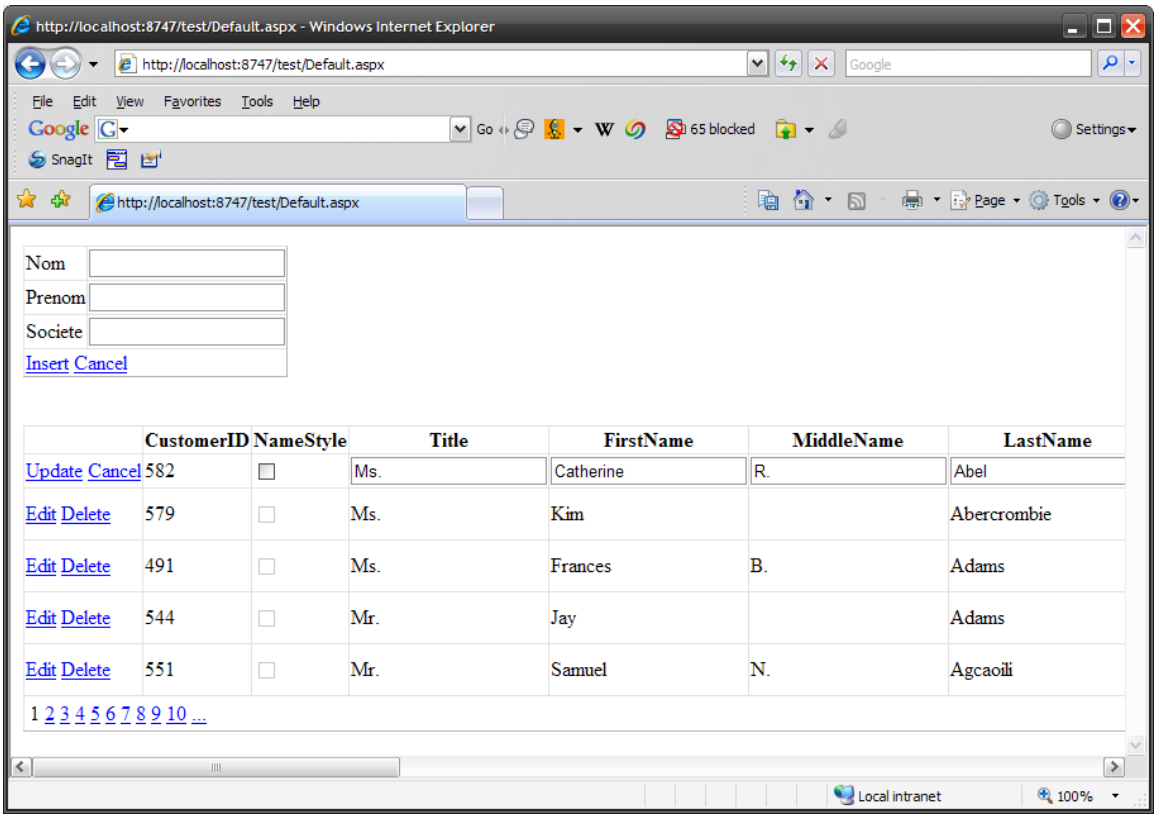
Ensuite, nous avons un GridView basique qui va afficher pour chaque ligne des boutons "Delete" et "Edit" (AutoGenerateEditButton et AutoGenerateDeleteButton a true). Notez que le contenu du GridView est pagine. N'oubliez pas d'indiquer le nom du champ clef (DataKeyNames). Il est indispensable pour faire les modifications dans la table.

Enfin, la DataSource LINQ, les nouvelles proprietes sont EnableUpdate, EnableInsert et EnableDelete a true. Les requetes LINQ seront automatiquement generees a partir du champ clef du gridview.

Voila le resultat :



On peut editer un enregistrement :



(Notez que le champ clef n'est pas modifiable)

13 ASP.NET et AJAX

La technologie Microsoft ASP.NET supporte AJAX (c'est la moindre des choses, AJAX a été invente par Microsoft en 1998 pour faire communiquer son client web mail avec le serveur de mail Exchange !). Jusqu'à la version 3 et Visual Studio 2008, les contrôles permettant d'utiliser AJAX était disponible sous forme de packages externes (connus d'abord sous le nom de Microsoft ATLAS, puis renomme en AJAX.NET). Ces contrôles sont integres au Framework depuis et directement disponibles dans Visual Studio (regardez dans la toolbox la partie nommée "AJAX Extensions").

Nous n'allons pas revenir ici sur le détail de la technologie AJAX en elle-même. Pour faire très rapide, AJAX est l'acronyme d'Asynchronous Javascript And XML. Cette technologie utilise un objet ActiveX (dans le cas d'internet Explorer) ou un objet du navigateur web (nomme XmlHttpRequest pour les autres navigateurs) en Javascript pour envoyer une requête HTTP vers un serveur web et récupérer en retour des informations. Contrairement aux requêtes HTTP get ou post, la page complète n'est pas renvoyée au client, ce dernier peut donc rafraichir une partie de sa page sans avoir besoin de la recharger entièrement. Ce qui en théorie, améliore l'expérience utilisateur (a condition de ne pas en abuser, trop de Javascript et trop de manipulation du DOM produisent parfois l'effet inverse : la page n'est pas rechargée, mais son contenu devient très lent a être manipule).

AJAX.NET a été conçu de manière a ne pas bouleverser les habitudes du développeur ASP.NET. En gros, pour rendre votre application "AJAX", vous ne devez, en gros, que placer les composants que vous voulez rafraichir dans un panel spécial et tout est fait automatiquement sans avoir besoin de taper de javascript.

13.1 Le contrôle UpdatePanel

Le principal contrôle utilise pour AJAX est le contrôle UpdatePanel. Il vous permet de mettre a jour une partie de la page sans avoir besoin de la recharger entièrement. Tous les contrôles que vous placez dans ce panel peuvent donc être modifies sur le serveur puis renvoyés a la page sans rechargement de cette dernière.

13.1.1 Le ContentTemplate

Commençons avec un exemple simple : un label et un bouton. A chaque fois qu'on clique sur le bouton, le label va être rafraichi avec l'heure courante sur le serveur. Pour éviter d'avoir a recharger toute la page a chaque fois, on va placer le label dans un UpdatePanel.

Notre page ASPX ressemble a ca :

```
<form id="form1" runat="server">
<div>
    <asp:ScriptManager ID="ScriptManager1" runat="server" />

    <asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>
            <asp:Label ID="Label1" runat="server"></asp:Label>
            <asp:Button ID="Button1" runat="server" Text="Heure" />
        </ContentTemplate>
    </asp:UpdatePanel>

</div>
</form>
```

Oublions pour l'instant le contrôle ScriptManager, sachez seulement que si vous voulez utiliser AJAX dans votre page, vous devez obligatoirement insérer ce contrôle au début de la page. C'est lui qui va générer tout le Javascript dont nous aurons besoin.

Intéressons-nous au contrôle UpdatePanel. Ce contrôle contient un template nomme "**ContentTemplate**". Vous placez dans ce template les contrôles que vous voulez modifier via AJAX. ET C'EST TOUT !

Cote serveur, ca se corse :

```
protected void Page_Load(object sender, EventArgs e)
    {
        Label1.Text = DateTime.Now.ToString("T");
    }
```

Non finalement ;-). Cote serveur, on continue de travailler EXACTEMENT de la même façon qu'avant, rien ne change, sauf qu'au final, le ScriptManager va générer tout le javascript nécessaire cote client et faire la requête. Le serveur saura que le postback qu'il reçoit est en fait une requête http Ajax et qu'il doit renvoyer l'information sous le format adéquat, le script cote client se chargera alors a la réception de modifier le DOM pour afficher l'information en question.

Tout ca sans aucune ligne de javascript et sans rien connaitre a XmlHttpRequest. Notez que ceci fonctionne sur Internet Explorer bien sur, mais également sur Firefox et Opera.

Le contrôle UpdatePanel fonctionne avec tous les contrôles que nous avons déjà utilise. Vous pouvez ainsi par exemple, renseigner le contenu d'une seconde liste déroulante en fonction de ce que l'utilisateur aura choisi dans la première.

Petit exemple : Une première liste déroulante affiche des marques de véhicules, en fonction du choix de la marque, la deuxième liste est renseignée avec les modèles.

La page ASPX :

```
<form id="form1" runat="server">
<div>
   <asp:ScriptManager ID="ScriptManager1" runat="server" />

   <asp:UpdatePanel ID="UpdatePanel1" runat="server">
   <ContentTemplate>
      <asp:DropDownList ID="DropDownList1" runat="server"
       onselectedindexchanged="DropDownList1_SelectedIndexChanged"
       AutoPostBack="true">

         <asp:ListItem Value="1" Text="Chevrolet" />
         <asp:ListItem Value="2" Text="Dodge" />
         <asp:ListItem Value="3" Text="Ford" />
      </asp:DropDownList>

      <asp:DropDownList ID="DropDownList2" runat="server">
      </asp:DropDownList>

   </ContentTemplate>
   </asp:UpdatePanel>
</div>
</form>
```

Qu'y trouvons-nous ? Le contrôle ScriptManager et l'UpdatePanel, puis le contenu de l'UpdatePanel qui est exactement ce que nous trouverions si nous n'utilisions pas AJAX (A ce sujet d'ailleurs, vous pouvez supprimer l'UpdatePanel pour voir, la page se comportera alors de façon classique, c'est-à-dire en renvoyant tout son contenu lors du postback).

Comme le fonctionnement est identique a une page non-AJAX, vous noterez les propriétés onselectedindexchanged qui indique quelle fonction appeler sur le serveur lorsque l'utilisateur clique sur une des entrees de la liste et AutoPostBack a true qui indique que l'appel a la fonction sur le serveur doit se faire des que l'utilisateur clique sur une entree.

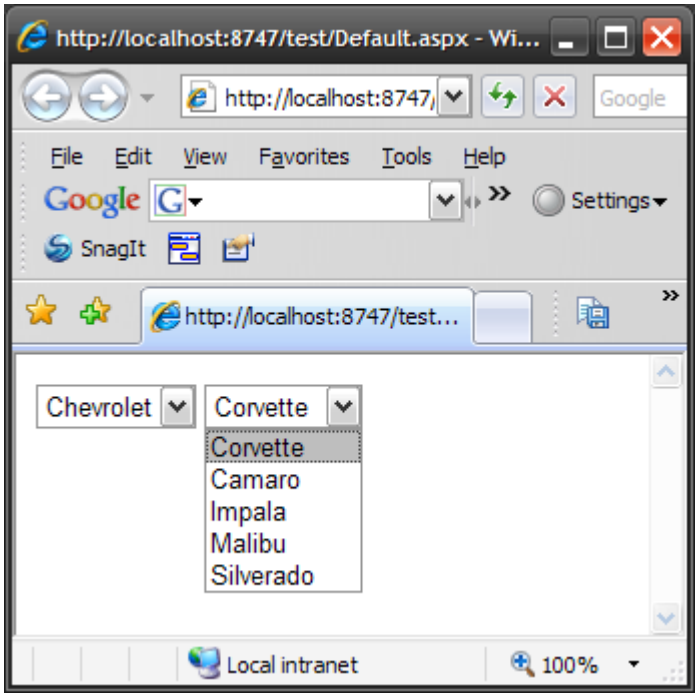
Cote serveur :

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
    {
    List<String> l;

    switch (int.Parse(DropDownList1.SelectedValue))
        {
```

```
        case 1: l = new List<string> { "Corvette", "Camaro", "Impala",
                    "Malibu", "Silverado" };
                    break;
            case 2: l = new List<string> { "Avenger", "Caliber",
                    "Charger", "Journey", "RAM" };
                    break;
            default: l = new List<string> { "Fusion", "Mustang",
                    "Taurus", "Edge", "F150" };
                    break;
        }
        DropDownList2.DataSource = l;
        DropDownList2.DataBind();
    }
```

Rien d'extraordinaire : en fonction de la marque choisie, les modeles sont initialises dans la deuxième liste. Dans la vie réelle, bien sur, on irait chercher ces informations dans une base de données.

Le résultat (pas très parlant) :



Nous n'allons pas repasser en revue tous les contrôles, car tous peuvent être utilises de la même façon. Notez cependant qu'une utilisation intéressante peut être avec un GridView : le tri des colonnes par exemple ne rechargera pas la page ce qui améliore l'interface utilisateur.

13.1.2 Les Triggers

Lorsqu'on clique sur un bouton place dans un UpdatePanel, le bouton génère une requête AJAX sur le serveur au lieu de faire un postback habituel, ce qui sous-entend que le contrôle générant la requête doit donc se trouver dans le même UpdatePanel que les contrôles a mettre a jour. Ca peut parfois poser des problèmes si on veut mettre a jour des données situées dans un autre UpdatePanel ou si le bouton en question est situe en dehors des UpdatePanels.
C'est pour ca qu'on été créé les triggers Ajax. Ils permettent de mettre a jour des UpdatePanel meme si le bouton responsable de cette mise a jour ne fait pas partie du panel.

Reprenons notre premier exemple avec l'heure et modifions la page ASPX (Le code behind ne change pas) :

```
<form id="form1" runat="server">
  <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server" />

      <asp:UpdatePanel ID="UpdatePanel1" runat="server">

      <ContentTemplate>
          <asp:Label ID="Label1" runat="server"></asp:Label>
      </ContentTemplate>

      <Triggers>
          <asp:AsyncPostBackTrigger ControlID="Button1" />
      </Triggers>

      </asp:UpdatePanel>

      <asp:Button ID="Button1" runat="server" Text="Heure" />

  </div>
  </form>
```

Le contrôle Button a été deplace hors de l'UpdatePanel et une nouvelle section est apparue dans cet UpdatePanel : la section "Triggers". La section Triggers contient des contrôles AsyncPostBackTrigger qui indiquent quels sont les autres contrôles de la page qui peuvent provoquer la mise a jour du panel. Vous pouvez bien sur avoir plusieurs contrôles AsyncPostBackTrigger dans cette section car plusieurs boutons situes sur la page peuvent forcer de façon indépendante la mise a jour du panel.

Vous pouvez bien sur imbriquer autant d'UpdatePanel entre eux que vous le voulez si vous voulez affiner le degré de mise a jour des informations : moins les UpdatePanel contiennent de contrôles, plus l'utilisation est fluide car leur mise a jour nécessite moins de transferts entre le client et le serveur.

Note : Le contrôle UpdatePanel possède une méthode Update(). Vous pouvez, en appelant cette méthode, forcer la mise a jour du panel par programme.

13.1.3 Utilisation de Javascript dans un UpdatePanel

On peut, a partir du code behind, envoyer du code Javascript a la page générée. Pour cela, on utilise une des propriétés de la page nommée ClientScript. ClientScript renvoie un ClientScriptManager qui est un objet qui gère le javascript généré dans la page renvoyée au client.
Une des méthodes de l'objet ClientScriptManager est la méthode "RegisterStartupScript". En gros, vous lui envoyez le code javascript que vous voulez insérer dans la page et il sera place au début de la page générée.

Petit Exemple : Une page ASPX simple avec un seul bouton. Quand on clique sur le bouton, un postback est fait, a ce moment, un script est créé (une bête boite d'alerte) et est ajoute a la génération de la page. Quand la page est affichée en retour, la boite d'alerte apparait.

La page ASPX :

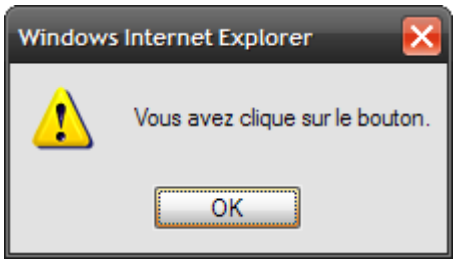
```
<form id="form1" runat="server">
    <div>
        <asp:Button ID="Button1" runat="server" Text="OK" />
    </div>
</form>
```

Le code behind :

```
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack)
    {
    String script = "alert ('Vous avez clique sur le bouton.');";
    Page.ClientScript.RegisterStartupScript(this.GetType(),
    "le_script", script, true);
    }
}
```

Le premier paramètre est le type Page, le second est une clef utilise pour ce script (pour éviter si vous générez deux fois ce script a différents endroits du code avec la même clef, il ne sera inséré qu'une fois), le troisième est le code javascript du script et le troisieme indique si le tag <script></script> doit encadrer le script lors de la génération.

Le résultat : après avoir clique sur le bouton, la boite d'alerte s'affiche :



Bien, tout ca pour dire que si le bouton est dans un UpdatePanel, le code Javascript ne sera pas renvoyé d'une requête Ajax. On ne peut pas envoyer de javascript en retour d'une requête ajax avec l'objet ClientScriptManager. Heureusement, une solution de contournement existe.
Nous allons utiliser pour cela l'objet ScriptManager que nous avons vu au début du chapitre. Cet objet gère le code javascript utilise pour les échanges client / serveur lors d'une requête Ajax, il permet aussi, en retour d'injecter du code Javascript dans la page.

Pour cela, trois méthodes sont intéressantes : les méthode **RegisterClientScriptBlock / RegisterStartupScript** qui vont envoyer un bloc de code Javascript (la première place le script après le <form>, la seconde avant) et la méthode **RegisterClientScriptInclude** qui va ajouter en début de page une ligne d'inclusion de fichier Javascript (<script type="text/javascript" src="..."></script>).

13.2 Le contrôle Timer

Le contrôle Timer vous permet de rafraichir un UpdatePanel (ou toute la page) a intervalle régulier. Il n'a en fait qu'une seule propriété vraiment intéressante : **Interval** qui indique en millisecondes, l'intervalle de temps entre deux rafraichissements. Vous pouvez associer un UpdatePanel a ce timer ou, si vous ne l'associez a aucun panel, il rafraichira la page complète (il fera un postback normal).

Petit exemple de rafraichissement de la page complète :

```
<form id="form1" runat="server">
<div>
    <asp:ScriptManager ID="ScriptManager1" runat="server" />
    <asp:Label ID="Label1" runat="server" />
    <asp:Timer ID="Timer1" runat="server" Interval="2000" />
</div>
</form>
```

Le code behind :

```
protected void Page_Load(object sender, EventArgs e)
 {
     Label1.Text = DateTime.Now.ToString("T");
 }
```

Faisons plus complique, associons le timer a un UpdatePanel. Le panel (et uniquement ce panel) sera rafraichi selon l'intervalle spécifié dans le timer.

La page ASPX :

```
<form id="form1" runat="server">
<div>
     <asp:ScriptManager ID="ScriptManager1" runat="server" />
     <asp:Timer ID="Timer1" runat="server" Interval="2000" />

     <asp:UpdatePanel ID="UpdatePanel1" runat="server">
     <Triggers>
       <asp:AsyncPostBackTrigger ControlID="Timer1" EventName="Tick" />
     </Triggers>
     <ContentTemplate>
       <asp:Label ID="Label1" runat="server" />
     </ContentTemplate>
     </asp:UpdatePanel>
</div>
</form>
```

Vous noterez que le nom de l'événement associe au trigger pour un contrôle Timer est "Tick".

Le code behind ne change pas.
Outre le fait que visuellement c'est plus beau (la page n'est pas rechargée a chaque "tick" du timer), le timer est également plus précis, car le temps de rechargement de la page ne modifie pas le temps déjà écoulé.

Vous pouvez utiliser ce contrôle pour aller vérifier a intervalle régulier l'état d'un traitement se déroulant sur le serveur.

13.3 Le contrôle UpdateProgress

Le contrôle UpdateProgress vous permet d'afficher un indicateur de progression pendant que l'UpdatePanel met a jour son contenu. La "tradition" veut que pendant qu'on met a jour via une requête AJAX une partie d'une page web, on affiche un petit indicateur de progression circulaire. Vous trouverez des générateurs de ce type d'images aux adresses suivantes : **http://www.ajaxload.info/** ou **http://www.webscriptlab.com/**.
Basiquement, le contrôle UpdateProgress contient un template nommée "ProgressTemplate". Dans ce template, vous placez ce que vous voulez. Ce contenu sera affiche lorsque l'UpdatePanel associe a cet UpdateProgress fera une requête Ajax et disparaitra des que la requête sera terminée.

Petit exemple : Dans notre exemple, nous allons volontairement faire durer le traitement sur le serveur plusieurs secondes pour que vous puissiez voir l'image s'afficher. L'image affichée (ajax-loader.gif a été générée sur le 1er site cite plus haut).

La page ASPX :

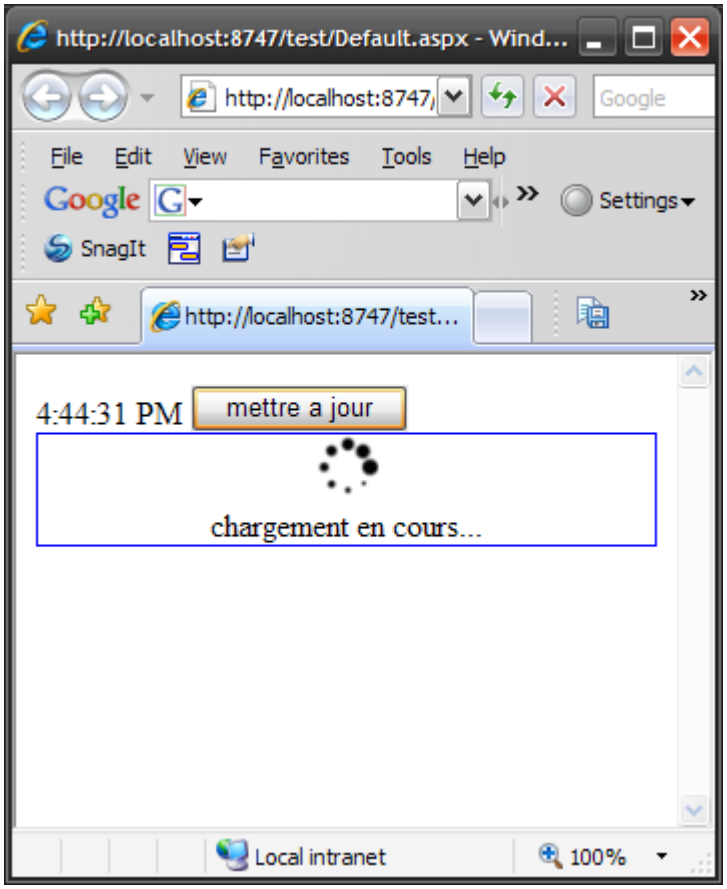
```
<form id="form1" runat="server">
<div>
  <asp:ScriptManager ID="ScriptManager1" runat="server" />
  <asp:UpdatePanel ID="UpdatePanel1" runat="server">
  <ContentTemplate>
     <asp:Label ID="Label1" runat="server" />
     <asp:Button ID="Button1" runat="server" Text="mettre a jour" />
  </ContentTemplate>
  </asp:UpdatePanel>

  <asp:UpdateProgress ID="UpdateProgress1"
   runat="server" AssociatedUpdatePanelID="UpdatePanel1">
  <ProgressTemplate>
      <div style="border: solid 1px blue; text-align:center">
          <img src="ajax-loader.gif" />
          <br />
          chargement en cours...
      </div>
  </ProgressTemplate>
  </asp:UpdateProgress>
    </div>
    </form>
```

Notez la propriété "AssociatedUpdatePanelID" de l'UpdateProgress. On y indique l'identifiant de l'UpdatePanel correspondant.

Le code behind est légèrement modifie pour faire patienter 5 secondes :

```
protected void Page_Load(object sender, EventArgs e)
 {
     System.Threading.Thread.Sleep(5000);
     Label1.Text = DateTime.Now.ToString("T");
 }
```

Le résultat :



Par défaut (et pour éviter des clignotements intempestifs si le temps de chargement est court) l'UpdateProgress attend 0.5 secondes avant d'afficher son contenu. Si vous voulez réduire ce délai, modifiez la propriété "DisplayAfter" du contrôle et indiquez le delai ici (en millisecondes).

13.4 L'AJAX Control Toolkit

Il existe d'autres contrôles AJAX pour ASP.NET proposes soit par Microsoft, soit par des sociétés tierces. Un des packs de contrôles AJAX pour ASP.NET les plus intéressants est celui développé par Microsoft et disponible a cette adresse :

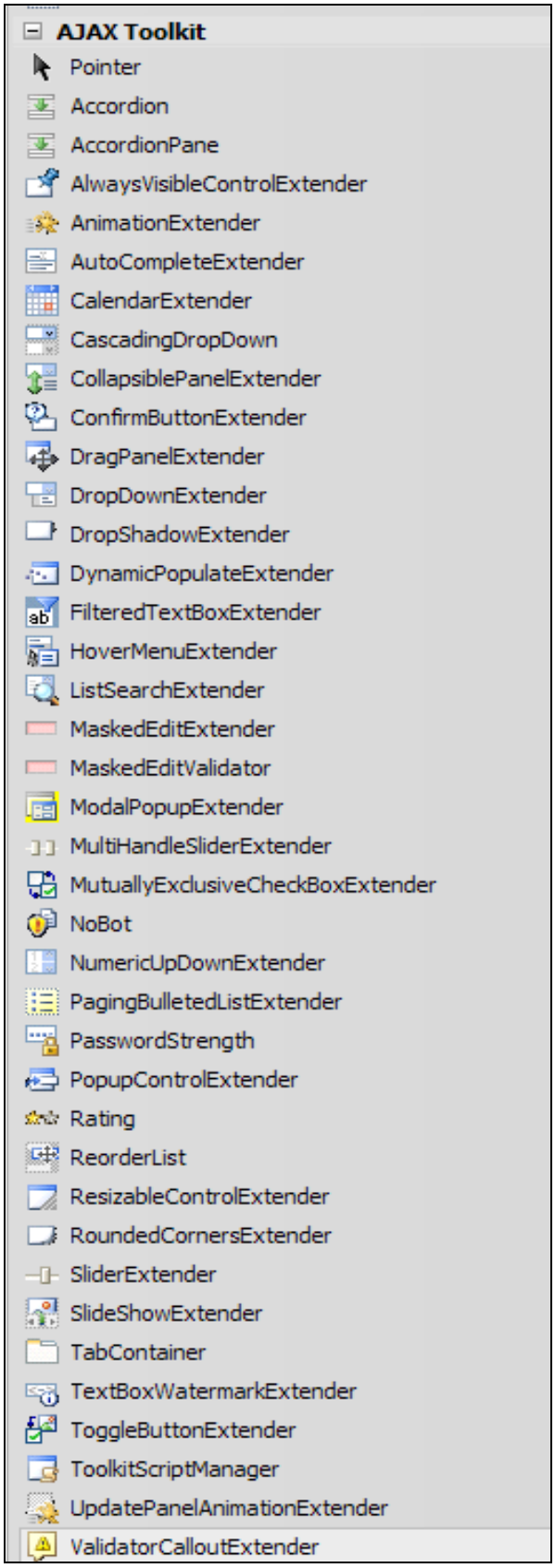
http://www.codeplex.com/AjaxControlToolkit

CodePlex est un site sur lequel sont déposés des projets open source libres pour la plupart dédiés a .NET.

L'AJAX Control Toolkit est un ensemble de contrôles utilisant Ajax (une trentaine a l'heure ou je tape ce texte). Le projet étant continuellement mis a jour, je vous conseille de vérifier de temps a autre son actualité. Il est distribue sous licence Microsoft Public License, une licence très souple, vous n'avez donc pas a vous inquiéter pour utiliser et distribuer ces contrôles.

Téléchargez le dans l'onglet "Releases" et suivez cette URL (en anglais of course) : **http://www.asp.net/AJAX/AjaxControlToolkit/Samples/Walkthrough/Setup.aspx** qui vous indiquera comment installer ces contrôles dans Visual Studio.

Une fois installes, vous disposerez des contrôles suivants :


AJAX Toolkit
Pointer
Accordion
AccordionPane
AlwaysVisibleControlExtender
AnimationExtender
AutoCompleteExtender
CalendarExtender
CascadingDropDown
CollapsiblePanelExtender
ConfirmButtonExtender
DragPanelExtender
DropDownExtender
DropShadowExtender
DynamicPopulateExtender
FilteredTextBoxExtender
HoverMenuExtender
ListSearchExtender
MaskedEditExtender
MaskedEditValidator
ModalPopupExtender
MultiHandleSliderExtender
MutuallyExclusiveCheckBoxExtender
NoBot
NumericUpDownExtender
PagingBulletedListExtender
PasswordStrength
PopupControlExtender
Rating
ReorderList
ResizableControlExtender
RoundedCornersExtender
SliderExtender
SlideShowExtender
TabContainer
TextBoxWatermarkExtender
ToggleButtonExtender
ToolkitScriptManager
UpdatePanelAnimationExtender
ValidatorCalloutExtender