Introduction rapide à ASP.NET avec Visual Studio 2010

serge.tahe at istia.univ-angers.fr, novembre 2011

1 Introduction

Nous nous proposons ici d'introduire, à l'aide de quelques exemples, les concepts importants d'ASP.NET. Cette introduction ne permet pas de comprendre les subtilités des échanges client / serveur d'une application web. Pour cela, on pourra lire d'anciens documents qui décrivent les échanges Http entre un navigateur et un serveur web :

- **Programmation ASP.NET** [http://tahe.developpez.com/dotnet/aspnet/vol1] et [http://tahe.developpez.com/dotnet/aspnet/vol2]

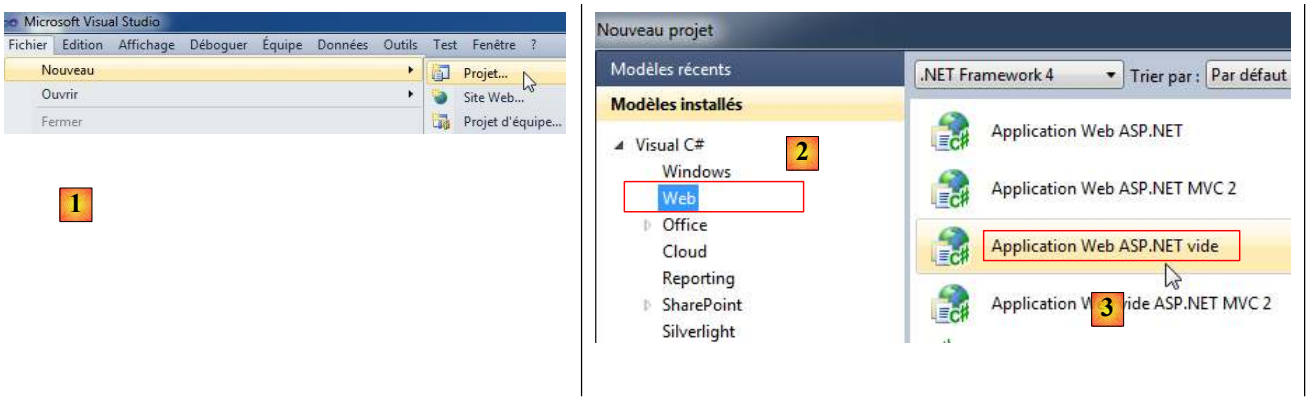
Cette introduction est faite pour ceux qui voudraient aller vite en acceptant, dans un premier temps, de laisser dans l'ombre des points qui peuvent être importants. Ce document peut être ultérieurement approfondi par une étude de cas :

- **Construction d'une application à trois couches avec ASP.NET, C#, Spring.net et Nhibernate** [http://tahe.developpez.com/dotnet/pam-aspnet/]

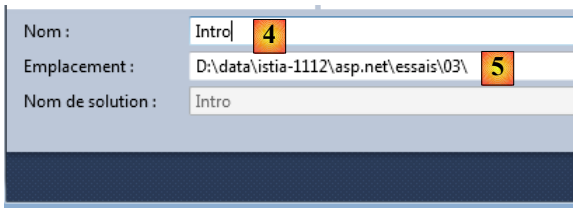
Les exemples ont été construits avec Visual Studio 2010 Professionnel. Des versions Express de Visual Studio 2010 sont disponibles à l'Url [http://msdn.microsoft.com/fr-fr/express/aa975050] (novembre 2011) et peuvent être téléchargées librement. On téléchargera ainsi Visual Web Developer Express 2010. Les copies d'écran de ce document sont celles de Visual Studio 2010 Professionnel . Elles pourront parfois différer des écrans de Visual Studio 2010 Express mais le lecteur devrait s'y retrouver néanmoins aisément.

2 Un projet exemple

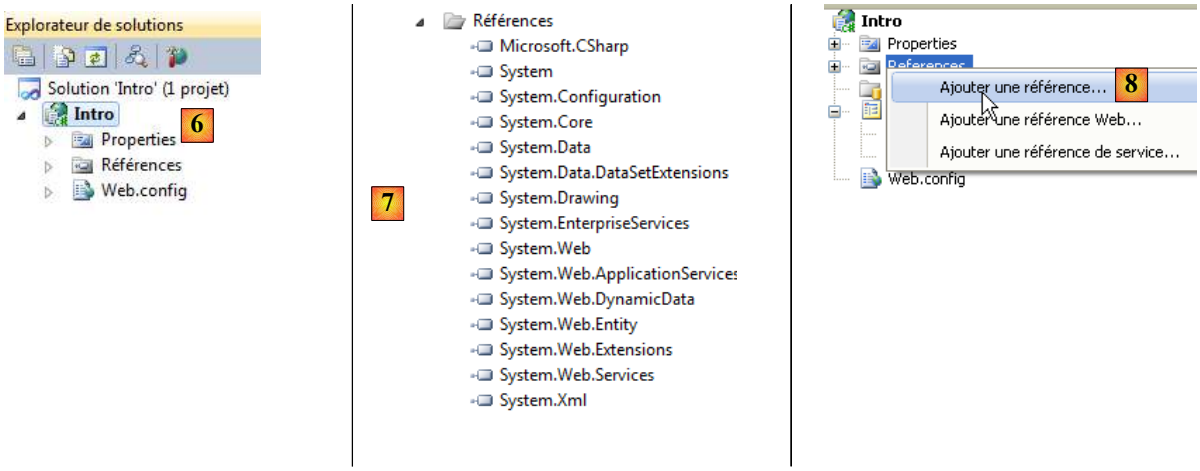
2.1 Création du projet



- en [1], on crée un nouveau projet avec Visual Studio
- en [2], on choisit un projet web en Visual C#
- en [3], on indique qu'on veut créer une application web ASP.NET vide



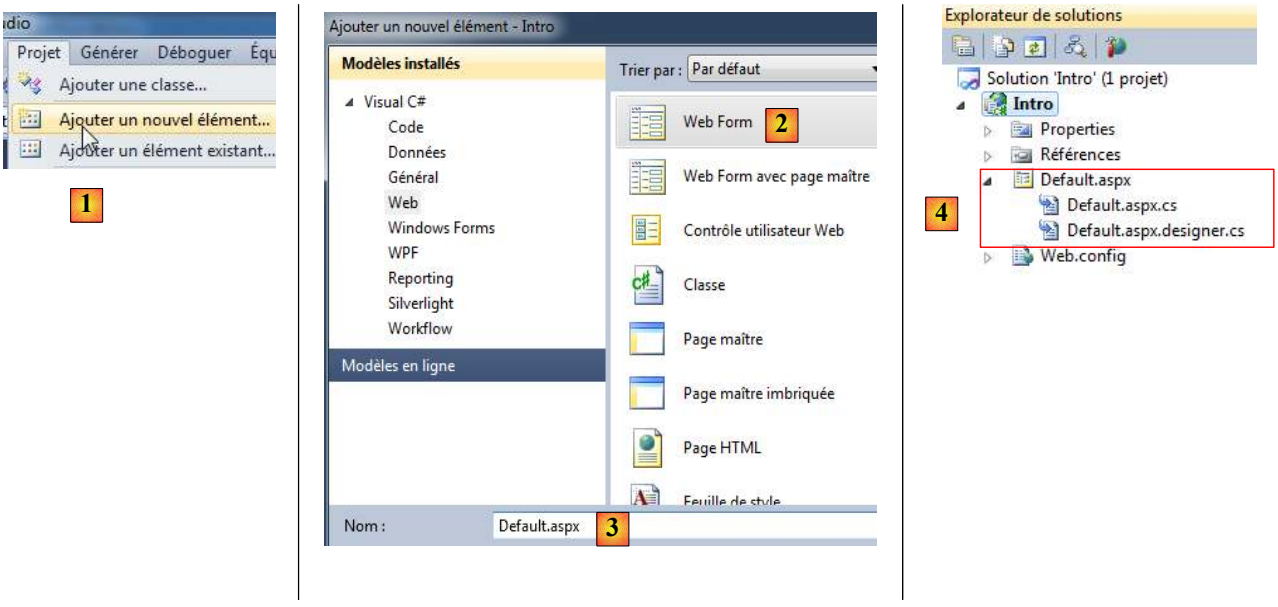
- en [4], on donne un nom à l'application. Un dossier sera créé pour le projet avec ce nom.
- en [5], on indique le dossier parent du dossier [4] du projet



- en [6], le projet créé
- [Web.config] est le fichier de configuration du projet ASP.NET.
- [References] est la liste des Dll utilisées par le projet web. Ces Dll sont des bibliothèques de classes que le projet est amené à utiliser. En [7] la liste des Dll mises par défaut dans les références du projet. La plupart sont inutiles. Si le projet doit utiliser une Dll non listée en [7], celle-ci peut être ajoutée par [8].

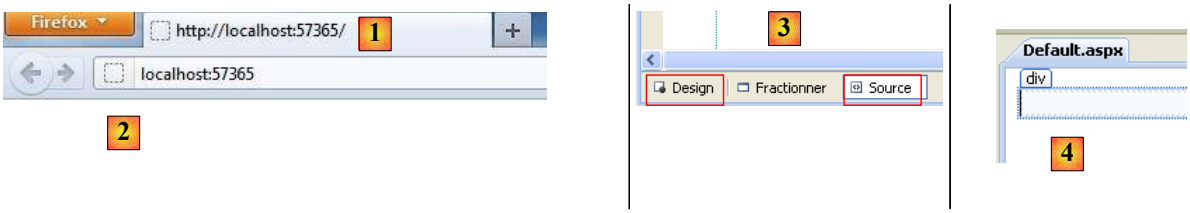
2.2 La page [Default.aspx]

Créons notre première page web :

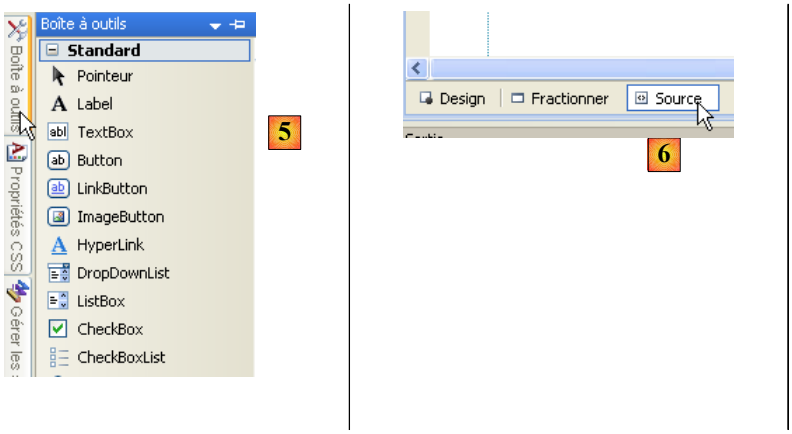


- en [1], nous ajoutons un formulaire web [2]
- en [3], nous le nommons [Default.aspx]. Cette page est particulière. C'est elle qui est servie lorsqu'un utilisateur fait une requête à l'application sans préciser de page. Elle contient des balises HTML et des balises ASP.NET
- en [4], les nouveaux fichiers du projet
- [Default.aspx.cs] contient le code de gestion des événements provoqués par l'utilisateur sur la page [Default.aspx] affichée dans son navigateur
- [Default.aspx.designer.cs] contient la liste des composants ASP.NET de la page [Default.aspx]. Chaque composant ASP.NET déposé sur la page [Default.aspx] donne naissance à la déclaration de ce composant dans [Default.aspx.designer.cs].

Si on exécute le projet par [Ctrl-F5], la page [Default.aspx] est affichée dans un navigateur :



- en [1], l'Url du projet web. Visual Studio a un serveur web intégré qui est lancé lorsqu'on demande l'exécution d'un projet web. Il écoute sur un port aléatoire, ici 57365. Le port d'écoute est habituellement le port 80. En [1], aucune page n'est demandée. Dans ce cas, c'est la page [Default.aspx] qui est affichée, d'où son nom de page par défaut.
- en [2], la page [Default.aspx] est vide.
- dans Visual Studio, la page [Default.aspx] [3] peut être construite visuellement (onglet [Design]) ou à l'aide de balises (onglet [Source])
- en [4], la page [Default.aspx] en mode [Design]. On la construit en y déposant des composants que l'on trouve dans la boîte à outils [5]. Si cette boîte n'est pas visible, on l'obtient par le menu [Affichage / Boîte à outils].



Le mode [Source] [6] donne accès au code source de la page :

```
1.  <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
    Inherits="Intro._Default" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head runat="server">
6.    <title></title>
7.  </head>
8.  <body>
9.    <form id="form1" runat="server">
10.   <div>
11.   </div>
12.   </form>
13. </body>
14. </html>
```

- la ligne 1 est une directive ASP.NET qui liste certaines propriétés de la page
  - la directive *Page* s'applique à une page web. Il y a d'autres directives telles que *Application*, *WebService*, ... qui s'applique à d'autres objets ASP.NET
  - l'attribut *CodeBehind* indique le fichier qui gère les événements de la page
  - l'attribut *Language* indique le langage .NET utilisé par le fichier *CodeBehind*
  - l'attribut *Inherits* indique le nom de la classe définie à l'intérieur du fichier *CodeBehind*
  - l'attribut *AutoEventWireUp="true"* indique que la liaison entre un événement dans [Default.aspx] et son gestionnaire dans [Defaul.aspx.cs] se fait par le nom de l'événement. Ainsi l'événement *Load* sur la page [Default.aspx] sera traité par la méthode *Page_Load* de la classe *Intro._Default* définie par l'attribut *Inherits*.
- les lignes 4-14 décrivent la page [Default.aspx] à l'aide de balises :
  - Html classiques telles que la balise <*body*> ou <*div*>

- ASP.NET. Ce sont les balises qui ont l'attribut *runat="server"*. Les balises ASP.NET sont traitées par le serveur web avant envoi de la page au client. Elles sont transformées en balises Html. Le navigateur client reçoit donc une page Html standard dans laquelle il n'existe plus de balises ASP.NET.

La page [Default.aspx] peut être modifiée directement à partir de son code source. C'est parfois plus simple que de passer par le mode [Design]. Nous modifions le code source de la façon suivante :

```
1.  <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
    Inherits="Intro._Default" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head runat="server">
6.    <title>Introduction ASP.NET</title>
7.  </head>
8.  <body>
9.    <h3>Introduction à ASP.NET</h3>
10.   <form id="form1" runat="server">
11.   <div>
12.   </div>
13.   </form>
14. </body>
15. </html>
```

En ligne 6, nous donnons un titre à la page grâce à la balise Html <title>. En ligne 9, nous introduisons un texte dans le corps (<body>) de la page. Si nous exécutons le projet (Ctrl-F5), nous obtenons le résultat suivant dans le navigateur :

Introduction à ASP.NET

2.3 Les fichiers [Default.aspx.designer.cs] et [Default.aspx.cs]

Le fichier [Default.aspx.designer.cs] déclare les composants de la page [Default.aspx] :

```
1.  //------------------------------------------------------------------------------
2.  // <auto-generated>
3.  //     Ce code a été généré par un outil.
4.  //     Version du runtime :2.0.50727.3603
5.  //
6.  //     Les modifications apportées à ce fichier peuvent provoquer un comportement incorrect et seront perdues si
7.  //     le code est régénéré.
8.  // </auto-generated>
9.  //------------------------------------------------------------------------------
10.
11. namespace Intro {
12.
13.
14.     public partial class _Default {
15.
16.         /// <summary>
17.         /// Contrôle form1.
18.         /// </summary>
19.         /// <remarks>
20.         /// Champ généré automatiquement.
21.         /// Pour modifier, déplacez la déclaration de champ du fichier de concepteur dans le fichier code-
    behind.
22.         /// </remarks>
23.         protected global::System.Web.UI.HtmlControls.HtmlForm form1;
24.     }
25. }
```

On trouve dans ce fichier la liste des composants ASP.NET de la page [Default.aspx] ayant un identifiant. Ils correspondent aux balises de [Default.aspx] ayant l'attribut *runat="server"* et l'attribut *id*. Ainsi le composant de la ligne 23 ci-dessus correspond à la balise

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

de [Default.aspx].

Le développeur interagit peu avec le fichier [Default.aspx.designer.cs]. Néanmoins ce fichier est utile pour connaître la classe d'un composant particulier. Ainsi on voit ci-dessous que le composant *form1* est de type *HtmlForm*. Le développeur peut alors explorer cette classe pour en connaître les propriétés et méthodes. Les composants de la page [Default.aspx] sont utilisés par la classe du fichier [Default.aspx.cs] :

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Web;
5.  using System.Web.UI;
6.  using System.Web.UI.WebControls;
7.
8.  namespace Intro
9.  {
10.   public partial class _Default : System.Web.UI.Page
11.   {
12.     protected void Page_Load(object sender, EventArgs e)
13.     {
14.
15.     }
16.   }
17. }
```

On notera que la classe définie dans les fichiers [Default.aspx.cs] et [Default.aspx.designer.cs] est la même (ligne 10) : *Intro._Default*. C'est le mot clé *partial* qui rend possible d'étendre la déclaration d'une classe sur plusieurs fichiers, ici deux.

Ligne 10, ci-dessus, on voit que la classe [_Default] étend la classe [Page] et hérite de ses événements. L'un d'entre-eux est l'événement *Load* qui se produit lorsque la page est chargée par le serveur web. Ligne 12, la méthode *Page_Load* qui gère l'événement *Load* de la page. C'est généralement ici qu'on initialise la page avant son affichage dans le navigateur du client. Ici, la méthode *Page_Load* ne fait rien.

La classe associée à une page web, ici la classe *Intro._Default*, est créée au début de la requête du client et détruite lorsque la réponse au client a été envoyée. Elle ne peut donc servir à mémoriser des informations entre deux requêtes. Pour cela il faut utiliser la notion de *session utilisateur*.

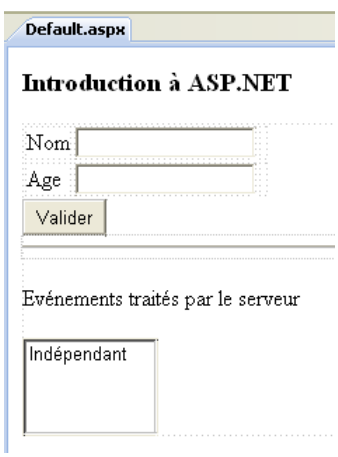
2.4 Les événements d'une page web ASP.NET

Nous construisons la page [Default.aspx] suivante :

```
1.  <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
    Inherits="Intro._Default" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head runat="server">
6.    <title>Introduction ASP.NET</title>
7.  </head>
8.  <body>
9.    <h3>Introduction à ASP.NET</h3>
10.   <form id="form1" runat="server">
11.   <div>
12.     <table>
13.       <tr>
14.         <td>
15.           Nom</td>
16.         <td>
17.           <asp:TextBox ID="TextBoxNom" runat="server"></asp:TextBox>
18.         </td>
19.         <td>
20.            </td>
21.       </tr>
22.       <tr>
23.         <td>
24.           Age</td>
25.         <td>
```

```
26.           <asp:TextBox ID="TextBoxAge" runat="server"></asp:TextBox>
27.         </td>
28.         <td>
29.            </td>
30.       </tr>
31.     </table>
32.   </div>
33.   <asp:Button ID="ButtonValider" runat="server" Text="Valider" />
34.   <hr />
35.   <p>
36.     Evénements traités par le serveur</p>
37.   <p>
38.     <asp:ListBox ID="ListBoxEvts" runat="server"></asp:ListBox>
39.   </p>
40.   </form>
41. </body>
42. </html>
```

Cette page peut être obtenue en modifiant directement le code source de la page ou en dessinant la page en mode [Design]. On trouvera les composants nécessaires dans la boîte à outils sous le thème *Standard*.



Le fichier [Default.aspx.designer.cs] est le suivant :

```
1.  namespace Intro {
2.      public partial class _Default {
3.          protected global::System.Web.UI.HtmlControls.HtmlForm form1;
4.          protected global::System.Web.UI.WebControls.TextBox TextBoxNom;
5.          protected global::System.Web.UI.WebControls.TextBox TextBoxAge;
6.          protected global::System.Web.UI.WebControls.Button ButtonValider;
7.          protected global::System.Web.UI.WebControls.ListBox ListBoxEvts;
8.      }
9.  }
```

On y retrouve tous les composants ASP.NET de la page [Default.aspx] ayant un identifiant.

Nous faisons évoluer le fichier [Default.aspx.cs] comme suit :

```
1.  using System;
2.
3.  namespace Intro
4.  {
5.    public partial class _Default : System.Web.UI.Page
6.    {
7.      protected void Page_Init(object sender, EventArgs e)
8.      {
9.        // on note l'événement
10.       ListBoxEvts.Items.Insert(0, string.Format("{0}: Page_Init",
    DateTime.Now.ToString("hh:mm:ss")));
11.     }
12.
13.     protected void Page_Load(object sender, EventArgs e)
14.     {
```

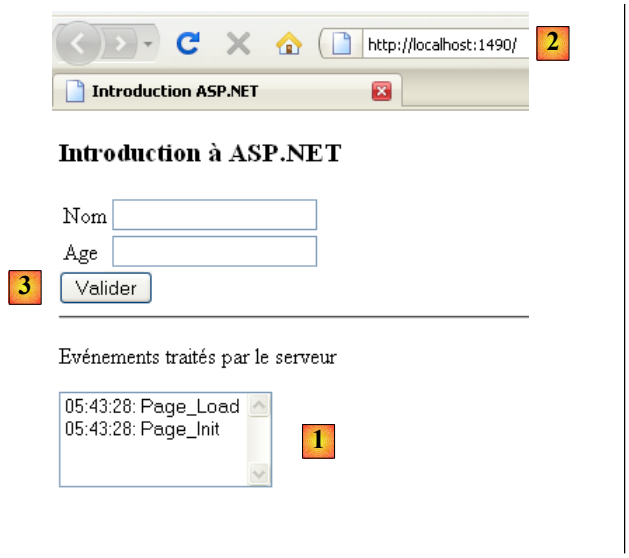
```
15.       // on note l'événement
16.       ListBoxEvts.Items.Insert(0, string.Format("{0}: Page_Load",
    DateTime.Now.ToString("hh:mm:ss")));
17.     }
18.
19.     protected void ButtonValider_Click(object sender, EventArgs e)
20.     {
21.       // on note l'événement
22.       ListBoxEvts.Items.Insert(0, string.Format("{0}: ButtonValider_Click",
    DateTime.Now.ToString("hh:mm:ss")));
23.     }
24.   }
25. }
```

La classe [_Default] (ligne 5) traite trois événements :

- l'événement **Init** (ligne 7) qui se produit lorsque la page a été initialisée
- l'événement **Load** (ligne 13) qui se produit lorsque la page a été chargée par le serveur web. L'événement *Init* se produit avant l'événement *Load*.
- l'événement **Click** sur le bouton **ButtonValider** (ligne 19) qui se produit lorsque l'utilisateur clique sur le bouton [Valider]

La gestion de chacun de ces trois événements consiste à ajouter un message au composant *Listbox* nommé *ListBoxEvts*. Ce message affiche l'heure de l'événement et le nom de celui-ci. Chaque message est placé en début de liste. Aussi les messages placés en haut de la liste sont les plus récents.

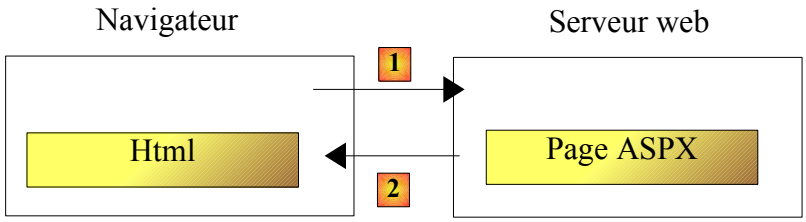
Lorsqu'on exécute le projet, on obtient la page suivante :



On voit en [1] que les événements *Page_Init* et *Page_Load* se sont produits dans cet ordre. On rappelle que l'événement le plus récent est en haut de la liste. Lorsque le navigateur demande la page [Default.aspx] directement par son Url [2], il le fait par une commande HTTP (HyperText Transfer Protocol) appelée **GET**. Une fois la page chargée dans le navigateur, l'utilisateur va provoquer des événements sur la page. Par exemple il va cliquer sur le bouton [Valider] [3]. Les événements provoqués par l'utilisateur une fois la page chargée dans le navigateur, déclenchent une requête à la page [Default.aspx] mais cette fois-ci avec une commande HTTP appelée **POST**. Pour résumer :

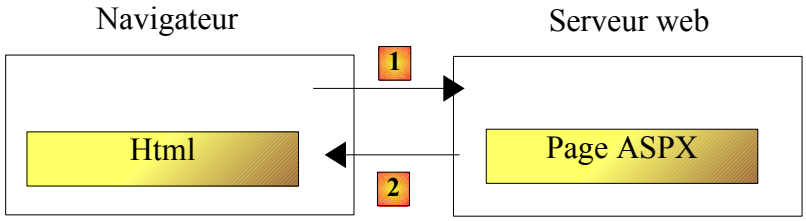
- le chargement initial d'une page P dans un navigateur est faite par une opération HTTP GET
- les événements qui se produisent ensuite sur la page produisent à chaque fois une nouvelle requête vers la même page P mais cette fois avec une commande HTTP POST. Il est possible pour une page P de savoir si elle a été demandée avec une commande GET ou une commande POST, ce qui lui permet de se comporter différemment si c'est nécessaire, ce qui est la plupart du temps le cas.

Demande initiale d'une page ASPX : GET



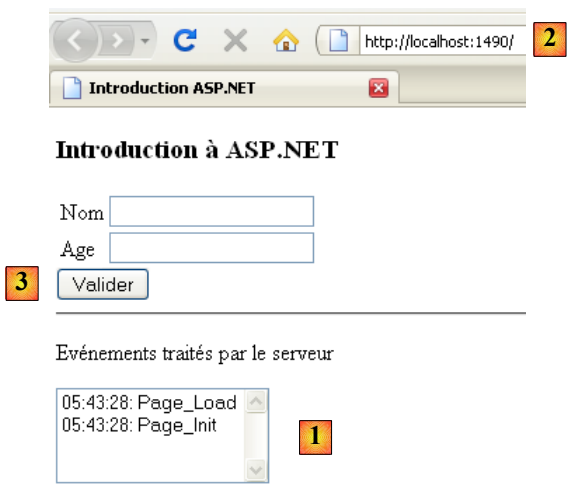
- en [1], le navigateur demande la page ASPX via une commande HTTP **GET** sans paramètres.
- en [2], le serveur web lui envoie en réponse le flux HTML traduction de la page ASPX demandée.

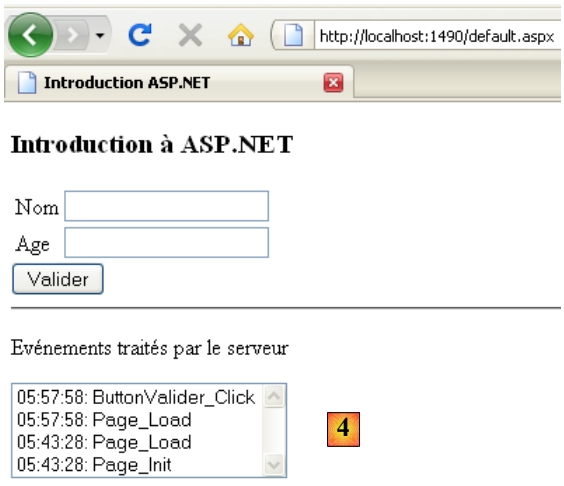
Traitement d'un événement produit sur la page affichée par le navigateur : POST



- en [1], lors d'un événement sur la page Html, le navigateur demande la page ASPX déjà acquise avec une opération GET, cette fois avec une commande HTTP **POST** accompagnée de paramètres. Ces paramètres sont les valeurs des composants qui se trouvent à l'intérieur de la balise <form> de la page HTML affichée par le navigateur. On appelle ces valeurs, les valeurs **postées** par le client. Elles vont être exploitées par la page ASPX pour traiter la demande du client.
- en [2], le serveur web lui envoie en réponse le flux HTML traduction de la page ASPX demandée initialement par le POST ou bien d'une autre page s'il y a eu **transfert** de page ou **redirection** de page.

Revenons à notre page exemple :





- en [2], la page a été obtenue par un GET.
- en [1], on voit les deux événements qui se sont produits lors de ce GET

Si, ci-dessus, l'utilisateur clique sur le bouton [Valider] [3], la page [Default.aspx] va être demandée avec un POST. Ce POST sera accompagné de paramètres qui seront les valeurs de tous les composants inclus dans la balise <form> de la page [Default.aspx] : les deux TextBox [TextBoxNom, TextBoxAge], le bouton [ButtonValider], la liste [ListBoxEvts]. Les valeurs postées pour les composants sont les suivantes :

- *TextBox* : la valeur saisie
- *Button* : le texte du bouton, ici le texte "Valider"
- *Listbox* : le texte du message sélectionné dans le ListBox

En réponse du POST, on obtient la page [4]. C'est de nouveau la page [Default.aspx]. C'est le comportement normal, à moins qu'il y ait transfert ou redirection de page par les gestionnaires d'événements de la page. On peut voir que deux nouveaux événements se sont produits :

- l'événement *Page_Load* qui s'est produit lors du chargement de la page
- l'événement *ButtonValider_Click* qui s'est produit à cause du clic sur le bouton [Valider]

On peut remarquer que :
- l'événement *Page_Init* ne s'est pas produit sur l'opération HTTP POST alors qu'il s'était produit sur l'événement HTTP GET
- l'événement *Page_Load* se produit tout le temps, que ce soit sur un GET ou un POST. C'est dans cette méthode qu'on a en général besoin de savoir si on a affaire à un GET ou à un POST.
- à l'issue du POST, la page [Default.aspx] a été renvoyée au client avec les modifications apportées par les gestionnaires d'événements. C'est toujours ainsi. Une fois les événements d'une page P traités, cette même page P est renvoyée au client. Il y a deux façons d'échapper à cette règle. Le dernier gestionnaire d'événement exécuté peut
  - transférer le flux d'exécution à une autre page P2.
  - rediriger le navigateur client vers une autre page P2.

  Dans les deux cas, c'est la page P2 qui est renvoyée au navigateur. Les deux méthodes présentent des différences sur lesquelles nous reviendrons.
- l'événement *ButtonValider_Click* s'est produit après l'événement *Page_Load*. C'est donc ce gestionnaire qui peut prendre la décision du transfert ou de la redirection vers une page P2.
- la liste des événements [4] a gardé les deux événements affichés lors du chargement initial GET de la page [Default.aspx]. C'est surprenant lorsqu'on sait que la page [Default.aspx] a été recréée lors du POST. On devrait retrouver la page [Default.aspx] avec ses valeurs de conception avec donc un *ListBox* vide. L'exécution des gestionnaires *Page_Load* et *ButtonValider_Click* devrait y mettre ensuite deux messages. Or on en trouve quatre. C'est le mécanisme du **VIEWSTATE** qui explique cela. Lors du GET initial, le serveur web envoie la page [Default.aspx] avec une balise HTML *<input type="hidden" ...>* appelé **champ caché** (ligne 10 ci-dessous).

```
1.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head><title>
4.          Introduction ASP.NET
5.  </title></head>
6.  <body>
7.    <h3>Introduction à ASP.NET</h3>
8.    <form name="form1" method="post" action="default.aspx" id="form1">
9.  <div>
10. <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
    value="/wEPDwUKLTMzMTEyNDMxMg9kFgICAw9kFgICBw8QZBAVAhMwNjoxNjozNjogUGFnZV9Mb2FkEzA2OjE2OjM2OiBQYWd
    lX0luaXQVAhMwNjoxNjozNjogUGFnZV9Mb2FkEzA2OjE2OjM2OiBQYWdlX0luaXQUKwMCZ2dkZGRW1AnTL8f/q7h2MXBLxctKD
    1UKfg==" />
11. </div>
12. ..............................
```

Dans le champ d'id "__VIEWSTATE" le serveur web met sous forme codée la valeur de tous les composants de la page. Il le fait aussi bien sur le GET initial que sur les POST qui suivent. Lorsqu'un POST sur une page P se produit :

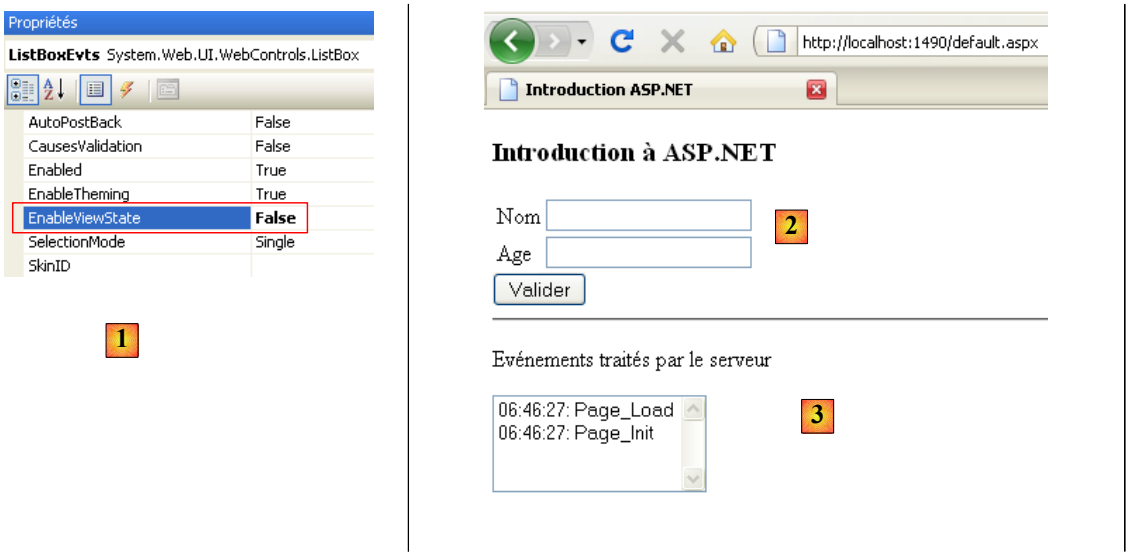
- le navigateur demande la page P en envoyant dans sa requête les valeurs de tous les composants qui sont à l'intérieur de la balise <form>. Ci-dessus, on peut voir que le composant "__VIEWSTATE" est à l'intérieur de la balise <form>. Sa valeur est donc envoyée au serveur lors d'un POST.
- la page P est instanciée et initialisée avec ses valeurs de construction
- le composant "__VIEWSTATE" est ensuite utilisé pour redonner aux composants les valeurs qu'ils avaient lorsque la page P avait été envoyée précédemment. C'est ainsi par exemple, que la liste des événements [4] retrouve les deux premiers messages qu'elle avait lorsqu'elle a été envoyée en réponse au GET initial du navigateur.
- les composants de la page P prennent ensuite pour valeurs, les valeurs postées par le navigateur. A ce moment là, le formulaire de la page P est dans l'état où l'utilisateur l'a posté.
- l'événement *Page_Load* est traité. Ici il rajoute un message à la liste des événements[4].
- l'événement qui a provoqué le POST est traité. Ici *ButtonValider_Click* rajoute un message à la liste des événements[4].
- la page P est renvoyée. Les composants ont pour valeur :

- soit la valeur postée, c.a.d. la valeur que le composant avait dans le formulaire lorsque celui a été posté au serveur
- soit une valeur donnée par l'un des gestionnaires d'événements.

Dans notre exemple,

- les deux composants *TextBox* retrouveront leur valeur postée car les gestionnaires d'événements n'y touchent pas
- la liste des événements [4] retrouve sa valeur postée, c.a.d. tous les événements déjà inscrits dans la liste, plus deux nouveaux événements créés par les méthodes *Page_Load* et *ButtonValider_Click*.

Le mécanisme du VIEWSTATE peut être activé ou inhibé au niveau de chaque composant. Inhibons-le pour le composant [ListBoxEvts] (clic droit sur ListBoxEvts / Propriétés) :

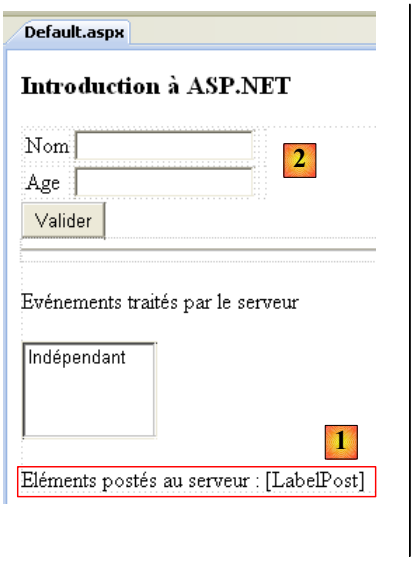


- en [1], le VIEWSTATE du composant [ListBoxEvts] est inhibé. Celui des TextBox [2] est activé par défaut.
- en [3], les deux événements renvoyés après le GET initial

- en [4], on a rempli le formulaire et on clique sur le bouton [Valider]. Un POST vers la page [Default.aspx] va être fait.
- en [6], le résultat renvoyé après un Clic sur le bouton [Valider]
  - le mécanisme du VIEWSTATE activé explique que les TextBox [7] aient gardé leur valeur postée en [4]
  - le mécanisme du VIEWSTATE inhibé explique que le composant [ListBoxEvts] [8] n'ait pas gardé son contenu [5].

2.5 Gestion des valeurs postées

Nous allons nous intéresser ici aux valeurs postées par les deux *TextBox* lorsque l'utilisateur clique sur le bouton [Valider]. La page [Default.aspx] en mode [Design] évolue comme suit :



Le code source de l'élément rajouté en [1] est le suivant :

```
1.    <p>
2.      Eléments postés au serveur :
3.      <asp:Label ID="LabelPost" runat="server"></asp:Label>
4.  </p>
```

Nous utiliserons le composant [LabelPost] pour afficher les valeurs saisies dans les deux *TextBox* [2]. Le code du gestionnaire d'événements [Default.aspx.cs] évolue comme suit :

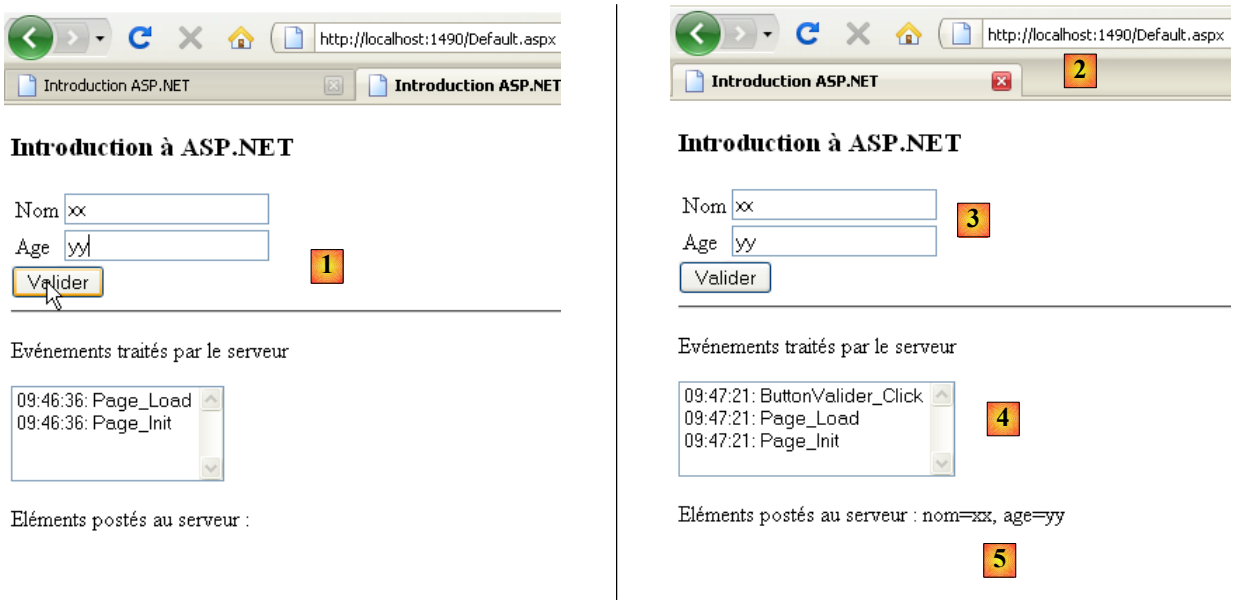
```
1.  using System;
2.
3.  namespace Intro
4.  {
5.    public partial class _Default : System.Web.UI.Page
6.    {
7.      protected void Page_Init(object sender, EventArgs e)
8.      {
9.        // on note l'événement
10.       ListBoxEvts.Items.Insert(0, string.Format("{0}: Page_Init",
    DateTime.Now.ToString("hh:mm:ss")));
11.     }
12.
13.     protected void Page_Load(object sender, EventArgs e)
14.     {
15.       // on note l'événement
16.       ListBoxEvts.Items.Insert(0, string.Format("{0}: Page_Load",
    DateTime.Now.ToString("hh:mm:ss")));
17.     }
18.
19.     protected void ButtonValider_Click(object sender, EventArgs e)
20.     {
21.       // on note l'événement
22.       ListBoxEvts.Items.Insert(0, string.Format("{0}: ButtonValider_Click",
    DateTime.Now.ToString("hh:mm:ss")));
23.       // on affiche le nom et l'âge
24.       LabelPost.Text = string.Format("nom={0}, age={1}", TextBoxNom.Text.Trim(),
    TextBoxAge.Text.Trim());
25.     }
26.   }
27. }
```

Ligne 24, on met à jour le composant *LabelPost* :

- *LabelPost* est de type [System.Web.UI.WebControls.Label] (cf Default.aspx.designer.cs). Sa propriété *Text* représente le texte affiché par le composant.
- *TextBoxNom* et *TextBoxAge* sont de type [System.Web.UI.WebControls.TextBox]. La propriété *Text* d'un composant *TextBox* est le texte affiché dans la zone de saisie.
- la méthode *Trim()* élimine les espaces qui peuvent précéder ou suivre une chaîne de caractères

Comme il a été expliqué précédemment, lorsque la méthode *ButtonValider_Click* est exécutée, les composants de la page ont la valeur qu'ils avaient lorsque la page a été postée par l'utilisateur. Les propriétés *Text* des deux *TextBox* ont donc pour valeur les textes saisis par l'utilisateur dans le navigateur.

Voici un exemple :



- en [1], les valeurs postées
- en [2], la réponse du serveur.
- en [3], les *TextBox* ont retrouvé leur valeur postée par le mécanisme du VIEWSTATE activé
- en [4], les messages du composant *ListBoxEvts* sont issus des méthodes *Page_Init*, *Page_Load*, *ButtonValider_Click* et d'un VIEWSTATE inhibé
- en [5], le composant *LabelPost* a obtenu sa valeur par la méthode *ButtonValider_Click*. On a bien récupéré les deux valeurs saisies par l'utilisateur dans les deux *TextBox* [1].

On voit ci-dessus que la valeur postée pour l'âge est la chaîne "yy", une valeur illégale. Nous allons rajouter à la page des composants appelés **validateurs**. Ils servent à vérifier la validité de données postées. Cette validité peut être vérifiée à deux endroits :

- **sur le client**. Une option de configuration du validateur permet de demander ou non que les tests soient faits sur le navigateur. Ils sont alors faits par du code JavaScript embarqué dans la page Html. Lorsque l'utilisateur fait un POST des valeurs saisies dans le formulaire, celles-ci sont tout d'abord vérifiées par le code Javascript. Si l'un des tests échoue, le POST n'est pas effectué. On évite ainsi un aller-retour avec le serveur rendant ainsi la page plus réactive.
- **sur le serveur**. Si les tests côté client peuvent être facultatifs, côté serveur ils sont obligatoires qu'il y ait eu vérification ou non côté client. En effet, lorsqu'une page reçoit des valeurs postées, elle n'a pas possibilité de savoir si elles ont été vérifiées par le client avant leur envoi. Côté serveur, le développeur doit donc **toujours** vérifier la validité des données postées.

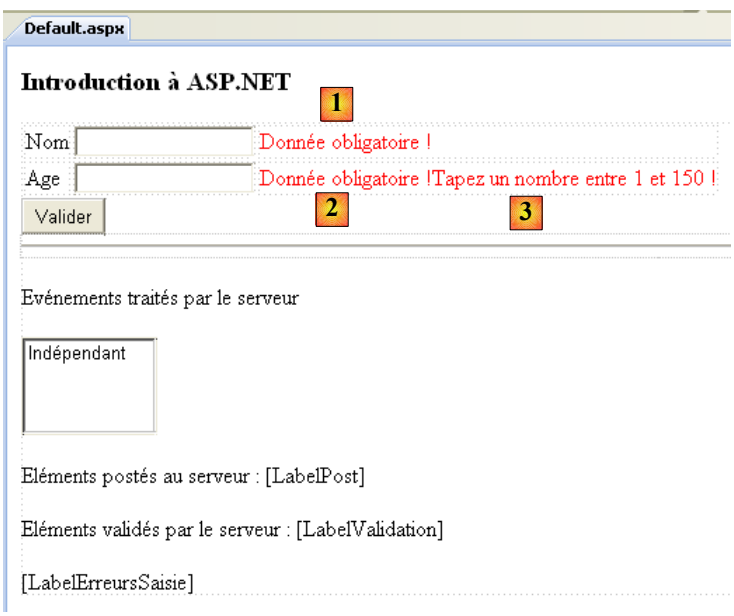
La page [Default.aspx] évolue comme suit :

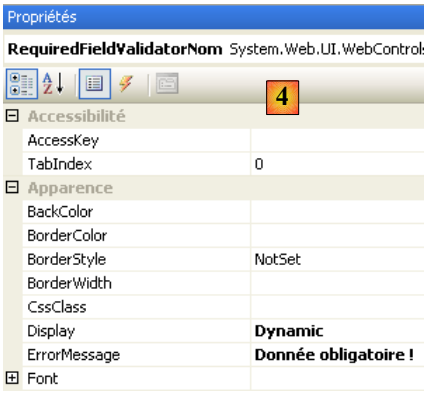
```
1. <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
   Inherits="Intro._Default" %>
2.
3. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4. <html xmlns="http://www.w3.org/1999/xhtml">
5. <head runat="server">
```

```
6.    <title>Introduction ASP.NET</title>
7.  </head>
8.  <body>
9.    <h3>Introduction à ASP.NET</h3>
10.   <form id="form1" runat="server">
11.   <div>
12.     <table>
13.       <tr>
14.         <td>
15.           Nom</td>
16.         <td>
17.           <asp:TextBox ID="TextBoxNom" runat="server"></asp:TextBox>
18.         </td>
19.         <td>
20.           <asp:RequiredFieldValidator ID="RequiredFieldValidatorNom" runat="server"
21.             ControlToValidate="TextBoxNom" Display="Dynamic"
22.             ErrorMessage="Donnée obligatoire !"></asp:RequiredFieldValidator>
23.         </td>
24.       </tr>
25.       <tr>
26.         <td>
27.           Age</td>
28.         <td>
29.           <asp:TextBox ID="TextBoxAge" runat="server"></asp:TextBox>
30.         </td>
31.         <td>
32.           <asp:RequiredFieldValidator ID="RequiredFieldValidatorAge" runat="server"
33.             ControlToValidate="TextBoxAge" Display="Dynamic"
34.             ErrorMessage="Donnée obligatoire !"></asp:RequiredFieldValidator>
35.           <asp:RangeValidator ID="RangeValidatorAge" runat="server"
36.             ControlToValidate="TextBoxAge" Display="Dynamic"
37.             ErrorMessage="Tapez un nombre entre 1 et 150 !" MaximumValue="150"
38.             MinimumValue="1" Type="Integer"></asp:RangeValidator>
39.         </td>
40.       </tr>
41.     </table>
42.   </div>
43.   <asp:Button ID="ButtonValider" runat="server" onclick="ButtonValider_Click"
44.     Text="Valider" CausesValidation="False"/>
45.   <hr />
46.   <p>
47.     Evénements traités par le serveur</p>
48.   <p>
49.     <asp:ListBox ID="ListBoxEvts" runat="server" EnableViewState="False">
50.     </asp:ListBox>
51.   </p>
52.   <p>
53.     Eléments postés au serveur :
54.     <asp:Label ID="LabelPost" runat="server"></asp:Label>
55.   </p>
56.   <p>
57.     Eléments validés par le serveur :
58.     <asp:Label ID="LabelValidation" runat="server"></asp:Label>
59.   </p>
60.    <asp:Label ID="LabelErreursSaisie" runat="server" ForeColor="Red"></asp:Label>
61.   </form>
62. </body>
63. </html>
```

Les validateurs ont été rajoutés aux lignes 20, 32 et 35. Ligne 58, un composant *Label* est utilisé pour afficher les valeurs postées valides. Ligne 60, un un composant *Label* est utilisé pour afficher un message d'erreur s'il y a des erreurs de saisie.

La page [Default.aspx] peut être également construite en mode [Design]. Les nouveaux composants peuvent être trouvés dans la boîte à outils sous le thème *Validation*.





- les composants [1] et [2] sont de type *RequiredFieldValidator*. Ce validateur vérifie qu'un champ de saisie est non vide.
- le composant [3] est de type *RangeValidator*. Ce validateur vérifie qu'un champ de saisie contient une valeur entre deux bornes.
- en [4], les propriétés du validateur [1].

Nous allons présenter les deux types de validateurs au travers de leurs balises dans le code de la page [Default.aspx] :

```
          <asp:RequiredFieldValidator ID="RequiredFieldValidatorNom" runat="server"
            ControlToValidate="TextBoxNom" Display="Dynamic"
ErrorMessage="Donnée obligatoire !"></asp:RequiredFieldValidator>
```

- **ID** : l'identifiant du composant
- **ControlToValidate** : le nom du composant dont la valeur est vérifiée. Ici on veut que le composant *TextBoxNom* n'ait pas une valeur vide (chaîne vide ou suite d'espaces)
- **ErrorMessage** : message d'erreur à afficher dans le validateur en cas de donnée invalide.
- **EnableClientScript** : booléen indiquant si le validateur doit être exécuté également côté client. Cet attribut a la valeur *True* par défaut lorsqu'il n'est pas explicitement positionné comme ci-dessus.
- **Display** : mode d'affichage du validateur. Il y a deux modes :
  - *static* (défaut) : le validateur occupe de la place sur la page même s'il n'affiche pas de message d'erreur
  - *dynamic* : le validateur n'occupe pas de place sur la page s'il n'affiche pas de message d'erreur.

```
          <asp:RangeValidator ID="RangeValidatorAge" runat="server"
            ControlToValidate="TextBoxAge" Display="Dynamic"
            ErrorMessage="Tapez un nombre entre 1 et 150 !" MaximumValue="150"
            MinimumValue="1" Type="Integer"></asp:RangeValidator>
```

- **Type** : le type de la donnée vérifiée. Ici l'âge est un entier.
- **MinimumValue**, **MaximumValue** : les bornes dans lesquelles doit se trouver la valeur vérifiée

La configuration du composant qui provoque le POST joue un rôle dans le mode de validation. Ici ce composant est le bouton [Valider] :

```
  <asp:Button ID="ButtonValider" runat="server" onclick="ButtonValider_Click"  Text="Valider"
CausesValidation="True" />
```

- **CausesValidation** : fixe le mode automatique ou nom des validations côté serveur. Cet attribut a la valeur par défaut "*True*" s'il n'est pas explicitement mentionné. Dans ce cas,
  - côté client, les validateurs ayant *EnableClientScript* à *True* sont exécutés. Le POST n'a lieu que si tous les validateurs côté client réussissent.

- côté serveur, tous les validateurs présents sur la page sont automatiquement exécutés avant le traitement de l'événement qui a provoqué le POST. Ici, ils seraient exécutés avant l'exécution de la méthode *ButtonValider_Click*. Dans cette méthode, il est possible de savoir si toutes les validations ont réussi ou non. **Page.IsValid** est "True" si elles ont toutes réussi, "False" sinon. Dans ce dernier cas, on peut arrêter le traitement de l'événement qui a provoqué le POST. La page postée est renvoyée telle qu'elle a été saisie. Les validateurs ayant échoué affichent alors leur message d'erreur (attribut *ErrorMessage*).

Si **CausesValidation** a la valeur *False*, alors

- côté client, aucun validateur n'est exécuté
- côté serveur, c'est au développeur de demander lui-même l'exécution des validateurs de la page. Il le fait avec la méthode *Page.Validate()*. Selon le résultat des validations, cette méthode positionne la propriété **Page.IsValid** à "True" ou "False".

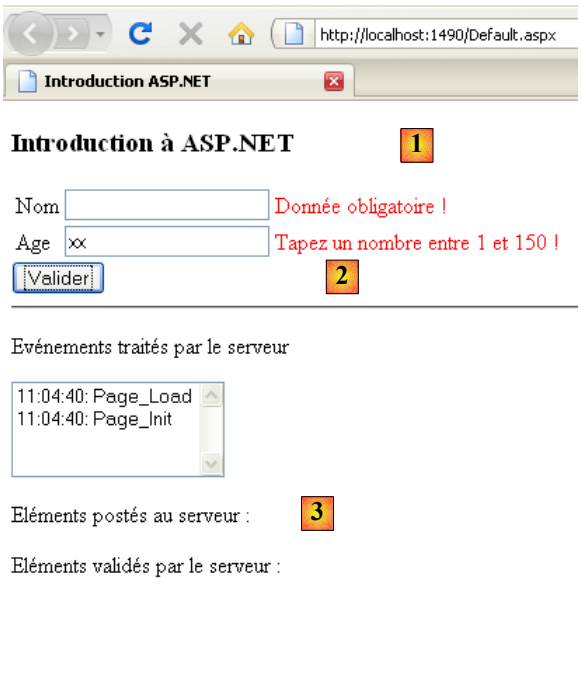
Dans [Default.aspx.cs] le code de traitement de *ButtonValider_Click* évolue comme suit :

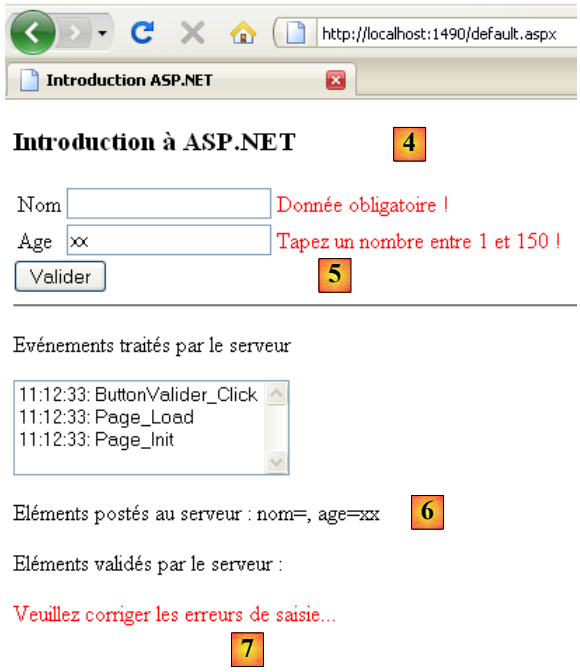
```
1.  protected void ButtonValider_Click(object sender, EventArgs e)
2.      {
3.        // on note l'événement
4.        ListBoxEvts.Items.Insert(0, string.Format("{0}: ButtonValider_Click",
    DateTime.Now.ToString("hh:mm:ss")));
5.        // on affiche le nom et l'âge
6.        LabelPost.Text = string.Format("nom={0}, age={1}", TextBoxNom.Text.Trim(),
    TextBoxAge.Text.Trim());
7.        // la page est-elle valide ?
8.        Page.Validate();
9.        if (!Page.IsValid)
10.       {
11.         // msg d'erreur global
12.         LabelErreursSaisie.Text = "Veuillez corriger les erreurs de saisie...";
13.         LabelErreursSaisie.Visible = true;
14.         return;
15.       }
16.       // on cache le msg d'erreur
17.       LabelErreursSaisie.Visible = false;
18.       // on affiche le nom et l'âge validés
19.       LabelValidation.Text = string.Format("nom={0}, age={1}", TextBoxNom.Text.Trim(),
    TextBoxAge.Text.Trim());
20. }
```

Dans le cas où le bouton [Valider] a son attribut *CausesValidation* à *True* et les validateurs leur attribut *EnableClientScript* à *True*, la méthode *ButtonValider_Click* n'est exécutée que lorsque les valeurs postées sont valides. On peut se demander alors le sens du code que l'on trouve à partir de la ligne 8. Il faut alors se rappeler qu'il est toujours possible d'écrire un client programmé qui poste des valeurs non vérifiées à la page [Default.aspx]. Donc celle-ci doit toujours refaire les tests de validité.

- ligne 8 : lance l'exécution de tous les validateurs de la page. Dans le cas où le bouton [Valider] a son attribut *CausesValidation* à *True*, ceci est fait automatiquement et il n'y a pas lieu de le refaire. Il y a ici redondance.
- lignes 9-15 : cas où l'un des validateurs a échoué
- lignes 16-19 : cas où tous les validateurs ont réussi

Voici deux exemples d'exécution :

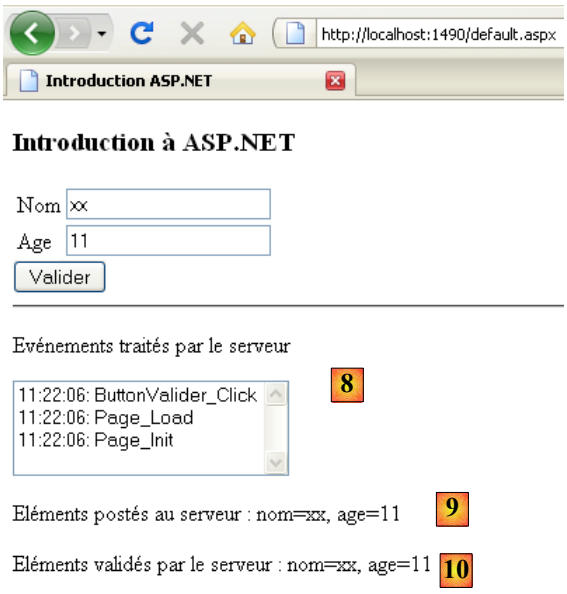




- en [1], un exemple d'exécution dans le cas où :
  - le bouton [Valider] a sa propriété *CausesValidation* à *True*
  - les validateurs ont leur propriété *EnableClientScript* à *True*

  Les messages d'erreurs [2] ont été affichés par les validateurs exécutés côté client par le code Javascript de la page. Il n'y a pas eu de POST vers le serveur comme le montre le label des éléments postés [3].
- en [4], un exemple d'exécution dans le cas où :
  - le bouton [Valider] a sa propriété *CausesValidation* à *False*
  - les validateurs ont leur propriété *EnableClientScript* à *False*

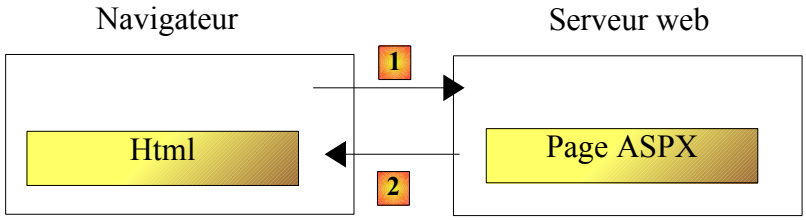
  Les messages d'erreurs [5] ont été affichés par les validateurs exécutés côté serveur. Comme le montre [6], il y a bien eu un POST vers le serveur. En [7], le message d'erreur affiché par la méthode [ButtonValider_Click] dans le cas d'erreurs de saisie.



- en [8] un exemple obtenu avec des données valides. [9,10] montrent que les éléments postés ont été validés. Lorsqu'on fait des tests répétés, il faut mettre à *False* la propriété *EnableViewState* du label [LabelValidation] afin que le message de validation ne reste pas affiché au fil des exécutions.

2.6 Gestion des données de portée Application

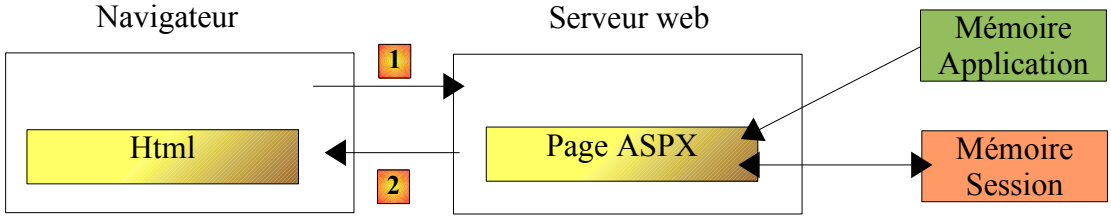
Revenons sur l'architecture d'exécution d'une page ASPX :



La classe de la page ASPX est instanciée au début de la requête du client et détruite à la la fin de celle-ci. Aussi elle ne peut servir à mémoriser des données entre deux requêtes. On peut vouloir mémoriser deux types de données :

- des données partagées par tous les utilisateurs de l'application web. Ce sont en général des données en lecture seule. Trois fichiers sont utilisés pour mettre en oeuvre ce partage de données :
  - [Web.Config] : le fichier de configuration de l'application
  - [Global.asax, Global.asax.cs] : permettent de définir une classe, appelée classe globale d'application, dont la durée de vie est celle de l'application, ainsi que des gestionnaires pour certains événements de cette même application.

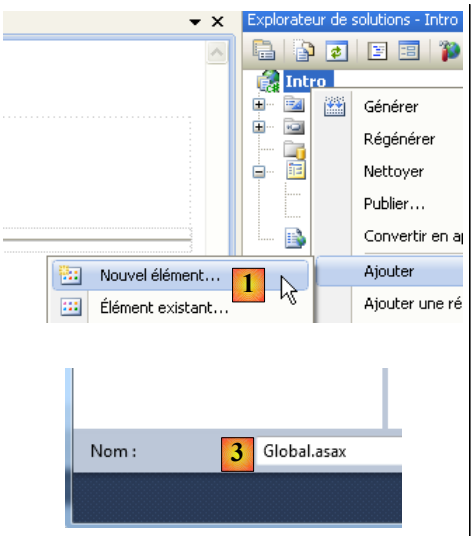
  La classe globale d'application permet de définir des données qui seront disponibles pour toutes les requêtes de tous les utilisateurs.
- des données partagées par les requêtes d'**un même client**. Ces données sont mémorisées dans un objet appelée **Session**. On parle alors de **session client** pour désigner la mémoire du client. Toutes les requêtes d'un client ont accès à cette session. Elles peuvent y stocker et y lire des informations

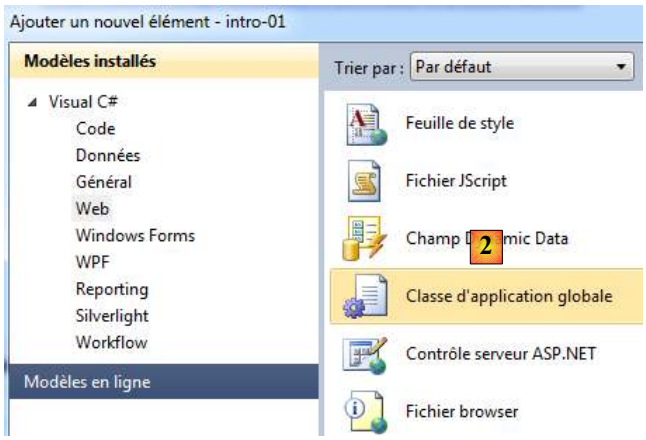


Ci-dessus, nous montrons les types de mémoire auxquels a accès une page ASPX :

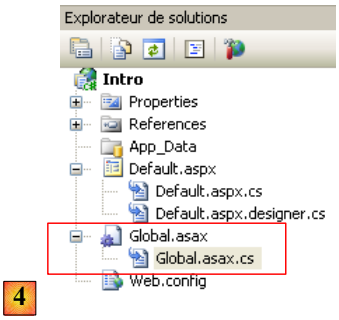
- la mémoire de l'application qui contient la plupart du temps des données en lecture seule et qui est accessible à tous les utilisateurs.
- la mémoire d'un utilisateur particulier, ou session, qui contient des données en lecture / écriture et qui est accessible aux requêtes successives d'un même utilisateur.
- non représentée ci-dessus, il existe une mémoire de requête, ou contexte de requête. La requête d'un utilisateur peut être traitée par plusieurs pages ASPX successives. Le contexte de la requête permet à une page 1 de transmettre de l'information à une page 2.

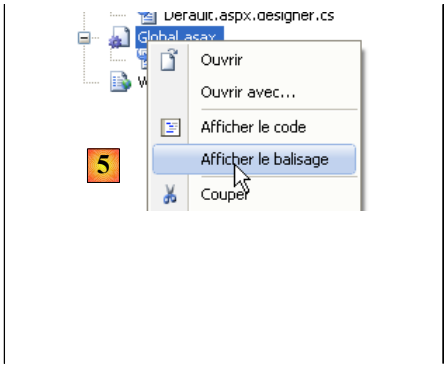
Nous nous intéressons ici aux données de portée *Application*, celles qui sont partagées par tous les utilisateurs. La classe globale de l'application peut être créée comme suit :





- en [1], on ajoute un nouvel élément au projet
- en [2], on ajoute la classe d'application globale
- en [3], on garde le nom par défaut [Global.asax] pour le nouvel élément





- en [4], deux nouveaux fichiers ont été ajoutés au projet
- en [5], on affiche le balisage de [Global.asax]

```
<%@ Application Codebehind="Global.asax.cs" Inherits="Intro.Global" Language="C#" %>
```

- la balise *Application* remplace la balise *Page* qu'on avait pour [Default.aspx]. Elle identifie la classe d'application globale
- **Codebehind** : définit le fichier dans lequel est définie la classe d'application globale
- **Inherits** : définit le nom de cette classe

La classe **Intro.Global** générée est la suivante :

```
1.  using System;
2.
3.  namespace Intro
4.  {
5.    public class Global : System.Web.HttpApplication
6.    {
7.
8.      protected void Application_Start(object sender, EventArgs e)
9.      {
10.
11.     }
12.
13.     protected void Session_Start(object sender, EventArgs e)
```

```
14.     {
15.
16.     }
17.
18.     protected void Application_BeginRequest(object sender, EventArgs e)
19.     {
20.
21.     }
22.
23.     protected void Application_AuthenticateRequest(object sender, EventArgs e)
24.     {
25.
26.     }
27.
28.     protected void Application_Error(object sender, EventArgs e)
29.     {
30.
31.     }
32.
33.     protected void Session_End(object sender, EventArgs e)
34.     {
35.
36.     }
37.
38.     protected void Application_End(object sender, EventArgs e)
39.     {
40.
41.     }
42.   }
43. }
```

- ligne 5 : la classe globale d'application dérive de la classe *HttpApplication*

La classe est générée avec des squelettes de gestionnaires d'événements de l'application :

- lignes 8, 38 : gèrent les événements *Application_Start* (démarrage de l'application) et *Application_End* (fin de l'application lorsque le serveur web s'arrête ou lorsque l'administrateur décharge l'application)
- lignes 13, 33 : gèrent les événements *Session_Start* (démarrage d'une nouvelle session client à l'arrivée d'un nouveau client ou à l'expiration d'une session existante) et *Session_End* (fin d'une session client soit explicitement par programmation soit implicitement par dépassement de la durée autorisée pour une session).
- ligne 28 : gère l'événement *Application_Error* (apparition d'une exception non gérée par le code de l'application et remontée jusqu'au serveur)
- ligne 18 : gère l'événement *Application_BeginRequest* (arrivée d'une nouvelle requête).
- ligne 23 : gère l'événement *Application_AuhenticateRequest* (se produit lorsqu'un utilisateur s'est authentifié).

La méthode [Application_Start] est souvent utilisée pour initialiser l'application à partir d'informations contenues dans [Web.Config]. Celui généré à la création initiale d'un projet a l'allure suivante :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.
3.  <!--
4.    Pour plus d'informations sur la configuration de votre application ASP.NET, consultez
5.    http://go.microsoft.com/fwlink/?LinkId=169433
6.    -->
7.
8.  <configuration>
9.      <system.web>
10.         <compilation debug="true" targetFramework="4.0" />
11.     </system.web>
12.
13. </configuration>
```

Pour notre application actuelle, ce fichier est inutile. Si on le supprime ou le renomme, l'application continue à fonctionner normalement. Lorsqu'il a été détruit, on peut le régénérer de la façon suivante : clic droit sur projet Ajouter / Nouvel élément / Fichier de configuration web.

Nous allons faire évoluer le fichier [Web.config] de la façon suivante :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.
3.  <!--
4.    Pour plus d'informations sur la configuration de votre application ASP.NET, consultez
```

```
5.    http://go.microsoft.com/fwlink/?LinkId=169433
6.    -->
7.
8.  <configuration>
9.    <system.web>
10.     <compilation debug="true" targetFramework="4.0" />
11.   </system.web>
12.   <appSettings>
13.     <add key="cle1" value="valeur1"/>
14.     <add key="cle2" value="valeur2"/>
15.   </appSettings>
16.   <connectionStrings>
17.     <add connectionString="connexion1" name="conn1"/>
18.   </connectionStrings>
19. </configuration>
```

- ligne 12 : la balise <**appsettings**> permet de définir un dictionnaire d'informations
- ligne 16 : la balise <**connectionStrings**> permet de définir des chaînes de connexion à des bases de données

Ce fichier peut être exploité par la classe globale d'application suivante [Global.asax.cs] :

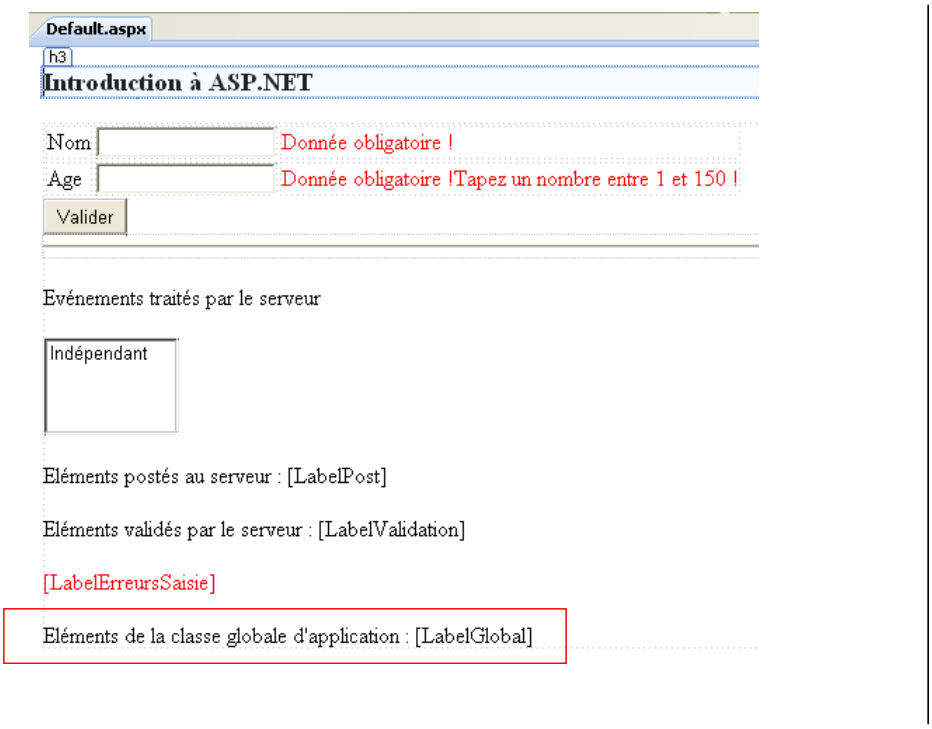
```
1.  using System;
2.  using System.Configuration;
3.
4.  namespace Intro
5.  {
6.    public class Global : System.Web.HttpApplication
7.    {
8.      public static string Param1 { get; set; }
9.      public static string Param2 { get; set; }
10.     public static string ConnString1 { get; set; }
11.     public static string Erreur { get; set; }
12.
13.     protected void Application_Start(object sender, EventArgs e)
14.     {
15.       try
16.       {
17.         Param1 = ConfigurationManager.AppSettings["cle1"];
18.         Param2 = ConfigurationManager.AppSettings["cle2"];
19.         ConnString1 = ConfigurationManager.ConnectionStrings["conn1"].ConnectionString;
20.       }
21.       catch (Exception ex)
22.       {
23.         Erreur = string.Format("Erreur de configuration : {0}", ex.Message);
24.       }
25.     }
26.
27.     protected void Session_Start(object sender, EventArgs e)
28.     {
29.
30.     }
31.
32.   }
33. }
```

- lignes 8-11 : quatre propriétés statiques P. Comme la durée de vie de la classe *Global* est celle de l'application, toute requête faite à l'application aura accès à ces propriétés P via la syntaxe *Global.P*.
- lignes 17-19 : le fichier [Web.config] est accessible via la classe [System.Configuration.ConfigurationManager]
- lignes 17-18 : récupère les éléments de la balise <*appSettings*> du fichier [Web.config] via l'attribut *key*.
- ligne 19 : récupère les éléments de la balise <*connectionStrings*> du fichier [Web.config] via l'attribut *name*.

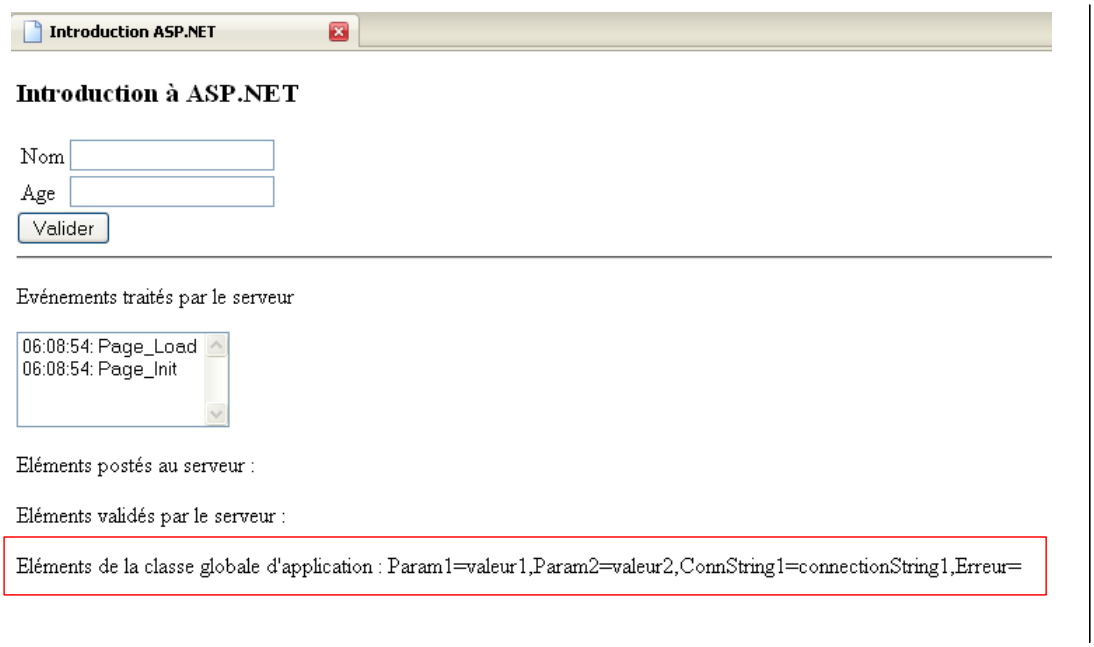
Les attributs statiques des lignes 8-11 sont accessibles de n'importe quel gestionnaire d'événement des pages ASPX chargées. Nous les utilisons dans le gestionnaire [Page_Load] de la page [Default.aspx] :

```
1.      protected void Page_Load(object sender, EventArgs e)
2.      {
3.        // on note l'événement
4.        ListBoxEvts.Items.Insert(0, string.Format("{0}: Page_Load", DateTime.Now.ToString("hh:mm:ss")));
5.        // on récupère les informations de la classe globale d'application
6.        LabelGlobal.Text = string.Format("Param1={0},Param2={1},ConnString1={2},Erreur={3}", Global.Param1,
    Global.Param2, Global.ConnString1, Global.Erreur);
7.  }
```

- ligne 6 : les quatre attributs statiques de la classe globale d'application sont utilisés pour alimenter un nouveau label de la page [Default.aspx]



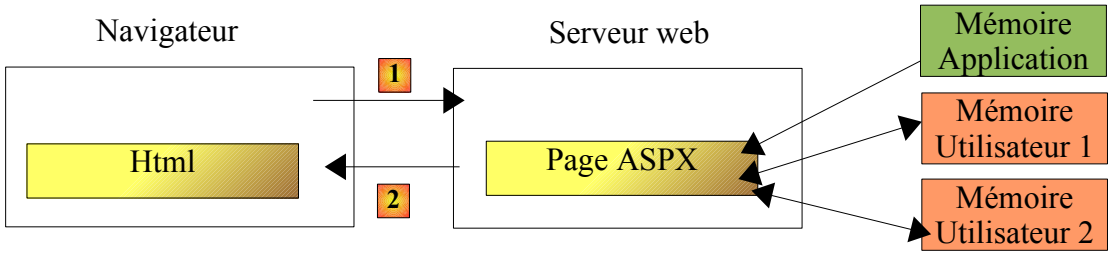
A l'exécution, nous obtenons le résultat suivant :



Ci-dessus, nous voyons que les paramètres de [Web.config] ont été correctement récupérés. La classe globale d'application est le bon endroit pour stocker des informations partagées par tous les utilisateurs.

2.7 Gestion des données de portée Session

Nous nous intéressons ici à la façon de mémoriser des informations au fil des requêtes d'un utilisateur donné :



Chaque utilisateur a sa propre mémoire qu'on appelle sa session.

Nous avons vu que la classe d'application globale disposait de deux gestionnaires pour gérer les événements :

- *Session_Start* : début d'une session
- *Session_end* : fin d'une session

Le mécanisme de la session est mis en oeuvre de la façon suivante :

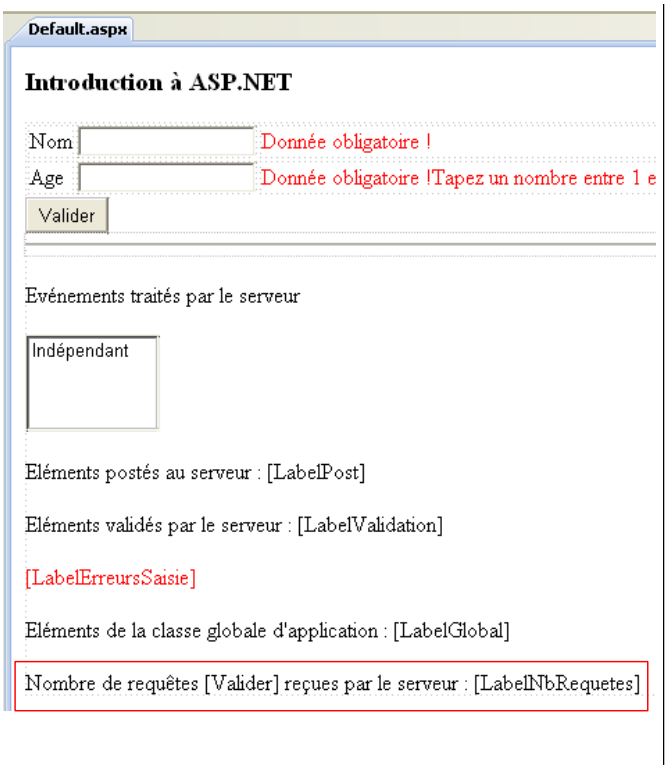
- lors de la première requête d'un utilisateur, le serveur web crée un jeton de session qu'il attribue à l'utilisateur. Ce jeton est une suite de caractères unique pour chaque utilisateur. Il est envoyé par le serveur dans la réponse faite à la première requête de l'utilisateur.
- lors des requêtes suivantes, l'utilisateur (le navigateur web) inclut dans sa requête le jeton de session qu'on lui a attribué. Aussi le serveur web est-il capable de le reconnaître.
- une session a une durée de vie. Lorsque le serveur web reçoit une requête d'un utilisateur, il calcule le temps qui s'est écoulé depuis la requête précédente. Si ce temps dépasse la durée de vie de la session, une nouvelle session est créée pour l'utilisateur. Les données de la précédente session sont perdues. Avec le serveur web IIS (Internet Information Server) de Microsoft, les sessions ont par défaut une durée de vie de 20 mn. Cette valeur peut être changée par l'administrateur du serveur web.
- le serveur web sait qu'il a affaire à la première requête d'un utilisateur parce que cette requête ne comporte pas de jeton de session. C'est la seule.

Toute page ASP.NET a accès à la session de l'utilisateur via la propriété **Session** de la page, de type [System.Web.SessionState.HttpSessionState]. Nous utiliserons les propriétés P et méthodes M suivantes de la classe *HttpSessionState* :

Nom	Type	Rôle
Item[String clé]	P	La session peut être construite comme un dictionnaire. *Item[clé]* est l'élément de la session identifié par *clé*. Au lieu d'écrire [*HttpSessionState].Item[clé]*, on peut également écrire [*HttpSessionState].[clé]*.
Clear	M	vide le dictionnaire de la session
Abandon	M	termine la session. La session n'est alors plus valide. Une nouvelle session démarrera avec la prochaine requête de l'utilisateur.

Comme exemple de mémoire utilisateur, nous allons compter le nombre de fois qu'un utilisateur clique sur le bouton [Valider]. Pour obtenir ce résultat, il faut maintenir un compteur dans la session de l'utilisateur.

La page [Default.aspx] évolue comme suit :



La classe globale d'application [Global.asax.cs] évolue comme suit :

```
1.  using System;
2.  using System.Configuration;
3.
4.  namespace Intro
5.  {
6.    public class Global : System.Web.HttpApplication
7.    {
8.      public static string Param1 { get; set; }
9.  ...
10.
11.     protected void Application_Start(object sender, EventArgs e)
12.     {
13. ...
14.     }
15.
16.     protected void Session_Start(object sender, EventArgs e)
17.     {
18.       // compteur de requêtes
19.       Session["nbRequêtes"] = 0;
20.     }
21.
22.   }
23. }
```

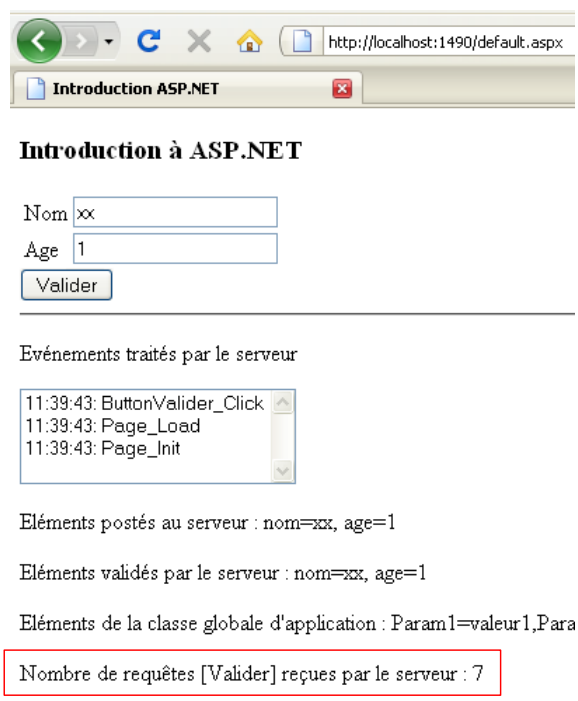
Ligne 19, on utilise la session de l'utilisateur pour y stocker un compteur de requêtes identifié par la clé *"nbRequêtes"*. Ce compteur est mis à jour par le gestionnaire [ButtonValider_Click] de la page [Default.aspx] :

```
1.  using System;
2.
3.  namespace Intro
4.  {
5.    public partial class _Default : System.Web.UI.Page
6.    {
7.  ....
8.
9.      protected void ButtonValider_Click(object sender, EventArgs e)
10.     {
11.       // on note l'événement
12.       ListBoxEvts.Items.Insert(0, string.Format("{0}: ButtonValider_Click",
    DateTime.Now.ToString("hh:mm:ss")));
13.       // on affiche le nom et l'âge postés
```

```
14.         LabelPost.Text = string.Format("nom={0}, age={1}", TextBoxNom.Text.Trim(),
    TextBoxAge.Text.Trim());
15.         // nombre de requêtes
16.         Session["nbRequêtes"] = (int)Session["nbRequêtes"] + 1;
17.         LabelNbRequetes.Text = Session["nbRequêtes"].ToString();
18.         // la page est-elle valide ?
19.         Page.Validate();
20.         if (!Page.IsValid)
21.         {
22. ...
23.         }
24. ...
25.     }
26.   }
27. }
```

- ligne 16 : on incrémente le compteur de requêtes
- ligne 17 : le compteur est affiché sur la page

Voici un exemple d'exécution :



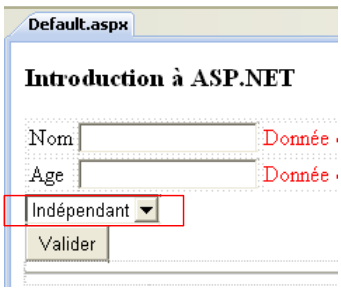
2.8 Gestion du GET / POST dans le chargement d'une page

Nous avons dit qu'il y avait deux types de requêtes vers une page ASPX :

- la requête initiale du navigateur faite avec une commande HTTP GET. Le serveur répond en envoyant la page demandée. Nous supposerons que cette page est un formulaire, c.a.d. que dans la page ASPX envoyée, il y a une balise <form runat="server"...>.
- les requêtes suivantes faites par le navigateur en réaction à certaines actions de l'utilisateur sur le formulaire. Le navigateur fait alors une requête HTTP POST.

Que ce soit sur une demande GET ou une demande POST, la méthode [Page_Load] est exécutée. Lors du GET, cette méthode est habituellement utilisée pour initialiser la page envoyée au navigateur client. Ensuite, par le mécanisme du VIEWSTATE, la page reste initialisée et n'est modifiée que par les gestionnaires des événements qui provoquent les POST. Il n'y a pas lieu de réinitialiser la page dans *Page_Load*. D'où le besoin pour cette méthode de savoir si la requête du client est un GET (la première requête) ou un POST (les suivantes).

Examinons l'exemple suivant. On ajoute une liste déroulante à la page [Default.aspx]. Le contenu de cette liste sera défini dans le gestionnaire *Page_Load* de la requête GET :



La liste déroulante est déclarée dans [Default.aspx.designer.cs] de la façon suivante :

```
        protected global::System.Web.UI.WebControls.DropDownList DropDownListNoms;
```

Nous utiliserons les méthodes M et propriétés P suivantes de la classe [DropDownList] :

Nom	Type	Rôle
Items	P	la collection de type *ListItemCollection* des éléments de type *ListItem* de la liste déroulante
SelectedIndex	P	l'index, partant de 0, de l'élément sélectionné dans la liste déroulante lorsque le formulaire est posté
SelectedItem	P	l'élément de type *ListItem* sélectionné dans la liste déroulante lorsque le formulaire est posté
SelectedValue	P	la valeur de type *string* de l'élément de type *ListItem* sélectionné dans la liste déroulante lorsque le formulaire est posté. Nous allons définir prochainement cette notion de valeur.

La classe **ListItem** des éléments d'une liste déroulante sert à générer les balises <*option*> de la balise Html <*select*> :

```
1. <select ....>
2.    <option value="val1">texte1</option>
3.    <option value="val2">texte2</option>
4. ....
5. </select>
```

Dans la balise <*option*>

- *textei* est le texte affiché dans la liste déroulante
- *vali* est la valeur postée par le navigateur si *textei* est le texte sélectionné dans la liste déroulante

Chaque option peut être générée par un objet *ListItem* construit à l'aide du constructeur *ListItem(string texte, string valeur)*.

Dans [Default.aspx.cs], le code du gestionnaire [Page_Load] évolue comme suit :

```
1.      protected void Page_Load(object sender, EventArgs e)
2.      {
3.        // on note l'événement
4.        ...
5.        // on récupère les informations de la classe globale d'application
6.        ...
7.        // initialisation du combo des noms uniquement lors du GET initial
8.        if (!Page.IsPostBack)
9.        {
10.         for (int i = 0; i < 3; i++)
11.         {
12.           DropDownListNoms.Items.Add(new ListItem("nom"+i,i.ToString()));
13.         }
14.       }
15. }
```

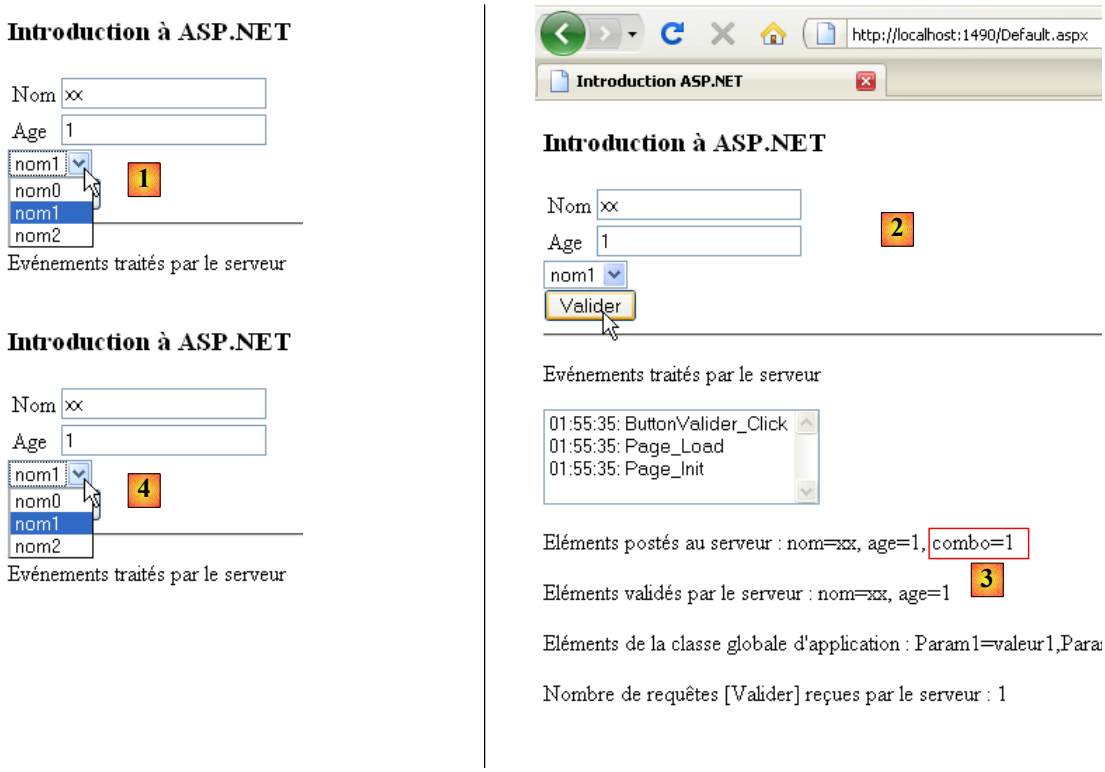
- ligne 8 : la classe *Page* a un attribut *IsPostBack* de type booléen. A vrai, il signifie que la requête de l'utilisateur est un POST. Les lignes 10-13 ne sont donc exécutées que sur le GET initial du client.

- ligne 12 : on ajoute à la liste [DropDownListNoms], un élément de type *ListItem(string texte, string value)*. Le texte affiché pour le (i+1)ème élément sera *nomi* et la valeur postée pour cet élément s'il est sélectionné sera *i*.

Le gestionnaire [ButtonValider_Click] est modifié afin d'afficher la valeur postée par la liste déroulante :

```
1.      protected void ButtonValider_Click(object sender, EventArgs e)
2.      {
3.        // on note l'événement
4.  ...
5.        // on affiche les valeurs postées
6.        LabelPost.Text = string.Format("nom={0}, age={1}, combo={2}", TextBoxNom.Text.Trim(),
    TextBoxAge.Text.Trim(), DropDownListNoms.SelectedValue);
7.        // nombre de requêtes
8.  ...
9.  }
```

Ligne 6, la valeur postée pour la liste [DropDownListNoms] est obtenue avec la propriété *SelectedValue* de la liste. Voici un exemple d'exécution :



- en [1], le contenu de la liste déroulante après le GET initial et juste avant le 1er POST
- en [2], la page après le 1er POST.
- en [3], la valeur postée pour la liste déroulante. Correspond à l'attribut *value* du *ListItem* sélectionné dans la liste.
- en [4], la liste déroulante. Elle contient les mêmes éléments qu'après le GET initial. C'est le mécanisme du VIEWSTATE qui explique cela.

Pour comprendre l'interaction entre le VIEWSTATE de la liste *DropDownListNoms* et le test *if (! IsPostBack)* du gestionnaire *Page_Load* de [Default.aspx], le lecteur est invité à refaire le test précédent avec les configurations suivantes :

Cas	DropDownListNoms.EnableViewState	test if(! IsPostBack) dans Page_Load de [Default.aspx]
1	true	présent
2	false	présent
3	true	absent
4	false	absent

Les différents tests donnent les résultats suivants :

1. c'est le cas présenté plus haut
2. la liste est remplie lors du GET initial mais pas lors des POST qui suivent. Comme *EnableViewState* est à faux, la liste est vide après chaque POST
3. la liste est remplie aussi bien après le GET initial que lors des POST qui suivent. Comme *EnableViewState* est à *vrai*, on a 3 noms après le GET initial, 6 noms après le 1er POST, 9 noms après le 2ième POST, ...
4. la liste est remplie aussi bien après le GET initial que lors des POST qui suivent. Comme *EnableViewState* est à *faux*, la liste est remplie avec seulement 3 noms à chaque requête que celle-ci soit le GET initial ou les POST qui suivent. On retrouve le comportement du cas 1. Il y a donc deux façons d'obtenir le même résultat.

2.9 Gestion du VIEWSTATE des éléments d'une page ASPX

Par défaut, tous les éléments d'une page ASPX ont leur propriété *EnableViewState* à *True*. A chaque fois que la page ASPX est envoyée au navigateur client, elle contient le champ caché __VIEWSTATE qui a pour valeur une chaîne de caractères codant l'ensemble des valeurs des composants ayant leur propriété *EnableViewState* à *True*. Pour minimiser la taille de cette chaîne, on peut chercher à réduire le nombre de composants ayant leur propriété *EnableViewState* à *True*.

Rappelons comment les composants d'une page ASPX obtiennent leurs valeurs à l'issue d'un POST :

1. la page ASPX est instanciée. Les composants sont initialisés avec leurs valeurs de conception.
2. la valeur __VIEWSTATE postée par le navigateur est utilisée pour donner aux composants la valeur qu'ils avaient lorsque la page ASPX a été envoyée au navigateur la fois précédente.
3. les valeurs postées par le navigateur sont affectées aux composants
4. les gestionnaires d'événements sont exécutés. Ils peuvent modifier la valeur de certains composants.

De cette séquence, on déduit que les composants qui ont leur valeur :

- postée
- modifiée par un gestionnaire d'événement

peuvent avoir leur propriété *EnableViewState* à *Faux* puisque leur valeur de VIEWSTATE (étape 2) va être modifiée par l'une des étapes 3 ou 4.

La liste des composants de notre page est disponible dans [Default.aspx.designer.cs] :

```
1.  namespace Intro {
2.      public partial class _Default {
3.          protected global::System.Web.UI.HtmlControls.HtmlForm form1;
4.          protected global::System.Web.UI.WebControls.TextBox TextBoxNom;
5.          protected global::System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidatorNom;
6.          protected global::System.Web.UI.WebControls.TextBox TextBoxAge;
7.          protected global::System.Web.UI.WebControls.RequiredFieldValidator RequiredFieldValidatorAge;
8.          protected global::System.Web.UI.WebControls.RangeValidator RangeValidatorAge;
9.          protected global::System.Web.UI.WebControls.DropDownList DropDownListNoms;
10.         protected global::System.Web.UI.WebControls.Button ButtonValider;
11.         protected global::System.Web.UI.WebControls.ListBox ListBoxEvts;
12.         protected global::System.Web.UI.WebControls.Label LabelPost;
13.         protected global::System.Web.UI.WebControls.Label LabelValidation;
14.         protected global::System.Web.UI.WebControls.Label LabelErreursSaisie;
15.         protected global::System.Web.UI.WebControls.Label LabelGlobal;
16.         protected global::System.Web.UI.WebControls.Label LabelNbRequetes;
17.     }
18. }
```

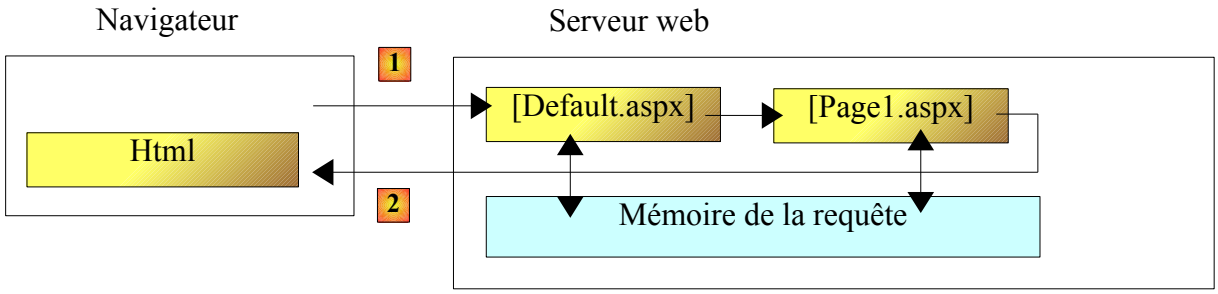
La valeur de la propriété *EnableViewState* de ces composants pourrait être la suivante :

Composant	*Valeur postée*	*EnableViewState*	*Pourquoi*
TextBoxNom	valeur saisie dans le TextBox	False	la valeur du composant est postée
TextBoxAge	idem		
RequiredFieldValidatorNom	aucune	False	absence de notion de valeur du composant
RequiredFieldValidatorAge	idem		
RangeValidatorAge	idem		

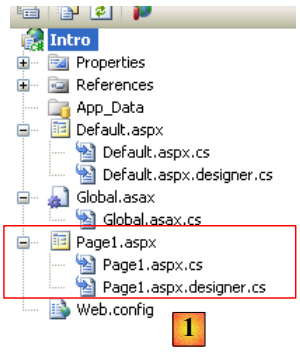
Composant	*Valeur postée*	*EnableViewState*	*Pourquoi*
LabelPost	aucune	False	obtient sa valeur par un gestionnaire d'événement
LabelValidation	idem		
LabelErreursSaisie	idem		
LabelGlobal	idem		
LabelNbRequetes	idem		
DropDownListNoms	"value" de l'élément sélectionné	True	on veut garder le contenu de la liste au fil des requêtes sans avoir à la régénérer
ListBoxEvts	"value" de l'élément sélectionné	False	le contenu de la liste est généré par un gestionnaire d'événement
ButtonValider	libellé du bouton	False	le composant garde sa valeur de conception

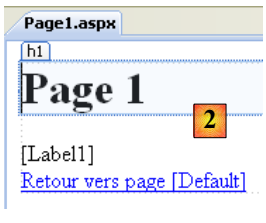
2.10 Forward d'une page vers une autre

Jusqu'à maintenant, les opérations GET et POST retournaient toujours la même page [Default.aspx]. Nous allons considérer le cas où une requête est traitée par deux pages ASPX successives, [Default.aspx] et [Page1.aspx] et où c'est cette dernière qui est retournée au client. Par ailleurs, nous verrons comment la page [Default.aspx] peut transmettre des informations à la page [Page1.aspx] via une mémoire qu'on appellera mémoire de la requête.



Nous construisons la page [Page1.aspx]. Pour cela, on suivra la méthode utilisée pour construire la page [Default.aspx] page 3.





- en [1], la page ajoutée
- en [2], la page une fois construite

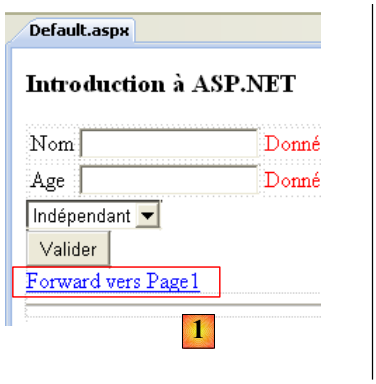
Le code source de [Page1.aspx] est le suivant :

```
1. <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Page1.aspx.cs" Inherits="Intro.Page1" %>
2.
```

```
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head id="Head1" runat="server">
6.    <title>Page1</title>
7.  </head>
8.  <body>
9.    <form id="form1" runat="server">
10.   <div>
11.     <h1>
12.       Page 1</h1>
13.     <asp:Label ID="Label1" runat="server"></asp:Label>
14.     <br />
15.     <asp:HyperLink ID="HyperLink1" runat="server" NavigateUrl="~/Default.aspx">Retour
16.       vers page [Default]</asp:HyperLink>
17.   </div>
18.   </form>
19. </body>
20. </html>
```

- ligne 13 : un label qui servira à afficher une information transmise par la page [Default.aspx]
- ligne 15 : un lien Html vers la page [Default.aspx]. Lorsque l'utilisateur clique sur ce lien, le navigateur demande la page [Default.aspx] avec une opération GET. La page [Default.aspx] est alors chargée comme si l'utilisateur avait tapé directement son Url dans son navigateur.

La page [Default.aspx] s'enrichit d'un nouveau composant de type *LinkButton* :



Le code source de ce nouveau composant est le suivant :

```
<asp:LinkButton ID="LinkButtonToPage1" runat="server" CausesValidation="False"
  EnableViewState="False" onclick="LinkButtonToPage1_Click">Forward vers Page1</asp:LinkButton>
```

- CausesValidation="False" : le clic sur le lien va provoquer un POST vers [Default.aspx]. Le composant [LinkButton] se comporte comme le composant [Button]. Ici, on ne veut pas que le clic sur le lien déclenche l'exécution des validateurs.
- EnableViewState="False" : il n'y a pas lieu de conserver l'état du lien au fil des requêtes. Il garde ses valeurs de conception.
- onclick="LinkButtonToPage1_Click" : nom de la méthode qui, dans [Defaul.aspx.cs], gère l'événement *Click* sur le composant *LinkButtonToPage1*.

Le code du gestionnaire *LinkButtonToPage1_Click* est le suivant :

```
1.    // vers Page1
2.    protected void LinkButtonToPage1_Click(object sender, EventArgs e)
3.    {
4.      // on met des infos dans le contexte
5.      Context.Items["msg1"] = "Message de Default.aspx pour Page1";
6.      // on passe la requête à Page1
7.      Server.Transfer("Page1.aspx",true);
8.  }
```

Ligne 7, la requête est passée à la page [Page1.aspx] au moyen de la méthode [Server.Transfer]. Le deuxième paramètre de la méthode qui est à *true* indique qu'il faut passer à [Page1.aspx] toute l'information qui a été envoyée à [Default.aspx] lors du POST. Cela permet par exemple à [Page1.aspx] d'avoir accès aux valeurs postées via une collection appelée *Request.Form*. La ligne 5 utilise

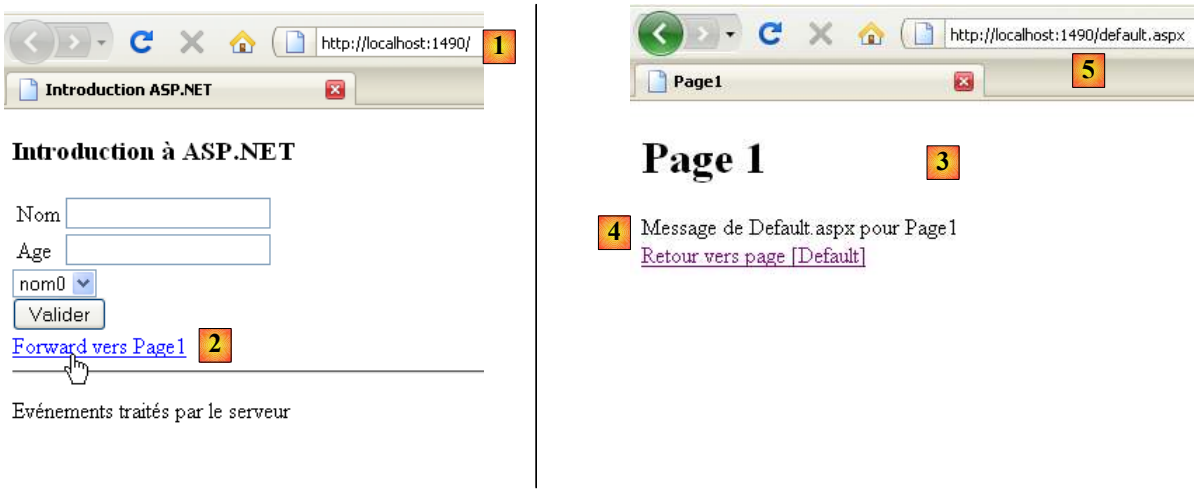
ce qu'on appelle le contexte de la requête. On y a accès via la propriété *Context* de la classe *Page*. Ce contexte peut servir de mémoire entre les différentes pages qui traitent la même requête, ici [Default.aspx] et [Page1.aspx]. On utilise pour cela le dictionnaire **Items**.

Lorsque [Page1.aspx] est chargée par l'opération *Server.Transfer("Page1.aspx",true)*, tout se passe comme si [Page1.aspx] avait été appelée par un GET d'un navigateur. Le gestionnaire *Page_Load* de [Page1.aspx] est exécuté normalement. Nous l'utiliserons pour afficher le message mis par [Default.aspx] dans le contexte de la requête :

```
1.  using System;
2.
3.  namespace Intro
4.  {
5.    public partial class Page1 : System.Web.UI.Page
6.    {
7.      protected void Page_Load(object sender, EventArgs e)
8.      {
9.        Label1.Text = Context.Items["msg1"] as string;
10.     }
11.   }
12. }
```

Ligne 9, le message mis par [Default.aspx] dans le contexte de la requête, est affiché dans *Label1*.

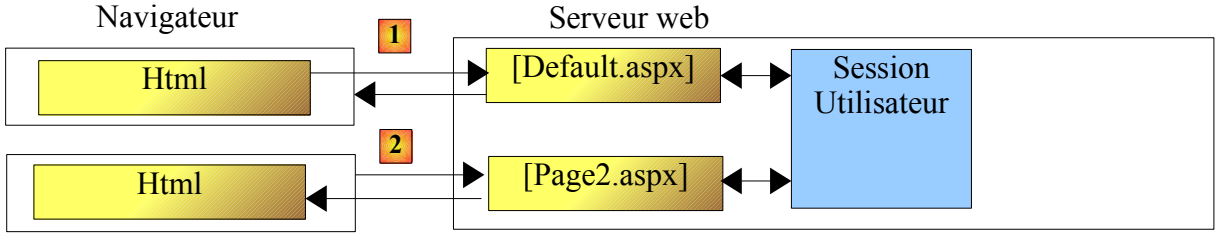
Voici un exemple d'exécution :



- dans la page [Default.aspx] [1], on clique sur le lien [2] qui nous mène vers la page *Page1*
- en [3], la page *Page1* est affichée
- en [4], le message créé dans [Default.aspx] et affiché par [Page1.aspx]
- en [5], l'Url affichée dans le navigateur est celle de la page [Default.aspx]

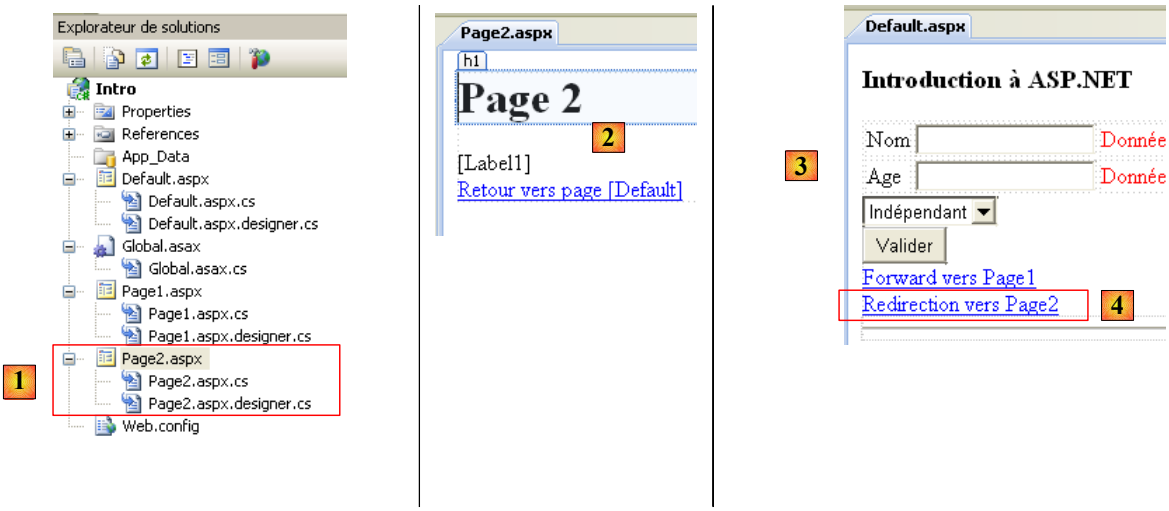
2.11 Redirection d'une page vers une autre

Nous présentons ici une autre technique fonctionnellement proche de la précédente : lorsque l'utilisateur demande la page [Default.aspx] par un POST, il reçoit en réponse une autre page [Page2.aspx]. Dans la méthode précédente, la requête de l'utilisateur était traitée successivement par deux pages : [Default.aspx] et [Page1.aspx]. Dans la méthode de la redirection de page que nous présentons maintenant, il y a deux requêtes distinctes du navigateur :



- en [1], le navigateur fait une requête POST à la page [Default.aspx]. Celle-ci traite la requête et envoie une réponse dite de redirection au navigateur. Cette réponse est un simple flux HTTP (des lignes de texte) demandant au navigateur de se rediriger vers une autre Url [Page2.aspx]. [Default.aspx] n'envoie pas de flux Html dans cette première réponse.
- en [2], le navigateur fait une requête GET à la page [Page2.aspx]. Celle-ci est alors envoyée en réponse au navigateur.
- si la page [Default.aspx] souhaite transmettre des informations à la page [Page2.aspx], elle peut le faire via la session de l'utilisateur. Contrairement à la méthode précédente, le contexte de la requête n'est pas utilisable ici, car il y a ici deux requêtes distinctes donc deux contextes distincts. Il faut alors utiliser la session de l'utilisateur pour faire communiquer les pages ensemble.

Comme il a été fait pour [Page1.aspx], nous ajoutons au projet la page [Page2.aspx] :



- en [1], [Page2.aspx] a été ajoutée au projet
- en [2], l'aspect visuel de [Page2.aspx]
- en [3], nous ajoutons à la page [Default.aspx] un composant *LinkButton* [4] qui va rediriger l'utilisateur vers [Page2.aspx].

Le code source de [Page2.aspx] est analogue à celui de [Page1.aspx] :

```
1.  <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Page2.aspx.cs" Inherits="Intro.Page2" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head id="Head1" runat="server">
6.    <title>Page2</title>
7.  </head>
8.  <body>
9.    <form id="form1" runat="server">
10.   <div>
11.     <h1>
12.       Page 2</h1>
13.     <asp:Label ID="Label1" runat="server"></asp:Label>
14.     <br />
15.     <asp:HyperLink ID="HyperLink1" runat="server" NavigateUrl="~/Default.aspx">Retour
16.       vers page [Default]</asp:HyperLink>
17.   </div>
18.   </form>
19. </body>
20. </html>
```

Dans [Default.aspx], l'ajout du composant *LinkButton* a généré le code source suivant :

```
<asp:LinkButton ID="LinkButtonToPage2" runat="server"
    onclick="LinkButtonToPage2_Click">Redirection vers Page2</asp:LinkButton>
```

C'est le gestionnaire [LinkButtonToPage2_Click] qui assure la redirection vers [Page2.aspx]. Son code dans [Default.aspx.cs] est le suivant :

```
1.      protected void LinkButtonToPage2_Click(object sender, EventArgs e)
2.      {
3.        // on met un msg dans la session
4.        Session["msg2"] = "Message de [Default.aspx] pour [Page2.aspx]";
5.        // on redirige le client vers [Page2.aspx]
6.        Response.Redirect("Page2.aspx");
7.  }
```

- ligne 4 : on met un message dans la session de l'utilisateur
- ligne 5 : l'objet *Response* est une propriété de toute page ASPX. Elle représente la réponse faite au client. Elle possède une méthode *Redirect* qui fait que la réponse faite au client va être un ordre HTTP de redirection.

Lorsque le navigateur va recevoir l'ordre de redirection vers [Page2.aspx], il va faire un GET sur cette page. Dans celle-ci, la méthode [Page_Load] va s'exécuter. On va l'utiliser pour récupérer le message mis par [Default.aspx] dans la session et afficher celui-ci. Le code [Page2.aspx.cs] est le suivant :

```
1.  using System;
2.
3.  namespace Intro
4.  {
5.    public partial class Page2 : System.Web.UI.Page
6.    {
7.      protected void Page_Load(object sender, EventArgs e)
8.      {
9.        // on affiche le msg mis dans la session par [Default.aspx]
10.       Label1.Text = Session["msg2"] as string;
11.     }
12.   }
13. }
```

A l'exécution, on obtient les résultats suivants :

- en [1], on clique sur le lien de redirection de [Default.aspx]. Un POST est fait vers la page [Default.aspx]
- en [2], le navigateur a été redirigé vers [Page2.aspx]. Cela se voit à l'Url affichée par le navigateur. Dans la méthode précédente, cette Url était celle de [Default.aspx] car l'unique requête faite par le navigateur l'était vers cette Url. Ici il y a un premier POST vers [Default.aspx], puis à l'insu de l'utilisateur un second GET vers [Page2.aspx].
- en [3], on voit que [Page2.aspx] a correctement récupéré le message mis par [Default.aspx] dans la session.

2.12 Conclusion

Nous avons introduit, à l'aide de quelques exemples, des concepts d'ASP.NET importants :

- la page ASPX avec ses deux modes de développement [Source] et [Design]
- la notion d'événement dans une page
- l'écriture des gestionnaires de ces événements
- le champ caché " _VIEWSTATE " qui est à la base du modèle événementiel de ASP.NET
- l'utilisation de validateurs pour contrôler la validité des saisies

- la différence entre les requêtes GET et POST d'une page
- la gestion des données de portée **Application** qui sont partagées par toutes les requêtes de tous les utilisateurs. Nous avons alors introduit le fichier [Web.config] qui permet de définir certaines de ces données.
- la gestion des données de portée **Session** qui sont partagées par toutes les requêtes d'un même utilisateur. Elles représentent la mémoire de l'utilisateur.
- la gestion des données de portée **Requête** qui sont partagées par toutes les pages traitant une même requête
- le traitement d'une requête par deux pages
- la redirection vers une autre page

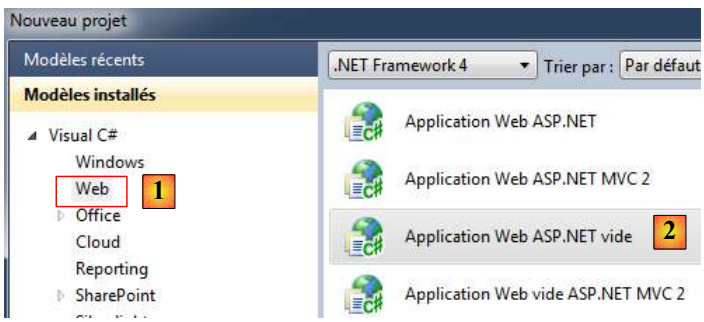
Le projet qui suit introduit la notion de page maître.

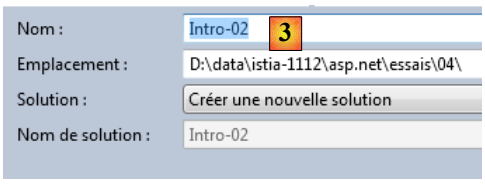
3 Création d'une page maître

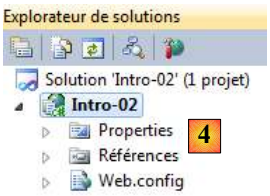
Nous souhaitons construire une application web dont toutes pages auraient la forme suivante :

Les zones 1-3 restent toujours les mêmes. La zone 4 affiche différentes pages ASPX.

Nous construisons le projet suivant :

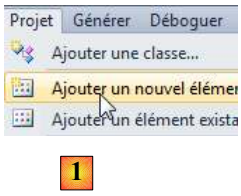


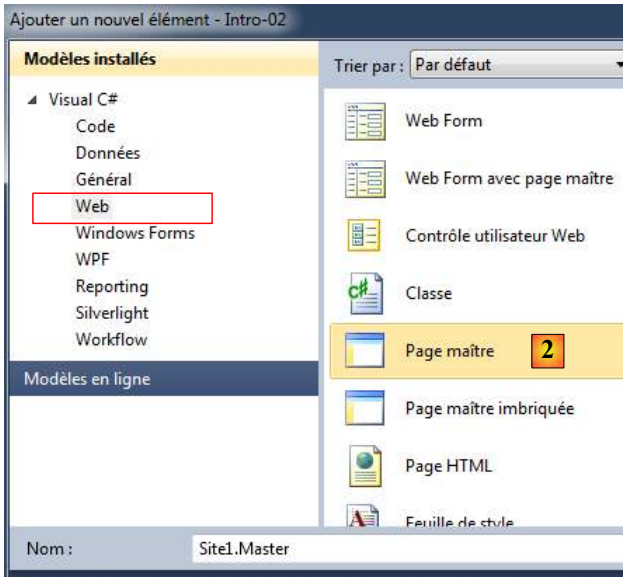


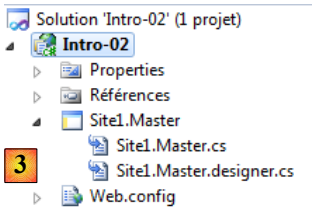


- en [1,2], on crée une application Web vide
- à laquelle on donne un nom et un emplacement [3].
- en [4], le nouveau projet

Nous créons maintenant la page maître de l'application :





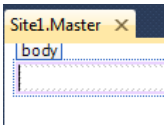


- en [1], on ajoute un nouvel élément au projet
- en [2], on choisit une page maître
- en [3], le nouveau projet

Présentons les fichiers ainsi créés :

- [Site1.Master] est la page maître qu'on peut construire comme les pages ASPX normales.
- [Site1.Master.cs] contient le code C# de traitement des événements de la page maître
- [Site1.Master.Deigner.cs] contient la déclaration des composants ASP utilisés par la page maître.

La page maître générée est la suivante (mode Design) :



Le code source (mode Source) :

```
1.  <%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site1.master.cs"
    Inherits="Intro_02.Site1" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.
5.  <html xmlns="http://www.w3.org/1999/xhtml">
6.  <head runat="server">
7.      <title></title>
8.      <asp:ContentPlaceHolder ID="head" runat="server">
9.      </asp:ContentPlaceHolder>
10. </head>
11. <body>
12.     <form id="form1" runat="server">
13.     <div>
14.         <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
15.
16.         </asp:ContentPlaceHolder>
17.     </div>
```

```
18.     </form>
19. </body>
20. </html>
```

Nous ne commentons que les nouveautés par rapport à une page ASPX classique :

- ligne 1 : la directive Master signale qu'on a affaire à une page maître
- lignes 8-9 : la balise *<asp:ContentPlaceHolder>* sert à délimiter un conteneur. Ce sont les pages ASPX utilisant cette page maître qui indique quoi mettre dans ces conteneurs. Ici, la page ASPX pourra y mettre des balises Html compatibles avec la balise englobante *<head>*.
- lignes 14-16 : un autre conteneur

Nous construisons la page maître suivante (mode Source) :

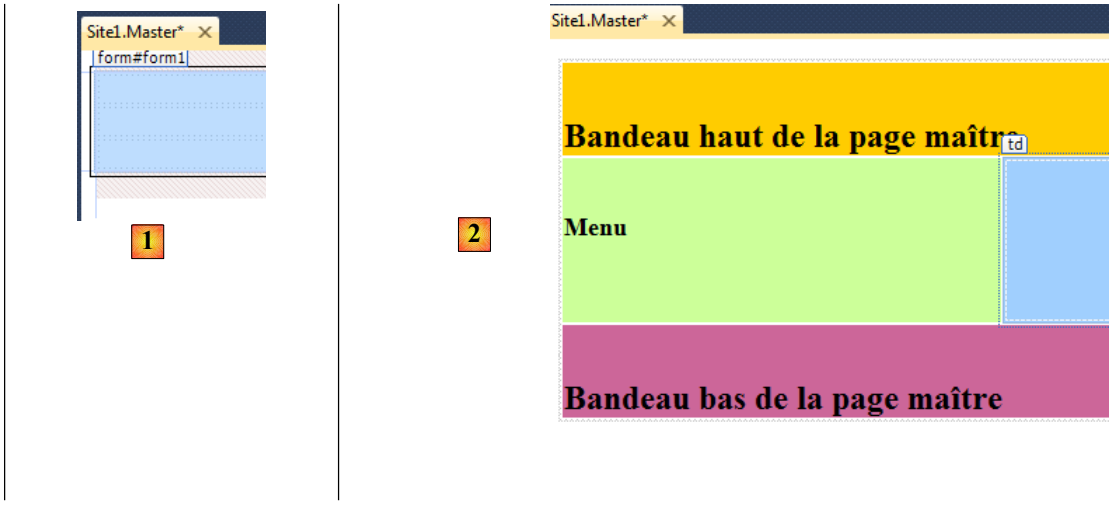
```
1.  <%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site1.master.cs"
    Inherits="Intro_02.Site1" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head runat="server">
6.      <title>Page Maître</title>
7.  </head>
8.  <body>
9.      <form id="form1" runat="server">
10.
11.     </form>
12. </body>
13. </html>
```

Nous avons éliminé les deux conteneurs. Puis avec le curseur sur la ligne 10, nous insérons une table Html (boîte à outils / Html). Le code évolue de la façon suivante :

```
1.  <%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site1.master.cs"
    Inherits="Intro_02.Site1" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head runat="server">
6.      <title>Page Maître</title>
7.  </head>
8.  <body>
9.      <form id="form1" runat="server">
10.     <table style="width: 100%;">
11.         <tr>
12.             <td>
13.                  
14.             </td>
15.             <td>
16.                  
17.             </td>
18.             <td>
19.                  
20.             </td>
21.         </tr>
22.         <tr>
23.             <td>
24.                  
25.             </td>
26.             <td>
27.                  
28.             </td>
29.             <td>
30.                  
31.             </td>
32.         </tr>
33.         <tr>
34.             <td>
35.                  
36.             </td>
37.             <td>
38.                  
39.             </td>
```

```
40.             <td>
41.                  
42.             </td>
43.         </tr>
44.     </table>
45.     </form>
46. </body>
47. </html>
```

Puis nous passons en mode [Design] :

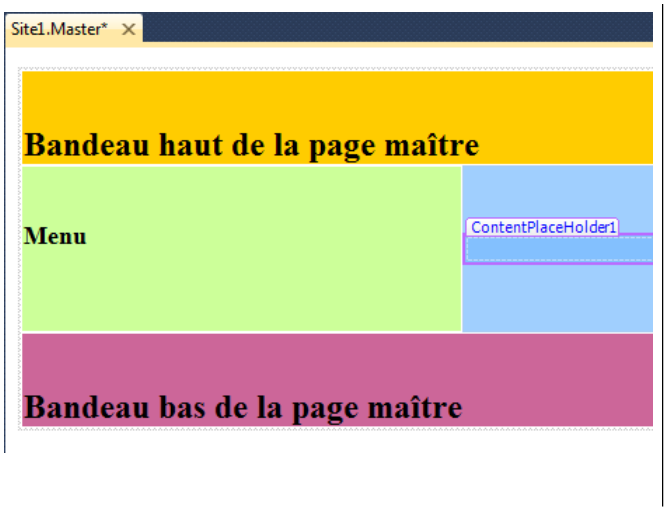


Nous avons un tableau à 3 lignes et 3 colonnes [1]. Nous construisons à partir de là, la page [2] construite avec un tableau de 3 lignes et 2 colonnes. Pour cela, nous :

- supprimons une colonne : clic droit sur la colonne / Supprimer / … les colonnes
- fusionnons les cellules de la 1ère puis de la 3ième ligne : sélectionner les cellules à fusionner / clic droit / Modifier / Fusionner les cellules
- donnons à la cellule (2,1) Menu, la largeur de 500 px.
- donnons à chaque cellule, une couleur de fond pour les différencier : propriété *bgcolor* de la cellule.

Les pages ASPX utilisant cette page maître s'afficheront dans la cellule (2,2). Pour cela, il nous faut mettre dans cette cellule un conteneur : sélection de la cellule / boîte à outils / Standard / ContentPlaceHolder. La page maître évolue ainsi :

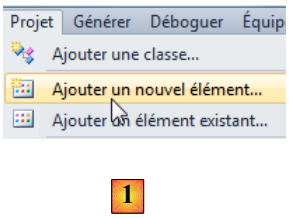
mode [Design] :

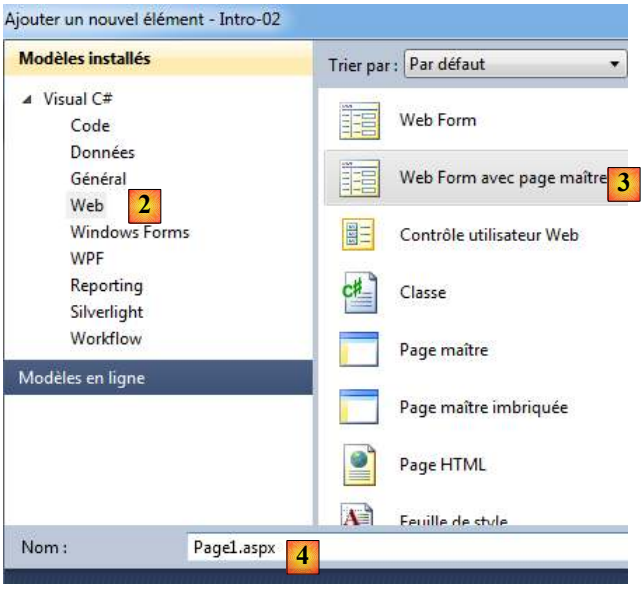


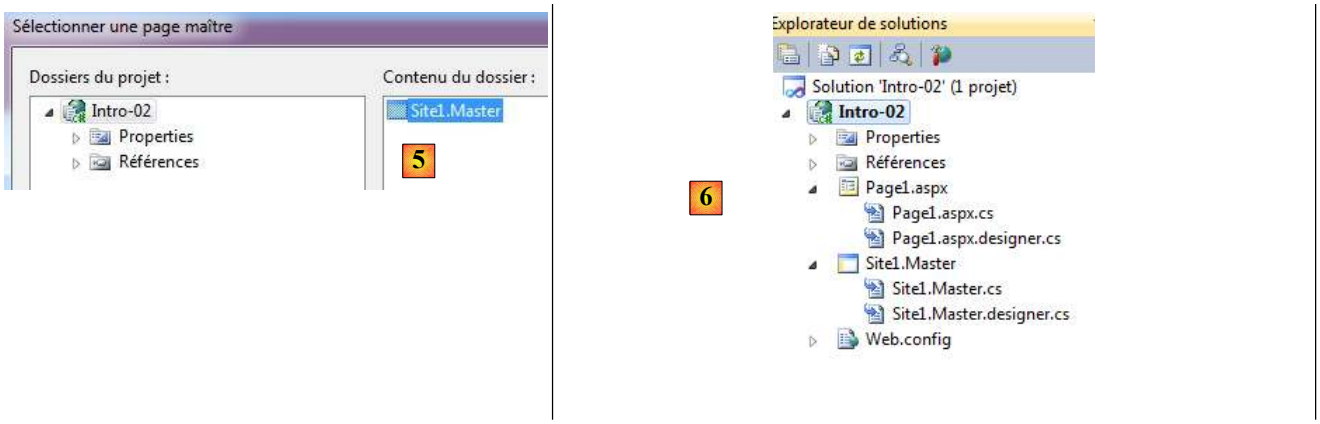
mode [Source] :

```
1.  <%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site1.master.cs"
    Inherits="Intro_02.Site1" %>
2.
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head runat="server">
6.      <title>Page Maître</title>
7.  </head>
8.  <body>
9.      <form id="form1" runat="server">
10.     <table style="width: 100%;">
11.         <tr>
12.             <td colspan="2" bgcolor="#FFCC00">
13.                 <h2>
14.                     Bandeau haut de la page maître</h2>
15.             </td>
16.         </tr>
17.         <tr>
18.             <td class="style1" width="300px" bgcolor="#CCFF99">
19.                 <h3>
20.                     Menu</h3>
21.                 <br />
22.             </td>
23.             <td bgcolor="#99CCFF">
24.                 <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
25.                 </asp:ContentPlaceHolder>
26.             </td>
27.         </tr>
28.         <tr>
29.             <td colspan="2" bgcolor="#CC6699">
30.                 <h2>
31.                     Bandeau bas de la page maître</h2>
32.             </td>
33.         </tr>
34.     </table>
35.     </form>
36. </body>
37. </html>
```

Nous avons désormais la page maître. Nous allons construire maintenant une première page ASPX utilisant cette page maître.







- en [5], on sélectionne la page Maître
- en [6], le nouveau projet

Le code source généré pour la page [Page1.aspx] est le suivant :

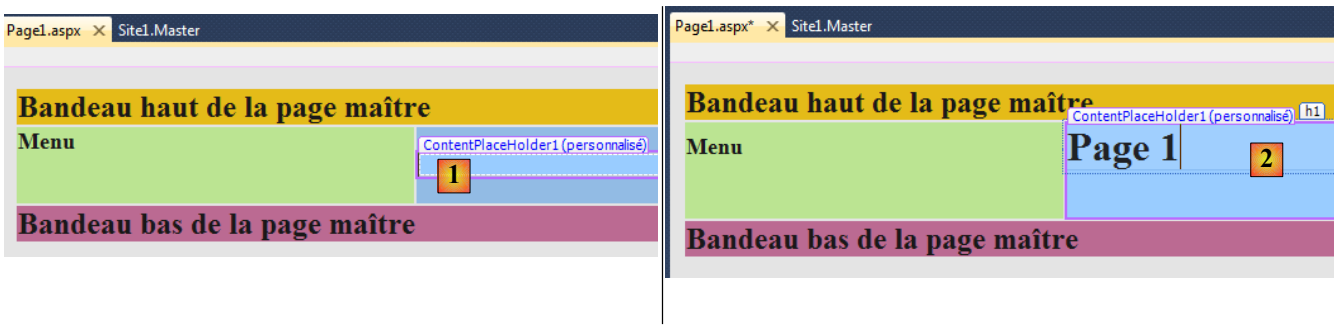
```
1.  <%@ Page Title="" Language="C#" MasterPageFile="~/Site1.Master" AutoEventWireup="true"
    CodeBehind="Page1.aspx.cs" Inherits="Intro_02.Page1" %>
2.  <asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
3.  </asp:Content>
```

- ligne 1 : l'attribut *MasterPageFile* indique que la page [Default.aspx] s'insère dans la page maître [Site1.Master]. Le signe ~ désigne le dossier de la page [Default.aspx].
- ligne 2 : la balise *<asp:Content>* sert à insérer du code ASP/Html dans un des conteneurs de la page maître. Celui-ci est désigné par l'attribut *ContentPlaceHolderID*. La valeur de cet attribut doit être l'ID d'un des conteneurs de la page maître. Si nous revenons au code de celle-ci :

```
1.              <td bgcolor="#99CCFF">
2.                  <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
3.                  </asp:ContentPlaceHolder>
4.  </td>
```

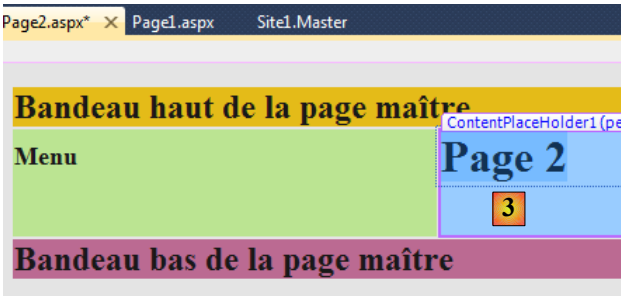
nous voyons que l'ID du conteneur est *ContentPlaceHolder1*.

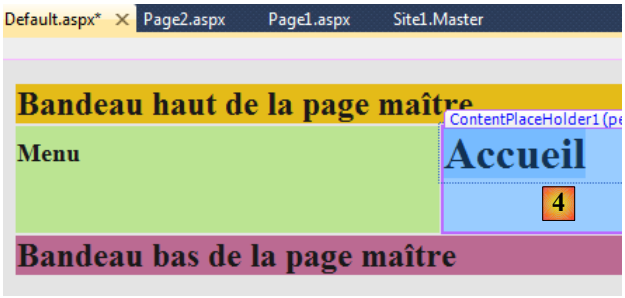
En mode [Design], la page [Page1.aspx] est la suivante :



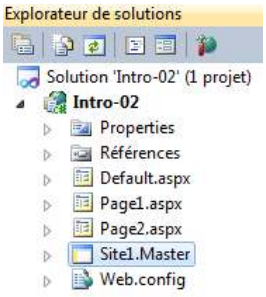
Nous ne pouvons déposer des composants que dans le conteneur [1]. Nous faisons évoluer la page pour obtenir [2].

Nous répétons l'opération respectivement pour les pages [Page2.aspx] [3] et [Default.aspx] [4] :

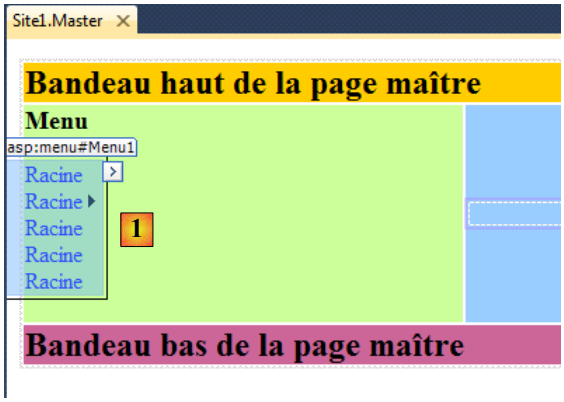


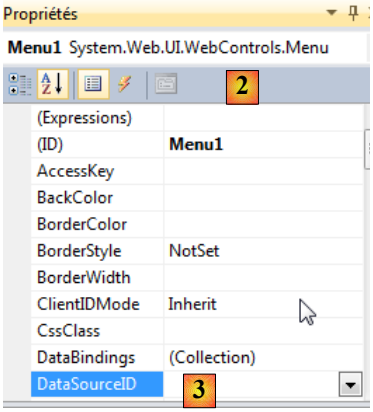


A ce stade, notre projet est le suivant :



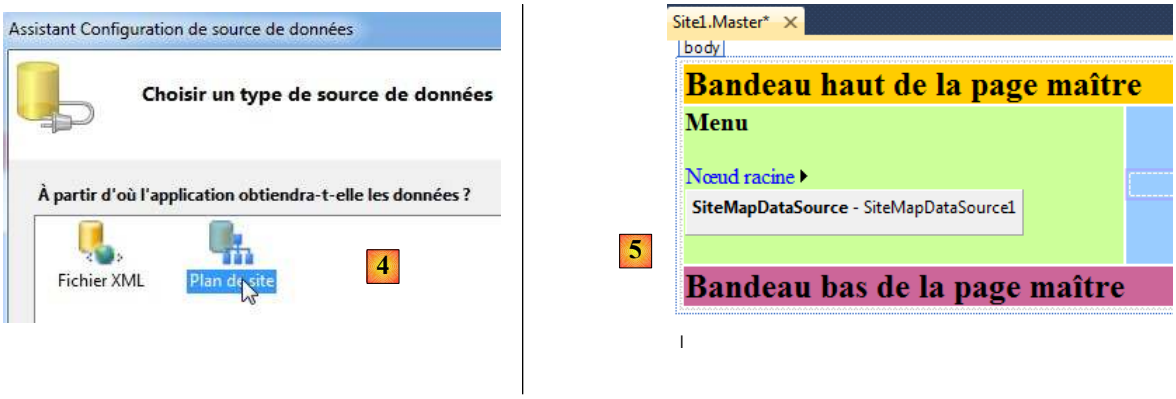
Si nous exécutons le projet maintenant (Ctrl-F5), nous obtenons la page [Default.aspx] [4]. Afin de pouvoir naviguer entre les pages, nous allons ajouter un composant [Menu] dans la cellule (2,1) de la page maître.



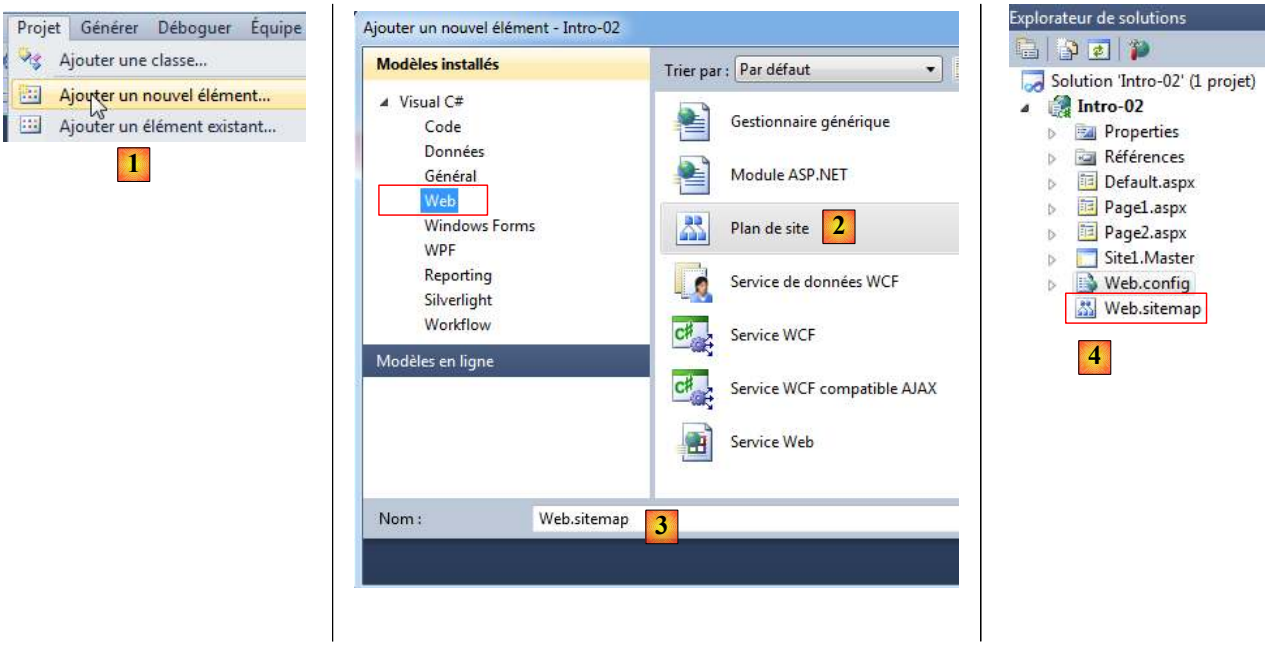


- en [1], le composant [Menu] est trouvé dans boîte à Outils / Navigation / Menu. Ce menu est alimenté par une source de données statique ou dynamique. Nous allons créer une souce statique pour naviguer entre les trois pages : Default.aspx, Page1.aspx, Page2.aspx.

Dans les propriétés [2] du menu, nous choisissons la propriété [DataSourceId] [3] puis l'option <*Nouvelle source de données*> :



- en [4], nous choisissons un plan de site comme source de données
- en [5], la page avec ce nouveau composant de type [SiteMapDataSource]. Ce composant est un intermédiaire entre le menu et un fichier Xml qui liste les options de menu. Nous créons maintenant le fichier Xml qui va décrire la structure du menu.



- en [1], on ajoute un nouvel élément au projet
- en [2], un élément de type [Plan de site] est choisi et son nom donné en [3]
- en [4], le nouveau visage du projet

Le fichier [Web.siteMap] est un fichier Xml :

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
3.     <siteMapNode url="" title=""  description="">
4.         <siteMapNode url="" title=""  description="" />
5.         <siteMapNode url="" title=""  description="" />
6.     </siteMapNode>
7. </siteMap>
```

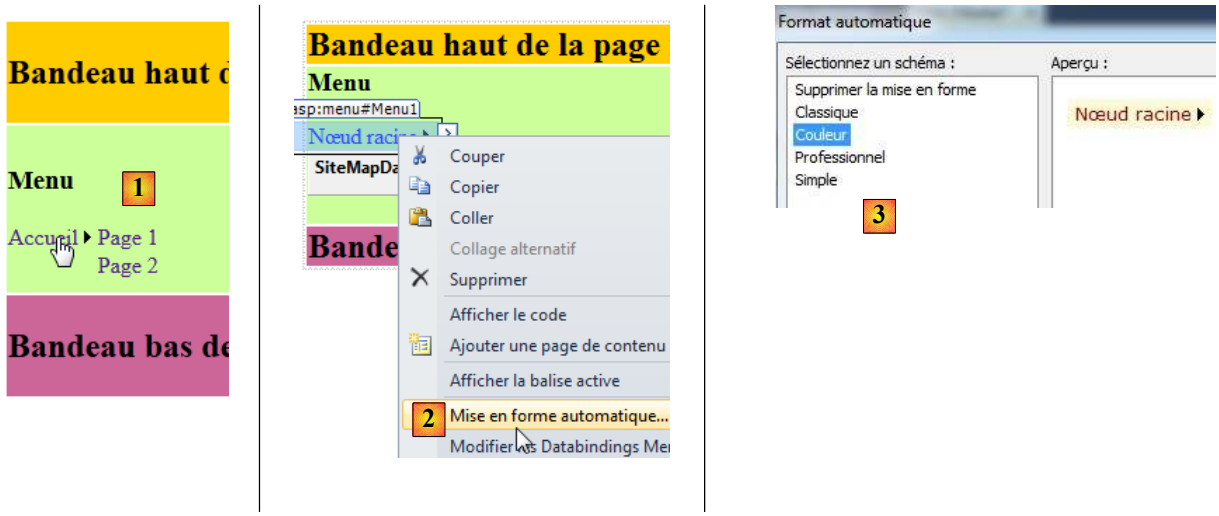
La balise <*siteMapNode*> va servir à définir une option du menu. L'imbrication des balises sera reflétée par une imbrication des options de menu. La balise <*siteMapNode*> a deux attributs obligatoires :

- **titre** : libellé de l'option de menu
- **url** : url de la page vers laquelle on doit naviguer lorsque l'option est cliquée

Nous faisons évoluer le fichier précédent comme suit :

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
3.      <siteMapNode url="Default.aspx" title="Accueil"  description="">
4.          <siteMapNode url="Page1.aspx" title="Page 1"  description="" />
5.          <siteMapNode url="Page2.aspx" title="Page 2"  description="" />
6.      </siteMapNode>
7.  </siteMap>
```

Ce fichier sera traduit dans le menu suivant [1] :



Le composant [Menu] peut être mis en forme [2]. Plusieurs options de mise en forme sont proposées [3].

Le projet peut être exécuté (Ctrl-F5). Cette fois-ci le menu permet de naviguer entre les pages Default.aspx, Page1.aspx et Page2.aspx. Ces pages partagent toutes l'architecture de la page maître.

- en [1], la page d'accueil
- en [2], on veut naviguer vers la page 2
- en [3], résultat de la navigation
- en [4], la Page 2 s'est insérée dans la page maître

Chaque page insérée dans la page maître peut être aussi complexe que la page étudiée au début de ce document. Pour le montrer, nous ajoutons des liens de navigation dans les pages [Page1.aspx] et [Page2.aspx].

[Page1.aspx] – Mode [Design]

[Page1.aspx] – Mode [Source]

```
1.  <%@ Page Title="" Language="C#" MasterPageFile="~/Site1.Master" AutoEventWireup="true"
    CodeBehind="Page1.aspx.cs" Inherits="Intro_02.Page1" %>
2.  <asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
3.      <h1>
4.      Page 1</h1>
5.  <p>
6.      <asp:Label ID="LabelMessage" runat="server" EnableViewState="False"></asp:Label>
7.      </p>
8.      <p>
9.          <asp:LinkButton ID="LinkButton1" runat="server"
    onclick="LinkButton1_Click">Server.Transfer vers Page 2</asp:LinkButton>
10.     </p>
11. </asp:Content>
```

[Page1.aspx.cs] – Code de contrôle de la page

```
1.  using System;
2.
3.  namespace Intro_02 {
4.    public partial class Page1 : System.Web.UI.Page {
5.      protected void Page_Load(object sender, EventArgs e) {
6.        // on récupère un éventuel message dans la session
7.        string message = Session["message"] as string;
8.        if (message == null) {
9.          message = "";
10.       }
11.       // on l'affiche
12.       LabelMessage.Text = message;
13.     }
14.
15.     protected void LinkButton1_Click(object sender, EventArgs e) {
16.       // on crée un message pour Page 2
17.       Context.Items["message"] = "Message de Page 1 pour Page 2";
18.       // tranfert vers Page 2
19.       Server.Transfer("Page2.aspx", true);
20.     }
21.   }
22. }
```

[Page2.aspx] – Mode [Design]

[Page2.aspx] – Mode [Source]

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site1.Master" AutoEventWireup="true"
CodeBehind="Page2.aspx.cs" Inherits="Intro_02.Page2" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
    <h1>
    Page 2</h1>
<p>
    <asp:Label ID="LabelMessage" runat="server" EnableViewState="False"></asp:Label>
    </p>
    <p>
        <asp:LinkButton ID="LinkButton1" runat="server" onclick="LinkButton1_Click">Redirection
vers Page 1</asp:LinkButton>
    </p>
</asp:Content>
```

[Page2.aspx.cs] – Code de contrôle de la page

```
using System;

namespace Intro_02 {
  public partial class Page2 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
      // on récupère un éventuel message dans le contexte de la requête
      string message = Context.Items["message"] as string;
      if (message == null) {
        message = "";
      }
      // on l'affiche
      LabelMessage.Text = message;
    }

    protected void LinkButton1_Click(object sender, EventArgs e) {
      // on crée un message pour Page 1
      Session["message"] = "Message de Page 2 pour Page 1";
      // on redirige le client vers Page1.aspx
      Response.Redirect("Page1.aspx");
    }
  }
}
```

Le lecteur est invité à tester cette nouvelle version.

4 Conclusion

Nous avons montré comment

- construire des pages Web
- gérer leurs événements
- les insérer dans une page maître

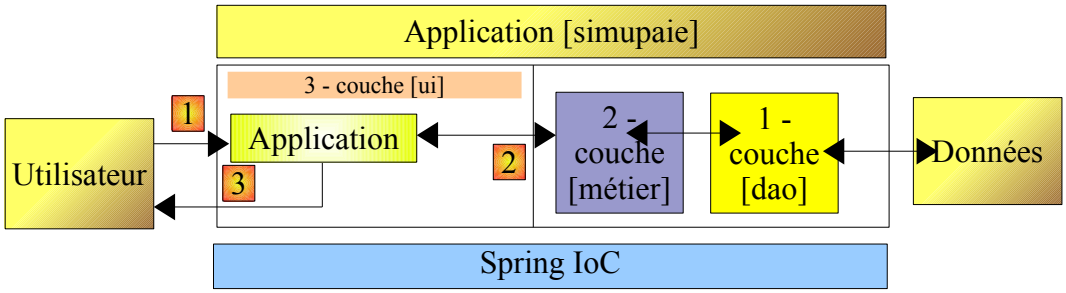
Cette introduction ne permet pas de comprendre les subtilités des échanges client / serveur d'une application web. Pour cela, on pourra lire un document ancien :

- **Programmation ASP.NET** [http://tahe.developpez.com/dotnet/aspnet/vol1] et [http://tahe.developpez.com/dotnet/aspnet/vol2]

Ce document peut être ultérieurement approfondi par une étude de cas :

- **Construction d'une application à trois couches avec ASP.NET, C#, Spring.net et Nhibernate** [http://tahe.developpez.com/dotnet/pam-aspnet/]

L'application de l'étude de cas, a la structure à trois couches suivante :



- la couche [1-dao] (dao=Data Access Object) s'occupe de l'accès aux données. Celles-ci sont placées dans une base de données.
- la couche [2-métier] s'occupe de l'aspect métier de l'application, le calcul de la paie.
- la couche [3-ui] (ui=User Interface) s'occupe de la présentation des données à l'utilisateur et de l'exécution de ses requêtes. Nous appelons [Application] l'ensemble des modules assurant cette fonction.
- les trois couches sont rendues indépendantes grâce à l'utilisation d'interfaces .NET
- l'intégration des différentes couches est réalisée par **Spring IoC**

Les pages ASP.NET nécessaires au dialogue avec l'utilisateur font partie de la couche [ui]. Ce qui a été vu dans ce document est suffisant pour construire ces pages web. Le reste de l'application est constitué des couches [métier] et [dao]. La compétence ASP.NET n'intervient pas dans ces couches. Il suffit d'avoir une compétence classique en VB.NET ou C#.

Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

1. le client fait une demande à l'application.
2. l'application traite cette demande. Pour ce faire, elle peut avoir besoin de l'aide de la couche [métier] qui elle-même peut avoir besoin de la couche [dao] si des données doivent être échangées avec la base de données. L'application reçoit une réponse de la couche [métier].
3. selon celle-ci, elle envoie la vue (= la réponse) appropriée au client.

L'interface présentée à l'utilisateur peut avoir diverses formes :

1. une application console : dans ce cas, la vue est une suite de lignes de texte.
2. une application graphique windows : dans ce cas, la vue est une fenêtre windows
3. une application web : dans ce cas, la vue est une page HTML
4. ...

Différentes versions de cette application sont proposées :

1. une version ASPNET comportant un unique formulaire et construite avec une architecture à une couche.
2. une version identique à la précédente mais avec des extensions Ajax
3. une version ASP.NET s'appuyant sur une architecture à trois couches où la couche d'accès aux données est implémentée avec le framework NHibernate. Elle a toujours l'unique formulaire de la version 1.
4. une version 4 ASP.NET multi-vues et mono-page avec l'architecture trois couches de la version 3.
5. la partie serveur d'une application client / serveur où le serveur est implémenté par un service web s'appuyant sur l'architecture en couches de la version 3.
6. la partie client de l'application client / serveur précédente, implémentée par une couche ASP.NET.
7. une version 7 ASP.NET multi-vues et multi-pages avec l'architecture trois couches de la version 3.
8. une version 8 ASP.NET multi-vues et multi-pages cliente du service web de la version 5.
9. une version 9 ASP.NET multi-vues et multi-pages avec l'architecture trois couches de la version 3 où la couche d'accès aux données est implémentée par des classes de Spring qui facilitent l'utilisation du framework NHibernate.
10. une version 10 implémentée en FLEX et cliente du service web de la version 5.

Table des matières