



Concepteur Développeur en Informatique

Développer des composants d'interface

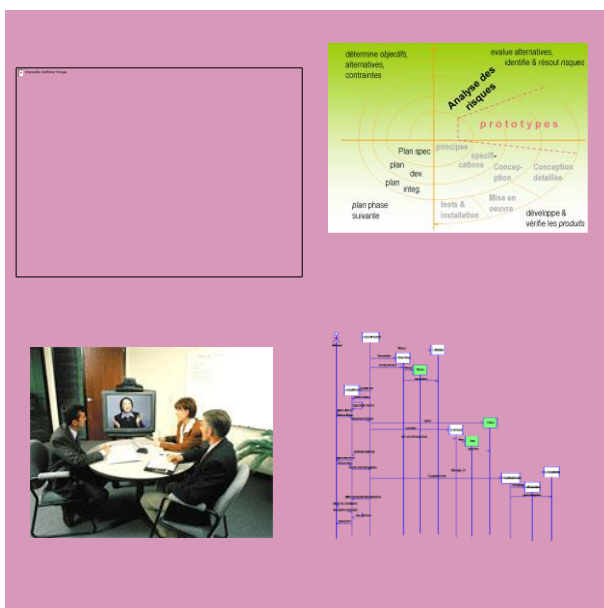
Présentation ASP Net-Partie 3 – Web Forms

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E05-S03

SOMMAIRE

1. Introduction.....	3
2. La structure d'une page aspx	3
3. Les contrôles	5
3.1. Les contrôles HTML classiques.....	5
3.2. Les contrôles HTML serveur.	7
3.3. Les contrôles serveur	10
3.3.1. Prise en compte du navigateur.....	11
3.3.2. Les événements.....	11
4. Gestion du ViewState.....	13
4.1.1. Fonctionnement	13
4.1.2. ViewState et performance	15
5. Cycle de vie d'une page asp.net.....	16
5.1. Les principaux événements	16
6. Contrôles de validation	19
6.1. Le contrôle de valeur requise : RequiredFieldValidator	21
6.2. Les contrôles de valeur CompareValidator et RangeValidator	22
6.3. Le contrôle personnalisé CustomValidator	23
6.3.1. La fonction côté client.....	23
6.3.2. La fonction côté serveur.	23
6.4. Le contrôle RegularExpressionValidator	24
6.5. Déterminer si la page est valide	25

1. Introduction

Ce document a pour objectif de vous présenter les bases du développement d'applications Web avec le modèle d'affichage Web Forms d'ASP net. Nous verrons comment mettre en œuvre le rendu de page en scindant le processus de dessin en contrôles côté serveur.

2. La structure d'une page aspx

Nous allons revenir un instant sur la structure d'une page aspx.

Dans ce premier exemple, une page minimaliste est créée en tant que formulaire Web.

Afficher en mode source, la page comporte 3 parties

❶

```

1 <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="R001.aspx.cs" Inherits="A001N.R001" %>
2
3 <!DOCTYPE html>
4
5 <html xmlns="http://www.w3.org/1999/xhtml">
6 <head runat="server">
7 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
8 <title></title>
9 </head>
10 <body>
11 <form id="form1" runat="server">
12 <div>
13 <p>Bonjour les CDI</p>
14 </div>
15 </form>
16 </body>
17 </html>

```

❷

❸

❶ La directive de page @Page

Les directives de traitement configurent l'environnement d'exécution qui exécutera la page. Dans ASP.Net, les directives peuvent se situer n'importe où dans la page, même s'il est courant et préférable de les placer en tête de fichier.

Les différentes natures de contrôles de base

@Page ne peut s'utiliser que dans une page .aspx. Il existe aussi une directive **@Control** utilisable dans les pages .ascx définissant des contrôles utilisateurs. Nous verrons cette technique ultérieurement.

La syntaxe d'une directive est unique : Plusieurs attributs doivent être séparés par des espaces et aucun espace ne doit figurer autour de =.

Chaque directive possède son propre ensemble d'attributs : la syntaxe générale s'écrit :

```
<% Nom_Directive attribut=« valeur ... %>
```

@Page comporte environ 30 attributs pouvant être rassemblés en 4 catégories :

Attributs pour la compilation

L'attribut Language définit le langage dans lequel le script de la page Web est écrit (C#, vb, JScript)

L'attribut de page CodeBehind indique le chemin d'accès vers la classe contenant la logique de prise en charge du formulaire Web. Le fichier source de la classe doit être déployé sur le serveur Web

Attribut d'héritage

L'attribut Inherits définit la classe de base de laquelle hérite la page : ce peut être n'importe laquelle, dérivée de la classe Page.

Attribut précisant le comportement de la page

L'attribut AutoEventWireup indique par sa valeur booléenne (true par défaut), que les événements de page sont automatiquement activés.

Cela évite de devoir mettre en place un mécanisme d'abonnement aux événements comme par exemple :

```
this.Load += R001_Load;
```

Attributs pour la sortie de la page

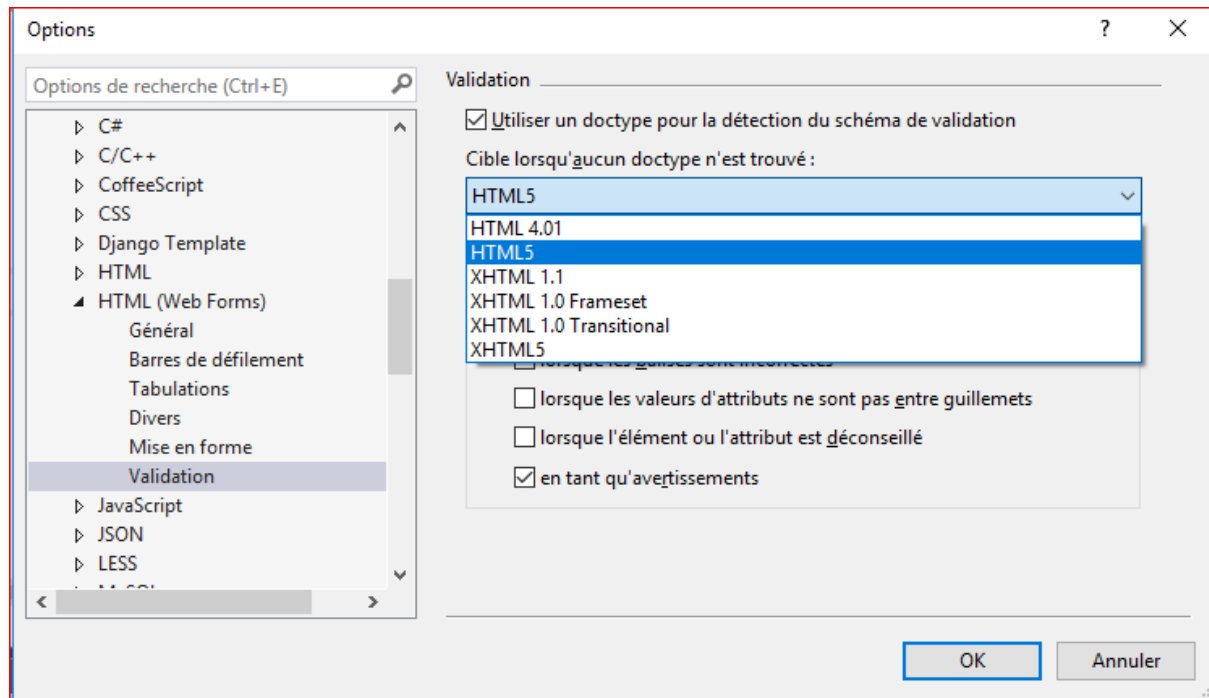
De nombreux attributs concernent la localisation de la page.

Les DTD

Les définitions de type de documents (Document Type Definition) sont établies par le consortium W3C. Il s'agit d'une norme applicable aux documents XML et HTML qui fixe les règles syntaxiques et sémantiques de constructions de documents à partir de balises.

```
<!DOCTYPE html>
```

Ici nous ciblons du HTML 5.



Vous pouvez vérifier dans les options de l'éditeur de texte, la version utilisée pour la validation des documents. Ici, du HTML 5.

Ces éléments sont modifiables pour pouvoir utiliser un autre schéma de validation.

3. Les contrôles

Pour construire le rendu visuel d'une page Web vous avez à votre disposition 3 types de contrôles disposant chacun de leurs avantages et leurs inconvénients.

1. Les contrôles HTML
2. Les contrôles HTML serveur
3. Les contrôles serveur

3.1. Les contrôles HTML classiques.

Prenons l'exemple d'une page asp permettant la récupération du nom d'un utilisateur.

Nous choisissons comme option de faire saisir ce nom dans une boîte de texte et de récupérer celui-ci pour le stocker dans une variable de session.

```
<form id="form1" action="Controle_1_HTML.aspx"
method="post" enctype="application/x-www-form-urlencoded">
<div>
    <label id="lblNom" for="txtNom">Entrez votre nom</label>
    <input id="txtNom" name="txtNom" type="text" /><br />
    <input id="Submit1" type="submit" value="Envoyer" /></div>
</form>
```

Le code associé à cette page pour récupérer les valeurs de cette dernière est :

```
public partial class Controle_1_HTML : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        this.Page.Session["Nom"] = this.Request.Form["txtNom"];
    }
}
```

Le code est excessivement simple mais d'une grande pauvreté fonctionnelle.

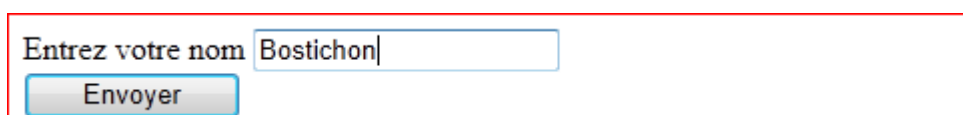
Avec un formulaire et des contrôles HTML côté client, je ne peux déterminer si nous traitons l'affichage initial de la page ou s'il s'agit du traitement des données postées. La propriété de l'objet page IsPostBack est sans effet.

Nous ne pouvons pas plus interagir avec les contrôles de formulaire depuis le serveur.

La mise en place d'une vérification de la présence des informations demandées au niveau du serveur n'est pas des plus aisées.

De même, une fois les données transmises, les valeurs ne sont pas conservées au niveau du formulaire. Si nous souhaitons donc réafficher la valeur saisie, il nous faut réécrire le contrôle HTML en lui précisant sa valeur.

Affichage initial



Envoi et traitement des données

Réaffichage après postage

Entrez votre nom

Par contre notre page Web est très légère. Regardons les informations en mode traçage (pour rappel directive de page `trace=true`)

UniqueID du contrôle	Type	Taille en octets du rendu (y compris les enfants)	Taille en octets de ViewState (sans les enfants)	Taille en octets de ControlState (sans les enfants)
__Page	ASP.controle_1_html_aspx	677	0	0
ctl02	System.Web.UI.LiteralControl	174	0	0
ctl00	System.Web.UI.HtmlControls.HtmlHead	135	0	0
ctl01	System.Web.UI.HtmlControls.HtmlTitle	19	0	0
ctl03	System.Web.UI.LiteralControl	103	0	0
ctl04	System.Web.UI.ResourceBasedLiteralControl	368	0	0

Si vous avez compris le modèle client/serveur Web et son fonctionnement en mode déconnecté, vous devez avoir compris que les gestionnaires http (httphandler) n'existent que durant une période très courte, celle du traitement de la requête.

3.2. Les contrôles HTML serveur.

Nous allons maintenant travailler avec des contrôles HTML serveur.
 Nous allons enregistrer la page précédente, où figuraient les contrôles HTML, sous un nouveau nom et ajouter au contrôle HTML l'attribut `runat=server`.
 A noter que le formulaire doit aussi disposer du même attribut.

```
<form id="form1" action="Controle_2_HTMLServeur.aspx" runat="server"
method="post" enctype="application/x-www-form-urlencoded">
<div>
  <label id="lblNom" for="txtNom" runat="server">Entrez votre nom</label>
  <input id="txtNom" name="txtNom" type="text" runat="server"/><br />
  <input id="Submit1" type="submit" value="Envoyer" runat="server" /></div>
</form>
```

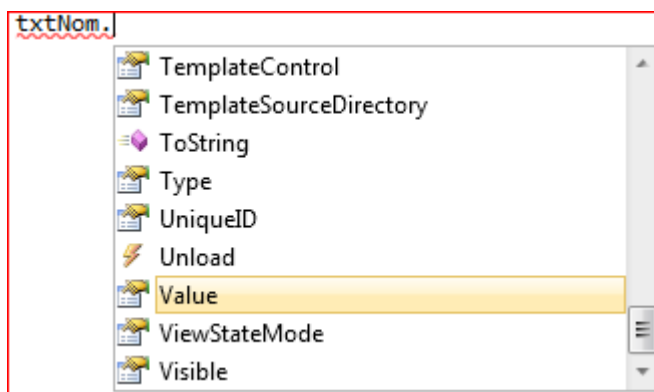
Nous constatons que :

Nous pouvons différencier l'affichage initial de l'affichage résultant d'un postage des données en testant la propriété **IsPostBack** de la page.

L'objet est accessible sous forme de champ dans notre page

```
if (IsPostBack)
    this.Page.Session["Nom"] = txtNom.Value;
```

L'ensemble des attributs du contrôle est accessible en lecture/écriture



Nous pouvons même lui ajouter des attributs personnalisés qui n'existent pas en HTML (non supportés) mais pourront toutefois être traités par le serveur :

```
<input id="txtNom" monattr="Vincent" name="txtNom" type="text" runat="server"/>
```

```
string attribut = txtNom.Attributes["monattr"];
```

La variable attribut contient bien Vincent.

Et la valeur du contrôle est maintenue après postage.

 A screenshot of the web application's output. It shows the text 'Entrez votre nom' followed by a text input field containing the value 'Bostichon'. Below the input field is a button with the text 'Envoyer'.

ASP Net Présentation

Par quel miracle ? De miracle point mais une technique, la gestion d'état par l'intermédiaire d'un objet **viewState** qui a été généré.

```
<div class="aspNetHidden">

```

Cette fonctionnalité est très importante. Il s'agit d'un des principes fondamentaux des applications Web Forms.

La page a pris du poids sans pour autant être tout à fait obèse...
995 octets au lieu de 677 précédemment.

UniqueID du contrôle	Type	Taille en octets du rendu (y compris les enfants)	Taille en octets de ViewState (sans les enfants)	Taille en octets de ControlState (sans les enfants)
_Page	ASP.controle_2_htmlserveur_aspx	995	0	0
ctl02	System.Web.UI.LiteralControl	174	0	0
ctl00	System.Web.UI.HtmlControls.HtmlHead	32	0	0
ctl01	System.Web.UI.HtmlControls.HtmlTitle	19	0	0
ctl03	System.Web.UI.LiteralControl	14	0	0
form1	System.Web.UI.HtmlControls.HtmlForm	755	0	0
ctl04	System.Web.UI.LiteralControl	22	0	0
lblNom	System.Web.UI.HtmlControls.HtmlGenericControl	56	0	0
ctl05	System.Web.UI.LiteralControl	16	0	0
ctl06	System.Web.UI.LiteralControl	11	0	0
txtNom	System.Web.UI.HtmlControls.HtmlInputText	83	0	0
ctl07	System.Web.UI.LiteralControl	16	0	0
Submit1	System.Web.UI.HtmlControls.HtmlInputSubmit	67	0	0
ctl08	System.Web.UI.LiteralControl	12	0	0
ctl09	System.Web.UI.LiteralControl	20	0	0

Les contrôles HTML serveur sont dans l'espace de nom System.Web.UI.HtmlControls.

Lorsque la page aspx est invoquée, le texte HTML correspondant au contrôle HTML serveur est généré en fonction, dans l'ordre :

- Des valeurs des attributs fixés dans le tag html du fichier aspx
- Des modifications apportées par l'utilisateur
- Des modifications apportées par l'objet Page.

L'état des contrôles HTML serveur est conservé entre deux invocations de page comme nous l'avons vu dans l'exemple.

Il est possible de traiter les événements des contrôles HTML serveur côté client :

- L'identifiant du contrôle est connu
- Le tag HTML *fabriqué* par un contrôle serveur HTML est toujours identique.

Il existe un événement côté serveur auquel il peut être associé un gestionnaire, l'événement `onserverchange`. Vous pouvez l'associer dans le fichier aspx comme présenté ci-dessous ou dans la classe Page en code behind.

```
<input id="txtNom" monattr="Vincent" name="txtNom" type="text" runat="server"
onserverchange="ChangementNom"/><br />
```

L'événement `serverchange` est déclenché sur un postback lorsque la nouvelle valeur du contrôle serveur HTML diffère de l'ancienne conservée dans le viewstate.

Vous pouvez utiliser pratiquement tous les contrôles HTML comme contrôles HTML serveur. Voici une liste de ces derniers :

- HtmlAnchor contrôle l'élément <a>.
- HtmlButton contrôle l'élément <input type="button">.
- HtmlForm contrôle l'élément <form>.
- HtmlSelect contrôle l'élément <select>.
- HtmlTable contrôle l'élément <table>.
- HtmlTableCell contrôle l'élément <td>.
- HtmlTableRow contrôle l'élément <tr>.
- HtmlTextArea contrôle l'élément <textarea>.
- HtmlInputButton contrôle l'élément <input type="button">.
- HtmlInputCheckBox contrôle l'élément <input type="checkbox">.
- HtmlInputFile contrôle l'élément <input type="file">.
- HtmlInputHidden contrôle l'élément <input type="hidden">.
- HtmlInputImage contrôle l'élément <input type="image">.
- HtmlInputRadioButton contrôle l'élément <input type="radio">.
- HtmlInputText contrôle l'élément <input type="text">.
- HtmlImage contrôle l'élément .

Comme vous pouvez le voir dans la liste ci-dessus, presque tous les contrôles HTML ont été traités. Certains éléments n'apparaissent toutefois pas. Pour ces derniers, vous pouvez utiliser la classe HtmlGenericControl. Elle est disponible pour tous les éléments HTML qui n'ont pas été implémentés comme HTMLControl.

Exemple pour un paragraphe et un label :

```
<p id="pBonjour" runat="server">Bonjour aux CDI</p>
<label id="lblNom" for="txtNom" runat="server">Entrez votre nom</label>
```

Sur rechargement de la page, après le postage des données, nous modifions le texte de la balise paragraphe et changeons le style couleur du label :

```
protected void Page_Load(object sender, EventArgs e)
{
    if (IsPostBack)
    {
        pBonjour.InnerHtml="La page a été rechargée";
        lblNom.Style["color"]= "#1d60ff";
    }
}
```

Bonjour aux CDI

Entrez votre nom

Envoyer

Affichage initial

La page a été rechargée

Entrez votre nom

Bost

Envoyer

Après postage

Cela ouvre la porte à de nombreuses possibilités d'interaction. Les contrôles HTML serveur peuvent aussi être liés à des données. Nous y reviendrons ultérieurement.

3.3. Les contrôles serveur

Les contrôles serveur Web sont un deuxième ensemble de contrôles.

Les contrôles serveurs Web sont abstraits contrairement aux contrôles HTML. Lorsque la page Web Forms s'exécute, le **contrôle html réel généré dépend du type du navigateur et des paramètres** que vous avez définis pour le contrôle.

Par exemple, un contrôle TextBox peut être rendu sous forme de balise <INPUT> ou <TEXTAREA> en fonction de ses propriétés.

Ils offrent les mêmes fonctionnalités que les contrôles serveur HTML ; de plus, ils s'adaptent au navigateur client en permettant de restituer un balisage HTML approprié, et permettent, sur certains événements, la publication sur le serveur (le postage).

Ils comprennent les contrôles de formulaires traditionnels, tels que les boutons et zones de texte mais aussi des contrôles complexes qui fournissent de nombreuses fonctionnalités.

Nous citerons parmi ces contrôles évolués, ceux qui permettent la manipulation de données en grille ou en liste, ceux qui permettent de gérer des éléments dans un treeView, des contrôles qui permettent de choisir des dates, afficher des menus...

La plupart de ces contrôles héritent de la classe Web.UI.WebControl et se situent dans l'espace de noms System.Web.UI.WebControls.

Prenons l'exemple suivant pour illustrer la notion d'**abstraction** de ces contrôles.

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label ID="Label1" runat="server" Text="Label">Votre Nom</asp:Label>
      <asp:TextBox ID="txtNom" runat="server" ></asp:TextBox><br />
      <asp:Label ID="Label2" runat="server" Text="Label">Votre mot de passe</asp:Label>
      <asp:TextBox ID="txtMotPasse" runat="server" TextMode="Password"></asp:TextBox><br />
      <asp:Label ID="Label3" runat="server" Text="Label">Votre texte</asp:Label>
      <asp:TextBox ID="txtTexte" runat="server" TextMode="MultiLine"></asp:TextBox>
    </div>
  </form>
</body>
```

Nous avons un formulaire et 3 contrôles Web qui nous intéressent particulièrement de type TextBox. Vous remarquerez que les contrôles serveur sont toujours préfixés de asp :.

Les propriétés de deux de ces contrôles ont été modifiées :

- La propriété TextMode du contrôle txtMotPasse vaut « PassWord »
- La propriété TexteMode du contrôle txtTexte vaut « MultiLine »

Exécutons notre page et regardons le code de la page web générée.

```
<div>
  <span id="Label1">Votre Nom</span>
  <input name="txtNom" type="text" id="txtNom" /><br />
  <span id="Label2">Votre mot de passe</span>
  <input name="txtMotPasse" type="password" id="txtMotPasse" /><br />
  <span id="Label3">Votre texte</span>
  <textarea name="txtTexte" rows="2" cols="20" id="txtTexte">
</textarea>
</div>
```

Vous pouvez constater que 3 contrôles HTML de type différents ont été générés :

- Un contrôle **input** de type **text**
- Un contrôle **input** de type **password**
- Un contrôle **textarea**

Vous avez ici l'illustration du caractère **abstrait** de ces contrôles, tous de type **TextBox**, qui donneront **différents contrôles HTML concrets**.

3.3.1. Prise en compte du navigateur

La génération du code effectué par le **parser** prend en compte la version du navigateur cible.

De même, en fonction du type de navigateur, les propriétés des contrôles serveur n'auront pas le même impact.

Nous pouvons citer pour l'exemple :

- La propriété `BackColor` qui n'est supportée que par les contrôles conteneur pour les navigateurs anciens
- La propriété `BorderStyle` qui n'a pas d'équivalent en HTML 3.2
- La propriété `Width` qui ne fonctionne qu'en pourcentage ou en pixels sur les navigateurs anciens
- ...

3.3.2. Les événements

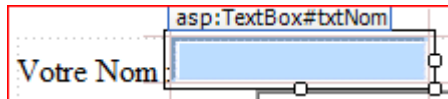
Vous pouvez toujours utiliser les événements côté client pour les contrôles serveur.

Les événements côté serveur sont par contre plus nombreux et leur comportement diffère des contrôles HTML serveur.

Vous pouvez assigner les gestionnaires d'événement serveur aux événements du contrôle dans le code behind comme vous le feriez dans l'environnement Windows :

```
this.txtNom.TextChanged+=new EventHandler(txtNom_TextChanged);
```

Mais aussi en mode Design , en exécutant un double-click sur le contrôle, le gestionnaire d'événement est généré dans le code behind et l'association avec l'événement est codé au niveau de la balise HTML.

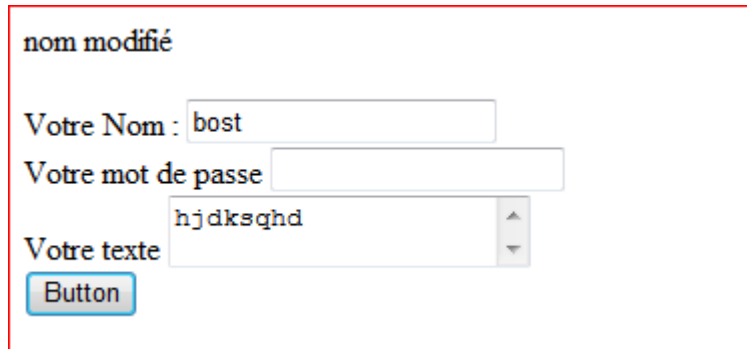


```
<asp:TextBox ID="txtNom" runat="server" ontextchanged="txtNom_TextChanged" >
```

Nous modifions le gestionnaire pour qu'il génère un paragraphe notifiant la modification du nom :

```
protected void txtNom_TextChanged(object sender, EventArgs e)
{
    this.Page.Response.Write("<p>nom modifié</p>");
}
```

Nous observons le résultat produit lorsque nous postons les données :



Les contrôles serveur possèdent une propriété **AutoPostBack**.

Si vous positionnez celle-ci à **true**, le postage des données sera effectué sur l'événement.

Attention à user de cet événement à bon escient pour éviter de nombreux aller-retour entre le client et le serveur. N'oubliez pas que votre application sera accessible via le réseau Internet...

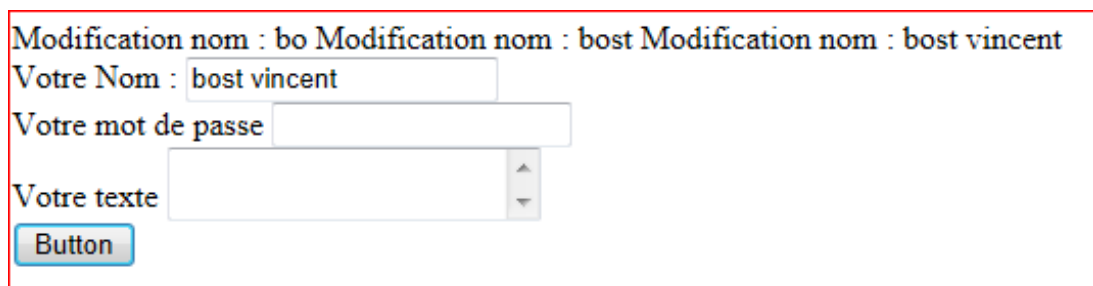
Après modification du gestionnaire d'événement associé à la modification de la boîte de texte.

```
protected void txtNom_TextChanged(object sender, EventArgs e)
{
    this.lModifications.Text += string.Format("Modification nom : {0} |", this.txtNom.Text) ;
}
```

Et l'assignation de la valeur true à la propriété AutoPostBack :

```
<asp:TextBox ID="txtNom" runat="server" ontextchanged="txtNom_TextChanged"
    AutoPostBack="True" ></asp:TextBox ><br />
```

Nous obtenons le résultat après 3 validations du champ txtNom :



4. Gestion du ViewState

Nous allons nous intéresser de plus près à la gestion d'état, terme d'usage en français pour qualifier le ViewState.

Le ViewState est un concept ajouté dans le Framework .NET afin d'améliorer les possibilités de développement dans le cas d'applications WEB.

Il s'agit en fait de dépasser la limite imposée par l'architecture du Web et pouvoir stocker de l'information entre deux états lors d'une navigation sur un site Web.

Nous avons déjà abordé ce sujet et avons vu plusieurs méthodes qui permettent d'obtenir ce résultat dont :

- Les Variables de Sessions
- Les Cookies
- Le Stockage en Base de données (seulement évoqué et non encore mis en œuvre)

Chacune de ces trois méthodes dispose de ces avantages mais aussi de limites :

- Les cookies ne sont pas fiables. Nous n'avons aucune garantie sur la configuration du client quant à l'acceptation de ces cookies.
- Les variables de Session sont limitées dans la durée et limitées à la session.
- Le stockage en Base de Données n'est pas adapté pour le maintien de valeurs entre deux appels de page du fait de sa complexité et de la dégradation des performances.

HTML permet par ailleurs de stocker de l'information sans qu'elle soit nécessairement affichée à l'utilisateur. Il s'agit d'une approche déjà mise en œuvre dans de nombreux développements Web, en ASP (précédente technologie) ou PHP.

Le Framework .NET utilise par ailleurs énormément le XML pour la "Serialisation" (les propriétés des contrôles par exemple).

La combinaison des deux a donné lieu à la technique du ViewState.

Microsoft permet donc le stockage de l'ensemble des informations de tous les composants contrôlés par le serveur dans un champ HIDDEN de la page en utilisant la "Serialisation" XML et surtout une classe développée pour cet usage : le StateBag.

4.1.1. Fonctionnement

Pour comprendre ces mécanismes, mettons en œuvre l'affichage d'une liste et le choix d'un élément de celle-ci.

Il s'agit ici d'une liste déroulante chargée à l'initialisation de la page.

Sur modification de l'élément sélectionné de la liste, les données sont postées et la propriété text de l'élément sélectionné affiché dans une boîte de texte.

```
<form id="form1" runat="server">
<div>
  <asp:Label ID="Label1" runat="server" Text="Label">
    Sélectionnez un stagiaire</asp:Label><br />
  <asp:DropDownList ID="ddlStagiaires" runat="server" AutoPostBack="True">
    </asp:DropDownList><br />
  <asp:Label runat="server" Text="Label">Nom du stagiaire</asp:Label>
  <asp:TextBox ID="txtStagiaire" runat="server"></asp:TextBox>
</div>
</form>
```

L'événement de changement de sélection dans la liste provoque le postage des données.

A l'exécution, le rendu est conforme à nos attentes :

Sélectionnez un stagiaire

Adeline ▼

Nom du stagiaire Adeline

Code de la page associée (code behind)

```
protected void Page_Load(object sender, EventArgs e)
{
    this.ddlStagiaires.SelectedIndexChanged
        += new EventHandler(ddlStagiaires_SelectedIndexChanged);
    if (!this.IsPostBack)
    {
        ListItem item = null;
        item = new ListItem("Serge", "1");
        ddlStagiaires.Items.Add(item);
        item = new ListItem("Adeline", "2");
        ddlStagiaires.Items.Add(item);
        item = new ListItem("Thomas", "3");
        ddlStagiaires.Items.Add(item);
    }
}

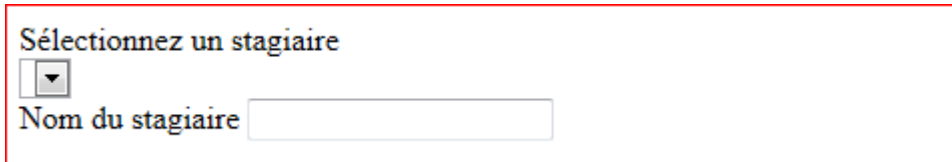
void ddlStagiaires_SelectedIndexChanged(object sender, EventArgs e)
{
    txtStagiaire.Text = ddlStagiaires.Items[ddlStagiaires.SelectedIndex].Text;
}
```

ASP Net Présentation

Que se passe-t-il si nous modifions maintenant la propriété `EnableViewState` de notre contrôle `DropDownList` et le faisons passer à `false`.

```
<asp:DropDownList ID="ddlStagiaires" runat="server" AutoPostBack="True" EnableViewState="false">
</asp:DropDownList><br />
```

Exécutons de nouveau la page. Sur modification de l'élément sélectionné nous obtenons le résultat suivant :



Sélectionnez un stagiaire

Nom du stagiaire

La liste n'est chargée que lors de l'initialisation lors que la page n'est pas Postback. Elle n'a pas conservée ces valeurs et nous avons ainsi perdu la référence à l'élément sélectionné.

Dans ce contexte où nous souhaitons réafficher la page, il est donc impératif de conserver l'état de ce contrôle pour que notre application Web conserve un fonctionnement correct.

Il est donc nécessaire d'agir à bon escient en fonction du contrôle choisi.

Si nous ne souhaitons pas réafficher la page sur postBack, nous pourrions alors décider de ne pas utiliser de viewState.

4.1.2. ViewState et performance

Le mécanisme de gestion d'état peut entrainer des dégradations sensibles de performances. Lorsque vous avez par exemple une grille de données comportant de nombreuses lignes, cela peut nuire aux temps de réponse.

La gestion d'état est par défaut activé pour l'ensemble des contrôles d'une page. Vous pouvez modifier ce comportement en spécifiant dans la directive de page que vous souhaitez par défaut désactiver la gestion d'état.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ControleViewState2.aspx.cs"
EnableViewState="false" Inherits="SupportCours.ControleViewState2" %>
```

Vous devrez alors préciser explicitement l'activation du Viewstate pour chaque contrôle devant conserver son état.

```
<asp:DropDownList ID="ddlStagiaires" runat="server" AutoPostBack="True" EnableViewState="true">
</asp:DropDownList><br />
```

Nous verrons aussi que le viewstate peut être désactivé pour certaines transactions, comme par exemple des grilles de données en lecture seule si vous souhaitez ne pas mettre en place les opérations de tri ou de pagination des éléments au niveau de la grille.

5. Cycle de vie d'une page asp.net

Une instance de page est créée à chaque requête du client. Le cycle de vie consiste en une séquence de phases et d'étapes. Certaines de ces phases peuvent être contrôlées par des événements du code utilisateur ; d'autres encore (sous-phases) ne sont pas publiques et sont hors de contrôle du développeur.

Une chose est certaine, il faut être extrêmement vigilant en asp.net sur l'ordre et la nature des événements, en particulier si vous souhaitez instancier des contrôles dynamiquement sur la page.

Le tableau suivant répertorie les principaux événements de base du cycle de vie de la page que vous utiliserez le plus souvent et explique aussi l'ordre dans lequel surviennent les événements sur les contrôles.

Vous ne pouvez intégrer toutes les précisions données ici pour le moment. Nous n'avons pas encore abordé les notions de thèmes et skin, ni la création dynamique de contrôles, ni les processus de validation des contrôles.

Nous y reviendrons ultérieurement et verrons par la suite comment utiliser au mieux certains événements dans des contextes moins courants et des techniques plus avancées.

5.1. Les principaux événements

Événement Page	Utilisation courante
DeterminePostBackMode	Détermine si l'on est dans une phase de postage des données. Si c'est le cas, la variable isPostBack est alors positionnée à true. C'est au cours de cet événement que sont chargées les données transmises par get ou post.
PreInit	Les propriétés définies en design sont initialisées. À l'aide de la propriété IsPostBack , vérifier si la page est traitée pour la première fois. C'est le bon endroit pour : <ul style="list-style-type: none"> Créer ou recréer des contrôles dynamiques. Définir dynamiquement une page maître. Définir la propriété Theme dynamiquement. Lire ou définir des valeurs de propriétés de profil.

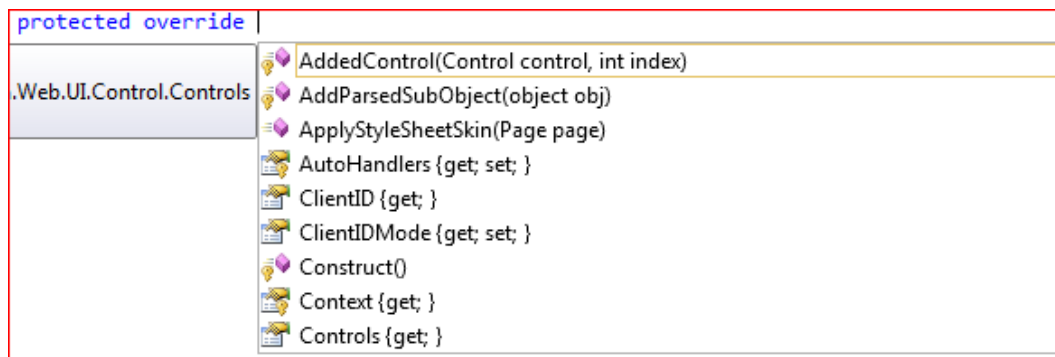
Init	<p>L'évènement Init est déclenché pour tous les contrôles enfant (de manière récursive) puis en dernier pour la page.</p> <p>Au cours de cet événement il est possible de lire les propriétés assignées en designTime mais nous n'avons pas encore accès aux nouvelles valeurs saisies : il faut attendre que LoadPostData soit survenu.</p>
PreLoad	<p>Évènement géré par code utilisateur. Intervient avant le load et le chargement récursif des contrôles figurant sur la page.</p>
Load	<p>Évènement géré par code utilisateur. L'évènement Load est déclenché pour la page, puis récursivement pour tous les contrôles enfant. L'ordre page/contrôles est inversé par rapport à init. Il se produit quand la page se charge. Les propriétés des contrôles sont accessibles.</p>
InitComplete	<p>Évènement uniquement valable pour la page. Utilisé pour effectuer des actions lorsque tous les éléments ont été initialisés (contrôles et page)</p>
LoadViewState	<p>Si isPostBack est vrai, asp.net désérerialize les informations du viewState pour les assigner aux contrôles.</p>
LoadPostBackData	<p>Si isPostBack est vrai, asp.net affecte les valeurs aux contrôles.</p>

Évènement Page	Utilisation courante
<p>Evénements associés aux contrôles</p> <p>Control Events</p>	<p>Traitement propre à votre application : Si la page contient des contrôles validateurs (chapitre suivant), vérifiez la propriété IsValid de la page et des contrôles de validation avant d'exécuter tout traitement.</p> <p>Les événements sont classés en 4 catégories et sont déclenchés dans l'ordre suivant :</p> <p>Evènements à l'origine du retour par un appel à la fonction <code>_doPostBack()</code></p> <p>Evènements de changement mis en cache à moins que la propriété <code>AutoPostBack</code> soit égale à true</p> <p>Evènements de changement : <code>TextChanged</code>, <code>SelectIndexChanged</code></p> <p>Evènements d'action tels que <code>Click</code></p>

PreRender	Se produit quand la page est sur le point d'être affichée Pour apporter des modifications définitives au contenu de la page affectant le rendu visuel.
SaveStateComplete	Le viewState est enregistré. Il ne pourra plus être modifié.
Render	Méthode qui effectue le rendu de chaque contrôle
Unload	Exécuter une ultime tâche de nettoyage et libérer les ressources. Cela peut être : Fermer des fichiers et des connexions de base de données ouverts. Terminer d'enregistrer le journal ou d'autres tâches spécifiques à la demande. L'instance de la page est détruite

Pour intervenir sur ces méthodes et les surcharger le cas échéant, vous devez indiquer que vous souhaitez substituer un comportement aux méthodes de base en tapant

protected override



Visual Studio vous proposera alors la liste des méthodes de la classe de base.

6. Contrôles de validation

La technique mise en œuvre au travers des contrôles de validation tente de satisfaire deux objectifs en lien avec le contrôle des données acquises au travers d'une interface utilisateur Web :

- Eviter de nombreux aller-retour inutiles entre le client et le serveur lorsque les données postées ne sont pas correctes et que les contrôles de ses données sont réalisés côté serveur
- Eviter la codification et la mise au point de méthodes de validation côté client pour s'assurer avant postage que les données sont correctes

Microsoft a donc réalisé des classes spécialisées dans le contrôle et la validation des formulaires. Elles sont livrées sous forme de composants disponibles dans l'espace de noms System.Web.UI.WebControls.

L'utilisation de ces contrôles génère une validation côté client et une validation côté serveur.

En effet, ASP.NET génère automatiquement du code javascript qu'il insère dans la page HTML renvoyée au client. Ce code sera exécuté si le navigateur du client supporte javascript.

Vous pouvez décider d'empêcher la validation côté client en positionnant la propriété EnableClientScript à false.

Après avoir placé le contrôle de validation d'entrée dans la page, utilisez la fenêtre des propriétés pour entrer les propriétés propres au contrôle.

Plusieurs contrôles de validation peuvent être associés à un même contrôle.

La syntaxe partagée des contrôles de validation des entrées est la suivante :

```
<asp: type_de_valdateur id="id_valdateur" runat="server"  
ErrorMessage="message erreur " ControlToValidate="id du contrôle"  
Display ="static/dynamic/none" Text ="Texte à afficher"></asp: type_de_valdateur  
>
```

La propriété Text correspond au texte de remplacement affiché à l'emplacement du contrôle de validation, lorsque la propriété ErrorMessage et la propriété Text sont utilisées lors du déclenchement du contrôle de validation.

Il existe un contrôle qui permet de regrouper l'ensemble des messages d'erreur, ValidationSummary.

Lorsqu'un contrôle ValidationSummary est utilisé pour regrouper les messages d'erreur, un astérisque (*) rouge est affiché.

Ces contrôles de validation partagent aussi d'autres points en commun :

- La validation est le plus souvent déclenchée sur le click associé à un bouton
- La validation peut être suspendue à l'aide de la propriété causesValidation des contrôles
- Côté serveur, la page n'est validée qu'après l'invocation de la méthode Validate invoquée automatiquement lors d'un postBack dû à un bouton où causesValidation vaut vrai. Une propriété IsValid peut être testée côté serveur.

Quelques explications sur les propriétés communes.

La propriété **Display** définit l'espacement des messages d'erreur de plusieurs contrôles de validation :

- Static (par défaut) : la présentation est fixe pour la page
- Dynamic empêche l'affichage d'espaces lorsque le contrôle n'a pas déclenché d'erreurs
- None : pas d'affichage

La propriété **ControlToValidate** permet de nommer le nom du contrôle à valider.

La propriété **EnableClientScript** définit si la validation côté client doit être effectuée

La propriété **IsValid** peut être testée pour savoir si le contrôle a été validé avec succès.

La propriété **SetFocusOnError** permet de préciser s'il faut mettre le focus sur le contrôle à valider si l'information vérifiée est invalide (indépendamment de l'échec client ou serveur).

Il existe également une notion de groupe de validation avec la propriété **ValidationGroup** qui permet de partitionner l'ensemble des contrôles d'une page.

Un clic sur un bouton déclenchera seulement la validation des contrôles de son groupe.

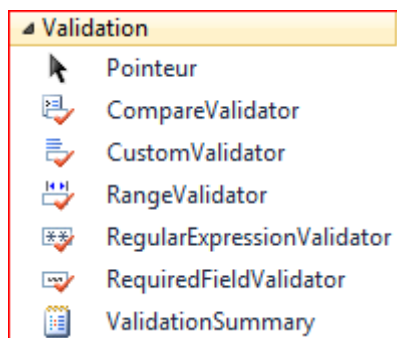
Une fonction ou un contrôle de validation unique est parfois insuffisant pour vérifier si l'utilisateur a correctement entré des données dans un contrôle d'entrée.

Par exemple, un contrôle TextBox d'un numéro de téléphone peut nécessiter :

- Une valeur obligatoire : **RequiredFieldValidator**
- Un contrôle de format : **RegularExpressionValidator**
- Un contrôle spécifique programmé au sein d'une fonction : **CustomValidator**

En résumé, il existe 6 classes de composants spécialisées dans la validation des formulaires :

1. Le composant RequiredFieldValidator : valeur requise
2. Le composant RangeValidator : opération de comparaison entre bornes
3. Le composant CompareValidator : opération de comparaison de valeur
4. Le composant RegularExpressionValidator : contrôle du format d'une chaîne à l'aide d'une expression régulière
5. Le composant CustomValidator : programmé au sein d'une fonction il autorise tout type de contrôle y compris des vérifications associées à des accès à la base de données (dans ce dernier cas il ne peut être joué côté client).



Les contrôles de validation se trouvent dans l'onglet de la boîte à outils Validation

6.1. Le contrôle de valeur requise : RequiredFieldValidator

Ce contrôle permet de vérifier que l'utilisateur a bien saisi une valeur.

Si la propriété InitialValue est précisée, le contrôle vérifiera que l'utilisateur a saisi une autre valeur que la valeur initiale définie.

Dans cet exemple, nous devons vérifier la présence du nom et du prénom.

Nous devons aussi nous assurer que la valeur affectée au nom diffère de Votre nom. Nous aurons donc besoin d'un deuxième contrôle de validation sur le nom.

La définition des boîtes de texte et des contrôles de validation.

Un contrôle résumé a été ajouté.

```
<div>
  <asp:Label ID="Label1" runat="server" Text="Votre Nom :"></asp:Label>
  <asp:TextBox ID="txtNom" runat="server" Text="Votre nom"></asp:TextBox>
  <asp:RequiredFieldValidator ID="rfvNom" runat="server" ErrorMessage="Le nom est obligatoire"
    Display="Dynamic" ControlToValidate="txtNom"> </asp:RequiredFieldValidator>
  <asp:RequiredFieldValidator ID="rfvNom1" runat="server" ErrorMessage="Le nom est obligatoire"
    Text="Nom obligatoire" Display="Dynamic" InitialValue="Votre nom" ControlToValidate="txtNom">
  </asp:RequiredFieldValidator>
  <br />
  <asp:Label ID="Label2" runat="server" Text="Label">Votre prénom :</asp:Label>
  <asp:TextBox ID="txtPrenom" runat="server"></asp:TextBox>
  <asp:RequiredFieldValidator ID="rfvPre" runat="server" ErrorMessage="Le prénom est obligatoire"
    Display="Dynamic" ControlToValidate="txtPrenom">
  </asp:RequiredFieldValidator>
  <br />
  <asp:Button ID="btnEnvoyer" runat="server" Text="Envoyer" />
  <asp:ValidationSummary ID="ValidationSummary1" runat="server"
    BorderStyle="Solid" BorderWidth="2" BorderColor="Red" Width="33 %" />
</div>
```

Résultat

Votre Nom : Nom obligatoire

Votre prénom : Le prénom est obligatoire

- Le nom est obligatoire
- Le prénom est obligatoire

Extrait du code généré côté client :

```
<script type="text/javascript">
  //
  var Page_ValidationSummaries = new Array(document.getElementById("ValidationSummary1"));
  var Page_Validators = new Array(document.getElementById("rfvNom"), document.getElementById("rfvNom1"),
  document.getElementById("rfvPre"));
  //]]&gt;
&lt;/script&gt;

&lt;script type="text/javascript"&gt;
  //<![CDATA[
  var rfvNom = document.all ? document.all["rfvNom"] : document.getElementById("rfvNom");
  rfvNom.controltovalidate = "txtNom";
  rfvNom.errorMessage = "Le nom est obligatoire";
  rfvNom.display = "Dynamic";
  rfvNom.evaluationfunction = "RequiredFieldValidatorEvaluateIsValid";
  rfvNom.initialvalue = "";</pre>
</div>
<div data-bbox="128 945 346 964" data-label="Page-Footer">Bost - Afpa Pays de Brive</div>
<div data-bbox="517 945 631 964" data-label="Page-Footer">Page : 21-26</div>
```

6.2. Les contrôles de valeur CompareValidator et RangeValidator

Nous ajoutons deux contrôles à notre formulaire dont les valeurs sont optionnelles accompagnés de deux contrôles de validation sur valeur.

Un champ qui représente la note facultative comprise entre 0 et 20

```
<asp:Label ID="Label3" runat="server" Text="Label">Saisissez la note si évaluation :</asp:Label>
<asp:TextBox ID="txtNote" runat="server"></asp:TextBox>
<asp:RangeValidator ID="RequiredFieldValidator1" runat="server"
ErrorMessage="La note doit être comprise entre 0 et 20"
Display="Dynamic" ControlToValidate="txtNote" MaximumValue="20" MinimumValue="0" Type="Double">
</asp:RangeValidator>
```

L'âge facultatif mais entier et supérieur à 18 ans si renseigné

```
<asp:Label ID="Label4" runat="server" Text="Label">Saisissez l'âge (facultatif) :</asp:Label>
<asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
<asp:CompareValidator ID="RangeValidator1" runat="server"
ErrorMessage="La personne doit être majeure Age > 18"
Display="Dynamic" ControlToValidate="txtAge" Type="Integer" ValueToCompare="18"
Operator="GreaterThanEqual">
</asp:CompareValidator>
```

Résultat :

Votre Nom : Nom obligatoire

Votre prénom : Le prénom est obligatoire

Saisissez la note si évaluation : La note doit être comprise entre 0 et 20

Saisissez l'âge (facultatif) : La personne doit être majeure Age > 18

- Le nom est obligatoire
- Le prénom est obligatoire
- La note doit être comprise entre 0 et 20
- La personne doit être majeure Age > 18

6.3. Le contrôle personnalisé CustomValidator

Si les validations proposées par les classes de contrôle précédentes sont insuffisantes, il est possible de fournir sa propre logique de validation.

Un gestionnaire (handler) client et/ou serveur doit être ajouté. Détails sur les propriétés :

- La propriété ClientValidationFunction spécifie le script que doit exécuter le contrôle côté client.
- OnServerValidate spécifie le script que doit exécuter le contrôle sur le serveur.
- La propriété ValidateEmptyText permet de définir ou d'obtenir une valeur booléenne indiquant si le texte vide doit être validé.

Exemple : Contrôle de la parité d'un entier

```
<asp:Label ID="Label5" runat="server" Text="Label">Saisissez un nombre pair :</asp:Label>
<asp:TextBox ID="txtPair" runat="server"></asp:TextBox>
<asp:CustomValidator ID="cv1" runat="server" ClientValidationFunction="IsPairC"
    ControlToValidate="txtPair" ErrorMessage="Doit être un nombre pair !"
    OnServerValidate="IsPairS"></asp:CustomValidator>
<br />
```

6.3.1. La fonction côté client.

La signature sera toujours constituée de deux objets source et args.

La propriété de args **IsValid** permet de définir la valeur du contrôle vérifié comme valide ou non.

```
<script type="text/javascript">
    function IsPairC(source, args) {
        if (args.Value % 2 == 0)
            args.IsValid = true;
        else
            args.IsValid = false;
    }
</script>
```

6.3.2. La fonction côté serveur.

La signature sera toujours constituée de deux objets source et données d'événement de type ServerValidateEventArgs qui possède lui aussi une propriété **IsValid** qui permet de définir la valeur du contrôle vérifié comme valide ou non.

```
protected void IsPairS(object source, ServerValidateEventArgs args)
{
    if ((Convert.ToInt32(args.Value) % 2) != 0)
        args.IsValid = false;
    else
        args.IsValid = true;
}
```

Résultat obtenu lors de l'exécution :

Votre Nom : Nom obligatoire
 Votre prénom : Le prénom est obligatoire
 Saisissez la note si évaluation :
 Saisissez l'âge (facultatif) :
 Saisissez un nombre pair : 3 Doit être un nombre pair

- Le nom est obligatoire
- Le prénom est obligatoire
- Doit être un nombre pair !

6.4. Le contrôle RegularExpressionValidator

Il est utilisé pour vérifier si l'entrée utilisateur correspond à un modèle prédéfini (numéro de téléphone, code postal ...),

La propriété ValidationExpression permet de rentrer une chaîne de caractères, conforme au modèle choisi : Visual Studio .Net propose un ensemble de modèles d'expressions régulières prédéfinies ; pour créer un modèle, sélectionner le modèle Personnalisé : le dernier modèle utilisé permet de créer son propre modèle.

Le tableau ci-dessous présente les différents caractères de contrôle :

Caractère	Définition
a	La lettre a doit être utilisée en minuscule
1	Le chiffre 1 doit être utilisé
?	0 ou 1 élément
*	De 0 à n éléments
+	De 1 à n éléments (au moins 1)
[0-n]	Valeur entière comprise entre 0 et n
{n}	La longueur doit être de n caractères
	Sépare plusieurs modèles
\	Le caractère suivant est un caractère de commande
\w	Doit contenir un caractère
\d	Doit contenir un chiffre
\.	Doit contenir un point

L'expression utilisée pour la vérification d'un mail génère l'expression suivante :

```

<asp:Label ID="Label6" runat="server" Text="Label">Saisissez votre adresse mail :</asp:Label>
<asp:TextBox ID="txtMail" runat="server"></asp:TextBox>
<asp:RegularExpressionValidator runat="server" ControlToValidate="txtMail"
    ErrorMessage="Mail non valide" ValidationExpression="\w+([-+.']\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*"
></asp:RegularExpressionValidator>
<br />

```


Code JavaScript généré :

```
var ctl02 = document.all ? document.all["ctl02"] : document.getElementById("ctl02");
ctl02.controltovalidate = "txtMail";
ctl02.errormessage = "Mail non valide";
ctl02.evaluationfunction = "RegularExpressionValidatorEvaluateIsValid";
ctl02.validationexpression = "\\w+([-+\\.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*";
//]]>
</script>
```

Résultat obtenu :

Votre Nom :

Votre prénom : Le prénom est obligatoire

Saisissez la note si évaluation :

Saisissez l'âge (facultatif) :

Saisissez un nombre pair :

Saisissez votre adresse mail :

- Le prénom est obligatoire

6.5. Déterminer si la page est valide

La plateforme .Net permet de vérifier si tous les contrôles d'une page sont valides avant d'exécuter une opération, côté client à l'aide du contrôle ValidationSummary, côté serveur par la propriété Page.IsValid

La propriété IsValid vérifie que tous les contrôles de validation présents sur la page sont valides : on peut l'utiliser, par exemple dans le gestionnaire d'évènement du bouton de soumission des données. En cas d'erreur, la page est automatiquement retournée au navigateur.

```
protected void btnEnvoyer_Click(object sender, EventArgs e)
{
    if (this.Page.IsValid)
    {
        // prendre en compte les données transmises SStockage ?
    }
}
```

Dans un formulaire avec plusieurs données à saisir, il peut être souhaitable de réunir tous les messages d'erreur des contrôles de validation d'un groupe particulier à un seul endroit.

C'est le cas dans l'exemple créé avec le contrôle **ValidationSummary**.

Il s'affiche lorsque la propriété `Page.IsValid` retourne la valeur `false`.

Il est généralement placé près du bouton d'envoi, il peut afficher un message ou une zone de texte comprenant une en-tête et une liste d'erreurs, affichée soit sous forme de liste à puces, soit sous un seul paragraphe.

- La propriété **HeaderText** spécifie l'en tête du message affiché
- La propriété **DisplayMode** (`List/ BulletList/SingleParagraph`) spécifie le mode d'affichage
- La propriété **ShowMessageBox** permet d'afficher ce contrôle dans une `MessageBox`
- La propriété **ShowSummary** affichera les erreurs sur la page Web

La propriété `ErrorMessage` s'affiche dans le contrôle `ValidationSummary` et un (*) indique la propriété en erreur.