Validation et navigation

Sommaire

1 La validation

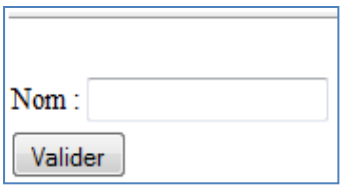
Il existe dans le Framework des contrôles dédiés à la validation. C'est-à-dire qu'il existe des contrôles permettant, par exemple, la vérification des champs d'un formulaire : si le champ est vide, si les deux mots de passe correspondent, si le champ correspond aux expressions régulières, etc. Ils nous permettent de vérifier du côté du client les informations qu'il va envoyer. Ce qui n'empêche pas de vérifier du côté serveur les informations envoyées par le client puisqu'il est tout à fait possible de contourner ces vérifications. Cela étant, ce sont des *Controls* déjà prêt à l'emploi qui permettent de gagner du temps lors du développement.

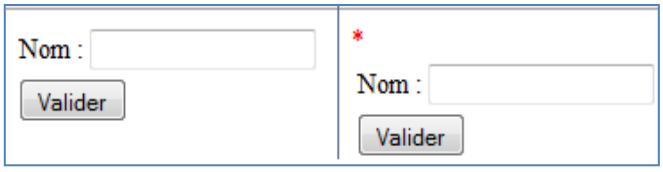
Ces contrôles sont indépendants du contrôle qu'ils vérifient. Il y a donc une manière pour les lier (nous verrons cela plus en détails pour chacun d'entre eux). Chacun de ces contrôles ont une propriété Display qui permet de choisir la façon dont il s'affiche. La valeur None signifie qu'il ne s'affichera pas, Dynamic veut dire que le contrôle ne sera pas affiché jusqu'à ce qu'il y ait un clic sur le bouton. A ce moment les contrôles qui ne sont pas validés auront le message contenu dans la propriété Text qui s'affiche. Enfin elle peut prendre la valeur Static (c'est la valeur par défaut). Cela veut dire que quoi qu'il arrive la place pour l'affichage du message d'erreur sera réservée. Donc affiché ou non, le message aura une place prise. Voyons cela avec un exemple.

ASPX

```
<asp:RequiredFieldValidator display="static" runat="server" Text="*"
ControlToValidate="LastName_id" />

<asp:Table runat="server">
     <asp:TableRow runat="server">
         <asp:TableCell runat="server">
             Nom :
         </asp:TableCell>
         <asp:TableCell runat="server">
             <asp:TextBox ID="LastName_id" MaxLength="20" runat="server" />
         </asp:TableCell>
     </asp:TableRow>

     <asp:TableRow ID="TableRow6" runat="server">
         <asp:TableCell ID="TableCell15" ColumnSpan="2" runat="server">
             <asp:Button ID="Button1" runat="server" Text="Valider" />
         </asp:TableCell>
     </asp:TableRow>
</asp:Table>
```

En *Static* comme sur l'image ci-contre on voit bien qu'il y a un espace en haut qui est blanc. Cet espace est l'endroit où s'affichera le texte du message d'erreur.





Cette image correspond à une valeur *Dynamic*. On remarque que lorsque la validation ne relève pas d'erreur alors il n'y

a pas d'espace en haut. Dans le cas contraire, le texte apparait et prend l'espace. Pour le *none*, le texte ne s'affichera jamais et la page ressemblera donc à la partie gauche de l'image du *Dynamic*.

1.1 RequiredFieldValidator

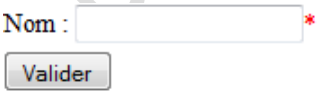
Le *RequiredFieldValidator* oblige l'utilisateur à ne pas laisser le champ ciblé vide. L'exemple ci-dessous affiche une *TextBox* où l'on doit rentrer son nom. Si la zone de texte est vide alors il ne passera pas la validation, le formulaire ne sera pas envoyé et par conséquent il n'y aura pas de *PostBack*.

Pour lier ce contrôle à un champ, il faut remplir la propriété *ControlToValidate* avec l'*ID* du champ à vérifier. La propriété *Text* correspond au message à afficher si le contrôle ne passe pas la validation. Attention : le texte va s'afficher à l'endroit où se trouve le contrôle de validation et non pas après le contrôle à valider. Pour qu'il s'affiche juste après il faut donc mettre le contrôle de validation juste après le contrôle qu'il valide.

ASPX

```
<asp:Table runat="server">
    <asp:TableRow runat="server">
        <asp:TableCell runat="server">
            Nom :
        </asp:TableCell>
        <asp:TableCell runat="server">
            <asp:TextBox ID="LastName_id" MaxLength="20" runat="server" />

            <asp:RequiredFieldValidator runat="server" Text="*"
ControlToValidate="LastName_id" />

        </asp:TableCell>
    </asp:TableRow>

    <asp:TableRow runat="server">
        <asp:TableCell ColumnSpan="2" runat="server">
            <asp:Button runat="server" Text="Valider" />
        </asp:TableCell>
    </asp:TableRow>
</asp:Table>
```

L'image de droite nous montre ce que donne le code ci-dessus lorsque la *TextBox* n'a pas encore été sélectionnée et que le bouton de validation n'a pas été cliqué. En revanche si on clic sur le bouton sans avoir mis de valeur dans la *TextBox,* le message d'erreur sera affiché, comme le montre l'image ci-contre.



La propriété *InitialValue* permet de définir quelle est la valeur interdite. Par exemple, si on met cette valeur à 0, alors laisser le champ vide sera autorisé mais pas le fait d'y mettre un 0.

1.2 CompareValidator

Ce contrôle de validation va nous permettre de comparer un champ à une valeur ou à un autre champ. Là encore on doit utiliser la propriété *ControlToValidate* tout comme le *RequiredFieldValidator*. *ControlToCompare* est une propriété prenant comme valeur l'*ID* du contrôle avec lequel il sera comparé (par exemple une autre TextBox pour rentrer le mot de passe). Par défaut il vérifiera si les deux champs sont égaux (Equal). On peut changer ceci avec la propriété Operator. Il peut prendre plusieurs valeurs comme le montre l'image.

DataTypeCheck	Vérifie le type de donnée
Equal	Vérifie l'égalité
GreaterThan	Vérifie si le contrôle à valider est plus grand
GreaterThanEqual	Vérifie si le contrôle à valider est plus grand ou égal
LessThan	Vérifie si le contrôle à valider et plus petit
LessThanEqual	Vérifie si le contrôle à valider et plus petit ou égal
NotEqual	Vérifie la non égalité

Comme on l'a dit on peut comparer avec un autre contrôle. Mais on peut aussi comparer avec une valeur que l'on rentrera nous même. La valeur *DataTypeCheck* permet de comparer le type des données ce qui n'est pas le cas par défaut. Il faut donc spécifier le type à l'aide de la propriété *Type* du contrôle de validation : *Currency* (valeur monétaire), *Date*, *Double*, *Integer* et *String*. Pour les autres opérateurs, il faut obligatoirement une valeur de comparaison contenu soit dans un contrôle spécifié avec *ControlToCompare,* soit avec une valeur que l'on indique dans *ValueToCompare*.

Pour mieux comprendre voici un exemple :

ASPX

```
<asp:Table runat="server">
   <asp:TableRow runat="server">
        <asp:TableCell runat="server">
             Mot de passe :
        </asp:TableCell>
        <asp:TableCell runat="server">
            <asp:TextBox ID="Password_id" MaxLength="20" runat="server" />

            <asp:CompareValidator runat="server" Text="*" ControlToValidate="Password_id"
ControlToCompare="CheckPassword_id" />

        </asp:TableCell>
    </asp:TableRow>

    <asp:TableRow runat="server">
        <asp:TableCell runat="server">
             Retaper le Mot de passe :
        </asp:TableCell>
        <asp:TableCell runat="server">
            <asp:TextBox ID="CheckPassword_id" MaxLength="20" runat="server" />
        </asp:TableCell>
    </asp:TableRow>

    <asp:TableRow runat="server">
        <asp:TableCell ColumnSpan="2" runat="server">
            <asp:Button runat="server" Text="Valider" />
        </asp:TableCell>
    </asp:TableRow>
</asp:Table>
```

Voici ce que donne ce code :

Mot de passe :
Retaper le Mot de passe :
Valider

Testez-le et vous remarquerez que l'étoile n'apparait que lorsque les deux champs ne sont pas égaux. Remarquez que la propriété *Operator* n'est pas spécifié ce qui montre bien que par défaut elle est sur *Equal*. Voici un autre exemple avec un *ValueToCompare*.

ASPX

```
<%-- A Mettre dans le Tableau--%>
<asp:TableRow runat="server">
    <asp:TableCell runat="server">
        Taper "Vrai" :
    </asp:TableCell>

    <asp:TableCell runat="server">
        <asp:TextBox ID="Password_id" MaxLength="20" runat="server" />
        <asp:CompareValidator runat="server" Text="*"
ControlToValidate="Password_id" ValueToCompare="Vrai" />
    </asp:TableCell>
</asp:TableRow>
```

Cet exemple demande à ce que l'on rentre la chaine de caractère "Vrai" dans la zone de texte. Si on ne le fait pas alors il n'y aura pas validation et l'étoile qui se situe dans la propriété Text s'affichera.

Remarque :

> Dans le cas de chaine de caractères, les opérateurs *Equal* et *NotEqual*, compareront les tailles de ces chaines et non leurs similitudes.

1.3 RangeValidator

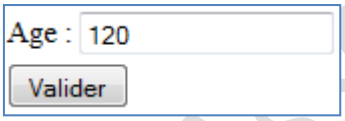
Ce contrôle de validation permet de définir deux valeurs entre lesquelles doit se trouver la valeur entrée dans le contrôle à valider. Ca peut être pratique par exemple pour un âge, un code postal … .

Les deux propriétés à connaitre pour ce contrôle sont MaximumValue et MinimumValue, respectivement la valeur maximum et la valeur minimum. Ce contrôle utilise lui aussi la propriété Type.

ASPX

```
<asp:Table runat="server">
  <asp:TableRow runat="server">
     <asp:TableCell runat="server">
         Age :
     </asp:TableCell>
     <asp:TableCell runat="server">
         <asp:TextBox ID="Age_id" MaxLength="10" runat="server" />
              <asp:RangeValidator  runat="server" Type="Integer" text="Age non
valide" ControlToValidate="Age_id" MaximumValue="120"
MinimumValue="0"></asp:RangeValidator>
         </asp:TableCell>
 </asp:TableRow>

 <asp:TableRow runat="server">
        <asp:TableCell ColumnSpan="2" runat="server">
            <asp:Button runat="server" Text="Valider" />
        </asp:TableCell>
 </asp:TableRow>
</asp:Table>
```

Voici ce que donne cet exemple :



1.4 RegularExpressionValidator

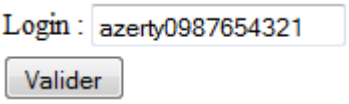
Ce contrôle permet d'utiliser des expressions régulières pour valider un champ.

Pour suivre cette partie vous aurez besoin de connaitre les bases des expressions régulières. Pour cela vous pouvez visiter http://msdn.microsoft.com/fr-fr/library/ms972966.aspx ou encore http://msdn.microsoft.com/fr-fr/library/2k3te2cs(VS.80).aspx.

Exemple d'utilisation d'expression régulière :

ASPX

```
<asp:Table runat="server">
     <asp:TableRow runat="server">
         <asp:TableCell runat="server">
              Login :
         </asp:TableCell>
         <asp:TableCell runat="server">
              <asp:TextBox ID="Login_id" MaxLength="20" runat="server" />
              <asp:RegularExpressionValidator runat="server" Text="*"
ControlToValidate="Login_id" ValidationExpression="^[a-zA-Z0-9]{6,20}" />
         </asp:TableCell>
      </asp:TableRow>

       <asp:TableRow runat="server">
         <asp:TableCell ColumnSpan="2" runat="server">
             <asp:Button runat="server" Text="Valider" />
         </asp:TableCell>
 </asp:TableRow>
</asp:Table>
```



On remarque que la *TextBox* ne permet que l'entrée de 20 caractères au maximum grâce à sa propriété *MaxLength*. Ensuite notre expression régulière vérifie, avec les nombres entres accolades, qu'il y a bien de 6 à 20 caractères. Remarquez que si on ne met aucun caractère, il y a quand même validation. Une solution serait de mettre un *RequiredFieldValidator* pour empêcher qu'il soit vide.

1.5 CustomValidator

Le *CustomValidator* va nous permettre de créer notre propre validateur. Il y a deux façons de le faire : avec du *javascript* ou avec du *CodeBehind*. Si on utilise du *JavaScript* dans la page ASPX il va falloir utiliser la propriété *ClientValidationFunction* du *CustomValidator*. Avec le *CodeBehind* on utilise l'évènement *OnServerValidate* pour mettre le nom de la fonction à utiliser. La méthode va recevoir deux arguments, la première est la source, la seconde est de type *ServerValidateEventArgs*. C'est avec l'accesseur *IsValid* de ce dernier que l'on va pouvoir définir si les conditions sont rempli ou non. On peut d'ailleurs utiliser cette propriété dans les autres contrôles depuis le code behind sur l'objet Page. On pourra définir, par exemple, lors d'un clic sur un bouton (donc avec un *PostBack*) si on veut effectuer certaines action. Pour cela, on l'utilise souvent sur l'évènement *onClick* d'un bouton avec un *if* (ce qui permet de dire que l'on effectuera les actions du *if* que si la page est valide ou si elle ne l'est pas ; ceci dépendant de l'expression que l'on met). Dans le cas du *CustomControl* on peut avoir un problème lié à l'évènement *onClick* du bouton qui valide le formulaire. Nous verrons à la fin de cette partie comme régler ce problème avec l'accesseur *IsValid*.

Une chose importante à avoir à l'esprit est que le *JavaScript* s'exécutera du coté client alors que le *CodeBehind*, lui, ne sera exécuté que lors du *PostBack* de la page. Une fois que l'utilisateur aura validé, la page sera envoyée au serveur qui vérifiera les données. Ensuite le serveur validera notre formulaire ou bien retournera des messages d'erreur.

ASPX

```
<asp:TextBox runat="server" />
<asp:CustomValidator ID="CustValidator" ControlToValidate="TextBox1"
onservervalidate="Verif_Pseudo" runat="server"
ErrorMessage="CustomValidator"></asp:CustomValidator>
<asp:Button ID="Button1" runat="server" Text="Button" />
```

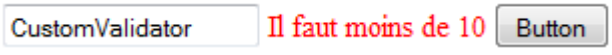
C#

```
protected void Verif_Pseudo(Object source, ServerValidateEventArgs args)
// Méthode de vérification du pseudo
{
    args.IsValid = false;// Par défaut le champ ne passe pas la validation

    if (args.Value.Length <= 3)
    // Si la taille de la valeur du champ est inférieure ou égale à trois
    CustValidator.Text = "Il faut plus de 3";
    // Alors on change le message de validation grâce à l'id du CustolValidator

    else if (args.Value.Length >= 10)
    // Si la taille de la valeur du champ est supérieure ou égale à dix
    CustValidator.Text = "Il faut moins de 10";
    // On change aussi le message

      else if (args.Value.Length > 3 && args.Value.Length < 10)
      // Si la taille de la valeur est (strictement) comprise entre 3 et 10
    args.IsValid = true;
    // Alors le champ passe la validation
}
```

VB.NET

```
Protected Sub Verif_Pseudo(ByVal source As Object, ByVal args As
ServerValidateEventArgs)
' Méthode de vérification du pseudo
      args.IsValid = False ' Par défaut le champ ne passe pas la validation


      If args.Value.Length <= 3 Then
      'Si la taille de la valeur du champ est inférieure ou égale à trois
            CustValidator.Text = "Il faut plus de 3"
            ' Alors on change le message de validation grâce à l'id du
CustolValidator

      ElseIf args.Value.Length >= 10 Then
      'Si la taille de la valeur du champ est supérieure ou égale à dix
            CustValidator.Text = "Il faut moins de 10"
            'On change aussi le message

      ElseIf args.Value.Length > 3 And args.Value.Length < 10 Then
      ' Si la taille de la valeur est (strictement) comprise entre 3 et 10
            args.IsValid = True
            ' Alors le champ passe la validation
      End If
End Sub
```

L'image ci-contre montre le résultat si on rentre un texte plus long ou égal à 10 caractères.



Remarquez que le message de validation à bien changé. Une autre chose à remarquer est que si on ne met rien dans le champ il passe quand même la validation malgré le *if* qui inclut pourtant les tailles inférieure à 3.

Maintenant que nous avons vu la création d'un *CustomControl*, nous allons voir comment gérer l'évènement *onClick* du bouton. Tout d'abord voyons un exemple du problème que nous évoquons.

Nous avons repris le *CustomValidator* que l'on vient de faire. Sur l'évènement *onClick* du bouton nous mettons du code qui doit s'exécuter après un clic sur le bouton et donc quand le formulaire est valide. Voyons un peu le code que l'on ajoute au code behind de la page ASPX (n'oubliez pas d'ajouter l'évènement *onClick* avec la méthode *Button_Click* comme valeur) :

C# - Code de la page ASPX

```
protected void Button_Click(object sender, EventArgs e)
{
    Button1.Text = TextBox1.Text;
}
```

VB.NET – Code de la page ASPX

```
Protected Sub Button_Click(ByVal sender As Object, ByVal e As EventArgs)
    Button1.Text = TextBox1.Text
End Sub
```

Si vous avez testé ce code, vous avez du remarquer que même si le formulaire n'est pas valide (aucune donnée, moins de 3 caractères, plus de 3 caractères), le code est quand même exécuté. Dans notre cas cela n'est pas bien grave, mais il y a certaines situations où cela entrainera des problèmes plus ennuyeux.

Il est donc important avant d'exécuter du code avec l'évènement *onClick* de vérifier si le formulaire est bien valide. Pour cela on peut aussi vérifier que la page elle-même est valide avec *Page.IsValid*. Comme nous l'avions vu *IsValid* va retourner un booléen qui va nous permettre, dans ce cas, de savoir si la page est valide ou non. Voici le code une fois corrigé :

C#

```
protected void Button_Click(object sender, EventArgs e)
{
    if (!Page.IsValid)
        return;

    Button1.Text = TextBox1.Text;
}
```

VB.NET

```
Protected Sub Button_Click(ByVal sender As Object, ByVal e As EventArgs)
    If Not Page.IsValid Then
        Return
    End If

    Button1.Text = TextBox1.Text
End Sub
```

Maintenant que la correction a été faite vous remarquerez que si le formulaire n'est pas valide est que l'on clic sur le bouton, le code ne sera pas exécuté. Plus précisément il est exécuté mais comme la page n'est pas valide il rentre dans la condition *if* et exécute le *return*.

1.6 ValidatonSummary

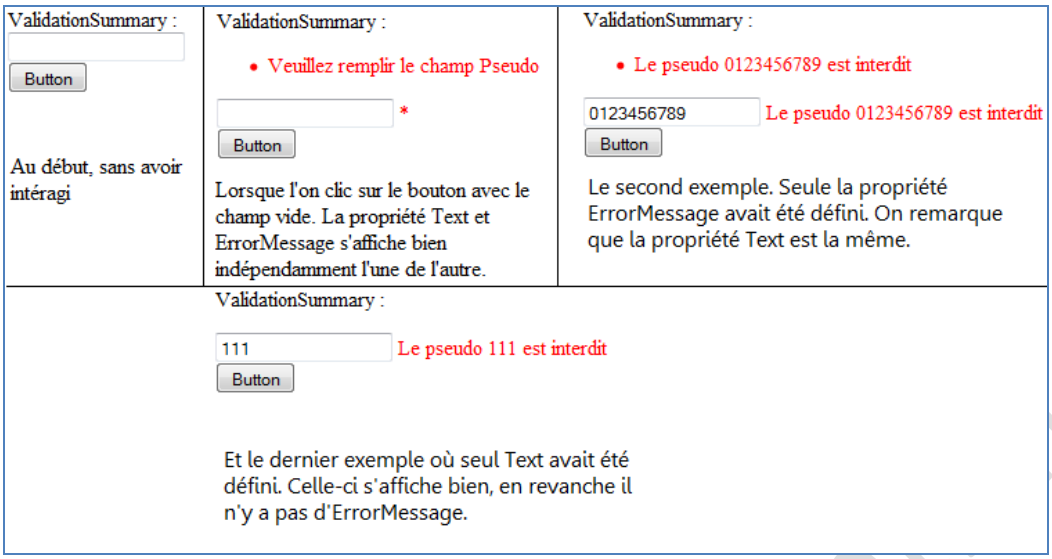
Le contrôle *ValidationSummary* est un contrôle qui va vous permettre d'afficher un sommaire, un résumé, de tous ce qui n'a pas été validé. Le texte qui y sera affiché sera celui de la propriété *ErrorMessage* que nous n'avions pas utilisé jusque là. Il faut savoir que si on ne spécifie pas la propriété *Text*, elle prendra la valeur de la propriété *ErrorMessage*. En revanche si on ne défini que la propriété *Text*, la propriété *ErrorMessage* sera vide et ne prendra pas *Text* comme message par défaut. Ce contrôle de validation va afficher la propriété *ErrorMessage* de chaque contrôle de validation qui ne sera pas validé (du moins du moment qu'il a une définition de la propriété *ErrorMessage* comme nous allons le voir dans l'exemple qui suit).

Voici un petit exemple avec les propriétés *Text* et *ErrorMessage* pour bien comprendre leurs différences :

ASPX

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

    ValidationSummary :<br />
    <asp:ValidationSummary runat="server" />
    <%--On ajoute le contrôle Validationsummary--%>
    <asp:TextBox ID="TextBox1" runat="server" />


    <asp:RequiredFieldValidator runat="server" Display="Dynamic"
       ControlToValidate="TextBox1" Text="*" ErrorMessage="Veuillez remplir le champ
       Pseudo" />
    <%--Sur le premier on voit que les propriétés Text et ErrorMessage sont défini.
        ErrorMessage va servir au remplissage du ValidationSummary et Text au
        message qui va apparaitre à côté du champ--%>

    <asp:RequiredFieldValidator runat="server" Display="Dynamic"
       InitialValue="0123456789" ControlToValidate="TextBox1" ErrorMessage="Le
       pseudo 0123456789 est interdit" />
    <%--Sur celui-ci, seul la propriété ErrorMessage est défini. On remarque à
        l'affichage que Text et ErrorMessage sont pareils.--%>

    <asp:RequiredFieldValidator runat="server" Display="Dynamic" InitialValue="111"
       ControlToValidate="TextBox1" Text="Le pseudo 111 est interdit" />
    <%--Sur celui-ci on a pas mis de propriété ErrorMessage. On remarque à
        l'affichage qu'il n'y a rien dans le summary lors de l'affichage du
        message.--%>


    <br /><asp:Button runat="server" Text="Button" />
</form>
```

Et voici le résultat suivant les différents cas :



1.7 ValidationGroup

ValidationGroup est une propriété que possèdent les contrôles de validation ainsi que le contrôle *Button*. Il va vous permettre de créer un groupe de validation. C'est-à-dire que lorsque vous allez appuyer sur le bouton, seul le groupe spécifié dans *ValidationGroup* du bouton sera validé/ vérifié. Pour créer le groupe de validation il va falloir ajouter cette propriété à chacun des contrôles que vous souhaitez mettre dans le groupe. Comme il a été dit, le bouton doit posséder cette propriété lui aussi.

Cela peut vous permettre par exemple de ne valider que ce que vous souhaitez suivant le groupe auquel appartient un utilisateur. Il suffit de créer le groupe de validation depuis le *CodeBehind* suivant le groupe de la personne en n'oubliant pas de mettre le bouton dans le même groupe.

1.8 Propriété CausesValidation

La propriété *CausesValidation* va permettre de définir si les validations seront effectuées pour ce contrôle lors d'un *PostBack*. En effet si dans votre formulaire vous avez un bouton Valider et un bouton Quitter, vous ne pourrez pas utiliser le bouton Quitter avant que tous les champs soient correctement rempli. Pour bien comprendre voir l'exemple ci-dessous :

ASPX

```
<form id="form1" runat="server">
<div>
   <asp:Label runat="server" Text="Nom" />
   <asp:TextBox ID="Nom" runat="server" />
   <asp:RequiredFieldValidator runat="server" ErrorMessage="Vous devez remplir ce
champ" ControlToValidate="Nom"/><br />

   <asp:Label runat="server" Text="Prenom" />
   <asp:TextBox ID="Prenom" runat="server" />
   <asp:RequiredFieldValidator runat="server" ErrorMessage="Vous devez remplir ce
champ" ControlToValidate="Prenom"/><br />

   <asp:Button runat="server" Text="Valider" />
   <asp:Button runat="server" Text="Quitter" />
</div>
</form>
```

Si vous essayez ce code vous remarquerez alors qu'on ne peut pas utiliser le bouton Quitter tant que les deux *TextBox* ne sont pas rempli. Pour contourner ce problème, la propriété *CausesValidation* va nous permettre de dire que le bouton Quitter ne doit pas effectuer de validation. Voici le bouton Quitter une fois la modification effectuée :

```
<asp:Button runat="server" Text="Quitter" CausesValidation="false" />
```

Maintenant que nous avons désactivé la validation sur le bouton Quitter, vous pourrez l'utiliser alors que le formulaire n'est pas rempli (cet exemple aurait pu aussi être avec un bouton Reset, Vérifier la disponibilité du pseudo …).

L'utilisation de cette propriété ne se limite pas aux boutons. Cette propriété se trouve aussi sur les *TextBox,* les *RadioButton* …. Cela peut être utile par exemple si vous définissez un *PostBack* sur une *TextBox* dès qu'il y a un changement de texte (*OnTextChanged*). Ainsi la validation ne sera pas effectuée si le texte de la *TextBox* change.

2 La Navigation dans un Site Web

Dans cette partie nous allons aborder les principes de navigations entre les pages Web de votre Site. Nous verrons comment créer des liens entre vos différentes pages Web, comment récupérer des données d'un formulaire depuis une autre page Web, ou encore améliorer la navigation de votre Site Web en créant un menu.

2.1 Les différentes méthodes de navigation

Voici les différentes méthodes que nous allons expliciter dans cette sous-partie :

Fournir du code qui s'exécutera sur le navigateur côté client.	Votre code demande une nouvelle page du Web en réponse à un événement du côté client, comme un clic de bouton.

Transfert de donnée par *PostBack*	Un control est configuré pour exécuter un *PostBack* vers une page Web différente.
Faire suivre une redirection par le navigateur côté client	Le code côté serveur envoie un message vers le navigateur, informant ce dernier de se rediriger vers une page Web différente.
Faire suivre un transfert côté serveur	Le code côté serveur exécute une page Web différente dans celle d'origine.

2.1.1 Fournir du code qui s'exécutera sur le navigateur côté client

L'une des façons les plus simples et les plus évidentes pour naviguer entre les pages Web est de créer des *HyperLinks* (Hyperliens).

En ASP.NET il existe deux manières de créer des *HyperLinks*, soit par les outils du langage ASP soit par ceux du langage HTML.

En ASP.NET, il suffit de créer un control de type *HyperLink* et de gérer ses propriétés.

ASPX

```
<asp:HyperLink ID="HyperLink1" runat="server"
NavigateUrl="~/NavigateTest2.aspx">
<%-- "~/" permet de n'écrire que le chemin relatif de votre page--%>
Vers NavigateTest2
<%-- Ci dessus le texte qui sera lié à l'hyperlien--%>
</asp:HyperLink>
```

```
NavigateUrl="~/NavigateTest2.aspx"
```

On peut remarquer que c'est la propriété ci-dessus qui gère la redirection vers la nouvelle page.

En HTML, c'est la balise *<a>* et une de ses propriétés, *href,* qui va nous permettre de changer de page.

HTML

```
<a id="HyperLink1"
href="NavigateTest2.aspx">
<%-- On peut remarquer que le chemin est directement relatif en HTML --%>
Vers NavigateTest2
</a>
```

Sinon, on peut tout aussi bien créer un *Button* qui, grâce à la méthode *onclick,* fera appel à une fonction JavaScript qui mettra à jour notre URL.

JavaScript

```
<script language="javascript" type="text/javascript">
        function updateURL() {
            document.location="NavigateTest2.aspx";
        }
</script>
```

ASPX

```
<input type="button" value="Valider" onclick="return updateURL()" />
```

Remarque :

> Il faut toutefois noter que pour avoir accès à notre fonction JavaScript nous avons du utiliser un *Button* de type HTML. En effet le control ASPX *Button* ne peut pas récupérer l'évènement *OnClick* qui est déjà utilisé par défaut. Si on veut utiliser un *Button* ASPX il faut ajouter l'évènement dans le *CodeBehind* de la page Web.

2.1.2 Transfert de données par *PostBack*

Le transfert de données par *PostBack* est très utilisé dans le cas où l'on veut récupérer des informations sur une autre page et les afficher par exemple. Lorsque que l'on utilise un *Button* (de type ASPX), on a alors accès à une propriété nommé *PostBackUrl* qui permet de définir vers quelle page vont être envoyé les informations que l'on souhaite conserver.

Sur la page cible, on a la propriété *PreviousPage* venant de l'objet *Page* qui nous permet de nous assurer qu'il existe bien une page précédente qui nous a envoyé des données. Dans le cas contraire *PreviousPage* contient la valeur *Null*. Une fois assuré que *PreviousPage* est différent de *Null* on peut alors récupérer les informations contenues dans les *Control* de notre page précédente grâce à la méthode *FindControl(*IdDuControlCible). L'exemple qui va suivre met justement en pratique cette explication. Dans un premier temps, nous allons créer deux *TextBox* : Nom et Prenom. Puis un *Button* avec la propriété *PostBackUrl.*

ASPX de la Page Default.aspx

```
<form id="form1" runat="server">
    <div>
        <h1>
            Formulaire</h1>
        <asp:ValidationSummary ID="ValidationSummary1" runat="server" />
        <table style="width: auto;">
            <tr>
                <td>
                    Nom
                </td>
                <td>
                    <asp:TextBox ID="Nom" runat="server"
                    Width="175px"></asp:TextBox>
                </td>
            </tr>
            <tr>
                <td>
                    Prénom
                </td>
                <td>
                    <asp:TextBox ID="Prenom" runat="server"
                    Width="175px"></asp:TextBox>
                </td>
            </tr>
        </table>
        <br />
        <asp:Button ID="Send" runat="server" Text="Envoyer"
         PostBackUrl="~/Page1.aspx" />
        <br />
    </div>
    </form>
```

Nous envoyons nos informations vers la page *Page1.aspx*. Regardons son code.

ASPX de la Page Page1.aspx

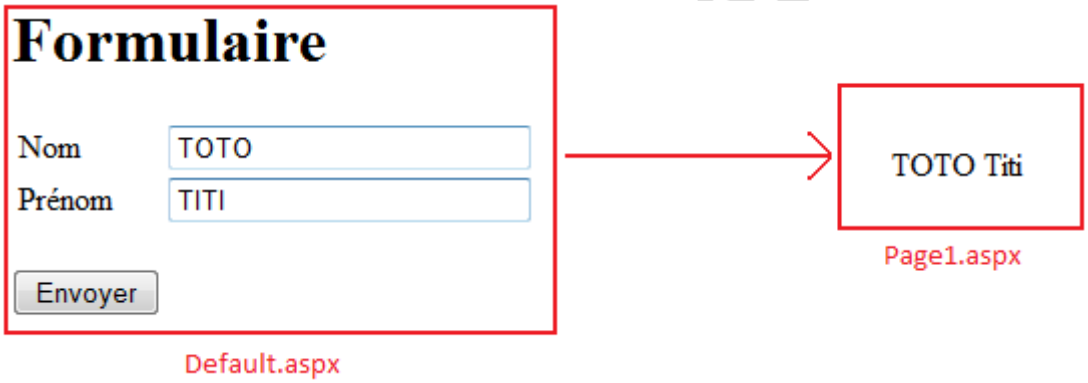
```
<form id="form1" runat="server">
    <div>
        <asp:Label ID="RecupData" runat="server" Text="Label"></asp:Label>
        <%-- Le label nous permetra d'afficher nos informations --%>
    </div>
    </form>
```

C# de la page Page1.aspx.cs

```
protected void Page_Load(object sender, EventArgs e)
        {
            if (PreviousPage == null)//Si il n'y a pas de page précédente
            {
                RecupData.Text = "Rien à afficher";//On met un texte par
                                                   défaut
            }
            else//Sinon on ajoute tout dans notre Label "RecupData"
            {
                RecupData.Text =
                    ((TextBox)PreviousPage.FindControl("Nom")).Text;
                RecupData.Text += " ";
                RecupData.Text +=
                    ((TextBox)PreviousPage.FindControl("Prenom")).Text;
            }
        }
```

```
VB.NET

Protected Sub Page_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load

    If (PreviousPage Is Nothing) Then ' Si il n'y a pas de page précédente
        RecupData.Text = "Rien à afficher" ' On met un texte par défaut

    Else ' Sinon on ajoute tout dans notre Label "RecupData"

        RecupData.Text = CType(PreviousPage.FindControl("Nom"),
TextBox).Text
        RecupData.Text += " "
        RecupData.Text += CType(PreviousPage.FindControl("Prenom"),
TextBox).Text
    End If

End Sub
```

Voici un aperçu du résultat que vous devriez avoir :



2.1.3 Faire suivre une redirection par le navigateur côté client

Il est possible d'effectuer une redirection vers une page en utilisant la méthode *Redirect* de l'objet *Response*. Ainsi on peut gérer une redirection grâce à la gestion d'un évènement sur un *Button* par exemple.

ASPX de la Page Default.aspx

```
    <form id="form1" runat="server">
    <div>
        <br />
        <asp:Button ID="Send" runat="server" Text="Envoyer"
         OnClick="Send_Click" />
        <br />
    </div>
    </form>
```

C# de la Page Default.aspx

```
        protected void Send_Click(object sender, EventArgs e)
        {
            Response.BufferOutput = true;
            Response.Redirect("Page1.aspx");
        }
```

VB.NET

```
Protected Sub Send_Click(ByVal sender As Object, ByVal e As
System.EventArgs)
    Response.BufferOutput = True
    Response.Redirect("Page1.aspx")
End Sub
```

On peut remarquer que l'on a défini la propriété *BufferOutput* à *true*. Cette initialisation est indispensable car elle empêche l'envoi de données avant la redirection. Dans le cas contraire, une *HttpException* est levée vous indiquant que vous ne pouvez pas effectuer de redirection après que les entêtes HTTP soient envoyés.

Ce code revient à rappeler la même page puis à effectuer notre redirection avant que les entêtes HTTP ne soient envoyés, en d'autres termes, on effectue un aller-retour sur le serveur.

Par ailleurs il faut noter que cette méthode rend impossible l'envoie de données tel qu'on a pu le voir auparavant. Du fait de cet aller-retour, il n'y a plus aucune donnée stockée. Il existe cependant d'autres méthodes pour envoyer nos données tels que les *cookies,* les variables sessions ou encore le passage de données par un *QueryString.*

2.1.4 Faire suivre un transfert côté serveur

Nous allons aborder l'instance *Server* qui est issue de la classe *HttpUtility* stocké dans l'objet *Page*, qui possède une méthode relativement intéressante : la méthode *Transfert.*

Cette méthode permet de transférer une page Web sur la page d'origine. Ainsi, bien que le contenu de la page change, l'URL reste identique. Les méthodes de l'objet *Page* vu plus haut peuvent cette fois-ci être utilisées (*PreviousPage)* ainsi que *PostBack*.

Pour illustrer cette méthode nous reprendrons l'exemple de la partie 2.1.2 : « Transfert de données par *Postback* ». Nous remplacerons `PostBackUrl="~/Page1.aspx"` par l'évènement `OnClick="Send_Click"`. Nous allons ensuite gérer cet évènement dans le *CodeBehind*.

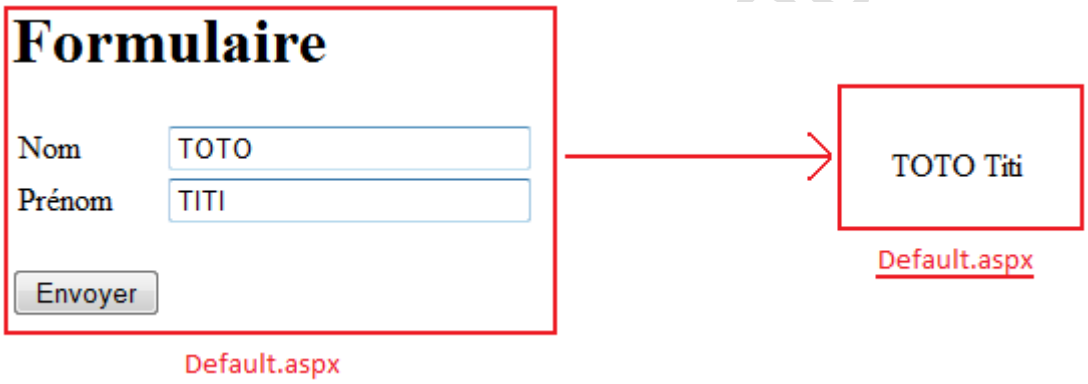
C# de la Page Default.aspx

```
protected void Send_Click(object sender, EventArgs e)
{
    Server.Transfer("Page1.aspx", false);
}
```

VB.NET de la page Default.apsx

```
Protected Sub Send_Click(ByVal sender As Object, ByVal e As
System.EventArgs)
    Server.Transfer("Page1.aspx", false)
End Sub
```

Comme on peut le constater, la méthode *Transfert* accepte un Booléen en paramètre appelé *preserveForm* qui permet d'indiquer si vous souhaitez conserver le formulaire et les données *QueryString*. Il est généralement conseillé de laisser cette valeur à *false*.



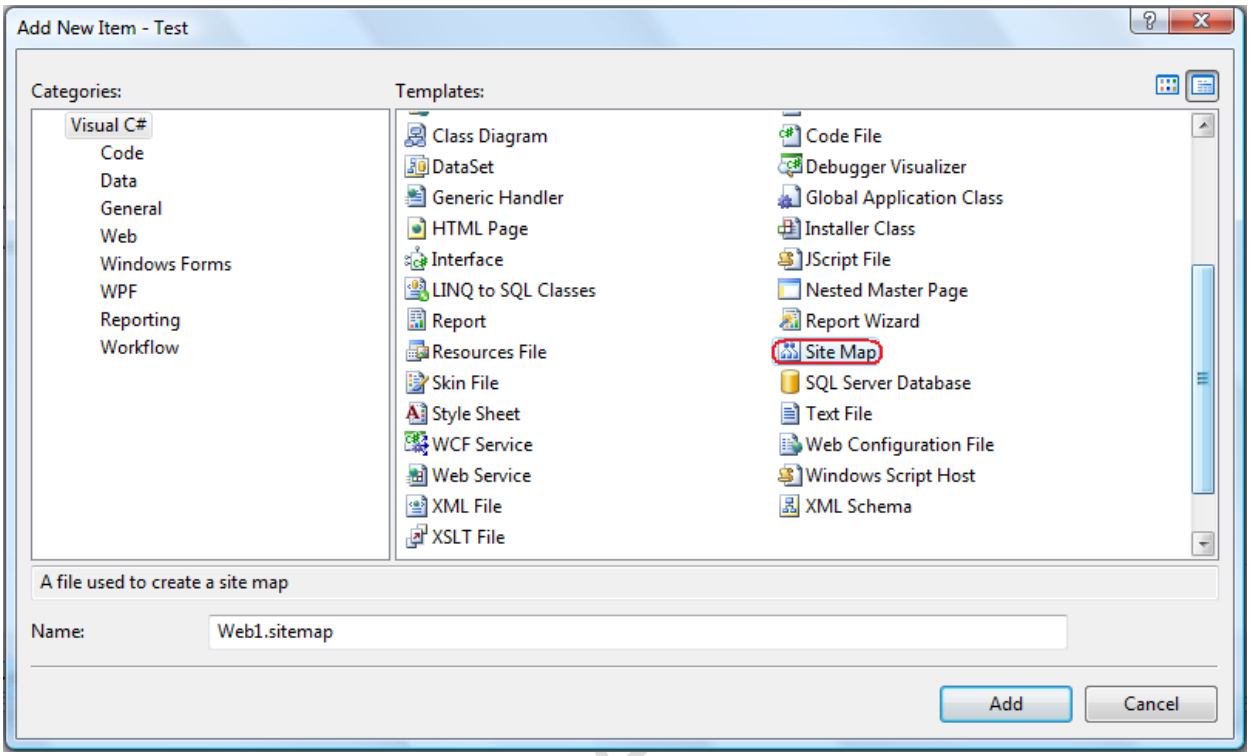
2.2 Utilisation des Site Map Web Server Control

Nous allons voir les différentes méthodes qui permettent une navigation plus intuitive et qui rendra toute nos pages Web facilement accessible.

Menu	Montre la structure du site et permet à l'utilisateur de choisir une page vers où il veut naviguer.
TreeView	Montre la structure du site dans une arborescence.
SiteMapPath	Affiche le chemin de notre page comme si nous naviguions dans un répertoire sous Windows. Ex : *DotNet > Cours > Chapitre1*

Afin d'utiliser chacun des Controls ci-dessus il faut et il suffit de les lier à une base de données. Il existe cela dit un élément dans Visual Studio adapté à la gestion de vos pages Web : le Site Map.

L'élément Site Map qui n'est ni plus ni moins qu'un fichier XML tout prêt que l'on va pouvoir lier tout comme une base de données à nos Controls ci-dessus. La création d'un Site Map se fait simplement par le menu dans l'explorateur de solution via l'ajout de nouvel élément.



Et voici ce que contient de base notre Site Map :

XML

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

Il suffit de le compléter de façon adéquate pour pouvoir le lier à un de nos Controls.

On notera simplement que :

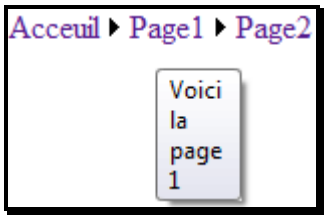
- *url* : URL de votre fichier ASPX
- *title* : Titre qui sera affiché dans votre menu
- *description* : Une description qui apparaitre lors du passage du pointeur de la souris.

Voici le code à ajouter pour lier notre data source :

```
<asp:SiteMapDataSource ID="MySiteMap" runat="server" />
```

Voici un aperçu de nos éléments :

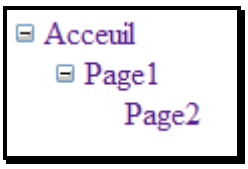
Menu



CODE :

```
<asp:Menu
ID="Menu1"
runat="server"
DataSourceID=
"MySiteMap">
</asp:Menu>
```

TreeView



CODE :

```
<asp:TreeView
ID="TreeView1"
runat="server"
DataSourceID=
"MySiteMap">
</asp:TreeView>
```

SiteMapPath

CODE :

```
<asp:SiteMapPath
ID="SiteMapPath1"
runat="server">
</asp:SiteMapPath>
```

Remarque :

Etant donné que nous avons affaire à un fichier XML, il est bien entendu possible d'en utiliser toutes les propriétés dans le *CodeBehind.* Ainsi on peut parfaitement créer des redirections en se servant des propriétés XML dans la gestion d'un évènement d'un *Button* par exemple.

3 Conclusion

Les contrôles de validation, comme nous l'avons vu, permettent de couvrir tous les cas possible dans lesquels on aura besoin de valider des données entrées par l'utilisateur. En effet même si les contrôles de validation existant ne suffisent pas, vous pouvez créer le votre selon votre envie. Pour cela deux moyens : utiliser du code côté client avec du Javascript ou encore utiliser du code côté serveur avec le CodeBehind. Le code côté client effectuera la validation avant l'envoi des données alors que le code côté serveur validera les données reçues par le serveur et, le cas échéant, enverra à nouveau la demande de saisie des données avec le(s) message(s) d'erreur. Par conséquent, le code côté serveur sera plus sécurisé que celui sur le client et cela pour le simple fait qu'un pirate pourrait facilement contourner la validation du côté client et ainsi envoyer des données incorrecte voir dangereuse pour le serveur. Il est plus difficile de contourner la vérification côté serveur.

Il est important, pour la sécurité d'un site Web, de vérifier les données envoyées par les utilisateurs. Mais il est aussi important d'avoir un site bien organisé. L'organisation d'un site Web, mais aussi son accessibilité ainsi que sa clarté, passe bien souvent par une navigation bien faite, facilement compréhensible et accessible à tous. Pour cela, ASP.NET implémente des contrôles permettant la gestion de la navigation. Il faut se rappeler que pour la navigation en ASP.NET on a un fichier XML en *.sitemap* avec une grammaire bien définie (pour qu'ASP.NET puisse le parser automatiquement). Dans ce fichier on va référencer toute l'arborescence de notre site. A l'aide d'un contrôle SiteMapDataSource, il va pouvoir être utilisé par un des trois contrôles de navigation : *Menu*, *TreeView* ou encore *SiteMapPath*.