Tutoriel : Comparaison de méthodes de communications AJAX avec ASP.NET

par Nico-pyright(c) (Page d'accueil)

Date de publication : 06/05/2009

Dernière mise à jour : 06/05/2009

Cet article présente plusieurs méthodes de rafraichissement partiel de page en AJAX pour un site web ASP.NET.
Commentez cet article :

1.Introduction

Vous êtes un développeur C#, vous travaillez avec ASP.NET et vous souhaitez pouvoir gérer le rafraichissement partiel de page ?
Alors ce tutoriel est pour vous.

A travers cet article, nous allons présenter différentes méthodes pour communiquer depuis la page web développée en ASP.NET avec une autre page, un web service ... afin de permettre le rafraichissement partiel de page et améliorer l'expérience utilisateur.
Tout au long de ce cours, je vais utiliser Visual C# 2008 et ASP.NET.

2.Le fil rouge

Nous allons étudier ces différentes méthodes à travers un petit développement.
Le but est de développer une page qui va permettre de vérifier qu'un pseudo est disponible ; un peu comme sur une page d'inscription à un forum.
Créons à cet effet une application Web *(Nouveau projet => ASP.NET Web Application)*.

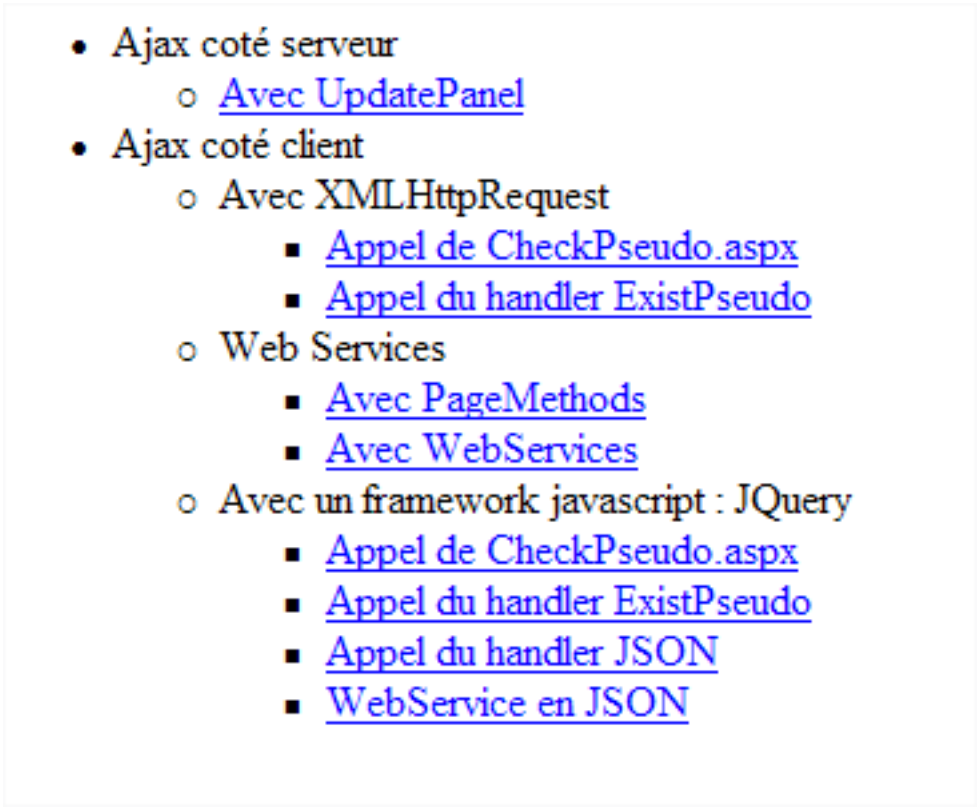
Bien sur, dans une architecture classique, pour vérifier qu'un pseudo existe, on ferait appel à un service (situé dans un projet à part), qui irait requêter dans une base de données à l'aide d'une DAL...
Ici, on va simuler ce fonctionnement avec une classe statique, grâce à un test en dur (et une pause, pour faire durer le suspens ...) incluse directement dans le projet.

Ajoutons une classe à notre projet : **UserService.cs** :

UserService.cs

```
public static class UserService
{
 public static bool PseudoLibre(string pseudo)
 {
  Thread.Sleep(3000);
  return string.Compare(pseudo, "nico", StringComparison.InvariantCultureIgnoreCase) == 0;
 }
}
```

Très simple, on compare si le pseudo est égal à Nico et dans ce cas, on retourne vrai ; faux sinon.
Pour illustrer tous nos cas d'exemples, on va modifier la page Default.aspx pour qu'elle nous permette d'accéder aux différentes pages exemple, comme ci-dessous.



Le code de la page sera :

Default.aspx

```
<ul>
  <li>AJAX coté serveur
    <ul>
      <li>
        <asp:HyperLink ID="HyperLink1" runat="server" NavigateUrl="~/Pages/PageUpdatePanel.aspx"
          Text="Avec UpdatePanel" />
      </li>
    </ul>
  </li>
  <li>AJAX coté client
    <ul>
        <li>Avec XMLHttpRequest
          <ul>
            <li>
              <asp:HyperLink ID="HyperLink2" runat="server"
                NavigateUrl="~/Pages/PageXMLHttpRequestPage.aspx" Text="Appel de CheckPseudo.aspx" />
            </li>
            <li>
             <asp:HyperLink ID="HyperLink3" runat="server"
                NavigateUrl="~/Pages/
PageXMLHttpRequestHandler.aspx" Text="Appel du handler ExistPseudo" />
            </li>
          </ul>
        </li>
        <li>Web Services
          <ul>
            <li>
             <asp:HyperLink ID="HyperLink4" runat="server"
                NavigateUrl="~/Pages/PagePageMethods.aspx" Text="Avec PageMethods" />
            </li>
            <li>
            <asp:HyperLink ID="HyperLink5" runat="server"
                NavigateUrl="~/Pages/PageWebService.aspx" Text="Avec WebServices" />
            </li>
          </ul>
```

Default.aspx

```
        </li>
        <li>Avec un framework javascript : jQuery
          <ul>
            <li>
           <asp:HyperLink ID="HyperLink6" runat="server"
                NavigateUrl="~/Pages/
PageXMLHttpRequestPageJQuery.aspx" Text="Appel de CheckPseudo.aspx" />
            </li>
            <li>
           <asp:HyperLink ID="HyperLink7" runat="server"
                NavigateUrl="~/Pages/
PageXMLHttpRequestHandlerJQuery.aspx" Text="Appel du handler ExistPseudo" />
            </li>
            <li>
           <asp:HyperLink ID="HyperLink9" runat="server"
                NavigateUrl="~/Pages/PageHandlerJSONJQuery.aspx" Text="Appel du handler JSON" />
            </li>
            <li>
           <asp:HyperLink ID="HyperLink8" runat="server"
                NavigateUrl="~/Pages/PageWebServiceJSONJQuery.aspx" Text="WebService en JSON" />
            </li>
          </ul>
        </li>
      </ul>
    </li>
</ul>
```

2.A la sauce ASP.NET classique

Dans le cadre d'une application ASP.NET, on aurait construit une page qui contiendrait un **Textbox** et un bouton associé à un handler sur le click.
Et pour faire joli, on aurait rajouté une image pour dire que le pseudo est libre, et une autre image pour dire que le pseudo est déjà pris.
Ce qui donne une page de ce genre :

Page.aspx

```
<asp:Label Text="Entrer le pseudo" runat="server" />
<asp:TextBox runat="server" ID="Login" />
<asp:Button Text="Vérifier" OnClick="VerifPseudo" runat="server" />
<asp:Image ID="OkImg" runat="server" ImageUrl="~/Images/ok.png" Visible="false" />
<asp:Image ID="ErrImg" runat="server" ImageUrl="~/Images/err.png" Visible="false" />
```

Le code behind associé sera :

Page.aspx.cs

```
protected void VerifPseudo(object sender, EventArgs e)
{
 if (UserService.PseudoLibre(Login.Text))
 {
  OkImg.Visible = true;
  ErrImg.Visible = false;
 }
 else
 {
  OkImg.Visible = false;
  ErrImg.Visible = true;
 }
}
```

Rien de bien compliqué. Ce qui nous donne :

Nous allons nous attacher désormais à rendre notre superbe page un peu plus AJAX friendly...
Note : dans toutes les pages que nous allons découvrir et créer, on rajoutera un contrôle Label témoin indiquant l'heure courante au chargement de la page.
Ceci nous permettra de montrer qu'on a bien eu un rendu partiel et que la page ne s'est pas rechargée totalement.
Le code behind correspondant dans chaque page sera :

Page.aspx.cs

```
protected override void OnInit(EventArgs e)
{
 LabelTemoin.Text = DateTime.Now.ToLongTimeString();
 base.OnInit(e);
}
```

Je n'en parlerai plus désormais et ferais comme si il était inclus dans chaque page.

3.Utilisation d'AJAX

3.1.Rendu partiel en utilisant le contrôle UpdatePanel

On utilise le contrôle **UpdatePanel** et son inséparable **ScriptManager** qui vont nous permettre de générer le code client javascript pour gérer le rendu partiel.
Pour ce faire, pas besoin d'énormes transformations de la page. On rajoute un contrôle ScriptManager et on encapsule le reste dans un UpdatePanel :

PageUpdatePanel.aspx

```
<asp:ScriptManager runat="server" ID="ScriptManager" />
<asp:Label runat="server" ID="LabelTemoin" />
<asp:UpdatePanel runat="server" ID="UpdatePanel1" UpdateMode="Conditional">
 <ContentTemplate>
  <asp:Label ID="Label1" Text="Entrer le pseudo" runat="server" />
  <asp:TextBox runat="server" ID="Login" />
  <asp:Button ID="Button1" Text="Vérifier" OnClick="VerifPseudo" runat="server" />
  <asp:Image ID="OkImg" runat="server" ImageUrl="~/Images/ok.png" Visible="false" />
  <asp:Image ID="ErrImg" runat="server" ImageUrl="~/Images/err.png" Visible="false" />
 </ContentTemplate>
</asp:UpdatePanel>
```

Le code behind permettant de vérifier que le pseudo est libre ne change pas, à savoir :

PageUpdatePanel.aspx.cs

```
protected void VerifPseudo(object sender, EventArgs e)
{
 if (UserService.PseudoLibre(Login.Text))
 {
  OkImg.Visible = true;
  ErrImg.Visible = false;
 }
 else
 {
  OkImg.Visible = false;
  ErrImg.Visible = true;
 }
```

PageUpdatePanel.aspx.cs
```
}
```

Et voilà, nous avons un beau rendu partiel. Nous sommes capables d'indiquer si un pseudo est libre sans avoir à recharger toute la page.
Un des avantages visible directement est qu'on a pu faire un rendu partiel sans écrire une seule ligne de javascript, tout est pris en charge par ASP.NET.

3.2.Rendu partiel en utilisant l'objet XMLHttpRequest, l'AJAX de la vieille école

L'objet **XMLHttpRequest** va nous permettre de faire un appel asynchrone (coté client) à une page (coté serveur) et de récupérer le retour de cette page.
A cet effet, toujours dans l'optique de vérifier si un pseudo est libre, on peut envisager d'appeler une page ASPX avec le pseudo en paramètre. Cette page va vérifier si le pseudo est libre, et nous renvoyer l'image correspondante. Il nous restera plus qu'à afficher cette image coté client.
Créons une page *CheckPseudo.aspx*.
On va nettoyer son contenu pour ne garder que :

CheckPseudo.aspx
```
<%@ Page Language="C#" AutoEventWireup="false" CodeBehind="CheckPseudo.aspx.cs" Inherits="CompareAjaxMethods.Page

<form id="form1" runat="server">
    <asp:Image runat="server" id="OkImg" Src="~Images/ok.png" />
    <asp:Image runat="server" id="ErrImg" Src="~Images/ok.png" />
</form>
```

J'ai décidé arbitrairement que la page serait appelée en POST, mais on peut également envisager de faire cet appel en GET. Voici le code behind de la page CheckPseudo.aspx :

CheckPseudo.aspx.cs
```
protected override void OnLoad(System.EventArgs e)
{
 string pseudo = Request.Form["pseudo"];
 if (UserService.PseudoLibre(pseudo))
 {
  OkImg.Visible = true;
  ErrImg.Visible = false;
 }
 else
 {
  OkImg.Visible = false;
  ErrImg.Visible = true;
 }
 base.OnLoad(e);
}
```

Dans le OnLoad de la page CheckPseudo.aspx, on récupère la valeur du paramètre **"pseudo"** passé en POST et on affiche l'image en fonction du retour de l'appel au service de vérification de pseudo.
Ce qui fait que la page de démonstration (PageXMLHttpRequestPage.aspx) n'aura pas besoin de contenir d'images, vu que c'est la réponse à CheckPseudo qui va nous les renvoyer. Par contre, elle aura besoin d'un container pour afficher cette réponse : par exemple un div "result".
Ce qui nous donne :

PageXMLHttpRequestPage.aspx
```
<form id="form1" runat="server">
<asp:Label runat="server" ID="LabelTemoin" /><br />
```

PageXMLHttpRequestPage.aspx
```
<asp:Label ID="Label1" Text="Entrer le pseudo" runat="server" />
<asp:TextBox runat="server" ID="Login" />
<input type="button" onclick="VerifPseudoJs('<%=Login.ClientID %>')" value="Vérifier" />
<div id="result"></div>
</form>
```

Notez qu'on passe en paramètre à la fonction VerifPseudoJs l'ID client du TextBox Login.
On inclura dans cette page deux fichiers javascript :

PageXMLHttpRequestPage.aspx
```
<script type="text/javascript" src="/js/Helper.js"></script>
<script type="text/javascript" src="/js/XmlHttpPage.js"></script>
```

Le premier contient des méthodes Helper, à savoir la récupération de l'objet XMLHttpRequest et une méthode pour changer le display d'une balise :

Helper.js
```
function getXhr()
{
    var xhr = null;
    if (window.XMLHttpRequest) // Firefox et autres
        xhr = new XMLHttpRequest();
    else if (window.ActiveXObject)
    { // Internet Explorer
        try
        {
            xhr = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e)
        {
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        }
    }
    else
    { // XMLHttpRequest non supporté par le navigateur
        alert("Votre navigateur ne supporte pas les objets XMLHTTPRequest...");
        xhr = false;
    }
    return xhr
}

function setDisplay(id, visible)
{
    var o;
    if (document.getElementById)
        o = document.getElementById(id).style;
    else if (document.layers)
        o = document.layers[id];
    else if (document.all)
        o = document.all[id].style;
    if (o)
        o.display = (visible ? 'block' : 'none');
}
```

Il restera à faire l'appel à la méthode VerifPseudoJs qui sera contenu dans le fichier XmlHttpPage.js :

XmlHttpPage.js
```
function VerifPseudoJs(idText)
{
    var xhr = getXhr();
```

XmlHttpPage.js

```
    if (xhr)
    {
        xhr.onreadystatechange = function()
        {
            // On ne fait quelque chose que si on a tout reçu et que le serveur est ok
            if (xhr.readyState == 4 && xhr.status == 200)
            {
                document.getElementById('result').innerHTML = xhr.responseText;
            }
        }
        xhr.open("POST", "CheckPseudo.aspx", true);
        xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
        var data = "pseudo=" + document.getElementById(idText).value;
        xhr.send(data);
    }
}
```

Il s'agit de récupérer l'objet **XMLHTTPRequest** et d'appeler la page CheckPseudo.aspx en POST, en lui passant le paramètre pseudo et le contenu du textbox.
Ensuite, une callback sera appelée une fois que la requete AJAX sera terminée. C'est à ce moment là qu'on remplira le div "result" avec le contenu de la réponse (utilisation de **innerHTML**).
Il est à noter que le contenu retourné, dans notre cas, est :

Réponse de CheckPseudo.aspx

```
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
 value="/wEPDwULLTExNzU3MzQyMTAPZBYCZg9kFgICAQ8PFgIeB1Zpc2libGVoZGRkX49ztZNge2YjgU7qgcf5ZKkqG3E
=" />
</div>

<img id="ErrImg" src="/Images/err.png" src="" style="border-width:0px;" />
```

et contient notamment le viewstate.
On peut constater qu'on n'a pas eu besoin d'écrire de code behind dans notre page. Par contre, il y en a dans la page CheckPseudo.aspx.
Notons également la présence du viewstate, qui dans ce cas n'est pas énorme, mais qui peut potentiellement l'être, par exemple si la page contient une grille...
L'obligation d'écrire une page pour traiter l'existence du pseudo est une contrainte non négligeable et qui peut devenir difficile à maintenir si on en augmente le nombre et la complexité.

3.3.Rendu partiel avec XMLHttpRequest et IHttpHandler

Pourquoi appeler la page CheckPseudo.aspx ?
On peut se dire que le but est de faire un appel de service et que ce n'est pas si logique que ca de devoir passer par une page ASPX pour y arriver. Une page est un système complexe, gérant son état (ViewState), passant par plusieurs étapes lors de son cycle de vie, générant un rendu, etc ... Pourquoi toute cette mécanique pour juste avoir à afficher une réponse ?
Effectivement, cette question a toute sa justification. De plus, ASP.NET nous fournit un mécanisme qui permet de nous en passer des pages dans ce cas : les handler.
Un handler est une classe qui doit implémenter **IHttpHandler**.
Cette classe doit également être déclarée dans le **web.config** avec un verbe associé.
Créons donc la classe CheckPseudoHandler.cs

CheckPseudoHandler.cs

```
public class CheckPseudoHandler : IHttpHandler
{
 public void ProcessRequest(HttpContext context)
```

CheckPseudoHandler.cs

```
  {
   context.Response.Cache.SetCacheability(HttpCacheability.NoCache);
   context.Response.ContentType = "text/plain";
   string pseudo = context.Request.Form["pseudo"];
   if (UserService.PseudoLibre(pseudo))
    context.Response.Write(true);
   else
    context.Response.Write(false);
  }

  public bool IsReusable
  {
   get { return false; }
  }
 }
```

Cette classe va faire grosso modo la même chose que la page CheckPseudo.aspx. C'est à dire lire le pseudo passé en paramètre en POST, appeler le service de vérification de pseudo et écrire la réponse.
La réponse ici en l'occurrence sera du "text/plain" et contiendra "True" ou "False" en fonction du résultat de l'appel au service.
Pour qu'ASP.NET puisse associer une requête à ce handler, il faudra le déclarer dans le web.config, à la section :

web.config

```
<system.web>
  <httpHandlers>
    <add verb="POST" path="ExistPseudo"
        type="CompareAjaxMethods.CheckPseudoHandler, CompareAjaxMethods" />
  </httpHandlers>
</system.web>
```

On indique ici qu'ASP.NET doit répondre aux requêtes POST de type "ExistPseudo" en utilisant le gestionnaire HTTP présent dans la classe CheckPseudoHandler.
La page sera donc :

PageXMLHttpRequestHandler.aspx

```
<form id="form1" runat="server">
<asp:Label runat="server" ID="LabelTemoin" /><br />
<asp:Label ID="Label1" Text="Entrer le pseudo" runat="server" />
<asp:TextBox runat="server" ID="Login" />
<input type="button" onclick="VerifPseudoJs('<%=Login.ClientID %>')" value="Vérifier" />
<div id="result"></div>
</form>
```

On aura deux scripts à inclure :

PageXMLHttpRequestHandler.aspx

```
<script type="text/javascript" src="/js/Helper.js"></script>
<script type="text/javascript" src="/js/XmlHttpPage.js"></script>
```

XmlHttpPage.js contiendra quasiment la même fonction que dans le chapitre précédent, seuls l'appel et le traitement de la réponse AJAX changent:

XmlHttpPage.js

```
function VerifPseudoJs(idText, idOk, idErr)
{
```

XmlHttpPage.js

```
    var xhr = getXhr();
    if (xhr)
    {
        xhr.onreadystatechange = function()
        {
            // On ne fait quelque chose que si on a tout reçu et que le serveur est ok
            if (xhr.readyState == 4 && xhr.status == 200)
            {
                if (xhr.responseText == 'True')
                {
                    setDisplay(idOk, true);
                    setDisplay(idErr, false);
                }
                else
                {
                    setDisplay(idOk, false);
                    setDisplay(idErr, true);
                }
            }
        }
        xhr.open("POST", "ExistPseudo", true);
        xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
        var data = "pseudo=" + document.getElementById(idText).value;
        xhr.send(data);
    }
}
```

On constate que l'appel de page se transforme en un appel à **ExistPseudo**.
Désormais, on compare la réponse à "True", simplement. On affichera les images en fonction de cette comparaison.
Pas de code behind ici non plus, par contre, il faudra écrire un handler de page. Ce qui peut être une charge non négligeable si on en a beaucoup à écrire.

3.4.Rendu partiel en utilisant les PageMethods

L'utilisation de PageMethods permet d'appeler une méthode statique "serveur" de la page, exposée comme une méthode de WebService depuis le code javascript client de la page.
Pour ce faire, il faut mettre la propriété **EnablePageMethods** du ScriptManager à **true**.
On va ainsi pouvoir appeler la méthode depuis le javascript, récupérer un résultat et faire le traitement approprié en fonction de ce résultat.
On écrira ce traitement dans un fichier javascript qu'on va référencer auprès du ScriptManager, grâce à ScriptReference :

PagePageMethods.aspx

```
<asp:ScriptManager ID="ScriptManager1" runat="server" EnablePageMethods="true">
 <Scripts>
  <asp:ScriptReference Path="~/js/PageMethod.js" />
 </Scripts>
</asp:ScriptManager>
<asp:Label runat="server" ID="LabelTemoin" /><br />
<asp:Label ID="Label1" Text="Entrer le pseudo" runat="server" />
<asp:TextBox runat="server" ID="Login" />
<input type="button" onclick="VerifPseudoJs('<%=Login.ClientID %>', '<%=OkImg.ClientID %>', '<
%=ErrImg.ClientID %>')" value="Vérifier" />
<asp:Image ID="OkImg" runat="server" ImageUrl="~/Images/ok.png" style="display:none" />
<asp:Image ID="ErrImg" runat="server" ImageUrl="~/Images/err.png" style="display:none" />
```

Comme on veut faire le traitement coté javascript, on remplace le asp:Button par le contrôle html <input type="button">, ainsi on pourra intervenir facilement sur le javascript grâce à l'événement **onclick**.
Non pas que ce ne soit pas possible de le faire avec un asp:button, mais pour ce faire, il faudrait désactiver plusieurs choses afin d'éviter le postback géré par le framework ASP.NET.

On appelle la fonction VerifPseudoJs, définie dans le fichier PageMethod.js, que l'on verra plus bas, en lui passant en paramètres l'id client du textbox et des images.
Coté code behind, on opère quelques modifications :

PagePageMethods.aspx.cs

```
[WebMethod]
public static bool VerifPseudo(string pseudo)
{
 return UserService.PseudoLibre(pseudo);
}
```

La méthode est décorée de l'attribut **[WebMethod]** qui permet d'indiquer que cette méthode est une méthode de web service.
Attention, la méthode doit ABSOLUMENT être statique pour être accessible depuis le javascript. Sans cela, cela ne fonctionnera pas et vous aurez l'erreur javascript suivante :

```
PageMethods is not defined
```

Comment cette méthode serveur peut-elle être accessible depuis le javascript ? C'est le ScriptManager qui s'occupe de toute la magie en générant une classe javascript proxy.
Ainsi, on pourra appeler la méthode VerifPseudo grâce au proxy PageMethods, de cette façon : PageMethods.VerifPseudo(...)
La signature de cette méthode est :

```
PageMethods.VerifPseudo(paramètres de la méthode serveur, callback OnSuccess, callback OnError, objet
 contexte)
```

Le code javascript de PageMethod.js sera alors :

PageMethod.js

```
function VerifPseudoJs(idText, idOk, idErr)
{
    PageMethods.VerifPseudo($get(idText).value,
        function(result, userContext, methodName)
        {
            if (result)
            {
                setDisplay(idOk, true);
                setDisplay(idErr, false);
            }
            else
            {
                setDisplay(idOk, false);
                setDisplay(idErr, true);
            }
        },
        function(error, userContext, methodName)
        {
            if (error !== null)
            {
                alert('Erreur : ' + error.get_message());
            }
        }, context);
}

if (typeof (Sys) !== "undefined") Sys.Application.notifyScriptLoaded();
```

La méthode "serveur" VerifPseudo attend un pseudo ; on va donc lui passer avec $get(idText).value pour aller chercher la valeur saisie dans le textbox.
Notez que **la fonction $get** est un raccourci défini par les extensions web pour utiliser **document.getElementById**.
En cas d'erreur, la callback OnError est appelée et on affiche l'erreur avec la fonction javascript Alert.
En cas de succès, la signature de la callback nous met à disposition un objet contenant le retour (booléen dans notre cas) de la webmethod, un objet de contexte ainsi que le nom de la méthode (on ne se servira pas du nom de la méthode dans notre cas).
Enfin, on change le style display des images en fonction du résultat.
Notez qu'on peut utiliser un objet de contexte pour faire passer des valeurs de la fonction javascript à la callback de succès. Cependant, on n'aura pas besoin de l'utiliser ici grâce aux closures et à la portée des variables à l'intérieur de celles-ci.
On peut constater qu'un peu de javascript a du être écrit, et que le ScriptManager nous a grandement simplifié la tache en générant une classe proxy.
Un des inconvénients est qu'on est cantonné à utiliser une méthode statique du codebehind de la page, ceci peut suffire dans beaucoup de cas mais impose que le code behind soit fortement couplé à la logique du service.

3.5.Rendu partiel en utilisant un web service WCF

Et si on devait vérifier la disponibilité d'un pseudo via un web service ? Ceci peut arriver fréquemment lorsqu'on utilise une architecture orientée service (SOA) ou simplement si le web service est géré par un partenaire externe.
Encore une fois, le ScriptManager nous fait une grande partie du travail.
Commençons par créer un web service.
Dans notre projet, ajoutons un nouvel élément (add new item -> Ajax-enabled WCF Service) : WCFUserService.svc
Visual studio nous génère une partie du code, notamment les attributs de la classe, ainsi que la configuration adéquate dans le web.config. (Note : si vous devez utiliser un web service existant, il faudra le configurer de manière à ce qu'il soit appelable par ASP.NET, voir **ce lien**.)
La classe du webservice sera :

WCFUserService.svc.cs

```
[ServiceContract(Namespace = "")]
[AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
public class WCFUserService
{
 [OperationContract]
 public bool VerifPseudo(string pseudo)
 {
  return UserService.PseudoLibre(pseudo);
 }
}
```

Le code de la page ASPX est sensiblement le même que pour les PageMethods, à part que l'on rajoute le tag ServiceReference dans le ScriptManager en précisant l'url du web service que l'on souhaite appeler.

PageWebService.aspx

```
<asp:ScriptManager runat="server" ID="ScriptManager">
 <Services>
  <asp:ServiceReference Path="~/WCFUserService.svc" />
 </Services>
 <Scripts>
  <asp:ScriptReference Path="~/js/WCF.js" />
 </Scripts>
</asp:ScriptManager>
<asp:Label runat="server" ID="LabelTemoin" /><br />
<asp:Label ID="Label1" Text="Entrer le pseudo" runat="server" />
<asp:TextBox runat="server" ID="Login" />
<input type="button" onclick="VerifPseudoJs('<%=Login.ClientID %>',
            '<%=OkImg.ClientID %>', '<%=ErrImg.ClientID %>')" value="Vérifier" />
<asp:Image ID="OkImg" runat="server" ImageUrl="~/Images/ok.png" Style="display: none" />
<asp:Image ID="ErrImg" runat="server" ImageUrl="~/Images/err.png" Style="display: none" />
```

PageWebService.aspx

Le javascript à écrire dans WCF.js ressemble grandement à celui écrit pour PageMethod, la seule différence résidence dans l'appel à la méthode de vérification de pseudo.
De la même façon, le ScriptManager nous génère une classe proxy que l'on pourra appeler par : WCFUserService.VerifPseudo(...) :

WCF.js

```
function VerifPseudoJs(idText, idOk, idErr)
{
    WCFUserService.VerifPseudo($get(idText).value,
                function(result, userContext, methodName)
                {
                    if (result)
                    {
                        setDisplay(idOk, true);
                        setDisplay(idErr, false);
                    }
                    else
                    {
                        setDisplay(idOk, false);
                        setDisplay(idErr, true);
                    }
                },
                function(error, userContext, methodName)
                {
                    if (error !== null)
                    {
                        alert('Erreur : ' + error.get_message());
                    }
                }, null);
}
```

De la même façon, on pourrait se servir de la possibilité d'utiliser un objet de contexte, inutile dans notre cas.
On peut constater ici que le javascript est quasiment identique à celui écrit dans le cadre de l'utilisation des PageMethods, cependant on n'est pas limité à l'appel d'une méthode statique de la page. On est libre d'appeler n'importe quel web service compatible avec ASP.NET.
Un avantage certain est que la page est dissociée du service, aucun code behind n'est présent dans la page.

4.Utilisation de l'AJAX avec jQuery

L'utilisation d'un framework javascript nous simplifie un peu l'écrire du javascript et nous permet de nous affranchir des contraintes liées aux différents navigateur.
Dans le cadre des appels AJAX, des fonctionnalités sont également encapsulées.
Nous allons voir comment utiliser **jQuery** dans les deux utilisations de XMLHttpRequest, ainsi que dans l'appel à un Web Service.
Pour utiliser jQuery, on va simplement ajouter la bibliothèque Js, soit **en la référencant directement**, soit en l'incluant à notre solution. C'est cette dernière solution que j'ai choisi.

4.1.Appel à la page CheckPseudo.aspx

Le code de la page est le même, à l'exception des fichiers JS à inclure :

PageXMLHttpRequestPageJQuery.aspx

```
<script type="text/javascript" src="/js/jquery.min.js"></script>
<script type="text/javascript" src="/js/JQueryPage.js"></script>
```

Plus besoin de Helper, jQuery sait tout faire.
Le code du fichier JS devient :

JQueryPage.js

```
function VerifPseudoJs(id)
{
    var value = $("#" + id).val()

    $.ajax({
        type: "POST",
        url: '/Pages/CheckPseudo.aspx',
        contentType: "application/x-www-form-urlencoded",
        dataType: "text",
        data: 'pseudo=' + value,
        success: function(result)
        {
            $("#result").html(result);
        },
        error: function(xhr)
        {
            alert("Erreur : " + xhr.responseText);
        }
    });
}
```

Outre l'accès aux éléments avec $ qui peut paraitre étrange aux personnes n'ayant jamais pratiqué jQuery, le javascript parle de lui même : on voit un appel de la page avec le paramètre en POST.
On retrouve les avantages et inconvénients d'un appel de page. Par contre, le code javascript bénéficie de l'efficacité de jQuery.

4.2.Appel au handler de page

De la même façon, il n'y a pas grand chose à changer pour l'appel au handler.

PageXMLHttpRequestHandlerJQuery.aspx

```
<script type="text/javascript" src="/js/jquery.min.js"></script>
<script type="text/javascript" src="/js/JQueryHandler.js"></script>
```

avec le code javascript suivant :

JQueryHandler.js

```
function VerifPseudoJs(idText, idOk, idErr)
{
    var value = $("#" + idText).val()

    $.ajax({
        type: "POST",
        url: '/ExistPseudo',
        contentType: "application/x-www-form-urlencoded",
        dataType: "text",
        data: 'pseudo=' + value,
        success: function(result)
        {
            if (result == 'True')
            {
                $("#" + idOk).show();
                $("#" + idErr).hide();
            }
```

JQueryHandler.js

```
            else
            {
                $("#" + idOk).hide();
                $("#" + idErr).show();
            }
        },
        error: function(xhr)
        {
            alert("Erreur : " + xhr.responseText);
        }
    });
}
```

Pour afficher/masquer un élément, on appelle juste les méthodes show()/hide().

4.3.Utilisation d'un handler de page renvoyant du JSON

La notation JSON (JavaScript Object Notation) est un format permettant le passage des données à un programme JavaScript. Ce format est un des formats utilisés dans le cadre d'AJAX. Il présente l'avantage d'être plus léger que XML et d'être très facilement récupérable par les scripts JavaScript.
jQuery parle très bien le JSON. Essayons de lui parler un langage qu'il connait bien alors.
Au lieu d'utiliser un handler qui renvoi du *"plain/text"*, faisons lui renvoyer du *"application/json"*.

JSONHandler.cs

```
public class JSONHandler : IHttpHandler
{
 public void ProcessRequest(HttpContext context)
 {
  context.Response.Cache.SetCacheability(HttpCacheability.NoCache);
  context.Response.ContentType = "application/json";
  string pseudo = context.Request.Form["pseudo"];
  if (UserService.PseudoLibre(pseudo))
   context.Response.Write("{result:true}");
  else
   context.Response.Write("{result:false}");
 }

 public bool IsReusable
 {
  get { return false; }
 }
}
```

Ce qui va changer, c'est surtout le ContentType que l'on passe à **application/json**.
De même, on renverra au format JSON que la variable result vaut vrai.
Le javascript va donc contenir un appel AJAX, précisant que le type de données reçues (dataType) sera du JSON.
La callback succès va récupérer un objet javascript (result) qui contiendra l'objet résultat qui a transité (result).
Pour simplifier les choses, je les ai appelé pareil ...
D'où le test : if (result.result)

JQueryJSONHandler.js

```
function VerifPseudoJs(idText, idOk, idErr)
{
    var value = $("#" + idText).val()

    $.ajax({
        type: "POST",
        url: '/JSONHandler',
        contentType: "application/x-www-form-urlencoded",
        dataType: "json",
```

JQueryJSONHandler.js

```
        data: 'pseudo=' + value,
        success: function(result)
        {
            if (result.result)
            {
                $("#" + idOk).show();
                $("#" + idErr).hide();
            }
            else
            {
                $("#" + idOk).hide();
                $("#" + idErr).show();
            }
        },
        error: function(xhr)
        {
            alert("Erreur : " + xhr.responseText);
        }
    });
}
```

L'avantage est qu'on manipule directement des objets javascript.
Cela peut paraitre anodin dans le cas d'un simple booléen, mais peut prendre toute son importance dans le cas d'objets complexes.
Il ne faudra bien sur pas oublier de déclarer le handler dans le web.config :

web.config

```
<add verb="POST" path="JSONHandler" type="CompareAjaxMethods.JSONHandler, CompareAjaxMethods" />
```

4.4.Utilisation de jQuery avec un Web Service JSON

On a dit que jQuery parlait très bien le JSON ? Ca tombe bien, WCF sait également le parler.
Faisons les communiquer ensemble...
La première chose à faire est d'écrire le web service et surtout de le paramétrer pour qu'il puisse comprendre et renvoyer du JSON.
Comme on a fait au **paragraphe 3.5**, dans notre projet, ajoutons un nouvel élément (add new item -> Ajax-enabled WCF Service) : JSONUserService.svc
Il va ressembler beaucoup au précédent, mais va être décoré différemment :

JSONUserService.svc.cs

```
[ServiceContract(Namespace = "")]
[AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Required)]
public class JSONUserService
{
    [OperationContract]
    [WebInvoke(Method = "POST", RequestFormat = WebMessageFormat.Json,
     ResponseFormat = WebMessageFormat.Json, BodyStyle = WebMessageBodyStyle.WrappedRequest)]
    public bool VerifPseudo(string pseudo)
    {
        return UserService.PseudoLibre(pseudo);
    }
}
```

On indique notamment que le web service sera interrogé en POST, et qu'on utilisera du JSON.
La **propriété BodyStyle** mis à WrappedRequest permet d'indiquer que les demandes sont encapsulées, les réponses ne le sont pas.
C'est à dire qu'on va envoyer les paramètres sous la forme JSON : *{"pseudo":"nico"}*, avec "pseudo" le nom du paramètre et "nico" la valeur.

Par contre, comme notre paramètre de retour est un bool, on a aucunement besoin qu'il soit encapsulé dans le format JSON, on recevra simplement true ou false, sous la forme d'un objet javascript.
Pour que ce web service soit interrogeable en JSON, il faudra modifier son comportement dans le web.config.
Au lieu de :

web.config

```
<behavior name="CompareAjaxMethods.JSONUserServiceAspNetAjaxBehavior">
 <enableWebScript />
</behavior>
```

on aura :

web.config

```
<behavior name="CompareAjaxMethods.JSONUserServiceAspNetAjaxBehavior">
  <webHttp />
</behavior>
```

WebHttp permet d'activer le modèle de programmation Web correspondant à un service WFC.
La page ASPX sera sensiblement la même, on inclura ces fichiers javascript :

PageWebServiceJSONJQuery.aspx

```
<script type="text/javascript" src="/js/jquery.min.js"></script>
<script type="text/javascript" src="/js/JQueryWebService.js"></script>
```

avec le contenu de JQueryWebService.js :

JQueryWebService.js

```
function VerifPseudoJs(idText, idOk, idErr)
{
    var value = $("#" + idText).val()

    $.ajax({
        type: "POST",
        url: "/JSONUserService.svc/VerifPseudo",
        contentType: "application/json; charset=utf-8",
        dataType: "json",
        data: '{"pseudo":"' + value + '"}',
        success: function(result)
        {
            if (result)
            {
                $("#" + idOk).show();
                $("#" + idErr).hide();
            }
            else
            {
                $("#" + idOk).hide();
                $("#" + idErr).show();
            }
        },
        error: function(xhr)
        {
            alert("Erreur : " + xhr.responseText);
        }
    });
}
```

On précise tout d'abord l'url du WebService et la méthode à appeler. On précise ensuite le type de contenu et le type de données échangées.
Les paramètres sont ensuite passés au format JSON.
Dans la callback de succès d'appel au web service, on teste le retour, qui est un booleen javascript.

Une remarque sur jQuery :
jQuery possède des méthodes de plus haut niveau, comme **$().load()**, qui permet de faire en un appel la mécanique de mettre à jour du contenu à partir du résultat d'une requête AJAX. Ou encore **$.post(url,data,callback)**.
J'ai préféré montrer ici l'utilisation avec la méthode de plus bas niveau **$().ajax**, mais l'utilisation de méthodes de plus haut niveau permettrait de réduire encore la complexité du javascript.

5.Considérations sur les appels AJAX

Grâce aux chapitres précédents, on a mis en avant les différences que comportent les différentes techniques de rendu partiel présentées.
Ces différences portent sur la nature du code, le type de code et l'architecture de la solution.
Pour pousser plus loin cette analyse, et pour vous permettre de choisir la méthode qui vous convient le mieux, je me devais d'analyser le contenu des échanges entre le navigateur et le serveur, et notamment la taille des données qui transitent.
C'est un facteur qui aujourd'hui peut perdre un peu de son importance avec l'avènement des larges bandes passantes, mais qui reste d'actualité lorsque la qualité de l'expérience utilisateur peut dépendre de la taille des données qui transitent.
Observons donc ce qu'il se passe dans tous les cas présentés.

5.1.Méthode avec l'UpdatePanel

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageUpdatePanel.aspx	2700	
JS ajoutés par ASP.NET	20794 + 313061 + 78646	
Envoi en POST	310	ScriptManager=UpdatePanel1%7CButton1&__EVE %2FwEPDwUKLTU5MjAxNzEzMQ9kFgICAw9kFgI %2FwEWAwLj1677DALvz %2FGACgKM54rGBhxgw6d1kAbguDZbghloMQwq %C3%A9rifier
Réponse de l'envoi	838	Pas mal de code html ...
ok.png (en réponse)	1058	

Il est à noter que la taille des fichiers JS ajoutés par le framework ASP.NET peut varier en fonction de l'environnement (ces tests ont été faits en debug par exemple).
On remarque que lors de l'envoi en POST, on passe notamment les valeurs des contrôles, les ID, le viewstate, etc ...

5.2.Méthode avec XMLHttpRequest et appel de CheckPseudo.aspx

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageXMLHttpRequestPage.aspx	1141	
Helper.js	914	
XmlHttpPage.js	562	
Envoi en POST	11	pseudo=nico
Réponse de l'envoi	340	<form name="form1" method="post" action="CheckPseudo.aspx"

		id="form1"> <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/ wEPDwULLTExNzU3MzQyMTAPZBYCZg9kFgICA =" /> </form>
ok.png (en réponse)	1058	

5.3.Méthode avec XMLHttpRequest et appel du handler

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageXMLHttpRequestHandler.aspx	1305	
Helper.js	914	
XmlHttpHandler.js	820	
ok.png	1058	
err.png	402	
Envoi en POST	11	pseudo=nico
Réponse de l'envoi	4	True

5.4.Avec les PageMethods

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PagePageMethods.aspx	5953	
JS ajoutés par ASP.NET	20794 + 313061 + 78646	
Helper.js	914	
PageMethod.js	728	
ok.png	1058	
err.png	402	
Envoi en POST	17	{"pseudo":"nico"}
Réponse de l'envoi	10	{"d":true}

Si la page est si grosse (5953), c'est parce qu'il y a pas mal de javascript généré, dont le proxy.
Notons que le format des données échangées est du JSON.

5.5.Avec un Web Service WCF

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageWebService.aspx	2995	
JS ajoutés par ASP.NET	20794 + 313061 + 78646	
Helper.js	914	
WCF.js	811	
Proxy généré	3062	
ok.png	1058	
err.png	402	
Envoi en POST	17	{"pseudo":"nico"}
Réponse de l'envoi	10	{"d":true}

Notons encore ici que le format des données échangées est toujours du JSON.
Le proxy généré est dans un fichier javascript à part, ce qui peut lui permettre d'être mis en cache par le navigateur. Ce qui n'est pas le cas avec les PageMethods.

5.6.Avec jQuery et CheckPseudo.aspx

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageXMLHttpRequestPageJQuery.aspx	1157	
jquery.min.js	55802	
JQueryPage.js	481	
Envoi en POST	11	pseudo=nico
Réponse de l'envoi	340	
ok.png (en réponse)	1058	

5.7.Avec jQuery et un handler de page

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageXMLHttpRequestHandlerJQuery.aspx	1305	
jquery.min.js	55802	
JQueryHandler.js	722	
ok.png	1058	
err.png	402	
Envoi en POST	11	pseudo=nico
Réponse de l'envoi	4	True

5.8.Avec jQuery et le handler de page JSON

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageHandlerJSONJQuery.aspx	1268	
jquery.min.js	55802	
JQueryJSONHandler.js	719	
ok.png	1058	
err.png	402	
Envoi en POST	11	pseudo=nico
Réponse de l'envoi	13	{result:true}

5.9.Avec jQuery et un web service WCF

Nom de l'élément	Taille (octets)	Contenu (si pertinent)
PageWebServiceJSONJQuery.aspx	1315	
jquery.min.js	55802	
JQueryWebService.js	741	
ok.png	1058	
err.png	402	
Envoi en POST	17	{"pseudo":"nico"}
Réponse de l'envoi	4	true

5.10.Totaux

Méthode	Taille total du chargement initial de la page (octets)	Taille de l'envoi de la requête AJAX (octets)	Taille de la réception de la requête AJAX(octets)
UpdatePanel	415201	310	1896
XMLHttpRequest et page	2617	11	1398
XMLHttpRequest et handler	4499	11	4
PageMethods	421556	17	10
ScriptManager et WebService	421743	17	10
jQuery et page ASP.NET	57440	11	1398
jQuery et handler de page	59289	11	4
jQuery et handler de page JSON	59249	11	13
jQuery et WebService JSON	59318	17	4

Ces résultats sont intéressant, mais il ne faut pas les prendre tels quels.
Il faut dans un premier temps garder à l'esprit que les fichiers javascripts peuvent être mis en cache par le navigateur, ce qui fait que tout n'est pas forcément à recharger à chaque navigation.
N'oubliez pas également que pour choisir une solution, il faut faire la part des choses entre le temps passé à développer la page, à développer le javascript adapté et le nombre d'octets échangés lors de chaque requête AJAX.
Si le nombre de requête a tendance à être important et si le confort de l'utilisateur et la réactivité de la page priment, alors on pourra avoir tendance à utiliser les méthodes où il faut écrire beaucoup de javascript.
Si la page n'est pas énormément utilisée, dans des conditions réseaux adéquates et si le code client risque d'être complexe à écrire et à maintenir, il est tout à fait envisageable d'utiliser l'UpdatePanel.
Enfin, il ne faut pas non plus négliger les avantages d'un découplage des pages ASPX.

6.Téléchargement

Vous pouvez télécharger ici les sources du projet de démo : **version rar (68 Ko)** , **version zip (79 Ko)**.

7.Conclusion

Cet article montre différentes méthodes pour effectuer des rendus partiels sur un site ASP.NET. Certaines méthodes utilisent abondamment les mécanismes mis en place par le framework ASP.NET (en utilisant par exemple un UpdatePanel, les PageMethods ou un service WCF). Elles permettent de s'affranchir de certaines lourdeurs automatiquement.
D'autres sont plus artisanales et permettent de maitriser complètement les données qui transitent et la façon dont elles doivent être interprétées.
A travers cette étude, on a pu estimer quels sont les points à retenir lors du choix d'une solution pour effectuer des rendus partiels sur une page. Il faut bien veiller à choisir une solution en fonction de ses besoins en sachant s'il vaut mieux maitriser le format des données au détriment de la simplicité de développement.

Remerciements

Je remercie l'équipe Dotnet pour leurs relectures attentives du document.

Contact

Si vous constatez une erreur dans le tutorial, dans le source, dans la programmation ou pour toutes informations, n'hésitez pas à me contacter **par mail**, ou **par le forum**.