Modification "inline" de données avec AJAX

par Olivier Lance (Accueil)

Date de publication : 15/08/06

Dernière mise à jour :

Ce tutoriel a pour but de vous montrer comment mettre en oeuvre simplement un script de modification "inline" de vos données affichées dans une page web.

Introduction

De plus en plus, la "mode" pousse les développeurs à créer des pages web dynamiques et interactives construites pour un confort d'utilisation toujours augmenté pour l'utilisateur final.

Aujourd'hui ce confort passe, entr'autres, par l'emploi du **Javascript** et de l'objet **XMLHTTPRequest**, qui permet d'effectuer des requêtes vers le serveur web de manière asynchrone. Couplé avec quelques scripts PHP, il permet de mettre à jour des informations au sein d'une page sans en recharger l'intégralité du contenu.

C'est cette méthode que je vous propose d'utiliser pour mettre en place un système de *modification inline* de données dans une page web.

Par *modification inline*, j'entends modification d'éléments distincts de la page, directement à leur emplacement d'origine.

Pour bien vous rendre compte de l'idée, vous pouvez d'ores et déjà trouver **ici** un exemple fonctionnel du résultat de ce tutoriel. L'utilisation est simple : en double-cliquant sur une donnée du tableau, vous entrez en mode d'édition. Vous pouvez alors changer la valeur de la donnée, puis valider votre saisie en appuyant sur Entrée ou en cliquant à l'extérieur du champ de texte.

Actualisez, la valeur que vous avez entrée est sauvegardée !

Ce tutoriel est découpé en deux parties : la première partie vous guidera et vous expliquera les étapes de création de ce script d'édition. La seconde partie, pour les plus chevronné(e)s, reprendra le code de la première partie pour le généraliser et permettre une plus grande flexibilité d'implémentation grâce à l'utilisation de classes.

Dans tout le tutoriel, je considèrerai que vous êtes à l'aise avec l'utilisation de l'objet XMLHTTPRequest et avec tout ce qui concerne les appels de requêtes MySQL à partir de PHP, de leur création à l'exploitation de leur résultat.

Depuis un moment, mon script comprenait une erreur qui empêchait l'enregistrement dans la table des modifications apportées par l'utilisateur.

*Je tiens à m'excuser pour ce problème et le temps que j'ai pris à le résoudre. Je vous propose de télécharger **ce zip** qui contient une version des fichiers qui fonctionne correctement en local sur ma machine.*

*(en cas de problème avec le lien précédent : **miroir**)*

I - Première partie : Création du script "simple"

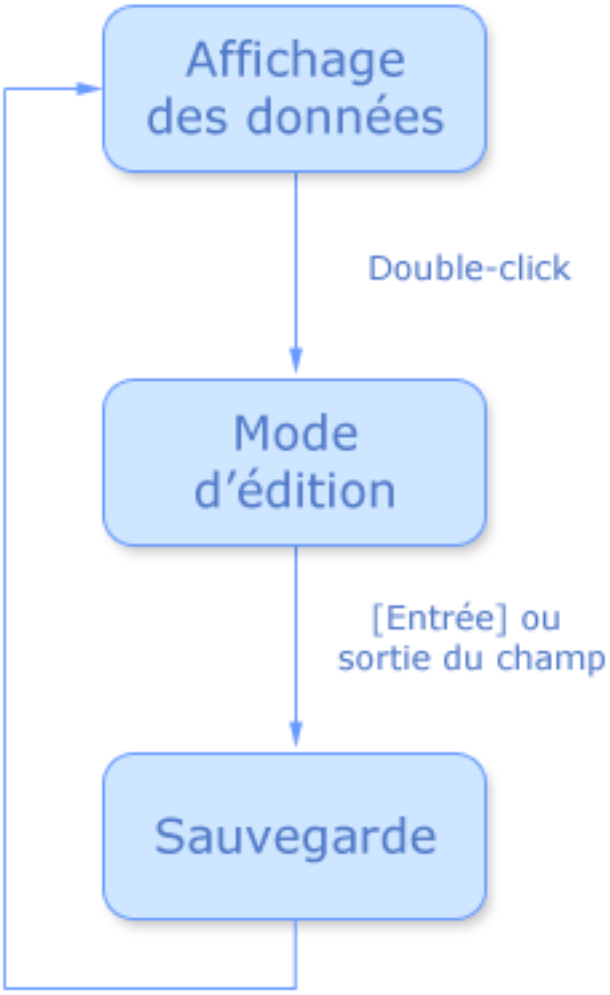
I-1 - Analyse préliminaire

Comme vous avez pu le voir dans la page donnée à l'introduction, l'idée est simple : lors d'un double-click sur l'une des cellules du tableau, son contenu est remplacé par un champ de saisie qui prend pour valeur le texte de la cellule.

Le double-click permet donc de passer d'un mode d'**affichage** à un mode d'**édition**.

Dans ce mode d'édition, la valeur affectée est librement modifiable, puis enregistrée soit en appuyant sur la touche Entrée, soit en sortant du champ de saisie.

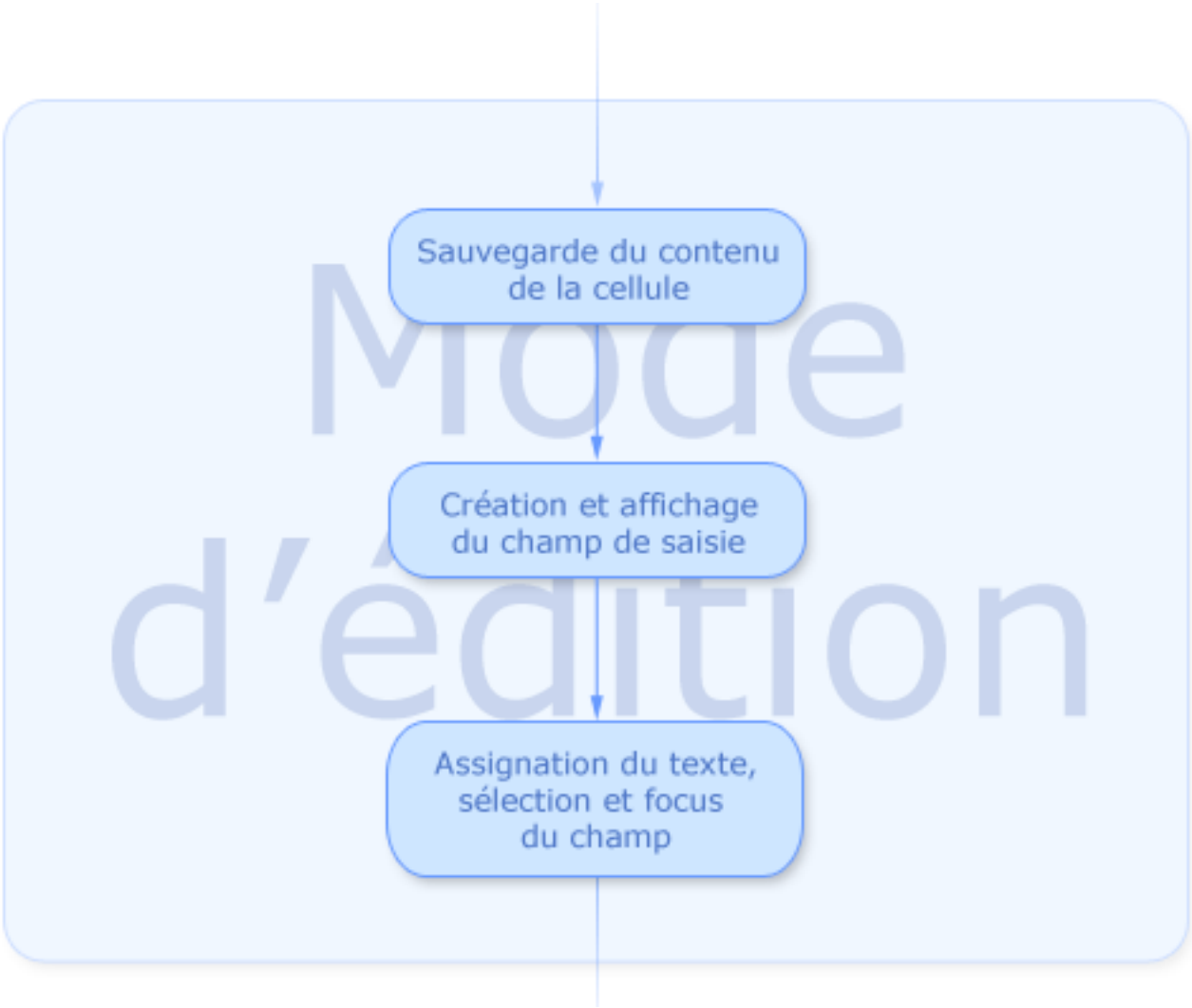
De ces simples constats on peut donc tirer l'analyse descendante suivante :



Analyse générale

L'affichage des données se fait dans un premier temps grâce à la page web dont le code sera présenté peu après. Lors du retour vers le mode d'affichage après la sauvegarde des données, nous devrons user d'un peu de Javascript afin de supprimer le champ de saisie et de remettre le texte qu'il contenait dans la cellule.

Le passage en mode d'édition peut être découpé en plusieurs actions principales :



Analyse du mode d'édition

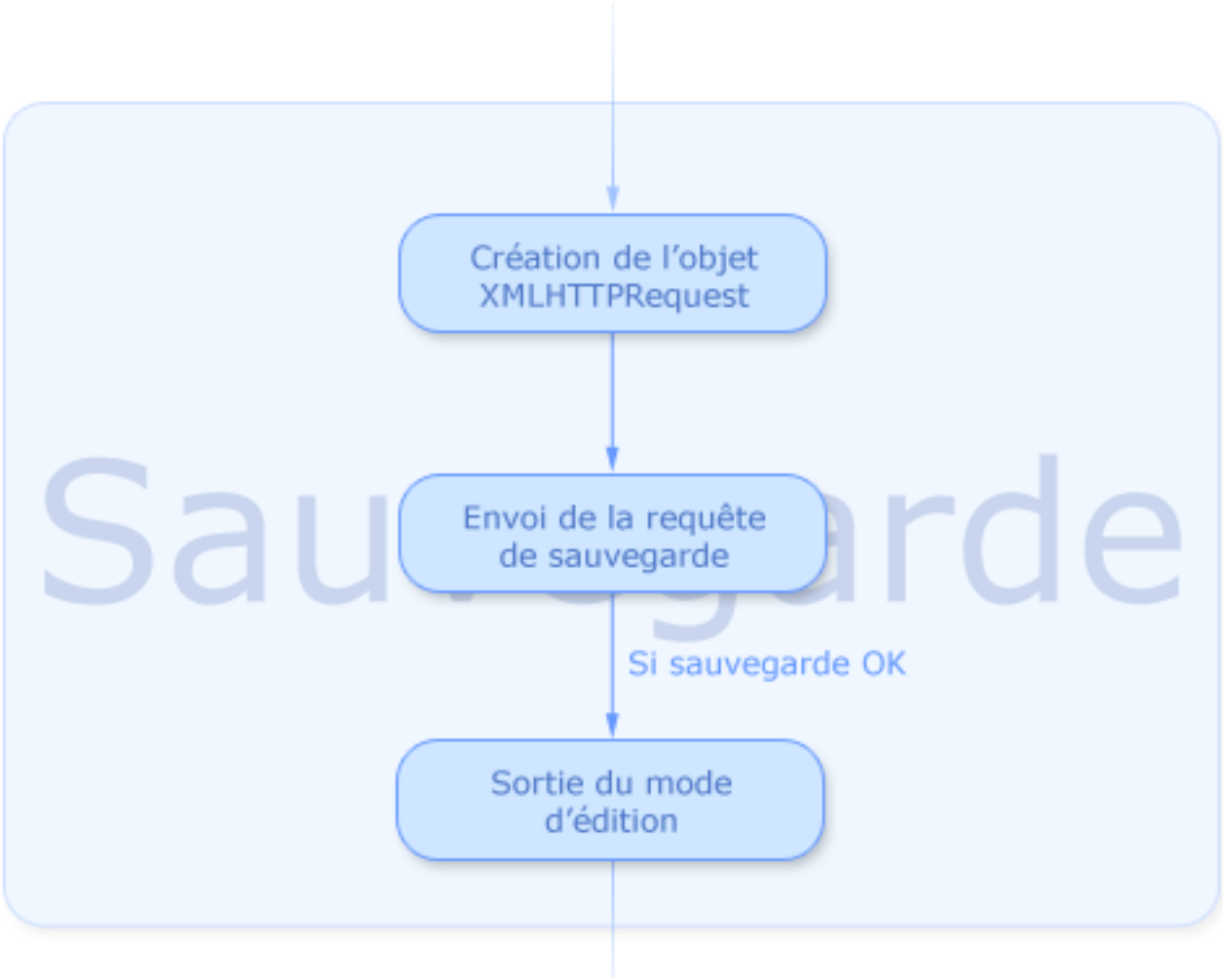
Dans un premier temps, avant de remplacer le contenu de la cellule, il faut en garder une copie pour pouvoir l'assigner au contenu du champ de saisie.

Ensuite, le champ de saisie est créé. Avant de l'afficher, il conviendra de modifier quelques unes de ses propriétés : sa taille, son style CSS...

Et enfin, une fois le champ affiché, on sélectionne son contenu et on lui donne le focus pour que l'édition puisse être immédiatement effectuée au clavier.

Selon le type de valeur à modifier (texte, texte "long" pouvant s'étendre sur plusieurs lignes, nombre...), nous introduirons un comportement différent. Pour ce tutoriel, les textes et nombres seront modifiés grâce à un champ de saisie classique (balise **input**) et les textes longs seront modifiés grâce à une zone de texte multilignes (balise **textarea**). Il faudra bien sûr, à un moment ou un autre, spécifier quel est le type de la donnée affichée.

De même, la phase de sauvegarde peut être ainsi découpée :



Analyse de la sauvegarde

Cette phase est relativement simple. Une fois l'objet XMLHTTPRequest créé, nous appelons un script PHP en lui passant en paramètres les valeurs à sauver, puis nous sortons du mode d'édition une fois la sauvegarde effectuée.

Afin de garder un script relativement général, nous ne différencierons pas la requête selon le type du champ modifié, ou bien selon le nom du champ correspondant dans la base de données (ce qui reviendrait à faire une requête par champ). Nous passerons donc en paramètres au script le nom du champ dans la base de données, son type, sa valeur et bien sûr l'id de l'enregistrement dont il faut modifier le champ.

Comme vous pouvez le voir dans le diagramme précédent, nous ne ferons pas de gestion d'erreur de la sauvegarde dans cette première partie. Je considère que vous savez utiliser l'objet XMLHTTPRequest et ses propriétés pour arriver à vos fins sur ce plan.

La sortie du mode d'édition se fait en deux étapes principales : remplacement du champ de saisie par son contenu dans la cellule en cours d'édition, puis suppression du champ.

Nous avons dorénavant en main tous les élements pour passer à la réalisation des différents codes qui permettront de donner corps à ce système d'édition inline.

I-2 - Côté client

La page web et le script PHP (côté serveur) qui seront utilisés sont bien sûr spécifiques à l'exemple que j'entends traiter dans ce tutoriel. Avant toute chose, il me semble donc judicieux de donner dès maintenant la structure de la table qui contiendra les données utilisées. Il s'agit d'une simple table recensant les états civils d'utilisateurs fictifs :

```
CREATE TABLE `inlinemod` (
  `id`    int(11) NOT NULL auto_increment,
  `nom`   varchar(255) NOT NULL default '',
  `prenom`   varchar(255) NOT NULL default '',
  `adresse`  tinytext NOT NULL,
  `code_postal`  varchar(5) NOT NULL default '',
  `ville`   varchar(255) NOT NULL default '',
  `enfants`  int(11) NOT NULL default '0',
  `email`   varchar(255) NOT NULL default '',
  PRIMARY KEY  (`id`)
)
```

Dans cette table, tous les champs affichés et modifiables sont donc de type **texte**, à l'exception du champ *adresse* qui sera de type **texte-multi** et du champ *enfants* qui sera de type **nombre**.

I-2-a - La page web

Le code de la page web de présentation des données est des plus basiques : une connexion à la base pour récupérer les données à afficher, puis une boucle afin de construire le tableau contenant celles-ci. Voici déjà le code complet pour une vue d'ensemble. Nous nous concentrerons ensuite sur les détails importants.

```
<?php
// Connexion à la base de données
mysql_connect(DB_HOST, DB_USER, DB_PASSWORD) or die(mysql_error());
mysql_select_db(DB_NAME) or die(mysql_error());

$sql = 'SELECT * FROM `'.DB_TABLE_NAME.'`';
$result = mysql_query($sql) or die(__LINE__.mysql_error().$sql);

?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=iso-8859-1" />
```

```
    <title>Modification "inline" de données grâce à XMLHTTPRequest</title>

    <link rel="StyleSheet" type="text/css" href="index.css"/>
    <script type="text/javascript" src="inlinemod.js"></script>
</head>

<body>
    <h1>Utilisateurs</h1>


    <table id="table-utilisateurs">
        <tr>
            <th>Nom</th>
            <th>Prénom</th>
            <th>Adresse</th>
            <th>Code Postal</th>
            <th>Ville</th>
            <th>Enfants</th>
            <th>Email</th>
        </tr>

        <?php
        while($user = mysql_fetch_assoc($result))
        {
        ?>
            <tr>
                <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this, 'nom',
 'texte')">
                    <?php echo $user['nom']; ?>
                </td>

                <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this,
 'prenom', 'texte')">
                    <?php echo $user['prenom']; ?>
                </td>

                <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this,
 'adresse', 'texte-multi')">
                    <?php echo $user['adresse']; ?>
                </td>

                <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this,
 'code_postal', 'texte')">
                    <?php echo $user['code_postal']; ?>
                </td>

                <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this,
 'ville', 'texte')">
                    <?php echo $user['ville']; ?>
                </td>

                <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this,
 'enfants', 'nombre')">
                    <?php echo $user['enfants']; ?>
                </td>

                <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this,
 'email', 'texte')">
                    <?php echo $user['email']; ?>
                </td>
            </tr>
        <?php
        }
        ?>
    </table>

</body>
```

```
</html>

<?php

 mysql_close();

?>
```

Pour votre propre code vous aurez bien évidemment à définir les variables de connexion à la base de données, mais là n'est pas notre sujet.

En premier lieu on remarque l'inclusion du fichier **inlinemod.js**, qui contiendra les fonctions utilisées pour le mode d'édition et la sauvegarde.

Le plus important se situe au niveau des cellules du tableau, dont nous allons détailler les attributs. Pour référence voici déjà une copie de la la balise d'ouverture de la première cellule :

```
<td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this, 'nom', 'texte')">
```

L'attribut **class** ne fait qu'assigner à la cellule une classe CSS que voici :

```
td.cellule {
 text-align: center;
 border: 1px solid #376ef9;

 cursor: pointer;
}
```

La modification de l'apparence du curseur sur les cellules permet de signifier de manière simple à l'utilisateur qu'une action est ici possible.

L'attribut **ondblclick** nous plonge plus en avant dans le sujet puisqu'il est le point d'entrée vers le mode d'édition, grâce à la fonction **inlineMod**. Celle-ci prend quatre paramètres :

1. L'**id** de l'enregistrement dans la base de données, afin de savoir quoi modifier lors de l'appel du script PHP
2. Une référence sur l'objet qui contient la valeur à modifier. Ici, **this** est passé pour désigner la balise **td**
3. Le **nom** du champ à modifier dans la base de données, toujours pour renseigner le script PHP
4. Le **type** de la valeur. Ici elle est de type **texte**, plus bas dans la page vous trouverez également un type **texte-multi** et un type **nombre**

Voyons maintenant ce que donne, justement, l'implémentation de cette fonction **inlineMod**.

I-2-b - Le Javascript

I-2-b-i - Mode d'édition

Le code que je vais vous présenter ici ne respecte pas *exactement* l'ordre de l'analyse présentée ci-dessus. L'esprit est le même bien sûr, mais le mode de fonctionnement du Javascript et notamment les possibilités offertes par la manipulation du **DOM** permettent de regrouper certaines étapes.

Avant toute chose, afin qu'une seule édition ne soit effectuée à la fois, nous allons introduire une variable globale de type booléen qui nous permettra de valider ou non le passage au mode d'édition suivant qu'une édition est déjà en cours. Un test sur cette variable en début de fonction permettra ce contrôle :

```
//On ne pourra éditer qu'une valeur à la fois
var editionEnCours = false;

//Fonction de modification inline de l'élément double-cliqué
function inlineMod(id, obj, nomValeur, type)
{
 if(editionEnCours)
 {
  return false;
 }
 else
 {
  editionEnCours = true;
 }
```

Si la fonction n'est pas stoppée, nous pouvons alors créer notre champ de saisie en fonction du **type** de la valeur à modifier :

```
 //Objet servant à l'édition de la valeur dans la page
 var input = null;

 //On crée un composant différent selon le type de la valeur à modifier
 switch(type)
 {
  //Valeur de type texte ou nombre
  case "texte":
  case "nombre":
   input = document.createElement("input");
   break;

  //Valeur de type texte multilignes
  case  "texte-multi":
   input = document.createElement("textarea");
   break;
 }
```

Comme annoncé précédemment nous créons, grâce au DOM, une balise **input** pour les types **texte** et **nombre**, et une balise **textarea** pour le type **texte-multi**.

L'élément créé peut ensuite être manipulé comme nous le désirons. Nous allons affecter le texte à éditer à sa propriété **value** puis adapter sa taille à la largeur de ce texte :

```
 //Assignation de la valeur
 if (obj.innerText)
  input.value = obj.innerText;
 else
  input.value = obj.textContent;

 input.value = trim(input.value);

 //On lui donne une taille un peu plus large que le texte à modifier
 input.style.width  = getTextWidth(input.value) + 30 + "px";
```

Dans **value** est placé le contenu sous forme de texte de l'objet parent. Par souci de compatibilité, un test est effectué pour savoir qui de **innerText** (Internet Explorer, Opera, Safari, Konqueror) ou **textContent** (Firefox, ...) doit être utilisé. Comme Firefox renvoie également les sauts de lignes et espaces présents dans la balise dont on appelle **textContent**, une fonction **trim** est utilisée pour supprimer ceux-ci. Son implémentation est donnée dans le prochain extrait de code.

Afin d'adapter la taille du champ de saisie à son contenu, la fonction **getTextWidth** est utilisée. Il s'agit d'une petite astuce utilisant le **DOM** et la propriété **offsetWidth** pour "mesurer" la taille d'un texte placé dans une balise **span** :

```
//Suppression des espaces/sauts de ligne inutiles
 (http://www.breakingpar.com/bkp/home.nsf/0/87256B280015193F87256C0C0062AC78)
function trim(value) {
   var temp = value;
   var obj = /^(\s*)([\W\w]*)(\b\s*$)/;
   if (obj.test(temp)) { temp = temp.replace(obj, '$2'); }
   var obj = /  /g;
   while (temp.match(obj)) { temp = temp.replace(obj, " "); }
   return temp;
}


//Fonction donnant la largeur en pixels du texte donné (merci SpaceFrog !)
function getTextWidth(texte)
{
 //Valeur par défaut : 150 pixels
 var largeur = 150;

 if(trim(texte) == "")
 {
  return largeur;
 }

 //Création d'un span caché que l'on "mesurera"
 var span = document.createElement("span");
 span.style.visibility = "hidden";
 span.style.position = "absolute";

 //Ajout du texte dans le span puis du span dans le corps de la page
 span.appendChild(document.createTextNode(texte));
 document.getElementsByTagName("body")[0].appendChild(span);

 //Largeur du texte
 largeur = span.offsetWidth;

 //Suppression du span
 document.getElementsByTagName("body")[0].removeChild(span);
 span = null;

 return largeur;
}
```

Les commentaires du code devraient suffire à décrire cette fonction qui n'est pas d'une difficulté particulière. Une fois les propriétés de notre champ ajustées, nous pouvons l'afficher dans la cellule, sélectionner son contenu et lui donner le focus :

```
 //Remplacement du texte par notre objet input
 obj.replaceChild(input, obj.firstChild);

 //On donne le focus à l'input et on sélectionne le texte qu'il contient
 input.focus();
```

```
    input.select();
```

Il reste maintenant à définir les événements qui déclencheront la sauvegarde de la saisie. La sortie du champ sera détectée grâce à l'événement **onblur**, tandis que l'appui sur la touche Entrée sera vérifié grâce à l'événement **onkeydown** et un test sur la touche appuyée lorsque l'événement est déclenché.

```
    //Sortie de l'input
    input.onblur = function sortir()
    {
     sauverMod(id, obj, nomValeur, input.value, type);
     delete input;
    };

    //Appui sur la touche Entrée
    input.onkeydown = function keyDown(event)
    {
           if (!event&&window.event)
           {
               event = window.event;
           }
     if(getKeyCode(event) == 13)
           {
      sauverMod(id, obj, nomValeur, input.value, type);
      delete input;
     }
    };
```

Pour la compatibilité, quelques tests s'imposent sur la propriété **event**. L'important est la comparaison avec le code caractère **13** qui représente le saut de ligne, et qui nous prévient donc d'un appui sur la touche Entrée.

La fonction getKeyCode renvoie ce code de caractère à partir de la propriété **event** :

```
    //Fonction renvoyant le code de la touche appuyée lors d'un événement clavier
    function getKeyCode(evenement)
    {
        for (prop in evenement)
        {
            if(prop == 'which')
            {
                return evenement.which;
            }
        }

        return evenement.keyCode;
    }
```

La fonction **inlineMod** est maintenant terminée. Son fonctionnement correspond à l'analyse que nous en avions fait en première partie, et il reste maintenant à implémenter la fonction de sauvegarde, **sauverMod**.

Cependant, vous l'avez peut-être déjà deviné, il existe un petit problème dans notre code. Après la sauvegarde, le champ de saisie est supprimé. Si le champ possédait encore le focus à cet instant, cela causera donc le déclenchement de l'événement **onblur**. Ainsi, si la sauvegarde est provoquée par l'appui sur la touche Entrée, la fonction **sauverMod** sera appelée deux fois.

Pour pallier ce problème, nous allons introduire une seconde variable d'état qui marchera tout comme **editionEnCours** pour vérifier que nous sommes déjà passés ou non par la fonction de sauvegarde.

Le code final de notre fonction **inlineMod** est donc :

Fonction inlineMod

```
//On ne pourra éditer qu'une valeur à la fois
var editionEnCours = false;

//variable évitant une seconde sauvegarde lors de la suppression de l'input
var sauve = false;

//Fonction de modification inline de l'élément double-cliqué
function inlineMod(id, obj, nomValeur, type)
{
 if(editionEnCours)
 {
  return false;
 }
 else
 {
  editionEnCours = true;
  sauve = false;
 }

 //Objet servant à l'édition de la valeur dans la page
 var input = null;

 //On crée un composant différent selon le type de la valeur à modifier
 switch(type)
 {
  //Valeur de type texte ou nombre
  case "texte":
  case "nombre":
   input = document.createElement("input");
   break;

  //Valeur de type texte multilignes
  case  "texte-multi":
   input = document.createElement("textarea");
   break;
 }

 //Assignation de la valeur
 if (obj.innerText)
  input.value = obj.innerText;
 else
  input.value = obj.textContent;

 input.value = trim(input.value);

 //On lui donne une taille un peu plus large que le texte à modifier
 input.style.width  = getTextWidth(input.value) + 30 + "px";

 //Remplacement du texte par notre objet input
 obj.replaceChild(input, obj.firstChild);

 //On donne le focus à l'input et on sélectionne le texte qu'il contient
 input.focus();
 input.select();

 //Assignation des deux événements qui déclencheront la sauvegarde de la valeur

 //Sortie de l'input
 input.onblur = function sortir()
 {
  sauverMod(id, obj, nomValeur, input.value, type);
```

Fonction inlineMod

```
     delete input;
    };

    //Appui sur la touche Entrée
    input.onkeydown = function keyDown(event)
    {
           if (!event&&window.event)
           {
               event = window.event;
           }
     if(getKeyCode(event) == 13)
           {
      sauverMod(id, obj, nomValeur, input.value, type);
      delete input;
     }
    };
   }
```

Tout est maintenant prêt pour l'implémentation de la fonction **sauverMod**.

I-2-b-ii - Sauvegarde

Notre fonction de sauvegarde s'articule autour de l'objet **XMLHTTPRequest**. Il convient donc de commencer par sa création. Pour cela nous allons utiliser une fonction, **getXMLHTTP**, qui nous renverra une instance de l'objet selon le navigateur utilisé.

```
// retourne un objet xmlHttpRequest.
// méthode compatible entre tous les navigateurs (IE/Firefox/Opera)
function getXMLHTTP()
{
    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)
        {
            try
            {
                xhr = new ActiveXObject("Microsoft.XMLHTTP");
            }
            catch(e1)
            {
                xhr = null;
            }
        }
    }
    else
    { // XMLHttpRequest non supporté par le navigateur
        alert("Votre navigateur ne supporte pas les objets XMLHTTPRequest...");
    }

    return xhr;
}
```

Je ne commenterai pas plus cette fonction. Elle est directement tirée du tutoriel de **Denis Cabasson**, **un autocomplétion pas à pas**, que je ne peux que vous conseiller de lire ! (Voyez également les autres liens en bas de page)

Voici comment nous allons créer notre objet :

```
//Objet XMLHTTPRequest
var XHR = null;

//Fonction de sauvegarde des modifications apportées
function sauverMod(id, obj, nomValeur, valeur, type)
{
 //Si on a déjà sauvé la valeur en cours, on sort
 if(sauve)
 {
  return false;
 }
 else
 {
  sauve = true;
 }

 //Si l'objet existe déjà on abandonne la requête et on le supprime
 if(XHR && XHR.readyState != 0)
 {
  XHR.abort();
  delete XHR;
 }

 //Création de l'objet XMLHTTPRequest
 XHR = getXMLHTTP();

 if(!XHR)
 {
  return false;
 }
```

Tout le code ne concerne pas directement la création de notre objet. Le premier bloc **if**, vous l'aurez compris, permet d'invalider un appel à la fonction de sauvegarde si celle-ci a déjà été appelée pour la même phase d'édition.

XHR est déclaré comme une variable globale à laquelle nous assignons le résultat de **getXMLHTTP**. Avant cette assignation, nous vérifions par sécurité que l'objet n'est pas déjà créé et en cours de transaction avec le serveur. Le cas échéant, la connexion est tout simplement coupée et l'objet détruit avant d'être recréé.

En suivant l'analyse effectuée en première partie, il reste donc à envoyer la requête vers le script PHP et à sortir du mode d'édition. Cela se fait de la manière suivante :

```
 //URL du script de sauvegarde auquel on passe la valeur à modifier
 XHR.open("GET", "sauverMod.php?id=" + id + "&champ=" + nomValeur + "&valeur=" + escape(valeur) +
 "&type=" + type + ieTrick(), true);

 //On se sert de l'événement OnReadyStateChange pour supprimer l'input et le replacer par son
 contenu
 XHR.onreadystatechange = function()
 {
  //Si le chargement est terminé
```

```
    if (XHR.readyState == 4)
    {
     //Réinitialisation de la variable d'état d'édition
     editionEnCours = false;

     //Remplacement de l'input par le texte qu'il contient
     obj.replaceChild(document.createTextNode(valeur), obj.firstChild);
    }
   }

   //Envoi de la requête
   XHR.send(null);
```

Le script est appelé avec les paramètres déjà annoncés : l'**id** de l'enregistrement à modifier, le **nom** du champ édité dans cet enregistrement, la **valeur** que celui-ci doit prendre et le **type** de cette valeur.

Le retour en mode affichage se fait dans l'événement **onreadystatechange** de XHR. Si l'appel du script PHP s'est bien passé, et donc si **readyState** vaut **4**, la variable **editionEnCours** est réinitialisée (**sauve** l'étant au début de la fonction), et le contenu de l'objet parent du texte édité est remplacé grâce au **DOM**.

Vous remarquerez également l'appel à la fonction **ieTrick** dans la construction de la requête. **ieTrick** renvoie une valeur "aléatoire" qui force le navigateur à envoyer la requête sans utiliser son cache. Sans cela, avec Internet Explorer par exemple, si vous modifiez deux fois le même champ du même enregistrement en lui donnant la même valeur à chaque fois, seule la première requête sera réellement envoyée.

Pour information, voici le code de cette fonction :

```
function ieTrick(sep)
{
 d = new Date();
 trick = d.getYear() + "ie" + d.getMonth() + "t" + d.getDate() + "r" + d.getHours() + "i"
  + d.getMinutes() + "c" + d.getSeconds() + "k" + d.getMilliseconds();

 if (sep != "?")
 {
  sep = "&";
 }

 return sep + "ietrick=" + trick;
}
```

Ci-dessous vous trouverez le code complet de la fonction **sauverMod** :

Fonction sauverMod

```
//Objet XMLHTTPRequest
var XHR = null;

//Fonction de sauvegarde des modifications apportées
function sauverMod(id, obj, nomValeur, valeur, type)
{
 //Si on a déjà sauvé la valeur en cours, on sort
 if(sauve)
 {
  return false;
 }
 else
```

Fonction sauverMod

```
    {
     sauve = true;
    }

    //Si l'objet existe déjà on abandonne la requête et on le supprime
    if(XHR && XHR.readyState != 0)
    {
     XHR.abort();
     delete XHR;
    }

    //Création de l'objet XMLHTTPRequest
    XHR = getXMLHTTP();

    if(!XHR)
    {
     return false;
    }

    //URL du script de sauvegarde auquel on passe la valeur à modifier
    XHR.open("GET", "sauverMod.php?id=" + id + "&champ=" + nomValeur + "&valeur=" + escape(valeur) +
    "&type=" + type + ieTrick(), true);

    //On se sert de l'événement OnReadyStateChange pour supprimer l'input et le replacer par son
    contenu
    XHR.onreadystatechange = function()
    {
     //Si le chargement est terminé
     if (XHR.readyState == 4)
     {
      //Réinitialisation de la variable d'état d'édition
      editionEnCours = false;

      //Remplacement de l'input par le texte qu'il contient
      obj.replaceChild(document.createTextNode(valeur), obj.firstChild);
     }
    }

    //Envoi de la requête
    XHR.send(null);
   }
```

Ce dernier extrait de code met fin à l'écriture de notre script d'édition inline, qui est maintenant presque fonctionnel. Il ne nous reste plus qu'à créer le script PHP appelé pour la sauvegarde des données dans la base et cette première partie du tutoriel sera terminée.

I-3 - Côté serveur

Du côté du serveur, seul le script PHP de sauvegarde est présent. Son rôle consiste à récupérer les valeurs passées en paramètres par l'objet **XMLHTTPRequest**, à construire une requête de mise à jour du champ spécifié en fonction de son type et à faire exécuter cette requête par MySQL.

```
    <?php

   //On sort en cas de paramètre manquant ou invalide
   if(empty($_GET['id']) or empty($_GET['type']) or empty($_GET['champ']) or empty($_GET['valeur'])
      or !is_numeric($_GET['id'])
      or !in_array(
        $_GET['champ'],
           array('nom', 'prenom', 'adresse', 'code_postal', 'ville', 'enfants', 'email')
           ))
```

```
{
    exit;
}

//Connexion à la base de données
mysql_connect(DB_HOST, DB_USER, DB_PASSWORD) or die(mysql_error());
mysql_select_db(DB_NAME) or die(mysql_error());

// Construction de la requête en fonction du type de valeur
switch($_GET['type'])
{
    case 'texte':
    case 'texte-multi':
        $sql  = 'UPDATE `'.DB_TABLE_NAME;
        $sql .= '` SET ' . mysql_real_escape_string($_GET['champ']) . '="';
        $sql .= mysql_real_escape_string($_GET['valeur']) . '" WHERE id=' . intval($_GET['id']);
        break;

    case 'nombre':
        $sql  = 'UPDATE `'.DB_TABLE_NAME;
        $sql .= '` SET ' . mysql_real_escape_string($_GET['champ']) . '=' .
 intval($_GET['valeur']);
        $sql .= ' WHERE id=' . intval($_GET['id']);
        break;

    default:
        exit;
}

// Exécution de la requête
mysql_query($sql) or die(mysql_error());

mysql_close();

?>
```

Si jamais l'un des paramètres est manquant (ce qui ne devrait jamais arriver par un appel du script) ou invalide, le script se termine. Une amélioration possible serait ici de gérer un retour d'erreur pour le script.

Ensuite selon le type du champ enregistré, une requête différente est construite. La différence réside ici dans l'ajout ou non de guillemets pour entourer la valeur d'un champ de type texte, qui est également formatée grâce à **mysql_real_escape_string**. Dans le cas d'un nombre, la valeur est passée telle quelle.

La requête est ensuite exécutée, et le script se termine.

II - Deuxième partie : Généralisation du script, emploi de classes

Comme annoncé en première partie, nous allons maintenant tenter d'améliorer un minimum ce script afin de le rendre plus modulable.

Vous trouverez une démonstration fonctionnelle de ce que nous allons réaliser maintenant à **cette adresse**.

Les fichiers sont téléchargeables **ici** (**miroir** en cas de problème avec le lien précédent).

II-1 - Les modifications apportées

II-1-a - Limites du script actuel

Le principal problème du script actuel est son manque de flexibilité. Afin de vous en convaincre, il suffit de se pencher sur la question : *que faut-il faire pour ajouter un type de champ ?*

A mes yeux, le principal défaut est qu'il est nécessaire de modifier le corps de la fonction **inlineMod** afin de fournir un nouveau type d'édition. Qui plus est, l'interactivité est relativement limitée puisqu'il n'y a création que d'un **input** dont les propriétés sont plus ou moins fixées en "dur" dans le script.

Bien sûr, il serait possible d'ajouter plus d'un **input**, voire d'autres éléments, en fonction du type de champ demandé, mais le code de la fonction **inlineMod** en deviendrait vite très chargé.

Dans le même esprit, comment gérer facilement les erreurs de saisie avec le script actuel ? Il faudrait utiliser un nouveau **switch** pour vérifier, selon le type du champ, que le texte entré représente bien un nombre, ou alors une adresse email, etc. Autant de code qui alourdirait la fonction **inlineMod**.

Il y a donc quelques modifications à opérer afin d'obtenir un script plus souple, modifications qui passeront principalement par une *décentralisation* du code de gestion des champs d'édition, et ce grâce à l'implémentation de classes.

II-1-b - Pourquoi des classes ?

A partir de ce point, je considère que vous possédez quelques notions de programmation objet, et que le concept de *classe* ou d'*objet* ne vous est pas étranger.

L'avantage d'utiliser des classes pour notre script est assez évident : une classe possède ses propres méthodes et ses propres propriétés, et peut éventuellement être instanciée plusieurs fois sans qu'il y ait besoin d'écrire plus d'une fois le code. Cet avantage est inutile pour notre exemple, mais il pourrait s'avérer intéressant dans d'autres applications du script.

Les classes nous apportent donc la modularité dont nous avons besoin. L'idée est de créer une classe pour chaque nouveau type de champ et de modifier la fonction **inlineMod** en conséquence, pour que, de manière transparente, elle instancie la classe correspondant au type de champ requis.

La classe permettra ensuite, grâce à quelques méthodes, de créer les contrôles appropriés, de vérifier la saisie et de valider l'envoi des informations pour les sauvegarder.

II-2 - Adaptation du code d'origine

Nous allons reprendre les fichiers déjà créés pour la première partie, et les adapter afin de permettre l'utilisation de nos futures classes.

II-2-a - Réorganisation des fichiers

En premier lieu, un minimum d'organisation s'impose. Chaque classe sera implémentée dans un fichier individuel ; les fichiers de classe seront nommés de manière homogène afin de pouvoir aisément les repérer et les inclure automatiquement dans le fichier HTML.

Afin d'éviter d'avoir trop de fichiers à la racine, j'ai pour ma part préféré stocker tous les scripts dans un sous-répertoire "scripts", mais libre à vous de créer l'aborescence qui vous convient.

Pour ne pas avoir à inclure chaque nouveau fichier de classe à la main, nous allons modifier le fichier **index.php** pour générer grâce à PHP les lignes d'inclusion de tous les scripts présents dans le répertoire "scripts" :

index.php - listing des scripts de classe

```
/// Listing des scripts JS disponibles
 ////////////////////////////////////////////////////////////
$scripts = array();
$i = 0;

foreach(glob('./scripts/inlinemod.class.*.js') as $fichier)
{
  $scripts[$i] = $fichier;
  $i++;
}
////////////////////////////////////////////////////////////////////////////////////////////////////
//
```

Ajoutez le code ci-dessus au tout début du document, après le code effectuant la requête MySQL par exemple. Vous devrez bien sûr adapter le code à votre arborescence.

Le processus est simple : la fonction **glob** liste tous les fichiers d'un répertoire avec le motif passé en paramètre. Utilisée avec l'itérateur **foreach**, elle permet d'ajouter chaque fichier nommé **inlinemod.class.NOM_CLASSE.js** au tableau **$scripts** qui est ensuite traité dans la section **head** :

index.php - génération des lignes d'inclusion

```
<head>

    (...)

    <script type="text/javascript" src="./scripts/utils.js"></script>

    <?php
        //Inclusion des fichiers javascript de classes
        foreach($scripts as $script)
        {
            print '<script type="text/javascript" src="' . $script . '"></script>';
```

index.php - génération des lignes d'inclusion

```
            }
        ?>

        <script type="text/javascript" src="./scripts/inlinemod.js"></script>
    </head>
```

Vous remarquerez la présence d'un script **utils.js**. J'y ai déporté les fonctions **getXMLHTTP**, **getKeyCode**, **trim**, **getTextWidth** et **ieTrick** afin d'alléger **inlinemod.js**

II-2-b - Modification de index.php

Outre les modifications relatives à la nouvelle organisation des fichiers javascript, il y a deux modifications à effectuer dans le fichier **index.php** pour la nouvelle version de ce sript.

Tout d'abord afin de prévoir un retour d'erreur de la part des objets javascript ou du script php, nous allons ajouter un élément **div** en dessous du titre de la page :

```
    <body>
        <h1>Liste d'utilisateurs</h1>

        <div id="erreur"></div>

        <br/>

        (...)
```

Ce div se verra appliqué le style suivant :

```
    div#erreur {
        color: #F00;
        text-align: center;
        font-weight: bold;
    }
```

Ainsi, si une erreur survient elle sera facilement repérable, puisqu'affichée en **rouge gras** au centre de la page.

L'autre modification de ce fichier, mineure, vise simplement à respecter les conventions de nommage Javascript. Plutôt que de prendre comme dernier paramètre le type de champ à créer, la fonction **inlineMod** recevra le nom de la classe qu'elle devra instancier. Ainsi, l'identificateur "texte-multi" n'est pas conforme à la manière de nommer des objets en Javascript, et il faut donc modifier la cellule *adresse* :

```
    <td class="cellule" ondblclick="inlineMod(<?php echo $user['id']; ?>, this, 'adresse',
     'TexteMulti')">
    <?php echo $user['adresse']; ?>
    </td>
```

La chaîne "texte-multi" a été modifiée en "TexteMulti", qui sera le nom de l'objet que nous créerons par la suite pour gérer les champs multilignes.

Dans la même idée, tous les "texte" et "nombre" doivent être modifiés en "Texte" et "Nombre".

II-2-c - Fonctionnalités des classes

Avant d'aller plus loin, il nous faut définir quelles seront les méthodes que chaque objet devra mettre à disposition du script. Afin de fournir les mêmes fonctionnalités (et plus) que le script précédent, chaque objet doit pouvoir :

- 1. Remplacer le texte affiché par un ou plusieurs contrôles d'édition
- 2. Donner le focus au contrôle d'édition principal
- 3. Fournir la valeur du champ à enregistrer
- 4. Terminer l'édition pour repasser à l'affichage des données
- 5. Identifier un problème dans la saisie effectuée

Pour les interactions internes des objets avec le script principal, il faut également pouvoir :

- 6. Connaître le nom du champ en base de données qui est édité
- 7. Connaître l'id en base de données de l'enregistrement à modifier
- 8. Savoir si, dans la requête SQL, la valeur du champ édité doit être "échappée" (utilisation de **mysql_real_escape_string**)

Basiquement, chacune de ces fonctionnalités correspondra à une méthode et/ou une propriété de nos objets. Voici pour chacune d'entre elles les noms que nous utiliserons :

- 1. **remplacerTexte(parent, sauvegarde)** où **parent** est l'élément parent de la donnée à éditer et **sauvegarde** la fonction à appeler pour lancer la sauvegarde de la valeur du champ
- 2. **activerChamp()**
- 3. **getValeur()**
- 4. **terminerEdition()**
- 5. **erreur()**, qui renvoie **true** si une erreur est trouvée, et qui se charge d'enregistrer un message d'erreur dans une propriété **texteErreur** de l'objet
- 6. **nomChamp**, propriété renseignée grâce à l'un des paramètres de la fonction **inlineMod**
- 7. **id**, propriété renseignée grâce à l'un des paramètres de la fonction **inlineMod**
- 8. **echapperValeur()**, qui renvoie **true** si la valeur doit être échappée dans la requête

Tout ceci étant défini, nous pouvons continuer nos modifications dans le script principal, puis nous implémenterons les classes correspondant à nos trois types de champs selon ce qui a été donné ci-dessus.

II-2-d - Modification d'inlinemod.js

II-2-d-i - La fonction inlineMod

Grâce au grand nombre de fonctionnalités déportées dans les classes, le code de la fonction **inlineMod** va pouvoir être largement allégé.

Un premier détail, modifions la liste des arguments pour refléter leur exacte utilité :

```
function inlineMod(id, obj, nomChamp, classe)
```

Le début du code reste inchangé : nous effectuons toujours des contrôles afin de ne pas lancer deux fois le mode d'édition :

```
function inlineMod(id, obj, nomChamp, classe)
{
    if(editionEnCours)
    {
        return false;
    }
    else
    {
        editionEnCours = true;
        sauve = false;
    }
```

Il faut maintenant créer l'objet correspondant au type de champ voulu, initialiser ses propriétés et appeler les méthodes qui créeront les contrôles d'édition :

```
    //Création de l'objet dont le nom de classe est passé en paramètre
    champ = eval('new ' + classe + '();');

    //Assignation des différentes propriétés
    champ.valeur = obj.innerText ? obj.innerText : obj.textContent;
    champ.valeur = trim(champ.valeur);

    champ.id = id;
    champ.nomChamp = nomChamp;

    //Remplacement du texte par notre objet input
    champ.remplacerTexte(obj, sauverMod);

    //"Activation" du champ (focus, sélection ou autres...)
    champ.activerChamp();
```

La variable **champ** est la variable **input** de la première partie ; elle a été renommée pour la simple raison qu'elle contiendra maintenant une référence sur un objet plutôt que directement sur un élément **input**.

Javascript nous fournit une fonction très utile pour remplir notre objectif : **eval**. Comme vous le savez sans doute, **eval** prend en paramètre du code javascript sous forme d'une chaîne de caractères, qu'elle se charge d'évaluer et d'exécuter. Ainsi, nous instancions la classe nécessaire à l'édition du champ de manière totalement transparente en faisant exécuter à **eval** le code *"new NOM_CLASSE();"*, où *NOM_CLASSE* est représenté par le paramètre **classe** de la fonction **inlineMod**.

Toutes les classes ayant les mêmes méthodes et propriétés, nous pouvons donc ensuite renseigner ce qu'il est nécessaire de renseigner et appeler les méthodes d'initialisation des contrôles d'édition.

Comme vous pouvez le voir, il n'y a plus aucun **switch** ou d'autres instructions faisant dépendre le code de tel ou tel type de champ. De plus, les événements **onkeydown** et **onblur** n'apparaissent plus ici : ce sera à chaque objet de définir son comportement et les conditions mettant fin à l'édition.

II-2-d-ii - La fonction sauverMod

La fonction **sauverMod** va subir un certain nombre de modifications. En premier lieu, elle ne prendra plus de paramètres et se basera sur les propriétés et les méthodes de **champ**.

Nous allons également ajouter un contrôle d'erreur basé soit sur la méthode **erreur()** qui affichera le contenu de la propriété **erreurTexte** si **True** est renvoyé, soit sur le retour du script PHP.

Les premières toutefois ne changent pas :

```
function sauverMod()
{
    //Si on a déjà sauvé la valeur en cours, on sort
    if(sauve)
    {
        return false;
    }
    else
    {
        sauve = true;
    }
```

Ici, nous insérons le contrôle d'erreur effectué par l'objet. Si jamais il y a une erreur de saisie, la sauvegarde ne doit pas continuer.

```
    //Vérification d'erreur
    if(champ.erreur())
    {
        document.getElementById("erreur").innerHTML = champ.texteErreur;
        sauve = false;
        return false;
    }
```

S'il y a une erreur, le texte de l'erreur est affiché dans le **div** prévu à cet effet.

Ensuite, les contrôles sur l'objet **XHR** et la création de l'objet **XMLHTTPRequest** sont effectués comme dans la première partie :

```
    //Si l'objet existe déjà on abandonne la requête et on le supprime
    if(XHR && XHR.readyState != 0)
    {
        XHR.abort();
        delete XHR;
    }

    //Création de l'objet XMLHTTPRequest
    XHR = getXMLHTTP();

    if(!XHR)
    {
        return false;
    }
```

A ce stade, il reste à envoyer la requête au serveur et à gérer la réponse du script. La requête est similaire à celle effectuée en première partie. Nous allons passer au script **sauverMod.php** l'id de l'enregistrement à modifier, le nom

du champ de cet enregistrement qui a été modifié, la valeur à lui affecter et une valeur spécifiant s'il faut "échapper" ou non cette valeur.

Toutes ces informations peuvent être connues grâce aux propriétés/méthodes de la classe instanciée :

```
    //URL du script de sauvegarde auquel on passe la requête à exécuter
    XHR.open("GET", "sauverMod.php?champ=" + escape(champ.nomChamp) + "&valeur=" +
escape(champ.getValeur())
            + "&echap=" + champ.echaperValeur() + "&id=" + champ.id + ieTrick(), true);
```

Pour terminer, nous allons intégrer une gestion d'erreur dans l'événement **onreadystatechange**, qui affichera un éventuel retour du script PHP :

```
    //On se sert de l'événement OnReadyStateChange pour supprimer l'input et le replacer par son
contenu
    XHR.onreadystatechange = function()
    {
        //Si le chargement est terminé
        if (XHR.readyState == 4)
            if(!XHR.responseText)
            {
                //Réinitialisation de la variable d'état d'édition
                editionEnCours = false;

                //Sortie du mode d'édition
                champ.terminerEdition();

                //Réinitialisation de l'affichage d'erreur
                document.getElementById("erreur").innerHTML = "";

                return true;
            }
            else //S'il y a une réponse texte, c'est une erreur PHP
            {
                //Affichage de l'erreur
                document.getElementById("erreur").innerHTML = XHR.responseText;
                sauve = false;
                return false;
            }
    }

    //Envoi de la requête
    XHR.send(null);
```

Comme vous pouvez le constater, le code est fondé sur le fait que si une réponse texte est présente, il s'agit nécessairement d'une erreur venant du script PHP. Ce sera effectivement le cas bien qu'il est vrai que cette méthode n'est pas des plus "propres". Libre à vous d'implémenter une gestion d'erreurs plus solide, ce serait s'éloigner du sujet de ce tutoriel d'y passer plus de temps ici.

II-2-e - Modification de sauverMod.php

Les changements du script de sauvegarde ne sont pas fondamentaux. Globalement, il s'agit juste d'adapter le code existant aux nouveaux paramètres qui sont passés en GET grâce à l'objet **XMLHTTPRequest**.

Puisque le nom du champ à modifier est fourni, il n'y aura pas besoin de **switch** pour générer une requête différente par type de champ ; la requête va pouvoir être construite directement à partir des paramètres envoyés au script.

Mais en premier lieu, des vérifications s'imposent :

```
<?php
    //On sort en cas de paramètre manquant ou invalide
    if(!isset($_GET['champ']) or empty($_GET['champ']) or !isset($_GET['valeur']) or
 (empty($_GET['valeur']) and
        ($_GET['valeur'] != 0)) or !isset($_GET['echap']) or empty($_GET['echap']) or
        !isset($_GET['id']))
    {
        print "Erreur dans les paramètres fournis";
        exit;
    }
```

Le changement est ici l'ajout d'un message d'erreur en retour. Cette erreur pourra être affichée dans la page s'il y a un problème avec les paramètres reçus par le script.

Viennent ensuite la connexion à la base de données, la construction de la requête et son exécution :

```
    //Connexion à la base de données
    mysql_connect(DB_HOST, DB_USER, DB_PASSWORD) or die("Erreur de connexion : " . mysql_error());
    mysql_select_db(DB_NAME) or die("Erreur BDD : " . mysql_error());

    //Construction de la requête
    $champ      = $_GET['champ'];
    $valeur     = $_GET['valeur'];
    $id         = $_GET['id'];

    $sql = "UPDATE `" . DB_TABLE_NAME . "` SET $champ=";

    //Il faut éventuellement formater la valeur fournie
    if($_GET['echap'] == "true")
    {
        $valeur = mysql_real_escape_string($valeur);
        $sql .= "'$valeur'";
    }
    else
        $sql .= $valeur;

    $sql .= " WHERE id=$id";

    //Exécution de la requête
    mysql_query($sql) or die("Erreur BDD : " . mysql_error());

    mysql_close($connexion);
?>
```

L'intégralité du script se trouve ci-dessus. Il est légèrement plus long que celui de la première partie, mais ne présente aucune difficulté supplémentaire. La requête est construite en plusieurs fois, au cas où un échappement serait nécessaire, puis est exécutée, et le script se termine.

II-3 - Implémentation des classes

Le script principal est maintenant entièrement prêt à utiliser des classes. Reste à les implémenter, ce à quoi nous allons nous atteler dans cette dernière partie.

Le code des trois classes correspondant aux types de champ dont nous avons besoin étant relativement similaires, nous ne détaillerons que la création de la première classe (champ de type texte), puis le code complet des autres classes sera donné avec quelques commentaires sur les modifications effectuées.

II-3-a - L'objet texte

La construction d'une classe en javascript se fait en deux étapes : tout d'abord la création du constructeur, où les propriétés de l'objet sont définies et initialisées, puis l'implémentation des méthodes de l'objet.

Voici donc pour commencer le constructeur de notre objet **texte** :

```
var input = null;

//Constructeur de l'objet
function Texte()
{
    this.id = -1;
    this.valeur = "";
    this.nomChamp = "";
    this.parent = null;
    this.texteErreur = "";
}
```

La variable globale **input** servira dans les méthodes de l'objet. Le constructeur prend le nom de la classe, c'est d'ailleurs lui qui définit le nom de l'objet.

Les propriétés définies sont celles qui ont déjà été annoncées précédemment, si ce n'est le parent qui sert en interne.

Voyons maintenant les méthodes. En premier lieu, la méthode **remplacerTexte**, qui permet de remplacer le texte affiché par les contrôles d'édition. Elle prend en paramètre l'élément **parent** qui contiendra les contrôles, et la fonction de **sauvegarde** à appeler lorsque le mode d'édition est quitté :

```
//Fonction de remplacement du texte de parent par le champ
Texte.prototype.remplacerTexte = function (parent, sauvegarde)
{
    if(!parent || !sauvegarde)
    {
        return false;
    }
    else
 {
        this.parent = parent;
    }

    input = document.createElement("input");

    input.value = this.valeur;
    input.style.width = getTextWidth(this.valeur) + 10 + "px";

    //Assignation des événements qui déclencheront la sauvegarde de la valeur
    //Sortie de l'input
    input.onblur = function ()
    {
        sauvegarde();
    };
```

```
        //Appui sur la touche Entrée
        input.onkeydown = function keyDown(event)
        {
            if((window.event && (getKeyCode(window.event) == 13)) || (getKeyCode(event) == 13))
            {
                sauvegarde.call();
            }
        };

        parent.replaceChild(input, parent.firstChild);
    }
```

Vous remarquerez que l'on retrouve globalement le même code que dans la première partie. Le champ de texte est créé, puis son style est modifié. Ensuite, les deux événements **onblur** et **onkeydown** sont initialisés pour permettre la sortie de l'édition.

La méthode **activerChamp()** permet ici de donner le focus à l'input :

```
//Fonction d'activation du champ
Texte.prototype.activerChamp = function ()
{
    input.focus();
    input.select();
}
```

La méthode **getValeur()** renvoie la valeur en cours du champ d'édition :

```
//Fonction permettant de récupérer la valeur du champ
Texte.prototype.getValeur = function ()
{
    return input.value;
}
```

La méthode **terminerEdition()** est appelée à la fin de l'enregistrement, pour revenir à l'affichage des données :

```
//Fonction de sortie du mode d'édition
Texte.prototype.terminerEdition = function ()
{
    this.parent.replaceChild(document.createTextNode(input.value), this.parent.firstChild);
    delete input;
}
```

La méthode **echapperValeur()**, ici, renvoie **true** pour signaler que la valeur à sauvegarder doit être échappée :

```
//Fonction déterminant si la valeur passée au script PHP doit être formatée par celui-ci ou pas
//Ici oui, car il s'agit de texte
Texte.prototype.echaperValeur = function ()
{
    return "true";
}
```

Enfin, la méthode **erreur()** renvoie **false** si le champ n'est pas vide, **true** sinon :

```
//Erreur si champ vide
Texte.prototype.erreur = function ()
{
    if(this.getValeur() == "")
    {
        this.texteErreur = "Aucune saisie effectuée !";
        return true;
 }
    else
        return false;
}
```

La propriété **texteErreur** est également renseignée pour que le message d'erreur puisse être affiché par le script princpal.

Comme vous pouvez le voir, le code n'est pas plus compliqué qu'auparavant, il n'est que découpé en plusieurs méthodes. Il existe bien sûr la contrainte du nombre fixe de méthodes, mais bien sûr si des fonctionnalités doivent être ajoutées, il n'y a aucune restriction pour compléter le script de base et les classes existantes en conséquence.

Enregistrez le fichier sous le nom **inlinemode.class.texte.js** dans le dossier de scripts.

II-3-b - L'objet nombre

L'objet **nombre** est identique à l'objet **texte**, si ce n'est que sa méthode **erreur()** renvoie **true** si le texte entré ne représente pas une valeur numérique.

```
var input = null;

//Constructeur de l'objet
function Nombre()
{
    this.id = -1;
    this.valeur = "";
    this.nomChamp = "";
    this.parent = null;
    this.texteErreur = "";
}

//Fonction de remplacement du texte de parent par le champ
Nombre.prototype.remplacerTexte = function (parent, sauvegarde)
{
    if(!parent || !sauvegarde)
    {
        return false;
    }
    else
    {
        this.parent = parent;
    }

    input = document.createElement("input");

    input.value = this.valeur;
    input.style.width = getTextWidth(this.valeur) + 10 + "px";

    //Assignation des événements qui déclencheront la sauvegarde de la valeur
    //Sortie de l'input
```

```
        input.onblur = function ()
        {
            sauvegarde();
        };

        //Appui sur la touche Entrée
        input.onkeydown = function keyDown(event)
        {
            if((window.event && (getKeyCode(window.event) == 13)) || (getKeyCode(event) == 13))
            {
                sauvegarde.call();
            }
        };

        parent.replaceChild(input, parent.firstChild);
    }

    //Fonction permettant de récupérer la valeur du champ
    Nombre.prototype.getValeur = function ()
    {
        return input.value;
    }

    //Fonction d'activation du champ
    Nombre.prototype.activerChamp = function ()
    {
        input.focus();
        input.select();
    }

    //Fonction de sortie du mode d'édition
    Nombre.prototype.terminerEdition = function ()
    {
        this.parent.replaceChild(document.createTextNode(input.value), this.parent.firstChild);
        delete input;
    }

    //Fonction déterminant si la valeur passée au script PHP doit être formatée par celui-ci ou pas
    //Ici non, car il s'agit de valeur numérique
    Nombre.prototype.echaperValeur = function ()
    {
        return "false";
    }

    //Si le texte entré ne représente pas un nombre, on renvoie true
    Nombre.prototype.erreur = function ()
    {
        if(isNaN(this.getValeur()))
        {
            this.texteErreur = "Vous devez entrer un chiffre !";
            return true;
        }
        else
            return false;
    }
```

Enregistrez le fichier sous le nom **inlinemode.class.nombre.js** dans le dossier de scripts.

II-3-c - L'objet texteMulti

Cette fois, le champ créé n'est plus un **input** mais un **textarea** pour pouvoir éditer du texte sur plusieurs lignes. Toujours pour l'édition multilignes, l'événement **onkeydown** a été supprimé afin de pouvoir retourner à la ligne en utilisant la touche entrée.

```
var textarea = null;

//Constructeur de l'objet
function TexteMulti()
{
    this.id = -1;
    this.valeur = "";
    this.nomChamp = "";
    this.parent = null;
    this.texteErreur = "";
}

//Fonction de remplacement du texte de parent par le champ
TexteMulti.prototype.remplacerTexte = function (parent, sauvegarde)
{
    if(!parent || !sauvegarde)
    {
        return false;
    }
    else
 {
        this.parent = parent;
    }

    textarea = document.createElement("textarea");

    textarea.value = this.valeur;
    textarea.style.width = getTextWidth(this.valeur) + 30 + "px";

    //Assignation des événements qui déclencheront la sauvegarde de la valeur
    //Sortie du textarea
    textarea.onblur = function ()
    {
        sauvegarde.call();
    };

    parent.replaceChild(textarea, parent.firstChild);
}

//Fonction permettant de récupérer la valeur du champ
TexteMulti.prototype.getValeur = function ()
{
    return textarea.value;
}

//Fonction d'activation du champ
TexteMulti.prototype.activerChamp = function ()
{
    textarea.focus();
    textarea.select();
}

//Fonction de sortie du mode d'édition
TexteMulti.prototype.terminerEdition = function ()
{
    this.parent.replaceChild(document.createTextNode(textarea.value), this.parent.firstChild);
    delete textarea;
}

//Fonction déterminant si la valeur passée au script PHP doit être formatée par celui-ci ou pas
//Ici oui, car il s'agit de texte
TexteMulti.prototype.echaperValeur = function ()
{
    return "true";
}

//Erreur si champ vide
```

```
TexteMulti.prototype.erreur = function ()
{
    if(this.getValeur() == "")
    {
        this.texteErreur = "Aucune saisie effectuée !";
        return true;
    }
    else
        return false;
}
```

Enregistrez le fichier sous le nom **inlinemode.class.texteMulti.js** dans le dossier de scripts.

Conclusion

Ce tutoriel est maintenant entièrement terminé. La deuxième partie a permis de construire un script plus solide encore, et plus modulaire, permettant d'ajouter facilement de nouveaux types de champs.

Le script présente bien sûr toujours quelques limites (l'ajout de champ avec liste déroulante n'est par exemple pas immédiat) mais cela vous donnera sans doute une bonne base pour développer votre propre script.

Dans tous les cas j'espère que ce tutoriel vous a été utile ; si vous avez la moindre remarque ou question concernant le contenu de ce tutoriel, n'hésitez pas à me contacter **par mp** !

Pour terminer, j'aimerais remercier l'équipe Web pour leur aide et leurs conseils, et plus particulièrement **Yogui**, **DenisC**, **BrYs** et **Arnolem** !

Entre la publication de la première partie et de la seconde partie de cet article, j'ai reçu un très grand nombre de retours de la part de différents lecteurs, pour me signaler des erreurs ou de simples remerciements. C'est à mon tour de vous remercier pour ces feedbacks, et de m'excuser pour avoir pris tant de temps à publier cette seconde partie ! ;)

Liens

- **Introduction à AJAX et interaction avec PHP**, de **Gaël Donat**.
- **, allez plus loin avec** , de **François Dussert**. Familiarisez-vous avec l'objet XMLHTTPRequest.
- **Une autocomplétion pas à pas**, de **Denis Cabasson**. Une autre application très utile d'AJAX.

Addenda

Suite à quelques retours, voici des notes supplémentaires concernant ce tutoriel ou les codes qu'il contient.

Constantes utilisées pour les accès BDD

Dans les différents codes PHP, j'utilise les constantes DB_HOST, DB_USER, DB_PASSWORD et DB_TABLE_NAME, sans précéder leurs noms du dollar conventionnel.

Il s'agit en réalité de constantes définies de la manière suivante :

```
define('DB_HOST',       'SERVEUR_MYSQL');
define('DB_USER',       'UTILISATEUR');
define('DB_PASSWORD',   'MOT_DE_PASSE');
define('DB_NAME',       'NOM_TABLE');
define('DB_TABLE_NAME', 'inlinemod');
```

Vous avez pu apercevoir ce code si vous avez téléchargé le zip du script final. Ces déclarations sont à faire dans un fichier de configuration, par exemple "config.php", qu'il faut inclure au début de vos fichiers **index.php** et **sauverMod.php**. Merci à **jujudellago** pour m'avoir fait prendre conscience de cet oubli.