Asynchronous **J**avaScript **A**nd **X**ML

→ Ce PDF aura pour but de vous apprendre ce qu'est **l'AJAX** et ce qu'il vous permet de faire.

Public visé :
→ Développeur Web

Auteur :
→ Spl3en

Fait le :
→ 19/07/2010

Version :
→ 1.1

Changelog :
19/07/2010 :
- Corrections orthographique et typographique

Sommaire

I) Introduction : Ajax, a.k.a. socket du web !

Commençons par introduction ce qu'est l'**AJAX.**
Citons notre ami Wikipédia qui nous donne une définition stricte de ce que c'est :

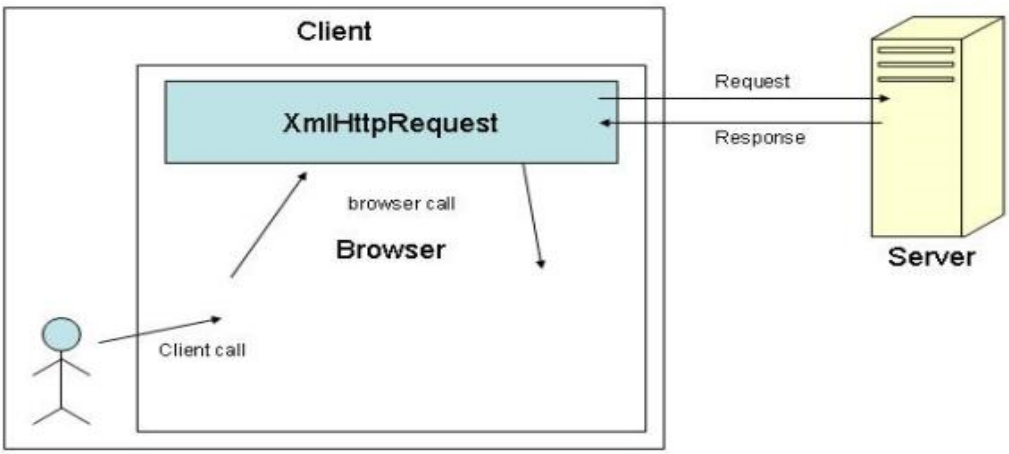
"

> **Ajax** est un acronyme pour Asynchronous JavaScript *and* XML (« XML et Javascript asynchrones ») et désignant une solution informatique libre pour le développement de pages dynamiques et d'applications Web.
>
> À l'image de DHTML ou de LAMP, AJAX n'est pas une technologie en elle-même, mais un terme qui évoque l'utilisation conjointe d'un ensemble de technologies libres couramment utilisées sur le Web :
>
> - HTML (ou XHTML) pour la structure sémantique des informations ;
> - CSS pour la présentation des informations ;
> - DOM et JavaScript pour afficher et interagir dynamiquement avec l'information présentée ;
> - l'objet XMLHttpRequest pour échanger et manipuler les données de manière asynchrone avec le serveur Web.
> - XML pour remplacer le format des données informatives (JSON) et visuelles (HTML).
>
> En alternative au format XML, les applications Ajax peuvent utiliser les fichiers texte ou JSON.
>
> Les applications Ajax peuvent être utilisées au sein des navigateurs Web qui supportent les technologies décrites précédemment. Parmi eux, on trouve Mozilla Firefox, Internet Explorer, Konqueror, Google Chrome, Safari et Opera.

En d'autres termes, cette technologie va nous servir à rapatrier *côté client* des données stockées *côté serveur**, et cela sans **rafraîchir la page.**
Certains osent parler d'Ajax en tant que « socket du web » ;
L'idée est là : On cherche à se connecter via Javascript à un script distant afin d'enrichir le contenu de la page.

En lisant la définition ci-dessus, vous aurez compris que si vous savez coder en JS/HTML, vous savez faire de l'AJAX.

Commençons donc doucement par coder un petit peu.



Petit shéma simplifié de ce qu'est AJAX

II) L'objet XmlHttpRequest

L'XmlHttpRequest (souvent abrégé XHR) est, si l'on voudrait continuer notre comparaison, la socket utile à la connexion.
Nous allons nous en servir pour dialoguer entre le client et le serveur.
Il faut tout d'abord savoir qu'il existe une différentiation entre cet objet côté IE et pour les autres navigateurs :
Internet Explorer utilise un composant ActiveX pour utiliser Ajax, alors que les autres navigateurs utilise une XmlHttpRequest simple. Il faudra donc bien y faire attention lorsque l'on voudra l'initiatliser.

Voici une fonction assez générique qui vous permettra, quelque soit le navigateur qui le supporte, d'obtenir un XHR :

```
/**
*    Permet de récupérer un objet XHR selon tout type de navigateur
*    @return : XHR object ou null en cas d'échec
*/
function getXhr()
{
    if (window.XMLHttpRequest)
        return new XMLHttpRequest();

    if (window.ActiveXObject)
    {
        var IeXhr =
        [
            "Msxml2.XMLHTTP.6.0",
            "Msxml2.XMLHTTP.3.0",
            "Msxml2.XMLHTTP",
            "Microsoft.XMLHTTP"
        ];

        for (var i in IeXhr)
        {
            try
            {
                  return new ActiveXObject(IeXhr[i]);
            }

            catch(e)
            {
            }
        }
    }

     // XHR non supporté
    return null;
}

xhr = getXhr();
```

III) Ouvrir une connexion

L'ouverture d'une connexion se fait avec la méthode « open ».
Généralement, on utilise comme méthodes de requête HTTP GET ou POST, mais il faut savoir que HEAD, PUT, DELETE, OPTIONS peuvent être utilisés selon les navigateurs.
Le prototype de open est le suivant : (merci le msdn)

Syntax

object.open(sMethod, sUrl [, bAsync] [, sUser] [, sPassword])

Parameters

sMethod	Required. **String** that specifies the HTTP method used to open the connexion: such as GET, POST, or HEAD. This parameter is not case-sensitive.
sUrl	Required. **String** that specifies either the absolute or a relative URL of the XML data or server-side XML Web services.
bAsync	Optional. **Variant** that specifies true for asynchronous operation (the call returns immediately), or false for synchronous operation. If true, assign a callback handler to the onreadystatechange property to determine when the call has completed. If not specified, the default is true. **Performance Note** When *bAsync* is set to false, send operations are synchronous, and Windows Internet Explorer does not accept input or produce output while **send** operations are in progress. Therefore, this setting should not be used in situations where it is possible for a user to be waiting on the **send** operation to complete.
sUser	Optional. **Variant** that specifies the name of the user for authentication. If this parameter is null ("") or missing and the site requires authentication, the component displays a logon window.
sPassword	Optional. **Variant** that specifies the password for authentication. This parameter is ignored if the user parameter is null ("") or missing.

Dans des termes français, ça donne :

- En premier paramètre, la méthode HTTP utilisée,
- En deuxième paramètre, le script côté serveur visé.

Les trois autres paramètres sont optionnels, mais l'un d'entre eux est très intéressant : bAsync, booléen déterminant si la connexion sera **synchrone ou asynchrone.**

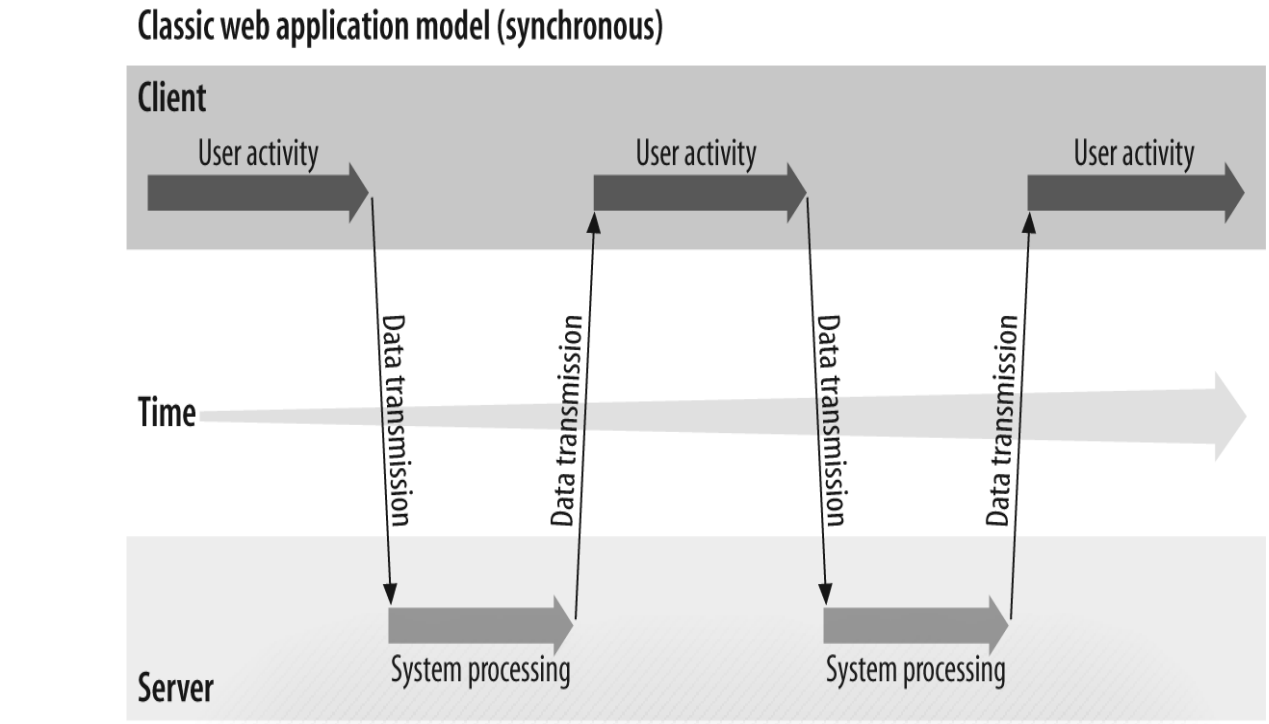
Quelle différence ?

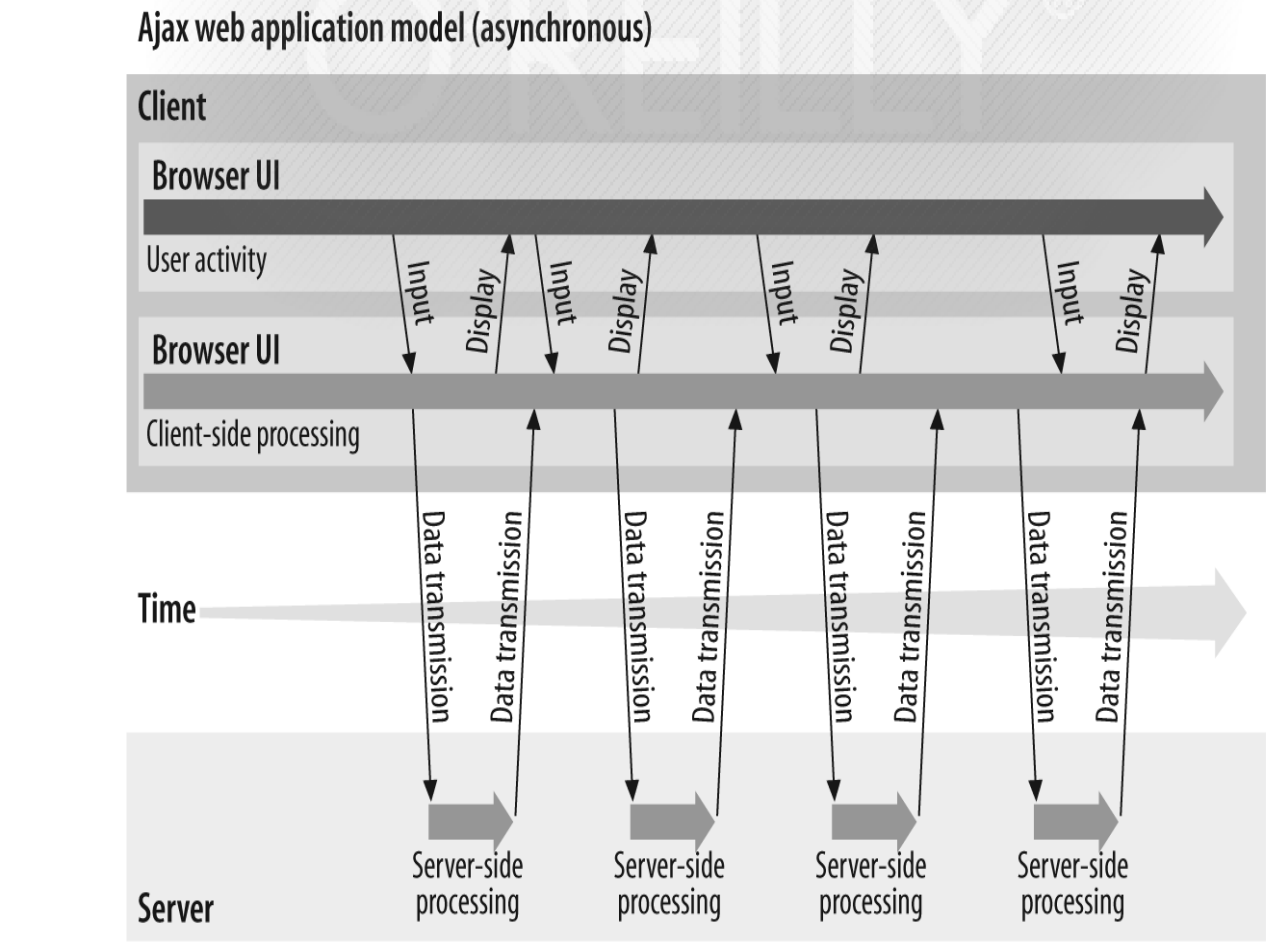
Si l'on voulait reprendre notre comparaison avec les sockets, on pourrait comparer le mode synchrone au mode bloquant, et le mode asynchrone au mode non bloquant ;

- En mode synchrone, le script ne continuera pas tant qu'aucune réponse du serveur ne sera reçue.
Cela a le fort désavantage de « freezer » le navigateur en attendant la réponse du serveur.

- En mode asynchrone, le script n'attendra aucune réponse du serveur pour continuer, il se contentera d'envoyer les données.
C'est une autre fonction que nous verrons par la suite, appelée lors de la réception des données, qui prendra le relai pour traiter la fin des instructions.

Nous utiliserons le mode **asynchrone** qui permet de garder une fluidité et une liberté d'action plus grande que le synchrone ; Une approche évènementielle est plus agréable que procédurale à mon avis.





AJAX … Une économie côté serveur aussi !

Finissons ce chapitre par un petit bout de code afin de voir comment ça marche en pratique :

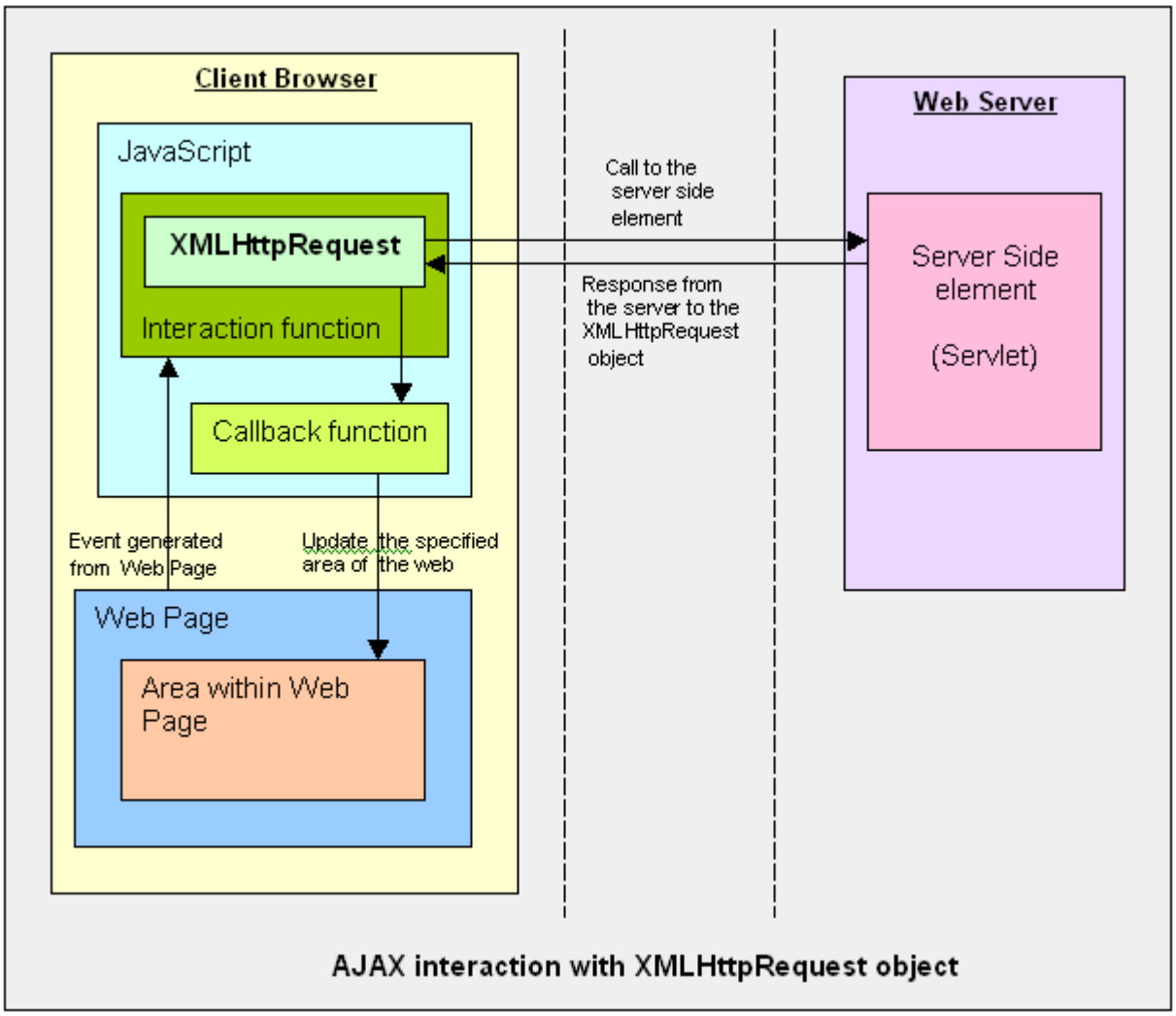
```
function doconnexion()
{
    var xhr = getXhr ();

    var async = true;
    var httpMethod = "get";
    var target = "script.php";

    xhr.open
    (
        httpMethod,
        target,
        async
    );
}
```

Voilà, notre connexion est faite.

Passons maintenant à l'envoi de données ☺



Une modélisation plus complexe d'AJAX

IV) L'envoi de données

Nous traiterons ici que l'envoi de données en GET et en POST, les autres cas étant assez particuliers.

Débutons par le plus simple : **GET**.
Ici, nullement besoin d'une fonction particulière !

Rappelez vous de la fonction open : Son deuxième argument est le script visé.

Mais rien ne vous empêche de faire passer des variables en argument, comme vous le feriez avec des URLs.

```
    var xhr = getXhr ();

    var async = true;
    var httpMethod = "post";
    var target = "script.php?variable1=a&variable2=b";

    xhr.open
    (
        httpMethod,
        target,
        async
    );
```

Et voilà, notre script.php a reçu deux arguments, variable1 et variable2, récupérables dans son array $_GET.
Cette méthode a l'avantage d'être plutôt simple à utiliser.

Néanmoins, lorsque l'on a besoin de faire passer de plus grosse données, il va y avoir un souci...

C'est pourquoi en général on préfèrera la méthode POST.
On aura besoin dans ce cas ci d'une fonction supplémentaire : send.

Nous aurons aussi besoin de changer le contenu de Content-Type dans le header de la requête HTTP via setRequestHeader, afin que notre requête ne soit pas mal interprétée par le serveur.

Voici ce que ça donne en pratique :

```
function sendPostData (data)
{
    var xhr = getXhr ();

    var async = true;
    var httpMethod = "post";
    var target = "script.php";

    xhr.open
    (
        httpMethod,
        target,
        async
    );


    // On n'oublie pas de changer le Content-type
    xhr.setRequestHeader
    (
        'Content-Type',
        'application/x-www-form-urlencoded;charset=ISO-8859-15'
    );

    // On envoie les données URLencodées via encodeURIComponent.
    xhr.send
    (
        'data='+encodeURIComponent(data)
    );
}
```

On récupérera naturellement nos données dans l'array $_POST['data'] dans notre script PHP.

Rien de bien compliqué je pense jusque là ! ☺
Maintenant que l'on sait envoyer de données, il serait bien de connaître comment en recevoir n'est-ce pas ?
Et bien ça tombe bien, puisque c'est notre prochain chapitre.

V) La réception des données

Reprenons notre objet XHR.

Il contient un évènement, « onreadystatechange », auquel on peut associer une fonction.
Cette fonction sera appelée à chaque fois que l'évenement onreadystatechange se provoquera.
Pour utiliser des termes simples, à chaque fois que quelque chose se passera sur votre XHR, une fonction que vous aurez définie sera appelée.
En général, cette fonction callback est utile pour la réception des données.

De plus, l'objet XHR contient un entier « **readyState** », allant de 1 à 4, qui nous indiquera l'état de la XHR à tout moment.
L'état qui nous intéressera ici sera le readyState = 4 : Lorsque la XHR indique qu'elle a reçue quelque chose.
Nous pouvons lire à ce moment cette réponse dans l'attribut « responseText » de la XHR.

Oui, pas évident à saisir en lisant tout cela d'un coup, mais rassurez avec un exemple, les choses iront mieux :)

```
function sendPostData (data)
{
    var xhr = getXhr ();

    var async = true;
    var httpMethod = "post";
    var target = "script.php";

    // On récupère notre XHR puis on ouvre la connexion comme on sait déjà faire
    xhr.open
    (
        httpMethod,
        target,
        async
    );

    xhr.setRequestHeader
    (
        'Content-Type',
        'application/x-www-form-urlencoded;charset=ISO-8859-15'
    );

    xhr.send
    (
        'data='+encodeURIComponent(data)
    );

    // C'est cette fonction qui sera appelée lors de la réception des données.
    xhr.onreadystatechange = function onReceiveData()
    {
        // Si la XHR indique qu'il y a une réponse reçue
        if (xhr.readyState == 4)
        {
            // On affiche le contenu de cette réponse.
            alert(xhr.responseText);
        }
    }
}
```

On aura donc un joli alert de ce que le script.php voudra bien nous renvoyer.
Imaginons que le contenu de script.php soit :
`<?php echo htmlentities('Hello '. $_POST['data']); ?>`
notre alert renverra donc un 'Hello '+ ce qu'on aura passé en paramètre de sendPostData.

VI) Un exemple : Le formulaire d'identification.

Nous allons illustrer ce que nous savons par un petit exemple afin de clarifier le tout.

Commencons par nous poser la problématique :
Nous voulons un formulaire avec un champ « password », qui viendra demander au serveur si le mot de passe envoyé est correct ou non.
Bien évidemment … Cette vérification se faisant en AJAX :)

Réfléchissons aux avantages :
- Dans un premier temps, l'utilisateur aura tout le **confort** de voir si son mot de passe est correct ou non en un quart de seconde, sans attendre un lourd refresh de la page qui peut coûter de longues secondes pour les plus longues connexions.

- Nous verrons qu'AJAX permet de produire du code plus **propre** et beaucoup plus lisible avec un peu de rigueur, en séparant distinctement l'HTML, le JS, et le PHP côté serveur.
En effet, on rencontre très fréquemment le problème de mélanger du code HTML en plein milieu d'un script PHP, ce qui au final donne un mélange pas franchement joli... Et bien on va pouvoir s'en passer en renforcer l'esprit client-serveur !

Et bien... c'est parti :)

En premier lieu, construisons notre client, le formulaire HTML.
Je ne vais pas utiliser de ''form'' ici, c'est pas forcément très respectueux des standards mais ça simplifiera les choses pour mon exemple.

Client.html

```
<html>

<head>

  <script type='text/javascript'
          src='client.js'>
  </script>

</head>

<body>

  Mot de passe :

  <input id='password_input'
          type= 'password'/>

  <input type='button'
         onClick='checkPassword()'/>

  <div id='reponse_serveur'>
  </div>

</body>

</html>
```

Remarquez le div 'reponse_serveur' en fin de code qui va nous servir à accueillir la réponse du serveur.

Côté Javascript, il va falloir construire le script en 3 étapes :
- La fonction checkPassword qui se chargera de **récupérer** le password du client, et éventuellement le filtrer.
- La fonction sendAjaxData et onReceiveDataCallback qui va **envoyer** et **recevoir** les données, comme nous l'avons vu précédemment,
- La fonction displayAnswer qui s'occupera de **l'affichage** sur la page de la réponse du serveur

On essaye de cette manière de bien dissocier l'affichage du traitement de données.

Tout d'abord notre checkPassword() :

```
function checkPassword ()
{
    var password = document.getElementById('password_input').value;
    var script = 'serveur.php';
    var callbackFunction =  displayAnswer;

    sendAjaxData(password, script, callbackFunction);
}
```

On récupère notre password via la fonction getElementById, on indique que le script visé sera « serveur.php », et on finit par déclarer que notre fonction callback sera displayAnswer.

Passons maintenant à notre fonction **sendAjaxData**.
Pour qu'on puisse la réutiliser un peu partout dans nos scripts, on va chercher à la rendre le plus générique possible.
On gardera notre fonction getXhr(), je ne la réecris pas ici.

```
function sendAjaxData (data, target, callbackFunction)
{
    var xhr = getXhr();

    xhr.open ('post', target, true);
    xhr.setRequestHeader ('Content-Type', 'application/x-www-form-urlencoded;charset=ISO-
8859-15');

    xhr.send('ajaxData=' + encodeURIComponent(data));

    xhr.onreadystatechange = function onReceiveData()
    {
        if (xhr.readyState == 4)
        {
            callbackFunction (xhr.responseText);
        }
    }
}
```

Rien de bien nouveau dans cette fonction n'est-ce pas ?

Passons donc à notre affichage, la fonction displayAnswer.

```
function displayAnswer (serverAnswer)
{
    var answerDiv = document.getElementById('reponse_serveur');
    var msg = '';

    switch (serverAnswer)
    {
        case '0':
            msg = 'Password invalide, veuillez recommencer !';
            break;

        case '1':
            msg = 'Password valide ! Bienvenue :)';
            break;
    }

    answerDiv.innerHTML = msg;
}
```

Le script parle ici de lui même :
Si la réponse du serveur est 0, on considérera que le mot de passe est incorrect, donc on enverra gentiment sur les roses l'utilisateur. Dans le cas où le serveur renvoie 1, on affiche un message de bienvenue.

Bien évidemment, rien n'empêcherait que le serveur renvoie lui même directement le message à afficher.
Mais c'est une attitude à éviter !
De cette manière, vous ne faîtes que faire transiter des informations inutiles, et difficilement traitable.
Retenez que **tout** ce qui est statique doit être côté Javascript (message d'erreur, scripts, affichage), et que seul le contenu dynamique et protégé doit être envoyé au client.
Gardez cette logique au maximum, vous maintiendrez ainsi une logique client – serveur rendant votre code bien plus propre et simple à entretenir en général.

On finit doucement par le script PHP côté serveur, afin de finaliser notre exemple.
Ici, on considérera que le mot de passe est en dur dans le code pour simplifier notre code.

```
<?php

    $data = $_POST['ajaxData'];
    $password = 'HelloWorld';

    if ($data == $password)
        echo '1';

    else
        echo '0';

?>
```

Évidemment, dans un cas concret, il aurait fallu stocker le mot de passe dans un fichier ou une base de données, voir comparer le hash du password avec le hash de ce qu'a entré l'utilisateur …
Mais ceci est une autre histoire :)

VII) L'envoi de données complexes

Viendra un moment où vous aurez besoin d'envoyer un ensemble de données complexes, telles qu'un tableau de données par exemple.
Deux choix s'offrent alors à vous :

- Construire votre propre format de données (ce qui est peu recommandable)
- Utiliser un format déjà existant.

Quel genre de format existe déjà ?

Comme l'indique le X de AJAX, nous avons à disposition l'échange de données client-serveur via XML.
Néanmoins, je vais plutôt vous indexer vers JSON, qui est un format remplissant le même rôle.
En effet, JSON a l'avantage d'être plus léger, et dispose surtout d'un support excellent en Javascript autant qu'en PHP... Et se trouve être plus simple à utiliser pour un néophyte.
Comprenez par là que les fonctions de conversion d'un format de données quelconque vers JSON se fait très simplement, et c'est justement ce qui va nous intéresser ici.

Voyons les fonctions en question :

Côté PHP :
- json_encode qui permet de convertir un format de données vers JSON.

- json_decode qui permet … de faire l'inverse ☺

Si vous passez *true* au deuxième argument de cette fonction, l'objet retourné est un array associatif, tout ce qu'il y a de plus classique en PHP.
Pour ces deux fonctions là, je vous invite à lire la documentation PHP … Plus explicite tu meurs ;)

Côté Javascript :

C'est à peine plus difficile... Il suffit d'eval la variable JSON reçue !
Si vous avez un problème avec eval (et ça peut se comprendre), il existe des solutions disponibles sur le net pour ne pas l'utiliser.
Pour encoder, il existe aussi des méthodes...
Je vous redirige sur ce lien pour tout ça, il n'y a pas grand chose à expliquer mais plus à appliquer :
http://www.openjs.com/scripts/data/json_encode.php

Le mot de la fin

Ce tutorial s'achève ici.
Je vous invite à vous lire un peu de documentation supplémentaire quand aux diverses fonctions que j'ai décrites, notamment l'objet XHR ; Vous y trouverez certaines propriétés que je n'ai pas détaillées mais qui pourraient vous être utiles dans certains cas particuliers.

N'hésitez pas à mon contacter pour toutes suggestions, erreurs qui se seraient glissées dans ce PDF ou autre.