Secteur Tertiaire Informatique
Filière Production Exploitation Administration

Développer une application Web

PROGRAMMER DES PAGES CLIENT RICHE AJAX

Accueil

Apprentissage

Période en entreprise

Evaluation

SOMMAIRE

Objectifs

Ce petit document a comme objectif de donner un éclairage sur l'utilisation d'AJAX. Le fonctionnement d'Ajax et l'utilisation de l'objet **XmlHttpRequest** sont présentés ; quelques cas d'utilisation standard sont détaillés et pourront être réalisés à titre d'exercices. Les codes sources associés sont donnés.

Ajax ou pas Ajax ?

Les appels AJAX (**Asynchronous JavaScript & XML**) permettent d'ajouter une fonctionnalité fondamentale aux pages Web d'aujourd'hui.

Au-delà de cette capacité à rafraîchir une portion de la page Web affichée par un navigateur, Ajax offre des fonctionnalités diverses liées à ses principes :

- Requêtes HTTP asynchrones (le client Web envoie la requête et passe à autre chose en attendant la réponse) ;
- Transfert de données au format non forcément HTML (XML, **JSON**) mieux adaptés aux traitements des serveurs ;
- Traitements côté client écrits en JavaScript.

Enfin, la prolifération de bibliothèques et Framework Ajax offrant des fonctionnalités toujours plus complètes et simples à mettre en œuvre, font **d'Ajax** une technique incontournable pour la réalisation de sites professionnels, dynamiques et ergonomiques.

AJAX

I.1 Présentation d'Ajax

Le fonctionnement des applications Web est basé sur *l'envoi au serveur Web, par le navigateur, d'une requête HTTP* ; le traitement du *serveur adresse alors une réponse HTTP contenant la page HTML* retrouvée sur disque ou générée à la volée. Lors d'un rafraîchissement de page demandé par le navigateur, c'est donc toute la page qui est reconstituée et renvoyée au navigateur.

Ceci n'est pas très performant, en terme de temps de réponse et de trafic réseau. En effet, une simple modification d'une petite partie de notre page web se traduira par la reconstitution et l'envoi de *l'ensemble* de la page.

Ajax permet de s'affranchir de ce concept de requête-réponse *en mode page*, et permet à tout moment d'aller chercher des informations sur le serveur, via requêtes et réponses HTTP, pour les *ajouter dans la page en cours*, ou *modifier des parties de la page courante*, et tout *cela sans avoir à recharger complètement la page*.

Ajax signifie **Asynchronous Javascript And XML**, et permet donc de faire des appels ***asynchrones*** au serveur depuis le client, et de récupérer, ceci de façon historique, un message au format XML renvoyé par le serveur. Aujourd'hui, c'est le format **JSON** qui est largement utilisé pour les échanges de données entre le client et le serveur.

Ajax est basé sur l'utilisation d'un composant particulier : l'objet **XmlHttpRequest** . Cet objet va permettre des échanges au format Xml mais aussi (HTML, Texte, **JSON**) entre le client et le serveur via des requêtes HTTP, qui seront adressées au serveur par le navigateur via des scripts JavaScript.

I.2 L'objet XmlHttpRequest

Cet objet est au cœur du fonctionnement d'Ajax. On va *l'instancier* dans chaque script nécessitant des échanges Ajax. Cet objet va pouvoir *envoyer une requête*, grâce à sa méthode ***open*** (en mode **synchrone** ou **asynchrone**, en utilisant la méthode HTTP **get** ou **post**, et en passant ou non des paramètres).
On va aussi pouvoir *intercepter les changements d'état de cet objet* (propriété ***onreadystatechange***) et l'on pourra ainsi *exécuter le traitement que l'on veut lorsque le traitement serveur sera terminé* (propriété ***readyState = 4***).

A l'origine, la création de l'objet **XmlHttpRequest** se faisait comme suit car Internet Explorer n'avait pas adopté le même objet que ses concurrents :

```
// on crée tout d'abord l'objet XMLHttpRequest
var xhr=null;
if (window.XMLHttpRequest) // Firefox et autres
      xhr = new XMLHttpRequest();
else if (window.ActiveXObject) // Internet Explorer
```

```
    {
        try
                {   xhr = new ActiveXObject("Msxml2.XMLHTTP");    } //IE
version <5
        catch (e)
                {   xhr = new ActiveXObject("Microsoft.XMLHTTP");  } // IE
version >=5
     }
```

Depuis Internet Explorer version 7, Microsoft s'est rallié au standard de fait et ne nécessite plus l'instanciation de son objet ActiveX spécifique. Le code peut devenir alors, pour tous navigateurs :

```
// on crée tout d'abord l'objet XMLHttpRequest
var xhr=null;

try
{   xhr = new XMLHttpRequest();} //tout navigateur" recent"
catch (e)
{… gestion de l'erreur … } // anciens navigateurs : soit refus, soit
traitements dégradés sans Ajax
```

Les différentes méthodes et propriétés de l'objet **XmlHttpRequest** sont les suivantes :

Propriété/Méthode	Description
Méthode ***open(méthode, url, indic)***	Cette méthode initialise une requête au serveur. • Le paramètre ***méthode*** vaut "get" ou "post" : c'est la méthode HTTP d'envoi de la requête. Si l'on est en méthode "get", les paramètres seront passés directement dans l'url. Attention : si l'on est en méthode "post", il faut aussi exécuter la méthode suivante de l'objet XmlHttpRequest : ***setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')***; • Le paramètre ***url*** contient l'adresse du script serveur que l'on veut exécuter (c'est un script jsp, aspx, php…). Ce pourra être par exemple le script serveur qui va générer le 'bout de page HTML' qui va être rafraîchi. Si l'on est en méthode "get", les paramètres doivent être passés dans cette url. • Le paramètre ***indic*** sert à préciser si l'on est en mode **synchrone** (indic=**false**) ou **asynchrone** (indic=**true**). En mode synchrone, le navigateur attend la réponse du serveur avant de rendre la main à l'internaute. Il est donc préférable d'utiliser le mode ***asynchrone***.

Méthode ***send(paramètres)***	Cette méthode permet l'envoi effectif de la requête au serveur. Si la méthode d'envoi est "**get**", la chaîne de paramètres passée vaut **null**. Si l'on est en méthode "**post**", la chaîne paramètres contient les paramètres sous la forme *nom=valeur ;* `ajax.send("nom="+ encodeURIComponent(nom) + "&prenom=" + encodeURIComponent(prenom));` La méthode *encodeURIComponent* garantit ici des valeurs nom et prenom sans caractères réservés.
Evenement ***onreadystatechange***	Evènement qui se déclenche lors de chaque changement d'état de l'objet **XmlHttpRequest**. Les différents états possibles sont donnés par la propriété suivante ***readyState***. Cet événement sera notamment utilisé pour exécuter le traitement désiré lorsque le traitement serveur sera terminé (mode asynchrone oblige). Ceci se fera en appelant une fonction spécifique comme suit : ***xhr.onreadystatechange = function() {…}***
Propriété ***readyState***	Cette propriété permet de connaître l'état du traitement de la requête. Les différentes valeurs possibles sont : • ***1*** : la méthode **open** vient de s'exécuter. La requête est préparée. • ***2*** : la méthode **send** vient de s'exécuter. La requête est envoyée • ***3*** : le traitement serveur s'exécute • ***4*** : Le **traitement serveur** est **terminé**, et les données ont été retournées. C'est cet état qui nous intéressera en général pour déclencher le rafraîchissement du 'bout de page HTML'.
Propriétés ***responseText*** ou **responseXML**	Propriété dans laquelle on va récupérer la réponse du serveur, soit au format Texte, soit au format XML …
Propriétés ***status*** et ***statusText***	Permet de récupérer le status de la réponse HTTP du serveur (200 = OK).

Quelques précisions utiles concernant cet objet **XmlHttpRequest** :

- Le fonctionnement de base est bien en *mode asynchrone*, c'est-à-dire que le navigateur Web exprime sa requête mais n'attend pas la réponse pour continuer

son travail ; il reste toutefois possible de programmer des traitements Ajax en mode synchrone ;

- En conséquence, on devra *placer une fonction JavaScript sur le changement d'état de l'objet **XmlHttpRequest*** de manière à réagir quand le traitement serveur est terminé, c'est-à-dire lorsque l'on a reçu la réponse HTTP ;
- Cette fonction est donc déclenchée à chaque changement d'état de l'objet **XmlHttpRequest** (**5 états successifs**, voir la doc de référence) ; *on déclenchera le rafraîchissement de l'affichage au moment où l'état vaut 4* ;
- En JavaScript, on peut définir une fonction, en extension, sans la nommer, « à la volée » comme dans ce code :
-

```
xhr.onreadystatechange = function() {
        if (xhr.readyState==4)
            document.getElementById('erreurmail').innerHTML = xhr.responseText;
}
```

I.3 Etude de cas

Mise en pratique à travers la présentation **d'Introduction à Ajax** (**JavaScript**).

Un champ de formulaire de type input texte permet de saisir un nom de pays.

A chaque fois que l'utilisateur fait la saisie d'un nouveau caractère dans cet input, une requête Ajax est envoyée au serveur sur le fichier **pays.json** qui renvoi la liste des pays concernés par cette saisie partielle. Cette liste est ensuite récupérée pour alimenter le contenu d'une liste déroulante de la page HTML sans qu'à aucun moment cette page ne soit rechargée.


```
<body>
    <h1>Introduction à Ajax (JavaScript)</h1>
    <br><br> Pays : <input type="text" id="chercher" name="chercher"
autocomplete="off">   Résultat : <select name="pays"
id="pays"></select>
</body>
```

Le script JavaScript suivant vous présente le code complet de cette page web.

```
<script>
        window.addEventListener("load", function() {
            // Déclaration d'un tableau vide qui contiendra les noms des pays en
retour de la réponse Ajax
            var pays = [];
            var chercher = document.getElementById("chercher");

            chercher.addEventListener("keyup", function() {
                ajaxPays(this.value);
            });
        });

        function ajaxPays(mot) {
            console.clear();
            console.log("Mot saisi : " + mot);

            /* Création de l'objet ajax */
            var ajax = new XMLHttpRequest();

            /* Définition du type d'appel et de l'url à charger */
            ajax.open("POST", "pays.json", true);

            /* Définition de l'appel en mode POST (obligatoirement après le open)
 */
            ajax.setRequestHeader("Content-type", "application/x-www-form-
urlencoded");

            /* Lancement de l'appel */
            ajax.send();

            ajax.addEventListener("readystatechange", function() {

                console.log("HTTP status = " + this.status);
                console.log("Etat traitement requête = " + this.readyState);
                console.log("**********************************************");

                if (this.status == 200 && this.readyState == 4) { /* Retour HTTP
réussi et état du traitement de la requête */

                    var html = "";
                    pays = JSON.parse(this.response);

                    console.log(pays);

                    pays.forEach(element => {

                        var position = element.nom_fr_fr.toUpperCase().substr(0,
mot.length).search(mot.toUpperCase());
```

```
                        if (position !== -1) {
                            console.log(element.nom_fr_fr);
                            html += "<option value='" + element.id + "'>" + eleme
nt.nom_fr_fr + "<br></option>";
                        }
                    });

                    document.getElementById("pays").innerHTML = html;

                }
            });


        }
    </script>
```

I.4 Créer un appel plus fiable

L'appel précédent a été conçu sans tenir compte du fait que des évènements perturbateurs peuvent se produire.

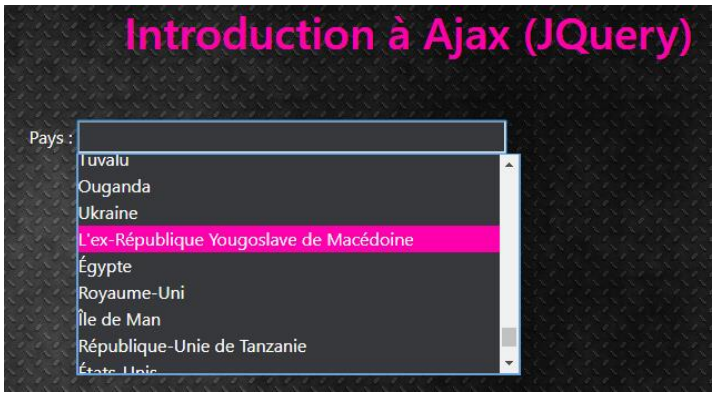
Pour garantir un fonctionnement cohérent dans toutes les situations, il est nécessaire de détecter :

1 – Une erreur dans l'appel, en utilisant l'événement **onerror** sur l'objet **XMLHttpRequest**.

2 – Un mauvais contenu **JSON** retourné par l'appel, en utilisant le bloc **try … catch** déjà rencontré.

TP – Vous allez devoir reprendre l'étude de cas précédente et en implémenter cette fois une version en **JQuery**. La bibliothèque **JQuery** permet d'adresser des requêtes **Ajax** simplifiées au serveur en s'affranchissant de l'utilisation directe de l'objet **XMLHttpRequest.** Bien entendu, c'est bien cet **objet JavaScript** qui est sollicité à travers les méthodes offertes par **JQuery** ;-)
De plus, cette fois, vous mettrez en place une liste **ul>li** qui récupèrera la liste renvoyée par la **requête Ajax**, dès que l'utilisateur choisira un élément dans la liste, cet élément s'affichera dans l'input et le reste des éléments de la liste sera alors caché.

Pour vous faciliter le développement de cette partie, intéressez-vous à la **délégation d'évènement** en **JQuery**. Vous pourrez consulter pour cela l'article suivant :
https://blog.infeeny.com/2013/10/10/la-delegation-devenement-avec-jquery/



Afin de vous aider dans vos recherches sur les appels **Ajax** avec **JQuery**, ce petit conseil de lecture …

https://openclassrooms.com/fr/courses/1567926-un-site-web-dynamique-avec-jquery/1569648-le-fonctionnement-de-ajax

Et bien sur le support : Support3 - JavaScript-JQuery.pdf

I.5 Les Contraintes liées à Ajax

Si Ajax répond à de nombreux besoins ergonomiques, il n'en reste pas moins une discipline assez complexe, avec de nombreuses contraintes.
Ce dernier paragraphe balaie l'ensemble des difficultés auxquelles un développeur web pourra être confronté.

I.5.1 La Sécurité

La sécurité est l'élément essentiel des appels **Ajax**. Les navigateurs ont introduit par défaut des processus de blocage d'appel assez stricts, pour empêcher les attaques par **XSS** (**Cross-Site Scripting**)
Ces attaques utilisent une faille de sécurité permettant d'injecter du contenu ou du code non légitime.
L'utilisateur est alors soumis à de fausses informations ou plus souvent à du code JavaScript dangereux, destiné à lui dérober des données de connexion, à lui afficher des publicités ou à le rediriger vers un site pirate.
Ainsi, par défaut, un appel **Ajax** vers un domaine différent du domaine d'origine est interdit.
Pour autoriser les appels vers des domaines externes, il faut indiquer dans **l'en-tête http** de la ressource appelée, les domaines autorisés grâce à la propriété **Access-Control-Allow-Origin**.

I.5.2 Ajax en local

Ajax en local, c'est-à-dire avec un script exécuté directement depuis le disque dur, avec le protocole **file://** ne fonctionnera pas correctement.

Par exemple, le code de retour ne sera jamais correctement renseigné à la valeur 200 identifiant un appel correct, car aucun serveur web n'est utilisé.
Il est recommandé de programmer et de tester ses scripts avec un véritable serveur web, soit avec un site distant chez l'hébergeur classique, soit via un serveur local sur une machine virtuelle dédiée.

I.5.3 Charge serveur

La programmation Ajax a comme conséquence de multiplier très facilement les appels vers le serveur. Avec une audience significative, le serveur peut être soumis à une charge élevée et entrainer des baisses de performances, des coupures ou des allongements de temps de réponse.

Les **temps de réponse** d'un site sont un des multiples éléments entrant en compte pour le **classement des résultats** dans les **moteurs de recherche**.

I.5.4 Référencement

Aujourd'hui, tous les **robots d'indexation** des moteurs de recherche savent parfaitement interpréter le JavaScript, les appels Ajax et leur effet sur le rendu des pages. *L'impact sur le référencement d'un site utilisant Ajax n'est donc plus significatif*.

La question qui reste posée est : L'accès aux informations est-il possible via une URL directe ? Si aucune URL directe n'est proposée, le moteur de recherche ne pourra pas la référencer et les utilisateurs ne pourront pas y accéder directement, ni par moteur de recherche, ni par partage de liens.

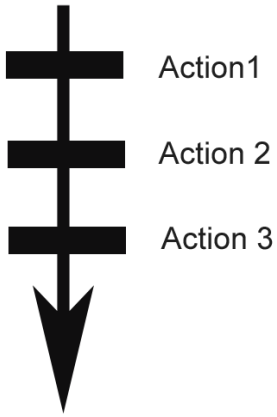
I.5.5 Ergonomie

La programmation **Ajax** implique une **gestion des erreurs** nettement plus poussée. Contrairement à une erreur de chargement d'une page web entière qui est signalée par le navigateur, **un appel Ajax qui échoue n'est pas signalé**.
C'est au **développeur** d'informer l'utilisateur que son action n'a pas été correctement enregistrée pour lui donner la possibilité de la relancer.

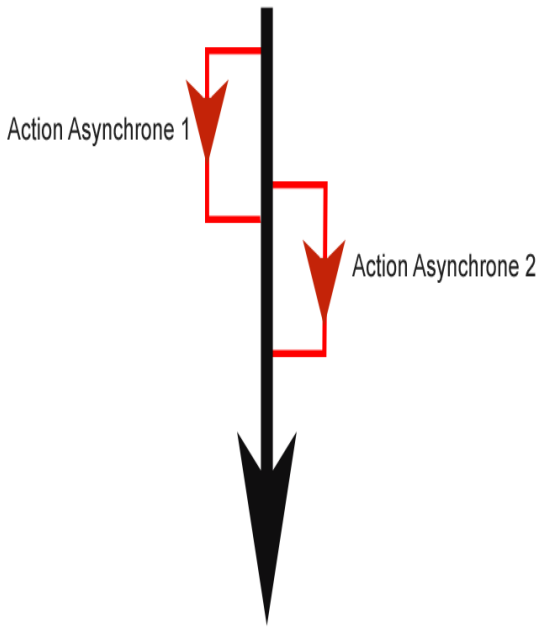
LES PROMESSES

I.1 Le principe des promesses en JavaScript

Par défaut le **JavaScript** s'exécute de manière **synchrone**, c'est-à-dire que les différentes actions s'enchaînent les unes derrières les autres (et pas en même temps comme le laisse supposer le mot synchrone) mais en suivant l'écriture du code source (interprétation du code de haut en bas sur un seul fil d'exécution).
Ceci est dû au caractère **"MonoThreadé"** du langage **JavaScript**.

	Le caractère **synchrone** de **JavaScript** a ses limites, en effet, si par exemple l'**Action 1** ne peut être réalisée toutes les actions qui suivent **Action 2** et **Action 3** seront aussi mises en défaut. Il y aura « *plantage* » du programme.

Il va donc falloir utiliser une autre méthode pour éviter autant que possible le blocage du programme lorsqu'une action ne pourra pas être réalisée. Ceci introduit la notion de **Promesse** en **JavaScript** ou déclenchement **d'action Asynchrone**.

	Ici des **actions asynchrones** sont lancées et le **programme principal** continu à s'exécuter sans attendre la réponse de ces **actions asynchrones** qui peuvent **réussir,** auquel cas le résultat rejoindra le fil d'exécution du programme principal. Ou bien être mises en **échec** ce qui pourra être géré par le développeur. Dans les **2 cas** le programme principal continuera à s'exécuter sans être mis en défaut.

Définition : Une **Promesse** représente une valeur qui peut être disponible **maintenant**, dans le **futur** ou bien **jamais**.
On parlera alors de **l'état de la Promesse**, soit elle sera **résolue** soit elle sera mise en **échec**.

Pour utiliser une Promesse en JavaScript il faudra s'y prendre en **2 étapes** :

1- Création d'une Promesse

```
const maPromesse = new Promise(function(resolve, reject){
    // ....
    resolve("C'est OK !!!");
    // ...
    reject("Ce n'est pas OK du tout !!!");
});
```

2- Capter l'état de la Promesse

```
maPromesse
            .then(dataResolve => {
                console.log(dataResolve ); // Affiche C'est OK

            })
            .catch(errorReject => {
                console.log(errorReject ); // Affiche Ce n'est pas OK du tout !!!
            });
```

Regardez l'exemple présenté par le formateur sur l'affichage des données issues d'un Tableau **felins** et d'un tableaux **actus** sur le monde des animaux pour bien comprendre ces notions mises en jeu lors d'**appels asynchrones** en **JavaScript**.
JQuery offre sa propre implémentation des **promesses,** vous pourrez l'étudier avant de la mettre en pratique sur l'exemple proposé par votre formateur.
https://api.jquery.com/promise/

I.2 L'API Fetch de JavaScript

Dans cette section, nous allons étudier **l'API Fetch** et sa méthode **fetch()** qui correspondent à la "nouvelle façon" d'effectuer des **requêtes HTTP**.
Cette **API** est présentée comme étant plus flexible et plus puissante que l'ancien objet **XMLHttpRequest**.

Cette méthode **fetch()** permet l'échange de données avec le serveur de manière **asynchrone**.

La méthode **fetch()** prend en **unique argument obligatoire** le **chemin de la ressource** qu'on souhaite récupérer.
On va également pouvoir lui passer en **argument facultatif** une liste d'options sous forme **d'objet littéral** pour préciser la méthode d'envoi, les en-têtes, etc.

La méthode **fetch()** renvoie une **promesse** (un objet de type Promise) qui va se résoudre avec un objet **Response**.
Notez que la promesse va être résolue dès que le serveur renvoie les en-têtes HTTP, c'est-à-dire avant même qu'on ait le corps de la réponse.
La promesse sera rompue si la requête HTTP n'a pas pu être effectuée.

On va donc devoir vérifier le **status HTTP** de la **réponse**. Pour cela, on va pouvoir utiliser les **propriétés ok** et **status** de l'objet **Response** renvoyé.

La propriété **ok** contient un booléen : **true** si le **status** code HTTP de la réponse est compris entre 200 et 299, **false** sinon.

La propriété **status** va renvoyer le statut code HTTP de la réponse (la valeur numérique liée à ce statut comme 200, 301, 404 ou 500).

Pour récupérer le corps de la réponse, nous allons pouvoir utiliser les méthodes de l'interface **Response** en fonction du format qui nous intéresse :

La méthode text() retourne la réponse sous forme de chaine de caractères ;
La méthode json() retourne la réponse en tant qu'objet JSON ;
La méthode formData() retourne la réponse en tant qu'objet FormData ;
La méthode arrayBuffer() retourne la réponse en tant qu'objet ArrayBuffer ;
La méthode blob() retourne la réponse en tant qu'objet Blob ;

Comme écrit ci-dessus, la méthode **fetch()** accepte un **deuxième argument facultatif**.
Cet argument est un objet qui va nous permettre de définir les options de notre requête.
On va pouvoir définir les options suivantes :

method : méthode utilisée par la requête. Les valeurs possibles sont GET (défaut), POST, etc.) ;
headers : les en-têtes qu'on souhaite ajouter à notre requête ;
body : un corps qu'on souhaite ajouter à notre requête ;
referrer : un référant. Les valeurs possibles sont "about:client" (valeur par défaut), "" pour une absence de référant, ou une URL ;
referrerPolicy : spécifie la valeur de l'en-tête HTTP du référent. Les valeurs possibles sont no-referrer-when-downgrade (défaut), no-referrer, origin, origin-when-cross-origin et unsafe-url ;
mode : spécifie le mode qu'on souhaite utiliser pour la requête. Les valeurs possibles sont cors (défaut), no-cors et same-origin ;
credentials : les informations d'identification qu'on souhaite utiliser pour la demande. Les valeurs possibles sont same-origin (défaut), omit et include ;
cache : le mode de cache qu'on souhaite utiliser pour la requête. Les valeurs possibles sont default (défaut), no-store, reload, no-cache, force-cache et only-if-cached ;
redirect : le mode de redirection à utiliser. Valeurs possibles : follow (défaut), manual, error ;

integrity : contient la valeur d'intégrité de la sous-ressource de la demande. Valeurs **possibles** : "" (défaut) ou un hash ;
keepalive : permet à une requête de survivre à la page. Valeurs possibles : false (défaut) et true ;
signal : une instance d'un objet AbortSignal qui nous permet de communiquer avec une requête fetch() et de l'abandonner.

Lien qui vous expliquera comment envoyer des requêtes en GET et en POST en utilisant l'API Fetch de JavaScript :

https://www.digitalocean.com/community/tutorials/how-to-use-the-javascript-fetch-api-to-get-data-fr

Mise en pratique de Fetch.

Pour la mise en pratique, vous pourrez reprendre le **TP Code Postal** en utilisant cette fois **l'API Fetch** afin de requêter sur **l'API Web** proposée par le **Web Service Zippopotam**.
Vous pourrez intégrer par ailleurs la librairie **JQuery** pour gérer les évènements sur votre page.

Service Web : C'est un ensemble de **services** qui permet l'échange de données et la communication entre **divers applications**.
Ils proposent tous une **API** que le **développeur** doit étudier avant de pouvoir la **consommer** dans ses **programmes**.
Il existe des milliers de Web Services sur la toile, on parle souvent de **Service Rest** ou **RestFull** car ils reposent sur une architecture qui repose sur le protocole **http** et sur les **URI**.
Certains de ces **Services** sont **payants** et d'autres sont **publiques.** Le projet **GitHub API Publique** propose une liste collective d'API gratuites à utiliser dans le développement de logiciels et de sites Web. (*https://github.com/public-apis/public-apis#text-analysis*).
Il y a aussi le site : *https://www.programmableweb.com/* qui propose un annuaire des **API Web** disponibles, la liste des API les plus populaires et des informations très riches sur l'actualité des API.

API : **A**pplication **P**rogramming **I**nterface c'est un ensemble normalisé de classes, de méthodes et de constantes qui sert de façade par laquelle **un logiciel** offre des **services** à un **autre logiciel**. Bref ce sont des **éléments** qui sont manipulés par des **développeurs**.

AJAX permet de **consommer** de façon **asynchrone** ces **services** afin d'enrichir une applications web.

Equipe de conception

M. Sacha RESTOUEIX : Formateur à l'Afpa de Brive la Gaillarde

Remerciements :

Merci à tous …

Afpa / Direction de l'Ingénierie13 place du Générale de Gaulle / 93108 Montreuil Cedex