LIVRET D'ACCOMPAGNEMENT DE FORMATION

Développeur Web ♦ Michael CORNILLON

Objectifs

Mettre à jour un contenu partiel d'une page web sans avoir à la recharger

Requêter un serveur depuis du code JavaScript et exploiter la réponse obtenue

Date du document
12/02/2019

Version
1.0

Table des matières

Présentation

Ajax c'est avant tout **l'assemblage de techniques** qui existaient avant **Ajax** dans le but de remplir une mission bien précise : interagir avec un serveur de manière efficace. Ces techniques sont : **XML/HTML**, **CSS**, **JavaScript** et l'API **DOM**, et de façon plus discrète tous langages de script pouvant être utilisé côté serveur (**PHP** par exemple). C'est donc l'ensemble de ces technologies qui font qu'Ajax existe.

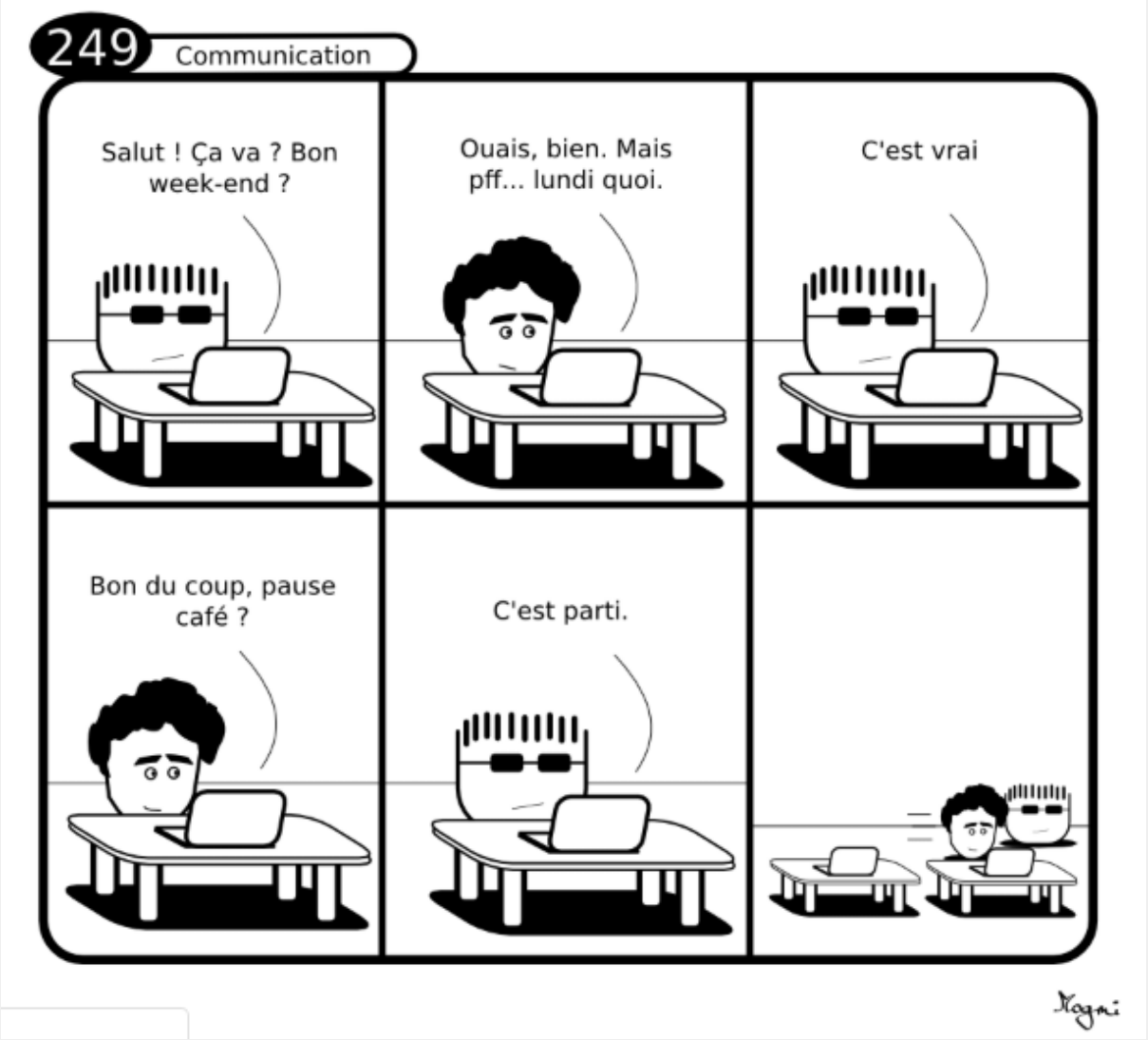
La technologie Ajax se base sur l'utilisation de l'objet XMLHttpRequest (un des objets natifs fourni par le navigateur, souvent abrégé **XHR**). Ce dernier permet d'envoyer des requêtes HTTP(S) à un serveur. Il a été créé par Microsoft au départ sous le format d'un objet ActiveX (technologie permettant le dialogue entre programmes).

Suite à une requête Ajax, la réponse retournée par le serveur sera exploitable côté client et livrée dans un des formats suivants : XML, HTML, JSON ou texte brut (*plain text*).

Le terme **Ajax** est un acronyme : Asynchronous JavaScript And XML (JavaScript et XML asynchrones). Cela nous renseigne donc sur une des qualités importantes concernant cette technologie : la possibilité de l'asynchrone. En effet, par défaut Ajax émet les requêtes HTTP de façon asynchrone, c'est-à-dire non bloquante. Autrement dit, le code JavaScript qui suit l'envoi d'une requête n'aura pas à attendre la réponse du serveur et pourra continuer son exécution.

Pour illustrer le principe de l'asynchronisme, prenons un exemple de la vie courante : la communication par téléphone et la communication par mail ou par voie postale. Un échange au téléphone s'apparente à une **communication**

asynchrone. C'est à dire qu'au moment où une personne parle, l'autre attend pour pouvoir parler à son tour (en partant du principe qu'il s'agisse d'un échange cordial entre les 2 personnes bien sûr). L'exemple aurait été plus parlant dans le cas d'une conversation par talkie walkie car alors il ne peut y en avoir qu'un qui écoute pendant que l'autre parle. Celui qui écoute, même s'il a un message à transmettre, reste bloqué et ne pourra le délivrer qu'après que l'autre ne lui rende la parole. Pour le cas d'un envoi par mail, on est bien dans une communication de type **asynchrone** : une fois l'email envoyé, l'utilisateur peut vaquer à d'autres occupations et réagir plus tard à la réponse, lorsqu'il en aura été notifié.



Mais ce qui a rendu Ajax très populaire depuis sa création, outre le fait de permettre la communication client/serveur via du JavaScript, c'est le fait de pouvoir mettre à jour une **portion seulement** de page web sans que cette dernière ne soit actualisée (renvoyée depuis le serveur). Le logo met d'ailleurs bien cela en évidence. Nous y voyons les 4 lettres symbolisant le contenu d'une page web dans son ensemble, alors que seule la lettre **J** est en cours de mise à jour (la double flèche évoquant l'aller-retour caractéristique d'une communication client/serveur).

Un peu d'histoire

L'expression **Web 2.0** a été introduit par Dale Dougherty en 2003 pour finalement être adopté au cours des années qui ont suivies. On parle alors de web participatif car à la différence du web traditionnel les pages offraient enfin de l'interactivité.

Ajax joue un rôle important durant cette période et même après grâce à sa capacité d'interaction avec l'utilisateur (par exemple: auto-complétion, surveillance bienveillante des saisies). Le **Web 2.0** désignait d'ailleurs la généralisation de l'utilisation des technologies Ajax qui permettaient de modifier l'apparence d'une page Web en fonction de données distantes (serveur) sans bloquer l'utilisation de cette dernière et sans qu'il n'y ait besoin de la recharger.

Comme déjà mentionné, à l'origine Microsoft développa **XMLHttpRequest** comme objet **ActiveX** pour son navigateur Internet Explorer 5.0 (fin 1998). Il devint une recommandation du W3C seulement début 2006.

Mise en pratique

Création de l'objet XMLHttpRequest

Avant de pouvoir créer la moindre requête en Ajax, il faut créer une instance de l'objet natif **XMLHttpRequest**. Le souci c'est qu'il faut prendre en compte la rétrocompatibilité avec Internet Explorer. Pour rappel, chez Microsoft l'**XHR** est disponible sous la forme d'un objet **ActiveX** :

```
let xhr;

if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
}
else if (window.ActiveXObject) { // IE
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Une bonne pratique consiste à encapsuler ce code dans une fonction pour récupérer l'objet à la demande et éviter d'avoir à réécrire toute la procédure. Voici une autre portion de code dans laquelle on procède non plus par une condition pour vérifier la présence de l'**XHR** mais par un bloc **try/catch** :

```
function createXHR() {
    let xhr;

    try {
        xhr = new XMLHttpRequest();
    }
    catch (e) { // IE
        xhr = new ActiveXObject("Microsoft.XMLHTTP");
    }

    return xhr;
}
```

Envoi de la requête au serveur

Pour envoyer une requête au serveur, 2 méthodes suffisent. Elles sont fournies par l'instance **XHR** :

- **open()** : permet de préparer la requête — pour cela, elle reçoit en paramètre **la méthode de requête** (GET ou POST), l'url cible (les données à récupérer), et éventuellement le choix du mode d'envoi (asynchrone ou non).
- **send()** : permet d'envoyer la requête préalablement préparée — pour cela, si la méthode de requête choisie est POST, il faudra renseigner en paramètre un objet contenant les données destinées au serveur.

```
14  function sendRequest(url) {
15      const xhr = createXHR();
16      xhr.open('get', url, true);
17      xhr.send(null);
18  }
```

```
sendRequest("https://data.toulouse-metropole.fr/api/records/1.0/search/?dataset=liste-elus-tm-old");
```

L'url ci-dessus permet de récupérer la liste des élus au Conseil Municipal de la Mairie de Toulouse (cliquez sur l'image pour se rendre à l'API).

Réception de la réponse

La requête (demande) envoyée au serveur, nous ne pouvons pas savoir quand ce dernier pourra nous répondre. Par exemple, une panne du réseau pourrait très bien couper toute communication avec ce dernier sans qu'il n'ait eu le temps de renvoyer la réponse appropriée. Il nous faut donc

être notifiés de l'arrivée de la réponse. L'objet **XHR** fournit une propriété pour cela : **onreadystatechange**. Il suffit d'y stocker une callback qui sera exécutée aux bons moments... *Aux bons moments* est au pluriel car la callback sera en fait appelée plusieurs fois au cours de la vie de la requête. Avant d'aller plus loin dans les explications, analysons le code suivant :

```
15  function sendRequest(url) {
16      const xhr = createXHR();
17      xhr.open('get', url, true);
18      xhr.onreadystatechange = function() {
19          if (xhr.readyState === 4 && xhr.status === 200) {
20              console.log("données reçues !");
21          }
22      };
23      xhr.send(null);
24  }
```

A la ligne 18, on valorise la propriété **onreadystatechange** avec une fonction (notre callback de traitement de la réponse). On peut remarquer la présence de 2 propriétés de l'objet **XHR** dont nous n'avons pas encore parlé : **readyState** et **status**. La première indique l'état (de 0 à 4) relatif au cycle de vie de la requête (voir la portion d'article ci-dessous provenant du site OpenClassrooms). La seconde indique l'état de la réponse HTTP.

- **0 :** L'objet XHR a été créé, mais pas encore initialisé (la méthode `open` n'a pas encore été appelée)
- **1 :** L'objet XHR a été créé, mais pas encore envoyé (avec la méthode `send`)
- **2 :** La méthode `send` vient d'être appelée
- **3 :** Le serveur traite les informations et a commencé à renvoyer des données
- **4 :** Le serveur a fini son travail, et toutes les données sont réceptionnées

Vous pouvez cliquer sur l'image pour vous rendre sur l'article en question.

Si le *status code* est à 200, cela signifie que la réponse a bien été générée (dans le sens où elle correspond bien à nos attentes, et contient les données désirées).

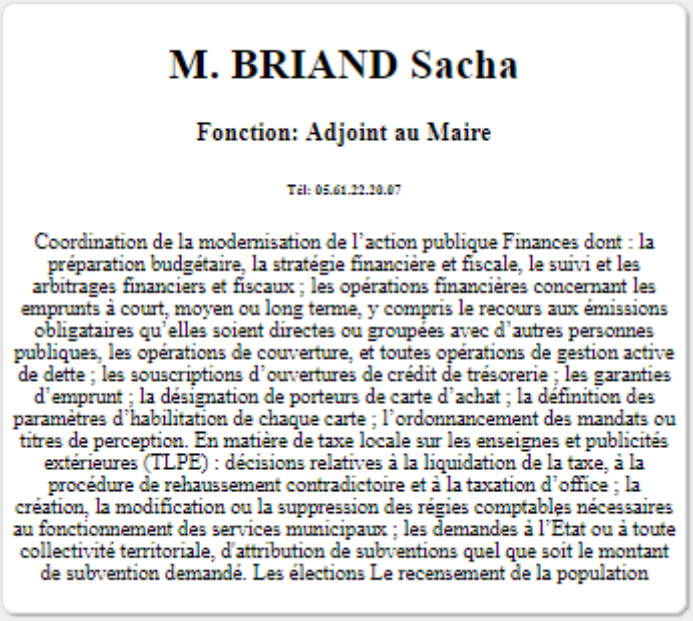
Traitement de la réponse

Les données fournies dans notre cas sont formatées en json. Deux solutions s'offrent alors. On peut les récupérer sans préciser de valeur pour la propriété **xhr.responseType**. Dans ce cas, il sera nécessaire de les parser à l'aide de la méthode **JSON.parse()** pour pouvoir les manipuler en JavaScript. Si l'on précise la valeur souhaitée (**xhr.responseType = "json"**), les données sont directement manipulables. L'exemple ci-dessous montre comment exploiter les données reçues de l'Open Data de Toulouse. Ici, nous récupérons uniquement le premier champs :

```
8   function sendRequest(url) {
9       const xhr = createXHR();
10      xhr.responseType = "json";
11      xhr.open('get', url, true);
12      xhr.onreadystatechange = function() {
13          if (xhr.readyState === 4 && xhr.status === 200) {
14              createHTMLCard(xhr.response.records[0].fields);
15          }
16      };
17      xhr.send(null);
18  }
```

```
27  function createHTMLCard(electInfos) {
28      const div = document.createElement('div');
29
30      with (electInfos) {
31          div.className = "elu-cm elu-cm-toulouse";
32          div.innerHTML = `
33              <h1>${civilite_abregee} ${nom} ${prenom}</h1>
34              <h3>Fonction: ${fonction}</h3>
35              <h6>Tél: ${numero_de_telephone}</h6>
36              <p>${delegation}</p>
37          `;
38      }
39
40      document.body.appendChild(div);
41  }
```

La fonction **createHTMLCard()** permet simplement de créer une structure HTML en y intégrant les données reçues et de les rendre sur la page Web :



Remarque sur l'utilisation du mot clé **with** : son usage n'est pas recommandé et n'est utilisé ici qu'à des fins pédagogiques.

> Il n'est pas recommandé d'utiliser l'instruction `with`. En effet, elle est parfois source de problèmes de compatibilité ou de bogues. Se référer au paragraphe « Inconvénient : l'ambiguïté » de la section « Description » pour plus de détails.

Vous pouvez cliquer sur l'image pour vous rendre sur l'article en question.

Voici le lien du code complet que vous pourrez manipuler à votre guise. Et voici également un code plus complet pour la création d'un objet **XHR** :

```
function createXhrObject()
{
    if (window.XMLHttpRequest)
        return new XMLHttpRequest();

    if (window.ActiveXObject)
    {
        var names = [
            "Msxml2.XMLHTTP.6.0",
            "Msxml2.XMLHTTP.3.0",
            "Msxml2.XMLHTTP",
            "Microsoft.XMLHTTP"
        ];
        for(var i in names)
        {
            try{ return new ActiveXObject(names[i]); }
            catch(e){}
        }
    }
    window.alert("Votre navigateur ne prend pas en charge l'objet XMLHTTPRequest.");
    return null; // non supporté
}
```

Vous pouvez cliquer sur l'image pour vous rendre sur l'article en question.