Programmation WEB

Ajax

Denis Monnerat

monnerat@u-pec.fr

IUT de Fontainebleau

Sommaire

Introduction

Introduction

- AJAX se base sur l'objet JavaScript `XMLHttpRequest` qui permet de requêter dynamiquement une url via le protocole HTTP(S).
- L'un des avantages est de pouvoir échanger des données entre la page et un serveur sans avoir à recharger entièrement la page.

Intéraction "traditionnelle"

page > click > attente > rafraîchissement de la page

Avec Ajax

- Seuls les éléments d'interface qui contiennent de nouvelles informations sont rafraîchis de manière asynchrone.
- Le reste de l'interface reste visible (pas de perte de contexte opérationnel)

IHM orientée données vs pages

L'IHM est gérée par le client, tandis que les données sont calculées et fournit par le serveur.

- Le modèle asynchrone remplace le modèle synchrone requête/réponse.

⇝ L'application reste utilisable pendant que le client demande des infos au serveur en arrière plan.
⇝ Séparation de la présentation et de l'accès aux données.

- La programmation côté serveur reste la même :

⇝ cgi qui recoivent des requêtes http : servlet, jsp, php, python, ruby, asp, etc.
⇝ et qui génére une réponse http avec un contenu pouvant être de type xml, json, javascript, texte brute, html, etc.

Quelques cas d'utilisations. Il y a en beaucoup d'autres.

Google map

- L'utilisateur draggue une partie de la carte.
- Le nouveau morceau de carte est chargé de manière asynchrone avec Ajax.
- Le reste de l'interface utilisateur est inchangée.

Auto-complétion

Emails, noms, code postaux, etc. peuvent être auto-complétés quand l'utilisateur fait sa saisie.

Eléments d'interfaces avancées

Menus, arbres, barre de progression, date picker peuvent être utilisés sans rafraîchir toute l'interface.

Un exemple qui utilise des données d'un document XML récupéré par le réseau :

```
function processData(data) {
  // taking care of data
}

function handler() {
  if(this.readyState == this.DONE) {
    if(this.status == 200 &&
       this.responseXML != null &&
       this.responseXML.getElementById('test').textContent) {
      // success!
      processData(this.responseXML.getElementById('test').textContent);
      return;
    }
    // something went wrong
    processData(null);
  }
}

var client = new XMLHttpRequest();
client.onreadystatechange = handler;
client.open("GET", "unicorn.xml");
client.send();
```

Log sur un serveur

```
function log(message) {
  var client = new XMLHttpRequest();
  client.open("POST", "/log");
  client.setRequestHeader("Content-Type",
              "text/plain;charset=UTF-8");
  client.send(message);
}
```

Récupérer le statut d'un document sur le serveur

```
function fetchStatus(address) {
  var client = new XMLHttpRequest();
  client.onreadystatechange = function() {
    if(this.readyState == this.DONE)
      returnStatus(this.status);
  }
  client.open("HEAD", address);
  client.send();
}
```

Implantation

Spécification W3C

```
interface XMLHttpRequest : XMLHttpRequestEventTarget {
  // event handler
  attribute EventHandler onreadystatechange;

  // states
  const unsigned short UNSENT = 0;
  const unsigned short OPENED = 1;
  const unsigned short HEADERS_RECEIVED = 2;
  const unsigned short LOADING = 3;
  const unsigned short DONE = 4;
  readonly attribute unsigned short readyState;

  // request
  void open(ByteString method, [EnsureUTF16] DOMString url);
  void open(ByteString method, [EnsureUTF16] DOMString url, boolean async,
      optional [EnsureUTF16] DOMString? username = null,
      optional [EnsureUTF16] DOMString? password = null);
  void setRequestHeader(ByteString header, ByteString value);
           attribute unsigned long timeout;
           attribute boolean withCredentials;
  readonly attribute XMLHttpRequestUpload upload;
  void send(optional (ArrayBufferView or Blob or Document
        or [EnsureUTF16] DOMString or FormData)? data = null);
  void abort();
```

Spécification W3C

```
// response
  readonly attribute unsigned short status;
  readonly attribute ByteString statusText;
  ByteString? getResponseHeader(ByteString header);
  ByteString getAllResponseHeaders();
  void overrideMimeType(DOMString mime);
  attribute XMLHttpRequestResponseType responseType;
  readonly attribute any response;
  readonly attribute DOMString responseText;
  readonly attribute Document? responseXML;
};
```

Spécification W3C

Sur les navigateurs qui implantent les recommandations récentes, `XMLHttpRequest` hérite de `XMLHttpRequestEventTarget` et rajoutent les handlers

```
// event handlers
  attribute EventHandler onloadstart;
  attribute EventHandler onprogress;
  attribute EventHandler onabort;
  attribute EventHandler onerror;
  attribute EventHandler onload;
  attribute EventHandler ontimeout;
  attribute EventHandler onloadend;
```

Création

Sur tous les navigateurs "modernes" :

```
var xmlhttp=new XMLHttpRequest();
```

Navigateurs anciens

```
function getXMLHttpRequest(){
  var xmlhttp;
  try{
    xmlhttp = new XMLHttpRequest();
  }
  catch (e1){
    try {
      xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
    }
    catch (e2) {
      try {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
      }
      catch (e3) {
        xmlhttp=null;
      }
    }
  }
  return xmlhttp;
}
```

L'objet `XMLHttpRequest`

Attributs

`onreadystatechange`	fonction réflexe appelée lors de l'événement `onreadystatechange` , qui se produit, en mode asynchrone, lors d'un changement d'états de la requête.
`readyState`	état courant de la requête
`responseText`	Réponse sous forme de texte
`responseXML`	Réponse sous forme XML
`status`	code du statut de retour HTTP de la réponse
`statusText`	texte du statut de retour HTTP de la réponse
`responseType`	Type de la réponse : "", "arraybuffer", "blob", "document", "json", "text"

La classe `XMLHttpRequest`

`abort`

annule la requête et réinialise l'objet

`getAllResponseHeaders`

Retourne tous les entêtes HTTP de la réponse sous forme d'une chaîne de caractères. Valable uniquement pour la valeur 3 et 4 de `readyState`

`getResponseHeader(nom)`

renvoie la valeur de l'entête dont le nom est spécifié en paramètre. (idem ci-dessus)

`setRequestHeader(header,value)`

positionne un entête HTTP pour la requête.

`open(methode,url,async,[user],[password])`

initialise l'objet pour une requête.

- méthode http : GET, POST, etc.
- url : url de la requête.
- async : Le type de fonctionnement détermine la manière dont est traitée la réponse à la requête (synchrone ou asynchrone par défaut).
- Enfin, des informations (user et password) de sécurité peuvent être utlisées.

`send(string/document)`

envoie une requête à l'adresse spécifiée avec la méthode HTTP souhaitée. Si le mode de réception est synchrone, la méthode est bloquante jusqu'à ce moment. On peut ajouter une chaîne ou un document xml en paramètre en cas de post.

Gestions des échanges

⇝ Synchrone, c'est à dire séquentiellement.

```
var request=new XMLHttpRequest()
request.open("get","data.js",false); // false => synchrone
requete.send(null);
vat test=request.reponseText;
```

⇝ Asynchrone : réception par une fonction réflexe qui est appelée à chaque changement d'états :

Mode asynchrone

Valeur	Etat	description
0	UNSENT	état initial
1	OPENED	méthode open exécutée avec succés
2	HEADERS_RECEIVED	headers http reçus, aucune donnée reçue
3	LOADING	en cours de reception
4	DONE	Le traitement requête est terminé

Fonctionnement asynchrone

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function(){
  if (this.readyState==this.DONE && this.status == 200){
    var texte=this.reponseText;
    ...
  }
}
xhr.open("get","data.js",true);
xhr.send(null);
```

La fonction reflexe onreadystatechange est appelée 4 fois.

Echanges de données

Envoi

Il existe deux façons de spécifier des données lors de l'envoie :

- Dans l'url de la requête :

```
var a=encodeURIComponent(vala);
var b=encodeURIComponent(valb);
requete.open("get","requete.php?a="+a+"&b="+b,true);
requete.send(null);
```

- comme paramètre de la méthode send :

```
var url = "get_data.php";
var params = "lorem=ipsum&name=binny";
http.open("POST", url, true);
http.setRequestHeader("Content-type",
    "application/x-www-form-urlencoded");
http.setRequestHeader("Content-length", params.length);
http.setRequestHeader("Connection", "close");
http.onreadystatechange = function() {}
http.send(params);
```

On peut préciser, dans l'entête http de la requête, le type des données envoyées à l'aide la méthode `setRequestHeader`

Données url-encodées

```
var url = "get_data.php";
var params = "lorem=ipsum&name=binny";
http.open("POST", url, true);
http.setRequestHeader("Content-type",
    "application/x-www-form-urlencoded");
http.setRequestHeader("Content-length", params.length);
http.setRequestHeader("Connection", "close");
http.onreadystatechange = function() {}
http.send(params);
```

Pour les données de "formulaires", on peut utiliser l'interface `FormData`.

Construction "à la main"

```
var data = new FormData()
data.append('name', 'John Doe')
data.append('email, 'contact@local.dev')



xhr.send(data);
```

À partir d'un formulaire

```
var form = document.querySelector('#form')
var data = new FormData(form)



xhr.send(data);
```

```
var formData = new FormData();

formData.append("username", "Groucho");
formData.append("accountnum", 123456);
// number 123456 is
// immediately converted to a string "123456"

// HTML file input, chosen by user
formData.append("userfile", fileInputElement.files[0]);

// JavaScript file-like object
var content = '<a id="a"><b id="b">hey!</b></a>'; // the body of the new file...
var blob = new Blob([content], { type: "text/xml"});

formData.append("webmasterfile", blob);

var request = new XMLHttpRequest();
request.open("POST", "http://foo.com/submitform.php");
request.send(formData);
```

Données au format XML

```
xhr.open("post","./test.php");
xhr.setRequestHeader("Content-Type","text/xml");
xhr.send("<user>"+
    "<nom>Monnerat</nom>"+
    "<prenom>Denis</prenom>"+
    "<mail>monnerat@u-pec.fr</mail>"+
    "</user>");
```

Traitement en php côté serveur

```
<?php
$dom = new DomDocument();
$dom->loadXML(file_get_contents("php://input"));
$nom = $dom->getElementsByTagName("nom")->item(0)->firstChild->nodeValue;
$prenom = $dom->getElementsByTagName("prenom")->item(0)->firstChild->nodeValue;
$mail = $dom->getElementsByTagName("mail")->item(0)->firstChild->nodeValue;

echo "$nom $prenom $mail";
?>
```

Format des données

XML

Outre le format texte (pas très pratique à parser), AJAX offre nativement un support pour `XML` :

⇝ à l'envoi avec la méthode `send` :

```
var parametres=document.createElement("parametres");
var item=document.createElement("parametre");
item.setAttribute("nom","le_nom");
var valeur=document.createTextNode("la_valeur");
item.appendChild(valeur);
parametres.appendChild(item);
...
requete.send(parametres);
```

qui envoie la structure

```
<parametres>
  <item nom="le_nom">la_valeur</item>
</parametres>
```

⇝ à la réception avec l'attribut `responseXML`

```
_xmlHttp.onreadystatechange=function() {

  if(this.readyState==4&&this.responseXML) {

    var xmldoc=this.responseXML;
    var donnees=xmldoc.childNodes[0];
    for(i=0;i<donnees.childNodes.length;i++)
    {
      ....
      ....
    }
  }
```

HTML

Il s'agit en fait de texte, représentant du code html déstiné à rafraîchir la partie d'une page :

```
<html>
  <head>
    <script>
      ....
      requete.onreadystatechange=function (){
        if (this.readyState==4){
          var html=this.responseText;
          document.getElementById("reponse").innerHTML=html;
        }
      }
    </script>
  </head>
  <body>
    <div id="reponse"></div>
  </body>
</html>
```

Remarque : il est possible également d'envoyer en réponse du code javascript, exécuté à l'aide de la commande `eval`.

JSON

JSON (Javascript Object Notation) permet de déclarer une donnée sous forme de collection d'éléments. Il utilise deux structures de données :

- l'objet, sous forme de (clé valeur) :

```
{
  cle1:valeur1,
  cle2:function(){},
  cle3:valeur3
}
```

- le tableau simple : liste de valeurs

```
[
valeur,
function(){},
valeur
]
```

On peut les imbriquer :

```
[
{name:"Gary", age:"24"},
{name:"Shane", age:"29"},
{name:"Kay", age:"54"},
{name:"Sarah", age:"30"},
{name:"Jerry", age:"56"}
]
{"menu": {
  "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
      ]
    }
     }
}
```

La méthode `parse` de l'objet JSON permet de parser une chaîne de caractères représentant un objet

```
var text = '{ "employees" : [' +
  '{ "firstName":"John" , "lastName":"Doe" },' +
  '{ "firstName":"Anna" , "lastName":"Smith" },' +
  '{ "firstName":"Peter" , "lastName":"Jones" } ]}';

var obj = JSON.parse(text);
```

que l'on peut utiliser directement en js.

```
<p id="demo"></p>

<script>
  document.getElementById("demo").innerHTML =
  obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>
```

La méthode `stringify` fait le contraire

```
var obj={"x":2,"name":"denis"};
var s=JSON.stringify(obj);
```

Attribut responseType

```
enum XMLHttpRequestResponseType {
  ""
  "arraybuffer",
  "blob",
  "document",
  "json",
  "text"
};
```

Très pratique.

Par exemple, si on positionne `responseType` à json, le réponse sera automatiquement parser vers un objet json.

Restriction

Cors

Pour des problèmes de sécurité, une requête ajax ne peut se faire que sur le même domaine (origine unique : url,protocole et port)

Le w3c a recommendé le nouveau mécanisme de Cross-Origin Resource Sharing qui fournit un moyen aux serveurs web de contrôler les accès en mode cross-site et aussi d'effectuer des transferts de données sécurisés en ce mode.

CORS

Le standard de partage de ressources d'origines croisées fonctionne grâce à l'ajout d'entêtes HTTP qui permettent aux serveurs de décrire l'ensemble des origines permises.

Un exemple avec php et l'entête `Access-Control-Allow-Origin`

```php
<?php
header("Access-Control-Allow-Origin: *");
header("content-type: application/json");
mysql_connect("localhost","root","password");
mysql_select_db("communes");
$code=$_GET['code'];
$res=mysql_query("SELECT Commune,Departement
  FROM ville
  WHERE Codepos='$code';");
$t=[];
while($row=mysql_fetch_object($res))
{
  $t[]=$row;
}
echo json_encode($t);
?>
```

iframe cachée

L'iframe, que l'on peut facilement crée et caché, permet de faire des requêtes, à la fois GET et POST.

- l'attribut `src` d'une iframe permet de faire une requête `GET`.
- Pour une requête POST, il faut formulaire HTML, et doit être relié à l'iframe via son attribut `target`.

La réponse est disponible dans le `body` de l'iframe. Quand en prendre connaissance ?

- La réponse de la requête est du code javascript (balise <script>....</script>) exécuté dans l'iframe.
- On peut associer à l'iframe un gestionnaire d'événement de chargement.

JSONP

JSON-Padding : il ajoute une balise script au DOM, et demande au serveur d'afficher du JS qui exécutera une fonction globale.

Exemple : supposons que l'url `http://exmeple.com/jsonp` renvoie les dates anniversaires de personnes passées en paramètre.

- On déclare une fonction callback :

```
window.jsonpcallback = function(birthdate) {
      console.log(birthdate);
};
```

- On injecte le script suivant dans le dom :

```
<script src="http://exemple.com/jsonp?name=jason&callback=jsonpcallback"></script>
```

- La réponse du serveur :

```
jsonpcallback("10/09/1988");
```


Le navigateur exécute donc le code correspondant.

Récupération des données avec PHP

- Lorsque les données de la requête sont "ulr-encodés", PHP met à disposition du script les données dans les super-globables `_SERVER`,`_GET` et `_POST`.

- Pour récuperer les données brutes depuis le corps de la requête, il faut utiliser le flux `php://input`

```php
<?php
$obj=json_decode(file_get_contents("php://input"));
?>
```

Les promesses

Promise

Objet qui prend en charge la réalisation d'un traitement asynchrone. Elle représente une valeur disponible :

- maintenant
- dans le futur
- jamais

```
new Promise( /* exécuteur */ function(resolve, reject) { ... } );
```

La fonction exécuteur lance un travail. Les fonctions arguments `resolve` et `reject`, lorsqu'elles sont appelés par l'exécuteur, permettent de tenir ou rompre la promesse.

Une Promise est dans un de ces états :

- pending (en attente) : état initial, la promesse n'est ni remplie, ni rompue ;
- fulfilled (tenue) : l'opération a réussi ;
- rejected (rompue) : l'opération a échoué ;
- settled (acquittée) : la promesse est tenue ou rompue mais elle n'est plus en attente.

Une promesse en attente peut être tenue avec une valeur ou rompue avec une raison (erreur). Quand on arrive à l'une des deux situations, les gestionnaires associés lors de l'appel de la méthode `then` sont alors appelés.

Promesse avec Ajax

```
function getFile(url){
  return new Promise(function(resolve,reject){
    var xhr = new XMLHttpRequest();
    xhr.open('GET',url);
    xhr.onload=function(){
      if (xhr.status == 200)
        resolve(request.reponse);
      else
        reject("Erreur : "+xhr.statusText);
    };
    xhr.onerror = function(){
      reject("Erreur reseau");
    };
    request.send();
  }
}
```

Promesse avec Ajax

```
getFile("http://www.iut-fbleau.fr").then(
  function(file){
    /* on recupere
     * le fichier
     * */

  },
  function(erreur){
    console.log(erreur);
  });
```

Chaînage des promesses

Sans les promesses, l'enchaînement de plusieurs opérations asynchrones donnait une imbrication des `callbacks`

```
faireQqc(function(result) {
  faireAutreChose(result, function(newResult) {
    faireUnTroisiemeTruc(newResult, function(finalResult) {
      console.log('Résultat final :' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

La valeur d'une promesse résolue, peut-être encore une promesse.

```
faireQqc().then(function(result) {
  return faireAutreChose(result);
})
.then(function(newResult) {
  return faireUnTroisiemeTruc(newResult);
})
.then(function(finalResult) {
  console.log('Résultat final : ' + finalResult);
})
.catch(failureCallback);
```