WIM - M4103C

Ajax

monnerat@u-pec.fr

IUT de Fontainebleau

Sommaire

Introduction

Implantation

Echanges de données

Restriction

Les promesses

await/async

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.
- Historiquement, c'est XML qui servait de format d'échange. Aujourd'hui largement remplacé par JSON.

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.

Asynchrone

- ⇝ 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 :

Côté serveur

- ⇝ 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.

Évolutions

Toute l'application est coté client ⇒ Single Page Application.

- Framework javascript : vue,angular, react, ember, etc.

Les serveurs offrent des services (données, calculs, etc.) que l'application utilise avec http ⇒ API http

- API REST

Implantation

Implantation

`XMLHttpRequest`

Les objets `XMLHttpRequest` permettent d'effectuer des requêtes HTTP(S), et d'échanger tout type de données (pas uniquement du XML) avec un serveur.

Ces objets implantent l'interface `XMLHttpRequestEventTarget` pour leur gestion évenementielle.

Utilisation

1. On crée un objet `XMLHttpRequest`.
2. On initialise la requête : méthode HTTP, URL, etc.
3. On crée des gestionnaires d'évènements pour prendre en charge la réponse du serveur.
4. On envoie la requête.

Création

```
let xhr = new  XMLHttpRequest()
```

Initialisation

```
xhr.open(method, url, async, user, password)
```

- `method` (obligatoire) : méthode http(s) de la requête : `GET`, `POST`, `PUT`, `DELETE`.
- `url` (obligatoire) : url de la requête http(s).
- `async` (optionnel) : requête asynchrone ou non. `true` par défaut.
- `user`,`password` (optionnel) permettent de préciser un nom et mot de passe pour aithentification.

Type de la réponse attendue de la part du serveur

```
xhr.responseType = "json"
```

Valeurs possibles

- `text` : chaîne de caractère (valeur par défaut).
- `arraybuffer` : un objet `ArrayBuffer` (tableau d'octets)
- `blob` : un objet `Blob` (Binary Large Objects)
- `document` : XML
- `json` : JSON

Le plus souvent dans la pratique : `text` ou `JSON`.

Réponse du serveur

On utilise désormais les gestionnaires évènementiels `load`, `error` et `progress` définis par l'interface `XMLHttpRequestEventTarget`.

- `load` : la requête a été effectuée, et le résultat est prêt.
- `error` : la requête n'a pas abouti.
- `progress` : déclenché à intervalles réguliersa pour suivre la progression de la requête.

Attention : obtenir un "résultat" du serveur ne correspond pas forcément à ce que l'on voulait. Il faut tester la propriété `status` (code de retour HTTP) de l'objet `XMLHttpRequest`.

Code	Message
10x	Information
20x	Réussite (200 → OK, etc.)
30x	Redirection
40x	Erreur client (400 → BAD REQUEST,404 → NOT FOUND , etc.)
50x	Erreur serveur (500 → INTERNAL ERROR, etc.)

Exploiter la réponse

```
let xhr = new XMLHttpRequest()
xhr.open("GET","mon/url/")
xhr.responseType = "json"

xhr.onload = (ev)=>{
  if (xhr.status == 200)
    processData(xhr.response)
})

xhr.onerror = ()=>{
  console.log("error")
}

xhr.send()
```

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
`response`	Réponse
`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"
`timeout`	timeout à la requête en millisecondes

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.

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 :

```
let url = "get_data.php";
let 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.send(params);
```

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

```
let url = "get_data.php";
let 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.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 du document

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

xhr.send(data);
```

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 côté serveur

```
$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";
```

XML

AJAX offre nativement un support pour XML : à l'envoi

```
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`

```
xhr.onload=function() {
    letr xmldoc=this.responseXML;
    let donnees=xmldoc.childNodes[0];
    for(i=0;i<donnees.childNodes.length;i++){
      ....
    }
  }
```

JSON

Le format d'échange le plus utilisé est désormais le json.

```
const json = {
    "email": "eve.holt@reqres.in",
    "password": "cityslicka"
};

// open request
xhr.open('POST', 'https://reqres.in/api/login');

// set `Content-Type` header
xhr.setRequestHeader('Content-Type', 'application/json');

// send request with JSON payload
xhr.send(JSON.stringify(json));
```

```
const xhr = new XMLHttpRequest()
xhr.responseType = "json"

xhr.onload = () => {
  const response = xhr.responseText
  // pas de besoin de parser
  // la reponse
  // c'est un objet json
  // (si le serveur renvoie
  // du json !)
  console.log(response)
}
```

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/` 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");
```

Récupération des données avec PHP

Lorsque les données de la requête sont "url-encodées", 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 ;

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.

```
const promise = new Promise((resolve, reject) => {})
console.log(promise)

// output
//__proto__: Promise
//[[PromiseState]]: "pending"
//[[PromiseResult]]: undefined
```

```
const promise = new Promise((resolve, reject) => {
  resolve('We did it!')
})
console.log(promise)
// output
// __proto__: Promise
//[[PromiseState]]: "fulfilled"
//[[PromiseResult]]: "We dit it !"
```

```
const promise = new Promise((resolve, reject) => {
  reject('Sorry !')
})
console.log(promise)

// output
// __proto__: Promise
//[[PromiseState]]: "rejected"
//[[PromiseResult]]: "Sorry !"
```

Evidemment, ces trois exemples sont "inutiles", car synchrone ...

then/catch

`Promise.prototype.then` permet d'enregistrer une fonction qui recevra la valeur de la promesse est une fois tenue :

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve('request is ok !'), 2000)
})

// Log the result
promise.then((response) => {
  console.log(response)
})
```

then/catch

`Promise.prototype.catch` permet d'enregistrer une fonction qui recevra la raison du rejet d'une promesse une fois rejetée :

```
function f(success){
  return new Promise ((resolve,reject) => {
    setTimeout(() => {
      if (sucess)
        resolve("OK")
      else
        reject("NOK")
    },1000)
  })
}
f(false)
  .then((response) => {
    console.log(response)
  })
  .catch((error) => {
    console.error(error)
  })
```

`then` et `catch` renvoient elle-même une promesse : on peut les composer (les chaîner).

Promesse avec Ajax

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

    xhr.onerror = () => reject("Erreur reseau")

    xhr.send()
  }
}
```

Promesse avec Ajax

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

  },
  (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);
```

On peut créer une promesse à partir d'un ensemble de promesses.

`Promise.all` : tenue lorsque toutes les promesses sont tenues.

```
// A simple promise that resolves after a given time
const timeOut = (t) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Completed in ${t}`)
    }, t)
  })
}

// Resolving a normal promise.
timeOut(1000)
  .then(result => console.log(result)) // Completed in 1000

// Promise.all
Promise.all([timeOut(1000), timeOut(2000)])
  .then(result => console.log(result))

// ["Completed in 1000", "Completed in 2000"]
```

cf aussi `Promise.race`

await/async

Fonction asynchrone `async function`

`async fonction` permet de déclarer une fonction asynchrone qui renvoie une promesse comme valeur de retour.

Une fonction asynchrone peut contenir une (ou plusieurs) expression `await` qui attend la résolution de la promesse correspondant à l'expression (si l'expression n'est pas une promesse, celle-ci est convertie en une promesse tenue)

```
async function test(){
return 2
}
console.log(test())

//Output
//__proto__: Promise
//[[PromiseState]]: "fulfilled"
//[[PromiseResult]]: 2
```

Le retour est bien une promesse

```
test().then(data => console.log(data))

//Output
//2
```

`await`

Une `async function` peut attendre la résolution (ou le rejet) d'une promesse avec `await`

```
let makeRequest = (() => {
  return new Promise((resolve,reject) => {
    let xhr = new XMLHttpRequest()
    xhr.open("GET","mon/url")
    xhr.onload = ()=>{
      if (this.status == 200)
        resolve(xhr.response)
      else
        reject({
          status: this.status,
          statusText: xhr.statusText
        })
    }
    xhr.onerror = function () {
      reject({
        status: this.status,
        statusText: xhr.statusText
      })
    }
    xhr.send();
  }
}
```

```
async function getInfo(){

let info = await makeRequest()

// le code à partir d'ici
// est exécuté lorsque la promesse
// est tenue ou rejetée

}
```

Remarques :

- si la promesse est rejetée, `await` lève une exception avec la raison. On peut donc utiliser un bloc `try/catch`.
- si la valeur de l'expression `await` n'est pas une promesse, elle est convertie en une promesse résolue ayant sa valeur.