



Cours Javascript

Date : 28/03/2023

Dans ce cours, nous allons apprendre les bases du fetch et des requêtes asynchrones en JavaScript. Nous allons expliquer les différences entre les deux en utilisant des exemples, et nous allons terminer en proposant plusieurs petits exercices pour vous aider à passer de la théorie à la pratique.

Qu'est-ce que le fetch ?

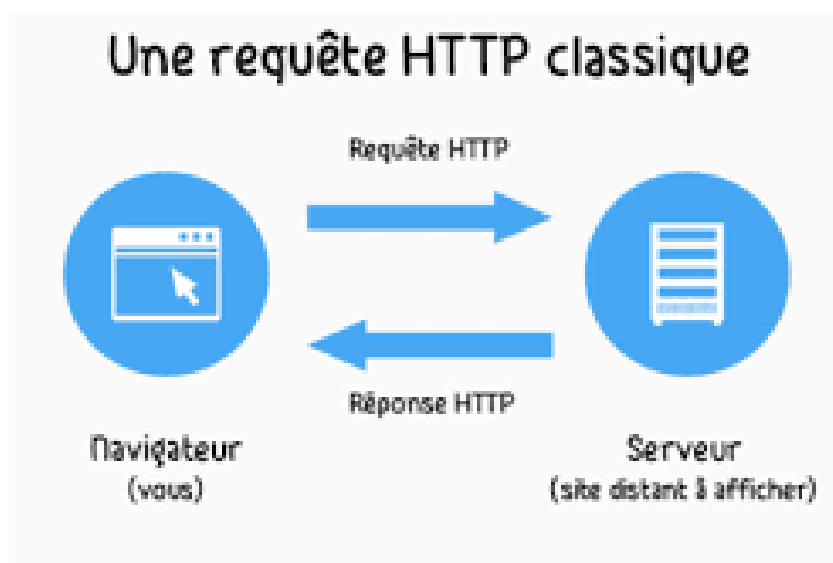
Le fetch est une fonction Javascript qui est utilisée pour effectuer des requêtes HTTP asynchrones vers des ressources sur le Web. Fetch utilise la syntaxe des promesses, ce qui signifie qu'il renvoie une promesse qui est résolue avec la réponse HTTP une fois que la requête est terminée.

Avant de voir un exemple, demandons-nous d'abord qu'est-ce que c'est des requêtes et des réponses HTTP, qu'est-ce que c'est des ressources Web, que veut dire asynchrones et qu'est-ce que c'est une promesse ?

Qu'est-ce que les requêtes et les réponses HTTP ?

HTTP (Hypertext Transfer Protocol) est le protocole standard utilisé pour transférer des données sur le Web. Les requêtes et les réponses HTTP sont échangées entre un client (tel qu'un navigateur Web) et un serveur pour demander ou fournir des ressources Web.

Voici un schéma qui représente les requêtes et les réponses HTTP:



Une requête HTTP est un message envoyé par un client à un serveur pour demander une ressource Web. La requête HTTP contient des informations telles que l'URL de la ressource demandée et les en-têtes (header) de requête, qui décrivent les caractéristiques de la demande.

Une réponse HTTP est un message renvoyé par le serveur en réponse à une requête HTTP. La réponse contient, elle aussi, des informations sur la réponse, qui décrivent les caractéristiques de la réponse.

Si je me rend sur le site de Simplon, dans l'inspecteur de mon navigateur, dans l'onglet Network, je peux voir les en-têtes de la requête et de la réponse:

Sur la capture d'écran ci-dessous, on peut voir que la réponse est de type text/html ou encore à quel moment elle s'est exécutée ici le 27 mars 2023 à 20h10 et 21 secondes etc... On peut voir aussi que la requête contient tout un tas d'informations dans le header, comme par exemple l'URL (simplon.co) la méthode utilisée ici GET, le chemin etc...

The screenshot shows the Simplon website (simplon.co) with a red banner for 'Simplon, formations numériques'. Below the banner, there's a section titled 'Des formations aux métiers du numérique pour tous' with three images. On the right, the browser's developer tools are open to the 'Network' tab, showing a list of resources. The 'Headers' sub-tab is selected for the 'simplon.co' resource, displaying both 'Response Headers' and 'Request Headers'.

Response Headers:

```

cache-control: no-cache, private
content-encoding: gzip
content-type: text/html; charset=UTF-8
date: Mon, 27 Mar 2023 20:10:21 GMT
referrer-policy: strict-origin-when-cross-origin
server: nginx/1.18.0
set-cookie: XSRF-TOKEN=eyJ3pd1I6IjNQT3ZlYVpCVGh1bWVhYXN4I4Q1wvbk3RFc1IyR1pja3h5dGg1R1N0QkQ1LCJtYVh1I1VzgxZGQ1ZTNmZjg4OWUyOGUyMjYyY2xUnpMb1RVRldlVzI0bDFldDRWdm1XdlhmbFA8R0p3XC9MejQ1LCJtYVh1I1ZjQyIn09M30
strict-transport-security: max-age=31536000
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block

```

Request Headers:

```

:authority: simplon.co
:method: GET
:path: /
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/svg+xml,*/*;q=0.8
accept-encoding: gzip, deflate, br
accept-language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7
cache-control: max-age=0
cookie: simplon_co_v2_session=FA18WaPE2qGaJ1tKLTpR1q3wESgEdmVCometothejungle":false,"simplon.co":true}; XSRF-TOKEN=eyJ3pd1I6IjNQT3ZlYVpCVGh1bWVhYXN4I4Q1wvbk3RFc1IyR1pja3h5dGg1R1N0QkQ1LCJtYVh1I1VzgxZGQ1ZTNmZjg4OWUyOGUyMjYyY2xUnpMb1RVRldlVzI0bDFldDRWdm1XdlhmbFA8R0p3XC9MejQ1LCJtYVh1I1ZjQyIn09M30

```

Qu'est-ce que les ressources Web ?

Une ressource Web est un élément identifiable du Web, tel qu'une page Web, une image, un fichier CSS ou JavaScript, ou encore une API. Les ressources Web sont identifiées par des URL qui permettent aux clients de les localiser sur le Web. Le cas des API REST on parlera alors de points de terminaison (end-point) car c'est la fin de l'URL qui va contenir la requête HTTP. Voici un exemple:

<https://jsonplaceholder.typicode.com/posts/1>

Dans cet exemple l'URL de l'API est "<https://jsonplaceholder.typicode.com>" et le end-point est "posts/1" ce qui signifie que la requête demande dans les posts celui qui a l'id 1.

Voici un exemple de requête fetch simple qui récupère un fichier JSON à partir d'une API:

```
1 fetch('https://jsonplaceholder.typicode.com/posts/1')
2   .then(response => response.json())
3   .then(data => console.log(data))
4   .catch(error => console.error(error));
```

Dans cet exemple, nous avons utilisé la méthode `fetch` pour envoyer une requête GET à l'URL <https://jsonplaceholder.typicode.com>. La réponse est ensuite analysée en tant que JSON en appelant la méthode `json()` sur l'objet de réponse. Enfin, les données sont affichées dans la console. Si une erreur se produit, elle est capturée dans la méthode `catch()`.

Qu'est-ce qu'une promesse ?

Une promesse est un objet JavaScript qui représente une valeur qui peut ne pas être disponible immédiatement. Les promesses sont couramment utilisées pour gérer les opérations asynchrones en JavaScript.

Une promesse peut être dans l'un des trois états suivants :

- En attente (pending) : la promesse est en cours d'exécution.
- Résolue (fulfilled) : la promesse a réussi et renvoie une valeur.
- Rejetée (rejected) : la promesse a échoué et renvoie une erreur.

Les promesses sont créées à l'aide de la fonction constructeur `Promise`. La méthode `then()` est utilisée pour gérer la résolution de la promesse, et la méthode `catch()` est utilisée pour gérer les erreurs.

Fetch veut littéralement dire “vas chercher!”



Lorsque l'on jette la balle ou le bâton à son chien, c'est évidemment dans l'espoir que celui-ci aille la chercher. Le moment où il court pour la chercher ou la ramener correspond à pending, nous sommes encore dans l'attente. Quand il nous l'a ramené, ce moment correspond fulfilled, l'instruction qu'on lui a donné est résolue. Mais il arrive parfois que le chien soit distrait par quelque chose et que celui-ci n'aille pas jusqu'à la balle ou la lâche en cours de route pour se diriger autre part, cela correspondrait plutôt à rejected, l'instruction qu'on lui a donnée à échouer.

Qu'est-ce que c'est une requête asynchrone ?

Une requête asynchrone est une requête qui est envoyée sans bloquer l'exécution du reste du code. Cela signifie que le code peut continuer à s'exécuter pendant que la requête est en cours d'exécution, et que la réponse de la requête sera traitée ultérieurement.

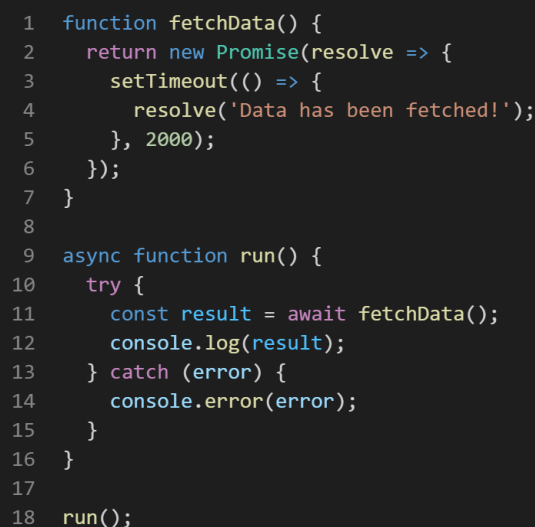
Les requêtes asynchrones sont utilisées pour éviter de bloquer l'interface utilisateur pendant les opérations longues, telles que les appels réseau. Elles utilisent les mots clés Async et Await. Async qui permet d'indiquer qu'il s'agit d'une requête asynchrone et await qui permet de demander au navigateur d'attendre la réponse de cette dernière avant d'exécuter le code qui suit. En d'autre terme:

La déclaration async est utilisée pour marquer une fonction comme asynchrone. Elle permet de retourner une promesse qui sera résolue avec la valeur de retour de la fonction.

Le mot clé await est utilisé à l'intérieur d'une fonction asynchrone pour indiquer que le navigateur doit attendre la résolution d'une promesse avant de continuer l'exécution du

code. Cela permet d'éviter les problèmes de synchronisation des données et de garantir que les résultats sont disponibles avant d'effectuer des opérations supplémentaires.

Voici un exemple d'utilisation de `await` pour attendre la résolution d'une promesse renvoyée par une fonction asynchrone :



```
1 function fetchData() {
2   return new Promise(resolve => {
3     setTimeout(() => {
4       resolve('Data has been fetched!');
5     }, 2000);
6   });
7 }
8
9 async function run() {
10  try {
11    const result = await fetchData();
12    console.log(result);
13  } catch (error) {
14    console.error(error);
15  }
16 }
17
18 run();
```

Dans cet exemple, nous avons créé une fonction `fetchData` qui renvoie une promesse qui se résoudra après 2 secondes avec la chaîne de caractères "Data has been fetched!". La fonction `run` est marquée avec `async` et utilise `await` pour attendre que la promesse retournée par `fetchData` soit résolue. La valeur résolue est ensuite stockée dans la variable `result`, et nous l'affichons dans la console.

Notez que même si nous n'utilisons pas `fetch` dans cet exemple, la fonction `fetchData` renvoie toujours une promesse asynchrone, et nous pouvons attendre la résolution de cette promesse avec `await`.

Mais nous pouvons aussi combiner les deux, à savoir le `fetch` et la requête asynchrone.

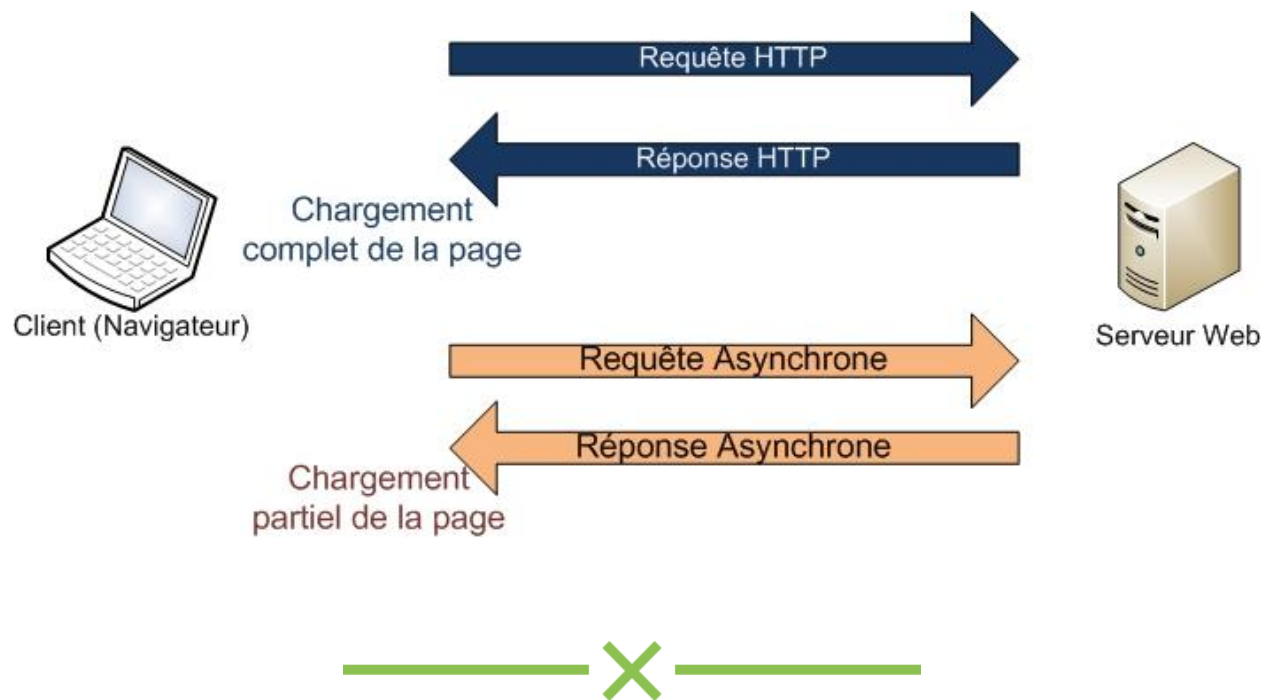
```
1  async function fetchData() {  
2    try {  
3      const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');  
4      const data = await response.json();  
5      console.log(data);  
6    } catch (error) {  
7      console.error(error);  
8    }  
9  }  
10  
11  fetchData();
```

Dans cet exemple, nous avons créé une fonction `fetchData` qui utilise `async` pour indiquer que cette fonction retourne une promesse. Nous avons ensuite utilisé `await` pour attendre la réponse de la requête `fetch` et la réponse est stockée dans la variable `response`. Nous avons ensuite appelé `json()` sur la réponse pour convertir la réponse en objet JavaScript, et stocké les données dans la variable `data`. Enfin, nous avons affiché les données dans la console.

Notez que dans ce cas, nous avons également utilisé un bloc `try...catch` pour gérer les erreurs potentielles qui pourraient survenir lors de l'exécution de la fonction.

L'utilisation de `async` et `await` peut rendre votre code plus facile à lire et à comprendre, car il permet d'éviter l'imbrication de plusieurs fonctions de rappel ou de chaînes de promesses.

Vous l'aurez compris, l'intérêt de ce procédé est de dialoguer avec un serveur sans avoir à recharger la page HTML affichée, comme le ferait une requête HTTP classique envoyée par le navigateur après que l'utilisateur ait cliqué sur un lien. Cela représente une **économie de temps** de chargement sur le navigateur, l'utilisateur attend moins longtemps, bien souvent il ne s'aperçoit pas qu'une requête a été envoyée. On appelle ce type d'échange "**requête asynchrone**" car le navigateur ne va pas bloquer l'utilisateur de la page en attendant la réponse, une fonction dite "callback" est invoquée dès que le serveur a répondu. Elles s'opposent aux requêtes synchrones pour lesquelles le navigateur doit avoir reçu la réponse pour que la page soit utilisable.



Exercices:

1. Utilisez la méthode fetch pour récupérer un fichier texte à partir d'un serveur distant, puis affichez-le dans la console.
2. Écrivez une fonction qui utilise la méthode fetch pour récupérer des données JSON à partir d'une API, puis retourne les données sous forme d'objet JavaScript.
3. Utilisez la méthode fetch pour récupérer une image à partir d'une URL, puis affichez-la dans un élément img de votre page HTML.
4. Écrivez une fonction qui utilise la méthode fetch pour envoyer des données à un serveur distant en utilisant une méthode HTTP POST.
5. Utilisez la méthode fetch pour récupérer des données JSON à partir de l'API themoviedb, puis utilisez ces données pour générer du contenu html en fonction de ce que l'utilisateur aura entré comme titre dans un input que vous aurez prévu à cet effet.