

L'API Fetch, récupérer des données depuis un serveur.

Pré-requis : https://developer.mozilla.org/fr/docs/Web/HTTP/Overview#les_messages_http

Documentations

- `fetch()` : https://developer.mozilla.org/fr/docs/Web/API/Fetch_API/Using_Fetch
- `Promise` : https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Using_promises
- `Promise.prototype.then()` : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise/then
- `Promise.prototype.catch()` : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch
- `Response` : <https://developer.mozilla.org/en-US/docs/Web/API/Response>
- `Response.prototype.json()` : <https://developer.mozilla.org/en-US/docs/Web/API/Response/json>
- `Response.prototype.text()` : <https://developer.mozilla.org/en-US/docs/Web/API/Response/text>

I - Qu'est ce que c'est ?

L'*API Fetch* est un client HTTP utilisable en JavaScript. Au même titre que les objets : `console` , `document` ou `localStorage` l'*API Fetch* est accessible via l'objet `window`.

II - Résumé de `fetch`

Le code suivant affiche le nom des 10 premiers pokémons dans la console.

```
fetch("https://pokebuildapi.fr/api/v1/pokemon/limit/10")
  .then(response=>response.json())
  .then(pokemons=>{
    pokemons.forEach(pokemon=>{
      console.log(pokemon.name);
    });
  });
```

L'API Fetch est utilisable via la méthode `window.fetch` .

```
fetch(url : string,options : Object) : Promise<Response>
```

Comme dit plus haut `fetch` est un client HTTP, ce qui signifie qu'il permet d'envoyer une requête HTTP à un serveur HTTP et ainsi recevoir une réponse HTTP.

Votre navigateur est déjà capable d'effectuer des requêtes HTTP. Il le fait d'ailleurs à chaque chargement de page. `Fetch` permet simplement de récupérer des données sans charger toute une nouvelle page. Les données les plus courantes peuvent être :

- Une liste de produits d'une boutique en ligne qui change en fonction d'un filtre de prix.
- Des suggestions de recherche lorsque l'utilisateur tape au clavier.
- Des données géographiques du serveur de google maps.
- L'envoi d'un message via un formulaire sans recharger toute la page.
- La mise en ligne d'un tweet sur twitter.
- L'affichage des posts Facebook au fur et à mesure du défilement de la page.

Toutes ces opérations nécessitent l'envoi d'une requête HTTP à un serveur, mais nécessitent également de rester sur la page actuelle. Il faut donc envoyer la requête en JavaScript via la méthode `fetch`.

L'envoi de requête HTTP avant `fetch` :

Avant l'envoi de requêtes HTTP en JavaScript, la seule façon de récupérer ou d'envoyer des données à un serveur HTTP était via un formulaire HTML ou l'url et le langage PHP.

PHP s'exécute sur le serveur et génère le HTML côté serveur, le navigateur devait donc recharger toute la page pour afficher les nouvelles données. Ce qui rendait la navigation sur le web bien moins fluide que sur des applications mobile ou bureau.

III - La méthode `fetch` en détail

La méthode `fetch` de l'objet `window` renvoie un objet de la classe `Promise` fournissant une réponse HTTP, soit : "une promesse de réponse HTTP". Elle prend en paramètre obligatoire l'url du serveur (`string`) vers lequel effectuer la requête HTTP.

Définition de la fonction

```
fetch(url [,options]) : Promise<Response>
```

Les paramètres notés entre crochets `[]` sont optionnels, cette norme est utilisée dans la plupart des documentations technique.

```
functionName(param [,paramOptionel1,paramOptionel2])
```

Ici *param* est obligatoire alors que les *paramOptionel1* et *2* non.

Les double points : après la fonction indique le type de la valeur de retour de la fonction, ici un objet de la classe `Promise` qui fournira un objet de la classe `Response` .

```
Math.random() : number  
prompt(message) : string  
Math.round() : number
```

Paramètres

url : string

Une chaine de caractère contenant l'URL de la ressource demandée

Cela peut etre un nom de domaine comme <https://www.leboncoin.fr/voitures/offres>, une adresse ip comme 127.0.0.1:8000/users ou encore la route d'une API REST fournissant du JSON comme l'api pokebuild : <https://pokebuildapi.fr/api/v1/pokemon/limit/10>

options : Object

Un objet JavaScript possédant de nombreux attributs optionnelles, les plus utiles étant :

- **method : string** - Une string contenant la méthode HTTP à utiliser (`GET` , `POST` , `DELETE` , ...).
Par défaut `"GET"` .
- **headers : Headers** - Un objet de la classe `Headers` contenant les eventuels Header HTTP à utiliser comme `"Content-type : application/json"` par exemple pour envoyer du JSON.
- **body : string** - Le body de la requete HTTP, la plupart du temps vous fournirez une string JSON mais d'autre formats sont possibles.

Exemple : Ajouter un produit

Nous voulons rajouter un produit à une base de donnée en utilisant la route `POST /product` d'une api rest, le produit à rajouter est contenu dans le body de la requête HTTP.

Il est d'usage que les serveur demande au client de définir la méthode d'envoi comme POST. Dans les fin ça ne change pas grand chose à l'envoi mais la plupart des serveur (Y compris ce que vous allez concevoir) vous demanderont de préciser la méthode pour leurs simplifier le travail.

Par exemple la route `GET /product` peut envoyer toutes les produits alors que `POST /product` sert à ajouter un nouveau produit.

```
let newProduct = JSON.stringify({name : "Adidias taille 42" , price : 99}); // Je tra
fetch("localhost/product",{
  method : "POST",          // Par défaut fetch fait un GET, donc je précise POST
  body : newProduct
})
.then((reponseHTTP)=>reponseHTTP.json())
.then((data)=>{
  console.log(data);
});
```

Voir la MDN pour plus d'attributs : <https://developer.mozilla.org/fr/docs/Web/API/fetch#init>

Valeur de retour

La méthode `fetch` renvoi une `Promise<Response>` .

La méthode `fetch` permet d'effectuer une requête et renvoie "une promesse de réponse HTTP", c'est à dire un objet de la classe `Promise` qui fournira un objet de la classe `Response` lors de sa résolution.

En JavaScript une `Promise` est un objet que l'on utilise lorsque une action nécessite un certain temps pour s'accomplir comme la lecture d'une fichier ou la réception d'une réponse HTTP.

La problématique du temps de réponse

Une réponse HTTP met à minima plusieurs centaines de millisecondes à arriver alors qu'une ligne de code met environ quelques millisecondes à s'exécuter. Il y a donc un soucis : le code s'exécute plus vite que le temps de réponse.

La solution à un cas comme celui-ci est l'asynchrone. C'est à dire une ligne de code qui ne va pas s'exécuter avant un temps plus ou moins connu. Vous avez déjà connu ce cas de figure par le passé avec pour des méthodes ancestrales du JavaScript comme `addEventListener` ou `setTimeout` .

```

const button = document.querySelector(".button_validate");
console.log("1");
button.addEventListener("click", ()=>{
    console.log("2");
});
console.log("3");
/* --- Résultat --- */
> 1
> 3
> 2

```

Les `console.log()` ne s'exécute pas de haut en bas comme habituellement en programmation. L'action est donc asynchrone.

```

console.log("1");
setTimeout(()=>{
    console.log("2");
}, 1000);
console.log("3");
/* --- Résultat --- */
> 1
> 3
> 2

```

Même chose avec `setTimeout`, "2" sera affiché dans la console une seconde après la fin du script, soit après l'affichage de "3".

Vous remarquez que le code asynchrone utilise les fonctions callback : "Dans X secondes tu appelle cette fonction", "Quand il se passe cet événement tu appelle cette fonction".

Une fonction callback est une fonction passée en paramètre d'une fonction.

IV - Les Promises - l'asynchrone

Théorie

Imaginez.

Vous êtes au restaurant et vous demandez un café au serveur, seulement le serveur ne peut pas faire instantanément apparaître un café sur votre table quand vous le demandez, alors il vous fait une promesse :

- "Je vous ramène votre café dans quelques instants.", quelque instant plus tard le serveur revient avec le résultat de sa promesse.
- "Voici votre café !"

Magnifique !

Lorsque vous avez demandé un café, le serveur vous a fourni un objet : **une promesse** intangible d'un café à venir et en tant qu'être humain vous êtes tout à fait à l'aise avec ce concept ! Vous demandez quelque chose et l'on vous donne, non pas la chose demandée mais une promesse, puis la chose demandée arrive au bout d'un certain temps.

En JavaScript les choses fonctionnent de la même manière. Le serveur est la méthode `fetch` qui vous fournit une promesse.

```
const promise = fetch(url);
```

La méthode `Response.prototype.then()`

```
Promise.prototype.then(callback : Function) : Promise
```

Le résultat de la promesse n'est pas un café mais une réponse HTTP fournie dans la méthode `then`.

```
promise.then(function(response){  
    /*Buvez votre café...*/  
    /*Ou plutôt consommez votre réponse HTTP.*/  
})
```

L'objet `response` fourni dans la méthode `then` est un objet de la classe `Response`. Il possède donc des attributs et des méthodes relative à une réponse HTTP, on peut accéder au code de status (200, 404, 500) aux headers et surtout au body (HTML pour une page, JSON pour une donnée).

Les `Promises` sont une façon moderne de gérer l'asynchrone en JavaScript. Elle s'utilise en deux étapes : la création d'une promesse puis le passage de la fonction callback de résolution qui fournit la donnée demandée via la méthode `then`.

```
console.log("1");
const promise = fetch(url);
promise.then((response)=>{
    console.log("2");
})
console.log("3");
/* --- Résultat --- */
> 1
> 3
> 2
```

La `Promise` "travail" et exécutera votre fonction callback quand elle aura fini de "travailler". Dans le cas de la `Promise` créée par `fetch`, lorsqu'elle recevra la réponse HTTP du serveur.

On récupère le résultat d'une promise en tant que paramètre de la fonction callback présente dans le `then`, ce qui signifie que le résultat est accessible uniquement localement dans cette fonction callback. Cela n'est pas gênant et nous permettra tout de même d'effectuer des `querySelector`, `createElement`, ou `addEventListener`; il nous faudra cependant les écrire dans la fonction callback du `then`.

Enchaîner les `then`

Chose importante à savoir la méthode `then` renvoie la `Promise` appelante ce qui permet d'enchaîner les appels de `then`.

```

console.log("Début");
let promise = fetch("https://api.open-meteo.com/v1/forecast?latitude=48.8534&longitude=2.3488");
promise.then(response=>{
    console.log("1er then !");
})
.then(()=>{
    console.log("2eme then !");
})
.then(()=>{
    console.log("3eme then !");
});
console.log("Fin");

```

```

> Début
> Fin
> 1er then !
> 2eme then !
> 3eme then !

```

Les `then` s'enchaînent à la manière d'une phrase en anglais.

Pour qu'une méthode renvoie son objet appelant il suffit qu'elle renvoie `this`.

```

const compteur = {
    value : 0,
    add(){
        this.value++;
        return this;
    }
};
compteur.add().add().add().add();
console.log(compteur.value); // => 4

```

`this == compteur`.

La valeur de retour de la fonction callback de la méthode `then`

Si l'on place un `return` dans la fonction callback celle-ci va fournir sa valeur de retour en paramètre du `then` suivant.


```

let promise = fetch("https://api.open-meteo.com/v1/forecast?latitude=48.8534&longitude=2.3488");
promise.then(response=>{
    return "Je viens du 1er then !";
})
.then((value)=>{
    console.log(value);    // => "Je viens du 1er then !"
    return "Je viens du 2eme then !";
})
.then((value)=>{
    console.log(value);    // => "Je viens du 2eme then !"
});

```

Retourner une Promise dans la fonction callback de la méthode then

Si l'on retourne une Promise dans la callback d'un then, le then suivant ne fournira pas la Promise mais la donnée promise par la Promise.

C'est exactement ce qui se passe lorsque l'on utilise la méthode `response.json()` pour récupérer un objet JavaScript à partir du body de la réponse HTTP reçu.

```

const promise = fetch(url)
promise.then((response)=>{
    return response.json();    // json() renvoie une Promise
})
.then((data)=>{ // Le résultat de cette Promise est le paramètre data
    console.log(data);
});

```

Ce fonctionnement évite de devoir imbriquer des Promises dans des Promises de cette façon :

```

const promise = fetch(url)
promise.then((response)=>{
    response.json().then(data=>{
        console.log(data);
    });
});

```

Gérer les erreurs avec les Promises

`Promise.prototype.catch(callback : Function) : Promise`

Si une erreur apparaît lors d'une promise la méthode `catch` est disponible. La méthode `catch` fonctionne de la même manière que `then` à la différence que sa fonction callback contient l'erreur.

```
const promise = fetch(url)
promise.then((response)=>{
  return response.json();    // json() renvoie une Promise
})
.then((data)=>{ // Le résultat de cette Promise est le paramètre data
  console.log(data);
})
.catch((error)=>{
  console.error(error.message);
})
```

`error` est un objet de la classe `Error`.

Une bonne pratique consiste à toujours ajouter un `catch` au appel de `Promise`. L'absence de `catch` est un indice d'un code trop fragile.

V - L'objet Response

L'objet `Response` est le resultat d'une `Promise` fabriquée par la méthode `fetch`. C'est la représentation objet d'une réponse HTTP.

Comparons une réponse HTTP brut reçu avec un client HTTP comme REST Client ou Postman avec l'objet réponse reçu par `fetch`.

Soit la requête suivante :

```
GET https://api.open-meteo.com/v1/forecast?latitude=48.8534&longitude=2.3488 HTTP/2
```

Envoyé avec `fetch` comme ceci :

```
fetch("https://api.open-meteo.com/v1/forecast?latitude=48.8534&longitude=2.3488");
```

On reçoit cette réponse HTTP :

HTTP/2 200

Content-type: application/json;

```
{
  "latitude":48.86,
  "longitude":2.3399997,
  "generationtime_ms":0.0020265579223632812,
  "utc_offset_seconds":0,
  "timezone":"GMT",
  "timezone_abbreviation":"GMT",
  "elevation":43.0
}
```

L'objet réponse fournit dans le `then` :

```
Response{
  body: ReadableStream { locked: false }
  bodyUsed: false
  headers: Headers { "content-type" → "application/json; charset=utf-8" }
  ok: true
  redirected: false
  status: 200
  statusText: ""
  type: "cors"
  url: "https://api.open-meteo.com/v1/forecast?latitude=48.8534&longitude=2.3488"
}
```

L'objet réponse contient des attributs analogue à la réponse HTTP brut :

- `200` - l'attribut `status`, `200` signifie un succès.
- `Content-type` : `application/json` - fournit dans l'attributs `headers`

Accéder au body de la réponse

La réponse HTTP présente dans le `then` ne fournit pas directement le body JSON, il nous faut y accéder via la méthode `Response.prototype.json()`.

La conversion d'un body JSON en tableau JS peut être une opération lourde et peut parfois échouer en cas d'erreur dans le JSON, voilà pourquoi il nous faut une promesse pour la conversion.

La méthode `json()` de la classe `Response`

`Response.prototype.json() : Promise<Objet | Array>`

La méthode `response.json()` renvoi une `Promise` qui se résout en un objet JavaScript.

Si le body contient un tableau JSON la `Promise` fournira un tableau JavaScript.

Attention si le body ne correspond pas à une string JSON valide `fetch()` produira une erreur.

```
fetch("https://api.open-meteo.com/v1/forecast?latitude=43.297&longitude=5.3811&hourly=temperature_2m")
  .then(response=>{
    return response.json();
  })
  .then(data=>{
    console.log(data);

    console.log("Température à Marseille aujourd'hui heure par heure");
    data.hourly.temperature_2m.forEach((temp, heure)=>{
      console.log(heure, "h :", temp, data.hourly_units.temperature_2m);
    });
  });
```

Voir la doc de l'api OpenMeteo : https://open-meteo.com/en/docs#latitude=43.297&longitude=5.3811&forecast_days=1

L'objet data ressemble à ceci, il contient des informations météorologique :

```
Object {
  elevation: 30
  generationtime_ms: 0.013947486877441406
  hourly: Object { time: (24) [...], temperature_2m: (24) [...] }
  hourly_units: Object { time: "iso8601", temperature_2m: "°C" }
  latitude: 43.3
  longitude: 5.379999
  timezone: "GMT"
  timezone_abbreviation: "GMT"
  utc_offset_seconds: 0
}
```

Les attributs de l'objets data sont analogues au body de la réponse HTTP brut suivante (celle de l'api OpenMeteo) :

HTTP/1.1 200 OK

Content-Type: application/json; charset=utf-8

```
{
  "latitude": 43.3,
  "longitude": 5.379999,
  "generationtime_ms": 0.016927719116210938,
  "utc_offset_seconds": 0,
  "timezone": "GMT",
  "timezone_abbreviation": "GMT",
  "elevation": 30.0,
  "hourly_units": {
    "time": "iso8601",
    "temperature_2m": "°C"
  },
  "hourly": {
    "time": [
      ...
    ],
    "temperature_2m": [
      5.1,
      6.6,
      6.2,
      5.3,
      4.6,
      3.8,
      4.7,
      6.0,
      6.7,
      7.8,
      9.8,
      8.5,
      8.8,
      9.6,
      9.7,
      9.9,
      9.7,
      9.5,
      9.0,
      8.1,
      7.4,
      7.2,
      7.0,
```

```
    7.1  
  ]  
}  
}
```

La méthode `response.text()` de la classe `Response`

```
Response.prototype.text() : Promise<string>
```

La méthode `response.text()` fonctionne comme `response.json()` à la différence qu'elle ne produit pas un objet JavaScript mais une `string` contenant le body de la réponse HTTP, quel qu'il soit.

Cela permet de récupérer le texte brut de la réponse.

```
const promesse = fetch("http://leboncoin.fr");  
promesse.catch(error=>{  
  console.log(error)  
});  
  
console.log(promesse);
```

1. Testez ce code dans le navigateur.

Le serveur de http de leboncoin.fr est protégé contre les clients http inhabituels, Voilà pourquoi la promesse est à l'état(*state*) *rejected* et qu'une erreur apparait dans la console.

J'ai volontairement omit les appels eux méthode `then` car je sais que la requete echoue dans tout les cas. J'ai juste récupéré (*catch*) sont erreur pour l'afficher dans la console.

Soit là requête http suivante :

```
GET leboncoin.fr HTTP/1.1
```

1. Testez cette requête.

- Téléchargez l'extension VSCode *Rest Client*.
- Ecrivez la requete HTTP précédente dans un fichier `client.http`
- Cliquez sur le lien *Send Request* qui est normalement apparu au dessus du texte de la requete.

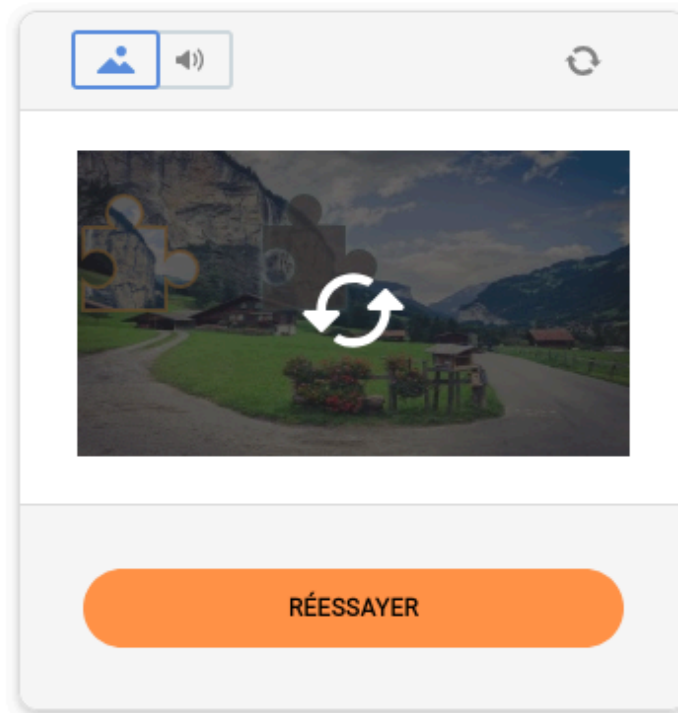
Le serveur de leboncoin.fr nous renvoi du html dans le body.

2. Cherchez le HTML dans le body de la réponse affiché dans VSCode.

Voici le resultat du html une fois ouvert dans un navigateur web.

leboncoin

On s'assure qu'on s'adresse bien à vous, et non pas à un robot.



Pourquoi cette vérification ? Quelque chose dans le comportement du navigateur nous a intrigué.

Diverses possibilités:

- vous surfez et cliquez à une vitesse surhumaine
- quelque chose bloque le fonctionnement de javascript sur votre ordinateur
- un robot est sur le même réseau (IP 37.66.155.77) que vous

Des difficultés pour accéder au site ? [Contacter le support](#)

ID: fcdea39a-ae4-0ac2-8b7d-426ee39711b0

Comprendre la réponse

Soir la réponse http brut de la requete `GET localhost:8080 :`

```
HTTP/1.1 403 Forbidden
Content-Type: text/html; charset=utf-8
Content-Length: 739
Connection: close
Date: Sun, 23 Feb 2025 17:39:33 GMT
set-cookie: datadome=NgrywLNkijyTRUj8PULoVqd1VR687dxkZfQkwabNLa~v6QREydP7Nan2~3XhLCnxGI
strict-transport-security: max-age=15768000
referrer-policy: no-referrer-when-downgrade
content-security-policy: frame-ancestors *.leboncoin.fr *.leboncoin.io *.leboncoin.ci;
content-security-policy-report-only: object-src *.leboncoin.fr *.leboncoin.io *.lebonco:
x-datadome: protected
accept-ch: Sec-CH-UA,Sec-CH-UA-Mobile,Sec-CH-UA-Platform,Sec-CH-UA-Arch,Sec-CH-UA-Full-
charset: utf-8
cache-control: max-age=0, private, no-cache, no-store, must-revalidate
pragma: no-cache
access-control-allow-credentials: true
access-control-expose-headers: x-dd-b, x-set-cookie
access-control-allow-origin: *
x-datadome-cid: AHrlqAAAAAMAZszSnSVyBa4AJUKbTQ==
x-dd-b: 1
X-Cache: Error from cloudfront
Via: 1.1 d5ee2aa873a3cb23609433e0272dd41c.cloudfront.net (CloudFront)
X-Amz-Cf-Pop: CDG50-P2
X-Amz-Cf-Id: Kg200rWNv7PxBi8L23HbQIhHavF_z3r74KAEEut8hka_yxtrIUUJ2A==
```

```
<html lang="en"><head><title>leboncoin.fr</title><style>#cmsg{animation: A 1.5s;}@keyfr
```

Voici la première ligne d'une réponse HTTP.

```
HTTP/1.1 403 Forbidden
```

On y voit :

- la version de HTTP utilisé par le serveur
- Le StatusCode de réponse (comme le fameux 404 Not Found)
- La message explicatif du StatusCode

1. Selon vous, pourquoi la requete de REST Client à été refusé ?(Forbidden : Interdite)

Réponse : Le user-agent(le client http) n'à pas été reconnu comme fiable par le serveur http de leboncoin.fr. Ilà donc renvoyez un status 403 : ressource interdite au client.

Ressource : la donnée que le client

Idée de projet pour apprendre fetch

- Créer un pokedex à partir des données de l'api rest Pokebuild.
- Si vous connaissez un langage back-end comme PHP, créer une boutique ecommerce ou un blog avec un back-end PHP(API REST JSON) et un front-end JavaScript.
- Créer une application météo à partir de l'api rest OpenMeteo et l'api Geolocation de JavaScript.