

Explications des notions clés

Site: [Ada Tech School](#)
Cours: [GHN] [C14] Fetch, async, await
Livre: Explications des notions clés

Imprimé par: Antoine Mourin
Date: mardi 11 novembre 2025, 22:27

Table des matières

1. La boucle d'évènements
2. Fetch
3. Async & await
4. Response

1. La boucle d'évènements

L'asynchrone est ce qui permet au JavaScript d'exécuter du code qui prend potentiellement du temps sans bloquer le flux d'exécution principal.

De base, les lignes de code ou instructions sont exécutées les unes à la suite des autres :

```
console.log("1");  
console.log("2");  
console.log("3");  
// ce code affiche 1, 2, puis 3
```

Il arrive, pour différentes raisons, qu'on ait besoin d'exécuter une ligne ou plusieurs lignes de code "plus tard" : c'est à ce moment là que la boucle d'évènements (Event loop) intervient.

Si tu as déjà manipulé le DOM par exemple, tu as déjà utilisé la boucle d'évènements :

```
<script TODO>  
// index.js  
console.log("1");  
document.querySelector("#myButton").addEventListener("click", () => {  
    console.log("2");  
});  
console.log("3");  
// ce code affiche 1 puis 3 directement  
// il n'affichera 2 que lorsque l'utilisateur clique sur #myButton
```

Ici, on passe une fonction "callback" en paramètre de `addEventListener`. C'est cette fonction qui sera exécutée à chaque fois qu'on clique sur `#myButton`.

Pour que ça fonctionne, JavaScript enregistre ce callback sur l'évent loop, pour que cette partie du code puisse être appelée plus tard.

2. Fetch

La fonction `fetch()` permet d'effectuer une requête HTTP. C'est-à-dire récupérer de la donnée à partir d'une URL.

Le problème, c'est qu'on ne peut pas prédire combien de temps va mettre le serveur à répondre, car cela dépend de la connexion internet. C'est pourquoi `fetch` est exécutée en asynchrone.

Contrairement aux événements du DOM, `fetch` utilise un autre mécanisme pour être exécutée en asynchrone : **les promesses**.

```
const promise = fetch('<https://codepassport.dev/api/offers>'); // on appelle fetch avec une URL d'API
console.log(promise); // affiche la promesse retournée par fetch
```

Comme on peut le voir, on n'a pas accès à la donnée directement puisque pour pouvoir être exécutée de façon asynchrone, `fetch()` retourne une promesse qui englobe le résultat.

0:00 / 0:19

3. Async & await

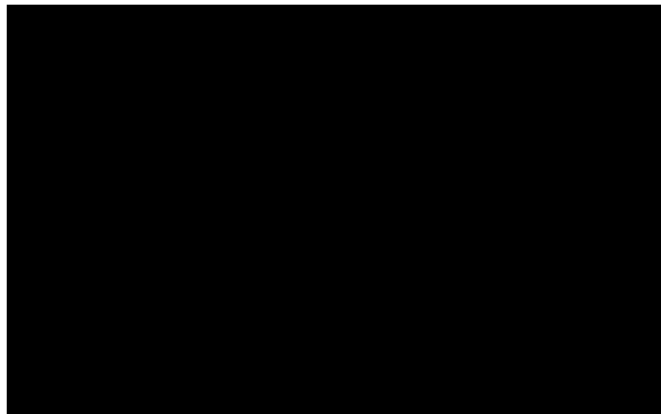
JavaScript a introduit les mots clé `async` et `await` afin de permettre de manipuler les promesses plus facilement dans le langage.

Avant, on utilisait les méthodes `.then()`, `.catch()`, etc. pour manipuler les promesses. Vous tomberez sans doute sur cette forme dans des exemples, mais on vous encourage à plutôt utiliser `async` & `await` qui est la syntaxe recommandée, plus lisible et simple à comprendre.

Pour récupérer la valeur contenue dans une promesse, il faut utiliser le mot clé `await` devant la promesse, par exemple devant l'appel de la fonction `fetch()` qui retourne une promesse :



Attention, comme tu peux le voir, on ne peut pas utiliser `await` n'importe où. Il faut forcément être à l'intérieur d'une fonction asynchrone, c'est-à-dire déclarée avec le mot clé `async` :



4. Response

Enfin, fetch ne retourne pas directement la donnée, mais un objet Response qui contient certaines informations additionnelles (dont on parlera dans une prochaine fiche 🤖). Pour accéder à la donnée, il faut donc utiliser la méthode `.json()` sur la réponse :

