Explications des notions clés

Site: Ada Tech School
Cours: [GHN] [C25] CRUD
Livre: Explications des notions clés

Imprimé par: Antoine Mourin
Date: lundi 26 janvier 2026, 15:02

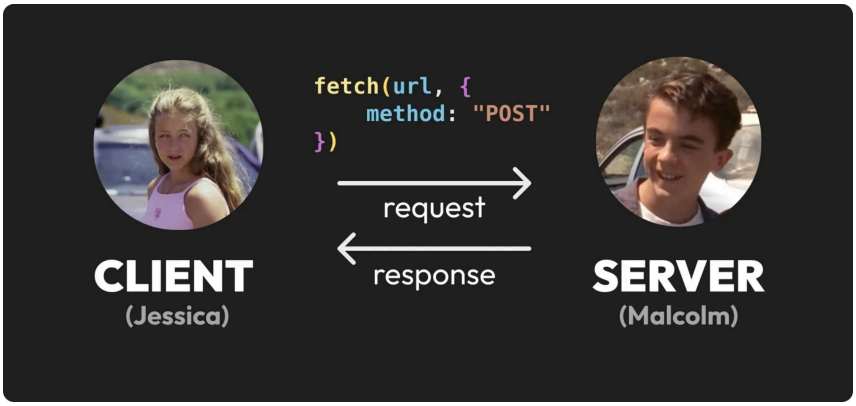
Table des matières

1. Les points clés

On a vu dans les branches précédentes comment utiliser `fetch()` pour récupérer des données depuis une API. Pour ce faire, on utilise le protocole HTTP. Je t'invite à aller voir la fiche sur les protocoles pour te faire une idée de comment ça fonctionne !

Le message envoyé par le client est appelé une **requête** (request) et le message renvoyé par le serveur est appelé une **réponse** (response).

Dans l'image ci-dessus, on peut imaginer que Malcolm est le **serveur** et Jessica le **client**. Jessica envoie une requête sous la forme d'un morceau de papier (protocole HTTP). Malcolm tente de la réceptionner, mais vu qu'il ne la reçoit jamais, il lui renvoie une réponse d'erreur avec le statut 408 (Timeout).



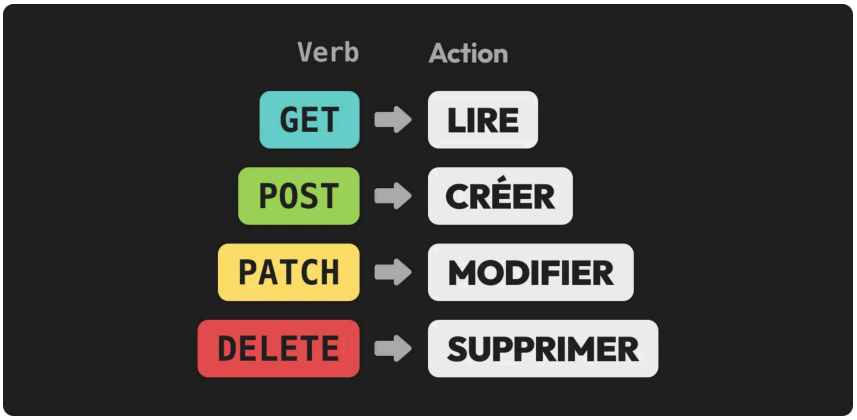
Anatomie d'une requête

Les requêtes HTTP suivent un format spécifique :

- La ligne de départ, ou "**start line**", contient deux informations importantes : l'**URI** de la requête et **le verbe**.

  Si on utilise `fetch(url)` avec un seul paramètre, on utilise par défaut le verbe **GET**. Sinon, on peut le préciser comme dans l'exemple plus haut ☝️ en passant un objet en deuxième paramètre avec `{method: 'POST'}`.

- Le **verbe**, aussi appelé **méthode**, permet de préciser quel type d'action le serveur doit effectuer :



- L'en-tête, ou **header**, permet de configurer la requête en ajouter des informations en plus des données.

  Dans l'exemple, `Accept` précise que le format de la réponse sera en JSON. Dans le cadre d'une requête authentifiées, on aurait les informations de connexion de l'utilisateur.

- Le corps ou **body** correspond aux données envoyées par la requête.

  Dans l'exemple, il s'agit du numéro de Jessica au format JSON mais on pourrait avoir également un fichier ou encore les données d'un formulaire.

2. Anatomie d'une réponse

Le format des réponses est similaire à celui des requêtes avec quelques différences :

La réponse se matérialise par l'objet retourné par la fonction `fetch()` dans sa promesse. Elle contient les différentes informations comme le **statut** ou le **body**. On peut écrire une fonction `giveNumber()` qui fetch la route `/api/giveNumber` et qui affiche des `console.log` en fonction des différents cas.

```
// jessica.js

const API_URL = '<https://www.codepassport.dev>';
async function giveNumber() {
    const response = await fetch(`${API_URL}/api/giveNumber`, {
        method: 'POST',
        headers: {'Content-Type': 'application/json'},
        body: `{ "number": "06XXXXXXXX" }`,
    });

    // affiche la réponse entière :
    console.log('Response: ', response);

     // on vérifie le statut pour voir si Malcolm a reçu le mot :
    if (response.status === 200) {
        console.log(`Malcolm is 🥳!`);
    } else if (response.status === 408) {
        console.log(`Malcolm is 😭`);
    } else {
        console.log('Something went wrong 🤔');
    }
    // on recupère le body (qui est un json) :
    const body = await response.json();
    console.log(body);
}

// on appelle la fonction
giveNumber();
```

💡 N'hésite pas à copier le code et à l'exécuter pour voir le contenu de la réponse !

3. Le côté serveur

Avec le code ci-dessus, nous avons vu le point de vue du client (Jessica), dans notre exemple. Mais il existe aussi un code qui correspond au serveur (Malcolm) qui envoie la réponse (c'est-à-dire le numéro de téléphone.

On peut par exemple utiliser le framework express en JavaScript pour écrire la route `/api/giveNumber` qui renvoie la réponse.

```
 // malcolm.js

const express = require('express');
const app = express();
const port = 3000;

// Crée la route qui est censée envoyer le numéro :
app.post('/api/giveNumber', (req, res) => {
  // Simule un délai d'attente pour déclencher l'erreur 408 :
  setTimeout(() => {
      // configure la réponse à renvoyer :
    res.status(408).json({
      error: "caught by 🐶",
      girlfriend: false,
    });
  }, 1000);
});

app.listen(port, () => {
  console.log(`Server started on <http://localhost>:${port}`);
});
```

Installation

Pour que le code ci-dessus fonctionne, il faut installer express en tapant la commande `npm install express` dans ton terminal dans le même dossier que `malcolm.js`.

Ensuite, tu peux démarrer le serveur avec la commande `node malcolm.js`.

> 💡 Tu peux tester ce code pour la partie serveur. Attention, si tu veux que ça marche avec le client `jessica.js` il faudra changer l'url par `http://localhost:3000` dans son code