Promises Async/await

Faire de la programmation concurrentielle en JS

Promises

Promises

Une promesse est un type qui représente une action (fonction) qui va se dérouler de façon asynchrone.

Une promesse peut réussir (resolve) ou rater (reject).

On utilise les promesses sur les calls qui prennent du temps et bloquent le processus, comme les requêtes ou encore les IO système.

Les promises peuvent être chaînées.

En somme, elles permettent d'éviter les “callback hell”, qui surviennent quand beaucoup de fonctions asynchrones utilisent la méthode du callback.

Créer une promesse

Bien que la plupart du temps on ne traite que de promesses déjà existantes, il est possible de créer une promesse :

```
const readFile = (filename) => {
    return new Promise((resolve, reject) => {

        fs.readFile(path.join(__dirname, filename),
'utf8', (err, data) => {
            if (err) {
                reject(err);
            }
            resolve(data);
        });
    });
}
```

Ici, on crée une promesse de la façon la plus primitive.

Promisify

Il existe une fonction qui permet de transformer automatiquement en promesse une fonction avec un simple callback :

```
const fs = require('fs');
const util = require('util');


const readFile = util.promisify(fs.readFile);


await readFile('./index.js', 'utf8')
```

Utiliser une promise

Une promise est lancée au moment où elle est interprétée, mais son résultat n'existe pas immédiatement.

```
readFile('file1.txt')
    .then((data) => {
        console.log(data);
    })
    .catch((err) => {
        console.log(err);
    });
```

then() Permet de récupérer le résultat de la promesse si tout s'est bien passé, et de chaîner d'autres actions.

catch() Permet de récupérer l'erreur survenue si la promise a été rejetée.

Utiliser async/await

les mots clés async/await permettent de reporter le traitement d'une promesse à un autre endroit dans l'implémentation : la ligne contenant le await est bloquante pour le programme et ce blocage doit être traité ailleurs.

```
const readFiles = async () => {
    try {
        const data = await readFile('file1.txt');
        console.log(data);
    } catch (err) {
        console.log(err);
    }
}
```

async indique qu'une fonction contient un "await" et donc qu'elle est bloquante.

await indique que l'on attendra la résolution de l'expression qui le suit, avant de continuer l'exécution du programme : il signifie que l'on bloque l'exécution du code en attendant l'arrivée des données.

Rassembler une liste de promesses en une seule

Il arrive que l'on fasse des requêtes groupées, dont on doit attendre chacun des résultats. Pour cela, on a une fonction :

```
const fileNames = ['file1.txt', 'file2.txt', 'file3.txt'];


const promises = fileNames.map((filename) => {
   return readFilePromise(filename);
});


await Promise.all(promises)
```

Sleep()

Afin de créer manuellement un temps d'attente, d'autres langages possèdent une fonction `sleep()` ou encore `wait()`. En javascript, nous avons setTimeout(), qui prend un callback et l'exécute après un temps donné

```
const shortTimeout = () => setTimeout(() => {
   console.log('short timeout');
   }, 1000);
const longTimeout = () => setTimeout(() => {
   console.log('long timeout');
   }
   , 5000);
longTimeout()
shortTimeout();
// short timeout
// long timeout
```

Pratique

Créer une promesse qui dure 2 secondes avant de résoudre.

Créer une seconde promesse qui dure 1 seconde avant de résoudre.

Chaîner(then) la seconde promesse avec la première.

Laquelle résoudra en premier ?

Pratique

Créer un array de nombres
Créer une fonction asynchrone qui prend un nombre en paramètre et attends ce nombre de secondes avant de le printer dans la console.

Exécuter cette fonction sur chaque élément du tableau.