7 minutes

Les fonctions asynchrones avec async / await

Cette leçon est extrêmement importante car `async / await` est utilisé énormément dans une application `JavaScript` moderne.

Les fonctions asynchrones

Une fonction asynchrone s'exécute de façon asynchrone en utilisant une promesse.

Pour déclarer une fonction asynchrone, il suffit d'utiliser le mot clé `async` devant `function` :

```
async function asynshrone() {
  return 42;
}
```

Cela équivaut en fait à faire :

```
function asynshrone() {
  return Promise.resolve(42);
}
```

Autrement dit le mot clé `async` permet de s'assurer qu'une fonction va toujours retourner une promesse.

Attendre le résultat d'une fonction asynchrone

L'opérateur `await` permet d'attendre la résolution d'une promesse uniquement dans une fonction asynchrone.

```
async function asynshrone() {
  const valeur = await promesse;
}
```

C'est une syntaxe extrêmement concise et élégante : **elle permet d'attendre l'acquittement d'une promesse** et d'obtenir sa valeur si elle est tenue.

Il est important de comprendre que cela ne rend pas votre code moins performant, le `thread` n'est pas bloqué comme pour une promesse standard et d'autres traitements, en dehors de la fonction asynchrone, peuvent avoir lieu.

Que se passe t-il si la promesse est rejetée ?

Une erreur sera automatiquement levée par le moteur (nous verrons en détails les erreurs dans un chapitre ultérieur).

Il est possible de la gérer dans un `catch()` :

```
const p1 = Promise.reject("Aïe !");
async function f() {
  const valeur = await p1;
  console.log(valeur); // rien
}
f().catch(e => console.error(e)); // "Aïe !"
```

Mais la méthode recommandée est l'utilisation de blocs `try / catch` qui permet de récupérer les erreurs du bloc `try` dans le bloc `catch` :

```
const p1 = Promise.reject("Aïe !");
async function f() {
  try {
    const valeur = await p1;
  } catch (err) {
    console.error(err);
  }
}
f();
```

Cette syntaxe est recommandée car elle est claire et concise.

A noter que le bloc `catch` n'a absolument rien à voir avec la méthode `catch()`. Il s'agit d'un opérateur qui n'est pas propre aux promesses et qui doit être utilisé uniquement avec `try` (nous y reviendrons en détails).

A noter également que `finally` dans les exemples ci-après, n'a rien à voir non plus avec la méthode `finally()` des promesses. Il s'utilise également uniquement avec des blocs `try / catch`.

Traitements séquentiels, concurrents et parallèles

Des traitements asynchrones peuvent être **séquentiels** : c'est-à-dire qu'ils sont effectués les uns après les autres.

Ils peuvent être **concurrents**, c'est-à-dire qu'ils s'effectuent en même temps et qu'ils sont indépendants, nous n'avons pas besoin d'obtenir leur résultats en même temps.

Ils peuvent être **parallèles**, c'est-à-dire qu'ils s'effectuent en même temps et que nous avons besoin d'attendre les résultats de tous les traitements pour passer aux instructions suivantes. A noter que des traitements parallèles sont concurrents car ils s'exécutent en même temps.

A noter également que les distinctions que nous faisons sont valables dans ce contexte et ne sont pas à généraliser à la programmation en général.

Traitements séquentiels

Voici un exemple de traitements séquentiels :

```
const p1 = () => new Promise(resolve => setTimeout(() => resolve(42), 2500));
const p2 = () => new Promise(resolve => setTimeout(() => resolve(12), 2000));

async function sequentiel() {
  try {
    const val1 = await p1();
    console.log(val1);
    const val2 = await p2();
    console.log(val2);
  } catch (err) {}
}
sequentiel();
```

Ici nous aurons la valeur 42 au bout de 2,5 secondes, puis la valeur 12 au bout de 2 secondes supplémentaires. Nous attendons en effet l'acquittement de la promesse retournée par `p1` avant d'exécuter la promesse retournée par la fonction `p2`.

Faites très attention ! Prenez le code suivant :

```
const p1 = new Promise(resolve => setTimeout(() => resolve(42), 2500));
const p2 = new Promise(resolve => setTimeout(() => resolve(12), 2000));

async function sequentiel() {
  try {
    const val1 = await p1;
    console.log(val1);
    const val2 = await p2;
```

```
    console.log(val2);
  } catch (err) {}
}
sequentiel();
```

Il n'a absolument pas le même résultat ! Dans ce cas, les promesses `p1` et `p2` commencent toute suite leur exécution. Avant même l'exécution de la fonction `sequentiel()`.

Nous aurons donc 42 et immédiatement après 12. Car lorsque le promesse `p1` est tenue, la promesse `p2` est déjà tenue depuis 500 millisecondes.
https://codesandbox.io/embed/js-c14-l6-1-zjsbl

Traitements concurrents

Voici un exemple de traitements concurrents :

```
const fp1 = () => new Promise(resolve => setTimeout(() => resolve(42), 2500));
const fp2 = () => new Promise(resolve => setTimeout(() => resolve(12), 2000));

async function concurrent() {
  try {
    const p1 = fp1();
    const p2 = fp2();
    const val1 = await p1;
    afficherLeResultat("p1", val1, true);
    const val2 = await p2;
    afficherLeResultat("p2", val2, true);
    toggleLoader();
    clearTimer();
  } catch (err) {}
}
concurrent();
```

Ici, les fonctions retournent les promesses qui commencent immédiatement leurs exécutions concurrentes. Nous aurons donc 42 et 12 au bout de 2,5 secondes car lorsque `p1` est tenue, `p2` est tenue depuis 500 millisecondes.
https://codesandbox.io/embed/js-c14-l6-2-6by2d

Traitements parallèles

Voici un exemple de traitements parallèles :

```
const fp1 = () => new Promise(resolve => setTimeout(() => resolve(42), 2500));
const fp2 = () => new Promise(resolve => setTimeout(() => resolve(12), 2000));

async function parrallel() {
  try {
    const [val1, val2] = await Promise.all([fp1(), fp2()]);
    afficherLeResultat("p2", val2, true);
    afficherLeResultat("p1", val1, true);
    toggleLoader();
    clearTimer();
  } catch (err) {
    console.log(err);
  }
}
parrallel();
```

Ici nous attendons le résultats de deux promesses lancées en même temps avant de procéder aux restes des instructions.

Nous utilisons une affectation par décomposition pour obtenir directement les résultats dans deux constantes. En effet, pour rappel, `Promise.all()` retourne un tableau de résultats.
https://codesandbox.io/embed/js-c14-l6-3-s7uek