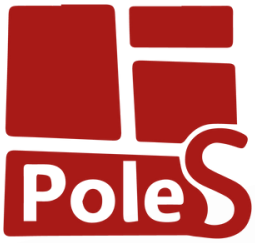

Méthodes asynchrones et Promesses

Par OMISCAR Johane
2024

Introduction à l'Asynchrone

En JavaScript, les opérations qui prennent du temps, les requêtes API ou les lectures de fichiers, peuvent être traitées de manière asynchrone pour éviter de bloquer l'exécution du code.
Les méthodes asynchrones utilisent des promesses pour gérer ces opérations.

Les Promesses

Une promesse est un objet représentant la réussite (ou l'échec) éventuel(le) d'une opération asynchrone. Une promesse peut être dans l'un des trois états suivants :

- **En attente (Pending)** : L'opération est en cours.
- **Réussie (Fulfilled) :** L'opération a réussi, et la promesse a une valeur.
- **Échouée (Rejected) :** L'opération a échoué, et la promesse a une erreur.

Création d'une Promesse

Pour créer une promesse, utilisez le constructeur **Promise** avec une fonction de callback qui prend deux arguments : **resolve** et **reject**.

```
let promesse = new Promise((resolve, reject) => {
    // Simuler une opération asynchrone avec setTimeout
    setTimeout(() => {
        resolve("Données reçues !");
    }, 2000);
});
```

Consommer une Promesse

Vous pouvez consommer (en récupérer les données) une promesse en utilisant les méthodes **then()** et **catch()**.

```
promesse
    .then((resultat) => {
        console.log(resultat); // Affiche "Données reçues !" après 2 secondes
    })
    .catch((erreur) => {
        console.error(erreur);
    });
```

Les méthodes asynchrones avec async et await

Les mots-clés async et await facilitent l'écriture du code asynchrone en permettant d'utiliser une syntaxe plus lisible, semblable à du code synchrone.

Le mot-clé async

Une fonction marquée avec async retourne automatiquement une promesse. Le résultat de la fonction est enveloppé dans une promesse qui se résout en ce que la fonction retourne.

```
async function hello() {
    return "Hello, monde !";
}

hello().then((resultat) => console.log(resultat)); // Affiche "Hello, monde !"
```

Le mot-clé await

À l'intérieur d'une fonction async, vous pouvez utiliser await pour attendre la résolution d'une promesse avant de continuer.

```
async function getData() {
    let promesse = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Données reçues !");
        }, 2000);
    });

    let resultat = await promesse;
    console.log(resultat); // Affiche "Données reçues !" après 2 secondes
}

getData();
```

Gestion des erreurs avec try...catch

Les erreurs dans les fonctions async peuvent être capturées avec un bloc try...catch.

```
async function getData() {
    try {
        let promesse = new Promise((resolve, reject) => {
            setTimeout(() => {
                reject("Erreur lors de la récupération des données !");
            }, 2000);
        });

        let result = await promesse;
        console.log(result);
    } catch (err) {
        console.error(err); // Affiche "Erreur lors de la récupération des données !"
    }
}

getData();
```

Résumé

- **Promesses :** Un objet qui représente l'achèvement ou l'échec d'une opération asynchrone.
  - **Création : new Promise((resolve, reject) => { /* code */ })**
  - **Consommation** : **then()** pour les réussites et **catch()** pour les erreurs.
  - **Chaînage** : **then()** peut être utilisé pour effectuer plusieurs opérations en série.
- **async** et **await** : Facilite la gestion des opérations asynchrones avec une syntaxe plus lisible.
  - **async** : Déclare une fonction qui retourne une promesse.
  - **await** : Attend la résolution d'une promesse avant de continuer.

Définitions :

- **Fonction Synchrone :** Une fonction qui fait une chose à la fois. Le programme attend que cette fonction termine son travail avant de passer à la suite.
  - *Exemple :* Si une fonction synchrone doit lire un fichier, le programme attend que la lecture soit terminée avant de continuer.

- **Fonction Asynchrone :** Une fonction qui peut faire plusieurs choses en même temps. Le programme peut continuer à s'exécuter pendant que cette fonction fait son travail en arrière-plan.
  - *Exemple* : Si une fonction asynchrone demande des données à un serveur, le programme peut continuer à faire d'autres choses pendant que l'attente des données se passe en arrière-plan.