Programmation Asynchrone

1WEBD – Javascript Web Development

Sommaire

1. Introduction.
2. Les Promesses.
3. Async Await.

1. Introduction

1. Introduction

Definition

- **Asynchronicité**: En programmation, l'asynchronicité se réfère à la capacité d'exécuter des tâches ou des opérations en arrière-plan, sans interrompre le flux principal d'exécution du programme. Cela permet à l'application de rester réactive, en particulier lorsqu'elle traite des tâches longues ou dépendantes de ressources externes comme les requêtes réseau

1. Introduction

Définition

- **Synchronicité vs. Asynchronicité**: Dans un contexte synchrone, les opérations s'exécutent l'une après l'autre, chaque opération devant être terminée avant que la suivante ne commence. En contraste, en mode asynchrone, une opération peut démarrer et le contrôle est immédiatement rendu au flux principal, permettant à d'autres opérations de s'exécuter en parallèle
- Pour plus d'information, regarder la video "what the heck is the event loop anyway"

1. Introduction

Exemple

- **Requêtes Réseau**: lors de la récupération de données depuis une API ou un serveur externe
- **Opérations de Fichiers**: lors de la lecture ou de l'écriture de fichiers volumineux, particulièrement pertinent en Node.js
- **Delais Programmés**: utilisation de **setTimeout** ou **setInterval** pour exécuter du code après un certain délai ou à des intervalles réguliers

1. Introduction

Avantages

- **Amélioration de la Réactivité:** Les applications restent réactives et interactives, même lors du traitement de tâches lourdes
- **Meilleure Expérience Utilisateur:** L'utilisateur n'est pas bloqué par des chargements ou des traitements longs et peut continuer à interagir avec d'autres parties de l'application
- **Efficient sur les Ressources:** Permet de mieux gérer les ressources, en effectuant des opérations en arrière-plan sans perturber les autres fonctionnalités

1. Introduction

2. Les Promesses

2. Les Promesses

Définition

- **Promesse (Promise)**: En JavaScript, une promesse est un objet représentant l'achèvement ou l'échec éventuel d'une opération asynchrone. Elle permet de gérer de façon plus flexible le résultat d'opérations qui ne sont pas immédiatement complétées
- On peut imaginer ça comme une « vraie » promesse: tant qu'elle n'est pas tenue ou rompue, on attend

2. Les Promesses

Structure

- Une promesse en JavaScript est créée en utilisant le constructeur **new Promise**. Ce constructeur prend une fonction exécuteur avec deux arguments: **resolve** et **reject**
- Si la promesse est résolue, on appelle **resolve**
- Si la promesse est rejetée, on appelle **reject**

2. Les Promesses

Structure

```
let maPromesse = new Promise((resolve, reject) => {
    // Code asynchrone ici
    if (/* condition de réussite */) {
        resolve(valeur); // La promesse est résolue avec une valeur
    } else {
        reject(raison); // La promesse est rejetée avec une raison
    }
});
```

2. Les Promesses

Etat

- **En Attente (Pending)**: L'état initial de la promesse, quand elle est encore en cours d'exécution
- **Accomplie (Fulfilled)**: L'état de la promesse quand l'opération asynchrone se termine avec succès
- **Rejetée (Rejected)**: L'état de la promesse quand l'opération échoue ou rencontre une erreur

2. Les Promesses

Utilisation

- Lorsqu'une promesse est résolue, la méthode **.then()** est utilisée pour gérer la valeur résultante

```
maPromesse.then(valeur => {
    console.log("Résultat : ", valeur);
});
```

2. Les Promesses

Utilisation

- Pour gérer les erreurs lorsqu'une promesse est rejetée, on utilise la méthode **.catch()**

```
maPromesse.catch(erreur => {
  console.log("Erreur:", erreur)
})
```

2. Les Promesses

Utilisation

```
new Promise((resolve, reject) => {
    setTimeout(() => {
            const valid = Math.random() > 0.5;
        if (valid) {
            resolve("ok")
        } else {
            reject("ko")
        }
    }, 2000)
}).then((value) => {
  console.log("success", value)
}).catch((error) => {
  console.log("error", error)
})
```

2. Les Promesses

Promesses Simultanées

- **Promise.all** permet d'exécuter plusieurs promesses en parallèle et d'attendre que toutes soient résolues. Il retourne un tableau avec les résultats de toutes les promesses
- Si toutes les promesses sont résolues, **Promise.all** retourne un tableau de leurs résultats. Si une des promesses est rejetée, **Promise.all** est rejeté immédiatement avec la raison du premier rejet
- Plutôt pratique pour utiliser une fonction .map avec des opérations asynchrones

2. Les Promesses

Promesses Simultanées

```
Promise.all([promesse1, promesse2, promesse3])
    .then(resultats => {
        // Traiter les résultats ici
    })
    .catch(erreur => console.error("Erreur dans les promesses :", erreur));
```

2. Les Promesses

Méthodes asynchrones en JS

- JS fournit déjà certaines méthodes qui fonctionnent comme des fonctions asynchrones
- setTimeout => exécute une fonction (callback) après un certain délai une fois
- setInterval => exécute une fonction (callback) tous les n millisecondes
- Fetch => exécute une requête réseau (on en parle dans le prochain cours)
- Certaines API Navigateur (comme celle pour obtenir la localisation de l'utilisateur)

2. Les Promesses

3. Async Await

3. Async Await

Introduction

- **async** et **await** sont des ajouts modernes à JavaScript qui simplifient l'écriture de code asynchrone
- Ils permettent d'écrire des opérations asynchrones de manière plus lisible, en utilisant une syntaxe qui ressemble davantage à du code synchrone traditionnel

3. Async Await

Conversion

- En déclarant une fonction avec **async**, elle retourne automatiquement une promesse
- Les opérations asynchrones à l'intérieur de la fonction peuvent être traitées en utilisant **await**, qui suspend l'exécution de la fonction jusqu'à ce que la promesse soit résolue ou rejetée.

3. Async Await

Conversion

```
async function maFonctionAsync() {
    let resultat = await uneOperationAsynchrone();
    console.log(resultat);
}
```

3. Async Await

Utilisation de await

```
async function obtenirDonnees() {
    try {
        let donnees = await getData(url);
        let resultat = await convertData(donnees);
        return resultat;
    } catch (erreur) {
        console.error("Erreur lors de la récupération des données :", erreur);
    }
}
```

3. Async Await

Gestion d’Erreur

- Dans un contexte asynchrone, **try...catch** offre une manière structurée et lisible de gérer les erreurs
- Lorsqu'une promesse à l'intérieur d'un bloc **try** est rejetée, l'exécution saute automatiquement au bloc **catch** correspondant
- L'erreur est alors passé en paramètre au block **catch**

3. Async Await

Gestion d’Erreur

```
async function chargerDonnees() {
    try {
        let resultat = await doSomething(); // Appel de promesse
        // Traitement des données...
    } catch (erreur) {
        console.error("Erreur lors du chargement des données :", erreur);
        // Gestion de l'erreur...
    }
}
```

3. Async Await

Patterns Avancées avec les Promesses

- **Promise.all**: utilisée lorsqu'il est nécessaire que toutes les promesses soient résolues avec succès. Si l'une des promesses est rejetée, **Promise.all** échoue immédiatement avec la raison du rejet de la première promesse qui échoue

3. Async Await

Patterns Avancées avec les Promesses

```
let promise1 = Promise.resolve(3);
let promise2 = 42;
let promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then(values => {
  console.log(values); // [3, 42, "foo"]
}).catch(reason => {
  console.log(reason);
});
```

3. Async Await

Patterns Avancées avec les Promesses

- **Promise.allSettled**: Cette méthode est utilisée pour attendre que toutes les promesses soient soit résolues, soit rejetées. Contrairement à **Promise.all**, elle ne court-circuite pas après une première promesse rejetée

3. Async Await

Patterns Avancées avec les Promesses

```
let promise1 = Promise.resolve(3);
let promise2 = new Promise((resolve, reject) => setTimeout(reject, 100, 'foo'));
let promise3 = new Promise((resolve, reject) => setTimeout(resolve, 100, 'bar'));

Promise.allSettled([promise1, promise2, promise3])
  .then(results => results.forEach(result => console.log(result.status, result.value,
result.reason)));
  // "fulfilled" 3 undefined
  // "rejected" undefined "foo"
  // "fulfilled" "bar" undefined
```

3. Async Await

Patterns Avancées avec les Promesses

- **Promise.race**: utilisé lorsque vous avez plusieurs promesses et que vous souhaitez réagir dès que la première d'entre elles est résolue ou rejetée. Contrairement à **Promise.all** ou **Promise.allSettled**, **Promise.race** se termine dès que l'une des promesses de l'itérable est résolue ou rejetée, avec la valeur ou la raison de cette promesse

3. Async Await

Patterns Avancées avec les Promesses

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'un');
});

let promise2 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, 'deux');
});

Promise.race([promise1, promise2]).then(value => {
  console.log(value); // "deux" - promise2 est plus rapide
}).catch(reason => {
  console.log(reason);
});
```

3. Async Await