Synchrone et Asynchrone : Gérer le Temps en JavaScript

Comprendre le Flux d'Exécution Asynchrone

Pour les étudiants de 1ère année Supinfo

Au Programme

1. Introduction - Définition et Cas d'Usage de l'Asynchrone
2. Les Promesses - Structure et Utilisation
3. Async / Await - Syntaxe Moderne et Gestion d'Erreur

Introduction : Définition

QU'EST-CE QUE L'ASYNCHRONICITÉ ?

L'**asynchronicité** est la capacité d'**exécuter des tâches 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.

POURQUOI C'EST IMPORTANT ?

- **Réactivité** - L'interface utilisateur reste fluide et réactive
- **Performance** - Les tâches longues n'immobilisent pas l'application
- **Expérience Utilisateur** - L'utilisateur peut continuer à interagir avec l'app
- **Gestion des Ressources** - Meilleure utilisation des ressources disponibles

Synchrone vs. Asynchrone

Synchrone

Les opérations s'exécutent **l'une après l'autre**. Chaque opération doit être **terminée avant** que la suivante ne commence. Le programme **attend** le résultat de chaque opération.

- Exécution séquentielle
- Bloquant (blocking)
- Facile à comprendre
- Peut être lent

```
Opération 1 ✓
Opération 2 ✓
Opération 3 ✓
Résultat Final
```

Asynchrone

Une opération **démarre** et le contrôle est **immédiatement rendu** au flux principal. D'autres opérations peuvent s'exécuter "en parallèle". Le programme **ne bloque pas**.

- Exécution non-bloquante
- Non-bloquant (non-blocking)
- Plus rapide et réactif
- Légèrement plus complexe

```
Opération 1 (en cours...)
Opération 2 (en cours...)
Opération 3 (en cours...)
Résultats au fur et à mesure
```

Cas d'Usage et Avantages

Cas d'Usage Concrets

- **Requêtes Réseau**
  Récupération de données depuis une API ou un serveur externe sans bloquer l'interface
- **Opérations de Fichiers**
  Lecture ou écriture de fichiers volumineux, particulièrement pertinent en Node.js
- **Délais Programmés**
  Utilisation de setTimeout ou setInterval pour exécuter du code après un délai ou à des intervalles réguliers
- **Requêtes Base de Données**
  Interrogation de bases de données sans bloquer les autres opérations
- **Géolocalisation**
  Récupération de la position de l'utilisateur via les API du navigateur

Avantages de l'Asynchrone

- **Réactivité Améliorée**
  Les applications restent réactives et interactives, même lors du traitement de tâches lourdes
- **Meilleure UX**
  L'utilisateur n'est pas bloqué par des chargements longs et peut continuer à interagir
- **Gestion des Ressources**
  Meilleure utilisation des ressources en effectuant des opérations en arrière-plan
- **Scalabilité**
  Possibilité de gérer plusieurs opérations simultanément sans ralentir l'application
- **Code Plus Lisible**
  Avec async/await, le code asynchrone ressemble à du code synchrone traditionnel

Les Promesses : Définition et Structure

QU'EST-CE QU'UNE PROMESSE ?

Une **promesse (Promise)** est un objet JavaScript qui représente 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 l'imaginer comme une "vraie" promesse : tant qu'elle n'est pas tenue ou rompue, on attend.

STRUCTURE D'UNE PROMESSE

Une promesse en JavaScript est créée en utilisant le constructeur **new Promise**. Ce constructeur prend une fonction exécuteur avec deux arguments : **resolve** et **reject**.

resolve

Appelé quand l'opération asynchrone réussit. Marque la promesse comme accomplie.

reject

Appelé quand l'opération asynchrone échoue. Marque la promesse comme rejetée.

EXEMPLE SIMPLE

```
const maPromesse = new Promise((resolve, reject) => {
// Opération asynchrone
const succes = true;

if (succes) {
resolve("Opération réussie !");
} else {
reject("Opération échouée !");
}
});
```

Les Promesses : Les 3 États

Pending (En Attente)

L'état initial d'une promesse, quand elle est encore en cours d'exécution.

- La promesse a été créée
- L'opération asynchrone est en cours
- Ni resolve ni reject n'ont été appelés

Exemple : Une requête réseau qui est en train de charger

Fulfilled (Accomplie)

L'état de la promesse quand l'opération asynchrone se termine avec succès.

- resolve() a été appelé
- La promesse a une valeur
- .then() est exécuté

Exemple : Les données de l'API ont été reçues avec succès

Rejected (Rejetée)

L'état de la promesse quand l'opération échoue ou rencontre une erreur.

- reject() a été appelé
- La promesse a une raison d'erreur
- .catch() est exécuté

Exemple : La requête réseau a échoué (erreur réseau, timeout)

💡 **Point Important**

Une promesse ne peut passer que d'un état à un autre **une seule fois**. Une fois qu'elle est Fulfilled ou Rejected, elle ne peut plus changer d'état. C'est ce qu'on appelle une promesse **settled**.

Les Promesses : Utilisation

.then()

La méthode **.then()** est utilisée pour gérer la valeur résultante lorsqu'une promesse est **résolue avec succès**. Elle prend une fonction callback qui reçoit le résultat.

- Exécutée quand la promesse est **fulfilled**
- Reçoit la valeur de **resolve()**
- Retourne une nouvelle promesse
- Peut être chaîné

```
const promise = fetch('/api/data');

promise.then((data) => {
console.log('Succès:', data);
return data.json();
});
```

.catch()

La méthode **.catch()** est utilisée pour gérer les erreurs lorsqu'une promesse est **rejetée**. Elle prend une fonction callback qui reçoit la raison du rejet.

- Exécutée quand la promesse est **rejected**
- Reçoit la raison de **reject()**
- Gère les erreurs et exceptions
- Retourne une nouvelle promesse

```
promise.catch((error) => {
console.error('Erreur:', error);
// Gérer l'erreur
return 'Erreur gérée';
});
```

💡 **Chaînage de Promesses**

Les méthodes **.then()** et **.catch()** retournent des promesses, ce qui permet de les chaîner : **promise.then(...).then(...).catch(...)**. Cela crée une chaîne d'opérations asynchrones qui s'exécutent séquentiellement.

Async / Await : Introduction

QU'EST-CE QUE ASYNC / AWAIT ?

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**.

AVANTAGES DE ASYNC / AWAIT

- **Lisibilité Améliorée** - Le code ressemble à du code synchrone, plus facile à comprendre
- **Gestion d'Erreur Simplifiée** - Utilisation de try...catch comme du code synchrone
- **Moins de Callbacks** - Pas de chaînage de .then() imbriqués (Callback Hell)
- **Débogage Plus Facile** - Les stack traces sont plus claires

COMPARAISON RAPIDE

Avec Promesses (.then)

```
fetch('/api/data')
.then(res => res.json())
.then(data => {
console.log(data);
})
.catch(err => {
console.error(err);
});
```

Avec Async / Await

```
async function getData() {
try {
const res =
await fetch('/api/data');
const data =
await res.json();
console.log(data);
} catch (err) {
console.error(err);
}
}
```

Async / Await : Conversion

Avec Promesses (.then)

Code utilisant les méthodes **.then()** et **.catch()** pour gérer les promesses.

```
function fetchData() {
return fetch('/api/data')
.then(response => {
if (!response.ok) {
throw new Error('Erreur');
}
return response.json();
})
.then(data => {
console.log('Données:', data);
return data;
})
.catch(error => {
console.error('Erreur:', error);
});
}
```

Avec Async / Await

Code utilisant **async** et **await** pour une syntaxe plus lisible.

```
async function fetchData() {
try {
const response =
await fetch('/api/data');
if (!response.ok) {
throw new Error('Erreur');
}
const data =
await response.json();
console.log('Données:', data);
return data;
} catch (error) {
console.error('Erreur:', error);
}
}
```

✨ Avantages de Async / Await

- **Plus lisible** - Ressemble à du code synchrone traditionnel
- **Moins d'imbrication** - Pas de "callback hell" ou "pyramid of doom"
- **Gestion d'erreur simple** - Utilise try...catch comme du code synchrone
- **Débogage facile** - Les stack traces sont plus claires

Async / Await : Gestion d'Erreur

CONCEPT : TRY...CATCH

Dans un contexte asynchrone avec **async/await**, **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.

STRUCTURE

Bloc try

Contient le code asynchrone à exécuter. Si une erreur ou un rejet se produit, le contrôle passe au bloc catch.

Bloc catch

Exécuté si une erreur se produit dans le bloc try. Reçoit l'erreur en paramètre et permet de gérer l'erreur gracieusement.

EXEMPLE COMPLET

```
async function fetchData() {
try {
const response = await fetch('/api/data');
if (!response.ok) {
throw new Error('Erreur réseau');
}
const data = await response.json();
console.log('Données reçues:', data);
return data;
} catch (error) {
console.error('Erreur capturée:', error.message);
// Gérer l'erreur (afficher un message, etc.)
return null;
}
}
```

Patterns Avancés : Promise.all et Promise.race

Promise.all()

Exécute **plusieurs promesses en parallèle** et attend que **toutes soient résolues**. Retourne un tableau avec les résultats de toutes les promesses.

- Attend **toutes** les promesses
- Rejette si **une seule échoue**
- Retourne un **tableau de résultats**
- Cas d'usage : Charger plusieurs ressources

```
const p1 = fetch('/api/users');
const p2 = fetch('/api/posts');
const p3 = fetch('/api/comments');

Promise.all([p1, p2, p3])
.then(([users, posts, comments]) => {
console.log('Tous les données:',
users, posts, comments);
})
.catch(err => {
console.error('Une erreur:', err);
});
```

Promise.race()

Exécute **plusieurs promesses en parallèle** et retourne le résultat de la **première qui se termine**, qu'elle soit résolue ou rejetée.

- Attend la **première** promesse
- Retourne dès qu'une se termine
- Peut être résolue **ou** rejetée
- Cas d'usage : Timeout, course de requêtes

```
const fetchData = fetch('/api/data');
const timeout = new Promise((_, reject) =>
setTimeout(() =>
reject('Timeout!'), 5000)
);

Promise.race([fetchData, timeout])
.then(data => {
console.log('Première résultat:',
data);
})
.catch(err => {
console.error('Erreur:', err);
});
```

💡 Quand les Utiliser ?

Utilisez **Promise.all()** quand vous devez attendre que toutes les opérations réussissent (ex: charger plusieurs ressources). Utilisez **Promise.race()** quand vous ne vous intéressez qu'au premier résultat ou pour implémenter un timeout (ex: la requête la plus rapide gagne).

Patterns Avancés : Promise.allSettled

QU'EST-CE QUE PROMISE.ALLSETTLED ?

Promise.allSettled attend que **toutes les promesses soient soit résolues, soit rejetées**. Contrairement à **Promise.all**, elle **ne court-circuite pas** après une première promesse rejetée. Elle retourne toujours un tableau avec le statut et la valeur/raison de chaque promesse.

> **Différence avec Promise.all**
>
> **Promise.all** : Rejette immédiatement si une promesse échoue.
> **Promise.allSettled** : Attend toutes les promesses, même si certaines échouent.

CAS D'USAGE

- **Requêtes Multiples** - Récupérer des données de plusieurs API, même si certaines échouent
- **Opérations Batch** - Traiter un lot d'opérations et gérer les succès et les erreurs
- **Téléchargements Multiples** - Télécharger plusieurs fichiers et connaître le statut de chacun

EXEMPLE

```
const promises = [fetch('/api/users'), fetch('/api/posts'), fetch('/api/invalid')];
Promise.allSettled(promises).then(results => results.forEach(r => console.log(r.status === 'fulfilled' ? 'Succès:' + r.value : 'Erreur:' + r.reason)));
```

Conclusion

5 CONCEPTS CLÉS À RETENIR

1	2	3	4	5
Synchrone vs. Asynchrone	**Les Promesses**	**Async / Await**	**Gestion d'Erreur**	**Patterns Avancés**

RESSOURCES POUR APPROFONDIR

MDN Web Docs - Documentation complète sur les promesses et async/await

JavaScript.info - Tutoriels interactifs sur l'asynchronicité

Eloquent JavaScript - Livre en ligne gratuit sur JavaScript avancé

Event Loop Explainer - Visualisation de la boucle d'événements JavaScript

Continuez à pratiquer et à explorer l'asynchronicité en JavaScript. C'est une compétence essentielle pour développer des applications web modernes et performantes !