# 📚 Cours : Les Promesses et Asynchrone en JavaScript - Gérer le Temps et les Requêtes

## 🎯 Objectif du cours :

Ce cours vous guide à travers les concepts de **programmation asynchrone** en JavaScript, en utilisant les **promesses** et **les fonctions asynchrones**. Vous apprendrez comment gérer le temps et les requêtes réseau sans bloquer l'exécution du programme, un concept essentiel pour développer des applications web réactives et efficaces.

---

## 1. 🕒 Pourquoi l'asynchrone ?

Dans un environnement **asynchrone**, certaines tâches peuvent prendre du temps, comme la **lecture d'un fichier**, **l'accès à une base de données**, ou les **requêtes réseau**. Si ces opérations étaient exécutées de manière synchrone, elles bloqueraient l'exécution du programme jusqu'à ce qu'elles soient terminées, ce qui ralentirait considérablement les applications.

Avec JavaScript, nous utilisons des techniques **asynchrones** pour exécuter ces tâches en arrière-plan, permettant à notre application de continuer à fonctionner pendant que ces opérations longues sont traitées.

---

## 2. 🔄 Callbacks : Le début de l'asynchronisme

Avant les promesses, l'asynchronisme était géré avec des **callbacks**. Un **callback** est une fonction qui est passée en argument d'une autre fonction et qui est exécutée après la fin d'une tâche.

*Exemple de callback :*

```
function obtenirDonnees(callback) {
    setTimeout(() => {
        let donnees = { utilisateur: "Jean", age: 30 };
        callback(donnees); // L'exécution de la fonction callback une fois les
données récupérées
    }, 2000);
}

obtenirDonnees(function(donnees) {
    console.log(donnees); // Affiche les données après 2 secondes
});
```

Bien que les callbacks soient utiles, ils peuvent rendre le code difficile à lire, surtout lorsqu'ils sont imbriqués (d'où le problème du **callback hell**).

---

## 3. 💡 Les Promesses : Simplification de l'asynchrone

Les **promesses** sont une amélioration du modèle basé sur les callbacks, qui permettent de gérer l'asynchronisme de manière plus claire et plus élégante. Une promesse représente une valeur qui peut être **résolue** ou **rejetée** à un moment donné, et permet de chaîner plusieurs actions à réaliser lorsque cette valeur devient disponible.

### *a. Créer une promesse*

Voici un exemple de création d'une promesse en JavaScript :

```
let promesse = new Promise(function(resolve, reject) {
    let reussi = true;

    if (reussi) {
        resolve("Opération réussie !");
    } else {
        reject("Une erreur est survenue.");
    }
});

promesse
    .then(function(resultat) {
        console.log(resultat); // Affiche "Opération réussie !" si la promesse
est résolue
    })
    .catch(function(erreur) {
        console.log(erreur); // Affiche l'erreur si la promesse est rejetée
    });
```

- **`resolve()`** : Indique que l'opération a réussi et fournit la valeur.
- **`reject()`** : Indique que l'opération a échoué et fournit l'erreur.
- **`then()`** : Permet de spécifier ce qu'il faut faire quand la promesse est résolue avec succès.
- **`catch()`** : Permet de gérer les erreurs si la promesse est rejetée.

### *b. Chaîner les promesses*

Les promesses permettent de chaîner plusieurs actions qui dépendent des résultats des précédentes. Cela simplifie la gestion des opérations asynchrones multiples.

```
let promesse = new Promise(function(resolve, reject) {
    resolve(5);
});

promesse
    .then(function(resultat) {
        console.log(resultat); // Affiche 5
        return resultat * 2;  // Multiplie le résultat par 2
    })
```

```
    .then(function(resultat) {
        console.log(resultat); // Affiche 10
    })
    .catch(function(erreur) {
        console.log(erreur); // Gère les erreurs
    });
```

---

## 4. □ Fonctions Asynchrones avec `async` et `await`

Les mots-clés **`async`** et **`await`** ont été introduits pour rendre le code asynchrone encore plus lisible. Une fonction **`async`** retourne toujours une promesse, et vous pouvez utiliser **`await`** pour attendre qu'une promesse soit résolue avant de passer à l'étape suivante.

### *a. Définir une fonction `async`*

Une fonction **`async`** est une fonction qui, lorsqu'elle est exécutée, retourne une promesse, et permet d'utiliser **`await`** à l'intérieur pour attendre la résolution des promesses.

Exemple avec **`async`** et **`await`** :

```
async function obtenirDonnees() {
    let reponse = await fetch("https://api.exemple.com/donnees");
    let donnees = await reponse.json();
    console.log(donnees); // Affiche les données récupérées depuis l'API
}

obtenirDonnees();
```

### *b. Utilisation de `await`*

**`await`** est utilisé pour attendre la résolution d'une promesse dans une fonction **`async`**. Il permet de "bloquer" l'exécution du code jusqu'à ce que la promesse soit résolue ou rejetée.

```
async function getData() {
    let data = await fetch('https://api.exemple.com');
    let result = await data.json();
    console.log(result); // Affiche le résultat une fois que la promesse est
résolue
}
```

---

## 5. 🌐 Les Promesses avec des Requêtes Réseau

Les promesses sont particulièrement utiles lors de l'envoi de **requêtes réseau**, par exemple avec **`fetch()`**, qui retourne une promesse.

### *a. Utilisation de `fetch()` avec des Promesses*

`fetch()` est une méthode native de JavaScript utilisée pour effectuer des requêtes HTTP. Elle retourne une promesse que nous pouvons manipuler avec **`then()`**, **`catch()`**, ou **`async/await`**.

Exemple avec `fetch()` pour récupérer des données :

```
fetch("https://jsonplaceholder.typicode.com/posts")
    .then(response => response.json())
    .then(posts => console.log(posts)) // Affiche les posts récupérés
    .catch(error => console.log(error)); // Affiche une erreur en cas de
problème
```

---

## 6. 🔄 Gestion des Erreurs avec **`async/await`** et Promesses

Lors de l'utilisation des **promesses** ou des **fonctions asynchrones**, il est essentiel de gérer les erreurs pour éviter que des erreurs non gérées ne provoquent des dysfonctionnements dans l'application.

### *a. Gestion des erreurs avec `try/catch`*

Lorsque vous utilisez des fonctions **`async`** et **`await`**, vous pouvez facilement capturer les erreurs avec un bloc **`try/catch`**.

Exemple de gestion des erreurs dans une fonction **`async`** :

```
async function getData() {
    try {
        let response = await fetch('https://api.exemple.com');
        if (!response.ok) {
            throw new Error('La requête a échoué');
        }
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.log('Erreur capturée :', error);
    }
}
```

---

## 7. 📌 Conclusion

La programmation asynchrone en JavaScript avec des **promesses** et des fonctions **`async/await`** permet de gérer efficacement les tâches longues, comme les appels API, sans bloquer le reste de l'exécution du code. En maîtrisant ces concepts, vous pouvez rendre vos applications plus réactives et performantes.