Promise

En JavaScript, une Promise est un objet qui représente la complétion (ou l'échec) d'une opération asynchrone et permet de gérer le résultat de cette opération lorsqu'elle est terminée. Elle est particulièrement utile pour des opérations comme les appels de réseau, où le résultat n'est pas immédiatement disponible.

Les promesses permettent d'enchaîner des exécutions de code en JavaScript.

- On commence par instancier (créer un objet à partir d'un constructeur) une nouvelle promessse.
- Cet objet instancié prend une callback avec deux arguments.
- Cette callback est exécutée au moment de la création de l'objet.

Si l'opération est un succès (appel d'API, animation, etc...), on appelle la fonction resolve avec le résultat de ce succès, ce qui nous permet d'enchaîner des exécutions de fonction asynchrone à l'aide des méthodes then disponibles dans le prototype des promesses.

Si c'est un échec, on appelle reject qui nous permet d'exécuter une méthode catch, elle aussi disponible dans le prototype des promesses.

Structure d'une Promise

Une Promise peut avoir trois états :

1. Pending (en attente) : l'opération n'est pas encore terminée.

2. Fulfilled(réussie) : l'opération est terminée avec succès.

3. Rejected (rejetée) : l'opération a échoué.

Une Promise est créée en utilisant le constructeur Promise, qui prend une fonction comme argument. Cette fonction reçoit deux paramètres : **resolve et reject.**

- resolve : est appelé lorsque l'opération est réussie.

- reject : est appelé lorsqu'il y a une erreur.

Prenons un exemple où l'on veut simuler une opération asynchrone (comme une requête réseau) pour vérifier la disponibilité d'un utilisateur.

```
// Création de la Promise
function checkUserAvailability(username) {
  return new Promise((resolve, reject) => {
    // Simuler une opération asynchrone (ex: appel réseau)
    setTimeout(() => {
      if (username === "JohnDoe") {
        resolve("L'utilisateur est disponible");
      } else {
        reject("Utilisateur non trouvé");
      }
    }, 2000); // Temps d'attente simulé de 2 secondes
  });
}
```

Pour utiliser cette Promise, nous pouvons appeler la fonction checkUserAvailability et utiliser les méthodes .then et .catch pour gérer le succès ou l'échec.

```
checkUserAvailability("JohnDoe")
  .then((message) => {
    console.log(message); // Affichera : "L'utilisateur est disponible" après 2 secondes
  })
  .catch((error) => {
    console.error(error); // Affichera une erreur si l'utilisateur n'est pas trouvé
  });
```

1. Création : La fonction checkUserAvailability retourne une Promise. Dans cette Promise, une opération asynchrone est simulée avec setTimeout.

2. Résolution ou Rejet :

- Si le username est "JohnDoe" , la Promise est résolue (resolve("L'utilisateur est disponible")), ce qui passe la Promise à l'état "fulfilled".

- Sinon, elle est rejetée (reject("Utilisateur non trouvé")), ce qui passe la Promise à l'état "rejected".

3. Utilisation avec .then et .catch:

- .then est appelé lorsque la Promise est réussie (état "fulfilled") et reçoit le message de succès.

- .catch est utilisé pour gérer les erreurs (état "rejected") et reçoit le message d'erreur.

async et await

Pour une syntaxe plus simple, JavaScript fournit async et await :

```
async function findUser(username) {
  try {
    const message = await checkUserAvailability(username);
    console.log(message);
  } catch (error) {
    console.error(error);
  }
}

findUser("JohnDoe"); // Affichera "L'utilisateur est disponible"
```

Dans cet exemple, await permet d'attendre que la Promise soit résolue ou rejetée avant de continuer.

Voici un autre exemple pour illustrer l'utilisation des **Promises** en JavaScript. Dans cet exemple, nous simulons une opération pour récupérer des données depuis un serveur.

Supposons que nous voulons obtenir des informations sur un produit à partir d'une "base de données" (simulée ici).

Récupération de données de produit

Imaginons une fonction fetchProductDetails qui simule une demande réseau pour récupérer les détails d'un produit en fonction de son identifiant (productId).

```
function fetchProductDetails(productId) {
  return new Promise((resolve, reject) => {
    // Simulons un délai réseau avec setTimeout
    setTimeout(() => {
      const productDatabase = {
        1: { name: "Ordinateur portable", price: 1500 },
        2: { name: "Smartphone", price: 800 },
        3: { name: "Casque audio", price: 200 }
      };

      // Vérifie si le produit existe dans notre base de données simulée
      const product = productDatabase[productId];
      if (product) {
        resolve(product); // Résout la Promise avec les détails du produit
      } else {
        reject("Produit non trouvé"); // Rejette la Promise si l'ID est invalide
      }
    }, 1000); // Délai simulé d'une seconde
  });
}
```

Appelons la fonction fetchProductDetails et gérons les cas de succès ou d'erreur.

```
fetchProductDetails(2)
  .then((product) => {
    console.log("Détails du produit :", product);
    // Affichera : "Détails du produit : { name: 'Smartphone', price: 800 }"
  })
  .catch((error) => {
    console.error("Erreur :", error);
    // Affichera : "Erreur : Produit non trouvé" si l'ID n'existe pas
  });
```

1. Création de la Promise : La fonction **fetchProductDetails** retourne une Promise.

2. Simuler une "base de données" : À l'intérieur de cette Promise, nous avons un objet productDatabase qui stocke les informations de quelques produits.

3. Résolution ou Rejet :

- Si le productId existe dans notre base de données, la Promise est résolue avec les informations du produit (resolve(product)).
- Sinon, la Promise est rejetée avec un message d'erreur (reject("Produit non trouvé")).

Pour une syntaxe plus lisible, nous pouvons utiliser async et await :

```
async function getProductDetails(productId) {
  try {
    const product = await fetchProductDetails(productId);
    console.log("Détails du produit :", product);
  } catch (error) {
    console.error("Erreur :", error);
  }
}
```

getProductDetails(3); // Affichera les détails du produit "Casque audio"