Entendendo Promises em JavaScript

Por Escola Dnc

Introdução

As Promises são um conceito fundamental na programação assíncrona em JavaScript. Elas permitem lidar com operações que podem levar algum tempo para serem concluídas, sem bloquear a execução do restante do código. Neste ebook, vamos explorar em detalhes o que são Promises, como funcionam e como utilizá-las na prática.

O que são Promises?

Uma Promise em JavaScript representa uma operação assíncrona que pode ser concluída no futuro. Ela é um objeto que encapsula o resultado de uma operação que ainda não foi finalizada, mas que eventualmente produzirá um valor.

Analogia do mundo real

Para entender melhor o conceito de Promises, podemos fazer uma analogia com uma situação do cotidiano:

- Imagine que você está ocupado fazendo uma tarefa e precisa comprar algo urgentemente
- Você pede a um amigo para ir à loja fazer essa compra para você
- Enquanto seu amigo vai à loja, você continua trabalhando normalmente
- Quando seu amigo retorna, ele entrega o item comprado ou informa que não conseguiu realizar a compra

Nessa analogia:

- O pedido ao amigo representa a criação de uma Promise
- A ida do amigo à loja representa a operação assíncrona
- Você continuar trabalhando representa a execução do restante do código
- O retorno do amigo representa a resolução ou rejeição da Promise

Estados de uma Promise

Uma Promise sempre estará em um dos três estados possíveis:

- 1. Pending (Pendente): Estado inicial, a operação ainda não foi concluída
- 2. Fulfilled (Realizada): A operação foi concluída com sucesso
- 3. Rejected (Rejeitada): A operação falhou

Importante: Uma Promise só pode mudar de estado uma única vez, seja para fulfilled ou rejected. Após essa mudança, seu estado se torna imutável.

Sintaxe e Uso de Promises

Criando uma Promise

A sintaxe básica para criar uma Promise é a seguinte:

```
const minhaPromise = new Promise((resolve, reject) => { // Código
assíncrono aqui if (/* operação bem-sucedida */) {
resolve('Sucesso'); } else { reject('Falha'); }});
```

- resolve : função chamada quando a operação é bem-sucedida
- reject : função chamada quando a operação falha

Utilizando uma Promise

Para utilizar uma Promise, usamos os métodos .then() e .catch():

```
minhaPromise .then((resultado) => { console.log(resultado); //
'Sucesso' }) .catch((erro) => { console.error(erro); //
'Falha' });
```

- then(): executado quando a Promise é resolvida
- .catch(): executado quando a Promise é rejeitada

Exemplo Prático

Vamos criar um exemplo prático baseado na analogia do amigo fazendo uma compra:

```
function pedirCaderno() { return new Promise((resolve, reject) =>
     console.log("Amigo: Estou indo comprar o caderno.");
setTimeout(() => { const sucesso = true; // Simula se a compra
foi bem-sucedida if (sucesso) { resolve("Amigo:
                           if (sucesso) {
                                                   resolve("Amigo:
Aqui está o seu caderno!");
                                  } else {
                                                 reject("Amigo:
                                          }, 5000); // Simula 5
Desculpe, a loja estava fechada.");
segundos para ir à loja });}console.log("Eu: Preciso de um caderno
novo.");pedirCaderno() .then((mensagem) => {
console.log(mensagem); console.log("Eu: Obrigado!"); })
.catch((erro) => { console.log(erro); console.log("Eu: Tudo
bem, tentaremos outro dia."); });console.log("Eu: Vou continuar
estudando enquanto espero.");
```

Neste exemplo:

- 1. Criamos uma função pedirCaderno() que retorna uma Promise
- 2. A Promise simula a ida à loja com um setTimeout de 5 segundos
- 3. Usamos .then() para lidar com o sucesso e .catch() para lidar com falhas
- 4. O código continua executando enquanto a Promise está pendente

Vantagens e Usos Comuns de Promises

As Promises oferecem diversas vantagens na programação assíncrona:

- Código mais limpo e legível: Evita o "callback hell"
- Melhor tratamento de erros: Com o uso de .catch()
- Composição: Promises podem ser encadeadas
- Paralelismo: Executar várias operações assíncronas simultaneamente

Usos comuns incluem:

- Requisições HTTP
- Leitura/escrita de arquivos
- Operações em banco de dados
- Animações complexas

Conclusão

Promises são uma ferramenta poderosa para lidar com operações assíncronas em JavaScript. Elas permitem escrever código mais limpo e manutenível, além de oferecer um controle mais preciso sobre o fluxo de execução assíncrona.

Praticar o uso de Promises é essencial para dominar a programação assíncrona em JavaScript. Experimente criar suas próprias Promises, encadeá-las e lidar com diferentes cenários de sucesso e erro.

Lembre-se: a chave para entender Promises é pensar nelas como representações de operações futuras, que podem ser bem-sucedidas ou falhar, mas que permitem que seu código continue executando enquanto espera pelo resultado.