Promises

Igor N Faustino

O que é Assincronia?

- Em JavaScript, a assincronia refere-se à capacidade de executar operações sem bloquear a execução do código.
- Isso é particularmente útil em situações onde operações podem levar tempo, como requisições de rede, acesso a banco de dados, e operações de I/O.

Callbacks

- Um mecanismo inicial para lidar com assincronia em JavaScript é usando callbacks.
- Um callback é uma função passada como argumento para outra função que será chamada quando uma operação assíncrona for concluída.

Problemas com Callbacks

- Callbacks podem levar a um padrão de código conhecido como "Callback Hell" tornando o código difícil de ler e manter.
- O código pode se tornar profundamente aninhado, dificultando o entendimento da sequência de execução.

Promises

- Promises são uma abstração sobre callbacks, projetadas para simplificar o código assíncrono e lidar melhor com erros.
- Permitem encadear operações assíncronas de forma mais legível, usando os métodos **.then()** e **.catch()**

Promise - pendente

- O estado inicial de uma Promise é "Pendente".
- Isso significa que a operação assíncrona associada à Promise ainda não foi concluída.
- A Promise está esperando pelo resultado ou pela rejeição da operação.

Promise - Realizada (Fulfilled)

- Uma Promise entra no estado "Realizada" quando a operação assíncrona é concluída com sucesso.
- Isso significa que a Promise foi resolvida com um valor específico.
- O valor resolvido pode ser qualquer tipo de dado, como uma string, um objeto, ou até mesmo outra Promise.

Promise - Rejeitada (Rejected)

- Uma Promise entra no estado "Rejeitada" quando a operação assíncrona falha.
- Isso significa que a Promise foi rejeitada com um motivo específico, geralmente representado por um erro.
- O motivo da rejeição pode ser qualquer tipo de dado, como uma string de erro, um objeto de erro ou uma exceção.

Exercício

- Criar uma Promise que retorna um objeto com dados de uma pessoa

Exercício

- Criar uma Promise simples que resolve após um determinado tempo.

Exercício

- Crie uma função **simularRequisicao** que aceita um parâmetro booleano. Se o parâmetro for verdadeiro, a função deve retornar uma **Promise resolvida** após 1 segundo, contendo a string **"Requisição bem-sucedida".** Se for falso, a função deve retornar uma **Promise rejeitada** após 1 segundo, contendo a string **"Erro na requisição".** Implemente um código para chamar esta função com ambas as situações e tratar tanto a resolução quanto a rejeição.

Exercício

- Crie uma função **obterUsuario** que simule a obtenção de informações de um usuário. Esta função deve retornar uma Promise que resolve com um objeto contendo o nome do usuário após 1 segundo. Em seguida, crie uma função **obterComentarios** que simule a obtenção dos comentários feitos por esse usuário, retornando uma Promise que resolve com um array de comentários após 2 segundos. Por fim, crie uma terceira função **exibirDetalhesUsuario** que utilize as duas funções anteriores para obter e exibir os detalhes do usuário e seus comentários. Utilize o encadeamento de Promises para garantir que os dados sejam obtidos de forma sequencial.

Exercício

- Crie uma função atrasar que aceita um número como parâmetro e retorna uma Promise que será resolvida após um período de tempo aleatório entre 0 e o número passado como parâmetro. Utilize esta função para simular atrasos variados em uma sequência de operações assíncronas.

Exercício

- Crie uma função processarDados que aceita um array de números como parâmetro. Esta função deve realizar uma operação assíncrona que leva 2 segundos para completar. A operação deve consistir em calcular a média dos números passados no array. A função deve retornar uma Promise que resolve com o resultado da média.

Exercício

- Crie uma função buscarDoCache que simule a busca de dados em um cache. Esta função deve aceitar um parâmetro chave e retornar uma Promise que resolve com o valor associado à chave após 1 segundo, se estiver disponível no cache, ou rejeita com uma mensagem de erro se o valor não estiver disponível. Em seguida, crie uma função buscarDado que aceita a mesma chave como parâmetro e simula a busca de dados em uma fonte externa, retornando uma Promise que resolve com os dados após 2 segundos. Encadeie essas duas funções de modo que, se o valor não estiver disponível no cache, os dados sejam buscados na fonte externa e armazenados no cache para uso futuro. Certifique-se de lidar com todos os possíveis cenários de sucesso e erro.

Exercício

- Crie uma função autenticarUsuario que simule um processo de autenticação assíncrona. Esta função deve receber um nome de usuário e uma senha como parâmetros e retornar uma Promise que seja resolvida após 2 segundos se as credenciais forem válidas, e rejeitada com uma mensagem de erro caso contrário.

Encadeando Promises

- O encadeamento de Promises é uma técnica utilizada para executar operações assíncronas em sequência, onde o resultado de uma operação é utilizado como entrada para a próxima.

Encadeando Promises

```
minhaPromise
  .then((resultado1) => {
    // Executar operações com resultado1
    return resultado2;
  })
  .then((resultado2) => {
    // Executar operações com resultado2
    return resultado3;
  })
  .then((resultado3) => {
    // Executar operações com resultado3
  })
  .catch((erro) => {
    // Lidar com erros durante o encadeamento
  });
```

Como isso funciona?

- Cada then(): Cada chamada de then() retorna uma nova Promise, permitindo encadear múltiplas operações assíncronas.
- Passagem de Resultados: O resultado retornado pelo then() anterior é automaticamente passado como argumento para a próxima função de then().
- Retorno de Valores: É possível retornar valores ou Promises de dentro de uma função de then(), que serão então passados para a próxima etapa do encadeamento.
- Tratamento de Erros: O catch() no final do encadeamento é usado para lidar com quaisquer erros que ocorram durante o encadeamento.

Promise.all()

- O método Promise.all() é usado para executar múltiplas Promises em paralelo.
- Ele recebe um array de Promises como argumento e retorna uma nova Promise que é resolvida quando todas as Promises do array forem resolvidas, ou rejeitada se uma das Promises for rejeitada.
- Promise.all() é útil quando você precisa executar várias operações assíncronas independentes e quer esperar que todas sejam concluídas antes de prosseguir.

Exercício

- A primeira função deve retornar uma Promise que resolve com informações básicas de um usuário após 1 segundo. A segunda função deve retornar uma Promise que resolve com os posts feitos por esse usuário após 2 segundos. Em seguida, crie uma terceira função buscarDetalhesUsuario que utilize Promise.all() para buscar tanto as informações do usuário quanto os posts, e retorne uma Promise que resolva com um objeto contendo ambos os conjuntos de dados. Certifique-se de incluir tratamento de erro para lidar com qualquer rejeição de Promise.

Async/Await

- Async/Await é uma sintaxe baseada em Promises que torna o código assíncrono ainda mais legível e fácil de entender.
- Async declara que uma função é assíncrona e sempre retorna uma Promise. Await pausa a execução da função até que a Promise seja resolvida.

Exercício

- Reescreva o exercício de encadeamento utilizando **async/await** para simplificar a leitura e a escrita do código. Garanta que o comportamento e a funcionalidade permaneçam os mesmos.

Exercício

- Crie uma função processarLista que aceita um array de números como parâmetro. Esta função deve iterar sobre o array e, para cada número, chamar uma função assíncrona que retorna o dobro desse número após 1 segundo. Utilize async/await para garantir que cada iteração do loop aguarde a conclusão da operação assíncrona antes de prosseguir para o próximo número. Retorne uma Promise que resolva com um array contendo os resultados finais de cada operação.