Lista 6 de exercicios valendo (1,0)

O objeto Promise em JavaScript é usado para lidar com operações assíncronas e facilitar o gerenciamento de callbacks. Uma promise pode estar em um dos três estados: pendente (pending), resolvida (fulfilled) ou rejeitada (rejected).

Aqui estão alguns exemplos de como usar o objeto Promise em JavaScript:

Exercício 1: Criando uma promise simples.

```
const myPromise = new Promise((resolve, reject) => {
    const success = true;

    if (success) {
      resolve('Operação bem-sucedida');
    } else {
      reject('Operação falhou');
    }
  });

myPromise
    .then((result) => console.log(result)) // Operação bem-sucedida
    .catch((error) => console.error(error));
```

Exercício 2 : Simulando uma chamada de API.

```
function fetchData(url) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const success = true;

        if (success) {
          resolve({ data: 'Dados da API', url: url });
        } else {
          reject('Erro ao buscar dados');
        }
      }, 2000);
    });
  }

fetchData('https://api.example.com/data')
    .then((response) => console.log(response))
```

```
    .catch((error) => console.error(error));
```

Exercício 3 : Usando Promise.all para lidar com várias promises.

```
const promise1 = Promise.resolve('Promise 1 resolvida');
const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise 2 resolvida');
  }, 1000);
});
const promise3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise 3 resolvida');
  }, 2000);
});

Promise.all([promise1, promise2, promise3])
  .then((results) => console.log(results))
  .catch((error) => console.error(error));
```

Exercício 4 : Usando Promise.race para retornar a primeira promise resolvida ou rejeitada.

```
const promiseA = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Promise A resolvida');
    }, 1000);
  });
const promiseB = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Promise B resolvida');
    }, 500);
  });

Promise.race([promiseA, promiseB])
    .then((result) => console.log(result)) // Promise B resolvida
    .catch((error) => console.error(error));
```

Esses são apenas alguns exemplos do uso do objeto Promise em JavaScript. Promises são especialmente úteis quando você precisa lidar com várias operações assíncronas, como chamadas de API ou operações de leitura/gravação de arquivos.

Exercício 5 : Simulando uma autenticação de usuário:

Crie uma função `authenticateUser` que recebe um nome de usuário e senha e retorna uma promise. Se a autenticação for bem-sucedida, a promise deve ser resolvida com uma mensagem de sucesso. Caso contrário, deve ser rejeitada com uma mensagem de erro.

```
function authenticateUser(username, password) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const validCredentials = username === 'user' && password === '1234';

        if (validCredentials) {
          resolve('Autenticação bem-sucedida');
        } else {
          reject('Nome de usuário ou senha incorretos');
        }
      }, 1000);
    });
  }

authenticateUser('user', '1234')
    .then((message) => console.log(message))
    .catch((error) => console.error(error));
```

Exercício 6 : Simulando um sistema de carregamento de imagens: Crie uma função `loadImage` que recebe um URL e retorna uma promise. A promise deve ser resolvida com uma mensagem de sucesso se a imagem for carregada corretamente e rejeitada com uma mensagem de erro se houver um problema.

```
function loadImage(url) {
    return new Promise((resolve, reject) => {
      const img = new Image();

      img.onload = () => resolve('Imagem carregada com sucesso');
      img.onerror = () => reject('Erro ao carregar a imagem');
      img.src = url;
    });
  }

loadImage('https://example.com/image.jpg')
    .then((message) => console.log(message))
    .catch((error) => console.error(error));
```

Exercício 7 : Encadeando Promises: Crie duas funções, `step1` e `step2`, que retornam promises. A função `step1` deve ser resolvida com o valor 10 após 1 segundo. A função `step2` deve receber um número, multiplicá-lo por 2 e resolvê-lo após 1 segundo. Use o encadeamento de promises para executar essas funções em sequência e imprimir o resultado final.

```
function step1() {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(10);
      }, 1000);
    });
  }
function step2(number) {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(number * 2);
      }, 1000);
    });
  }

step1()
    .then((result1) => {
      console.log('Resultado 1:', result1);
```

```
    return step2(result1);
  })
  .then((result2) => {
    console.log('Resultado 2:', result2);
  });
```

Exercício 8 : Promises em sequência

Crie uma função que execute várias promises em sequência, ou seja, a próxima promise só será executada quando a promise anterior for resolvida. Por exemplo, você pode criar três funções que retornem promises e executá-las em sequência.

```
function task1() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Task 1 completed");
      resolve();
    }, 1000);
  });
}

function task2() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Task 2 completed");
      resolve();
    }, 1000);
  });
}

function task3() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Task 3 completed");
      resolve();
    }, 1000);
  });
}
```

```
  task1()
    .then(() => task2())
    .then(() => task3())
    .then(() => console.log("All tasks completed"));
```

Exercício 9: Tratamento de erros em promises

Crie uma função que retorne uma promise, simulando uma operação que pode falhar. Utilize o método catch para lidar com possíveis erros.

```
function fetchData() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const success = Math.random() > 0.5;

        if (success) {
          resolve("Data fetched successfully");
        } else {
          reject("Error fetching data");
        }
      }, 1000);
    });
  }

fetchData()
    .then((data) => console.log(data))
    .catch((error) => console.error(error));
```

Exercício 10: Promise.all com tratamento de erros

Crie várias promises e utilize o Promise.all para executá-las simultaneamente. Adicione tratamento de erros para que, se alguma promise falhar, você possa identificar qual falhou e continuar executando as outras.

```
const promises = [
    Promise.resolve("Promise 1"),
    new Promise((resolve, reject) => setTimeout(() => reject("Promise 2 failed"),
1000)),
    Promise.resolve("Promise 3"),
  ];

Promise.allSettled(promises)
    .then((results) => {
      results.forEach((result, index) => {
        if (result.status === "fulfilled") {
          console.log(`Promise ${index + 1} succeeded: ${result.value}`);
        } else {
          console.error(`Promise ${index + 1} failed: ${result.reason}`);
        }
      });
    });
```