CONSUMO DE API COM REACT JS

Para começar, vamos entender o JavaScript por trás da conexão que criamos com as API's.

PROMISES (Promessas)

O que são Promises?

Promises são objetos em JavaScript que representam o resultado eventual de uma operação assíncrona. Elas são usadas para lidar com computações assíncronas de forma mais limpa e fácil, evitando o chamado "callback hell" (aninhamento excessivo de callbacks) e permitindo um melhor controle do fluxo de código.

Características das Promises:

1. **Estado**: Uma Promise pode estar em um de três estados:
   - **Pendente**: Estado inicial, quando a operação assíncrona está sendo executada e ainda não foi concluída.
   - **Resolvida**: A operação assíncrona foi concluída com sucesso.
   - **Rejeitada**: A operação assíncrona falhou ou foi rejeitada.
2. **Encadeamento**: Promises podem ser encadeadas usando os métodos `.then()` e `.catch()`. Isso permite que você defina ações a serem executadas após a resolução ou rejeição da Promise.

Como Criar uma Promise:

```
const minhaPromise = new Promise((resolve, reject) => {
  // Simula uma operação assíncrona
  setTimeout(() => {
    const resultado = Math.random();
    if (resultado > 0.5) {
      resolve(resultado); // Se tivermos sucesso, resolvemo
s a Promise com o resultado
    } else {
      reject(new Error('Erro ao processar')); // Se houver
um erro, rejeitamos a Promise com um erro
    }
  }, 1000);
});
```

Neste exemplo, criamos uma nova Promise que simula uma operação assíncrona (uma espera de 1 segundo) e resolve ou rejeita com base em um resultado aleatório.

Como Usar uma Promise:

```
minhaPromise
  .then(resultado => {
    console.log('Sucesso:', resultado);
  })
  .catch(error => {
    console.error('Erro:', error);
  });
```

Neste exemplo, usamos o método `.then()` para lidar com o caso em que a Promise é resolvida com sucesso, e o método `.catch()` para lidar com o caso em que a Promise é rejeitada.

Benefícios das Promises:

- **Legibilidade**: As Promises permitem um código mais limpo e legível, especialmente quando se trata de operações assíncronas encadeadas.
- **Gestão de Erros**: Elas fornecem um mecanismo claro para lidar com erros em operações assíncronas.

- **Encadeamento Simples**: O encadeamento de Promises facilita a execução de várias operações assíncronas em sequência.

Funções Assíncronas (`async` `functions`)

O que são Funções Assíncronas?

Funções assíncronas são funções em JavaScript que permitem a execução de código de forma assíncrona, ou seja, sem bloquear a execução do restante do código. Isso é particularmente útil quando você precisa lidar com operações que levam tempo, como fazer solicitações de rede, ler/escrever arquivos, esperar por temporizadores, entre outros.

Como Funcionam as Funções Assíncronas?

As funções assíncronas são declaradas usando a palavra-chave `async`. Quando você chama uma função assíncrona, ela retorna uma Promise. Dentro dessa função, você pode usar a palavra-chave `await` para esperar que outra Promise seja resolvida. Isso permite que o código aguarde a conclusão de operações assíncronas sem bloquear a execução.

Exemplo de Função Assíncrona:

```
async function exemploAssincrono() {
  // Operação assíncrona, por exemplo, aguardando uma Promi
se
  let resultado = await algumaFuncaoAssincrona();

  // O código aqui só será executado depois que a Promise f
or resolvida
  console.log(resultado);
}

exemploAssincrono();
```

Neste exemplo:

- `exemploAssincrono` é uma função assíncrona que aguarda a resolução de `algumaFuncaoAssincrona` usando a palavra-chave `await`.

- A execução da função `exemploAssincrono` é pausada na linha com `await` até que a Promise retornada por `algumaFuncaoAssincrona` seja resolvida.
- Quando a Promise é resolvida, o valor retornado é armazenado na variável `resultado`, e o restante do código dentro da função é executado.

Benefícios das Funções Assíncronas:

- **Clareza de Código**: As funções assíncronas permitem escrever código de forma mais clara e concisa, especialmente ao lidar com operações assíncronas.
- **Tratamento de Erros Simplificado**: Você pode usar `try/catch` para lidar com erros de forma semelhante ao código síncrono, facilitando o tratamento de erros em operações assíncronas.

Fetch (Busca) JavaScript

O que é o `fetch`?

O `fetch` é uma função nativa do JavaScript que permite fazer solicitações HTTP assíncronas. Ele fornece uma maneira fácil e poderosa de buscar recursos de uma URL. O `fetch` retorna uma Promise que resolve a resposta da solicitação.

Como usar o `fetch`?

Aqui está um exemplo básico de como usar o `fetch` para fazer uma solicitação GET para uma API:

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Erro ao buscar os dados');
    }
    return response.json();
  })
  .then(data => {
    console.log(data); // Dados recebidos da API
  })
```

```
  .catch(error => {
    console.error('Erro:', error);
  });
```

Neste exemplo:

- Usamos o `fetch` para fazer uma solicitação GET para `https://api.example.com/data`.
- O método `then` é usado para lidar com a resposta da solicitação. Verificamos se a resposta foi bem-sucedida (status 200-299). Se não for, lançamos um erro.
- Se a resposta for bem-sucedida, usamos o método `json()` para converter os dados da resposta em um objeto JavaScript.
- Por fim, usamos outro método `then` para acessar os dados recebidos e lidar com eles.

Funções Assíncronas para Efetuar Fetches

Em muitos casos, é mais conveniente usar funções assíncronas, especialmente ao trabalhar com `fetch`, pois isso torna o código mais limpo e legível.

Veja como você pode usar uma função assíncrona para realizar um fetch:

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/d
ata');
    if (!response.ok) {
      throw new Error('Erro ao buscar os dados');
    }
    const data = await response.json();
    console.log(data); // Dados recebidos da API
  } catch (error) {
    console.error('Erro:', error);
  }
}

fetchData();
```

Neste exemplo, usamos a palavra-chave `async` para declarar a função `fetchData` como uma função assíncrona. Dentro da função, usamos a palavra-chave `await` para aguardar a conclusão da solicitação `fetch` e a conversão dos dados da resposta em JSON.

Enfim... Vamos ao que interessa...

Consumindo uma API no ReactJS ⚛️

Vamos criar um novo componente React que será responsável por consumir a API. Você pode criar um componente chamado `Posts.js`, por exemplo:

```
// Posts.js

import React, { useState, useEffect } from 'react';

const Posts = () => {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response => response.json())
      .then(data => setPosts(data))
      .catch(error => console.log(error));
  }, []);

  return (
    <div>
      <h1>Posts</h1>
      <ul>
        {posts.map(post => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
};
```

```
export default Posts;
```

Neste componente, estamos usando o `useState` para armazenar os posts obtidos da API e o `useEffect` para realizar a solicitação à API assim que o componente é montado. Estamos usando a função `fetch` para fazer a solicitação GET para a API e atualizando o estado dos posts com os dados recebidos.

Passo 3: Renderizando o Componente na Aplicação

Agora, vamos importar e renderizar o componente `Posts` no componente principal `App`:

```
// App.js

import React from 'react';
import Posts from './Posts';

function App() {
  return (
    <div className="App">
      <Posts />
    </div>
  );
}

export default App;
```

Passo 4: Testando a Aplicação

Agora, inicie o servidor de desenvolvimento executando `npm start` no terminal dentro do diretório do seu projeto React. Isso iniciará o servidor e abrirá automaticamente a aplicação no seu navegador padrão.

Você deverá ver uma lista de posts sendo exibidos na página renderizada pelo componente `Posts`, consumidos da API.

AXIOS (Biblioteca para requisições HTTP)

O Axios é uma biblioteca JavaScript popular usada para fazer requisições HTTP tanto no navegador quanto no Node.js. Ele fornece uma interface simples e intuitiva para lidar com requisições assíncronas de forma mais eficiente e fácil do que as APIs nativas do navegador, como o `fetch`.

Principais Características do Axios:

1. **Sintaxe Simples**: O Axios oferece uma sintaxe simples e intuitiva para fazer requisições HTTP, o que torna o código mais legível e fácil de entender.
2. **Suporte a Promises**: O Axios é baseado em Promises, o que significa que você pode usar o encadeamento de `.then()` e `.catch()` para lidar com respostas e erros de maneira elegante e eficiente.
3. **Compatibilidade**: O Axios é compatível com todos os navegadores modernos e também com o Node.js, o que o torna uma escolha versátil para fazer requisições HTTP em qualquer ambiente JavaScript.
4. **Suporte a Cancelamento de Requisições**: O Axios permite cancelar requisições pendentes, o que é útil em casos onde você precisa interromper uma requisição em andamento, por exemplo, quando um componente é desmontado em uma aplicação React.
5. **Suporte a Interceptors**: Os interceptors do Axios permitem que você intercepte e transforme requisições e respostas antes que elas sejam tratadas pelo código do aplicativo. Isso é útil para adicionar headers comuns, autenticação, manipulação de erros e muito mais.

Exemplo de Uso do Axios:

Veja um exemplo simples de como usar o Axios para fazer uma requisição GET para uma API:

```
import axios from 'axios';

// Fazendo uma requisição GET para uma API
axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data); // Dados recebidos da API
```

```
  })
  .catch(error => {
    console.error('Erro:', error);
  });
```

Neste exemplo, usamos o método `.get()` do Axios para fazer uma requisição GET para a URL especificada. Em seguida, usamos o método `.then()` para lidar com a resposta da requisição e o método `.catch()` para lidar com qualquer erro que possa ocorrer durante a requisição.