Aula 25 – Consumindo APIs com Fetch

Bem-vindo à Aula 25! Hoje, vamos mergulhar em um dos pilares do desenvolvimento web moderno: a comunicação entre sua aplicação e o mundo exterior.

A Ponte Digital

Conectando sua aplicação ao mundo

Imagine que seu site ou aplicativo é uma cidade vibrante, mas para que ela funcione de verdade, precisa se conectar com outras cidades, trocar informações, receber suprimentos e enviar dados. Essa conexão é feita através das **APIs**, e a ferramenta que usaremos para essa ponte é a poderosa **Fetch API**.

Nesta aula, você descobrirá como as aplicações web dinâmicas que usamos todos os dias — desde redes sociais até e-commerces — conseguem exibir informações atualizadas em tempo real.

Entender a comunicação com APIs não é apenas uma habilidade técnica; é a chave para construir experiências de usuário ricas e interativas. Ao final, você não só compreenderá o que são APIs REST e como a Fetch API funciona, mas também será capaz de integrar esses conhecimentos em uma aplicação React, tratando respostas e erros de forma robusta.

Prepare-se para desvendar os segredos por trás da troca de dados na web. Vamos explorar desde os conceitos fundamentais de uma API REST até a implementação prática de requisições HTTP, garantindo que sua aplicação possa conversar fluentemente com qualquer serviço online. Esta jornada não só ampliará seu repertório técnico, mas também o capacitará a criar soluções mais dinâmicas e conectadas, um diferencial crucial no mercado atual.

Desvendando as APIs REST: A Linguagem da Web Moderna

O Cardápio Digital

No universo da internet, onde bilhões de dispositivos e serviços se comunicam incessantemente, existe uma linguagem universal que permite essa interação fluida: as **APIs**, ou Interfaces de Programação de Aplicações.

Como Funciona

Pense nelas como um "cardápio" de um restaurante. Você não precisa saber como a comida é preparada na cozinha (o código interno do servidor), mas o cardápio (a API) lista o que você pode pedir (os dados ou funcionalidades) e como fazer o pedido (os métodos de requisição).

APIs REST

Entre os diversos tipos de APIs, as **APIs REST (Representational State Transfer)** se destacam como o padrão ouro para a web.

Elas são um conjunto de princípios arquitetônicos que guiam a forma como os sistemas distribuídos se comunicam. A ideia central é que os recursos (como usuários, produtos, posts) são representados por URLs únicas, e as operações sobre esses recursos são realizadas usando os métodos padrão do protocolo HTTP.

Analogia: É como ter um endereço específico para cada prato no cardápio e usar verbos como "pedir", "alterar" ou "cancelar" para interagir com eles.

Essa abordagem simplifica a comunicação, tornando-a escalável e flexível. Ao seguir os princípios REST, desenvolvedores podem criar serviços que são fáceis de entender, consumir e manter, garantindo que diferentes aplicações — seja um site, um aplicativo móvel ou outro serviço de backend — possam interagir de forma padronizada e eficiente.

O Contrato da Comunicação: Entendendo os Endpoints e Dados

01

Endpoints: Os Endereços

Se as APIs REST são o cardápio, então os **endpoints** são os itens específicos desse cardápio, cada um com seu próprio "endereço" e "instruções". Um endpoint é uma URL que representa um recurso específico no servidor.

02

Exemplo Prático

Por exemplo, em uma API de e-commerce, `/produtos` pode ser o endpoint para listar todos os produtos, e `/produtos/123` para acessar um produto específico com ID 123. É o ponto de contato onde sua aplicação envia requisições e de onde recebe as respostas.

03

Formato JSON

Quando sua aplicação faz uma requisição a um endpoint, ela geralmente espera receber dados de volta. O formato mais comum para essa troca de dados na web moderna é o **JSON (JavaScript Object Notation)**.

Por que JSON?

JSON é um formato leve e legível por humanos para troca de dados, que se assemelha muito à forma como objetos são estruturados em JavaScript. Ele é ideal porque é fácil para as máquinas analisarem e para os desenvolvedores entenderem.

> **Analogia da Pizza:** Imagine que você está pedindo uma pizza. O endpoint seria o tipo de pizza (Ex: /pizzas/calabresa). A requisição seria seu pedido, e a resposta do restaurante viria em uma caixa (o JSON), contendo todas as informações sobre a pizza: ingredientes, preço, tempo de preparo.

Essa estrutura padronizada garante que tanto quem envia quanto quem recebe os dados consiga interpretá-los corretamente, estabelecendo um contrato claro para a comunicação.

```
// Exemplo de dados JSON recebidos de um endpoint /usuarios
[
 {
 "id": 1,
 "nome": "João Silva",
 "email": "joao.silva@example.com"
 },
 {
 "id": 2,
 "nome": "Maria Souza",
 "email": "maria.souza@example.com"
 }
]
```

Fetch API: Sua Ferramenta para Buscar Dados

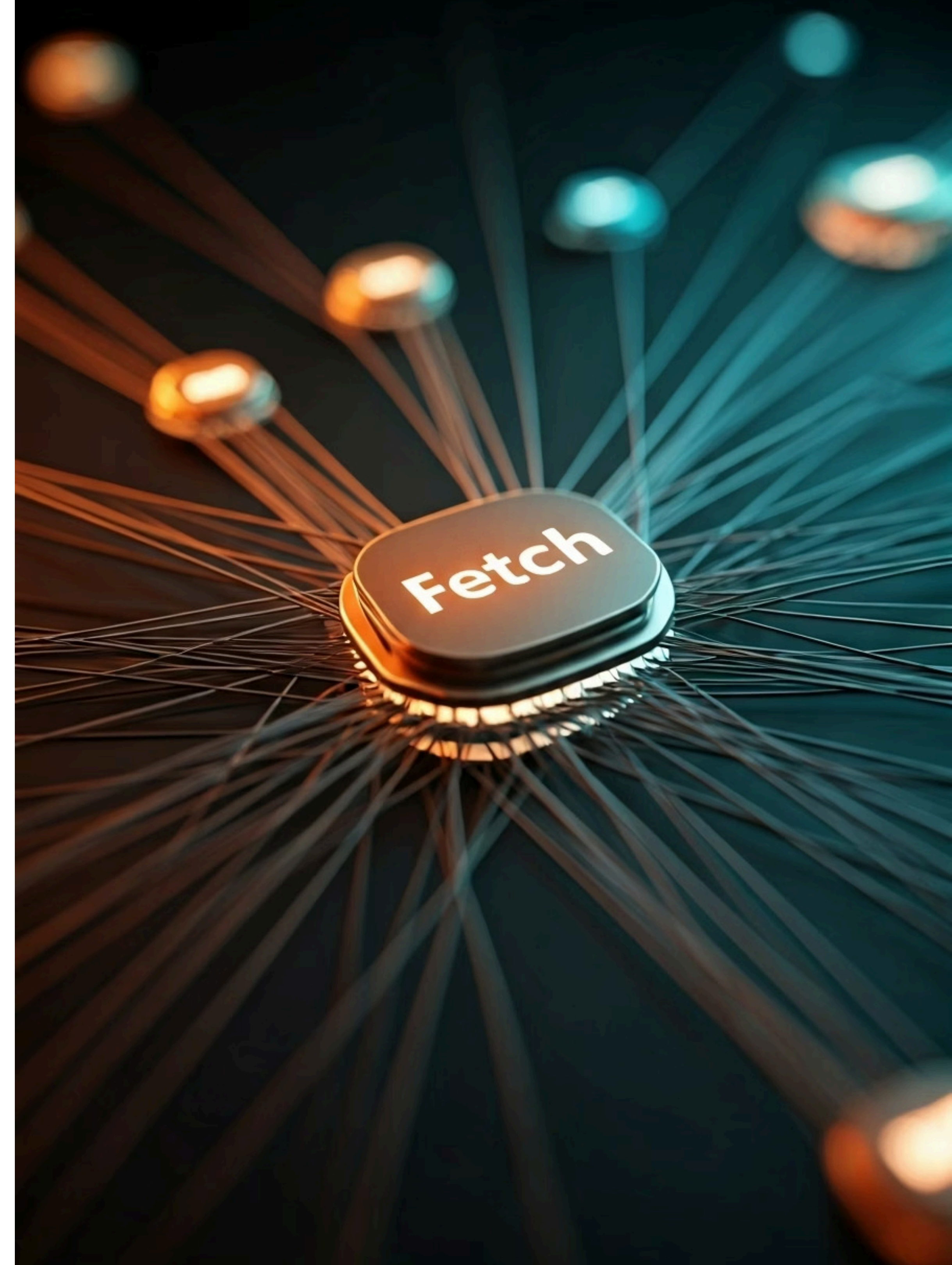
Agora que entendemos o que são APIs REST e como elas funcionam, é hora de conhecer a ferramenta que nos permitirá interagir com elas diretamente no navegador: a **Fetch API**.

Antes da Fetch

Era comum usar bibliotecas como jQuery com seu método `$.ajax()` ou o objeto `XMLHttpRequest` nativo, que era mais verboso e complexo.

A Revolução

A Fetch API veio para simplificar e modernizar a forma como fazemos requisições HTTP.



Promises: O Coração da Fetch

Interface Poderosa

A Fetch API oferece uma interface poderosa e flexível para buscar recursos na rede. Ela é baseada em **Promises**, o que significa que as operações de rede são assíncronas e não bloqueiam a execução do seu código.

Quando você faz uma requisição com `fetch()`, ela retorna uma Promise que, eventualmente, será resolvida com um objeto Response (em caso de sucesso) ou rejeitada com um erro (em caso de falha na rede).

Analogia do Serviço de Entrega: Pense na Fetch API como um serviço de entrega de encomendas. Você faz um pedido (a requisição fetch()), e o serviço promete entregar a encomenda. Enquanto a encomenda não chega, você pode continuar fazendo outras coisas. Quando ela chega, o serviço te avisa. Se houver um problema na entrega, ele também te informa.

Essa natureza assíncrona é fundamental para construir aplicações web responsivas, que não "congelam" enquanto esperam por dados do servidor.

A sintaxe básica é surpreendentemente simples, começando com `fetch(url)`. Essa única linha já é capaz de iniciar uma requisição GET para a URL especificada, abrindo as portas para a interação com qualquer API disponível na web.

Fazendo Sua Primeira Requisição GET com Fetch

Com a Fetch API em mãos, vamos dar o primeiro passo e realizar uma requisição GET simples. O método GET é usado para solicitar dados de um recurso especificado. É como pedir para ver o cardápio ou perguntar o preço de um item. Ele não altera nada no servidor, apenas busca informações.

Para fazer uma requisição GET, você só precisa passar a URL do endpoint para a função `fetch()`. O resultado é uma Promise, que podemos encadear com o método `.then()` para lidar com a resposta. A primeira `.then()` geralmente recebe o objeto Response e é responsável por verificar se a requisição foi bem-sucedida e, em seguida, converter o corpo da resposta para o formato desejado, que na maioria das vezes é JSON. O método `response.json()` também retorna uma Promise, por isso precisamos de um segundo `.then()` para acessar os dados processados.

Veja um exemplo prático:

```
fetch('https://jsonplaceholder.typicode.com/posts/1') // URL de um post de exemplo
  .then(response => {
    // Verifica se a resposta foi bem-sucedida (status 200-299)
    if (!response.ok) {
      throw new Error('Erro na requisição: ' + response.status);
    }
    return response.json(); // Converte a resposta para JSON
  })
  .then(data => {
    console.log('Dados recebidos:', data);
    // Aqui você pode atualizar a interface do usuário com os dados
  })
  .catch(error => {
    console.error('Houve um problema com a operação fetch:', error);
  });
```

Neste código, estamos buscando um post específico de uma API de testes. O primeiro `.then()` verifica a integridade da resposta e a transforma em JSON. O segundo `.then()` finalmente nos dá acesso aos dados que podemos usar em nossa aplicação, como exibir o título e o corpo do post em uma página web.

Lidando com Respostas: Status HTTP e Dados

Códigos de Status

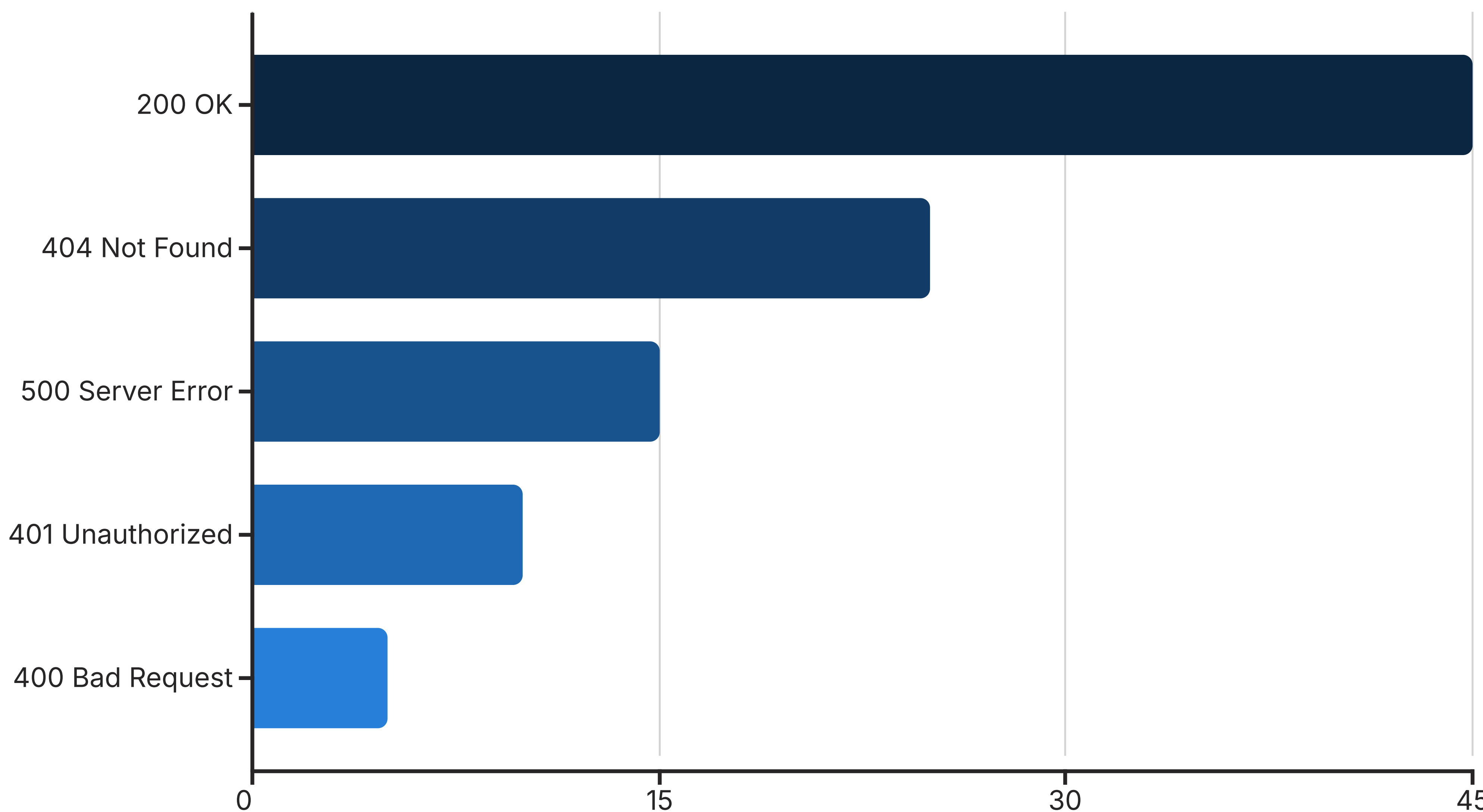
Após fazer uma requisição, o servidor nos envia uma resposta, e essa resposta contém mais do que apenas os dados que pedimos. Ela também inclui informações importantes sobre o status da requisição, comunicadas através dos **Códigos de Status HTTP**.

Esses códigos são como sinais de trânsito que nos dizem se a requisição foi bem-sucedida, se houve um erro do cliente (como uma URL inválida) ou um erro do servidor.

O objeto Response que recebemos na primeira Promise do `fetch()` possui propriedades úteis, como `status` (o código numérico, ex: 200, 404) e `ok` (um booleano que é true para códigos de status na faixa 200-299, indicando sucesso). É crucial verificar essas propriedades para garantir que a requisição foi processada como esperado antes de tentar usar os dados. Ignorar essa verificação pode levar a erros inesperados na sua aplicação.

> **Analogia do Atendimento:** Imagine que você está ligando para um serviço de atendimento. O código de status é a mensagem automática que você recebe: "Sua solicitação foi processada com sucesso" (200 OK), "O número discado não existe" (404 Not Found), ou "Houve um problema técnico em nossos sistemas" (500 Internal Server Error).

Entender esses códigos permite que sua aplicação reaja de forma inteligente a diferentes cenários, oferecendo feedback adequado ao usuário.



Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
200 OK	Requisição bem-sucedida	Sucesso	Dados de um usuário foram carregados.
201 Created	Recurso criado com sucesso	Sucesso	Um novo post foi adicionado ao blog.
400 Bad Request	Requisição malformada pelo cliente	Erro do Cliente	Parâmetros inválidos na URL.
401 Unauthorized	Autenticação necessária ou falhou	Erro do Cliente	Tentativa de acesso sem login.
403 Forbidden	Acesso negado, mesmo com autenticação	Erro do Cliente	Usuário sem permissão para deletar recurso.
404 Not Found	Recurso não encontrado	Erro do Cliente	URL do endpoint incorreta.
500 Internal Server Error	Erro inesperado no servidor	Erro do Servidor	Falha interna ao processar a requisição.

Tratando Erros de Rede e Requisição com catch()

Nem toda requisição HTTP termina em sucesso. Erros podem acontecer por diversas razões: falha na conexão de internet do usuário, o servidor da API estar offline, a URL estar incorreta, ou até mesmo problemas de permissão. É fundamental que sua aplicação esteja preparada para lidar com esses cenários de forma elegante, sem quebrar ou apresentar mensagens confusas ao usuário.

Erros de Rede

Problemas como falta de conexão, DNS inválido ou o servidor não respondendo. Nestes casos, a `fetch()` Promise é rejeitada diretamente.

Erros Lançados

Se a `response.ok` for false (indicando um status HTTP de erro como 404 ou 500), podemos lançar um Error dentro do `.then()`, que será capturado pelo `.catch()` subsequente.

A Fetch API, por ser baseada em Promises, oferece um mecanismo robusto para tratamento de erros: o método `.catch()`. Este bloco é executado quando a Promise é rejeitada.

> **Analogia da Carta:** Imagine que você está tentando enviar uma carta. Se o correio (rede) estiver fora do ar, a carta nem sai. Isso é um erro de rede. Se a carta chega, mas o endereço está errado (status 404), o carteiro te avisa. Em ambos os casos, você precisa de um plano B. O `.catch()` é esse plano B, permitindo que você exiba uma mensagem de erro amigável, registre o problema ou tente novamente.

```
fetch('https://api.exemplo.com/dados-inexistentes') // URL que provavelmente causará um erro 404
 .then(response => {
 if (!response.ok) {
 // Lança um erro se a resposta não for bem-sucedida (ex: 404, 500)
 throw new Error(`Erro HTTP! Status: ${response.status}`);
 }
 return response.json();
 })
 .then(data => {
 console.log('Dados recebidos:', data);
 })
 .catch(error => {
 // Captura erros de rede ou erros lançados no .then()
 console.error('Ocorreu um erro ao buscar os dados:', error.message);
 // Aqui você pode mostrar uma mensagem de erro ao usuário na UI
 document.getElementById('mensagem-erro').textContent =
 'Não foi possível carregar os dados. Tente novamente mais tarde.';
 });
```

Requisições POST: Enviando Dados para o Servidor

Preencher Dados

Usuário insere informações no formulário

Empacotar JSON

Converter dados para formato JSON

Enviar POST

Requisição com método POST

Processar no Servidor

Servidor cria o novo recurso

Até agora, focamos em buscar dados com o método GET. Mas e se quisermos enviar informações para o servidor, como criar um novo usuário, adicionar um produto ao carrinho ou enviar um formulário? Para isso, usamos o método **POST**. O POST é um dos métodos HTTP mais comuns para enviar dados ao corpo da requisição, geralmente para criar um novo recurso no servidor.

Configurações Necessárias

- `method: 'POST'`: Indica que é uma requisição POST.
- `headers`: Define os cabeçalhos da requisição. O mais comum é `Content-Type: 'application/json'`.
- `body`: O corpo da requisição, que contém os dados. Use `JSON.stringify()`.

Analogia do Formulário: Imagine que você está preenchendo um formulário de inscrição online. Você digita seu nome, e-mail e senha. Quando clica em "Enviar", esses dados são empacotados (convertidos para JSON), rotulados como "formulário de inscrição" (Content-Type) e enviados para o servidor através de uma requisição POST.

```
const novoUsuario = {
 nome: 'Ana Paula',
 email: 'ana.paula@example.com',
 senha: 'senhaSegura123'
};

fetch('https://jsonplaceholder.typicode.com/users', { // Endpoint para criar usuários
 method: 'POST', // Define o método como POST
 headers: {
 'Content-Type': 'application/json', // Informa que o corpo é JSON
 },
 body: JSON.stringify(novoUsuario), // Converte o objeto JavaScript para string JSON
})
 .then(response => {
 if (!response.ok) {
 throw new Error('Erro ao criar usuário: ' + response.status);
 }
 return response.json(); // O servidor geralmente retorna o recurso criado
 })
 .then(data => {
 console.log('Usuário criado com sucesso:', data);
 // Aqui você pode redirecionar o usuário ou exibir uma mensagem de sucesso
 })
 .catch(error => {
 console.error('Houve um problema ao enviar os dados:', error);
 });
```

Outros Métodos HTTP: PUT e DELETE em Ação

GET

Buscar e visualizar dados existentes

POST

Criar novos recursos no servidor

PUT

Atualizar recursos existentes completamente

DELETE

Remover recursos do servidor

Além de GET e POST, existem outros métodos HTTP essenciais para interagir com APIs REST, completando as operações básicas de CRUD (Create, Read, Update, Delete). O método **PUT** é usado para **atualizar** um recurso existente, enquanto o **DELETE** é, como o nome sugere, para **remover** um recurso.

PUT - Atualização

O método PUT é similar ao POST em sua estrutura, pois também envia dados no corpo da requisição. A diferença fundamental é que PUT é **idempotente**, o que significa que fazer a mesma requisição PUT várias vezes terá o mesmo efeito que fazê-la uma única vez (o recurso será atualizado para o mesmo estado).

DELETE - Remoção

Já o método DELETE é mais simples. Ele não requer um corpo de requisição, pois a informação de qual recurso deletar já está contida na URL (o ID do recurso).

> **Analogia do Perfil:** Imagine que você tem um perfil online. GET é para ver seu perfil. POST é para criar um novo post. PUT é para editar seu nome de usuário ou foto de perfil. DELETE é para apagar um post antigo. Cada ação tem seu método HTTP correspondente, garantindo clareza e padronização na comunicação com a API.

```
// Exemplo de requisição PUT (atualizar um post)
const postAtualizado = {
 id: 1, // O ID do recurso a ser atualizado
 title: 'Meu Novo Título de Post',
 body: 'Este é o conteúdo atualizado do meu post.',
 userId: 1
};

fetch('https://jsonplaceholder.typicode.com/posts/1', { // Endpoint para atualizar o post com ID 1
 method: 'PUT',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(postAtualizado),
})
 .then(response => response.json())
 .then(data => console.log('Post atualizado:', data));

// Exemplo de requisição DELETE (remover um post)
fetch('https://jsonplaceholder.typicode.com/posts/1', { // Endpoint para deletar o post com ID 1
 method: 'DELETE',
})
 .then(response => {
 if (response.ok) {
 console.log('Post deletado com sucesso!');
 } else {
 throw new Error('Falha ao deletar post.');
 }
 })
 .catch(error => console.error(error));
```

Configurações Avançadas do Fetch: Headers e Opções

A Fetch API é bastante flexível e permite que você personalize suas requisições de diversas maneiras, indo além dos métodos HTTP básicos e do corpo da requisição. O segundo argumento da função `fetch()` é um objeto de opções que pode conter uma série de propriedades para controlar o comportamento da requisição.

Headers (Cabeçalhos)

São metadados que acompanham a requisição e fornecem informações adicionais ao servidor, como o tipo de conteúdo que está sendo enviado (`Content-Type`), o tipo de conteúdo que o cliente aceita (`Accept`), ou tokens de autenticação (`Authorization`).

Mode

Controla o CORS (Cross-Origin Resource Sharing), definindo como o navegador deve lidar com requisições entre diferentes origens.

Cache

Define como o navegador deve lidar com o cache da requisição, podendo forçar uma nova busca ou usar dados em cache.

Credentials

Controla o envio de cookies e credenciais de autenticação junto com a requisição.

Signal

Permite abortar requisições em andamento usando um `AbortController`.

Analogia da Embalagem: Pense nos cabeçalhos como a "embalagem" da sua encomenda. Além do conteúdo (o body), você pode adicionar etiquetas especiais: "Frágil" (cache), "Urgente" (mode), "Confidencial" (Authorization). Essas etiquetas dão instruções adicionais para o serviço de entrega e para o destinatário, garantindo que a encomenda seja tratada da maneira correta.

```
fetch('https://api.exemplo.com/recursos-protegidos', {
 method: 'GET',
 headers: {
 'Content-Type': 'application/json',
 'Authorization': 'Bearer seu_token_jwt_aqui', // Exemplo de token de autenticação
 'Accept': 'application/json' // Indica que o cliente aceita JSON como resposta
 },
 // Outras opções, como cache: 'no-cache' para evitar o cache do navegador
 cache: 'no-cache'
})
 .then(response => {
 if (!response.ok) {
 throw new Error('Falha na autenticação ou acesso negado.');
 }
 return response.json();
 })
 .then(data => console.log('Dados protegidos:', data))
 .catch(error => console.error('Erro ao acessar recurso protegido:', error));
```

Integrando Fetch em Aplicações React: Onde a Mágica Acontece

Agora que dominamos a Fetch API, é hora de conectá-la ao mundo das aplicações frontend modernas, especificamente com React. Em um componente React, a busca de dados de uma API é uma operação com "efeitos colaterais" – ela interage com o mundo exterior e não é uma computação pura.

O Hook useEffect

Por que useEffect?

Para gerenciar esses efeitos colaterais de forma eficiente e segura, o React nos oferece o hook **useEffect**. O useEffect é o local ideal para realizar requisições de dados.

Ele permite que você execute código após a renderização do componente, e o mais importante, permite que você especifique quando esse código deve ser reexecutado (por exemplo, apenas uma vez após a montagem do componente, ou quando certas dependências mudam).

Além disso, precisaremos do hook `useState` para armazenar os dados recebidos da API, o estado de carregamento e quaisquer erros que possam ocorrer.

> **Analogia do Gerente de Palco:** Pense no useEffect como um "gerente de palco" para seu componente React. Ele garante que a busca de dados aconteça no momento certo (quando o componente entra em cena), que os dados sejam atualizados quando necessário, e que a "cena" seja limpa quando o componente sai (por exemplo, cancelando requisições pendentes para evitar vazamentos de memória).

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

function ListaDePosts() {
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    // Função assíncrona para buscar os dados
    const fetchPosts = async () => {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts');
        if (!response.ok) {
          throw new Error(`Erro HTTP! Status: ${response.status}`);
        }
        const data = await response.json();
        setPosts(data);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false); // Garante que o estado de carregamento seja desativado
      }
    };

    fetchPosts(); // Chama a função de busca
  }, []); // O array vazio [] garante que o useEffect execute apenas uma vez (na montagem)

  if (loading) return
```

Carregando posts...

; if (error) return

Erro ao carregar posts: {error.message}

; return (

Posts do Blog

{posts.map(post => ( ))}

- **{post.title}** {post.body}

); } export default ListaDePosts;

Construindo um Componente React para Consumir Dados

01

Estado de Dados

Armazena os dados quando a requisição for bem-sucedida

02

Estado de Loading

Booleano para indicar se a requisição está em andamento

03

Estado de Erro

Armazena qualquer erro que ocorra durante a requisição

Vamos aprofundar no exemplo anterior e construir um componente React mais completo que não só busca dados, mas também gerencia os estados de carregamento e erro, proporcionando uma experiência de usuário mais robusta. A chave aqui é o uso combinado de `useState` para gerenciar o estado da UI (dados, carregando, erro) e `useEffect` para orquestrar a requisição assíncrona.

Ao buscar dados, é uma boa prática ter pelo menos três estados. Esses estados permitem que o componente reaja dinamicamente ao ciclo de vida da requisição, desde o início do carregamento até a exibição dos dados ou de uma mensagem de erro. Isso é crucial para a **acessibilidade (A11Y)**, pois usuários de leitores de tela precisam ser informados sobre o status da aplicação.

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

function ProdutoDetalhes({ productId }) {
  const [produto, setProduto] = useState(null);
  const [carregando, setCarregando] = useState(true);
  const [erro, setErro] = useState(null);

  useEffect(() => {
    const buscarProduto = async () => {
      try {
        setCarregando(true); // Inicia o carregamento
        setErro(null); // Limpa erros anteriores
        const response = await fetch(`https://fakestoreapi.com/products/${productId}`);
        if (!response.ok) {
          throw new Error(`Falha ao buscar produto: ${response.status}`);
        }
        const dados = await response.json();
        setProduto(dados);
      } catch (e) {
        setErro(e); // Armazena o erro
      } finally {
        setCarregando(false); // Finaliza o carregamento, independente do sucesso ou falha
      }
    };

    if (productId) { // Garante que a requisição só ocorra se houver um productId
      buscarProduto();
    }
  }, [productId]); // A requisição será refeita se o productId mudar

  if (carregando) {
    return
```

Carregando detalhes do produto...

; // A11Y: informa o status } if (erro) { return

Erro: {erro.message}

; // A11Y: alerta sobre o erro } if (!produto) { return

Nenhum produto encontrado.

; } return (

{produto.title}

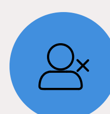
Preço: R$ {produto.price}

{produto.description}

); } export default ProdutoDetalhes;

Boas Práticas e Otimização: Pensando no Futuro

Construir aplicações que consomem APIs vai além de apenas fazer requisições. Para garantir que sua aplicação seja robusta, performática e ofereça uma excelente experiência ao usuário, é crucial adotar algumas boas práticas e técnicas de otimização. Isso inclui desde o tratamento de erros mais sofisticado até a gestão de requisições em cenários complexos.

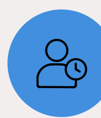


AbortController

Cancela requisições pendentes para evitar dados desatualizados e melhorar a performance

Core Web Vitals

Otimize requisições para melhorar métricas como LCP e FID, impactando SEO e UX



Debounce

Evite requisições excessivas em campos de busca usando técnicas de debounce

---

AbortController em Ação

Uma técnica importante é o uso do **AbortController** para cancelar requisições pendentes. Imagine que um usuário digita algo em um campo de busca e, antes que a primeira requisição termine, ele digita novamente.

Sem o AbortController, a primeira requisição pode retornar depois da segunda, exibindo dados desatualizados. Com ele, você pode cancelar a requisição anterior antes de iniciar uma nova, garantindo que apenas a requisição mais recente seja processada.

Outro ponto vital é a **performance web**, que está diretamente ligada às **Core Web Vitals**. Requisições de API lentas podem impactar métricas como LCP (Largest Contentful Paint) e FID (First Input Delay). Otimizar a busca de dados (por exemplo, com paginação, cache no cliente ou pré-carregamento) é fundamental para uma boa pontuação e, consequentemente, para o SEO e a experiência do usuário.

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

function BuscaDinamica() {
 const [termoBusca, setTermoBusca] = useState('');
 const [resultados, setResultados] = useState([]);
 const [carregando, setCarregando] = useState(false);

 useEffect(() => {
  const controller = new AbortController(); // Cria um novo AbortController
  const signal = controller.signal; // Obtém o sinal para a requisição

  const buscarTermo = async () => {
   if (!termoBusca) {
    setResultados([]);
    return;
   }
   setCarregando(true);
   try {
    const response = await fetch(`https://api.exemplo.com/busca?q=${termoBusca}`, { signal });
    if (!response.ok) throw new Error('Falha na busca.');
    const data = await response.json();
    setResultados(data);
   } catch (error) {
    if (error.name === 'AbortError') {
     console.log('Requisição abortada:', termoBusca);
    } else {
     console.error('Erro na busca:', error);
    }
   } finally {
    setCarregando(false);
   }
  };

  const debounceTimeout = setTimeout(buscarTermo, 500); // Debounce para evitar muitas requisições

  return () => {
   clearTimeout(debounceTimeout);
   controller.abort(); // Aborta a requisição se o componente for desmontado ou o termo de busca mudar
  };
 }, [termoBusca]);

 return (
```

setTermoBusca(e.target.value)} /> {carregando &&

Buscando...

}

{resultados.map((item) => ( ))}

- {item.nome}

); } export default BuscaDinamica;

Acessibilidade (A11Y) e Fetch: Garantindo Inclusão

A acessibilidade (A11Y) é um pilar fundamental no desenvolvimento web moderno, e isso se estende diretamente à forma como interagimos com APIs. Quando sua aplicação busca dados, o estado da interface do usuário muda: elementos podem aparecer, desaparecer ou ter seu conteúdo alterado. Para usuários com deficiência, especialmente aqueles que dependem de leitores de tela, é crucial que essas mudanças sejam comunicadas de forma clara e eficaz.

1

Estados de Carregamento

Use `aria-live="polite"` em mensagens de "Carregando..." para anunciar mudanças sem interromper o usuário

2

Mensagens de Erro

Utilize `aria-live="assertive"` para erros críticos que precisam de atenção imediata

3

Gerenciamento de Foco

Mova o foco do teclado para novo conteúdo após requisições bem-sucedidas

---

Comunicação Clara

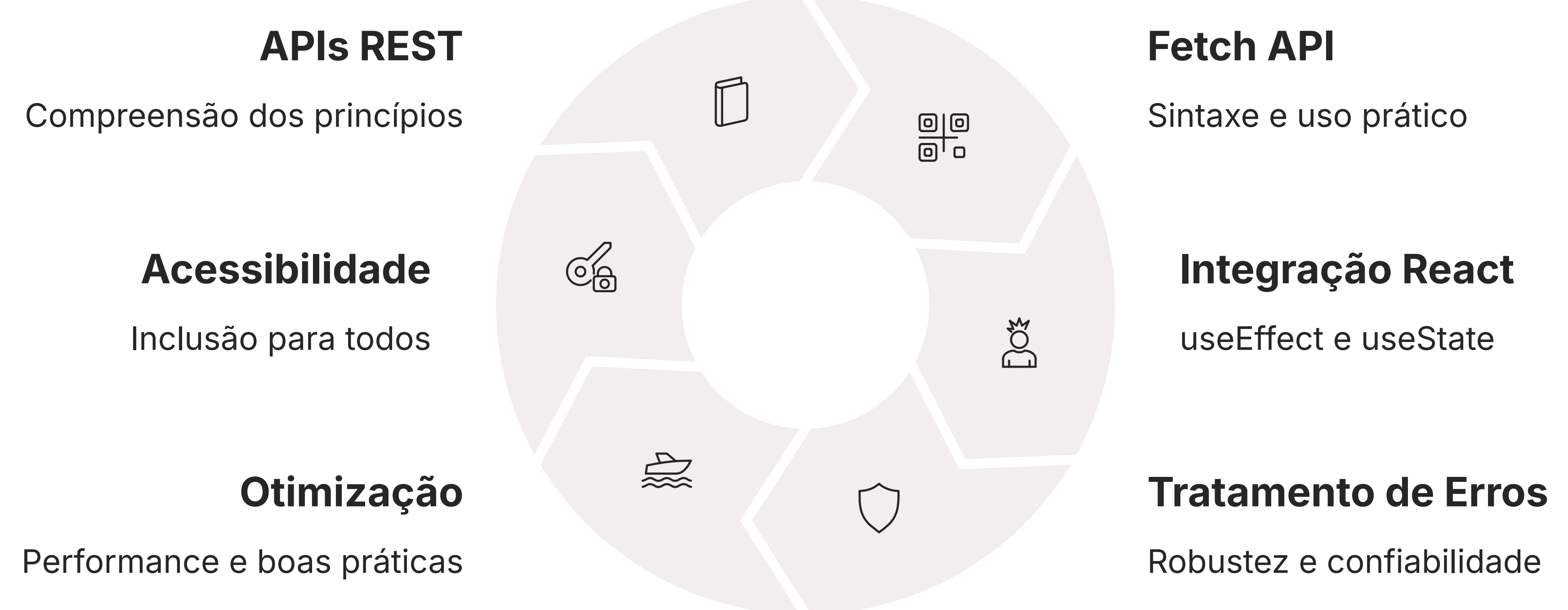
Integrar a acessibilidade ao processo de consumo de APIs significa pensar em como o usuário será informado sobre o que está acontecendo. Por exemplo, ao exibir um estado de carregamento, use atributos ARIA como `aria-live="polite"` em um elemento que contém a mensagem "Carregando...".

Isso instrui o leitor de tela a anunciar a mudança de conteúdo sem interromper imediatamente o que o usuário está fazendo. Da mesma forma, mensagens de erro devem ser visíveis e anunciadas, utilizando `aria-live="assertive"` para erros críticos.

> **Princípio Fundamental:** A acessibilidade não é um "extra", mas uma parte integrante do design e desenvolvimento, garantindo que todos possam usar sua aplicação, independentemente de suas habilidades.

Além disso, considere o gerenciamento de foco. Após uma requisição bem-sucedida que adiciona novos elementos à página, pode ser útil mover o foco do teclado para o novo conteúdo ou para uma área relevante, garantindo que usuários de teclado possam navegar eficientemente.

Conclusão e Próximos Passos



Chegamos ao fim da nossa jornada sobre o consumo de APIs com Fetch. Percorremos desde a compreensão do que são APIs REST e seus princípios, passando pela sintaxe básica da Fetch API, até a integração em aplicações React, com tratamento de erros e boas práticas de otimização e acessibilidade. Você agora entende que a Fetch API é uma ferramenta poderosa e essencial para construir aplicações web dinâmicas e interativas, capazes de se comunicar com o vasto ecossistema de serviços online.

> **Em prática:**
>
> Lembre-se de sempre verificar a propriedade `response.ok` e o status da resposta para tratar erros de forma robusta. Utilize o `.catch()` para capturar falhas de rede e erros lançados. Ao integrar em React, o `useEffect` e `useState` são seus melhores amigos para gerenciar o ciclo de vida da requisição e o estado da UI. E, acima de tudo, priorize a acessibilidade, garantindo que todos os usuários possam compreender o que está acontecendo em sua aplicação.