Aula 05 — Consumindo APIs com React

Requisições HTTP e Fetch API

Prof. Me. José Olinda

Objetivo desta aula

Vamos criar uma aplicação de **loja online** que consome dados de uma API externa.

O que você aprenderá na prática:

1. Usar `fetch()` para fazer requisições HTTP
2. Trabalhar com JSON
3. Renderizar dados dinâmicos de uma API
4. Alternar entre diferentes endpoints
5. Tratar erros e estados de carregamento

A FakeStore API

Vamos usar a **FakeStore API** — uma API pública e gratuita que fornece dados simulados de uma loja.

Documentação: https://fakestoreapi.com/docs

Endpoints principais:

- `GET /products` → lista todos os produtos
- `GET /products/{id}` → detalhe de um produto
- `GET /users` → lista todos os usuários

A API retorna dados em **JSON**, que é nativo do JavaScript.

Seu primeiro projeto: FakeStore App

Vamos criar uma aplicação que:

- Exibe uma lista de **produtos** ou **usuários** da FakeStore API
- Permite alternar entre "Produtos" e "Usuários" com botões
- Mostra mensagem de carregamento enquanto busca dados
- Trata erros de requisição

Iniciando o projeto

01 - Crie o projeto Vite

```
npm create vite@latest fakestore-app --template react
cd fakestore-app
npm install
npm run dev
```

Estrutura inicial

Seu projeto deve ficar assim:

```
fakestore-app/
├── src/
│   ├── App.jsx
│   ├── main.jsx
│   ├── App.css
│   └── index.html
├── vite.config.js
└── package.json
```

App.css — Estilos básicos

Limpe o `App.css` e adicione:

```css
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  font-family: Arial, sans-serif;
  background-color: #1e1e2e;
  color: #f8f8f2;
}
```

App.css — Listagem (continuação)

Adicione ao final do `App.css`:

```
.loading {
  text-align: center;
  font-size: 18px;
  color: #8be9fd;
  margin: 40px 0;
}

.error {
  background-color: #ff5555;
  color: white;
  padding: 15px;
  border-radius: 5px;
```

App.jsx — Estrutura inicial

Limpe o `App.jsx` e comece com:

```jsx
import { useState } from "react";
import "./App.css";

function App() {
  const [items, setItems] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");
  const [dataType, setDataType] = useState("products");

  return (
    <div className="container">
      <h1>🛒 FakeStore</h1>
```

App.jsx — Função loadProducts()

Adicione esta função **dentro do componente**, antes do `return`:

```
  async function loadProducts() {
    setDataType("products");
    setLoading(true);
    setError("");
    setItems([]);

    try {
      const response = await fetch("https://fakestoreapi.com/products");

      if (!response.ok) {
        throw new Error("Erro ao buscar produtos");
      }
```

App.jsx — Função loadUsers()

Adicione esta função também **dentro do componente**, antes do `return`:

```
  async function loadUsers() {
    setDataType("users");
    setLoading(true);
    setError("");
    setItems([]);

    try {
      const response = await fetch("https://fakestoreapi.com/users");

      if (!response.ok) {
        throw new Error("Erro ao buscar usuários");
```

Seu App.jsx completo (parte 1 - estado)

```
import { useState } from "react";
import "./App.css";

function App() {
  const [items, setItems] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");
  const [dataType, setDataType] = useState("products");
```

Seu App.jsx completo (parte 2 - funções)

```
  async function loadProducts() {
    setDataType("products");
    setLoading(true);
    setError("");
    setItems([]);

    try {
      const response = await fetch("https://fakestoreapi.com/products");
      if (!response.ok) throw new Error("Erro ao buscar produtos");
      const data = await response.json();
      setItems(data);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
```

Seu App.jsx completo (parte 3 - JSX)

```
  return (
    <div className="container">
      <h1>🛒 FakeStore</h1>

      <div className="buttons">
        <button
          className={dataType === "products" ? "active" : ""}
          onClick={() => loadProducts()}
        >
          📦 Produtos
        </button>
        <button
          className={dataType === "users" ? "active" : ""}
          onClick={() => loadUsers()}
        >
```

Testando a aplicação

01 - Certifique-se que o servidor está rodando

```
npm run dev
```

Seu navegador deve abrir em `http://localhost:5173`

Testando a aplicação

02 - Clique nos botões

1. Clique em "📦 **Produtos**"
    - Veja a mensagem "⏳ Carregando..."
    - Aguarde alguns segundos
    - A lista de produtos deve aparecer
2. Clique em "👥 **Usuários**"
    - A lista muda para usuários
    - Veja detalhes de cada usuário

Testando a aplicação

03 - Experimente

- Passe o mouse sobre os itens (hover)
- Observe a mudança de cor no botão ativo
- Teste a responsividade (redimensione o navegador)
- Abra o Console (F12) para ver as respostas do servidor

Conceitos importantes (use agora)

`fetch()` — Requisição HTTP

```
const response = await fetch(url);
```

- `fetch()` retorna uma **Promise**
- `await` aguarda a resposta
- `response.ok` verifica se deu sucesso (status 200-299)

Conceitos importantes (use agora)

`.json()` — Parse JSON

```
const data = await response.json();
```

- Converte a resposta bruta em objeto JavaScript
- A resposta da FakeStore API é um array ou objeto JSON

Conceitos importantes (use agora)

`try/catch/finally` — Tratamento de erros

```
try {
  // código que pode falhar
} catch (err) {
  // o que fazer se falhar
} finally {
  // sempre executado (mesmo se erro)
}
```

- Uma boa prática para requisições de rede
- Evita travamento da aplicação

Conceitos importantes (use agora)

`async/await` — Código assíncrono

```
async function loadData() {
  // permite usar 'await'
}
```

- Torna o código mais legível que callbacks ou Promise.then()
- Aguarda resultado sem bloquear a UI

Conceitos importantes (use agora)

Renderização condicional com dados da API

```
{dataType === "products" ? (
  <>
    <h3>{item.title}</h3>
    <p>${item.price}</p>
  </>
) : (
  <>
    <h3>{item.username}</h3>
    <p>{item.email}</p>
  </>
)}
```

Exercícios

Exercício 1: Adicione um botão "📂 Categorias" que busca apenas as categorias de produtos. (Dica: endpoint é `/products/categories`)

Exercício 2: Modifique a exibição de produtos para mostrar também a **estrela** (rating). (Dica: `item.rating.rate`)

Exercício 3: Adicione um botão "Limpar" que reseta os dados e esconde a lista.

Exercício 4: Crie um campo de `<input>` que permite buscar um produto específico pelo ID. (Dica: endpoint `/products/id`)

Desafio Extra

Altere a aplicação para que os produtos sejam carregados **automaticamente** quando a página abre (sem o usuário clicar no botão).

Dica: Você precisará do hook `useEffect`, que veremos na próxima aula. Por enquanto, experimente chamar `loadProducts()` diretamente no código, mas verifique o console para entender o que acontece.

Resumo da aula

✅ Aprendemos a usar `fetch()` para requisições HTTP
✅ Trabalhamos com JSON em React
✅ Renderizamos dados dinâmicos de uma API
✅ Alternamos entre diferentes endpoints
✅ Tratamos erros e estados de carregamento

Na próxima aula: veremos `useEffect` para automatizar requisições e explicaremos os conceitos `async/await`, `Promise` e tratamento de erros em profundidade.

Referências

- **FakeStore API:** https://fakestoreapi.com/docs
- **Fetch API (MDN):** https://developer.mozilla.org/pt-BR/docs/Web/API/fetch
- **async/await (MDN):** https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/async_function