Aula 01 — Introdução ao React

Criando um projeto React com Vite

Comandos iniciais (terminal):

```
npm create vite@latest meu-app --template react
cd meu-app
npm install
```

`package.json`

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview"
}
```

- `npm run dev` → ambiente de desenvolvimento (HMR)
- `npm run build` → gera versão de produção (pasta `dist/`)
- `npm run preview` → simula produção localmente

Estrutura de pastas inicial

Estrutura típica do Vite:

- `index.html`
- `src/`
  - `main.jsx`
  - `App.jsx`
  - `assets/`
  - `index.css`, `App.css`
- `vite.config.js`
- `package.json`

`index.html` e o elemento `#root`

Exemplo simplificado:

```html
<!doctype html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8" />
    <title>Meu SPA React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

`#root` é o ponto onde o React é montado.

`main.jsx` e o ReactDOM

Exemplo de `main.jsx`:

```jsx
import React from "react";
import { createRoot } from "react-dom/client";

createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

- `createRoot` cria a raiz React e conecta a árvore de componentes ao DOM. Algumas observações importantes:
- `React.StrictMode` é uma ferramenta de desenvolvimento que ativa verificações e avisos adicionais (não aparece em produção).
- O arquivo `main.jsx` é o ponto de entrada da aplicação: normalmente registra estilos globais, provê provedores (ex.: roteamento, context providers) e monta o componente raiz.
- Em aplicações que usam Server Components ou hidratação, o fluxo de montagem pode envolver APIs de hidratação (`hydrateRoot`) — ver documentação específica quando usar SSR.
- `<App />` é o componente raiz da SPA.

Criando o primeiro componente

Exemplo `App.jsx`:

```jsx
function App() {
  return (
    <div>
      <h1>Hello React</h1>
      <p>Minha primeira SPA com Vite.</p>
    </div>
  );
}

export default App;
```

Um componente em React é uma função que retorna JSX.

O que é JSX?

JSX (JavaScript XML) permite escrever HTML dentro do JavaScript. Durante o build, JSX é convertido para chamadas `React.createElement(...)`.

Exemplo:

```
<h1>Hello React</h1>
```

Conceitualmente convertido para:

```
React.createElement("h1", null, "Hello React");
```

JSX:

- JSX não é obrigatório, mas melhora legibilidade. Ferramentas modernas (Vite, bundlers) suportam JSX nativamente.
- Tags devem ser fechadas (`<br />`, `<img />`) e componentes precisam começar com letra maiúscula para o React diferenciá-los de elementos HTML.
- Use fragments (`<>...</>`) para retornar múltiplos elementos sem adicionar nós extras ao DOM.

Regras importantes de JSX

- Cada componente deve retornar apenas um elemento raiz.
- Atributos usam `camelCase`: `class` → `className`, `for` → `htmlFor`.
- Expressões JavaScript ficam entre `{ }`.

Exemplo:

```
const title = "React";
function App() {
  return <h1>Hello {title}</h1>;
}
```

Interpolando dados em JSX

Você pode interpolar qualquer expressão JS válida entre `{ }`:

```
const user = { name: "Ana", age: 22 };
function App() {
  const greeting = `Olá, ${user.name}`;
  return (
    <div>
      <h1>{greeting}</h1>
      <p>Idade: {user.age}</p>
      <p>Dobro da idade: {user.age * 2}</p>
    </div>
  );
}
```

Interpolação e atributos:

- Você pode usar expressões em atributos: `src={user.imageUrl}` ou `className={isActive ? 'active' : ''}`.
- Para estilos inline use um objeto JS: `style={{ width: user.imageSize, height: user.imageSize }}` (as chaves externas delimitam a expressão JS, as internas formam o objeto).
- Evite executar operações pesadas diretamente no JSX — calcule valores fora do retorno para manter o JSX limpo.

Componentes de função e arrow functions

Declaração tradicional:

```
function Header() {
  return <h1>Minha SPA</h1>;
}
```

Arrow function:

```
const Header = () => <h1>Minha SPA</h1>;
```

Composição de componentes

Exemplo:

```
const Header = () => <h1>Minha SPA</h1>;
const Footer = () => <small>Desenvolvido em React 19</small>;

const App = () => (
  <div>
    <Header />
    <main>
      <p>Conteúdo principal...</p>
    </main>
    <Footer />
  </div>
);
export default App;
```

Hierarquia de componentes

- A aplicação forma uma árvore de componentes (App → Header, SearchBar, ArticleList, ArticleItem, etc.).
- É importante manter componentes pequenos e focados em uma única responsabilidade (Single Responsibility Principle).
- Use pastas para organizar componentes por funcionalidade ou tipo (ex.: `components/`, `pages/`, `layouts/`).
- A composição facilita reutilização, testes e manutenção do código.

Renderizando listas com `map`

```
const articles = [
  { id: 1, title: "Introdução ao React", author: "Dan" },
  { id: 2, title: "SPA com React Router", author: "Ryan" },
];

const ArticleList = () => (
  <ul>
    {articles.map(article => (
      <li key={article.id}>
        <strong>{article.title}</strong> - {article.author}
      </li>
    ))}
  </ul>
);
```

Use a prop `key` com um identificador estável e único. Boas práticas:

Eventos em React

Sintaxe de eventos em JSX:

```
function App() {
  function handleClick() {
    alert("Você clicou no botão");
  }

  return <button onClick={handleClick}>Clique aqui</button>;
}
```

Eventos usam camelCase: `onClick`, `onChange`, `onSubmit`.

Mais detalhes sobre eventos:

- Os handlers recebem um objeto de evento sintético (SyntheticEvent) que normaliza diferenças entre navegadores.
- Não chame o handler ao atribuí-lo (`onClick={handleClick}` e não `onClick={handleClick()}`).
- Se precisar do evento ou de argumentos, use uma função de flecha: `onClick={(e) => handleClick(e, id)}` .

Introdução ao estado com `useState`

Exemplo de contador:

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  function handleIncrement() { setCount(count + 1); }
  return (
    <div>
      <p>Valor atual: {count}</p>
      <button onClick={handleIncrement}>Incrementar</button>
    </div>
  );
}
```

Re-renderização

Quando um setter de estado é chamado, o componente é executado novamente; React compara o Virtual DOM anterior com o novo e atualiza somente o necessário no DOM real.

Boas práticas:

- Não modifique o DOM diretamente
- Evite mutações diretas em objetos/arrays de estado (use cópias imutáveis)

Fluxo de dados com `props`

```
function Greeting({ name }) { return <h2>Olá, {name}</h2>; }

function App() {
  return (
    <div>
      <Greeting name="Ana" />
      <Greeting name="Bruno" />
    </div>
  );
}
```

PROPS

`props` são somente leitura; dados fluem de cima para baixo (top-down). Algumas boas práticas:

- Trate `props` como imutáveis dentro do componente — para alterar dados, suba o estado para o componente pai (lifting state up) e passe callbacks.
- Use nomes de props autoexplicativos e, quando útil, documente seus tipos; em projetos JS puro considere `prop-types` ou TypeScript para checagem estática.

Componentes controlados

Formulário controlando o valor de um input:

```
import { useState } from "react";

function SearchBar() {
  const [term, setTerm] = useState("");
  function handleChange(event) { setTerm(event.target.value); }
  return (
    <div>
      <label htmlFor="search">Buscar:</label>
      <input id="search" value={term} onChange={handleChange} />
      <p>Você digitou: {term}</p>
    </div>
  );
}
```

Navegação em uma SPA

Use bibliotecas de roteamento (ex.: React Router) para controlar quais componentes são exibidos em cada URL.

Fluxo:

- Acessar `/artigos` → exibe `<ArticlePage />`
- Clicar em detalhes → URL muda (ex.: `/artigos/42`) mas o HTML base permanece o mesmo

React atualiza apenas a parte necessária da tela.

Referência: https://reactrouter.com

Virtual DOM e reconciliação

React mantém um Virtual DOM e, quando o estado muda, reconcilia diferenças para aplicar apenas as mudanças mínimas ao DOM real — resultado: código declarativo com boa performance.

Client-side vs Server-side Rendering

SPA clássica: renderização inicial no cliente. Com avanços recentes (Server Components, pré-rendering) há abordagens híbridas que mantêm a experiência de SPA após o carregamento.

React 19: visão geral

Principais pontos em React 19:

- Server Components e diretivas `use client` / `use server`
- Actions para formulários e mutações de dados
- Novos hooks para formulários e estados otimistas: `useActionState`, `useFormStatus`, `useOptimistic`
- Melhorias em refs e mensagens de erro de hidratação

Actions para formulários (React 19)

Conceito: funções que representam operações de envio de dados (síncronas ou assíncronas), podendo rodar no cliente ou no servidor.

Vantagens:

- Gerenciamento automático de estados (`enviando`, `erro`, `sucesso`)
- Atualização otimista com rollback em falhas

Transitions assíncronas

`useTransition` marca atualizações como não urgentes; em React 19 é mais simples usar funções assíncronas dentro de transições para carregar dados, tratar erros e mostrar estados de carregamento.

Server Components e diretivas

- `use client` — marca um arquivo como componente de cliente (permite hooks e eventos)
- `use server` — marca funções que rodam apenas no servidor

Benefícios: menos JS enviado ao cliente e melhor desempenho.

Partial Pre-rendering (React 19.2)

Permite pré-renderizar partes estáticas (o “shell”) e completar o conteúdo dinâmico depois, melhorando tempo para conteúdo interativo.

Boas práticas para SPAs

- Dividir a aplicação em páginas (rotas), componentes de layout e componentes de UI
- Separar lógica de dados (hooks, serviços) da lógica de apresentação
- Componentes pequenos e bem nomeados

Estrutura de pastas sugerida

Exemplo:

```
src/
  main.jsx
  App.jsx
  pages/
    HomePage.jsx
    ArticlesPage.jsx
  components/
    Header.jsx
    Footer.jsx
    ArticleList.jsx
    ArticleItem.jsx
  hooks/
    useArticles.js
  services/
```

Fluxo de desenvolvimento com Vite

Rotina:

- `npm run dev` para desenvolvimento (HMR)
- Editar `src/` e ver atualizações instantâneas
- `npm run build` para produção
- Servir `dist/` em um servidor estático

Depuração e ferramentas

- DevTools do navegador: Console, Network
- React DevTools: inspecionar árvore de componentes, props e estado

Boas práticas: manter componentes pequenos, nomear bem props/estado e usar logs/erros claros.

Checklist: sua primeira SPA com React 19

1. Instalar Node e escolher editor
2. Criar projeto com Vite
3. Garantir React 19 (`npm install react@latest react-dom@latest`)
4. Entender a estrutura: `index.html`, `main.jsx`, `App.jsx`
5. Criar componentes reutilizáveis (Layout, páginas, UI)
6. Usar JSX, `useState` e `props` para fluxo de dados

Encerramento e próximos passos

Próximos tópicos recomendados:

- Hooks adicionais: `useEffect`, `useReducer`, `useContext`
- React Router para navegação
- Integração com APIs e gerenciamento de dados
- Explorar Actions e Server Components em mais profundidade

Notas e referências: documentação oficial do React e Vite.