Tutorial Prático: React do Básico ao Intermediário (Vite + VS Code)

Neste tutorial, vamos construir passo a passo uma pequena aplicação React, do básico ao início do intermediário, usando **Visual Studio Code** e **Vite**. Veremos: **JSX, componentes funcionais, props, estado com** `useState`**, eventos, renderização condicional, listas e keys, formulários controlados e** `useEffect` — sempre com exemplos práticos, onde colocar cada código no projeto Vite, comandos de terminal e a saída esperada no navegador.

---

1. Introdução — Criando o Projeto React com Vite

O **React** é uma biblioteca JavaScript para construir UIs baseadas em **componentes**: blocos reutilizáveis com sua própria lógica e aparência. O React é eficiente ao atualizar o DOM usando um **DOM virtual** (atualiza só o que mudou).

Usaremos o **Vite** para criar e rodar o projeto — é rápido e simples. **Pré-requisito:** Node.js (recomendado **v18+**).

Passo a passo

1) Criar o projeto (terminal aberto na pasta desejada):

```
npm create vite@latest
```

Siga o prompt: - **Project name:** (ex.: `react-app`) - **Select a framework:** `react` - **Select a variant:** `javascript`

> Dica: em um comando só:
> `npm create vite@latest my-react-app -- --template react`

2) Instalar dependências e abrir o projeto

```
cd reactapp   # (ou o nome que você escolheu)
npm install
code .        # opcional: abre a pasta no VS Code
```

3) Iniciar o servidor de desenvolvimento

```
npm run dev
```

Acesse `http://localhost:5173`. Você verá a página inicial do Vite/React.

Estrutura inicial gerada:

```
/
├─ index.html
└─ src/
   ├─ main.jsx        # ponto de entrada do React DOM
   ├─ App.jsx         # componente raiz
   └─ assets/         # (pasta de assets)
```

> **Hot reload:** deixe `npm run dev` rodando; ao salvar arquivos no VS Code, o navegador atualiza automaticamente.

---

2. JSX e Renderização Básica

JSX é uma sintaxe parecida com HTML dentro do JavaScript. Ele vira chamadas JS reais (ex.: `React.createElement`), por isso: - Feche todas as tags corretamente. - Atributos seguem **camelCase** (ex.: `className`, `htmlFor`, `onClick`). - Um componente deve **retornar um único elemento pai** (use uma `<div>` ou `<>...</>`). - Use `{ ... }` para **expressões JS** dentro do JSX (variáveis, funções, operações, etc.).

Exemplo — renderização simples com JSX

Arquivo: `src/App.jsx`

```jsx
function App() {
  const nome = 'Mundo';
  return <h1>Olá, {nome}!</h1>;
}

export default App;
```

Salve e veja "Olá, Mundo!" no navegador.

---

3. Componentes Funcionais e Props

Um **componente funcional** é uma função JS que retorna JSX. **Props** são dados passados do pai para o filho.

Exemplo — componente de saudação com `props`

Organização: crie uma pasta `components` dentro de `src/`.

Arquivo: `src/components/Saudacao.jsx`

```jsx
function Saudacao(props) {
  return <h2>Olá, {props.nome}!</h2>;
}

export default Saudacao;
```

Usando em `App.jsx`**:**

```jsx
import Saudacao from './components/Saudacao';

function App() {
  return (
    <div>
      <Saudacao nome="Alice" />
      <Saudacao nome="Bruno" />
    </div>
  );
}

export default App;
```

Boas práticas com componentes/props - Um arquivo por componente (`PascalCase`). - Responsabilidade única. - **Props são somente leitura** (não modifique em filhos). - Tipos de props podem ser strings, números, booleanos, funções, objetos, elementos, etc.

---

4. Estado e o Hook `useState`

Estado é a "memória interna" do componente. Em funções, usamos **Hooks**: `const [valor, setValor] = useState(valorInicial)`.

Exemplo — `Contador` (cliques)

Arquivo: `src/components/Contador.jsx`

```jsx
import { useState } from 'react';

function Contador() {
  const [numero, setNumero] = useState(0);

  return (
    <div>
      <p>Você clicou {numero} vezes.</p>
      <button onClick={() => setNumero(n => n + 1)}>Incrementar</button>
    </div>
  );
}
```

```
export default Contador;
```

Usando em `App.jsx`:

```
import Contador from './components/Contador';

function App() {
  return (
    <div>
      <Contador />
    </div>
  );
}

export default App;
```

Dicas sobre estado: nunca faça `numero++` diretamente; use sempre `setNumero(...)`. Atualizações podem ser agrupadas/assíncronas.

---

5. Manipulação de Eventos

Eventos em JSX usam **camelCase** e você passa **funções** (não strings): `onClick={minhaFuncao}`.

Exemplo curto:

```
<button onClick={() => alert('Você clicou no botão!')}>
  Clique para ver o alerta
</button>
```

Para **submit** de formulário, use `event.preventDefault()` para evitar recarregar a página.

---

6. Renderização Condicional

Formas comuns: - `if/else` antes do `return` (múltiplos `return`s). - **Ternário**: `{cond ? <A/> : <B/>}`. - **AND lógico**: `{cond && <A/>}` (cuidado com `0`/string vazia).

Exemplo — `PainelLogin`

Arquivo: `src/components/PainelLogin.jsx`

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

function PainelLogin() {
```

```
  const [logado, setLogado] = useState(false);

  return (
    <div>
      {logado ? <p>Bem-vindo de volta!</p> : <p>Por favor, faça login.</p>}
      <button onClick={() => setLogado(v => !v)}>
        {logado ? 'Logout' : 'Login'}
      </button>
    </div>
  );
}

export default PainelLogin;
```

Usando em `App.jsx`**:**

```
import PainelLogin from './components/PainelLogin';

function App() {
  return (
    <div>
      <PainelLogin />
    </div>
  );
}

export default App;
```

---

7. Listas e Keys

Para renderizar listas, use `array.map(...)` e **sempre** defina `key` única por item.

Arquivo: `src/App.jsx` (exemplo simples)

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

function App() {
  const [tarefas, setTarefas] = useState([
    { id: 1, texto: 'Estudar React' },
    { id: 2, texto: 'Ler a documentação' }
  ]);

  return (
    <div>
      <h2>Lista de Tarefas</h2>
      <ul>
        {tarefas.map(t => (
```

```
          <li key={t.id}>{t.texto}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Use um **id** estável vindo dos dados. Evite `index` do array ou `Math.random()`.

---

8. Comunicação entre Componentes (Filho → Pai)

Fluxo de dados é **top-down**. Para o filho "avisar" o pai, o pai passa uma **função via props** (callback) e o filho a chama.

Exemplo — remover item da lista

Arquivo: `src/components/TaskItem.jsx`

```
function TaskItem({ tarefa, onRemover }) {
  return (
    <li>
      {tarefa.texto}
      {' '}
      <button onClick={() => onRemover(tarefa.id)}>Remover</button>
    </li>
  );
}

export default TaskItem;
```

Atualize `src/App.jsx`:

```
import { useState } from 'react';
import TaskItem from './components/TaskItem';

function App() {
  const [tarefas, setTarefas] = useState([
    { id: 1, texto: 'Estudar React' },
    { id: 2, texto: 'Ler a documentação' }
  ]);

  const removerTarefa = (id) => {
    setTarefas(ts => ts.filter(t => t.id !== id));
  };
```

```
  return (
    <div>
      <h2>Lista de Tarefas</h2>
      <ul>
        {tarefas.map(t => (
          <TaskItem key={t.id} tarefa={t} onRemover={removerTarefa} />
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Resumo: Pai → Filho = **props**; Filho → Pai = **função via props**. Irmãos compartilham estado "elevado" no pai comum.

---

9. Formulários Controlados

Em React, preferimos **inputs controlados**: o valor exibido vem do **estado** e atualizamos com `onChange`.

Atualize `src/App.jsx` para adicionar tarefas:

```
import { useState } from 'react';
import TaskItem from './components/TaskItem';

function App() {
  const [tarefas, setTarefas] = useState([
    { id: 1, texto: 'Estudar React' },
    { id: 2, texto: 'Ler a documentação' }
  ]);
  const [novaTarefa, setNovaTarefa] = useState('');

  const removerTarefa = (id) => {
    setTarefas(ts => ts.filter(t => t.id !== id));
  };

  const adicionarTarefa = (e) => {
    e.preventDefault();
    if (novaTarefa.trim() === '') return;
    const nova = { id: Date.now(), texto: novaTarefa.trim() };
    setTarefas(ts => [...ts, nova]);
    setNovaTarefa('');
  };

  return (
    <div>
```

```
      <h2>Lista de Tarefas</h2>

      {/* Formulário controlado */}
      <form onSubmit={adicionarTarefa}>
        <input
          type="text"
          value={novaTarefa}
          onChange={(e) => setNovaTarefa(e.target.value)}
          placeholder="Nova tarefa"
        />
        <button type="submit">Adicionar</button>
      </form>

      <ul>
        {tarefas.map(t => (
          <TaskItem key={t.id} tarefa={t} onRemover={removerTarefa} />
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Resumo do fluxo controlado: usuário digita → `onChange` atualiza estado → componente re-renderiza com o novo `value`. No submit → usamos o estado atual para adicionar a tarefa.

---

10. Introdução ao `useEffect`

`useEffect` lida com **efeitos colaterais** (APIs, timers, atualizar título, logs etc.).

Assinatura:

```
useEffect(() => {
  // efeito
  return () => { /* limpeza (opcional) */ };
}, [dependencias]);
```

- **Sem array:** roda após **toda** renderização.
- `[]` **vazio:** roda **uma vez** (montagem).
- `[foo, bar]`: roda ao montar e quando `foo` ou `bar` mudam.

Exemplo — boas-vindas e título dinâmico

Arquivo: `src/App.jsx` (com lista de tarefas)

```jsx
import { useState, useEffect } from 'react';
import TaskItem from './components/TaskItem';

function App() {
  const [tarefas, setTarefas] = useState([
    { id: 1, texto: 'Estudar React' },
    { id: 2, texto: 'Ler a documentação' }
  ]);
  const [novaTarefa, setNovaTarefa] = useState('');

  useEffect(() => {
    console.log('Aplicação iniciada! Bem-vindo.');
  }, []);

  useEffect(() => {
    document.title = `Tarefas: ${tarefas.length}`;
  }, [tarefas]);

  // ... (mesmo JSX da seção de formulários)
}
```

Exemplo de limpeza:

```jsx
useEffect(() => {
  const id = setInterval(() => {
    // ...
  }, 1000);
  return () => clearInterval(id);
}, []);
```

Boas práticas: - Liste corretamente as **dependências** usadas no efeito. - Separe efeitos por **responsabilidade**. - Evite loops de renderização ao atualizar estado dentro do próprio efeito. - Faça **cleanup** de timers, listeners e conexões.

Conclusão e Próximos Passos

Você criou uma app de tarefas com: JSX, componentes, props, `useState`, eventos, renderização condicional, listas/keys, comunicação filho→pai, formulário controlado e `useEffect`.

Sugestões para continuar: - **Estilização:** CSS, CSS Modules, Styled-Components, Tailwind. - **Testes:** Jest + React Testing Library. - **TypeScript:** use o template `react-ts` do Vite. - **Performance:** `React.memo`, `useMemo`, `useCallback`, `React.lazy`/`Suspense`, listas virtuais (`react-window`). - **APIs:** `fetch`/Axios em `useEffect` para carregar/persistir dados. - **Deploy:** `npm run build` e hospede (Netlify, Vercel, GitHub Pages, etc.).

Boas codificações!