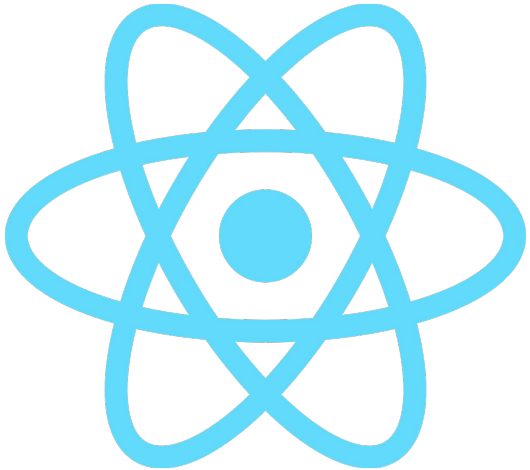

NETRA
TECNOLOGIA



Treinamento prático React

Dos Fundamentos à Arquitetura

Horário: Todas as manhãs das 09h às 12h

Sobre o Instrutor

- Nome: Hugo Henrique Oliveira Sampaio da Silva
- Formação: Graduação e Mestrado em Sistemas e Computação
- Experiência: +16 anos em desenvolvimento de software, atuando como arquiteto, líder técnico, já trabalhei em ReactJS, NodeJS, PHP, arquitetura de software, DDD e DevOps
- Atuação: Liderança técnica na Netra
- Propósito: Ajudar desenvolvedores a se tornarem profissionais completos, dominando de fundamentos a arquitetura avançada
- Curiosidade: Apaixonado por aprender, tecnologia, carros e família

Estrutura do Curso

- Duração: 26 horas (aproximadamente)
- Modalidade: Online (manhã)
- Formato: Teórico + Prático
- Entregas: Slides e Projeto

Objetivos de Aprendizagem

- Entender profundamente os fundamentos e a filosofia do React.
- Dominar os principais hooks e boas práticas associadas.
- Saber gerenciar estado local e global de forma eficiente.
- Consumir APIs externas com diferentes abordagens práticas.
- Construir formulários robustos e bem validados.
- Aplicar estilos modernos com CSS Modules, Styled Components e Tailwind.
- Escrever testes automatizados confiáveis para componentes React.
- Estruturar projetos escaláveis com boas práticas arquiteturais.
- Realizar deploy de aplicações React com processos automatizados (CI/CD).

Conteúdo Programático

1. Fundamentos do React
2. Hooks
3. Context API
4. Consumo de APIs
5. Formulários e Validação
6. Estilização de Componentes
7. Boas Práticas e Arquitetura
8. Redux com Redux Toolkit
9. React Router
10. Testes Automatizados
11. React com TypeScript
12. Deploy e CI/CD

Requisitos para o Curso

1. Conhecimento básico em JavaScript e React
2. Editor instalado (VScode/Cursor)
3. Node.js + npm/pnpm
4. Conta no GitHub (Desejável)

Ferramentas e Tecnologias

1. ReactJS
2. Redux Toolkit
3. React Router
4. React Hook Form / Zod
5. Styled Components / CSS Modules
6. React Testing Library / Jest
7. TypeScript
8. Create React App / Vite

Vamos começar?

Aplicações Tradicionais vs. SPA

Característica	Aplicação Tradicional	SPA (Single Page Application)
Navegação	Recarrega a página inteira	Carrega uma única vez
Tempo de resposta	Mais lento (novo HTML a cada clique)	Rápido (somente troca o necessário)
Experiência do usuário	Pode piscar ou reiniciar estado	Fluida, sem recarregamentos
Back-end	Renderiza HTML	Entrega dados (JSON/API)
Front-end	Mais simples, centrado no HTML	Mais dinâmico, usa JS frameworks
Exemplos	Wordpress, sistemas PHP tradicionais	Gmail, Facebook, Trello

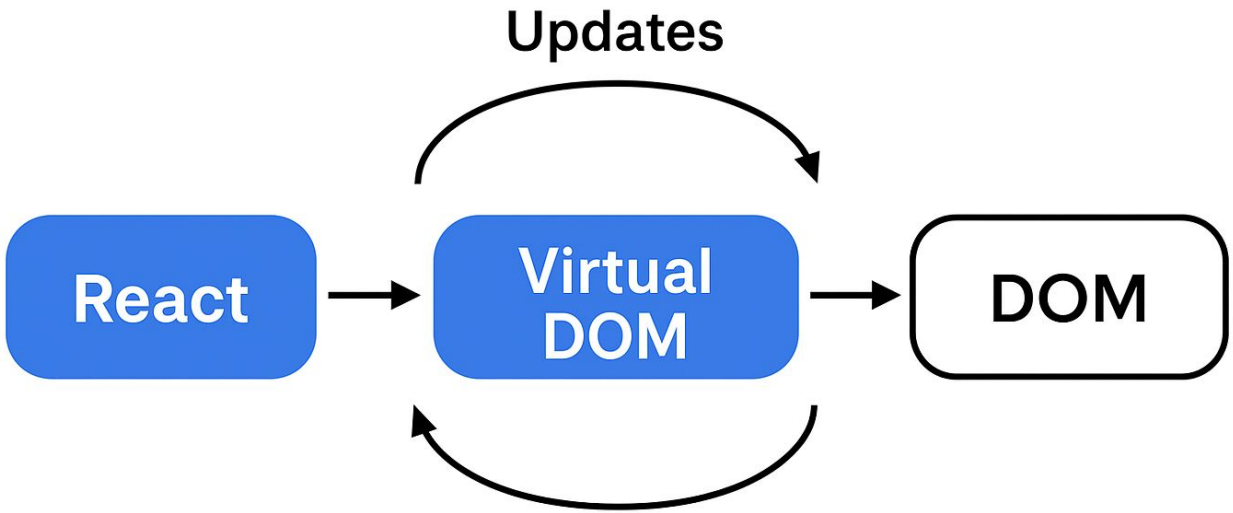
Fundamentos do ReactJS

- O que é React?
- JSX: JavaScript + HTML
- Template Expressions
- Componentes funcionais e reutilizáveis
- Props e State
- Eventos e manipulação de elementos
- Renderização condicional
- Listas com `.map()` e uso de `key`
- Hooks Essenciais

O que é ReactJS

React é uma biblioteca JavaScript para construção de interfaces de usuário, criada pelo Facebook. Sua principal vantagem é permitir a criação de **componentes reutilizáveis** e o **gerenciamento eficiente do DOM** com uma **abordagem declarativa**.

Virtual DOM



Virtual DOM

O React utiliza um conceito chamado Virtual DOM para otimizar a atualização da interface do usuário. Funciona assim:

- Renderização Inicial
- Alterações no Estado
- Comparação (Diffing)
- Atualização Eficiente

Essa abordagem é muito mais eficiente do que atualizar diretamente o DOM a cada mudança de estado, como acontece em manipulações imperativas com JavaScript puro.

JSX: JavaScript + HTML

JSX é uma sintaxe que combina HTML com JavaScript, permitindo escrever estruturas HTML diretamente dentro do código JS, facilitando a leitura e a manutenção.

Exemplo de um template JSX

```jsx
function BemVindo(props) {
  return <h1>Olá, {props.nome}!</h1>;
}

function App() {
  return (
    <div>
      <BemVindo nome="Maria"/>
      <BemVindo nome="João"/>
    </div>
  );
}
```

Template Expressions

- Inserção de expressões no JSX
- Renderização dinâmica
- Avaliação de variáveis e operações
- Boas práticas e limitações
- Inserção de expressões no JSX
- Renderização dinâmica
- Avaliação de variáveis e operações

Essas expressões podem conter:

- Variáveis
- Operações matemáticas ou lógicas
- Chamadas de função
- Operadores ternários (condicionais)

Exemplo 1 – variáveis

```
const nome = "Maria";
function Saudacao() {
  return <h1>Olá, {nome}!</h1>;
}
```

Exemplo 2 – operador ternário para condicional

```
const logado = true;
function Mensagem() {
  return <p>{logado ? "Bem-vindo de volta" : "Faça login para continuar"}</p>;
}
```

Exemplo 3 – chamada de função dentro do JSX:

```
function saudacaoUsuario(nome) {
  return `Olá, ${nome}!`;
}

function App() {
  return <h2>{saudacaoUsuario("Lucas")}</h2>;
}
```

Importante: dentro das chaves só é permitido usar expressões, não instruções (como if, for, etc.). Para esses casos, deve-se usar ternários ou lógica fora do retorno JSX.

Componentes

São funções (ou classes, em versões antigas) que retornam elementos React e formam a base da UI. Eles podem receber props, que são valores passados de pais para filhos, e controlar o state, que representa dados internos que podem mudar ao longo do tempo.

Componentes

```
function Saudacao() {
  return <p>Bem-vindo ao curso de React!</p>;
}

// Versão com propriedades
function Saudacao(props) {
  return <p>Olá, {props.nome}!</p>;
}

// Uso:
<Saudacao nome="Hugo" />
```

Composição de Componentes

Ao invés de criar interfaces grandes e complexas dentro de um único componente, o React incentiva a divisão da interface em componentes menores e reutilizáveis. Isso permite que um componente "pai" contenha e organize vários componentes "filhos", promovendo melhor organização, reuso de código e testabilidade.

Composição de Componentes

```
function Cabecalho() {
  return (
    <header><h1>Meu Site</h1></header>
  );
}

function Rodape() {
  return (
    <footer><p>&copy; 2025</p></footer>
  );
}

function Pagina() {
  return (
    <div>
      <Cabecalho />
      <main>
        <p>Conteúdo principal aqui</p>
      </main>
      <Rodape />
    </div>
  );
}
```

Componentes Funcionais e Reutilizáveis

- O que são componentes funcionais
- Benefícios da reutilização
- Separação de responsabilidades
- Exemplos práticos

Componentes Funcionais

Componentes funcionais são funções JavaScript que retornam elementos React. Eles são a forma mais comum e recomendada de escrever componentes atualmente, especialmente após a introdução dos Hooks.

```
function Alerta(props) {
  return <div className="alert">{props.mensagem}</div>;
}
```

Esse componente pode ser reutilizado em diferentes partes da aplicação:

```
<Alerta mensagem="Erro ao salvar os dados." />
<Alerta mensagem="Usuário cadastrado com sucesso." />
```

Props e State

Props (propriedades) são valores passados de um componente pai para um componente filho. Elas são imutáveis dentro do componente que as recebe e servem para personalizar ou parametrizar o comportamento e o conteúdo de um componente:

```
function Saudacao(props) {
  return <h1>Olá, {props.nome}!</h1>;
}

function App() {
  return <Saudacao nome="Hugo" />;
}
```

Neste exemplo, **nome** é passado como uma prop para o componente **Saudacao**.

Eventos

Eventos e manipulação de elementos no React seguem um padrão semelhante ao do JavaScript tradicional, mas com algumas particularidades.

- A nomeação dos eventos seguem camelCase: onClick, onChange, onSubmit, etc.
- O valor do manipulador de eventos deve ser uma função, e não uma string.
- O React utiliza um sistema de eventos sintéticos, que funciona de forma consistente em todos os navegadores.

```
function Botao() {
  function handleClick() {
    alert('Botão clicado!');
  }

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

Eventos também podem acessar o objeto de evento padrão:

```
function InputTexto() {
  function handleChange(event) {
    console.log('Valor digitado:', event.target.value);
  }

  return <input type="text" onChange={handleChange} />;
}
```

Além disso, é possível manipular diretamente elementos do DOM usando useRef quando necessário, embora isso seja raro em React moderno, pois a maioria das interações deve ser feita de forma declar

Renderização condicional

É feita com operadores como &&, ternários ou funções auxiliares para renderizar conteúdo com base em condições lógicas.

Para renderizar listas, usamos o método `.map()` e fornecemos uma `key` única para cada item, garantindo performance e comportamento consistente.

Fim do primeiro dia

Resumo do Dia 1 – Fundamentos do React

- O que é React e o conceito de SPA
- Criação e uso de componentes funcionais
- Utilização de props para passar dados
- Manipulação de eventos (ex: clique de botão)
- Diferença entre componentes de classe e componentes funcionais
- Como montar e rodar um ambiente React moderno
- Prática: Criar componentes, exibir informações e responder a eventos

Entendendo o Ciclo de Vida no React (Classes vs Hooks)

Fase do Ciclo de Vida	Com Classes	Com Hooks
Montagem (Mounting)	`constructor, componentDidMount`	useState inicial, useEffect (array vazio [])
Atualização (Updating)	`componentDidUpdate`	useEffect (com dependências específicas)
Desmontagem (Unmounting)	`componentWillUnmount`	Retorno do useEffect (cleanup)

Ciclo de Vida - Classes vs Hooks

Método (Classe)	Equivalente com Hooks	Quando utilizar?
`constructor`	`useState`	Inicialização de estados
`componentDidMount`	`useEffect([], ...)`	Carregamento inicial, requisições
`componentDidUpdate`	`useEffect([dependências])`	Atualizações específicas
`componentWillUnmount`	`useEffect(() ⇒ return cleanup, [])`	Limpeza, cancelamento de requisições

Class Components x Functional Components

Class Components

- Sintaxe baseada em ES6 classes.
- Ciclo de vida por métodos (componentDidMount, componentDidUpdate, etc).
- this obrigatório para acessar props, state e métodos.
- Mais verboso, mais difícil de reutilizar lógica.
- Antes de 2019, era necessário para recursos avançados (state, ciclo de vida).

```jsx
import React, { Component } from 'react';

class Contador extends Component {
  constructor(props) {
    super(props);
    this.state = { valor: 0 };
  }
  incrementar = () => {
    this.setState({ valor: this.state.valor + 1 });
  }
  render() {
    return (
      <div>
        <p>Valor: {this.state.valor}</p>
        <button onClick={this.incrementar}>Incrementar</button>
      </div>
    );
  }
}
```

Usando Classes:

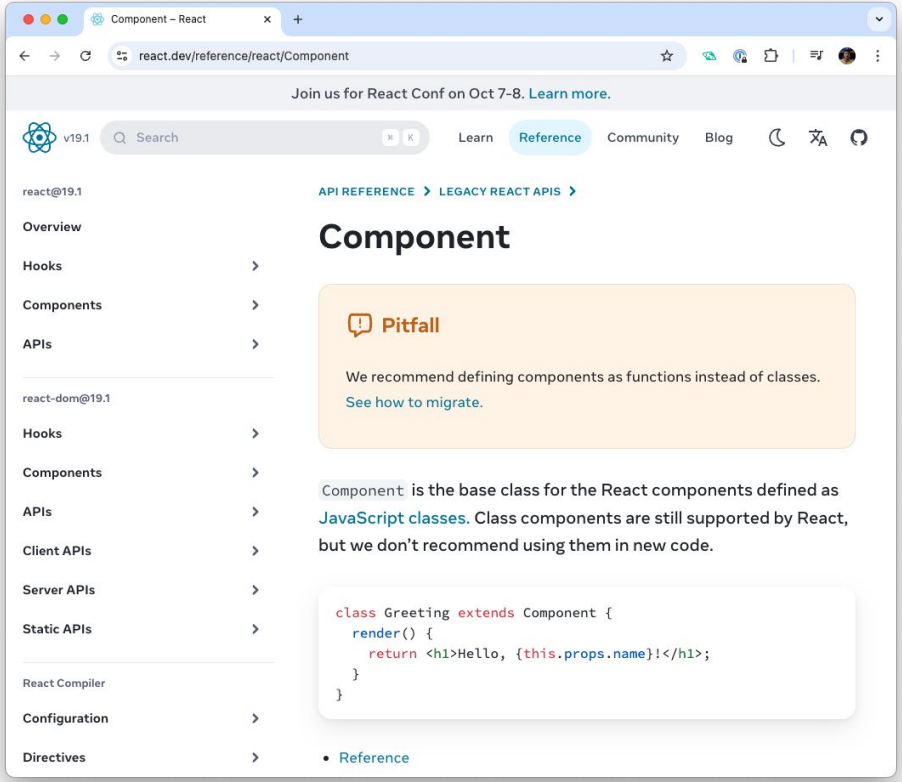
```
class MeuComponente extends React.Component {
  componentDidMount() {
    fetchDados();
  }

  componentWillUnmount() {
    limparDados();
  }

  render() {
    return <div>Meu Componente</div>;
  }
}
```

Usando Hooks:

```
function MeuComponente() {
  useEffect(() => {
    fetchDados();

    return () => limparDados();
  }, []);

  return <div>Meu Componente</div>;
}
```

A documentação oficial do React recomenda utilizar componentes funcionais com Hooks em vez de classes.

Essa recomendação é baseada em tornar o desenvolvimento mais simples, intuitivo, eficiente e menos propenso a erros, especialmente para novos projetos e funcionalidades.



Motivos da recomendação oficial

Simplicidade e Legibilidade
Componentes funcionais são mais fáceis de entender, escrever e testar.
Código menos verboso e mais claro.

Menos Propenso a Erros
Evitam problemas comuns com this, *binding* e métodos de ciclo de vida complexos.

Melhor Reutilização com Hooks
Hooks permitem a fácil reutilização de lógica de estado e ciclo de vida entre componentes sem a necessidade de complexidade como Higher-Order Components (HOC) ou Render Props.

Performance e Otimização
Hooks facilitam técnicas de performance (memoização, carregamento sob demanda) que antes eram mais complexas com classes.

Tendência de Mercado
Grande parte do ecossistema React (bibliotecas, tutoriais, documentação) está migrando para componentes funcionais e Hooks, tornando-os o padrão.

Hooks Essenciais

- **useState** controle de estado local
- **useEffect** efeitos colaterais e ciclo de vida
- **useRef** referência a elementos do DOM
- **useMemo** e **useCallback**: são utilizados para evitar cálculos ou recriações de funções desnecessárias, aumentando a performance da aplicação.

Hooks foram introduzidos no React 16.8 para permitir o uso de estado e outros recursos sem classes.

- **useState** permite armazenar e atualizar valores locais no componente.
- **useEffect** substitui métodos de ciclo de vida como **componentDidMount**, sendo executado após a renderização ou quando dependências mudam.
- **useRef** armazena uma referência mutável que persiste entre renderizações e pode ser usada para acessar elementos do DOM.
- **useMemo** e **useCallback** são utilizados para evitar cálculos ou recriações de funções desnecessárias, aumentando a performance da aplicação.
- Hooks customizados permitem encapsular lógica reutilizável, como formulários, autenticação, requisições, entre outros.

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

function Contador() {
  const [contador, setContador] = useState(0);

  return (
    <div>
      <p>Você clicou {contador} vezes</p>
      <button onClick={() => setContador(contador + 1)}>Clique aqui</button>
    </div>
  );
}
```

O **useState** retorna um array com dois valores: o estado atual e uma função que o atualiza. Ele é ideal para lidar com qualquer valor que precise mudar na interface.

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

function InputFoco() {
  const inputRef = useRef(null);

  const focarInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={focarInput}>Focar no input</button>
    </div>
  );
}
```

useRef cria uma referência que persiste entre renderizações, útil para acessar elementos do DOM diretamente.

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

function CalculoPesado({ numero }) {
  const resultado = useMemo(() => {
    let total = 0;
    for (let i = 0; i < 100000000; i++) {
      total += numero;
    }
    return total;
  }, [numero]);

  return <p>Resultado: {resultado}</p>;
}
```

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

function BotaoIncrementar() {
  const [contador, setContador] = useState(0);

  const incrementar = useCallback(() => {
    setContador((prev) => prev + 1);
  }, []);

  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={incrementar}>Incrementar</button>
    </div>
  );
}
```

Criando um Hook Customizado

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

function useWindowWidth() {
  const [largura, setLargura] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setLargura(window.innerWidth);
    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize', handleResize);
  }, []);

  return largura;
}

function App() {
  const largura = useWindowWidth();
  return <p>Largura da janela: {largura}px</p>;
}
```

Fim do segundo dia

Recapitulação – Dia 2: Classes, useState e useEffect

O que aprendemos no segundo dia de treinamento?

- Diferenças entre componentes de classe e funcionais
- Uso do this.state, setState e ciclo de vida em classes
- Introdução aos Hooks
    - useState – Gerenciar estado local
    - useEffect – Executar efeitos colaterais (ex: fetch, timer)
- Vantagens dos hooks:
    - Código mais limpo
    - Sem this
    - Mais reutilização e organização

*Especialmente hoje, vamos finalizar as 11h30

Context API

- Compartilhamento de estado global
- Cria e consume contextos
- Exemplo com `createContext`, `Provider` e `useContext`
- Boas práticas e performance

Exemplo

```
import { createContext, useContext } from 'react';

const TemaContext = createContext('claro');

function Botao() {
  const tema = useContext(TemaContext);
  return <button className={tema}>Clique aqui</button>;
}

function App() {
  return (
    <TemaContext.Provider value="escuro">
      <Botao />
    </TemaContext.Provider>
  );
}
```

```
// resposta.json
export const conta = {
  corrente: {
    saldo: 1000,
  },
  previdencia: {
    saldo: 5000,
  }
};

// conta-context.js
export const ContaContext = React.createContext(conta.corrente);

// app.js
import { ContaContext } from "./conta-context";

<ContaContext.consumer>
  <MostraSaldo/>
</ContaContext.consumer>

// mostraSaldo.js
import React from "react";
import { Text } from "react-native";

export default MostraSaldo()  {
  let conta = this.context;
  return (<Text>conta.saldo</Text>);
}
```

Context API

A estrutura básica envolve três passos:

1. Criar o contexto com `createContext`
2. Fornecer um valor via `Context.Provider`
3. Consumir o valor usando o hook `useContext`

É ideal para cenários como:

- Autenticação de usuário
- Tema (claro/escuro)
- Dados de configuração global
- Carrinho de compras ou sessão

Context API

- Evite contextos grandes demais com muitos dados e responsabilidades
- Separe diferentes domínios em múltiplos contextos (ex: `AuthContext`, `ThemeContext`)
- Prefira `useReducer` em conjunto com `Context` para lógica de atualização mais complexa
- Memorize o valor do Provider com `useMemo` quando ele depender de estado ou cálculo
- Documente os valores que fazem parte do contexto e sua finalidade

Context API com useReducer

Para estados globais complexos, combine Context com **useReducer**.

useReducer permite lógica mais organizada e escalável comparado a múltiplos estados.

```
const TemaContext = createContext();

function temaReducer(estado, ação) {
  switch (ação.tipo) {
    case 'ALTERAR_TEMA':
      return { tema: ação.tema };
    default:
      return estado;
  }
}

function TemaProvider({ children }) {
  const [estado, dispatch] = useReducer(temaReducer, { tema: 'claro' });

  return (
    <TemaContext.Provider value={{ estado, dispatch }}>
      {children}
    </TemaContext.Provider>
  );
}
```

Fim do terceiro dia

Estilização de Componentes

- Estilo global vs escopo local
- CSS Modules
- Styled Components com props dinâmicas
- CSS-in-JS vs pré-processadores tradicionais
- Temas e responsividade
- Boas práticas e organização
- Exemplos práticos com layout, botões e estados visuais

Em React, podemos utilizar **diferentes estrategias** para aplicar estilos aos componentes. Essa flexibilidade permite adotar abordagens modernas como: CSS Modules, Styled Components ou mesmo bibliotecas de design system.

CSS Modules – Exemplo

```
/* Botao.module.css */
.botao {
  background: #007bff;
  color: white;
  padding: 0.5rem 1rem;
  border: none;
  border-radius: 4px;
}
```

```
import styles from './Botao.module.css';

export function Botao() {
  return <button className={styles.botao}>Enviar</button>;
}
```

Styled Components – Exemplo com props dinâmicas

```
import styled from 'styled-components';

const Caixa = styled.div`
  background: ${props => (props.alerta ? 'tomato' : '#eee')};
  padding: 1rem;
  border-radius: 8px;
`;

function Alerta({ alerta }) {
  return <Caixa alerta={alerta}>Mensagem</Caixa>;
}
```

Styled Components – Exemplo com props dinâmicas

Com styled-components, o estilo responde dinamicamente a props.

Também é possível integrar temas globais via **ThemeProvider**.

```
const theme = {
  cores: {
    primario: '#6200ee',
    fundo: '#f5f5f5',
  },
};

<ThemeProvider theme={theme}>
  <App />
</ThemeProvider>
```

Tailwind CSS

```
export function Botao() {
  return <button className="bg-blue-500 text-white px-4 py-2">Clique</button>;
}
```

Fim do quarto dia

Formulários e Validação

Formulários e Validação

- Inputs controlados e estado
- Manipulação de formulários com `onChange` e `onSubmit`
- Diferenças entre inputs controlados e não controlados
- Boas práticas com formulários nativos
- Validação com Zod
- Integração com React Hook Form
- Máscaras com bibliotecas como `react-input-mask`
- Feedback visual de erro nos campos

Formulários e Validação

Os Formulários são frequentemente controlados, ou seja, os valores dos campos de input são sincronizados com o estado da aplicação usando `useState`.

Inputs controlados significam que o valor do campo vem diretamente do estado do componente e toda mudança é tratada por um evento `onChange`.

Diferenças principais:

- **Controlado**: Valor via state (value + `onChange`). Total controle sobre o input.
- **Não Controlado**: Acesso via ref, manipulação direta do DOM.

Inputs controlados vs não controlados

Controlados

- Valor é armazenado no useState.
- Todo evento onChange atualiza o estado.
- Permite validação em tempo real, consistência, e controle total.

```
const [nome, setNome] = useState("");
<input value={nome} onChange={(e) => setNome(e.target.value)} />
```

Não controlados

- Acesso via ref, não por useState.
- Mais rápido e leve, porém menos flexível.

```
const ref = useRef();
<input ref={ref} />
```

Vantagens de inputs controlados:

- Validação e transformação de dados em tempo real
- Consistência no comportamento
- Integração com bibliotecas como **Zod** e **React Hook Form**

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

function Formulario() {
  const [nome, setNome] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    alert(`Nome enviado: ${nome}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input value={nome} onChange={(e) => setNome(e.target.value)} />
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Criando formulários com React Hook Form

Para instalar

```
npm install react-hook-form
```

Exemplo básico

```
import { useForm } from "react-hook-form";

function Formulario() {
  const { register, handleSubmit } = useForm();

  const onSubmit = (dados) => console.log(dados);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("nome")} placeholder="Nome" />
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Fundamentos da validação de formulários

Antes de introduzir qualquer biblioteca (como Zod), é essencial garantir que os alunos entendam os conceitos fundamentais de validação:

Tipos de validação

Tipo	Onde acontece	Exemplo
HTML5	no navegador	required, type="email"
JS manual	em eventos onChange, onSubmit	if (!nome) return
Biblioteca externa	centraliza e organiza validações	Zod, Yup, Joi etc.

```
const onSubmit = (data) => {
  if (!data.nome) {
    alert("O campo nome é obrigatório!");
    return;
  }
  if (data.idade < 18) {
    alert("Você precisa ser maior de idade.");
    return;
  }
  console.log("Dados válidos:", data);
};
```

Validação com Zod

Para instalar

```
npm install zod @hookform/resolvers
```

```tsx
import { z } from "zod";
import { useForm } from "react-hook-form";
import { zodResolver } from "@hookform/resolvers/zod";

const schema = z.object({
  nome: z.string().min(3, "Nome muito curto"),
  idade: z.number().min(18, "Precisa ser maior de idade"),
});

function FormularioValidado() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm({ resolver: zodResolver(schema) });

  return (
    <form onSubmit={handleSubmit(console.log)}>
      <input {...register("nome")} />
      {errors.nome && <p>{errors.nome.message}</p>}

      <input type="number" {...register("idade", { valueAsNumber: true })} />
      {errors.idade && <p>{errors.idade.message}</p>}

      <button type="submit">Enviar</button>
    </form>
  );
}
```

Máscaras de entrada com react-imask

Para instalar

```
npm install react-imask
```

```
import { useForm, Controller } from "react-hook-form";
import { IMaskInput } from "react-imask";

<Controller
  name="cpf"
  control={control}
  render={({ field }) => (
    <IMaskInput
      mask="000.000.000-00"
      value={field.value}
      onAccept={(value) => field.onChange(value)}
      onBlur={field.onBlur}
    />
  )}
/>
```

Para validar formulários de forma moderna e declarativa, podemos utilizar o Zod, uma biblioteca de validação baseada em schemas.

Esse exemplo mostra a integração moderna entre React Hook Form e Zod, oferecendo validação tipada, feedback imediato e estrutura limpa de código.

```tsx
// 1. Schema de validação com Zod
const schema = z.object({
  nome: z.string().min(2, 'Nome muito curto'),
});

// 2. Inferência do tipo
type FormularioData = z.infer<typeof schema>;

function Cadastro() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm<FormularioData>({
    resolver: zodResolver(schema),
  });

  function onSubmit(data: FormularioData) {
    console.log('Dados validados:', data);
  }

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <div>
        <label>Nome:</label>
        <input type="text" {...register('nome')} />
        {errors.nome && <p>{errors.nome.message}</p>}
      </div>
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Fim do quinto dia

Consumo de APIs com React

Consumo de APIs com React

- Introdução ao ciclo de vida e side effects com useEffect
- Uso de fetch() com Promises e async/await
- Lidando com erros, estados de carregamento e feedback visual
- Estratégias de separação de responsabilidades: hooks customizados e serviços
- Introdução ao axios como solução mais completa

O que é consumir uma API?

Consumir uma API" significa fazer uma requisição HTTP (GET, POST, etc.) a um servidor remoto para buscar ou enviar dados.

Métodos comuns: fetch vs axios

	`fetch` (nativo)	`axios` (externo)
Simples de usar	✅	✅
Precisa processar `.json()`	✅	❌
Interceptadores e configs globais	❌	✅
Suporte automático a timeout, cancelamento	❌	✅
Mais leve (sem dependência)	✅	❌

Consumo de APIs com React

APIs REST são frequentemente utilizadas para buscar dados que alimentam a interface do usuário.

```
function ListaUsuarios() {
  const [usuarios, setUsuarios] = useState([]);
  const [carregando, setCarregando] = useState(true);
  const [erro, setErro] = useState(null);

  useEffect(() => {
    async function carregarUsuarios() {
      try {
        const resposta = await fetch('https://app.domain.com');
        if (!resposta.ok) {
          throw new Error('Erro ao buscar usuários');
        }
        const dados = await resposta.json();
        setUsuarios(dados);
      } catch (err) {
        setErro(err.message);
      } finally {
        setCarregando(false);
      }
    }

    carregarUsuarios();
  }, []);

  return (
    <ul>
      {usuarios.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Quando fazer a requisição? `useEffect`

Componentes funcionais não tem ciclo de vida "mounted" como classes antigas.

Por isso usamos o hook `useEffect` para fazer chamadas de API no momento certo.

Requisições com `axios`

```
npm install axios
```

```
import axios from 'axios';

useEffect(() => {
  axios.get('https://jsonplaceholder.typicode.com/users')
    .then(res => setUsuarios(res.data))
    .catch(err => setErro(err.message))
    .finally(() => setLoading(false));
}, []);
```

Tratamento de erros

- Sempre usar try/catch
- Tratar res.ok em fetch
- Mostrar mensagens ao usuário

Boas práticas

- Mostrar loading enquanto carrega
- Mostrar mensagem de erro se falhar
- Evitar chamadas desnecessárias (com dependências corretas no `useEffect`)
- Separar requisições em um serviço (`api.js`)

Recomendação de estrutura

```
src/
  services/
    api.js
  pages/
    Usuarios.jsx
```

api.js

```
import axios from 'axios';

export const api = axios.create({
  baseURL: 'https://jsonplaceholder.typicode.com',
  timeout: 5000,
});
```

Usuarios.jsx

```
useEffect(() => {
  api.get('/users')
    .then(res => setUsuarios(res.data))
    .catch(err => setErro(err.message))
    .finally(() => setLoading(false));
}, []);
```

Consumo de APIs com React

Boas práticas com fetch

- Encapsular lógica em hooks reutilizáveis (useUsuarios, useProdutos)
- Utilizar estados de carregamento e erro
- Evitar then/catch aninhados – prefira async/await

```
// api/usuarios.js
export async function getUsuarios() {
  const res = await fetch('/api/usuarios');
  if (!res.ok) throw new Error('Erro ao buscar');
  return await res.json();
}

// Componente
import { useEffect, useState } from 'react';
import { getUsuarios } from './api/usuarios';

function App() {
  const [usuarios, setUsuarios] = useState([]);

  useEffect(() => {
    getUsuarios().then(setUsuarios).catch(console.error);
  }, []);

  return (
    <ul>{usuarios.map(u => <li key={u.id}>{u.nome}</li>)}</ul>
  );
}
```

Fim do sexto dia

Boas Práticas e Arquitetura em Projetos React

Boas Práticas e Arquitetura em Projetos React

- Compreender como estruturar um projeto React de forma escalável.
- Aplicar padrões de código que facilitam manutenção e evolução.
- Conhecer práticas recomendadas para componentes, hooks, contexto e estado global.
- Entender separação de responsabilidades e padrões arquiteturais.

```
src/
  app/                  # Configurações globais e bootstrapping
    providers/          # Providers globais (Theme, Auth, QueryClient, etc.)
    routes/             # Definição de rotas e layouts raiz
    store/              # Estado global compartilhado (Redux/Zustand)
    styles/             # Estilos e temas globais
    utils/              # Funções utilitárias compartilhadas

  shared/               # Código genérico reutilizável entre domínios
    components/         # Botões, inputs, modais reutilizáveis
    hooks/              # Hooks genéricos
    services/           # Serviços externos (ex: api.js, httpClient)
    types/              # Tipos e interfaces globais
    constants/          # Constantes globais

  modules/              # Módulos/Domínios principais (Bounded Contexts)
    users/              # Contexto de Usuários
      components/       # Componentes específicos do domínio
      hooks/            # Hooks específicos (ex: useUserList, useUserForm)
      pages/            # Páginas específicas do domínio
      services/         # Lógica de integração do domínio
      types/            # Tipos e interfaces do domínio
    orders/             # Contexto de Pedidos
      components/
      hooks/
      pages/
      services/
      types/
    products/           # Contexto de Produtos
      ...

  index.tsx
  App.tsx
```

Principais características dessa abordagem

Isolamento por domínio

- Cada contexto (ex: users, orders, products) tem seus próprios componentes, hooks, serviços e tipagens.
- Evita que mudanças em um domínio afetem outros desnecessariamente.

Reuso via camada shared/

- Tudo que é genérico e pode ser usado por qualquer módulo vai para shared.
- Exemplo: shared/components/Button, shared/hooks/useDebounce.

Bootstrap central no app/

- Providers globais, configuração de rotas, inicialização de estado global.
- O app/ é o ponto de entrada para compor tudo.

Facilita testes e manutenção

- Módulos podem ser testados e até migrados para micro frontends sem impacto em outros.

Separação de Responsabilidades

Os componentes devem ser organizados com responsabilidades claramente definidas para facilitar manutenção e escalabilidade.

Benefícios dessa estrutura para projetos grandes

- **Escalabilidade**: fácil adicionar novos domínios sem bagunçar os existentes.
- **Clareza**: código de cada domínio fica próximo e autoexplicativo.
- **Refatoração segura**: um módulo pode ser movido ou alterado sem impacto global.
- **Preparado para microfrontends**: cada module pode virar um microfrontend.

Smart Components (Container)

Características principais:

- Responsáveis por gerenciar estados complexos.
- Realizam operações assíncronas (ex.: chamadas a APIs).
- Controlam a lógica e o fluxo de dados da aplicação.
- Interagem diretamente com serviços externos.

Quando usar?

- Componentes de página ou visualizações principais.
- Componentes que precisam buscar e manipular dados externos

Smart Components (Container)

Exemplo de uso:

```
import React, { useState, useEffect } from 'react';
import ProductList from '../components/ProductList';
import { fetchProducts } from '../services/api';

function ProductsContainer() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetchProducts()
      .then(data => setProducts(data))
      .catch(err => setError(err.message))
      .finally(() => setLoading(false));
  }, []);

  if (loading) return <div>Carregando produtos...</div>;
  if (error) return <div>Erro: {error}</div>;

  return <ProductList products={products} />;
}
```

Dumb Components (Presentational/UI)

Características principais:

- Não têm lógica complexa ou operações assíncronas.
- Recebem todos os dados via props.
- Focados exclusivamente na apresentação visual.
- Facilmente reutilizáveis e testáveis.

Quando usar?

- Elementos de interface reutilizáveis (botões, inputs, cards, etc.).
- Apresentação pura de dados recebidos.

Componentes Dumb (Presentational/UI)

Exemplo de uso:

```
import React from 'react';

function ProductList({ products }) {
  return (
    <ul>
      {products.map(product => (
        <li key={product.id}>
          {product.name} - R$ {product.price}
        </li>
      ))}
    </ul>
  );
}

export default ProductList;
```

Resumo visual da responsabilidade:

`Recebe Props → Renderização direta`

Arquitetura com Smart e Dumb Components

Exemplo prático de uso em um cenário real

```jsx
// Smart (Container)
function Dashboard() {
  const [userData, setUserData] = useState(null);

  useEffect(() => {
    fetchUserData().then(data => setUserData(data));
  }, []);

  if (!userData) return <Loader />;

  return (
    <div>
      <ProfileCard user={userData.user} />
      <Statistics data={userData.stats} />
    </div>
  );
}
```

```jsx
// Dumb (Presentational/UI)
function ProfileCard({ user }) {
  return (
    <div>
      <img src={user.avatar} alt="User Avatar" />
      <h1>{user.name}</h1>
      <p>{user.bio}</p>
    </div>
  );
}

function Statistics({ data }) {
  return (
    <ul>
      <li>Seguidores: {data.followers}</li>
      <li>Seguindo: {data.following}</li>
    </ul>
  );
}
```

Por que Separar essas Responsabilidades?

Vantagens dessa abordagem:

- **Testabilidade:** Dumb Components são facilmente testáveis isoladamente.
- **Manutenção:** Lógica concentrada em Containers simplifica manutenção e refatoração.
- **Escalabilidade:** Facilita expansão do projeto, já que componentes reutilizáveis são claramente definidos.

Melhores práticas sugeridas:

- Mantenha componentes UI o mais simples possível.
- Prefira poucos Containers com lógica centralizada.
- Não permita Dumb Components acessarem diretamente APIs ou contextos complexos.

Boas práticas de código

Nomeação

- Componentes: `PascalCase` → `UserProfile.jsx`
- Funções e variáveis: `camelCase`
- Constantes globais: `SCREAMING_SNAKE_CASE`

Separação de responsabilidades

- Não misture lógica de API dentro de componentes de UI.
- Use services para chamadas HTTP.

```
// services/userService.js
export async function getUsers() {
  const res = await fetch('/api/users');
  if (!res.ok) throw new Error('Erro ao buscar usuários');
  return res.json();
}
```

Hooks personalizados

Encapsular lógica reutilizável.

```
// hooks/useFetch.js
import { useEffect, useState } from 'react';

export function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url).then(res => res.json()).then(data => {
      setData(data);
      setLoading(false);
    });
  }, [url]);

  return { data, loading };
}
```

```
const { data: users, loading } = useFetch('/api/users');
```

Gerenciamento de estado
Encapsular lógica reutilizável.

- Local: useState, useReducer
- Global: Context API, Redux, Zustand
- Server State: React Query, SWR

Boas práticas de performance
Encapsular lógica reutilizável.

- **Memoização**: `React.memo, useCallback, useMemo`
- **Lazy loading**: `React.lazy, Suspense`
- **Code splitting**: carregamento sob demanda de módulos/páginas

```
const LazyComponent = React.lazy(() => import('./MyComponent'));
```

Configurações e ferramentas de qualidade

- **ESLint:** regras de lint para consistência de código.
- **Prettier:** formatação automática.
- **Husky** + **lint-staged:** validações antes do commit.

Padrões arquiteturais úteis

- **Atomic Design** (Atoms, Molecules, Organisms, Templates, Pages)
- **Feature-based folders** (agrupar por funcionalidade)
- **Clean Architectur**e adaptada para front-end

Hexagonal Architecture (Ports & Adapters)

No contexto de React:

- **Core (Domínio)**:
  - Regras de negócio puras, sem dependência de libs de UI.
  - Tipos, validações, lógica de cálculo.
- **Ports (Interfaces)**:
  - Definem **o que** a aplicação precisa de fora (ex: `IUserRepository`).
- **Adapters (Implementações)**:
  - Concretizam os ports usando tecnologias específicas (ex: `UserApiRepository` com Axios).
- **UI Layer**:
  - Consome os ports e apresenta os dados.

```
src/
  core/                  # Domínio puro
    user/
      entities/          # Modelos de domínio (User.ts)
      usecases/          # Casos de uso (GetUserList.ts)
      ports/             # Interfaces (IUserRepository.ts)
  infrastructure/
    http/                # Cliente HTTP (axiosInstance.ts)
    repositories/        # Implementações de ports (UserApiRepository.ts)
  ui/
    components/          # UI Components
    pages/               # Páginas (UserListPage.tsx)
  app/
    providers/           # Contextos, roteamento
```

Fluxo resumido:

1. **UI** → chama caso de uso (`GetUserList`)
2. Caso de uso → depende de `IUserRepository` (port)
3. Implementação concreta (`UserApiRepository`) → busca dados com HTTP
4. Dados retornam ao caso de uso → UI renderiza

Testabilidade

- Componentes devem ser pequenos e testáveis.
- Use Jest + React Testing Library.
- Evite dependências fortes de dados externos (mock de API).

```
test('renderiza nome do usuário', () ⇒ {
  render(<UserCard name="João" email="joao@email.com" />);
  expect(screen.getByText('João')).toBeInTheDocument();
});
```

Fim do sétimo dia

Gerenciamento de Estado Global em Aplicações React (Avançado)

- Entender por que e quando usar estado global.
- Comparar diferentes abordagens de gerenciamento de estado.
- Aplicar Redux moderno com Redux Toolkit (RTK).
- Criar uma aplicação com fluxo completo (actions ➜ reducer ➜ UI).
- Introduzir padrões avançados (middleware, RTK Query).

O que é Estado Global?

- Estado compartilhado entre múltiplos componentes.
- Qualquer alteração deve refletir automaticamente em todos os pontos da aplicação que consomem aquele estado.

Exemplos de uso:

- Autenticação do usuário.
- Tema (dark/light mode).
- Carrinho de compras.
- Dados carregados de API que precisam ser exibidos em várias telas.

Gerenciamento de Estado Global em Aplicações React (Avançado)

Por que pensar em gerenciamento de estado?

- À medida que a aplicação cresce, múltiplos componentes passam a depender do mesmo conjunto de dados.
- Componentes distantes na árvore precisam compartilhar e reagir a mudanças no estado.
- O gerenciamento inadequado leva a prop-drilling, duplicação de lógica e inconsistência de dados.

Abordagem	Uso Ideal	Limitações
useState + Props	Estado local, simples, curto alcance	Prop-drilling, difícil manutenção
Context API	Estado global simples (tema, idioma, auth)	Performance em grandes apps, pouco flexível
Redux	Estado global complexo e escalável	Requer estruturação e aprendizado extra

Por que não usar apenas Context API?

- Context é ótimo para dados pouco mutáveis.
- Mas em casos com **muitas atualizações** ou **lógica complexa**, Context pode gerar re-renders **desnecessários** e ser mais difícil de escalar.
- Redux traz:
    - **Imutabilidade controlada**
    - **Ferramentas de debug (Redux DevTools)**
    - **Escalabilidade** para projetos grandes

Gerenciamento de Estado com Redux Toolkit

Por que Redux?

O Redux é uma biblioteca para gerenciamento previsível de estado, ideal para aplicações em que muitos componentes precisam acessar e modificar os mesmos dados.

O que é Redux Toolkit?

O Redux Toolkit (RTK) é a forma moderna e oficial recomendada para usar Redux. Ele abstrai a complexidade do Redux clássico, simplifica a criação de store e promove boas práticas por padrão.

Arquitetura do Redux

- **Store**: armazena o estado global.
- **Slice**: conjunto de estado + reducers + actions.
- **Dispatch**: dispara ações para alterar o estado.
- **Selector**: lê o estado da store.

```
UI → dispatch(action) → reducer → novo estado → UI re-renderiza
```

Principais Conceitos do Redux Toolkit

Slice – representa uma fatia do estado global da aplicação. Ele contém:

- O estado inicial dessa fatia.
- Os reducers (funções que atualizam o estado).
- As ações associadas a esses reducers.

O **createSlice** automaticamente gera **action creators** e o **reducer** correspondente.

Store – é o objeto que contém o **estado global da aplicação**. Ela é criada com **configureStore**, que combina todos os **reducers** (ou **slices**) e aplica middlewares automaticamente.

Thunk – é uma função assíncrona usada para lidar com efeitos colaterais, como requisições a APIs. O RTK facilita isso com o **createAsyncThunk**, que gera automaticamente ações para pending, fulfilled e rejected.

Estrutura típica com Redux Toolkit

```
src/
├── store/
│   ├── index.ts            // Configuração da store
│   ├── produtosSlice.ts    // Slice de exemplo
│   └── ...                 // Outros slices
```

Redux Toolkit – Passo a Passo

1. Criação do slice

```
// produtosSlice.ts
import { createSlice } from '@reduxjs/toolkit';

const produtosSlice = createSlice({
  name: 'produtos',
  initialState: [],
  reducers: {
    adicionarProduto: (state, action) => {
      state.push(action.payload);
    },
    limparProdutos: () => []
  }
});

export const { adicionarProduto, limparProdutos } = produtosSlice.actions;
export default produtosSlice.reducer;
```

Redux Toolkit – Passo a Passo

2. Configuração da store

```
// store/index.ts
import { configureStore } from '@reduxjs/toolkit';
import produtosReducer from './produtosSlice';

export const store = configureStore({
  reducer: {
    produtos: produtosReducer
  }
});
```

Redux Toolkit – Passo a Passo

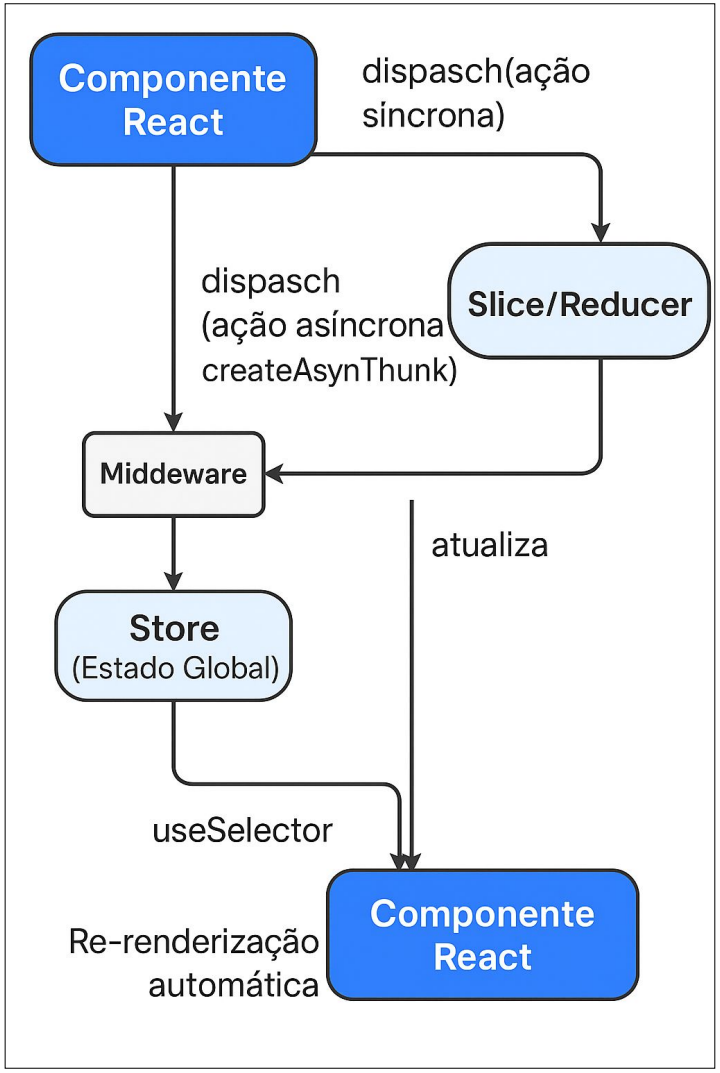
3. Conectando com React

```
// index.tsx ou App.tsx
import { Provider } from 'react-redux';
import { store } from './store';

<Provider store={store}>
  <App />
</Provider>
```

Redux Toolkit – Passo a Passo

4. Usando o estado em componentes

```
import { useSelector, useDispatch } from 'react-redux';
import { adicionarProduto } from '../store/produtosSlice';

function Produtos() {
  const produtos = useSelector((state) => state.produtos);
  const dispatch = useDispatch();
  return (
    <div>
      <button onClick={() => dispatch(adicionarProduto({ id: 1, nome: 'Mouse' }))}>
        Adicionar Produto
      </button>
      <ul>
        {produtos.map((p) => (<li key={p.id}>{p.nome}</li>))}
      </ul>
    </div>
  );
}
```


Componente React
dispasch(ação síncrona)
dispasch (ação asíncrona createAsynThunk)
Slice/Reducer
Middeware
atualiza
Store (Estado Global)
useSelector
Componente React
Re-renderização automática

Quando usar Redux?

- Estados compartilhados entre muitos componentes.
- Regras de negócio complexas.
- Estado que precisa ser persistido entre rotas ou telas.
- Casos em que é necessário histórico, logs ou ações previsíveis.

Quando não usar Redux?

- Estado local simples e específico de um componente.
- Aplicações pequenas ou com poucos fluxos e interações.

💾 Persistindo o Redux

A biblioteca mais usada é o **redux-persist**, que integra com Redux Toolkit de forma simples.

Benefícios de persistir

1. **Experiência do usuário melhor**
    - Estado é mantido após recarregar a página.
    - Ex.: carrinho de compras, preferências de tema, idioma.

2. **Menos chamadas de API**
    - Dados já carregados anteriormente podem ser reaproveitados.
    - Bom para listas grandes ou dashboards.

3. **Sessão mais consistente**
    - Mantém status de autenticação (token, dados do usuário) sem precisar logar novamente.

⚠️ Desvantagens e riscos

1. **Armazenar dados obsoletos**
    - Dados podem ficar desatualizados se a fonte mudar e você não fizer sincronização.
2. **Segurança**
    - Tudo que vai para localStorage ou sessionStorage pode ser lido pelo usuário.
    - Jamais persistir senhas ou dados sigilosos sem criptografia.
3. **Tamanho e performance**
    - Objetos muito grandes aumentam o tempo de leitura e escrita no storage.
4. **Controle de limpeza**
    - Necessário limpar estado persistido em eventos como logout.

Boas práticas ao persistir Redux

- Persistir **apenas o que é necessário** (usar `whitelist` ou `blacklist` no `persistConfig`).
- Atualizar o estado ao iniciar a aplicação (sincronizar com a API).
- Limpar dados no logout `persistor.purge()`
- Usar storage adequado:
    - **localStorage**: persiste até ser apagado.
    - **sessionStorage**: expira quando o navegador é fechado.
    - **IndexedDB**: para dados grandes.

💾 Persistindo o Redux

A biblioteca mais usada é o **redux-persist**, que integra com Redux Toolkit de forma simples.

Instalação

```
npm install redux-persist
```

```
// store.ts
import { configureStore } from '@reduxjs/toolkit';
import { persistStore, persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage'; // usa localStorage
import counterReducer from './counterSlice';

const persistConfig = {
  key: 'root',
  storage,
  whitelist: ['counter'] // slices que serão persistidos
};

const persistedReducer = persistReducer(persistConfig, counterReducer);

export const store = configureStore({
  reducer: {
    counter: persistedReducer
  }
});

export const persistor = persistStore(store);
```

Integrando com o Provider

```tsx
// main.tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import { PersistGate } from 'redux-persist/integration/react';
import { store, persistor } from './store';
import App from './App';

ReactDOM.createRoot(document.getElementById('root')!).render(
  <Provider store={store}>
    <PersistGate loading={<div>Carregando...</div>} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>
);
```

Fim do oitavo dia

React Router – Navegação em Aplicações React

Por que usar o React Router?

- Navegação sem recarregar a página
- Roteamento baseado em URL
- Rotas aninhadas
- Rotas dinâmicas
- Proteção de rotas (ex: autenticação)
- Navegação programática

React Router – Navegação em Aplicações React

Vantagens

- Navegação sem recarregar a página
- URLs amigáveis
- Suporte a parâmetros e rotas protegidas
- Possibilita rotas aninhadas e layouts complexos

Como instalar

```
npm install react-router-dom
```

React Router – Navegação em Aplicações React

Principais conceitos e componentes

- **BrowserRouter:** Componente de alto nível, habilita o roteamento baseado na URL.
- **Routes:** Componente que contém todas as rotas.
- **Route:** Define cada caminho e o componente a ser exibido.
- **Link:** Substitui <a>, permite navegação sem reload.
- **useNavigate:** Hook para navegação programática.

<BrowserRouter>

Componente que envolve a aplicação inteira para ativar o roteamento baseado em histórico.

```
import { BrowserRouter } from 'react-router-dom';

<BrowserRouter>
  <App />
</BrowserRouter>
```

`<Routes>` e `<Route>`

Define o conjunto de rotas e quais componentes serão renderizados.

```
import { Routes, Route } from 'react-router-dom';

<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
</Routes>
```

`<Link>` e `<NavLink>`

Permite navegação entre rotas sem recarregar a página.

```
import { Link } from 'react-router-dom';

<Link to="/about">Sobre</Link>
```

useParams

Captura parâmetros dinâmicos da URL.

```
<Route path="/user/:id" element={<User />} />

// no componente User
const { id } = useParams();
```

`useNavigate`

Permite navegação programática via código.

```
const navigate = useNavigate();
navigate('/dashboard');
```

Rota coringa (NotFound)

```
<Route path="*" element={<NotFound />} />
```

Exemplo Básico de Roteamento

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link> | <Link to="/produtos">Produtos</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/produtos" element={<ProdutoList />} />
        <Route path="/produto/:id" element={<ProdutoDetalhe />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Rotas com parâmetros

```
import { useParams } from "react-router-dom";

function ProdutoDetalhe() {
  const { id } = useParams();
  return <h2>Produto ID: {id}</h2>;
}

// Uso:
<Route path="/produto/:id" element={<ProdutoDetalhe />} />
```

Navegação programática

```jsx
import { useNavigate } from "react-router-dom";

function BotaoLogout() {
  const navigate = useNavigate();
  return (
    <button onClick={() => {
      // Lógica de logout...
      navigate("/login");
    }}>
      Sair
    </button>
  );
}
```

Protegendo Rotas (Auth) – Exemplo Básico

```
import { Navigate, useLocation } from "react-router-dom";

function PrivateRoute({ children }) {
  const logado = Boolean(localStorage.getItem("token"));
  const location = useLocation();
  return logado ? children : <Navigate to="/login" state={{ from: location }} />;
}

<Route path="/dashboard" element={<PrivateRoute><Dashboard /></PrivateRoute>} />
```

```
function PrivateRoute({ children }) {
  const auth = useAuth();
  return auth.user ? children : <Navigate to="/login" />;
}

<Route path="/admin" element={<PrivateRoute><Admin /></PrivateRoute>} />
```

Navegação com estado

```
navigate('/checkout', { state: { total: 100 } });

const location = useLocation();
console.log(location.state.total);
```

Organização das rotas por contexto de domínio

Separar arquivos de rotas por domínio de negócio

```
routes/
  publicRoutes.jsx
  authRoutes.jsx
  dashboardRoutes.jsx
```

E importar todos num AppRoutes.jsx central.

React Router – Navegação em Aplicações React

Boas práticas

- Centralize as rotas principais do app em um único arquivo (ex: AppRoutes.tsx).
- Use Outlet para layouts de páginas que compartilham navegação ou menus.
- Use useParams e useNavigate para rotas dinâmicas e navegação controlada por código.
- Prefira <Link> a <a> para não recarregar a página.
- Não exagere em rotas aninhadas se não houver necessidade de layouts compartilhados.
- Componha rotas protegidas via HOCs ou wrappers.

Fim do nono dia

Testes Automatizados em React

Testes Automatizados em React

Por que testar?

Testes automatizados são essenciais para garantir a qualidade, robustez e evolução saudável de qualquer aplicação. Eles ajudam a:

- Encontrar erros rapidamente antes que cheguem ao usuário final.
- Facilitar refatorações e mudanças no código sem medo de “quebrar” funcionalidades.
- Servir como documentação viva do funcionamento dos componentes.

Testes Automatizados em React

Tipos de teste em aplicações React

- **Testes unitários:** Verificam o funcionamento isolado de funções, hooks ou componentes pequenos.
- **Testes de integração:** Avaliam como diferentes componentes ou partes do sistema trabalham juntas.
- **Testes end-to-end (E2E):** Simulam o uso real do usuário em todo o sistema (usando ferramentas como Cypress ou Playwright).

Testes Automatizados em React

Principais ferramentas para testes em React

- **Jest**: Framework de testes padrão para aplicações JavaScript/TypeScript, fácil de configurar, rápido e com excelente integração ao React.
- **React Testing Library**: Foca em testar o comportamento do usuário, simulando interações reais na interface. Incentiva testes mais robustos, sem depender de detalhes de implementação.

Exemplo simples de teste com Jest + React Testing Library

```
import { render, screen, fireEvent } from "@testing-library/react";
import ProdutoCard from "./ProdutoCard";

test("renderiza o nome e o preço do produto", () => {
  render(<ProdutoCard produto={{ id: 1, nome: "Notebook", preco: 2000 }} />);
  expect(screen.getByText(/Notebook/)).toBeInTheDocument();
  expect(screen.getByText(/R\$ 2000.00/)).toBeInTheDocument();
});

test("chama ação ao clicar no botão", () => {
  const onAdicionar = jest.fn();
  render(<ProdutoCard produto={{ id: 1, nome: "Notebook", preco: 2000 }} onAdicionar={onAdicionar} />);
  fireEvent.click(screen.getByRole("button"));
  expect(onAdicionar).toHaveBeenCalled();
});
```

Testes Automatizados em React

Boas práticas em testes React

- Escreva testes que simulem o uso real do componente pelo usuário.
- Prefira getByText, getByRole, etc. ao invés de buscar por classes ou IDs específicos.
- Teste cenários positivos e negativos (exemplo: mostrar erro de validação quando campo está vazio).
- Mantenha os testes rápidos e fáceis de entender.

React com TypeScript

Vantagens do TypeScript no React

- **Segurança de tipos:**
  Detecta muitos erros em tempo de desenvolvimento (antes mesmo de rodar o app).

- **Refatoração facilitada:**
  Trocar nome de props, extrair componentes, alterar estados: tudo fica mais seguro.

- **Autocompletar mais inteligente:**
  Melhora a produtividade e reduz dúvidas, mostrando as propriedades, métodos e tipos esperados em cada contexto.

- **Documentação implícita:**
  O próprio código já mostra o que cada função/componente espera, tornando mais fácil para outros devs entenderem.

- **Melhor integração com IDEs:**
  A maioria das IDEs oferece navegação de código, sugestões e highlight aprimorados.

- **Facilita manutenção de projetos grandes:**
  Equipes grandes conseguem manter padronização e evitar inconsistências.

Tipagem de Props em Componentes

```tsx
interface CardProps {
  titulo: string;
  valor: number;
  emDestaque?: boolean; // opcional
}

export function Card({ titulo, valor, emDestaque = false }:
CardProps) {
  return (
    <div style={{ fontWeight: emDestaque ? "bold" : "normal" }}>
      <h2>{titulo}</h2>
      <p>R$ {valor.toFixed(2)}</p>
    </div>
  );
}
```

Tipagem em State e Eventos

```tsx
import { useState, ChangeEvent } from "react";

export function ExemploInput() {
  const [idade, setIdade] = useState<number>(0);

  function handleChange(e: ChangeEvent<HTMLInputElement>) {
    setIdade(Number(e.target.value));
  }

  return (
    <input type="number" value={idade} onChange={handleChange} />
  );
}
```

Tipagem em Funções e Arrays

```
function soma(a: number, b: number): number {
  return a + b;
}

const nomes: string[] = ["Ana", "Bruno", "Carlos"];
```

Tipando objetos complexos

```
type Produto = {
  id: number;
  nome: string;
  preco: number;
  categoria?: string;
};

const produtos: Produto[] = [
  { id: 1, nome: "Camisa", preco: 59.9 },
  { id: 2, nome: "Tênis", preco: 199.9, categoria: "Calçados" }
];
```

Tipagem em Hooks Customizados

```
function useToggle(inicial: boolean): [boolean, () => void] {
  const [ativo, setAtivo] = useState<boolean>(inicial);
  const toggle = () => setAtivo(a => !a);
  return [ativo, toggle];
}
```

Principais vantagens

- Detecção antecipada de erros de tipo (menos bugs em produção).
- Melhor experiência de desenvolvimento: autocompletar, dicas de tipo e navegação facilitada no código.
- Documentação automática de interfaces e componentes.

Principais conceitos ao usar React com TypeScript

- Tipagem de Props: Defina as propriedades dos componentes usando interfaces ou types.
- Tipagem de Estado e Refs: Deixe explícito o tipo dos estados (**useState**) e referências (**useRef**).
- Tipagem em funções, hooks customizados e contextos.

Dicas de configuração – React + TypeScript

1. Criação do projeto

O jeito mais simples e rápido é usar o Vite ou o Create React App (CRA):

Vite (recomendado para projetos modernos):

```
npm create vite@latest nome-do-projeto -- --template react-ts
cd nome-do-projeto
npm install
npm run dev
```

2. Create React App (alternativa tradicional):

```
npm create vite@latest nome-do-projeto -- --template react-ts
cd nome-do-projeto
npm install
npm run dev
```

2. Estrutura básica do tsconfig.json

Este arquivo controla as regras do compilador TypeScript. O template Vite/CRA já cria um bom ponto de partida:

```
{
  "compilerOptions": {
    "target": "ESNext",
    "jsx": "react-jsx",
    "module": "ESNext",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

Principais dependências:

- `typescript`
- `@types/react`
- `@types/react-dom`

Integração com o Projeto

Como adaptar componentes:

- Renomeie de `.js` para `.tsx` e comece a declarar tipos.
- Tipar `props`, `state`, retornos de funções, hooks, contextos.
- Adote interfaces e types para descrever dados, eventos, objetos, etc.

Exemplo prático de migração de props:

Antes (JavaScript):

```
function Produto({ nome, preco }) {
  return <div>{nome} - {preco}</div>
}
```

Depois (TypeScript):

```
type ProdutoProps = { nome: string; preco: number };
function Produto({ nome, preco }: ProdutoProps) {
  return <div>{nome} - {preco}</div>
}
```

3. Instalando tipos adicionais

Quando instalar bibliotecas externas (por exemplo, styled-components, react-router-dom), instale também seus tipos:

```
npm install @types/react-router-dom @types/styled-components --save-dev
```

Deploy em Projetos React

Build de Produção

O que é o build de produção?

- É a etapa onde o código fonte do seu projeto é “empacotado” e otimizado para ser publicado em servidores (internet ou intranet).
- Remove arquivos e instruções de desenvolvimento, minifica e agrupa arquivos JS/CSS, e garante melhor desempenho e segurança para o usuário final.

Como gerar o build de produção?

Se estiver usando Create React App:

```
npm run build
```

Isso gera a pasta /build (ou /dist em outras ferramentas).

O conteúdo desta pasta é estático: arquivos HTML, JS, CSS, imagens otimizadas, prontos para qualquer servidor web.

O que acontece durante o build?

- **Minificação:** reduz o tamanho dos arquivos removendo espaços, comentários e renomeando variáveis.
- **Tree-shaking**: remove código não utilizado dos pacotes JS.
- **Code splitting**: separa seu código em múltiplos arquivos menores para “carregar sob demanda”.
- **Hash nos arquivos**: para facilitar cache no navegador e evitar servir código antigo ao usuário.

Como fazer deploy do build de produção?

O conteúdo da pasta /build pode ser publicado em:

- **Servidores estáticos**:
  (Vercel, Netlify, Firebase Hosting, GitHub Pages, AWS S3 + CloudFront, etc.)
- **Servidores próprios**:
  (Nginx, Apache, etc.)

Exemplo com Vercel/Netlify:

- Basta conectar o repositório, selecionar o comando de build (npm run build) e a pasta de saída (build).
- O deploy é automático a cada novo push no GitHub/Gitlab.

Como Melhorar o Build para um Código Menor e Mais Enxuto

Por que otimizar?

- Menor tempo de carregamento para o usuário.
- Menos consumo de banda no servidor e no cliente
- Melhor desempenho SEO e ranking em buscadores.
- Menor risco de erros por excesso de código não utilizado.

Boas Práticas e Técnicas

1. Remova imports não utilizados

- O Tree-shaking funciona melhor se você evitar import * as ... ou imports desnecessários.

2. Utilize Code Splitting

- Separe páginas, rotas ou componentes grandes para carregarem “on demand”.
- Use React.lazy e Suspense:

```
const Dashboard = React.lazy(() => import('./Dashboard'));
<Suspense fallback={<div>Carregando...</div>}>
  <Dashboard />
</Suspense>
```

No React Router:

```
const Detalhe = React.lazy(() => import('./pages/Detalhe'));
// dentro do <Routes>:
<Route path="/detalhe" element={<Detalhe />} />
```

Boas Práticas e Técnicas

Use bibliotecas modernas e modulares

- Prefira libs como date-fns ao invés de moment.js, pois são menores.
- Sempre que possível, importe apenas o que for usar:

```
import isAfter from 'date-fns/isAfter';
// Evite: import * as dateFns from 'date-fns';
```

Otimize imagens e fontes

- Use formatos modernos como .webp e .avif.
- Importe fontes com peso/estilo realmente necessários.
- Prefira carregamento assíncrono/lazy para imagens grandes.

Ative Gzip ou Brotli no servidor

- Compacta ainda mais os arquivos servidos (Nginx, Vercel, Netlify ativam isso por padrão).

Remova console.log e código morto

- Use plugins/bibliotecas ou scripts que eliminam logs e funções não usadas.

Checklist para um build enxuto:

- Está usando code splitting (React.lazy/Suspense)?
- Só importa funções que usa nas libs?
- Imagens otimizadas e compactadas?
- Removeu todos os logs/debugs do código?
- Conferiu dependências no package.json (remova libs não usadas)?
- Rodou ferramenta de análise de bundle?
- Usou variáveis de ambiente para esconder dados sensíveis no build?