

---

# **REACT EM PROFUNDIDADE ANOTACÕES E REFLEXÕES SOBRE ENGENHARIA DE INTERFACES**

Por Caio Rossi



---

# ÍNDICE

## Módulo 0 - Intenção, Método e Preparação Conceitual

Como este material foi construído (processo, constância e prática deliberada).....	05
Por que este material existe e qual problema ele resolve.....	06
Metodologia de Estudo e Estrutura dos Módulos.....	07
Preparação para os módulos avançados.....	08

## Módulo 1 - Filosofia e Paradigma do React

Paradigma declarativo vs imperativo.....	10
React como UI derivada de estado.....	11
Composição de componentes e a ideia de árvore de renderização.....	14
O papel da previsibilidade e da pureza de funções.....	16
Abstração entre dados → estado → interface.....	18
O que diferencia o React de frameworks MVVM/MVC tradicionais.....	20

## Módulo 2 - O motor do React (Virtual DOM e Fiber)

O que é o virtual DOM e por que ele existe.....	25
Reconciliação: como o React compara árvores.....	26
Diffing algorithm e heurísticas internas.....	27
React Fiber: scheduler, prioridade e interrupção de tarefas.....	29
Tempo de renderização, commit phase e mutation phase.....	30
Como o React garante fluidez sob carga (Concurrent mode).....	31
Batching de atualizações e event loop.....	33

## Módulo 3 - Hooks em nível conceitual

Por que os hooks foram criados (substituição de classes).....	35
O ciclo de vida funcional: render → commit → cleanup.....	37
Regras dos hooks (ordem, pureza e chamadas).....	38
useState: closures e estado persistente entre renderizações.....	40
useEffect: sincronização com o mundo externo.....	42
useRef: identidade e persistência fora do ciclo de render.....	44
useMemo e useCallback: memoização e reatividade controlada.....	46
useReducer e o padrão de isolamento de lógica.....	48
Hooks customizados: composição e reutilização de comportamento.....	51
O problema do stale state e as closures em React.....	53

---

# ÍNDICE

## Módulo 4 - Estado, dados e arquitetura

Tipos de estado: local, derivado, global e remoto;.....	56
Fonte de verdade e estado derivado.....	58
Sincronização de estados entre componentes.....	60
Context API: conceito, escopo e custo de renderização.....	61
Patterns de gerenciamento (lifting state up, prop drilling, context splitting).....	63
Gerenciamento externo: Redux Toolkit, Zustand, React Query (conceito, não uso).....	64
Cache, sincronização e invalidação de dados.....	66
“Server state” vs “UI state”.....	67
Estratégias para evitar re-renderizações desnecessárias”.....	69

## Módulo 5 - Renderização e performance

Ciclo de renderização e commit detalhado.....	72
Identificação de causas de re-render.....	73
Reatividade granular e isolamento de componentes.....	74
Suspense e streaming de dados (React 18+).....	75
Lazy loading e divisão de código.....	76
Concurrent rendering: scheduling, interrupção e prioridade.....	78
Interação com Web Vitals e tempo de pintura.....	79

## Módulo 6 - Composição e padrões de arquitetura

Padrão de composição (props.children, render props, compound components).....	82
Inversão de controle e abstração progressiva.....	85
Componentes controlados vs não controlados.....	88
Patterns de isolamento: container/presentational, smart/dumb.....	90
Prop drilling e patterns alternativos.....	92
Patterns de extensibilidade (slot pattern, HOCs conceitualmente).....	94
Princípios de design componível.....	97

---

# ÍNDICE

## Módulo 7 - React moderno e Server Components

Server Components: o que são e por que surgiram.....	99
SSR, SSG, CSR e hidratação.....	101
Transmissão de dados e boundaries entre client/server.....	102
Suspense for Data Fetching e streaming.....	104
Limitações e padrões emergentes (Next.js 14/15, React 19).....	105
O futuro da renderização no React.....	107

## Módulo 8 - Filosofia e Engenharia Frontend

Trade-offs: abstração vs clareza.....	110
Decisões guiadas por intenção, não por ferramenta.....	112
Evolução de código e consistência em equipe.....	113
Design orientado a previsibilidade e legibilidade.....	114
Pensamento sistêmico no frontend.....	115
React como ferramenta, não fim.....	116
Como pensar como um engenheiro de interface.....	117

## Perguntas

Módulo 1.....	119
Módulo 2.....	121
Módulo 3.....	124
Módulo 4.....	128
Módulo 5.....	131
Módulo 6.....	134
Módulo 7.....	137
Módulo 8.....	140

---

# **MÓDULO 0**

## **INTENÇÃO, MÉTODO E PREPARAÇÃO**

### **CONCEITUAL**

**Contextualizando o leitor sobre o propósito do  
material e estabelecendo as premissas  
conceituais necessárias para acompanhar os  
módulos avançados**

# COMO ESTE MATERIAL FOI CONSTRUÍDO (PROCESSO, CONSTÂNCIA E PRÁTICA DELIBERADA)

Este projeto começou como um exercício de aprofundamento pessoal no ecossistema React. A intenção inicial não era produzir este E-book, mas compreender a biblioteca em um nível conceitual e teórico. A partir disso, os módulos foram planejados e serviram como um mapa para estruturar o estudo.

O processo consistiu em investigar um tema por dia e produzir anotações estruturadas, sem formalidade acadêmica, mas com rigor conceitual. Cada sessão de estudo envolvia leitura e, principalmente, escrita: transformar conceitos em explicações, revisar, reescrever e reorganizar. Este hábito contínuo de escrever enquanto estudava foi o que permitiu consolidar o entendimento e construir uma progressão coerente entre os capítulos.

Houveram dias improdutivos, travas conceituais e momentos em que o estudo parecia não avançar. Ainda assim, a constância prevaleceu, sempre retornando ao propósito maior: criar um material que sintetizasse o que foi aprendido e pudesse ser compartilhado com outras pessoas. A visão do produto final manteve o projeto em movimento, mesmo quando o progresso diário era pequeno.

Por se tratar de um material estruturado, organizado e escrito por uma única pessoa, é importante reconhecer que imprecisões, simplificações excessivas ou interpretações limitadas podem existir. Este não é um trabalho de revisão coletiva nem um documento oficial do React, mas um registro honesto de estudo e compreensão em evolução. Sempre que possível, conceitos foram checados, revisitados e refinados, mas este material deve ser lido como um conjunto de anotações técnicas consolidadas, não como uma fonte definitiva ou incontestável.

A mensagem fundamental desta seção é simples: **aprofundar-se tecnicamente não é um produto de talento, mas de método**. Entendimento profundo é consequência de prática deliberada, disciplina, revisões sucessivas e disposição para revisitar conceitos até que se tornem claros. Este material é fruto desse processo, não de inspiração, mas de **constância**. Ele demonstra que qualquer pessoa, com estrutura e intenção, pode produzir conhecimento significativo e construir entendimento sólido camada por camada.

# POR QUE ESSE MATERIAL EXISTE E QUAL PROBLEMA ELE RESOLVE

Embora eu utilizasse React há anos, sentia uma lacuna persistente: conseguia aplicar a biblioteca, criar interfaces e estruturar aplicações, mas não comprehendia plenamente por que React funcionava como funcionava, nem como suas abstrações operavam internamente. Faltava entendimento conceitual para justificar decisões, avaliar trade-offs e enxergar o design da biblioteca de forma sistêmica. Essa ausência de base teórica não é incomum, ela é consequência direta da maneira como React costuma ser ensinado.

Grande parte do material disponível foca em tutoriais, padrões de uso ou exemplos práticos. Ensina-se o que fazer, mas raramente **por que fazer**. Explica-se a sintaxe dos hooks, mas não seu fundamento. Demonstra-se reatividade, mas não o modelo declarativo que a sustenta. Mostra-se o Virtual DOM, mas não o runtime, o scheduler ou o processo de reconciliação. Essa abordagem gera profissionais que dominam APIs, mas **não dominam o sistema**.

Este e-book surge justamente para preencher essa lacuna.

Ele não é um curso, nem um guia formal; é uma estrutura modular de anotações de estudo, curtas e diretas, escritas no ritmo do aprofundamento pessoal. A informalidade é intencional: trata-se de registrar conceitos de forma acessível, sem comprometer a precisão técnica. Cada módulo foi pensado para construir entendimento cumulativo, permitindo que o leitor desenvolva uma base sólida sobre o que React realmente faz, e por que faz dessa maneira.

A iniciativa de transformar esse material em e-book só surgiu quando, com os módulos quase concluídos, ficou evidente que o conteúdo poderia beneficiar outras pessoas que desejam ir além do uso cotidiano da biblioteca. A proposta é oferecer um caminho claro para quem quer deixar de simplesmente **usar React** e passar a **entender React**.

No fim, o problema que este material resolve é justamente o que motivou sua criação: a carência de uma abordagem concisa, progressiva e conceitual que explique o React como um sistema completo, e não apenas como uma coleção de APIs.

# METODOLOGIA DE ESTUDO E ESTRUTURA DOS MÓDULOS

Este e-book foi estruturado em módulos independentes, organizados de maneira cumulativa, mas com textos curtos e de leitura rápida. A intenção é que cada tópico possa ser consumido em aproximadamente cinco minutos, tempo suficiente para ler cerca de 500 palavras e refletir sobre a ideia central daquele trecho. A proposta não é estudar React em longas sessões, mas avançar **um conceito por dia**, permitindo que o conhecimento seja absorvido de forma **progressiva e sustentável**.

Cada módulo aborda um eixo conceitual específico e cada tópico dentro dele foi escrito como uma unidade autônoma de estudo. Isso permite ao leitor escolher entre seguir a sequência completa ou utilizar o material para revisitar conceitos específicos conforme necessidade.

O foco deste e-book está em compreender fundamentos. Por isso, a leitura sugerida consiste em:

- Ler um tópico por dia
- Refletir sobre o conceito apresentado, relacionando-o com sua experiência prévia.
- Repetir esse ciclo diariamente, criando um ritmo de estudo consistente.

Essa abordagem incremental funciona porque React é construído a partir de camadas conceituais: entender uma prepara naturalmente o terreno para a próxima. Em vez de tentar dominar tudo de uma vez, o leitor avançará gradualmente, assimilando um bloco de entendimento por vez.

O objetivo não é apenas consumir conteúdo, mas **construir clareza mental** sobre como React pensa, como decide e como opera internamente. Cada tópico foi escrito para se manter curto o suficiente para caber no dia, e denso o suficiente para transformar a forma como o leitor enxerga a biblioteca.

# PREPARAÇÃO PARA OS MÓDULOS MAIS AVANÇADOS

Os módulos seguintes abordam conceitos fundamentais da arquitetura do React, desde seu paradigma declarativo até os mecanismos internos do Fiber, hooks, modelo de estado, reconciliação, performance e server components. Para aproveitar esse conteúdo em profundidade, o leitor precisa assumir uma postura ativa de estudo: ler, refletir e conectar cada tópico ao que já comprehende sobre JavaScript e interfaces.

A preparação necessária não é técnica, mas conceitual. O leitor deve avançar para os módulos seguintes com três expectativas claras:

- **React será estudado como um sistema, não como uma coleção de APIs.**

Os capítulos abordam o funcionamento interno do runtime, os princípios de design que guiam a biblioteca e as implicações arquiteturais por trás de cada decisão. A compreensão desses fundamentos é o que torna possível raciocinar com clareza sobre hooks, estado, re-renderizações e performance.

- **Os conceitos se acumulam camada por camada.**

A leitura dos módulos foi estruturada para criar um efeito cumulativo: a filosofia do React prepara o terreno para entender seu motor; o motor fundamenta os hooks; os hooks fundamentam o estado; o estado fundamenta a arquitetura e a performance; todos convergem no entendimento do React moderno. O leitor deve avançar esperando que cada módulo se apoie no anterior.

- **A assimilação será mais valiosa que a velocidade.**

Embora cada tópico seja curto, ele condensa uma ideia-chave. O objetivo não é “terminar o livro”, mas permitir que cada conceito reposicione a maneira como o leitor enxerga a biblioteca. Avançar aos módulos avançados significa estar disposto a entender como o React pensa, e não apenas como ele é utilizado no dia a dia.

Ao completar o Módulo 0, o leitor estará preparado para iniciar a jornada conceitual que permeia o restante do e-book. Os próximos capítulos expandem progressivamente o entendimento sobre o React e revelam o sistema que opera sob suas abstrações. Esse é o fundamento necessário para navegar pela biblioteca com autonomia e maturidade técnica.

---

# **MÓDULO 1**

# **FILOSOFIA E PARADIGMA DO REACT**

Entendendo o porquê do React existir e o  
que ele representa no contexto da  
engenharia de interfaces.

# PARADIGMA DECLARATIVO X IMPERATIVO

No paradigma imperativo, escrevemos passo a passo o que deve acontecer (Algoritmamente)

Com um exemplo do mundo real:

Para o preparo de um café:

"Pegue a chaleira, encha de água, aqueça até ferver, coloque o pó, coe, despeje na xícara"

Você descreve a ação necessária

Exemplo em Javascript:

```
const button = document.createElement('button');
button.textContent = 'Clique aqui';
button.addEventListener('click', () => {
  alert('Olá!');
});
document.body.appendChild(button);
```

Você diz explicitamente como criar, inserir e reagir ao botão

No paradigma declarativo, você descreve o resultado desejado e o sistema decide como chegar lá

Voltando ao exemplo do café: "Quero um café pronto":

Você apenas declara a intenção, quem executa sabe o processo interno em React:

```
function App() {
  return <button onClick={() => alert('Olá')}>Clique aqui </button>
}
```

Você apenas declara: Quero um botão que mostre um alerta ao clicar. O react cuida do como: quando criar, como colocar, como atualizar.

Um ponto importante é observar que o paradigma declarativo só é viável quando a linguagem ou framework fornece uma camada que abstrai o imperativo  
Por exemplo, em C, você precisaria controlar tudo: alocação de memória, fluxo, lógica de renderização, é um ambiente puramente imperativo

Em react, a biblioteca implementa toda essa lógica invisível.  
O declarativo é, portanto, um nível de abstração acima do imperativo, ela se apoia nele

Quando programamos em um modelo declarativo, precisamos pensar como o navegador:

- Criar elementos
- Inserir nós
- Remover o que mudou
- Atualizar somente as partes específicas

Com a declaração imperativa do react, todas essas decisões são feitas automaticamente via dom Diffing por baixo dos panos.

# REACT COMO UI DERIVADA DE ESTADO

O conceito central é que, no react, a UI não é algo que você manipula diretamente, ela é uma função do estado:

- O estado é a fonte de verdade
- A interface é uma projeção desse estado, calculada automaticamente pelo React
- Quando o estado muda, o React recalcula a UI - Você não mexe no DOM diretamente

A interface que você vê na tela é completamente determinada pelo estado atual da aplicação

Em termos matemáticos:

**UI = f(state)**

Como pensar:

Você não toca na tela diretamente, você modifica o estado (os dados que representam a situação atual do app)

O react lê esse novo estado e calcula uma nova UI a cada alteração

O react aplica as mudanças de forma inteligente, só altera os elementos que realmente mudaram, sem redesenhar tudo do 0, esse processo é chamado de reconciliação ou DOM diffing

O resultado é que o usuário vê a tela atualizada, mas você não precisou se preocupar com os detalhes de atualização do DOM, o react fez.

Em uma UI sem derivar do estado (imperativa), imagine que você cria um elemento `<ul>` manualmente, pra cada tarefa no array, você cria um `<li>` e adiciona ao `<ul>`, quando uma tarefa é adicionada, você precisa atualizar o DOM manualmente: criar `<li>`, inserir na lista, remover se necessário, se o usuário marcar uma tarefa como concluída, você precisa localizar o `<li>` específico e mudar a sua aparência diretamente.

Isso traz alguns problemas.

Se a UI não deriva de uma função do estado, ela tem vida própria e precisa ser sincronizada constantemente com os novos dados.

Se você esquecer de atualizar um elemento, a interface fica fora de sincronia com o estado real

Se a UI não deriva de uma função do estado, ela tem vida própria e precisa ser sincronizada constantemente com os novos dados.

Se você esquecer de atualizar um elemento, a interface fica fora de sincronia com o estado real

Em uma UI derivada de estado, você mantém um estado tasks com todas as tarefas, a UI é sempre calculada a partir desse estado, pra atualizar uma tarefa, você apenas atualiza o estado, o react automaticamente recalcula a UI e atualiza apenas o que mudou no dom

Problemas de manipular diretamente o dom:

1. Inconsistência entre estado e UI

Quando você atualiza o dom manualmente, não há nenhuma "fonte de verdade central"

Outros elementos ou funções que dependem desse valor podem ficar desatualizados, gerando inconsistências

# REACT COMO UI DERIVADA DE ESTADO

Exemplo:

```
const h1 = document.getElementById('contador');
let valor = Number(h1.textContent);
const dobro = valor * 2;
console.log(dobro);
```

Se você esquecer de atualizar o h1 corretamente, dobro vai refletir um valor antigo, quebrando a lógica do app

## 2. Repetição de lógica

Cada elemento que depende desse valor precisa ser atualizado manualmente  
Se você tiver 10 lugares mostrando o contador, precisará garantir que todos mudem juntos

Mais componentes = Mais chances de erro

## 3. Problemas de otimização

Manipular o DOM manualmente pode exigir reescrever ou redesenhar partes inteiras da UI, mesmo quando só uma pequena parte mudou.  
Em apps grandes, isso degrada a performance

## 4. Dificuldade de teste

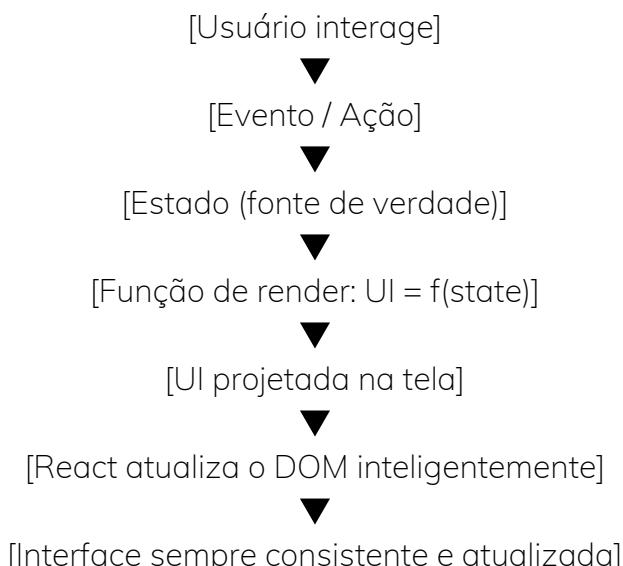
Funções que manipulam o DOM diretamente são difíceis de testar isoladamente  
Você precisa recriar o DOM em ambiente de teste, tornando o processo mais pesado

Como a UI derivada de estado resolve isso?

O estado é a fonte da verdade (A fonte de verdade é o lugar único onde os dados realmente existem e de onde todas as partes do sistema obtêm informação confiável), é como uma variável central que guarda todos os dados corretos e atualizados da aplicação

No contexto do react, todos os dados que definem como a interface deve aparecer estão guardados no estado

A UI não armazena valores por conta própria, ela apenas projeta o estado na tela



# REACT COMO UI DERIVADA DE ESTADO

Problemas que o imperativo teria:

- UI tem vida própria → risco de inconsistência
- Repetição de lógica → chance de erro
- Atualizações manuais → performance ruim
- Difícil de testar

Benefícios do declarativo + estado:

- Fonte de verdade central
- UI sempre consistente
- Atualização automática e eficiente
- Facilidade de teste

Tipos de estado em react:

## 1. Estado local

Guardado dentro de um componente específico

Controla dados que só interessam a esse componente

Ex: um contador dentro de um card

**const [count, setCount] = React.useState(0)**

Quando usar? Se o dado não precisa ser compartilhado com outros componentes

## 2. Estado derivado

Calculado a partir de outros estados

Não precisa ser armazenado separadamente, pode ser gerado sob demanda

Ex: filtrar tarefas concluídas a partir de uma lista de tarefas

const tarefasConcluidas = tasks.filter(task => task.done)

Quando usar? Para evitar duplicação de dados e inconsistências

## 3. Estado global

Compartilhado entre vários componentes da aplicação

Necessário quando múltiplos componentes precisam acessar ou atualizar o mesmo dado

Ex: Usuário logado, tema do app, carrinho de compras

Geralmente gerenciado com Context API, Redux ou ferramentas similares

Estado local:

Só interessa ao componente que o possui

Ex: contador de cliques, input temporário em um formulário

Estado derivado:

Pode ser calculado a partir de outros estados, não precisa ser armazenado separadamente

Ex: lista de tarefas concluídas (**task.filter(task => task.done)**), total de itens no carrinho

Estado global:

Necessário quando vários componentes precisam do mesmo dado

Ex: Usuário logado, tema de aplicação, carrinho compartilhado em múltiplos componentes

# COMPOSIÇÃO DE COMPONENTES E ÁRVORE DE RENDERIZAÇÃO

No react, a UI é construída a partir de componentes, que são blocos reutilizáveis

Cada componente pode ter:

Estado local

Props

Funções internas

Componentes são organizados em hierarquias

Cada componente recebe dados via props e pode manter o estado local

A renderização é recursiva, react começa do topo (raiz), renderiza cada filho, que pode renderizar seus próprios filhos, formando a árvore completa da UI

Composição vs herança

Herança: componente filho herda comportamento de outro (como classes em OOP)

Composição: componente é montado a partir de outros componentes

Existe um ponto a se considerar: componentes profundos ou grandes podem custar mais tempo para recalcular. Sempre que o estado muda o react precisa re-renderizar os componentes afetados

No react, é uma boa prática dividir as responsabilidades entre componentes para manter o código todo organizado, previsível e reutilizável.

O componente pai:

Gerencia o estado e lógica quando necessário

Passa dados e funções para os filhos via props

O componente filho:

Focados em UI/apresentação

Recebem dados e callbacks do pai via props

Não alteram diretamente o estado do pai

Exemplo só com UI :

```
function Home() {
  return (
    <div>
      <Hero />
      <Projects />
      <About />
      <Contact />
    </div>
  );
}

function Hero() { return <section>Seção Hero</section>; }
function Projects() { return <section>Seção Projects</section>; }
function About() { return <section>Seção About</section>; }
function Contact() { return <section>Seção Contact</section>; }
```

# COMPOSIÇÃO DE COMPONENTES E ÁRVORE DE RENDERIZAÇÃO

Exemplo com props e estado (pai -> filho):

```
function Home() {
  const [ projetos, setProjetos] = React.useState([
    { id: 1, nome: "Projeto A" },
    { id: 2, nome: "Projeto B" }
  ]);

  return (
    <div>
      <Hero />
      <Projects lista={projetos} />
    </div>
  );
}

function Projects({ lista }) {
  return (
    <ul>
      {lista.map(p => (
        <li key={p.id}>{p.nome}</li>
      ))}
    </ul>
  );
}
```

Resumo:

- Componentes pequenos e focados são mais fáceis de testar e manter
- Filhos devem ser reativos e dependentes de props, sem mexer no estado diretamente
- Pai centraliza o estado e a lógica, mantendo a fonte de verdade
- Mesmo em apps simples, essa separação ajuda a preparar a aplicação para crescer sem se tornar confusa

# O PAPEL DA PREVISIBILIDADE E DA PUREZA DE FUNÇÕES

Uma função pura é aquela que depende de seus argumentos (sem acessar variáveis externas mutáveis)

Não causa efeitos colaterais (não modifica nada fora dela)

Retorna sempre o mesmo resultado para a mesma entrada

Ex de função pura:

```
function somar(a, b) {  
  return a + b;  
}
```

Ex de função impura:

```
let total = 0;
```

```
function somar(a) {  
  total += a;  
  return total;  
}
```

Aqui, o resultado depende do valor anterior de total, não apenas do argumento.

Por que o react exige que a função de renderização seja pura?

Quando o react chama o componente (por exemplo App()), ele espera que:

- Dada a mesma entrada (props e estado)
- A função retorne a mesma UI (jsx)

Isso permite ao react prever exatamente o que a tela deve mostrar sem depender de estados escondidos ou efeitos colaterais

Se a função de renderização não for pura (por exemplo, se ela modificar variáveis globais, alterar o DOM diretamente ou buscar dados ali dentro), o React não consegue garantir que o resultado será consistente, o que quebra sua capacidade de reconciliação

No javascript puro, quase tudo é efeito colateral.

Efeito colateral (side effect) é qualquer impacto que uma função causa fora do seu próprio escopo, se uma função muda algo que não é apenas o seu retorno, ela está produzindo um efeito colateral

Ex:

```
let contador = 0;
```

```
function incrementar() {  
  contador++;  
  console.log("Contador:", contador);  
}
```

Mas se no React precisamos de funções puras por padrão, onde colocar os efeitos colaterais?

O hook useEffect existe justamente para isso

# O PAPEL DA PREVISIBILIDADE E DA PUREZA DE FUNÇÕES

O papel do useEffect é resolver o problema do react exigir que a renderização seja pura: não pode modificar o dom diretamente, não pode buscar dados, não pode usar temporizadores ou logs que alterem o ambiente. Mas aplicações reais precisam interagir com o mundo externo, por exemplo: buscar os dados de uma api, armazenar algo no localStorage, adicionar ou remover event listeners.

O use effect permite executar efeitos colaterais de forma controlada, após o react renderizar a interface, ou seja, ele não interfere no cálculo da UI e ocorre apenas quando necessário, mais ou menos em uma ideia de:  
"Deixe a renderização ser pura. Quando ela terminar, execute esse efeito com base no estado atual"

Mas o que entra no useEffect?

- Requisições assíncronas como buscar dados de apis, carregar arquivos, etc.
- Manipulação do dom fora do Jsx
- Event listeners
- Temporizadores
- Sincronização com armazenamento externo
- Tudo que interage com o mundo externo ou que não pode ser determinístico pela renderização

# ABSTRAÇÃO ENTRE DADOS, ESTADO, INTERFACE

Abstração significa esconder a complexidade de um processo, oferecendo uma interface mais simples para interagir com ele.

Quando você dirige um carro, você vira o volante e o carro faz a curva. Você não precisa saber como o sistema hidráulico, os sensores e as engrenagens do eixo funcionam. O sistema de direção abstrai toda essa complexidade

O react funciona como uma camada de abstração entre:

- Os dados reais
- O estado interno da aplicação
- A interface renderizada

Os dados brutos só existem no ambiente externo, ainda não são controlados pelo react

O estado representa os dados dentro do react, é a versão controlada e filtrada dos dados recebidos

A interface é a projeção do estado, ela não guarda dados, ela apenas mostra o que o estado representa

O fluxo completo é representado por:

1. Dado entra (API, evento, input)
2. Atualiza o estado
3. React recalcula a interface
4. Usuário vê a nova UI

O react é uma abstração sobre o DOM.

Você não atualiza o DOM quando um dado muda, você atualiza o estado, e o react cuida de atualizar a tela, ao invés de dizer COMO mudar a tela, você só diz o que QUER VER com base nos dados

No JavaScript puro, manipulamos o DOM diretamente com métodos como `innerHTML`, controlando cada detalhe da mudança na tela.

No React, descrevemos o que queremos ver e deixamos que o React manipule o DOM internamente, de forma declarativa e otimizada.

O React possui um mecanismo próprio de atualização (via Virtual DOM), que funciona como um “innerHTML controlado”, mas o desenvolvedor não interage com ele diretamente.

## Nem toda abstração é benéfica.

Abstrair demais pode dificultar a leitura e o rastreamento dos dados.  
O ideal é buscar um equilíbrio

Abstrair significa esconder complexidade, mas esconder não é eliminar. Ao criar uma abstração, está simplificando o uso, mas afastando o programador da lógica real.

O react segue o princípio de one-way data flow, o que significa que os dados fluem do componente pai para o componente filho via props, toda alteração de estado ocorre em um único ponto de verdade- estado do componente que o controla

# ABSTRAÇÃO ENTRE DADOS, ESTADO, INTERFACE

A abstração garante consistência e previsibilidade sem a necessidade de controlar manualmente todos os estados e atualizações do DOM.

No React, a abstração serve para simplificar a interação com a interface e os dados, escondendo a complexidade do DOM e das operações internas. Isso significa que você não precisa manipular cada elemento manualmente; basta atualizar o estado e o React cuida do resto.

O fluxo unidirecional de dados (one-way data flow) garante que as informações sigam sempre de um componente pai para seus filhos via props. O estado que reside no componente pai é a fonte de verdade, ou seja, o ponto central onde os dados são armazenados e controlados. Essa abordagem evita inconsistências que surgiriam se cada componente tentasse atualizar a UI de forma independente.

A UI derivada do estado funciona como uma projeção: ela não mantém dados próprios, apenas reflete o estado atual da aplicação. Essa separação entre dados → estado → interface permite:

- Consistência: a interface sempre corresponde ao estado real; não há “espelhos” desatualizados no DOM.
- Previsibilidade: dado um estado conhecido, a UI resultante é sempre a mesma.
- Escalabilidade: mesmo aplicações grandes podem ser mantidas de forma organizada, porque cada alteração de estado atualiza a UI automaticamente.

Em termos práticos, se você manipular o DOM diretamente em JavaScript puro, precisaria recalcular manualmente cada item quando os dados mudassem. Com React, essa responsabilidade é abstraída: o Virtual DOM calcula diferenças e aplica apenas as mudanças necessárias, garantindo eficiência e evitando erros comuns de sincronização.

Resumo dos pontos-chave:

1. Abstração simplifica o uso sem eliminar a lógica subjacente.
2. Fluxo unidirecional centraliza as alterações de estado, evitando inconsistências.
3. UI derivada do estado garante previsibilidade e consistência.
4. Virtual DOM otimiza atualizações, minimizando operações custosas no DOM real.
5. Nem toda abstração é benéfica: o equilíbrio entre clareza e ocultação de complexidade é essencial.

# O QUE DIFERENÇIA O REACT DE FRAMEWORKS MVVM-MVC TRADICIONAIS

## MVC – Separação manual e explícita

No modelo MVC (Model–View–Controller), cada parte do sistema tem uma responsabilidade clara:

- Model: guarda os dados — por exemplo, a lista completa de produtos.
- View: exibe os dados na tela — a lista filtrada visível ao usuário.
- Controller: recebe as ações do usuário (como digitar no campo de busca), processa a lógica e decide o que mostrar.

Em um filtro de busca, quando o usuário digita algo, o Controller escuta o evento de digitação, filtra os dados do Model e manda para a View renderizar o novo resultado. Ou seja, a atualização da tela acontece porque o Controller mandou, de forma imperativa e controlada.

Esse padrão é mais verboso, mas dá clareza sobre “quem faz o quê”.

## MVVM – Atualização automática (two-way data binding)

No MVVM (Model–View–ViewModel), a ideia é que a View e o Model fiquem sincronizados automaticamente.

A camada do ViewModel funciona como um “espelho inteligente” dos dados: se o usuário altera algo na interface (como digitar uma letra no campo de busca), o modelo é atualizado automaticamente, e vice-versa.

Assim, a View se atualiza sem que o programador precise escrever a lógica manualmente.

No filtro de busca, isso significa que a cada letra digitada, a lista de resultados já é filtrada em tempo real, porque a view está “vinculada” ao dado.

Esse comportamento é comum em frameworks como Angular e Vue, com o chamado two-way data binding ('v-model' ou 'ngModel'), mas tem o custo de ser mais difícil rastrear o fluxo, às vezes não fica claro de onde veio a mudança no dado.

## React – UI como função do estado (one-way data flow)

O React adota um modelo diferente: a UI é apenas uma função pura do estado.

Não existe uma ligação automática entre input e dados.

Quando o usuário digita algo no campo de busca, você decide manualmente quando e como atualizar o estado — usando, por exemplo, useState.

O React então recalcula a interface com base no novo estado.

Ou seja, o fluxo é sempre unidirecional:

o usuário gera um evento → o estado é atualizado → a UI se atualiza com base nesse estado.

# O QUE DIFERENÇIA O REACT DE FRAMEWORKS MVVM-MVC TRADICIONAIS

No filtro de busca, o input chama `setBusca(e.target.value)` e a lista renderiza `{produtos.filter(p => p.includes(busca))}`.

Não há vínculo automático, mas há previsibilidade total, o estado é a única fonte de verdade, e a UI é derivada dele.

Antes do react, a maioria dos frameworks seguia o padrão MVC (model-view-controller) ou MVVM (model-view-viewmodel)

## MVC

O padrão MVC separa uma aplicação em três camadas principais, cada uma com uma responsabilidade específica

1. Model (modelo) - É a fonte dos dados e das regras de negócio, ex: buscar produtos de uma api, salvar no banco, validar campos
2. View (visão) - É a interface que o usuário vê e interage, ex: o HTML que mostra os produtos na tela
3. Controller (controlador) - Faz a ponte entre o model e a view - Ele recebe eventos da interface (como um clique), altera o model e atualiza o view

exemplo em código:

```
// Model
let contador = 0;

// View
function render() {
  document.getElementById('app').textContent = contador;
}

// Controller
document.getElementById('btn').addEventListener('click', () => {
  contador++;
  render();
});

render();
```

## MVVM (Model-view-viewmodel)

O MVVM surgiu como uma evolução do MVC, muito usado em frameworks como angular, vue e knockout

1. Model: ainda é a camada dos dados
2. View: continua sendo a interface visual
3. ViewModel: Substitui o controller, mas com uma diferença importante: ele mantém um espelho reativo do estado da view

O mvvm introduz o two-way data binding, ou seja, quando o model muda, a view muda automaticamente

# O QUE DIFERENCIA O REACT DE FRAMEWORKS MVVM-MVC TRADICIONAIS

exemplo no código:

```
// ViewModel com two-way binding
let nome = observable('Caio');

nome.subscribe(novoValor => {
  document.getElementById('titulo').textContent = novoValor;
});

// Quando o usuário digita algo:
document.getElementById('input').addEventListener('input', e => {
  nome.set(e.target.value);
});
```

Se o nome mudar no código, o input é atualizado automaticamente.

Se o usuário digitar no input, o Model muda também.

Isso é bidirecional (two-way) e funciona bem em aplicações pequenas, mas pode gerar complexidade e bugs em grandes sistemas, porque as mudanças podem vir de vários lugares ao mesmo tempo.

O MVC separa responsabilidades, mas exige controle manual da sincronização entre dados e interface.

O MVVM automatiza isso com two-way binding, mas sacrifica previsibilidade.

O React elimina o intermediário e o two-way binding, tratando a UI como uma função pura do estado, tornando o fluxo de dados unidirecional e mais fácil de entender.

O react elimina o controlador explícito, nos frameworks tradicionais, existe uma camada de controle intermediária, o controller no mvc e o.viewmodel no mvvm, que decide o que a interface deve mostrar e quando atualizá-la  
No react, essa camada desaparece, a UI é uma função direta do estado

## **UI = f(state)**

Ou seja, não há mais um controlador mandando renderizar, o react simplesmente recalcula a interface toda vez que o estado muda, isso traz previsibilidade, dado um estado, o resultado visual é sempre o mesmo

Frameworks como angular afotam two-way data binding, se o usuário altera um input, o modelo é atualizado automaticamente, se o modelo muda, a interface também muda. Isso é prático, mas pode causar comportamentos difíceis de rastrear, pois os dados fluem nos dois sentidos;

O react adota o one-way data flow, em que os dados fluem sempre do pai para o filho, via props, e só mudam quando o estado é atualizado explicitamente.

Isso dá controle total sobre onde e quando as mudanças ocorrem tornando o fluxo previsível e fácil de debugar.

Enquanto o MVC tradicional atualiza o dom real diretamente (re-renderizando ou alterando elementos manualmente), o react utiliza virtual DOM, uma representação virtual leve da árvore de elementos.

# O QUE DIFERENCIA O REACT DE FRAMEWORKS MVVM-MVC TRADICIONAIS

Quando o estado muda, o React compara, (Faz o diff) entre o novo virtual dom e o anterior e aplica apenas as diferenças necessárias no DOM real  
Isso garante eficiência e desempenho, mesmo em aplicações com centenas de atualizações por segundo

O react simplifica o modelo tradicional ao remover a camada intermediária entre lógica e interface  
Ele transforma a UI em uma função pura do estado, com um fluxo de dados unidirecional e abstração do DOM via virtual DOM.

Enquanto o MVC e o MVVM tentam sincronizar manualmente a lógica e a interface, o react unifica tudo isso dentro de um conceito simples:  
"Dado um estado, a interface é o resultado"

---

# MÓDULO 2

## O MOTOR DO REACT (VIRTUAL DOM E FIBER)

Entendendo como o React transforma  
estado em UI e o que realmente acontece  
na renderização.

# O QUE É O VIRTUAL DOM E POR QUE ELE EXISTE?

O DOM é a representação em árvore de todos os elementos html de uma página. Quando você altera algo no DOM, por exemplo, muda o texto de uma div, o navegador precisa recalcular posições, tamanhos, estilos e repintar a tela.

Essas operações são chamadas de:

- Reflow: o navegador recalcula o layout (posições, tamanhos, espaçamentos)
- Repaint: o navegador redesenha os pixels na tela.

Esses processos são custosos, principalmente quando correm muitas vezes em sequência.

Custo, nesse caso, significa, custo computacional, ou seja, o tempo e os recursos de hardware (CPU, memória, GPU) que o navegador precisa gastar para recalcular e redesenhar a página após cada mudança.

O virtual dom é uma cópia leve do DOM real, mantida na memória e não na tela. Serve como uma camada intermediária que permite ao react planejar as atualizações antes de aplicá-las.

O fluxo é:

1. Você altera o estado no react
2. O react gera uma nova árvore de virtual dom, com base no novo estado
3. Compara a árvore nova com a anterior (diffing/reconciliação)
4. Descobre exatamente o que mudou
5. Atualiza somente esas partes específicas no DOM real

isso evita que o navegador precise recalcular todo o layout da página a cada pequena alteração.

O virtual DOM é o espelho entre os dois mundos, é o espaço onde o react desenha a versão ideal da interface, compara com a versão anterior, decide o mínimo necessário para deixar o dom real em sincronia com essa visão.

É um não-lugar, não está nem no código puro nem na tela, mas é o reflexo fiel da UI derivada do estado.

Sobre performance:

O DOM real é pesado, sempre que você muda algo nele, o navegador precisa recalcular, até pequenas mudanças podem fazer o DOM ter que recalcular tudo.

O virtual DOM aplica somente a mínima mudança necessária, economizando recursos.

Sobre previsibilidade:

Dado um mesmo estado, a UI sempre será a mesma, não existem atualizações escondidas, o fluxo de dados é unidirecional e rastreável.

Se algo aparece errado na tela, você não precisa "caçar" onde o DOM foi manipulado. Você só precisa inspecionar o estado, porque é dele que tudo deriva.

Sobre abstração das atualizações:

Antes do react, o desenvolvedor precisava dizer como atualizar a interface, com o virtual DOM e o modelo declarativo, você apenas diz, "esse é meu estado atual e é essa a aparência que ele deve ter"

Resumo conceitual

O virtual DOM é mais que uma técnica de otimização, é a base filosófica que permite ao react unir simplicidade declarativa com eficiência real.

# RECONCILIAÇÃO COMO O REACT COMPARA ÁRVORES

Reconciliação é o processo que o react usa para sincronizar a interface (DOM real) com o estado da aplicação, de maneira eficiente.

Toda vez que o estado muda, o React:

- Recria uma nova árvore virtual
- Compara essa árvore com a árvore virtual anterior
- Descobre o que mudou
- Aplica apenas essas diferenças ao DOM real

Mas quais são os critérios do react para essa comparação?

O react usa regras heurísticas simples para decidir o que reaproveitar ou recriar:

- Mesmo tipo de elemento (nó): Se o tipo de elemento é o mesmo, ele mantém o nó e atualiza somente as suas props e filhos
- Tipo diferente: Se o tipo muda, por exemplo, de div para section, o react descarta o nó anterior e cria um novo do zero
- Componentes customizados: Se o componente é o mesmo < Card /> e < Card /> o react apenas renderiza de novo, se mudar, ele remove tudo e recria

Por essa estrutura, as keys são importantes no react, são identificadores únicos usados para que o react saiba qual item corresponde a qual entre renderizações, sem as keys, o react compara os elementos da lista posição a posição, o que poderia gerar trocas erradas, por exemplo, uma lista com 100 filmes, se o filme trocar a posição 100 pela 80, o react poderia ter problemas no diffing porque não conseguiria analisar se é o mesmo elemento ou não, com uma key identificadora, ele consegue garantir que o elemento seja sempre o mesmo independente da sua posição na lista

A reconciliação é o processo que torna o react previsível e eficiente:

- Evita reconstruir toda a interface a cada mudança
- Minimiza re-renderizações desnecessárias
- Garante consistência entre estado e DOM sem perda de performance

# DIFFING ALGORITHM E HEURÍSTICAS INTERNAS

Quando o estado muda, o React cria uma nova árvore de virtual DOM e precisa comparar essa árvore com a anterior para descobrir o que mudou.  
Mas comparar duas árvores arbitrárias nó por nó e filho por filho seria um processo muito pesado, teria complexidade de  $O(n^3)$  no pior caso.

O React implementa um algoritmo de diffing baseado em heurísticas para detectar diferenças entre árvores de Virtual DOM de forma eficiente.

O que seria uma complexidade de  $O(n^3)$ ?

Deriva de notação big-O, que serve para descrever o quanto tempo (ou custo) de um algoritmo cresce quando o tamanho da estrada aumenta

$n$  representa o tamanho da entrada, por exemplo:

- O número de elementos em uma lista
- O número de nós em uma árvore
- O número de operações que o algoritmo precisa preparar

O expoente indica o crescimento, a taxa de crescimento do custo

Complexidade  $O(1)$  seria um custo constante, independente do tamanho, seria como acessar um índice em um array, só precisa de uma operação

Complexidade  $O(n)$  seria um aumento de custo linear, ou seja, a quantidade de operações deriva da quantidade da entrada, como percorrer uma lista

Complexidade  $O(n^2)$  o custo cresce quadraticamente, como comparar todos os pares em um sistema de bubble sort

Complexidade  $O(n^3)$  é um aumento de custo cúbico, compara todas as combinações possíveis de 3 níveis, como, por exemplo, comparar nós e sub-nós de uma árvore

Complexidade  $O(2^n)$  representa um crescimento exponencial, ele testa todas as combinações possíveis, como, por exemplo, a tentativa de quebra de senha via força bruta.

No cenário do react, o desafio é comparar duas árvores, imagine que temos a árvore antiga e a árvore nova, e precisamos descobrir todas as diferenças possíveis entre elas

Cada nó pode ter:

- Filhos diferentes
- Estar em outra ordem
- Ter atributos alterados
- Ter sido movido, removido, ou criado

O problema é: como saber se um nó A na primeira árvore é o mesmo que o nó B na segunda?

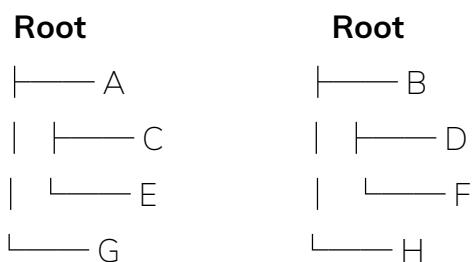
Se fizermos isso da maneira "ingênua":

Para cada nó da primeira árvore, precisaríamos comparar com cada nó da segunda árvore para descobrir se são equivalentes, isso, por si, já geraria um custo de  $O(n^2)$   
Mas não para por aí, cada nó pode ter subárvores inteiras, e cada uma dessas subárvores precisa ser comparada, a cada nível de profundidade, a complexidade cresce de forma combinatória , resultando em  $O(n^3)$  no pior caso

# DIFFING ALGORITHM E HEURÍSTICAS INTERNAS

Imagine uma árvore simples com 3 níveis e 100 nós:

**Árvore antiga:**      **Árvore nova:**



Se você não souber quais nós correspondem entre si, o algoritmo teria que tentar todas as combinações possíveis:

- Será que “A” virou “B”?
- Ou “A” virou “H”?
- E “C”? Foi para “D” ou “F”?

Para descobrir o mapeamento ideal, o algoritmo testaria todas as possibilidades, gerando um número cúbico de comparações.

Por isso o react usa heurísticas.

O react evita esse custo absurdo adotando regras simplificadas:

- Nunca tente adivinhar se um elemtno foi movido para outro lugar na árvore
- Assume que se o tipo é o mesmo, é o mesmo nó (atualiza só as props)
- Assume que se o tipo é diferente, o nó inteiro foi recriado
- Usa key para identificar nós estáveis em listas

Essas regras eliminam a necessidade de testar todas as combinações possíveis, e o algoritmo passa a rodar em tempo linear,  $O(n)$

Vale destacar que reconciliação e diffing são processos diferentes:

- Reconciliação é o processo completo, detectar as mudanças e atualizar o DOM real
- Diffing é a parte dentro da reconciliação, responsável apenas por comparar as árvores

# REACT FIBER: SCHEDULER, PRIORIDADE E INTERRUPÇÃO DE TAREFAS

O Fiber é uma reescrita interna do mecanismo de reconciliação, no modelo original (antes do react 16), todo o processo de reconciliação e atualização era sincrônico e bloqueante, ou seja, quando o estado mudava, o react reconstruía toda a árvore virtual, fazia o diff e aplicava as mudanças ao dom sem parar, isso podia fazer com que o navegador travasse por alguns milissegundos, o fiber foi criado para resolver exatamente isso

Ao invés de processar toda a árvore de uma vez, o react pode dividir o trabalho em partes menores e escolher quando continuar ou pausar cada uma delas

Cada nó da árvore de componentes agora é representado como um fiber node, um objeto que contém:

- As informações do componente (tipo, props, estado)
- Um ponteiro para o pai e para os filhos
- Metadados de prioridade
- O processo de renderização

O papel do Scheduler

O scheduler é o gerente de tarefas do react

Ele decide:

- Quantos pedacinhos do trabalho serão feitos por vez
- Quando pausar uma renderização longa
- Quando interromper um update de baixa prioridade

Esse modelo é chamado de cooperative scheduling, porque o react não tem controle total sobre a thread principal, ele apenas coopera com o navegador, fazendo um pouco de trabalho e depois devolvendo o controle

Mas como o react sabe o que tem mais prioridade e o que tem menos?

O react classifica as atualizações em diferentes níveis, por exemplo:

Alta prioridade: digitar em um input, clicar em um botão

Média prioridade: rolagem, animações leves

Baixa prioridade: recarregar dados, renderizar listas longas, fetch em background

Assim, se o usuário estiver digitando, o react pode adiar um re-render de uma lista grande até que a digitação seja processada.

O Fiber divide a renderização em duas fases, a render phase, com o reconciliation e o diffing (o processo onde o react recalcula o que precisa mudar) e a commit phase, com o mutation, a fase onde o react aplica as mudanças no dom real

Essa separação permite o agendamento (scheduling) e o controle de prioridade entre atualizações

Muitas pessoas confundem e acreditam que o fiber faz o react renderizar mais rápido, mas na realidade ele faz o react renderizar de maneira mais inteligente. O foco é responsividade, não tempo total de execução

# TEMPO DE RENDERIZAÇÃO, COMMIT PHASE E MUTATION PHASE

O que acontece quando o estado muda?

Quando um componente no react tem seu estado alterado (setState, useState, useReducer), o react inicia um novo ciclo de renderização, esse ciclo tem duas grandes fases:

- Render phase (fase de cálculo)
- Commit phase (fase de aplicação)

Na render phase, o react cria uma nova árvore de fiber nodes com base no estado atualizado, ele compara essa nova árvore com a anterior e descobre o que mudou.

Essa fase é pura e sem efeitos colaterais, o react apenas planeja o que deve acontecer, ela pode ser pausada, interrompida e refeita se um update mais urgente surgir

Depois que o react sabe o que mudou, ele parte para a commit phase, que é curta e síncrona (não pode ser pausada). Nessa fase, ele aplica as mudanças desejadas no DOM real

A commit phase é dividida internamente em três subetapas

- Before mutation phase: O react prepara o DOM antes da atualização, hooks como getSnapshotBeforeUpdate acontecem aqui.
- Mutation phase: o react altera o DOM real, adiciona, remove ou atualiza elementos, é o momento em que os nós reais do navegador são modificados
- Layout phase: o react executa efeitos sincronizados com o layout como useLayoutEffect. Depois disso, a UI já reflete o novo estado

Por que separar render e commit melhora a performance?

A separação permite:

- Pausar a renderização
- Evitar travamentos na thread principal
- Melhor controle sobre prioridades

O react só faz o trabalho pesado no DOM quando tem certeza do que precisa mudar

Você não enxerga as fases, o react controla isso internamente, você só nota o resultado, a interface responde rápido.

# COMO O REACT GARANTE FLUIDEZ SOB CARGA (CONCURRENT MODE)

Antes do Concurrent mode, o react trabalhava em modo síncrono e bloqueante (aquele modelo antigo pré-Fiber)

Isso significa que quando o estado mudava, o React precisava renderizar tudo de uma vez, até terminar

Se essa renderização fosse demorada, por exemplo, listas grandes, o navegador ficava travado, inputs paravam de responder, scroll engasgava, etc.

Com o concurrent mode, introduzido no react 18, veio a renderização interruptible e cooperativa, isso quer dizer que:

- O react pode pausar um trabalho de renderização longo
- Devolver o controle ao navegador (para manter a UI fluida)
- E retomar depois, do ponto onde parou

Ele faz isso por meio de duas técnicas principais:

- Time-slicing: Ele divide o trabalho em pequenas fatias
- Prioritização: o React classifica as tarefas (Um input digitado tem mais prioridade do que o re-render de uma lista longa por exemplo)

Abstrações que usam o concurrent mode:

Suspense: permite "pausar" a renderização enquanto dados são carregados  
useTransition: marca uma atualização como "não urgente"

Exemplos práticos em um E-commerce:

Sem o concurrent mode:

- O usuário digita no campo de busca
- O app trava por 200ms enquanto renderiza os resultados

Com o concurrent mode + useTransition:

- O usuário digita no campo de busca
- O react prioriza a digitação e renderiza os resultados em segundo plano - O campo responde imediatamente

O que significa responder imediatamente, ou priorizar, na prática:

Ao digitar uma letra no campo de busca, o react foca justamente em exibir o caractere que o usuário digitou na tela, imediatamente, mesmo que o react ainda esteja passando por processos de outras tarefas mais pesadas, como filtrar uma lista ou renderizar os resultados da busca

Quando usar e quando não usar useTransition?

- Use `useTransition` para atualizações não urgentes (buscas, filtros, listas longas).
- Evite em atualizações críticas (inputs controlados, toggles, formulários).
- O React já prioriza automaticamente eventos de input (useTransition) é um refinamento.

Apesar do nome "concorrente", o React não executa tarefas em paralelo. Ele continua rodando em uma única thread (JavaScript é single-thread), mas intercala as tarefas de modo cooperativo, liberando o controle para o navegador entre uma fatia e outra.

# COMO O REACT GARANTE FLUIDEZ SOB CARGA (CONCURRENT MODE)

Essa capacidade de pausar e retomar renderizações só é possível graças ao modelo Fiber, que representa cada componente como uma unidade de trabalho independente e rastreável.

Além de `useTransition`, o Concurrent Mode também é utilizado por hooks como `useDeferredValue` (que adia a atualização de valores pesados) e pela API Suspense, usada para lidar com carregamento assíncrono de dados e componentes.

# BATCHING DE ATUALIZAÇÕES E EVENT LOOP

O que é batching?

Batching significa agrupar múltiplas atualizações de estado, por exemplo, vários useState ou setContador) em uma única renderização.

A ideia é simples: se o React fosse re-renderizar o componente toda vez que um estado mudasse, o desempenho cairia muito

Para entender quando o react aplica essas atualizações, precisamos lembrar que o Javascript é single-threaded e organiza as tarefas em macrotasks e microtasks

- Microtasks: eventos como click, setTimeout, setInterval
- Microtasks: promises, async/await, queueMicrotask

O react espera o fim do ciclo de evento atual (macrotask) antes de aplicar o render. Ou seja, se você fizer múltiplos setState dentro de um mesmo evento de clique, o react vai:

1. Registrar todas as mudanças
2. Esperar o término do evento
3. Fazer o render apenas uma vez

Relação com o event Loop

O React espera o fim da macrotask atual (ex: clique, timer, requisição) antes de aplicar as mudanças.

Isso garante que todas as atualizações de estado que ocorreram dentro daquela tarefa sejam processadas juntas.

Em termos do event loop:

- O React acumula os updates (pendentes).
- Quando o ciclo de evento termina, ele agenda uma microtask interna para disparar o re-render.

Ligação com performance e reconciler:

O batching reduz:

- O número de passagens pelo reconciler (comparação entre árvores virtuais);
- O número de diffs no Virtual DOM;
- E o número de mutações no DOM real.

---

# MÓDULO 3

# HOOKS EM NÍVEL CONCEITUAL

Compreendendo o funcionamento  
interno, as restrições e as razões  
conceituais dos hooks.

# POR QUE HOOKS FORAM CRIADOS? (SUBSTITUIÇÃO DE CLASSES)

Antes dos Hooks, o React usava classes para criar componentes com estado e comportamento próprio.

Uma classe é uma forma de definir um "molde" para um componente, dizendo como ele deve se comportar, quais dados guarda e o que exibe.

Dentro da classe, usamos métodos especiais (como constructor, render, componentDidMount) para controlar cada momento da vida do componente.

Antes dos hooks, apenas componentes de classe podiam ter:

- Estado interno (this.state)
- Métodos de ciclo de vida (componentDidMount, componentDidUpdate, etc)
- E acesso direto a side effects

Os componentes funcionais eram simples, recebiam props e retornavam JSX, sem estado ou efeitos, isso criava uma divisão artificial entre:

- Componentes inteligentes (classes com estado e lógica)
- Componentes burros (funções puras de renderização)

As classes apresentavam algumas limitações:

1. Boilerplate e verbosidade

- Precisavam declarar constructor, chamar **super()**, fazer **this.state = {...}**, etc
- O código era repetitivo, mesmo para tarefas simples

2. Problemas com o this

- Era necessário fazer binding manual (**this.handleClick = this.handleClick.bind(this)**)
- Fácil cometer erros, especialmente para iniciantes

3. Lógica de lifecycle espalhada

- Um mesmo comportamento (ex: buscar dados de uma API) era dividido em vários métodos
- Dificultava a reutilização e manutenção de lógica entre componentes

## A proposta dos Hooks

Os hooks foram criados para trazer poder das classes para funções, mas com menos complexidade e mais previsibilidade

Hooks seguem a filosofia da composição, pequenas funções que compartilham comportamento, em vez da herança usada nas classes.

Isso torna o código mais previsível, fácil de testar e de reutilizar, já que podemos extrair lógicas de estado em custom hooks reutilizáveis

# POR QUE HOOKS FORAM CRIADOS? (SUBSTITUIÇÃO DE CLASSES)

Exemplo simplificado de classe:

```
class Contador extends React.Component {
  constructor() {
    super();
    this.state = { contador: 0 };
  }

  incrementar = () => {
    this.setState({ contador: this.state.contador + 1 });
  }

  render() {
    return (
      <div>
        <p>{this.state.contador}</p>
        <button onClick={this.incrementar}>+1</button>
      </div>
    );
  }
}
```

Exemplo equivalente com Hook:

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

  return (
    <div>
      <p>{contador}</p>
      <button onClick={() => setContador(contador + 1)}>+1</button>
    </div>
  );
}
```

Os hooks permitem fazer a mesma coisa das classes, ter estado, efeitos e lógica, mas usando funções simples, com código mais claro e previsível.

Resumo conceitual

- Antes: só classes tinham estado, efeitos e ciclo de vida.
- Agora: qualquer função pode ter comportamento complexo com Hooks.
- Benefício: menos código, menos erros, mais composição.

# O CICLO DE VIDA FUNCIONAL: RENDER → COMMIT → CLEANUP

Nos componentes funcionais, o React organiza a execução em fases bem definidas, que substituem antigos métodos de ciclos de vida das classes (`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`)

O fluxo pode ser resumido em três momentos principais:

## **Render Phase (fase de renderização):**

- O react chama a função do componente para descobrir o que deve ser exibido
- Nenhuma alteração é feita no dom ainda, é como uma simulação
- Cada render é independente: ele cria uma nova versão da função com seus próprios valores de estado e props.
- Aqui, o react prepara o virtual dom com base no retorno do componente (jsx)

## **Commit phase (fase da aplicação)**

- O react aplica as mudanças calculadas na render phase no DOM real
- Essa é a única fase onde o navegador realmente exibe as alterações visuais
- Após o commit, o react executa os efeitos (`useEffect`, `useLayoutEffect`), dependendo do tipo: `useEffect` executa de forma assíncrona após o commit, `useLayoutEffect` executa sincronamente logo após a atualização do DOM, mas antes da tela ser repintada

## **Cleanup phase (fase da limpeza)**

- Antes de um novo efeito ser aplicado, o react executa a função de limpeza retornada por `useEffect`
- Também ocorre quando o componente é removido do DOM (desmontado)
- Serve para cancelar timers, remover event listeners ou limpar assinaturas de APIs e sockets

Cada render é uma nova execução da função, não uma atualização da anterior

O React separa a lógica de cálculo (render) da aplicação real (commit)

O cleanup garante que efeitos antigos não causem vazamentos ou comportamentos inesperados

Mas por que saber isso?

Entender o ciclo de vida funcional do React (render → commit → cleanup) é essencial para compreender como e quando seu componente realmente “acontece” na tela. Esse conhecimento permite prever o comportamento do React ao atualizar estados, aplicar efeitos ou remover componentes. Saber o que ocorre em cada fase evita erros comuns, como efeitos executando mais vezes do que o esperado, dados “desatualizados” dentro de closures, ou vazamentos de memória por falta de limpeza. Além disso, compreender esse ciclo ajuda a otimizar a performance, reduzindo renderizações desnecessárias e garantindo que o código se mantenha previsível, eficiente e fácil de depurar. Em resumo, dominar o ciclo de vida é o que transforma o uso do React em engenharia de interface de verdade.

# REGRAS DOS HOOKS (ORDEM, PUREZA E CHAMADAS)

Por que existem regras para os hooks?

Os hooks (como useState, useEffect, useMemo) são a forma do react ligar o ciclo de vida e o estado a funções

Mas esse é o ponto crítico, o react não identifica hooks pelos seus nomes, e sim pela ordem em que são chamados durante o render

Quando um componente react é renderizado, ele roda a função do componente do início ao fim

Exemplo simples:

```
function Saudacao() {
  const nome = "Caio";
  return <p>Olá, {nome}</p>;
}
```

O react chama essa função, obtém o JSX e o mostra na tela. Até aqui, tudo é simples e estático, não há estado mudando. Mas, e se quisermos guardar um valor que muda?

Para isso usamos hooks, como useState

```
function Contador() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

Aqui acontece algo mágico:

- useState(0) pede ao react para guardar um valor (0)
- O react guardará esse valor entre renders
- Quando clicamos no botão, setCount pede ao react para renderizar de novo
- Na nova renderização, o react devolve o valor guardado, não recria o 0)

Ao clicar no botão, o react não muda a variável count diretamente, o que o react faz é registrar o novo valor (1) em sua memória interna e pedir uma nova renderização do componente

Quando você clica no botão e chama **setCount(count + 1)**

O react atualiza o valor dentro da posição de 0, de 0 para 1

Em seguida, renderiza o componente de novo

Agora, quando a função roda novamente, o react olha para a posição 0 e devolve o valor 1, o mais recente

O react não "lê" nomes de variáveis, ele só sabe que o primeiro hook chamado pertence à gaveta 0, o segundo hook é da gaveta 1 e daí por diante

Se a sequência mudar, como por exemplo, chamar um hook dentro de um if, o react abre a gaveta errada e os estados se misturam

Isso é o que chamamos de consistência posicional

# REGRAS DOS HOOKS (ORDEM, PUREZA E CHAMADAS)

Os Hooks do React, como useState e useEffect, só podem ser usados dentro de funções de componente ou outros hooks customizados porque eles dependem do contexto de execução controlado pelo React. Esse contexto permite que o React rastreie o ciclo de vida do componente, mantenha o estado entre renders e gerencie efeitos colaterais de forma previsível. Se um hook for chamado fora desse contexto, como dentro de uma função comum ou em escopo global, o React não consegue associar corretamente o estado ou reagendar atualizações, resultando em erros ou comportamentos inesperados. Em outras palavras, para que o hook funcione corretamente, ele precisa existir dentro de um ambiente onde o React possa “olhar” e controlar cada chamada, garantindo consistência e integridade do estado.

As funções de componente no React devem ser puras, o que significa que, para os mesmos inputs (props e estado), elas sempre retornam o mesmo JSX e não causam efeitos colaterais durante a renderização. Efeitos colaterais incluem ações como chamadas de API, manipulação direta do DOM ou alterações de variáveis externas. Manter a função pura garante que o React possa renderizar o componente de forma previsível, repetir renders, interromper ou reagendar renderizações sem riscos de resultados inconsistentes. Quando efeitos colaterais são necessários, eles devem ser isolados dentro de hooks como `useEffect`, permitindo que o React controle quando e como eles ocorrem, mantendo a integridade e previsibilidade do fluxo de renderização.

# USESTATE: CLOSURES E ESTADO PERSISTENTE ENTRE RENDERIZAÇÕES

Estado persistente via "array interno"

Cada useState cria uma gaveta interna no react, que guarda o valor do estado entre renders.

Mesmo que a função do componente seja chamada várias vezes, o react recupera o valor da gaveta correta, mantendo o estado atualizado

```
function Contador() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

- O 0 é passado ao useState e só é usado na primeira renderização
- Nas renderizações seguintes, o react devolve o valor guardado internamente, que pode ser 1, 2, 3, conforme os cliques

Pense em useState como uma gaveta numerada de memória

- Primeiro useState = gaveta 0, segundo = gaveta 1, etc
- setCount troca o conteúdo da gaveta mas não muda a variável temporária dentro da função, o react devolve o novo valor na próxima renderização

O hook é uma função especial do react, quando você chama o hook ele cria uma gaveta interna no react para guardar o estado daquele componente, o retorno do hook é um array de dois itens:

1. O valor atual do estado, (count) que é uma cópia daquele dado no render
  2. Uma função de atualização (setCount) que o react fornece para alterar o valor dentro da gaveta e disparar uma nova renderização
- Ou seja, o hook é a máquina e [valor, função] é o kit que a máquina te entrega para ler e atualizar o estado de forma segura e controlada pelo react,

## Closures e estado persistente (próximo nível)

Cada render fecha sobre a versão do estado via closure

- Cada render do componente cria uma nova execução da função
- As variáveis declaradas dentro da função, incluindo o valor retornado pelo hook, ficam "fechadas" naquela render
- Callbacks (como onClick) lembram do valor do estado de render em que foram criadas, graças às closures

Uma closure é uma função que "lembra" do ambiente em que foi criada, incluindo todas as variáveis que estavam visíveis naquele momento, mesmo que seja executada depois, em outro contexto

## Batched updates e setState assíncrono

O setState não muda o estado imediatamente

O react agrupa várias chamadas de setState dentro do mesmo evento para otimizar renderizações (batched updates)

# USESTATE: CLOSURES E ESTADO PERSISTENTE ENTRE RENDERIZAÇÕES

No React, quando usamos useState, cada render cria uma closure que “lembra” do valor do estado naquele momento. Por isso, se chamamos **setCount(count + 1)** várias vezes dentro do mesmo evento, cada chamada ainda usa o valor fechado na closure da render atual. Isso significa que, mesmo que façamos duas chamadas seguidas, ambas podem calcular **count + 1** usando o mesmo valor antigo, resultando em apenas uma atualização visível após o render. Para contornar esse comportamento e garantir que cada atualização use o valor mais recente da gaveta de estado, devemos usar a função de atualização (**setCount(prev => prev + 1)**), que acessa diretamente o estado atual armazenado pelo React e aplica a mudança de forma segura, independentemente das closures ou do batching de atualizações.

# USEEFFECT: SINCRONIZAÇÃO COM O MUNDO EXTERNO

Enquanto o useState guarda dados que mudam, o useEffect lida com efeitos colaterais que ocorrem fora da renderização pura

Exemplos de efeitos colaterais:

- Fazer uma chamada de API
- Atualizar o título da aba (document.title)
- Adicionar e remover listeners do DOM
- Sincronizar estado com localStorage

Essas ações não podem acontecer durante o render, pois o render deve ser puro, apenas calcular o que mostrar, sem modificar o mundo ao redor

Por que o useEffect não é chamado durante o render?

Durante o render, o React está calculando o que deve aparecer na tela, executar efeitos nessa fase quebraria a pureza e interromperia o processo de reconciliação

Então o React:

- Roda o render
- Atualiza o DOM
- Depois disso, executa os efeitos definidos em useEffect

Antes de avançar, precisamos entender o que é uma closure, closure são funções que acessam variáveis que estão fora do seu próprio escopo

Ex:

**let nome = Caio**

```
function sayHello() {
  console.log(nome)
}

sayHello()
```

Agora, pense em um componente React como uma função que é chamada várias vezes.

Cada vez que o React renderiza o componente, ele executa de novo a função, criando um novo escopo e, portanto, novas closures

Ex:

```
function Contador() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  console.log("Renderizou com count =", count);
  return <button onClick={handleClick}>{count}</button>;
}
```

# USEEFFECT: SINCRONIZAÇÃO COM O MUNDO EXTERNO

Quando o componente é renderizado: o react executa **Contador()** e cria uma nova closure onde count vale, por exemplo, 0; handleClick é criado dentro dessa closure, ele lembra do count daquela renderização. Quando o botão é clicado, o handleClick ainda vê o count antigo, mesmo que o componente já tenha sido re-renderizado depois

Se dentro de um evento ou useEffect você usa uma variável de estado, o react não atualiza essa variável dentro da closure antiga, ela é uma cópia congelada da render anterior

Por isso o nome stale closure (closure obsoleta):

Ela carrega dados antigos que não refletem mais o estado atual do react

## Por que useEffect não é chamado durante o render?

O useEffect não é executado durante a renderização porque o React precisa manter a pureza da função de componente. Durante o render, o React está apenas calculando o JSX que será exibido, sem alterar nada fora desse cálculo. Executar efeitos nesse momento, como chamadas de API, manipulação do DOM ou timers, quebraria essa previsibilidade e poderia gerar inconsistências no fluxo de reconciliação. Por isso, o React aguarda o commit phase, atualiza o DOM e só então roda os efeitos declarados com useEffect.

## Como evitar stale closures dentro de efeitos?

Stale closures ocorrem quando uma função dentro do useEffect “lembra” de valores antigos de estado ou props do render em que foi criada, em vez de acessar os valores mais recentes. Para evitar esse problema, é essencial usar corretamente o array de dependências do useEffect, listando todas as variáveis externas que o efeito utiliza. Outra abordagem é usar a função de atualização do estado (**setState(prev => newState)**), que sempre acessa o valor mais recente da gaveta de estado. Em casos específicos, refs (useRef) podem ser utilizadas para armazenar valores mutáveis que precisam ser lidos por callbacks assíncronos, sem depender da closure do render.

## Diferença entre efeitos de montagem, atualização e desmontagem

Os efeitos no React podem ser classificados em três categorias. Os efeitos de montagem são executados apenas uma vez, após o componente ser renderizado pela primeira vez, e são úteis para buscar dados iniciais ou adicionar listeners. Os efeitos de atualização rodam toda vez que uma das dependências do useEffect muda, permitindo sincronizar o efeito com alterações de estado ou props. Já os efeitos de desmontagem são tratados pelo cleanup retornado pelo useEffect, que é executado antes de reexecutar o efeito ou antes de o componente ser removido da tela, garantindo a liberação de recursos, como timers ou listeners, evitando vazamentos e efeitos duplicados.

# USEREF: IDENTIDADE E PERSISTÊNCIA FORA DO CICLO DE RENDER

useRef é um hook do react utilizado para armazenar um valor persistente entre renderizações sem causar uma nova renderização quando esse valor muda. Ele retorna um objeto imutável no formato { current: valor }

A principal ideia é a seguinte:

- Enquanto useState guarda dados reativos (que fazem o componente re-renderizar quando mudam), useRef guarda dados mutáveis (que o React ignora durante o ciclo de render)

Ou seja, o valor dentro de ref.current permanece o mesmo entre as chamadas da função do componente, o react não recria o objeto a cada renderização. Isso faz com que ele seja ideal para armazenar referências ou valores que precisam sobreviver à múltiplos renders.

Quando se pensa em estado, normalmente se pensa em dados que mudam e precisam aparecer na tela, e esse é exatamente o papel do useState.

Por exemplo:

```
const [count, setCount] = useState(0);
```

Cada vez que o count muda, o react re-renderiza o componente para refletir o novo valor visualmente.

Mas há valores que mudam, porém não fazem parte da interface, são dados "operacionais", usados só para controle interno, e é aqui que o useRef brilha.

O useRef é como uma gaveta de valor estável que:

- O react não monitora (não dispara renders)
- Mas você pode abrir e alterar quando quiser
- E o conteúdo persiste entre renders

## Mas por que as alterações do useRef não disparam re-render?

O useRef cria um objeto persistente com uma propriedade .current, que o React nunca rastreia como parte do ciclo de renderização. Isso significa que, quando você altera ref.current, o valor é atualizado mutavelmente, sem passar pelo mecanismo de reconciliação do react. Em outras palavras, o React não "sabe" que algo mudou dentro da ref, porque ela não faz parte do estado, e isso é intencional, refs são projetadas para armazenar dados que não afetam a interface, como elementos do dom, contadores internos ou valores de controle. Essa característica evita renders desnecessários e mantém a performance previsível

Em geral, usamos useRef em vez de useState quando precisamos armazenar informações que mudam ao longo do tempo, mas não impactam o que é renderizado. Isso inclui valores auxiliares usados em cálculos, controle de timers, manipulação de eventos, integração com APIs externas ou referência direta a elementos do DOM. Alterar um estado (useState) notifica o React para atualizar a interface, enquanto atualizar uma ref (useRef.current = ...) ocorre de forma silenciosa, sem disparar o ciclo de renderização. Essa distinção é essencial para manter a performance e a previsibilidade do componente: useState para dados visuais, useRef para dados operacionais.

# USEREF: IDENTIDADE E PERSISTÊNCIA FORA DO CICLO DE RENDER

Quando alteramos o valor de `ref.current`, a mudança acontece apenas em memória, "por baixo dos panos", o react não detecta e nem reage a ela, isso acontece porque o objeto retornado por `useRef` é estável e não faz parte do ciclo reativo do react. Em outras palavras, o react não "observa" o que acontece dentro do `ref.current`.

# USEMEMO E USECALLBACK: MEMOIZAÇÃO E REATIVIDADE CONTROLADA

Memoizar, no contexto de programação e react significa armazenar o resultado de um cálculo ou função para reutilizá-lo futuramente sem precisar recalculará-lo

Imagine que você tem uma função que realiza um cálculo pesado, como somar milhares de números ou filtrar uma lista gigante. Toda vez que o componente renderiza, essa função seria executada de novo, mesmo que os dados não tenham mudado. Memoizar resolve isso: o resultado da função é guardado em memória junto com as dependências que influenciam o cálculo. Enquanto essas dependências não mudarem, o react simplesmente retorna o valor já calculado, poupando processamento.

Quando você usa useMemo ou useCallback, o react cria internamente uma espécie de variável oculta para armazenar o resultado da função (no caso do useMemo) ou a própria referência da função (no caso de useCallback)

Essa variável não aparece no seu código como uma constante declarada, mas o react mantém o valor entre renders.

É como se o react tivesse uma memória interna por componente, organizada por gavetas numeradas (semelhante ao que faz com useState) e cada memoização ocupa uma dessas gavetas enquanto o componente existir.

## **useMemo**

O useMemo é um hook que memoiza valores computados. Ele recebe uma função que retorna um valor e um array de dependências. Enquanto as dependências não mudarem, o react aproveita o valor já calculado em vez de recalcular a função.

Isso é útil para cálculos pesados ou objetos complexos que não precisam ser recalculados a cada renderização, evitando desperdício de processamento e re-renderizações desnecessárias de componentes filhos que dependem desse valor

## **useCallback**

O useCallback é semelhante, mas memoiza funções, não valores. Ele retorna a mesma função entre renders enquanto suas dependências não mudarem.

Isso é importante quando passamos funções como props para componentes filhos que usam React.memo ou dependem de referência estável. Sem useCallback, a função seria recriada a cada render, disparando re-renderizações desnecessárias dos filhos

O processo de memoização ajuda no desempenho, evitando cálculos e render repetidos desnecessários

Mas se tudo parece tão ótimo, por que não memoizar absolutamente tudo? Memoizar tudo aumenta a complexidade e o consumo de memória, podendo até prejudicar o desempenho se feito sem critério

## **Como eu devo decidir se devo memoizar um valor?**

Pergunte-se: "Esse valor ou função precisa ser recalculado em todo render, mesmo quando os dados não mudaram?"

# USEMEMO E USECALLBACK: MEMOIZAÇÃO E REATIVIDADE CONTROLADA

Exemplos em um e-commerce:

- Filtrar produtos por categoria ou preço em listas grandes
- Calcular total do carrinho com descontos e impostos complexos
- Gerar recomendações personalizadas com base em várias regras

Utilizemos como base o primeiro exemplo:

```
const produtosFiltrados = useMemo(() => {
  return produtos.filter(p => p.categoria === categoriaSelecionada);
}, [produtos, categoriaSelecionada]);
```

O que deve ser memoizado?

A resposta é: o resultado do filtro, ou seja, o array de produtos já filtrado

O raciocínio é o seguinte:

- Produtos e categoria selecionada são as dependências
- Enquanto esses valores não mudarem, o react utiliza o array já filtrado, sem recalcular o filter() a cada renderização
- Isso é especialmente útil em listas grandes, onde recalcular o filtro toda vez que o componente renderizar seria custoso

No React, o useMemo por si só não mantém histórico de valores; ele apenas memoiza o resultado do cálculo da renderização atual com base nas dependências fornecidas. Isso significa que, se você tem um filtro de produtos por input, ao digitar "abr" o resultado é calculado e armazenado temporariamente. Ao digitar "abra", o useMemo recalcula o filtro para o novo valor e substitui o resultado anterior. Se o usuário voltar para "abr", o cálculo será feito novamente, pois o valor memoizado antigo não é preservado. Para manter histórico de buscas ou resultados antigos, é necessário combinar o useMemo com uma estrutura de armazenamento persistente, como um objeto ou useRef, que armazene cada resultado antigo e permita reutilizá-lo sem recalcular, criando assim uma cache manual dentro do componente

# USEREDUCER E O PADRÃO DE ISOLAMENTO DE LÓGICA

O useReducer é um hook do React que oferece uma forma estruturada de gerenciar estados complexos, especialmente quando há múltiplas transições de estado ou lógica que depende do valor anterior. Ele é baseado no conceito de reducer puro, que é uma função **(state, action) => newState**. O reducer recebe o estado atual e uma ação, e devolve um novo estado sem modificar o anterior, mantendo a imutabilidade. Essa abordagem facilita a previsibilidade, pois cada ação gera um efeito conhecido e também simplifica testes unitários da lógica de atualização de estado, isolando a transformação do estado do componente em si.

Ele é como um gerenciador de estado centralizado para um componente. Ao invés de usar vários useState para cada pedaço de dado e espalhar lógica de atualização por todo lado, você cria uma função (o reducer) que recebe o estado atual e uma ação que descreve o que você quer fazer. Essa função devolve um novo estado, sem alterar o antigo.

Quando falamos em novo estado no react, não significa que você tem que redesenhar a interface manualmente, significa que você tem uma nova versão dos dados que definem a UI.

Imagine que você tem um carrinho de compras, com quantidade de itens e total:

```
const initialState = {
  itens: [],
  total: 0
};

function carrinhoReducer(state, action) {
  switch (action.type) {
    case 'ADICIONAR_ITEM':
      const novoltens = [...state.itens, action.item];
      const novoTotal = state.total + action.item.preco;
      return { itens: novoltens, total: novoTotal };

    case 'REMOVER_ITEM':
      const filtrado = state.itens.filter(i => i.id !== action.id);
      const totalAtualizado = filtrado.reduce((acc, i) => acc + i.preco, 0);
      return { itens: filtrado, total: totalAtualizado };

    default:
      return state;
  }
}
```

- Estado atual: `{ itens: [], total: 0 }`
- Ação: `{ type: 'ADICIONAR_ITEM', item: { id: 1, nome: "Tênis", preco: 100 } }`
- Reducer: calcula `{ itens: [{id:1, nome:"Tênis", preco:100}], total: 100 }`

# USEREDUCER E O PADRÃO DE ISOLAMENTO DE LÓGICA

com useState:

```
function Carrinho() {
  const [itens, setItens] = useState([]);
  const [total, setTotal] = useState(0);

  function adicionarItem(item) {
    setItens(prevItens => [...prevItens, item]);
    setTotal(prevTotal => prevTotal + item.preco);
  }

  function removerItem(id) {
    setItens(prevItens => {
      const filtrado = prevItens.filter(i => i.id !== id);
      setTotal(filtrado.reduce((acc, i) => acc + i.preco, 0));
      return filtrado;
    });
  }

  return (
    <div>
      <h2>Total: {total}</h2>
      <ul>
        {itens.map(i => <li key={i.id}>{i.nome} - R${i.preco}</li>)}
      </ul>
      {/* Botões chamariam adicionarItem ou removerItem */}
    </div>
  );
}
```

Comparando useState vs useReducer

#### **useState:**

- Cada pedaço do estado é independente (itens, total)
- Cada função precisa atualizar todos os estados relacionados
- Se houver muitas regras, a lógica fica espalhada e mais difícil de manter
- Ex: removerItem precisa atualizar itens e total manualmente

#### **useReducer:**

- Todo estado relacionado é centralizado em um único objeto ({itens, total})
- A lógica de atualização fica toda dentro do reducer, previsível e testável
- Facilita lidar com múltiplas ações e transições de estado complexas

Em resumo, o useReducer é como um controlador central, enquanto useState é como vários interruptores independentes. Para casos simples, useState é suficiente, para lógica mais complexa ou interdependente, useReducer deixa o código mais limpo e previsível

# USEREDUCER E O PADRÃO DE ISOLAMENTO DE LÓGICA

O useReducer serve justamente para agrupar e coordenar ações que pertencem ao mesmo contexto lógico

Com useState, cada parte do estado é independente, e cabe a você atualizar todas as partes relacionadas manualmente

Em um exemplo de um ecommerce, o useReducer seria especialmente útil para gerenciar estados complexos e interligados como um carrinho de compras. Imagine que o carrinho contenha diversos elementos, lista de livros, total, desconto, frete. Cada uma dessas partes depende das outras, ao adicionar um item, o total muda, ao aplicar um cupom, o desconto altera o valor final, ao calcular o frete, o preço total é atualizado. Se usássemos vários useState, seria necessário atualizar manualmente cada parte do estado e garantir que todas permanecessem sincronizadas. Com o useReducer, todo o controle fica centralizado em uma única função (carrinhoReducer), que recebe uma ação (**ADICIONAR\_ITEM**, **REMOVER\_ITEM**, **APLICAR\_DESCONTO**) e calcula o novo estado completo do carrinho de forma previsível. Isso simplifica a lógica, evita inconsistências e torna o código mais fácil de testar e manter, já que cada mudança de estado segue um fluxo único e claro: ação -> reducer -> novo estado -> atualização da UI

A diferença de um reducer local (via useReducer) e um gerenciador global de estado (como Redux ou zustand) está no escopo e na finalidade do controle de estado. O useReducer atua dentro de um único componente ou árvore pequena de componentes, sendo ideal para isolar e gerenciar estados complexos de forma local, como o carrinho de um e-commerce, um formulário com validações ou um fluxo de checkout. Já bibliotecas como Redux ou Zustand aplicam o mesmo princípio de reducer, mas em um nível global, permitindo compartilhar o estado entre múltiplas partes da aplicação sem precisar repassar props manualmente. Ou seja, useReducer resolve a lógica complexa dentro de um componente específico, enquanto react e zustand estendem essa lógica para toda a aplicação, centralizando o estado e tornando-o acessível em qualquer ponto da árvore de componentes.

# HOOKS CUSTOMIZADOS: COMPOSIÇÃO E REUTILIZAÇÃO DE COMPORTAMENTO

Hooks customizados são uma forma de organizar e reutilizar lógica de estado e efeitos entre componentes

Eles permitem combinar hooks nativos (useState, useEffect, useReducer, useMemo) dentro de uma função que começa com use, por exemplo **useCarrinho()**, **useFetchProdutos()**

A ideia é extrair padrões repetitivos de dentro dos componentes e colocá-los em um local reutilizável.

Por exemplo, em um e-commerce você pode ter

- useCarrinho() -> controla itens, total e desconto
- useFiltroProdutos() -> lida com busca e filtragem
- useAuth() -> gerencia login e logout do usuário

Geralmente os custom hooks combinam vários hooks nativos quando o comportamento é composto

No caso do useCarrinho, por exemplo, ele precisa coordenar múltiplas responsabilidades:

- Gerenciar o estado do carrinho (itens, total, descontos, frete, etc), para isso, usa-se o useReducer (ou useState, em casos mais simples)
- Persistir os dados localmente, para isso, usa-se o useEffect, para salvar o carrinho no localStorage sempre que o estado mudar
- Derivar valores sem recalcular tudo (como o total com desconto), para isso, usa-se o useMemo, para evitar cálculos desnecessários

Um custom hook é um agrupador de hooks que trabalha em torno de uma lógica de domínio. Ele não cria novas capacidades, apenas organiza e orquestra hooks nativos para que o componente final não precisa lidar com todos esses detalhes.

## Mas por que eu usaria um custom hook?

- Reutilização de lógica:

Quando dois ou mais componentes precisam da mesma regra de negócio, você não quer copiar e colar os mesmos useState, useEffect, useMemo.

Ao invés disso, você cria um custom hook, por exemplo (useAuth) e simplesmente chama onde precisar

Ex:

Tanto o header quanto a página de checkout precisam saber se o usuário está logado.

Ambos usam const { user, logout } = useAuth()

- Isolamento e clareza

Mesmo que só um componente use aquela lógica, se ela for grande ou envolver muitos hooks, o componente pode ficar confuso.

Mover a lógica para um custom hook deixa o componente mais legível e focado apenas em renderizar a UI, não em lidar com estados e efeitos.

O componente vira declarativo e o hook fica responsável pela "mecânica"

# HOOKS CUSTOMIZADOS: COMPOSIÇÃO E REUTILIZAÇÃO DE COMPORTAMENTO

- Testabilidade e manutenção

Você cria um hook não para ter "novos poderes", mas para organizar, reutilizar e dar coesão a pedaços de lógica relacionados ao mesmo comportamento.  
É a forma de encapsular "como algo funciona" e deixar os componentes apenas mostrarem "o que deve aparecer"

Antes dos hooks, a principal forma de compartilhar lógica entre componentes era com high-order components (HOCs) ou render props.

Um HOC é uma função que recebe um componente e devolve um novo componente com funcionalidades adicionais.

Com os hooks, o react passou a permitir reutilizar lógica de estado e efeitos diretamente dentro das funções, sem precisar criar novos componentes

Os HOCs foram uma ótima solução antes dos hooks, mas traziam complexidade e profundidade desnecessária.

Os hooks customizados substituíram esse padrão de forma mais natural, mantendo o mesmo poder de reutilização de lógica, mas com menos camadas, menos acoplamento e mais clareza

# O PROBLEMA DO STALE STATE E AS CLOSURES EM REACT

O termo stale state (estado "velho", "obsoleto") descreve situações em que uma função dentro do react usa uma versão antiga do estado, mesmo que o estado já tenha mudado.

Isso acontece por causa das closures (funções fechando sobre variáveis do momento em que foram criadas).

Em react, cada renderização é uma fotografia isolada, com seus próprios valores de estado e props.

Quando você cria uma função dentro de um componente (ou dentro de um hook), ela captura os valores da renderização atual.

Mesmo que o estado mude depois, aquela função continua "presa" aos valores antigos

Pense nesse exemplo simples:

```
function Contador() {
  const [count, setCount] = useState(0);

  function logCount() {
    console.log(count);
  }

  useEffect(() => {
    const interval = setInterval(logCount, 1000);
    return () => clearInterval(interval);
  }, []);

  return <button onClick={() => setCount(count + 1)}>Aumentar</button>;
}
```

Mesmo clicando várias vezes no botão, o **console.log(count)** sempre mostra 0, por quê?

Porque a função logCount foi criada na primeira renderização e a closure dela "fechou" sobre o count = 0.

Quando o estado muda, o componente re-renderiza, mas o intervalo existente continua chamando a versão antiga de logCount

Esse intervalo é um intervalo de tempo criado pela função nativa do javascript setInterval

Ele executa a função callback repetidamente, a cada x milissegundos

No exemplo

```
useEffect(() => {
  const interval = setInterval(logCount, 1000);
  return () => clearInterval(interval);
}, []);
```

o setInterval está chamando a função logCount() a cada 1 segundo. Mas aí que está o problema, como o useEffect foi executado só uma vez (por causa do []), ele cria apenas um intervalo, usando a versão de logCount que existia na primeira renderização

Logo:

- Mesmo que o estado count mude depois
- O intervalo continua rodando com a função antiga, que lembra apenas do **count = 0**

# O PROBLEMA DO STALE STATE E AS CLOSURES EM REACT

Esse comportamento acontece porque o array de dependências `[]` diz ao React que o efeito não depende de nenhum valor externo, portanto ele deve ser executado somente uma vez, na montagem do componente. O resultado é que o `setInterval` fica rodando indefinidamente com a versão inicial da função, presa à closure antiga, sem acesso ao estado atualizado.

Para evitar isso, é preciso informar ao React quando o efeito deve ser reexecutado, adicionando dependências, por exemplo, `[count]` ou `[logCount]`, para que o `useEffect` recrie o intervalo sempre que esses valores mudarem. Assim, a função usada dentro do intervalo passa a enxergar o estado mais recente.

Outra forma é usar a função de atualização do estado, que recebe o valor anterior como argumento (`setCount(prev => prev + 1)`), ou usar refs para armazenar valores mutáveis sem depender da renderização. Essas estratégias evitam que a lógica fique “presa” a versões antigas do estado, garantindo que os efeitos e callbacks do React sempre trabalhem com dados atualizados.

Em alguns casos, adicionar dependências ao `useEffect` resolve o problema do stale state, mas pode gerar reexecuções desnecessárias se o efeito for custoso, por exemplo, recriar intervalos, websockets ou assinaturas a cada atualização. Nesses cenários, uma alternativa é usar refs para armazenar o valor mais recente de uma variável sem depender da reatividade do estado, permitindo que o efeito permaneça estável enquanto ainda acessa dados atualizados. É importante também manter a lista de dependências correta: se faltar alguma variável usada dentro do efeito, o React não conseguirá sincronizar a lógica com o estado atual, gerando comportamentos inconsistentes. Em essência, o problema de stale state é uma quebra na reatividade, o efeito ou callback deixa de reagir ao estado real do componente. Entender esse vínculo entre closures, ciclo de renderização e dependências é essencial para escrever hooks previsíveis e evitar bugs sutis em comportamentos assíncronos ou contínuos.

---

# MÓDULO 4

## ESTADO, DADOS E ARQUITETURA

Entendendo como o React lida com o estado, quais são os tipos e os padrões arquiteturais envolvidos.

# TIPOS DE ESTADO - LOCAL, DERIVADO, GLOBAL E REMOTO

Em uma aplicação react, compreender os tipos de estado é essencial para estruturar bem a arquitetura de dados e evitar bugs ou re-renderizações desnecessárias.. O estado que dá "vida" à interface, definindo o que deve ser exibido em cada momento, mas ele pode existir em diferentes níveis de escopo e responsabilidade

## Estado local:

O estado local é aquele que pertence a um único componente, ele controla apenas o comportamento interno. É criado com hooks como useState ou useReducer e deve ser usado sempre que a informação não precisa ser compartilhada.

Ex:

```
function Counter() {
  const [count, setCount] = useState(0)
  return (
    <button onClick={() => setCount(count + 1)}>
      Cliquei {count} vezes
    </button>
  )
}
```

Esse estado vive e morre junto com o componente, nenhum outro componente precisa saber dele.

## Estado derivado:

O estado derivado é calculado a partir de outros estados ou props. Ele não deve ser armazenado com useState, pois isso cria redundância e possíveis inconsistências. Ao invés disso, ele deve ser derivado dinamicamente  
ex:

Errado:

```
const [total, setTotal] = useState(preco * quantidade)
```

Certo:

```
const total = preco * quantidade
```

Se preço ou quantidade mudarem, o total será recalculado automaticamente, sem necessidade de sincronizar manualmente.

## Estado global:

O estado global é aquele que precisa ser compartilhados entre vários componentes. Ele pode ser gerenciado por ferramentas como Context API, Redux ou Zustand, dependendo da complexidade da aplicação.

ex:

```
const CarrinhoContext = createContext()
```

```
function CarrinhoProvider({ children }) {
  const [itens, setItens] = useState([])
  return (
    <CarrinhoContext.Provider value={{ itens, setItens }}>
      {children}
    </CarrinhoContext.Provider>
  )
}
```

# TIPOS DE ESTADO - LOCAL, DERIVADO, GLOBAL E REMOTO

Estados globais devem ser usados com moderação, se tudo vira global, a aplicação perde isolamento e performance

## Estado remoto:

O estado remoto vem de uma fonte externa, como uma API ou banco de dados. Ele é assíncrono e pode mudar fora do controle da aplicação, por isso requer sincronização e tratamento de carregamento e erro

ex:

```
function Usuarios() {
  const [usuarios, setUsuarios] = useState([])
  const [loading, setLoading] = useState(true)

  useEffect(() => {
    fetch("/api/usuarios")
      .then(res => res.json())
      .then(data => setUsuarios(data))
      .finally(() => setLoading(false))
  }, [])

  if (loading) return <p>Carregando...</p>
  return <ul>{usuarios.map(u => <li key={u.id}>{u.nome}</li>)}</ul>
}
```

Em aplicações maiores, esse tipo de estado é melhor tratado com libs como React Query ou SWR, que automatizam cache, revalidação e sincronização

# FONTE DE VERDADE E ESTADO DERIVADO

A fonte de verdade é o único lugar do código onde um dado é realmente armazenado e atualizado. Todos os outros valores que dependem dele devem ser calculados a partir dessa origem, e não duplicados. Essa abordagem previne inconsistências, pois quando há uma única fonte de verdade, qualquer atualização é refletida automaticamente em toda a aplicação, sem risco de partes diferentes manterem informações divergentes.

Um erro comum é armazenar o mesmo dado em dois estados diferentes, por exemplo, manter tanto itens quanto total em useState, e atualizar o total manualmente quando os itens mudam. Isso cria um risco de descompasso, se os itens forem alterados sem atualizar o total, a interface mostra um valor incorreto.

O ideal é derivar o total dinamicamente com **const total = items.reduce((sum, item) => sum + item.price, 0)**; garantindo que ele sempre reflete o estado atual sem precisar de sincronização manual.

O estado derivado é exatamente isso: um valor que pode ser calculado a partir de outros estados ou props e, portanto, não precisa ser armazenado. Armazenar valores derivados gera re-renderizações desnecessárias e aumenta a complexidade de manutenção, pois o react precisará recalcular e reconciliar mais variáveis do que o necessário.

Em um exemplo de um cenário de carrinho de compras, é comum que o total seja calculado a partir dos itens adicionados, cada um com seu preço. Um exemplo errado seria declarar dois estados separados, um para a lista de itens (**useState[]**) e outro para o total **useState(0)**, atualizando manualmente o total sempre que um item é adicionado ou removido. Essa abordagem criaria duplicação de estado, o que abre caminho para inconsistências, pois a lista de itens e o total podem divergir.

A forma correta é tratar a lista de itens como a fonte de verdade e derivar o total dinamicamente, assim o total nunca está "defasado" em relação aos itens, não precisa de sincronização manual e mantém a UI alinhada com a lógica de dados.

Errado:

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

function CarrinhoErrado() {
  const [itens, setItens] = useState([]);
  const [total, setTotal] = useState(0); // estado duplicado!

  function adicionarItem(item) {
    setItens([...itens, item]);
    setTotal(total + item.preco); // atualiza manualmente o total
  }

  return (
    <div>
      <button onClick={() => adicionarItem({ nome: "Livro", preco: 50 })}>
        Adicionar Livro
      </button>
      <p>Total: R$ {total}</p>
    </div>
  );
}
```

# FONTE DE VERDADE E ESTADO DERIVADO

Nesse exemplo, o total está sendo armazenado e atualizado manualmente. Se algum item for removido ou alterado e setTotal não for chamado corretamente, o valor exibido ficará inconsistente com a lista de itens, ou seja, a fonte de verdade (itens) e o estado derivado (total) estão desalinhados.

Correto:

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

function CarrinhoCerto() {
  const [itens, setItens] = useState([]);

  function adicionarItem(item) {
    setItens([...itens, item]);
  }

  const total = useMemo(() => {
    return itens.reduce((soma, item) => soma + item.preco, 0);
  }, [itens]);

  return (
    <div>
      <button onClick={() => adicionarItem({ nome: "Livro", preco: 50 })}>
        Adicionar Livro
      </button>
      <p>Total: R$ {total}</p>
    </div>
  );
}
```

Aqui a fonte de verdade é apenas itens.

O total é calculado automaticamente a partir dessa lista, usando useMemo para evitar cálculos desnecessários a cada renderização, assim, não há risco de inconsistência, o total sempre reflete exatamente o estado atual do carrinho.

# SÍNCRONIZAÇÃO DE ESTADOS ENTRE COMPONENTES

A sincronização de estado entre componentes é um desafio comum, especialmente quando múltiplos componentes precisam refletir ou alterar o mesmo dado. Quando dois ou mais componentes compartilham um mesmo valor, o padrão mais seguro é o lifting state up, ou seja, mover o estado para o ancestral comum mais próximo e repassar o valor (e as funções de atualização) via props. Isso garante que todos os componentes dependam da mesma fonte de verdade, prevenindo inconsistências.

Prop Drilling é o termo usado em react para descrever o processo de passar dados (props) de um componente pai até um componente filho através dos vários níveis intermediários que não precisam realmente desses dados

Por exemplo, imagine que o App tem um dado user, mas quem precisa dele é um componente chamado UserProfile. Para que o UserProfile receba esse dado, ele precisa ser passado como prop por todos os componentes intermediários, mesmo que nenhum use o user, eles apenas o repassam

Ex:

[App lida com]



[Layout que lida com]



[Sidebar que renderiza]



[UserProfile]

A hierarquia é definida pela estrutura visual e lógica da interface

O prop drilling é aceitável em componentes pequenos ou quando a profundidade é baixa. Porém, quando o estado precisa ser acessado em muitas partes da aplicação, isso se torna difícil de manter.

(Um exemplo de informação que precisa ser acessado em múltiplas partes da aplicação é uma verificação de se o usuário está autenticado, pense no header (que exibiria o nome do usuário), na página do perfil (para mostrar as informações pessoais), no carrinho, (para vincular os itens ao usuário).

Se cada componente tentasse manter seu próprio estado de usuário ou recebesse o dado via props de forma manual, isso geraria muito prop drilling e inconsistências.

Nesse caso, o ideal é ter um estado global de autenticação, armazenado em um contexto (por exemplo, AuthContext), de onde qualquer componente pode acessar as informações do usuário diretamente, sem depender de múltiplas camadas intermediárias

Em muitos casos, vale a pena considerar uma solução global (como ContextAPI, Zustand, Redux) ou um padrão híbrido, onde parte do estado é global (dados com partilhados) e parte é local (controle interno do componente).

O React garante consistência automaticamente através de seu modelo declarativo e reatividade: Quando o estdo muda, ele dispara novas renderizações nos componentes dependentes, garantindo que a UI reflita sempre o estado atual sem atualizações manuais

# CONTEXT API - CONCEITO, ESCOPO E CUSTO DE RENDERIZAÇÃO

A context API é o mecanismo nativo do react para compartilhar estado global entre componentes sem precisar passar props manualmente por vários níveis da árvore (ou seja, sem prop drilling).

Ela funciona através de um context provider, que armazena um valor (por exemplo, o usuário autenticado, tema da interface, idioma) e de consumidores, que podem acessar esse valor de qualquer lugar da árvore onde o provider estiver acima.

O grande trade-off do context é entre simplicidade e performance: Ele é fácil de usar e ótimo para estados globais pequenos, mas cada atualização no valor do provider provoca re-render em todos os componentes que consomem aquele contexto, mesmo que apenas um deles realmente precise do valor atualizado. Isso pode impactar a performance em aplicações grandes.

Para mitigar esse problema, recomenda-se dividir contextos conforme o tipo do dado. Ao invés de um único contexto gigante para toda a aplicação (como um ApplicationContext com tema, idioma, usuário, carrinho), pode-se criar contextos menores e mais específicos (ThemeContext, UserContext, CartContext), garantindo que apenas os componentes relevantes sejam atualizados quando algo mudar.

A diferença entre context e ferramentas como ou Zustand está principalmente na escala e no controle do fluxo de dados. O context é uma solução leve e declarativa, ideal para compartilhar estado simples dentro da árvore react. Já redux, zustand e afins, oferecem gestão de estado mais robusta, com middlewares, persistência, time travel, imutabilidade e controle refinado de renderização, úteis em aplicações mais complexas, com muitos estados e interações assíncronas..

O escopo de um context define até onde na árvore de componentes o valor desse contexto é visível. Todo componente abaixo de um Provider tem acesso ao valor, mas nada acima ou fora dele. Por isso, é importante criar o contexto no nível mais baixo possível de hierarquia, onde ele ainda atende a todos os consumidores necessários.

Em resumo:

- **Escopo:** define onde o contexto é acessível
- **Granularidade:** define o quão específico e isolado é cada contexto
- **Objetivo:** Evitar re-renderizações em massa e manter o controle do estado global de forma eficiente

Exemplo de context:

Imagine uma aplicação de e-commerce com três grandes áreas

- Header, que mostra o nome do usuário logado
- Carrinho, que exibe a quantidade de itens
- Tema, que muda entre claro e escuro

Em um contexto mal escopado (granularidade ruim), o desenvolvedor cria um único ApplicationContext contendo o usuário, o tema e o carrinho, esse contexto envolve toda a aplicação;

Agora, quando o usuário adiciona um item no carrinho, todo o aplicativo re-renderiza (inclusive o Header e os componentes de tema, mesmo que eles não tenham relação com o carrinho)

Ou quando o tema muda, os componentes de autenticação também re-renderizam sem necessidade.

Isso acontece porque todos estão "ouvindo" o mesmo contexto, e qualquer alteração em qualquer parte dispara re-render em todos os consumidores

# CONTEXT API - CONCEITO, ESCOPO E CUSTO DE RENDERIZAÇÃO

Agora, em um contexto bem dividido (granularidade boa), o desenvolvedor cria três contextos separados:

- AuthContext - Responsável apenas por informações do usuário (login, nome, token)
- CartContext - Cuida apenas do carrinho de compras (itens, total, descontos)
- ThemeContext - Controla o tema claro/escuro

Nesse caso, se o usuário adiciona um item ao carrinho, somente os componentes que consomem o CartContext (como o ícone do carrinho e a página de checkout) re-renderizam

O header (que depende do AuthContext) e o tema (que depende do Theme context) continuam estáveis.

Esse exemplo mostra como o escopo (onde o contexto é usado) e a granularidade (o quanto ele é especializado) impactam diretamente na performance e na manutenção da aplicação.

# PATTERNS DE GERENCIAMENTO (LIFTING STATE UP, PROP DRILLING, CONTEXT SPLITTING)

Os patterns de gerenciamento de estado representam diferentes formas de organizar e compartilhar dados entre componentes React, dependendo da escala, complexidade e frequência de atualização do estado. A escolha correta evita acoplamento, duplicação e re-renderizações desnecessárias

## 1. Lifting state up

É o padrão mais simples, quando dois ou mais componentes precisam compartilhar o mesmo dado, o estado é movido (levantado) para o ancestral comum e passado como prop

Funciona bem em árvores pequenas, garantindo uma única fonte de verdade, mas pode gerar prop drilling se a árvore crescer demais.

Problema se mal planejado: excesso de props atravessando muitos níveis e perda de clareza sobre onde o estado é controlado

## 2. Prop drilling

É o ato de passar dados manualmente de um componente pai para um filho distante, atravessando intermediários que não usam o dado.

Acessível em estruturas rasas, mas torna-se difícil de manter em aplicações grandes, pois qualquer mudança na hierarquia exige ajustar várias props intermediárias

Problema se mal planejado: aumento da complexidade, acoplamento e maior risco de erros de sincronização

## 3. Context Splitting

Quando o context API é usado para evitar prop drilling, surge outro risco: o custo de re-renderização.

Contextos grandes, com muitos valores e consumidores, podem causar re-renders em toda a aplicação sempre que o valor muda.

A estratégia de context splitting é dividir o contexto em partes menores, cada uma cuidando de um domínio (ex: AuthContext, ThemeContext, CartContext).

Problema se mal planejado: Se for dividido demais, o código pode ficar fragmentado e difícil de compreender; se for dividido de menos, gera re-renders desnecessários

Como escolher o pattern adequado?

- Aplicações pequenas

Use lifting state e props (simples e previsível)

- Aplicações médias

Introduza Context API com divisão estratégica (context splitting)

- Aplicações grandes ou complexas

Adote gerenciadores externos como Redux, Zustand ou jotai, que isolam o estado global e otimizam performance

A escolha ideal é equilibrar simplicidade e escalabilidade: quanto mais compartilhado e dinâmico for o estado, mais você tende a migrar de lifting state up -> context -> gerenciador externo

# GERENCIAMENTO EXTERNO (REDUX, ZUSTAND, REACT QUERY)

O gerenciamento externo de estado é usado quando o estado da aplicação se torna muito compartilhado ou difícil de coordenar com lifting state up e context API.

Enquanto o estado local (useState, useReducer) vive dentro de um componente e pertence apenas a ele, o estado externo vive fora da árvore de componentes e pode ser acessado de qualquer lugar da aplicação

## Diferença conceitual entre estado local e externo:

O estado local é controlado diretamente dentro do componente e ideal para dados que só afetam ele (ex: abrir e fechar um modal, controlar um input)

Já o estado externo é centralizado em um único lugar (store) e serve para dados compartilhados globalmente (ex: usuário autenticado, tema do app, carrinho de compras)

O React continua sendo declarativo, mas a responsabilidade de armazenar e atualizar os dados passa para uma ferramenta externa que conversa com os componentes via hooks ou contextos internos

Quando vale a pena usar uma ferramenta externa?

- Quando muitos componentes precisam acessar o mesmo estado
- Quando há múltiplas fontes de autenticação
- Quando é necessário controle fino de performance
- Quando o projeto cresce e o context API se torna difícil de escalar ou manter

Em aplicações pequenas, ferramentas externas são um exagero, em médias ou grandes, elas trazem previsibilidade e organização

### • Redux

O redux segue o princípio de uma store global (única fonte de verdade). Ele trabalha com actions (intenções de mudança), reducers (funções puras que calculam o novo estado) e dispatch (mecanismo que dispara uma mudança); Seu foco é imutabilidade e rastreabilidade, cada mudança no estado é previsível e registrada, o que facilita debugs e testes, mas adiciona mais "cerimônia" e código.

Fluxo:

**Dispatch(action) -> reducer -> novo estado -> UI re-renderiza**

### • Zustand

O zustand oferece o mesmo conceito de store global, mas de forma muito mais simples e baseada em hooks.

Ao invés de actions e reducers, ele permite atualizar o estado diretamente dentro de uma função, sem boilerplate.

Seu grande diferencial é que somente os componentes que consomem o valor alterado re-renderizam, otimizando a performance

Exemplo de uso comum:

```
const { user, login } = useAuthStore();
```

# GERENCIAMENTO EXTERNO (REDUX, ZUSTAND, REACT QUERY)

- **React Query**

Diferente de redux e zustand, o react query não é para gerenciar UI state, mas server state, ou seja, dados vindos de uma API.

Ele lida com cache, refetching, sincronização automática e invalidação de dados de forma declarativa

Exemplo: Se o usuário adiciona um produto ao carrinho, o react query pode automaticamente refazer o fetch dos itens no servidor, mantendo a UI sempre sincronizada com o backend

Essas ferramentas não substituem o modelo declarativo do React, elas o complementam.

O react continua reagindo a mudanças de estado, mas o controle desse estado passa a estar fora dos componentes.

Em vez de o componente "gerar" o dado, ele apenas declara o que deve renderizar com base no estado atual da store externa.

# CACHE, SÍNCRONIZAÇÃO E INVALIDAÇÃO DE DADOS

Em aplicações que consomem APIs, é comum armazenar temporariamente os dados no cliente, esse é o processo chamado de cache. O objetivo é evitar requisições repetidas e melhorar a performance e a experiência do usuário. Por exemplo, se o usuário visitar uma lista de produtos e depois voltar para a mesma página, o app pode exibir os dados já armazenados em cache em vez de refazer o fetch imediatamente.

Entretanto, o cache traz um desafio: os dados podem ficar desatualizados. Por isso, é essencial definir estratégias de invalidação, que determinem quando e como o cache deve ser considerado obsoleto e atualizado novamente a partir do backend.

Algumas estratégias comuns incluem:

- Manual: O desenvolvedor decide explicitamente quando invalidar o cache (por exemplo, após criar ou editar um item, refazendo o fetch da lista)
- TTL (time to live): o cache expira automaticamente após um período de tempo configurado
- Otimistic updates: A UI é atualizada de forma imediata (otimista), assumindo que a operação no servidor vai dar certo, caso contrário, é revertida

Sem uma estratégia de invalidação, surgem problemas como:

- Dados inconsistentes: a UI exibe informações antigas que já mudaram no servidor
- Requisições redundantes: o app refaz fetches desnecessariamente aumentando o custo e o tempo de resposta
- Experiência confusa: O usuário pode ver dados diferentes em telas distintas ou ver o conteúdo "piscando" entre versões antigas e novas

Ferramentas como react query e swr resolvem esse problema automaticamente com um modelo reativo de cache e sincronização.

Elas mantêm um cache local estruturado e sabem quando precisam refazer o fetch, por exemplo, quando uma mutation ocorre, quando o usuário volta à tela, ou após um tempo de stale configurado. Além disso, gerenciam estados como loading error, e refetching sem necessidade de código manual.

Por fim, é importante a **diferença entre cache de UI e cache de servidor**:

- O cache de UI (como useMemo ou estados locais) serve apenas para otimizar renderizações dentro do React, evitando recomputações
- O cache de servidor (como o do react query ou SWR) armazena respostas de apis e gerencia sua validade e sincronização com o backend

O cache é uma ferramenta poderosa para performance, mas só é seguro quando acompanhado de uma boa política de invalidação e sincronização, garantindo que o usuário veja sempre dados atualizados, sem prejudicar a aplicação.

# SERVER STATE VS UI STATE

O **UI state (estado de interface)** representa tudo que existe apenas no cliente e não depende de um backend, como o estado de um modal aberto, a aba selecionada em uma interface. Ele é totalmente controlado pelo react e armazenado via useState ou useReducer, e só interessa à experiência do usuário.

Já o **server state**, representa dados vindos de uma fonte externa, geralmente uma API. Ele é assíncrono, pode mudar fora do controle da aplicação (por exemplo, se outro usuário alterar os dados no servidor), e precisa lidar com fetching, erro, cache e sincronização.

Exemplos: Lista de produtos, perfil de usuário, pedidos, mensagens

Tratar dados de API como se fossem puramente locais é um erro comum, por exemplo:

```
const [produtos, setProdutos] = useState([]);
useEffect(() => {
  fetch("/api/produtos").then(r => r.json()).then(setProdutos);
}, []);
```

Esse código até funciona, mas ele não lida com revalidação, erro, cache ou atualização externa.

Se outro usuário alterar os produtos no servidor, o seu estado local ficará defasado (stale).

Além disso, cada componente que fizer fetch repetirá a mesma lógica, desperdiçando requisições e piorando a performance.

Separar os dois corretamente traz algumas vantagens, como:

- Evita re-renders desnecessários
- Garante a sincronização automática com o backend (via cache e invalidação)
- Permite uma arquitetura previsível, onde o react lida com o fluxo da interface e a ferramenta de dados cuida da comunicação com o servidor

Ferramentas como react query e SWR são projetadas justamente para gerenciar server state, elas mantêm cache local, revalidam automaticamente quando o usuário volta à tela e garantem consistência entre o cliente e o servidor.

Enquanto isso, o UI stata permanece controlado por react puro (useState, useReducer), sem complexidade extra.

## Mas como fazer de maneira correta?

Segue um exemplo de gerenciamento de server state com react query

```
import { useQuery } from "@tanstack/react-query";

function ListaDeProdutos() {
  const { data, isLoading, error } = useQuery({
    queryKey: ["produtos"],
    queryFn: async () => {
      const res = await fetch("/api/produtos");
      if (!res.ok) throw new Error("Erro ao buscar produtos");
      return res.json();
    },
    staleTime: 1000 * 60 * 5, // 5 minutos de cache
  });
}
```

# SERVER STATE VS UI STATE

```
if (isLoading) return <p>Carregando...</p>;
if (error) return <p>Erro ao carregar produtos</p>

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

No primeiro exemplo de código nesse tópico, os dados do servidor foram tratados como estado local, usando useState e useEffect, para buscar produtos manualmente. Embora funcione, esse padrão ignora conceitos fundamentais do server state, como cache, revalidação e sincronização automática. Isso faz com que o estado local possa ficar desatualizado e gera código repetitivo em diferentes componentes.

Já no exemplo com react query, o server state é tratado de forma declarativa: a biblioteca lida com o ciclo de vida dos dados (fetch, erro, cache, revalidação) e mantém o estado sincronizado entre o cliente e o servidor. O componente não precisa mais controlar o fluxo manualmente, ele apenas declara que quer exibir os produtos, e o react query garante que os dados estejam atualizados. Assim, o react continua responsável pelo UI state enquanto o server state é abstruído por uma ferramenta que entende suas particularidades assíncronas e mutáveis. Essa separação deixa o código mais previsível, escalável e performático.

# ESTRATÉGIAS PARA EVITAR RE-RENDERIZAÇÕES DESNECESSÁRIAS

O react re-renderiza um componente sempre que seu estado interno muda ou quando ele recebe novas props

Essa re-renderização é parte do modelo declarativo, se mal controlada, ela pode causar lentidão perceptível, especialmente em árvores grandes ou componentes com cálculos pesados

## 1. Memoização: React.memo, useMemo, useCallback

A memoização evita que um componente ou cálculo seja refeito se suas dependências não mudaram

- O React.memo memoriza a renderização de um componente funcional, o react só re-renderiza se as props mudarem
- useMemo: evite recomputar valores caros entre renderizações, o cálculo só é refeito quando uma variável chave muda, não a cada renderização
- useCallback: memoiza funções, útil quando passada como props, sem isso, toda renderização criaria uma nova referência de função, o que poderia forçar re-renderizações de componentes filhos

## 2. Divisão granular de contexts

Quando um contexto muda, todos os componentes que o consomem re-renderizam, mesmo que só uma parte do dado tenha mudado.

Por isso, é importante dividir contextos grandes em múltiplos contextos menores, cada um cuidando de uma responsabilidade específica:

- Ao invés de um AppContext com user, theme e cart, crie:
  - UserContext
  - ThemeContext
  - CartContext

Assim, mudanças no carrinho não forçariam re-renderizações desnecessárias no header, por exemplo

## 3. Evitar atualizações de estado redundantes

Atualizações desnecessárias de estado são uma das principais causas de re-renders

- Evite setState com o mesmo valor
- Combine estados relacionados, ao invés de múltiplos useState, use um único objeto se eles sempre mudarem juntos
- Atualize no nível certo da árvore, use lifting state up apenas quando o estado realmente precisa ser compartilhado

## 4. Entendendo o custo das renderizações

Renderizar não é um problema, o problema é renderizar mais do que o necessário. O react é eficiente em reconciliar a árvore, mas se o componente executar cálculos pesados, manipular o dom manualmente ou passar muitas props, cada re-renderização pode se tornar cara

- Use o react DevTools profiles para identificar componentes que renderiam demais
- Prefira componentes puros e props simples
- Não abuse de useMemo e useCallback

## 5. Quando a memoização é desnecessária ou prejudicial

Memoizar tudo é um erro, usar useMemo e useCallback em operações simples como cálculos triviais ou funções curtas pode piorar a performance, já que o react precisa comparar dependências e armazenar as referências em memória

Só use memoização quando houver re-renderizações perceptíveis ou cálculos pesados.

# ESTRATÉGIAS PARA EVITAR RE-RENDERIZAÇÕES DESNECESSÁRIAS

## Como perceber sinais excessivos de re-render em uma aplicação grande?

Os principais sinais de que isso está acontecendo são:

- Lentidão perceptível na interface (lag ou travamentos)
- Logs de renderização aparecendo várias vezes sem motivo
- Re-renderizações em cascata (componentes filhos renderizando sem mudança de props)
- Atualizações globais disparando re-renders gerais
- Uso excessivo de estado em níveis alto da árvore

## Ferramentas que podem ajudar a identificar a baixa performance:

- React dev tools profiles: mostra graficamente o tempo de renderização de cada componente
- Why did you render: biblioteca que avisa no console sempre que um componente renderiza sem mudança real de props
- Console logs estratégicos, rápidos e eficazes para detectar re-renders locais

---

# MÓDULO 5

## RENDERIZAÇÃO E PERFORMANCE

Compreendendo o modelo de renderização e como o React otimiza o processo.

# CICLO DE RENDERIZAÇÃO E COMMIT DETALHADO

O processo de atualização de interface no react é dividido em duas fases principais: Render phase e commit phase

Essa separação torna o react previsível, eficiente e capaz de otimizar performance, especialmente com o fiber architecture

## 1.Render phase (fase de renderização )

É a fase pura e sem efeitos colaterais

- O react: executa o componente novamente (chamando sua função)
- Calcula o virtual DOM
- Compara com a versão anterior (diffing) para determinar o que mudou
- Cria uma lista de mudanças (mutations) que precisam ser aplicadas no DOM real

Nenhum acesso ou alteração no dom acontece aqui

Tudo acontece sem impactar a tela, é uma fase pura e interrompível.

## 2.Commit phase (fase de commit)

Depois de calcular o que precisa mudar, o react entra no commit

- Aplica as mudanças no DOM real
- Executa os efeitos colaterais (useEffect, useLayoutEffect)
- Atualiza refs e sincroniza o estado visual com o lógico

É o momento em que o usuário realmente vê a atualização na tela

## Por que a separação importa?

Essa divisão permite que o react:

- Priorize atualizações, por exemplo, uma digitação é mais importante que um carregamento de lista
- Evite bloqueios de renderização, ele pode pausar e reagendar cálculos longos
- Melhore a experiência do usuário, exibindo sempre o que é mais relevante primeiro

Os impactos dos side effects no commit

Hooks como useEffect e useLayoutEffect só são executados após o commit, pois eles interagem com o mundo externo (DOM, APIs, timers, etc).

Isso garante que:

- O DOM já esteja atualizado antes de rodar o efeito
- Não haja inconsistência entre o estado lógico e a tela

# IDENTIFICAÇÃO DE CAUSAS DE RE-RENDER

No react, um re-render acontece sempre que o componente precisa recalcular sua UI. Isso nem sempre é um problema, muitos re-renders são necessários e fazem parte do modelo declarativo.

O problema surge quando um componente re-renderiza sem que nada relevante tenha mudado, impactando performance

As três causas principais de re-render são

## 1. Mudança de estado local (`useState`, `useReducer`)

Sempre que o estado muda, o componente que o possui re-renderiza

## 2. Atualização via props vindas do componente pai

Se o pai re-renderiza, os filhos também re-renderizam por padrão. E se qualquer prop muda de referência, mesmo que o conteúdo seja igual, o filho re-renderiza

## 3. Context atualizado

Context é um "broadcast de re-renders"

Se um value muda, todo o componente que consome aquele contexto re-renderiza, mesmo que só use parte desse valor.

E um dos principais causadores de re-renders ocultos.

Padrões que reduzem re-renders

### 1. Componentes mais granulares

Dividir componentes em partes menores evita que um re-render suba ou desça demais pela árvore

### 2. Memoização seletiva

- `React.memo` reduz re-render baseado em props
- `useCallback` evita recriar funções toda hora
- `useMemo` evita recriar objetos/payloads caros

### 3. Contexts bem divididos

Separar contextos, `userContext`, `ThemeContext`, `CartContext`, etc e o mais importante, nunca colocar tudo em um único contexto gigante

### 4. Estado movido para o lugar certo

Quanto mais alto o estado está na árvore, mais re-render ele produz

Mover o estado para o componente que realmente precisar dele reduz re-renders colaterais.

# REATIVIDADE GRANULAR E ISOLAMENTO DE COMPONENTES

A ideia central é reduzir o custo de re-renderizações mantendo a UI reativa apenas onde é necessário. No React, um componente re-renderiza sempre que seu estado muda ou recebe novas props. Se toda interface estiver em um único componente grande, qualquer pequena atualização dispara re-renderizações em árvore, mesmo em partes que não mudaram.

Componentes isolados evitam isso: cada pedaço da UI é encapsulado e só re-renderiza quando precisa, isso melhora a performance, especialmente em interfaces grandes.

**Granularidade** é a prática de dividir a interface em componentes pequenos e focados. Por exemplo, em uma lista de produtos, cada item pode ser um componente isolado. Se apenas um item muda, apenas ele re-renderiza, não a lista inteira.

**Memoização** (React.memo, useMemo, useCallback) complementa esse isolamento, garantindo que os componentes ou cálculos pesados não sejam refeitos desnecessariamente.

Mas há de se considerar alguns trade-offs e pontos de atenção:

- Granularidade vs complexidade: componentes muito pequenos aumentam a complexidade da árvore e dificultam a manutenção.
- Isolamento reduz re-render global: melhora a performance, mas exige atenção ao design de props e estado.
- Exagero é contraproducente: quebrar tudo em unidades mínimas pode gerar overhead de gerenciamento de props e contextos, tornando o código menos legível.

Isolamento e granularidade são opções poderosas pra otimizar renderizações, mas precisam ser aplicadas de forma estratégica, equilibrando performance e complexidade da árvore de componentes.

# SUSPENSE E STREAMING DE DADOS (REACT 18+)

O suspense é um recurso que permite que partes da interface aguardem dados de carregamento de componentes sem bloquear a renderização completa da página.

Ao invés de mostrar uma tela em branco e travar a UI inteira, o React pode exibir um fallback\* (como um spinner ou skeleton) enquanto os dados ainda não estão prontos. Isso melhora significativamente a experiência do usuário, tornando a interface mais responsiva e progressiva

\*Fallback é um componente ou UI temporário enquanto o dado/componente assíncrono não está disponível)

Existem dois usos principais:

- Suspensa para código (lazy loading): usado para carregamento de componentes de forma assíncrona (`React.lazy`) permitindo que a árvore de componentes continue renderizando enquanto o módulo é baixado
- Suspense para dados: usando com fetching assíncrono de dados. Permite "suspending" a renderização de um componente até que os dados estejam disponíveis, sem bloquear o resto da UI

O streaming de dados complementa o suspense ao enviar pedaços da UI para o usuário conforme os dados ficam disponíveis, ao invés de esperar que tudo esteja pronto. Isso é especialmente útil em server components e ssr (server-side rendering), onde partes da página podem ser enviadas ao cliente assim que forem processadas

Impacto no ciclo de render e commit:

- Durante a render phase, componentes que dependem de dados assíncronos podem "susender" e devolver o fallback
- A commit phase aplica o que já está pronto no dom e, quando os dados chegam, novas renderizações são aplicadas automaticamente
- Isso mantém a UI responsiva e evita bloqueio, mas exige atenção ao gerenciamento de fallback e prioridade de carregamento

Algumas boas práticas são:

- Envolver apenas o que precisa suspender
- Usar fallbacks informativos e leves
- Combinar suspense com streaming pra SSR e server components
- Preferir caching e pré-fetch para reduzir tempo de espera

# LAZY LOADING E DIVISÃO DE CÓDIGO

Lazy loading é a técnica de carregar um recurso (componente, módulo, imagem, etc) apenas quando ele é realmente necessário, em vez de carregar tudo na inicialização do aplicativo.

No react, isso é feito geralmente com `React.lazy()` combinado com `<Suspense>`

Vantagens:

- Reduz o tamanho do bundle inicial
- Diminui o tempo de carregamento da primeira tela
- Melhora a performance, principalmente em aplicativos grandes ou complexos

Lazy loading de componentes no react:

```
import React, { Suspense } from "react";

const PainelAdmin = React.lazy(() => import("./PainelAdmin"));

function App() {
  return (
    <div>
      <h1>Minha Aplicação</h1>
      <Suspense fallback={<p>Carregando painel...</p>}>
        <PainelAdmin />
      </Suspense>
    </div>
  );
}
```

O `React.lazy` carrega o componente `PainelAdmin` somente quando ele precisa ser renderizado

O suspense mostra o fallback enquanto o componente é baixado

Divisão de código (code splitting)

- Automática: Webpack ou Vite detectam imports dinâmicos (`React.lazy`) e geram chunks separados do bundle principal
- Manual: você define pontos estratégicos para dividir o código, por exemplo, carregando apenas partes da UI em rotas específicas (`React Router` com `lazy routes`)

Exemplo com rotas:

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
const Home = React.lazy(() => import("./Home"));
const Sobre = React.lazy(() => import("./Sobre"));

function App() {
  return (
    <BrowserRouter>
      <Suspense fallback={<p>Carregando...</p>}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/sobre" element={<Sobre />} />
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
}
```

# LAZY LOADING E DIVISÃO DE CÓDIGO

Cada rota é carregada apenas quando o usuário navega para ela, reduzindo o bundle inicial

Lazy loading depende de suspense para mostrar fallbacks enquanto o componente baixa

Streaming (SSR) combina bem, pois a UI pode ser enviada em partes conforme o código e os dados ficam disponíveis

Essa abordagem estratégica traz alguns benefícios

- Ela resolve os problemas de bundle gigante, tempo de carregamento lento, experiência ruim em conexões lentas
- Resolve os problemas da experiência de navegação, o usuário vê a UI inicial rapidamente, enquanto as partes secundárias carregam em background
- Diminui o tempo de pintura (paint time), porque menos JS precisa ser parseado/executado na inicialização

# CONCURRENT RENDERING – SCHEDULING, INTERRUPÇÃO E PRIORIDADE

O react 18 introduziu a renderização concorrente, que **NÃO** é multithreading de verdade, mas faz o react \*parecer\* multitarefa: ele consegue pausar, interromper, descartar e retomar renderizações sem travar a interface

O que é o concurrent rendering?

Antes do react 18, a renderização era síncrona e bloqueante

- Se um componente demorasse 200ms para renderizar, a UI inteira ficava congelada por 200ms

Como funciona o concurrent rendering?

- A renderização é dividida em unidades menores (time slicing)
- Entre uma unidade e outra o React verifica se o navegador precisa responder a algo mais urgente
- Se houver algo mais prioritário, o react interrompe a renderização atual
- Quando possível, retoma a renderização de onde parou

Isso mantém a interface responsiva mesmo durante tarefas pesadas

O react não tem acesso direto ao event loop do javascript, mas utiliza mecanismos como MessageChannel e scheduling cooperativo para dividir o trabalho em pequenos blocos. Cada bloco é executado dentro do evento loop, permitindo que o navegador processe eventos de alta prioridades entre eles

No react 18, praticamente todas as atualizações são automaticamente "batchadas", isso significa que ao invés de causar renderizações separadas, o react junta tudo e causa uma única renderização acumulada.

Batching reduz o volume de renderizações enquanto o concurrent rendering reduz o custo de cada render longo

Blocking mode vs concurrent mode

## 1. Blocking mode

- Renderização síncrona
- Não há interrupção: O react só devolve o controle ao navegador ao final do render
- Grandes componentes ou cálculos pesados travam a UI
- Suspense funciona apenas para código (lazy loading)

## 2. Concurrent mode:

- Renderização cooperativa e interrompível
- O react pode pausar o render atual, descartar renders obsoletos, priorizar renders mais urgentes, retomar trabalhos iniciados
- Suspense funciona também para dados e se integra a server componentes

O concurrent mode cria uma UI mais estável sob cargas variáveis e melhora o tempo de resposta geral.

# INTERAÇÃO COM WEB VITALS E TEMPO DE PINTURA

As métricas de performance conhecidas como core web vitals (por exemplo, largest contentful paint - LCP, interaciton to next paint - INP, cumulative layout shift - CLS) medem pontos críticos da experiência real do usuário: carregamento, interatividade e estabilidade visual

No contexto de aplicações react, decisões arquiteturais de renderização tem impacto direto nessas métricas, porque definem quando e como a UI aparece, se está pronta para interação e se ela se move inesperadamente ou bloqueia a thread principal

## Como decisões de renderização afetam as métricas?

- Se um componente react realiza muitos cálculos ou atualizações pesadas antes de produzir o primeiro "paint", isso pode atrasar o LCP ou o tempo até a página ficar interativa (TTI/INP)
- Se a thread principal está ocupada em renderizações sincronizadas longas (bloqueantes), isso aumenta o tempo de resposta a input (aumentando a INP ou equivalentes)
- Se elementos visuais são adicionados ou reposicionados após a render inicial sem espaço reservado, isso gera deslocamentos de layout, impactando o CLS

## Influência de granularidade, memoização e isolamento:

- Dividir componentes grandes em unidades menores e isoladas permite atualizações menores que não forcem renderizações extensas. Isso reduz bloqueios e melhora o tempo até a interatividade
- Uso de React.memo, useMemo e useCallback reduz trabalho desnecessário no render, ajudando a manter a thread principal livre para tarefas de alta prioridade, favorecendo métricas como INP
- Garantir que os layout iniciais tenham dimensões reservadas e evitar alterações inesperadas após o carregamento ajuda a minimizar CLS

Quando o react usa técnicas como concurrent rendering, time-slicing e prioridade de tarefas, ele consegue priorizar updates que impactam diretamente o usuário (input, foco, visibilidade) em relação a tarefas menos urgentes (pré-carregamento, cálculos de fundo). Essa priorização reduz "tempo até o usuário poder interagir" e evita que o carregamento de funcionalidades secundárias bloquee aquilo que mais importa, isso se reflete nas métricas de interatividade e carga percebida

Como melhorar Web Vitals em aplicações React:

### 1. Otimize o bundle inicial

- Use code splitting em rotas e componentes pesados (React.lazy + Suspense)
- Remova dependências desnecessárias do bundle inicial
- Evite carregar gráficos, tabelas inteligentes, carrosséis e dashboards no primeiro paint
- Use importação dinâmica para módulos raramente usados

### 2. Evite renderizações pesadas no carregamento inicial

- Não coloque cálculos custosos dentro de useEffect, useMemo, ou diretamente no componente inicial da página
- Pré-calcule dados no servidor quando possível

### 3. Garanta layout estável (evite jumps)

- Sempre defina altura/largura para imagens, banners e componentes dinâmicos
- Reserve espaço para skeletons
- Não injete novos elementos acima do conteúdo já renderizado
- Evite alteração de fontes que causem relayout tardio

# INTERAÇÃO COM WEB VITALS E TEMPO DE PINTURA

## 4. Dívida com componentes grandes em unidades menores

- Múltiplos componentes pequenos são mais fáceis de interromper e retomar no concurrent rendering
- Minimiza re-renders propagados
- Reduz trabalho por renderização

## 5. Use memoização seletiva (apenas onde faz sentido)

- React.memo para componentes que recebem grandes objetos/arrays
- useMemo para cálculos ou objetos reutilizados
- useCallback para funções passadas como props
- Evite memoização excessiva (custo de comparação pode superar o ganho)

## 6. Use Context com granularidade adequada

- Context mal utilizado gera ripple re-renders em ávore
- Dívida contexts por domínio (Auth, theme, cart, UI)
- Evite usar context para valores atualizados com alta frequência

## 7. Prefira gerenciamento de estado externo para server state

- React query/SWR evitam múltiplos fetches, refetch redundante
- Cache eficiente -> menos trabalho na renderização inicial
- Evita recriar lógica assíncrona em cada componente

## 8. Evite atualização de estado redundante

- Não faça múltiplos setState para atualizar partes relacionadas
- Prefira reducers para transições agrupadas
- Evite "cálculo derivado" guardado em estado (duplicação)

## 9. Use Suspense para isolar carregamentos sem bloquear o restante da UI

- Coloque Suspense em volta apenas da parte que realmente espera por dados
- Evite envolver a página inteira em um suspense global
- Combine suspense com streaming (SSR) quando possível

## 10. Use transições (startTransition) para updates não-urgentes

- Para buscas, filtros, tabelas pesadas
- Transições dizem ao react: "isso não é urgente, priorize input"

## 11. Monitore real performance com ferramentas adequadas

- Chrome performance profiles
- Lighthouse
- Web Vitals JS library
- React profiler
- RUM (real user monitoring) em produção

---

# MÓDULO 6

## COMPOSIÇÃO E PADRÕES DE ARQUITETURA

Entendendo como estruturar  
componentes e lógica de forma escalável.

# PADRÃO DE COMPOSIÇÃO (PROPS.CHILDREN, RENDER PROPS, COMPOUND COMPONENTS)

No react, composição é a forma de construir interfaces reutilizáveis e flexíveis sem depender de herança. Esses padrões determinam como um componente recebe conteúdo ou comportamento externo, e cada abordagem resolve problemas diferentes de organização e escalabilidade.

## 1. props.children (composição direta)

O padrão mais básico é o uso de `props.children`. Ele permite que um componente receba conteúdo arbitrário dentro dele, tornando-o genérico e altamente flexível. É ideal para containers, layouts e wrappers que não definem a estrutura interna, apenas o "invólucro". Ele funciona como um slot que aceita qualquer JSX passado pelo componente pai.

Ex:

```
function Card({ children }) {
  return (
    <div style={{ padding: 20, border: "1px solid #ccc", borderRadius: 8 }}>
      {children}
    </div>
  );
}
```

Uso:

```
<Card>
  <h2>Produto</h2>
  <p>Descrição do produto...</p>
  <button>Comprar</button>
</Card>
```

## 2. Render props (função que define UI ou comportamento)

Render props são usadas quando o componente precisa expor lógica, mas permitir que o consumidor escolha o que renderizar. Nesse padrão, o componente recebe uma função como prop, e essa função recebe dados internos (como estado ou coordenadas do mouse) e retorna o JSX desejado.

Esse padrão permite que a lógica seja centralizada e compartilhada entre vários componentes, garantindo reutilização sem acoplamento visual. O consumidor tem total controle sobre a interface.

Render props surgiram como solução para compartilhar lógica antes de hooks customizados existirem. Em projetos modernos, raramente são necessárias, mas ainda fazem sentido quando o visual precisa ser completamente configurável, sem reescrever a lógica.

Ex:

```
function MouseTracker({ render }) {
  const [pos, setPos] = useState({ x: 0, y: 0 });

  function handleMove(e) {
    setPos({ x: e.clientX, y: e.clientY });
  }

  render();
}
```

# PADRÃO DE COMPOSIÇÃO (PROPS.CHILDREN, RENDER PROPS, COMPOUND COMPONENTS)

```
return (
  <div onMouseMove={handleMove} style={{ height: 200, background: "#eee" }}>
    {render(pos)}
  </div>
);
}
```

Uso:

```
<MouseTracker
  render={({ x, y }) => (
    <p>Posição do mouse: {x}, {y}</p>
  )}
/>
```

### 3. Compound components (componentes que trabalham em conjunto)

Compound components seguem o padrão onde componentes relacionados compartilham estado por meio de contexto interno. O componente pai controla toda a lógica (como qual tab está ativa) e os componentes filhos acessam esse estado automaticamente.

Esse padrão permite criar APIs expressivas e organizadas, como **<Tabs>, <Tabs.List>, <Tabs.Trigger>, <Tabs.Panel>**. O usuário monta a interface combinando essas partes de maneira declarativa, sem lidar com os detalhes internos.

Ele resolve problemas de escalabilidade ao criar componentes complexos extensíveis sem exigir dezenas de props, evitando config pesada e melhorando a legibilidade da API.

Ex:

Componente principal:

```
const TabsContext = createContext();
```

```
function Tabs({ children }) {
  const [active, setActive] = useState(0);

  return (
    <TabsContext.Provider value={{ active, setActive }}>
      <div>{children}</div>
    </TabsContext.Provider>
  );
}
```

Subcomponentes:

```
function TabList({ children }) {
  return <div style={{ display: "flex", gap: 10 }}>{children}</div>;
}
```

```
function Tab({ index, children }) {
  const { active, setActive } = useContext(TabsContext);
```

# PADRÃO DE COMPOSIÇÃO (PROPS.CHILDREN, RENDER PROPS, COMPOUND COMPONENTS)

```
return (
  <button
    onClick={() => setActive(index)}
    style={{ fontWeight: active === index ? "bold" : "normal" }}
  >
    {children}
  </button>
);
}
```

```
function TabPanel({ index, children }) {
  const { active } = useContext(TabsContext);
  return active === index ? <div>{children}</div> : null;
}
```

Uso:

```
<Tabs>
  <TabList>
    <Tab index={0}>Descrição</Tab>
    <Tab index={1}>Avaliações</Tab>
  </TabList>

  <TabPanel index={0}>Conteúdo da descrição</TabPanel>
  <TabPanel index={1}>Conteúdo das avaliações</TabPanel>
</Tabs>
```

## Como cada padrão resolve problemas diferentes?

- children resolve flexibilidade estrutural, o componente não precisa saber o que renderiza, é apenas um recipiente
- Render props resolve a personalização profunda de comportamento, a lógica é compartilhada, a UI é totalmente controlada pelo consumidor
- Compound components resolvem escalabilidade em componentes complexos. Eles possibilitam APIs ricas com múltiplas partes que funcionam juntas

Cada padrão trata um tipo específico de composição: conteúdo, comportamento ou estrutura conjunta.

A escolha depende da necessidade de controle: mínimo (children), médio (render props) ou máximo (compound components com estado compartilhado)

# INVERSÃO DE CONTROLE E ABSTRAÇÃO PROGRESSIVA

Inversão de controle e abstração progressiva são dois princípios fundamentais para criar componentes flexíveis, reutilizáveis e fáceis de manter no React. Eles evitam que um componente seja rígido demais ou acumule responsabilidades que pertencem ao consumidor

## **Inversão de controle é um princípio de design.**

Não é um hook do react, não é uma função específica, não é uma biblioteca

É uma forma de organizar componentes para que:

O componente deixe de decidir sozinho “o que fazer” e permite que algo externo decida isso.

### **1. Inversão de controle**

Inversão de controle é quando o componente deixa de tomar decisões internas e passa a permitir que o usuário do componente decida comportamento, lógica ou partes da UI.

Ao invés de impor regras rígidas, o componente expõe os pontos de tensão

Um componente sem IoC (rígido):

```
function Modal() {
  return (
    <div>
      <h1>Título fixo</h1>
      <p>Conteúdo fixo</p>
      <button>OK</button>
    </div>
  );
}
```

Não serve para quase nada além desse caso específico

Um componente com inversão de controle:

```
function Modal({ title, children, footer }) {
  return (
    <div>
      <h1>{title}</h1>
      <div>{children}</div>
      <div>{footer}</div>
    </div>
  );
}
```

Agora o consumidor controla:

- Conteúdo
- Comportamento
- Aparência
- Estrutura do footer

Isso é inversão de controle, o componente não dita as decisões, ele expõe a mecânica e o usuário decide o resto

# INVERSÃO DE CONTROLE E ABSTRAÇÃO PROGRESSIVA

Quando aplicar IoC melhora flexibilidade?

- Quando o componente tem muitas variações possíveis de layout
  - Quando você quer separar responsabilidade visual da lógica interna
  - Quando um componente deve ser genérico (Card, Modal, Button, List, FormField)
  - Quando evita dezenas de props específicas tipo showIcon, alignRight, etc
- IoC reduz expesão de props e aumenta a vida útil do componente

## 1. Abstração progressiva

Abstração progressiva é a filosofia de não tentar prever todas as necessidades futuras, mas sim criar componentes que podem ser estendidos quando necessário sem quebrar os existentes

É construir componentes que começam fáceis e podem ser estendidos conforme a complexidade aumenta, sem quebrar quem usa e sem multiplicar componentes.

A abordagem é:

- Criar o componente simples
- Permitir extensão conforme novas necessidades surgirem
- Evoluir a API sem forçar reescrita dos consumidores

Exemplo ruim (abstração antecipada demais -> over-engineering):

Criar um form genérico com schema, campos dinâmicos, passos, validações e 50 props antes mesmo de a aplicação precisar disso

Exemplo correto (progressivo):

Começa simples:

```
function Input({ label, ...props }) {
  return (
    <label>
      {label}
    <input {...props} />
  </label>
);
}
```

Se um dia precisar de controle de erro, você evolui:

```
function Input({ label, error, ...props }) {
  return (
    <label>
      {label}
    <input {...props} />
    {error && <span>{error}</span>}
  </label>
);
}
```

Se um dia precisar renderizar um ícone, se usa children ou slots como extensão natural.

Isso é abstração progressiva, crescer conforme a necessidade real, não por adivinhação

# INVERSÃO DE CONTROLE E ABSTRAÇÃO PROGRESSIVA

## Relação com SOLID no frontend

- Single responsibility (SRP):

Cada componente tem uma responsabilidade clara, UI, lógica, dados. IoC ajuda a manter isso, porque o componente não tenta fazer tudo

- Open/Closed (OCP)

Componentes devem estar abertos para extensão e fechados para modificação, exatamente o que children, slots e IoC permitem.

# COMPONENTES CONTROLADOS VS NÃO CONTROLADOS

Em formulários e inputs, o React permite duas abordagens: componentes controlados e não controlados. A diferença está em quem possui e atualiza a fonte da verdade do valor: o componente react ou o próprio DOM. Entender essa distinção é essencial para criar interfaces previsíveis, testáveis e com comportamento consciente.

## 1. Componentes controlados

Um componente controlado é aquele cujo valor é totalmente gerenciado pelo react. O input não "decide" seu próprio valor, ele apenas exibe o que o estado determina:  
Ex:

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

Aqui:

- A fonte de verdade é nome no estado React
- O input é apenas uma janela para esse estado
- Toda mudança passa por setNome

Vantagens

- Total previsibilidade: o React sempre sabe o valor atual
- Fácil de validar, formatar e mascarar entradas
- Excelente para formulários complexos
- Fácil de testar

Desvantagens

- Mais verboso
- Pode gerar re-renders frequentes em inputs muito dinâmicos
- Pode ser pesado em formulários grandes se não for bem otimizado

## 2. Componentes não controlados

Nos componentes não controlados, o DOM é a fonte de verdade.  
O React apenas lê o valor quando necessário, geralmente via refs

Ex:

```
function CampoNaoControlado() {
  const inputRef = useRef();
  function enviar() {
    console.log(inputRef.current.value);
  }
  return (
    <>
    <input ref={inputRef} defaultValue="João" />
    <button onClick={enviar}>Enviar</button>
    </>
  );
}
```

# COMPONENTES CONTROLADOS VS NÃO CONTROLADOS

Aqui:

- O valor é gerenciado internamente pelo DOM
- React não sabe o valor até você acessá-lo via ref.current.value

Vantagens:

- Simplicidade: menos estado, menos re-renders
- Bom para formulários muito grandes e pouco interativos
- Util quando não é necessário monitorar o valor em tempo real

Desvantagens:

- Menos previsível: o DOM pode ter valores que o react não conhece
- Validação e lógica de formatação ficam mais complexas
- Menos testável, pois depende de manipulação do DOM
- Pode quebrar a filosofia de fonte de verdade única

Um dos princípios centrais do React é ter uma única fonte de verdade para garantir consistência

- Componentes controlados seguem esse princípio: todo valor é definido por um único estado react
- Não controlados podem quebrar esse princípio: você tem duas fontes, o DOM e o estado React, podendo gerar inconsistências

Por isso, em sistemas com regras de negócio complexas, validações encadeadas, máscaras, sincronização com APIs ou formulários longos e interativos, a recomendação é usar componentes controlados

# PATTERNS DE ISOLAMENTO: CONTAINER/PRESENTATIONAL, SMART /DUMB

Esse padrão organiza componentes separando o que mostra a UI (camada visual) do que contém lógica e dados (camada inteligente). Ele surgiu muito antes dos hooks personalizados, mas continua extremamente útil conceitualmente para pensar em organização, responsabilidades e escalabilidade

## 1. Presentational components (Dumb components)

São componentes focados exclusivamente em exibir informação

- Não possuem estado complexo
- Não fazem fetch de dados
- Não conhecem regras de negócio
- Só sabem renderizar o que recebem

Eles funcionam como "componentes de UI puros"

Exemplos típicos: cards, botões estilizados, caixas de produtos, listas visuais, modais sem lógica

## 2. Container components

São responsáveis pela parte inteligente da aplicação

- Lidam com estado
- Fazem requisições de API
- Decidem o que mostrar
- Processam cálculos
- Orquestram interações

Eles recebem os dados "crus" e passam para os presentational já formatados

Containers cuidam de como os dados chegam

Presentamentals cuidam de como os dados aparecem

### Por que isso melhora a escalabilidade?

Quando você separa lógica e UI:

#### 1. Aumenta a reutilização:

A mesma UI pode ser usada em múltiplos contextos diferentes, porque ela não está acoplada a um tipo específico de estado ou API

#### 2. Facilita a manutenção

A lógica fica toda em um único lugar (o container)

Se amanhã houver mudanças nas regras, você altera o container, não todas as telas

#### 3. Código fica mais claro

Componentes visuais são simples e previsíveis, componentes inteligentes são organizados e concentrados

#### 4. Testabilidade melhora

A UI pode ser testada visualmente e por snapshot e a lógica pode ser testada sem renderizar interface

# PATTERNS DE ISOLAMENTO: CONTAINER/PRESENTATIONAL, SMART /DUMB

## Prop Drilling e esse padrão

Quando você separa um container de vários presentational components, pode surgir o problema de prop drilling, que é quando você precisa passar informações por muitos níveis intermediários

Ex:

Um componente inteligente obtém dados e precisa repassar esses dados para um componente visual que está vários níveis abaixo. Os componentes intermediários acabam recebendo props apenas para passar adiante

Como evitar prop drilling nesses casos?

- Mover o estado mais próximo dos componentes que realmente o usam
- usar Context para estados compartilhados
- Extraír a lógica para hooks customizados, reduzindo dependência da árvore

## Container/Presentational vs Hooks customizados

O padrão container/presentational foi, por muitos anos, a forma principal de separar responsabilidades no react: um componente container concentra toda a lógica (estado, requisições, regras de negócio) enquanto um componente presentational é responsável apenas pela UI, recebendo dados e callbacks como props. Essa abordagem organiza bem a aplicação, mas aumenta a quantidade de componentes e modifica a estrutura da árvore, já que sempre há um container envolvendo a interface. Com a chegada dos hooks, especialmente os custom hooks, essa separação deixou de depender de componentes adicionais, agora a lógica pode ser isolada em funções reutilizáveis, independente de hierarquia visual. Um custom hook extrai e encapsula comportamentos, não estruturas, permitindo que qualquer componente use aquela lógica sem precisar de um container intermediário.

Enquanto containers isolam a lógica por componente (criando uma camada estrutural), os custom hooks isolam a lógica por meio do comportamento (criando uma camada funcional). Por isso são considerados a evolução natural do padrão container, oferecendo os mesmos benefícios de organização, mas com menos acoplamento, menor profundidade na árvore e muito mais flexibilidade na reutilização

# PROP DRILLING E PATTERNS ALTERNATIVOS

**Prop drilling** é o padrão em que um valor precisa ser passado por múltiplos níveis de componentes até chegar ao lugar onde realmente será usado, mesmo quando os componentes intermediários não utilizem aquele dado; eles apenas o repassam.

Esse padrão não é um erro por si só, ele emerge naturalmente do modelo de composição do React, mas pode se tornar problemático à medida que a árvore de componentes cresce e mais camadas precisam participar desse repasse. Em pequenas estruturas, prop drilling é absolutamente aceitável e até desejável, pois mantém a visibilidade explícita de onde os dados entram e para onde vão, o que facilita a leitura e reduz a "mágica" no fluxo de dados.

O problema surge quando muitos componentes são obrigados a carregar props que não fazem parte da sua responsabilidade, aumentando o acoplamento, dificultando refatorações e gerando re-renderizações desnecessárias em partes da árvore que não deveriam sequer participar da atualização

A decisão entre aceitar o prop drilling ou substituí-lo por outro padrão envolve equilibrar simplicidade e acoplamento. Em muitos casos, introduzir uma solução "global" apenas para evitar três níveis de props é um exagero. Porém, quando o dado é realmente compartilhado por diversas partes da aplicação (como o usuário autenticado, preferências de tema, carrinho ou permissões), o prop drilling rapidamente se torna inviável. Nesse momento, alternativas como context API, hooks customizados ou padrões estruturais surgem naturalmente. O objetivo não é eliminar o drilling por estética, mas evitar que responsabilidade e dependência se espalhem artificialmente pela árvore

**Context API** é a alternativa mais direta ao prop drilling, ao invés de passar os dados manualmente entre cada componente intermediário, o valor é disponibilizado por um Provider e acessado por qualquer consumidor dentro daquele escopo. Isso reduz a necessidade de props intermediárias, mas traz um custo: qualquer atualização do valor do contexto re-renderiza todos os consumidores. Esse impacto precisa ser considerado, especialmente em contextos com valores muito dinâmicos. Nesse sentido, dividir contextos por domínio e manter o contexto o menor possível é a principal estratégia para reduzir re-renderizações colaterais

**Composition** é outra forma de minimizar drilling sem introduzir globalidade. Muitas vezes, permitir que um componente receba children ou funções (render props) evita que dados precisam subir e descer arbitrariamente na árvore; Em outras palavras, composição reduz a distância entre o lugar onde o dado está e onde ele é utilizado, reorganizando a estrutura para que o drilling nem seja necessário

Os **HOCs** (Higher-order-componentes), embora menos usados hoje, também resolvem drilling ao injetar dados diretamente no componente sem que os níveis intermediários precisem trazê-los. Contudo, esse padrão introduz camadas extras e perde de clareza estrutural, por isso foi amplamente substituído por hooks customizados, que oferecem o mesmo benefício com menos acoplamento e sem modificar a árvore de componentes.

**Hooks customizados**, por sua vez, não eliminam o drilling estruturalmente, mas extraem a lógica que antes ficava presa ao topo e permitem que qualquer componente localize e consuma apenas o estado necessário, reduzindo tanto o acoplamento quanto a necessidade de passar dados manualmente

# PROP DRILLING E PATTERNS ALTERNATIVOS

No fim, prop drilling é aceitável quando o escopo é pequeno, mas deve ser evitado quando começa a distorcer a estrutura da aplicação. As alternativas (context, composition, HOCs e hooks customizados) existem para resolver problemas diferentes e devem ser escolhidas com base no impacto real na manutenção, no isolamento de responsabilidades e na performance.

# PATTERNS DE EXTENSIBILIDADE (SLOT PATTERN, HOCs CONCEITUALMENTE)

Em interfaces complexas, é comum que componentes precisem ser estendidos, personalizados ou combinados com comportamentos adicionais. O React, por adotar composição ao invés de herança, oferece dois padrões principais de extensibilidade: slot pattern e HOCs. Ambos resolvem problemas diferentes e possuem trade-offs importantes em termos de previsibilidade, testabilidade e manutenção.

O **slot pattern** deriva diretamente da ideia de composição. Ao invés de expor dezenas de props específicas (como headerComponent, leftIcon, actions, footerButtons, etc) o componente expõe áreas customizáveis (slots) onde o consumidor pode injetar qualquer conteúdo. É como dar ao usuário pontos fixos de encaixe, mantendo a estrutura geral, mas permitindo personalização profunda. Esse padrão elimina configuração excessiva por props e torna o componente naturalmente extensível. Um modal, por exemplo, pode definir slots para título, corpo e ações. O padrão é ideal quando o componente é essencialmente estrutural e precisa permitir variação visual sem perder a coerência da API.

Já os **HOCs** são um padrão mais antigo e poderoso, mas hoje menos usado, cuja função é acrescentar comportamento, não estrutura.

Um HOC é uma função que recebe um componente e retorna outro, adicionando lógica, estado ou side effects. Antes dos hooks, essa era a principal forma de compartilhar lógica entre componentes. Padrões como conectar componentes a uma store global ou adicionar detecção de scroll, resize ou permissões eram implementados como HOCs. Eles ainda são úteis quando você precisa estender múltiplos componentes com o mesmo comportamento sem modificar a hierarquia nem a UI interna. HOCs têm desvantagens conhecidas: aumentam a profundidade da árvore, escondem o fluxo de props, dificultam debugging e podem gerar "wrapper hell". Por isso, foram amplamente substituídos por custom hooks, que compartilham lógica sem alterar a estrutura ou criar camadas artificiais.

Ao decidir entre slot pattern e HOC, a pergunta central é: **você precisa adicionar flexibilidade estrutural ou comportamento reutilizável?** Slots resolvem personalização de UI, permitindo extensões visuais previsíveis. HOCs resolvem compartilhamento de comportamento, mas com mais custo de manutenção, por isso, hooks customizados são quase sempre preferíveis no react moderno.

Em termos conceituais, o slot pattern representa composição, o coração da filosofia do React: componentes pequenos, previsíveis e combináveis. Já os HOCs são uma forma de herança funcional, pois criam componentes derivados, adicionando capacidades por "envolvimento" em outra função. Embora não seja herança tradicional, a relação pai-filho fica menos explícita, o que pode comprometer previsibilidade e testabilidade. Slot pattern é mais transparente, fácil de debugar, intuitivo para outros desenvolvedores e naturalmente alinhado com a manutenção de longo prazo. HOCs são poderosos, mas opacos, e devem ser reservados para cenários específicos onde hooks customizados não bastam (como integração com APIs externas de terceiros que expõe HOCs).

Em síntese, use composition (slot pattern) sempre que estiver lidando com estrutura e personalização de UI, use hooks customizados para lógica, use HOCs apenas quando não houver alternativa funcional mais moderna.

# PATTERNS DE EXTENSIBILIDADE (SLOT PATTERN, HOOKS CONCEITUALMENTE)

Exemplos:

1. Slot Pattern

Card com slots (Header, body, footer)

```
function Card({ children }) {
  return (
    <div style={{ border: "1px solid #ccc", borderRadius: 8, padding: 16 }}>
      {children}
    </div>
  );
}

function CardHeader({ children }) {
  return <div style={{ marginBottom: 8 }}>{children}</div>;
}

function CardBody({ children }) {
  return <div style={{ marginBottom: 8 }}>{children}</div>;
}

function CardFooter({ children }) {
  return <div style={{ marginTop: 16, textAlign: "right" }}>{children}</div>;
}
```

Uso:

```
<Card>
  <CardHeader>
    <h2>Produto: Tênis</h2>
  </CardHeader>

  <CardBody>
    <p>Descrição do produto...</p>
  </CardBody>

  <CardFooter>
    <button>Comprar</button>
  </CardFooter>
</Card>
```

2. Custom Hooks (padrão moderno pra lógica compartilhada)

```
function useContador(inicial = 0) {
  const [valor, setValor] = useState(inicial);

  function incrementar() {
    setValor(v => v + 1);
  }

  return { valor, incrementar };
}
```

# PATTERNS DE EXTENSIBILIDADE (SLOT PATTERN, HOC'S CONCEITUALMENTE)

```
function decrementar() {
  setValor(v => v - 1);
}

return { valor, incrementar, decrementar };
}
```

Uso:

```
function Produto() {
  const { valor: qtd, incrementar } = useContador();
  return <button onClick={incrementar}>Adicionar ({qtd})</button>;
}

function Likes() {
  const { valor, incrementar } = useContador();
  return <p onClick={incrementar}>Likes: {valor}</p>;
}
```

3. HOC

```
function withLogger(Componente) {
  return function Wrapper(props) {
    useEffect(() => {
      console.log(`Montou: ${Componente.name}`);
      return () => console.log(`Desmontou: ${Componente.name}`);
    }, []);
  }
}
```

Uso:

```
function Painel() {
  return <h1>Painel</h1>;
}

const PainelComLog = withLogger(Painel);

// <PainelComLog /> agora loga automaticamente
```

# PRINCÍPIOS DE DESIGN COMPONÍVEL

O design componível é a base arquitetural do React. Mais do que uma escolha estética, é uma filosofia que determina como os componentes devem ser estruturados para permanecerem fáceis de entender, previsíveis, reutilizáveis e escaláveis. Em aplicações grandes, a qualidade do design dos componentes tem impacto direto na produtividade da equipe, na facilidade de manutenção e até na performance. Esse design parte de três pilares fundamentais: coesão, previsibilidade e isolamento, todos derivados de princípios clássicos como Single responsibility, composição em vez de herança e separação de efeitos colaterais.

O princípio de coesão diz que um componente deve ter uma responsabilidade clara e limitada. Ele não deve ser ao mesmo tempo responsável por buscar dados, processá-los, renderizar UI, aplicar formatação, validar entrada e gerenciar animações. Componentes que acumulam responsabilidades violam esse princípio e se tornam difíceis de testar, difíceis de modificar e frágeis diante de mudanças. Um sinal de violação é quando o componente cresce demais, possui várias seções independentes, exige dezenas de props ou tem lógica desconexa misturada com UI.

A previsibilidade vem de componentes que expõe APIs claras (props), controlam estado internamente quando necessário e evitam alterar estado ou efeitos de maneira implícita. Componentes que realizam side effects durante o render, dependem de valores globais ocultos ou mudam comportamento baseado em condições internas são difíceis de compreender e testar. O react reforça essa previsibilidade ao separar render e commit, mas o desenvolvedor precisa isolar os efeitos colaterais em useEffect e manter a UI sempre baseada em estado declarativo e estável.

O princípio de isolamento determina que cada parte da UI deve re-renderizar apenas quando aquilo que ela depende mudou. Isso evita re-renderizações desnecessárias e reduz acoplamento entre partes de uma interface. Isolamento também significa separar lógica de domínio (hooks, reducers, loaders) da camada visual (componentes presentacionais), além de evitar dependências globais ou contextos mal utilizados que disparem re-renders em massa.

## Identificar violações desses princípios é relativamente simples:

- Componentes grandes demais, com múltiplas funções internas
- Muitos condicionais alterando visual e lógica de forma misturada
- Props que não são claras ou que carregam múltiplas funções
- Necessidade frequente de "consertar" comportamento interno com novos flags como isDisabled, compact, variantA, variantB (sinal de API mal projetada)
- Dependência excessiva de contexto ou estado global para coisas que poderiam ser locais

Os trade-offs entre generalização e simplicidade aparecem quando tentamos criar componentes flexíveis demais. Generalizar cedo demais leva a APIs enormes, repletas de props específicas, caindo em over-engineering. Simplicidade extrema pode gerar duplicação ou componentes pouco reutilizáveis. A solução ideal é abstração progressiva: começar simples, evoluir conforme novas necessidades surgem e permitir extensibilidade por composição (children, slots, compound components) ao invés de tentar adivinhar todos os usos futuros

Combinar **composição, abstração e isolamento** adequadamente resulta em componentes altamente previsíveis, fáceis de estender e manter, sem comprometer simplicidade. É isso que faz o react escalar bem em sistemas complexos: cada componente faz pouco, mas faz bem (e todos se encaixam como peças claras e articuladas)

---

# MÓDULO 7

# REACT MODERNO E SERVER

# COMPONENTS

Entendendo a nova arquitetura e a  
separação entre client e server  
components.

# SERVER COMPONENTS - O QUE SÃO E POR QUE SURGIRAM

Os server components representam uma mudança profunda no modelo de renderização do React. Eles foram criados para resolver gargalos de performance que surgiram conforme aplicações frontend cresceram e passaram a carregar cada vez mais código javascript no navegador. A ideia central é simples, mas poderosa: executar componentes React inteiros no servidor, enviando ao cliente apenas o HTML ou um formato serializado leve, evitando que o navegador baixe, parseie execute Javascript desnecessário. Com isso, o React distribui responsabilidades de forma mais inteligente (lógica pesada e acesso a dados ficam no servidor, apenas interatividade e comportamentos dinâmicos permanecem no cliente).

O principal problema que os server components resolvem é o peso do bundle. Em uma aplicação tradicional (client-rendered), todo componente escrito em React (até mesmo aqueles que só exibem dados estáticos ou fazem fetch) precisa ser enviado ao cliente como Javascript. Isso gera bundles grandes, alto tempo de download, parsing lento e bloqueio da thread principal. Ao mover parte desses componentes para o servidor, o React reduz drasticamente o volume de código enviado ao navegador. Em algumas aplicações reais (especialmente Next.js com RSC), é comum cortar o bundle em 30 a 60%, melhorando LCP, TTI e consumo de memória. Além disso, como o servidor executa código sem limitações do ambiente do browser, pode acessar banco de dados, arquivos, APIs internas e fazer processamento pesado sem penalizar a experiência do usuário.

A diferença fundamental entre Server components (RSC) e client components é a **natureza da execução**.

Server components **não tem interatividade e não possuem estado local imperativo** (useState, useEffect, refs). Eles são essencialmente funções puras que rodam no servidor e retornam UI já pronta. Não existe hidratação nem listeners no cliente para eles. Já client components dependem do navegador para gerir estado, efeitos e eventos, precisam de javascript enviado ao cliente, precisam ser hidratados e podem interagir diretamente com o DOM. A maneira moderna de pensar é: tudo começa como Server component, só vira Client component se precisar.

O impacto no tempo de carregamento inicial é direto. Quando o navegador recebe uma página com server components ele não precisa esperar o javascript da aplicação ser baixado e executado para exibir o conteúdo. O HTML já chega pronto, compatível com o que o usuário deveria ver. E como a porção de Javascript enviada é menor, tanto o download quanto o parse ficam significativamente mais rápidos. Isso melhora LCP e reduz o tempo até a página ficar utilizável.

Na prática, server components criam um modelo híbrido: parte da árvore React roda no servidor, parte roda no cliente, com boundaries bem definidas. Um client components pode importar um server component, mas não ao contrário, isso reforça a arquitetura de responsabilidade clara: lógica pesada no servidor, interatividade no cliente. Isso leva a uma UI mais rápida, bundles mais leves e uma arquitetura mais alinhada com aplicações modernas que combinam server e client de forma transparente

Server components são o padrão ideal para tudo que não depende do navegador:

- Renderização de páginas
- Listagem de produtos
- Carregamento de posts
- Montagem de tabelas estáticas

# SERVER COMPONENTS - O QUE SÃO E POR QUE SURGIRAM

- Menus
- Headers
- Footers
- Qualquer conteúdo que possa ser pré-renderizado ou trazer dados direto do banco/servidor sem custo no cliente

Eles reduzem drasticamente o bundle e chegam ao navegador já como HTML pronto. Já os client components entram apenas quando há interatividade real:

- Botões que abrem modal
- Inputs controlados
- Formulários com validação em tempo real
- Menus dropdown animados
- Carrosséis
- Componentes que usam useState/useEffect
- Leitura de localStorage
- Uso de window/document

Em um e-commerce, por exemplo, o produto, a descrição, avaliações e preço devem ser server components, enquanto o carrinho flutuante, botão comprar, seletor de quantidade, favoritos e busca em tempo real obrigatoriamente viram client components. Isso cria uma divisão clara: o servidor monta a página e entrega tudo pronto, o cliente só ativa o que precisa de interação humana

Para criar um **server component**, você simplesmente escreve o componente normalmente, sem incluir "use client" no topo. Ele será executado no servidor, não terá acesso a APIs do navegador e não poderá usar hooks como useState ou useEffect. Esse é o comportamento padrão do React moderno: tudo é server a menos que você peça o contrário.

Para criar um **client component**, você precisa colocar a diretiva "use client" na primeira linha do arquivo. Isso instrui o React a enviar esse componente para o navegador, permitindo uso de interatividade, hooks de estado, efeitos, acesso ao window, document, eventos e animações. Sem essa diretiva, o componente não é tratado como client-side, mesmo que você tente usar APIs do navegador. A regra mental é simples: se precisa de interatividade, declare como client, se não precisa, deixe como server.

# SSR, SSG, CSR E HIDRATAÇÃO

A renderização de aplicações web modernas pode ocorrer em diferentes etapas e em diferentes lugares, no servidor, no cliente ou em ambos. SSR, SSG, CSR e hidratação representam estratégias distintas para equilibrar performance, interatividade e SEO, e entender como cada uma funciona é fundamental para projetar aplicações react eficientes, especialmente em conjunto com server components.

**CSR (Client-side rendering)** foi o modelo dominante no react por muitos anos. Nele, o navegador recebe praticamente um HTML vazio e um bundle javascript grande. Só depois de baixar, interpretar e executar o JS é que a página aparece e fica interativa. É simples, poderoso para SPAs, mas causa problemas como time-to-first-paint lento e SEO ruim em páginas públicas. SSR e SSG surgiram para resolver exatamente essas limitações.

**SSR (Server-side rendering)** renderiza o HTML no servidor a cada requisição. O cliente recebe uma página já montada (texto, estrutura, conteúdo) e vê algo imediatamente. Depois disso, o React hidrata esse HTML, conectando os componentes interativos no navegador. SSR melhora muito o SEO, já que os robôs recebem HTML completo, e reduz o tempo até o primeiro paint para o usuário. É ideal para páginas dinâmicas mas públicas, como produtos, blogs com conteúdo frequente, dashboards iniciais, homepages personalizadas e páginas que dependem de dados atualizados por request. A principal desvantagem é que cada requisição exige trabalho do servidor, aumentando custo, latência e necessidade de infraestrutura mais forte.

**SSG (Static site generation)** gera o HTML em build time, antes da aplicação ser publicada. Cada página vira um arquivo estático extremamente rápido, entregue quase instantaneamente por CDN. É perfeito para conteúdo que muda pouco: documentação, blog estático, landing pages, marketing, páginas institucionais. Como o HTML já está pronto, o tempo de render é praticamente zero. A limitação é óbvia: SSG não é adequado para conteúdo altamente dinâmico, a menos que seja combinado com técnicas como revalidação, incremental static regeneration ou mistura com SSR.

Tanto SSR quanto SSG dependem do processo de hidratação. Hidratação é o momento em que o React pega aquele HTML estático com o que seria produzido pelo render do cliente, e se tudo bate corretamente, ele "anexa" a interatividade sem reciar o DOM do zero. Se houver divergências, o React pode descartar parte do DOM e renderizar novamente, criando layout shifts. Por isso é essencial garantir que o HTML do servidor e do cliente sejam determinísticos.

Sobre quando SSR ou SSG são mais vantajosos que CSR, a resposta depende do equilíbrio entre interatividade imediata, SEO e dinamismo dos dados. SSR é mais indicado quando o conteúdo muda frequentemente, mas precisa ser indexado e exibido rapidamente. SSG é imbatível para sites com conteúdo estável. CSR só faz sentido quando a aplicação é essencialmente interativa e não depende de conteúdo inicial renderizado rapidamente, como dashboards internos após login, painéis em tempo real ou ferramentas puramente SPA.

SSR e SSG têm impacto direto na experiência do usuário: melhoram first paint, LCP, estabilidade visual e SEO. O custo é maior complexidade, necessidade de hidratação e potencial aumento do trabalho do servidor. Já CSR simplifica a infraestrutura, mas sacrifica carregamento inicial e indexabilidade. Uma arquitetura moderna combina tudo, gerando aplicações mais rápidas, escaláveis e coerentes.

# TRANSMISSÃO DE DADOS E BOUNDARIES ENTRE CLIENT SERVER

A arquitetura moderna do React, especialmente quando combinada com server components e frameworks como Next.js, introduz uma separação clara entre o que roda no servidor e o que roda no cliente. Essa separação é chamada de boundary: um ponto explícito onde um componente deixa de ser server e passa a ser client (ou vice-versa), e onde dados precisam atravessar essa fronteira. Entender esses limites é crucial para evitar trabalho desnecessário, melhorar performance e manter previsibilidade no fluxo de dados.

Um boundary client/server existe quando um server component precisa renderizar (ou envolver) um client component. Isso acontece porque componentes cliente possuem capacidades que server components não têm: interatividade, uso de estado local dinâmico, hooks como useEffect, acesso ao DOM e APIs do navegador. O boundary funciona como um ponto de transição, o servidor monta a interface estática até aquele ponto e, dali em diante, delega responsabilidade ao cliente. É importante manter esses boundaries o mais alto possível na árvore: quando mais no topo estiver um client component, menor a quantidade de código que cai no bundle do navegador. Quanto mais alto ele estiver, mais partes da árvore perdem a vantagem da renderização server-only. Por isso, o design moderno recomenda deixar a maior parte da UI como server components e isolar interatividade apenas nos locais necessários.

A diferença entre estado server-only e estado client-only decorre desse boundary. Estado server only inclui tudo que depende do backend, consultas ao banco, leitura de arquivos, dados sensíveis, cálculos pesados, sincronização entre requisições ou passos que não podem ocorrer no navegador. Esse estado existe apenas durante a renderização no servidor; ele não é preservado entre interações, pois server components são sempre puros e sem estado persistente. Já o estado client-only representa interatividade: inputs, toggles, temas, abertura de modal, animações, navegação local, campos do formulário, carregamento incremental, seleções do usuário. Ele é mantido no navegador e precisa de client components. Separar corretamente esses tipos de estado evita bugs estruturais e reduz drasticamente a quantidade de javascript enviada ao cliente.

Um dos grandes riscos ao conectar server e client é o **over-fetching** (trazer dados do servidor mais vezes que o necessário ou em locais onde não são usados) e a duplicação de dados, quando fetches separados repetem consultas idênticas ou mantêm versões conflitantes do mesmo estado. Em uma arquitetura híbrida, o ideal é centralizar o fetching em server components, pois eles rodam no servidor com acesso direto e eficiente a bancos, caches e autenticação. O cliente deve receber apenas os dados processados que precisa, geralmente via props, evitando fetch redundante no browser. Quando mais de um client component necessita do mesmo dado, o boundary deve estar acima deles, garantindo que o fetch ocorra apenas uma vez. Ferramentas como cache de server actions ou react cache ajudam a garantir consistência e minimizar duplicações;

## Definir boundaries ideais significa colocar:

- No servidor tudo que envolve dados, processamento, lógica pesada, autenticação ou o que não requer interação
- No cliente, apenas o que exige estado dinâmico, eventos do usuário ou APIs do navegador

# TRANSMISSÃO DE DADOS E BOUNDARIES ENTRE CLIENT SERVER

Essa separação reduz bundle, melhora tempos de carregamento, simplifica o fluxo de dados e mantém a aplicação estável. O segredo é não misturar responsabilidades: equilíbrio entre as partes e boundaries bem definidos criam aplicações mais rápidas, previsíveis e sustentáveis.

# SUSPENSE FOR DATA FETCHING E STREAMING

Suspense e streaming representam uma das maiores mudanças conceituais no React moderno, especialmente quando combinados com Server Components. Eles alteram profundamente o fluxo de renderização ao permitir que partes da interface esperem por dados sem bloquear toda a tela, ao mesmo tempo em que tornam possível enviar fragmentos da UI ao cliente conforme ficam prontos. Isso melhora o tempo de resposta percebido e reorganiza completamente como pensamos sobre carregamento, fetch de dados e experiência do usuário.

Tradicionalmente, em React client-side, o componente iniciava o fetch dentro de um `useEffect`, exibia um loading e só então renderizava os dados. Esse padrão bloqueava e renderização útil e fazia com que o usuário visse flashes de carregamento, layouts incompletos ou conteúdos reposicionados, gerando uma experiência fragmentada. Suspense quebra esse modelo: ao invés de esperar os dados depois da renderização começar, o React é capaz de detectar que algo ainda não está disponível e interromper a renderização apenas naquele ponto, exibindo um fallback enquanto o restante da árvore continua sendo processado normalmente. O resultado é fluidez, a UI não trava, não congela e não fica dependente de uma única requisição.

A diferença entre suspense para código e suspense para dados é conceitualmente importante. Suspense para código já existe há mais tempo e é usado para divisão de bundles: componentes carregados dinamicamente via `React.lazy` são "suspenso" até o código ser carregado. Já suspense para dados, introduzido com a arquitetura de server components e novas APIs de fetch integradas, permite que o React trate dados da mesma forma que tratava código: como algo que pode atrasar a renderização apenas do trecho dependente dele. Isso significa que componentes podem ser escritos declarativamente, sem hooks de efeito, e ainda assim esperar dados de forma integrada ao próprio mecanismo de renderização.

O **streaming** entra como complemento dessa ideia. Quando a renderização ocorre no servidor, ela pode ser dividida em partes: assim que um fragmento da UI está pronto, ele é enviado ao navegador imediatamente, mesmo que outros trechos ainda estejam aguardando dados. Isso é especialmente poderoso quando há componentes que dependem de fetches lentos (eles não bloqueiam a página inteira). O usuário começa a ver o conteúdo quase instantaneamente, e apenas os trechos dependentes exibem fallback até seus dados estarem disponíveis.

## Esse modelo resolve problemas profundos de UX e performance:

- Evita "tela branca" durante carregamentos longos, pois parte da UI já é entregue antes mesmo do fetch terminar
- Remove a necessidade de múltiplos loadings espalhados pelo código
- Reduz o custo cognitivo, pois o usuário percebe a página carregando progressivamente
- Melhora muito métricas como LCP, pois conteúdos não bloqueados aparecem antes
- Minimiza shifts de layout, já que os fallbacks são posicionados de maneira previsível

Com server components, suspense se torna ainda mais natural: o servidor pode resolver dados antes de enviá-los e sinalizar ao cliente quais partes ainda estão "penduradas". A interatividade do cliente só é hidratada quando necessário, criando um fluxo híbrido altamente otimizado. Em síntese, suspense e streaming transformam a experiência de carregamento em algo fluido, progressivo e arquiteturalmente integrado ao React, não mais um detalhe manual tratado por componente.

# LIMITAÇÕES E PADRÕES EMERGENTES NEXT.JS 14-15, REACT 19

Os server components introduzem um novo paradigma no React, mas junto com os benefícios surgem restrições claras e novos padrões arquiteturais que ainda estão se estabilizando, especialmente em frameworks como Next.js e no react 19. Entender essas limitações e os caminhos emergentes é crucial para tomar decisões de design consistentes e evitar armadilhas comuns.

A primeira limitação importante é que server components não podem usar APIs que dependem do ambiente do navegador. Isso inclue useState, useEffect, useLayoutEffect, useRef para valores mutáveis, APIs do DOM, listeners de eventos e qualquer coisa que dependa de interatividade direta. Um server component precisa ser puro: ele recebe dados, processa e retorna UI estática. Isso coloca limites claros sobre o que pode ou não ser escrito no servidor. Da mesma forma, qualquer lógica que dependa de identidade ou sincronia com a sessão do navegador precisa ser migrada para components client-side ou tratada por ações do servidor.

Outro ponto crítico é que server components rodam em um ambiente com acesso ao filesystem, banco de dados, APIs seguras e variáveis protegidas, o que é uma vantagem enorme. Porém, essa capacidade exige que você mantenha uma boundary clara entre o que pode ser executado no servidor e o que precisa ser entregue ao cliente. Misturar responsabilidades pode gerar desperdício de rede, aumento no bundle ou "vazamento" acidental de lógica para o client. Por isso, frameworks modernos estimulam a organização explícita entre módulos server e client, garantindo que cada parte execute apenas onde faz sentido.

Com next.js 14, o modelo de server components foi consolidado, mas ainda coexistia com padrões híbridos legados (como getServerSideProps e getStaticProps). Já no Next.js 15, a direção é mais clara: quase tudo começa como server component e o desenvolvedor precisa optar explícitamente por client-side apenas quando necessário usando "use client". Além disso, a integração com server actions se tornou mais profunda, permitindo mutações e form handling diretamente no servidor, reduzindo a dependência de APIs REST internas ou endpoints duplicados. Isso simplifica a arquitetura, mas exige disciplina ao projetar boundaries: estados interativos residem no client, dados, mutações e carregamento pesado ficam no server.

React 19 reforça esses padrões trazendo APIs mais coesas entre server e client. Hooks como use, que permite esperar por promessas diretamente em componentes, e mecanismos de caching server-side integrados tornam a arquitetura mais fluida e menos baseada em artifícios. Além disso, streaming progressivo em conjunto com suspense se torna o caminho mais predominante para lidar com latência, substituindo gradualmente modelos tradicionais de loading.

Essas mudanças mudam profundamente as decisões de design. Como parte significativa da lógica pode ser movida para o servidor, componentes passam a ser menores, mais simples e mais declarativos. O fluxo de dados deixa de ser controlado apenas por hooks e passa a ser dirigido por boundaries explícitas. Além disso, padrões antigos como fetch em useEffect, torna-mse obsoletos em favor de buscas diretas no servidor, com cache automático e invalidação via ações.

# LIMITAÇÕES E PADRÕES EMERGENTES NEXT.JS 14-15, REACT 19

Por fim, a arquitetura híbrida moderna exige escolher conscientemente onde cada parte vive. O objetivo não é "fazer tudo no server", mas sim colocar cada responsabilidade no lugar certo, renderização pesada e acesso a dados no servidor, interatividade, animações e estados efêmeros no cliente. Essa separação melhora performance, reduz bundle e aumenta previsibilidade, mas exige que o desenvolvedor compreenda as limitações e as ferramentas emergentes que sustentam esse novo modelo.

# O FUTURO DA RENDERIZAÇÃO NO REACT

O futuro da renderização no react já está se desenhando, e ele gira em torno de três grandes eixos: componentização distribuída entre client e server, renderização concorrente como base do runtime moderno e streaming progressivo como padrão de entrega de UI. Esse conjunto de mudanças redefine o papel do react: de uma biblioteca focada exclusivamente no cliente para um sistema completo de renderização universal, onde o servidor assume um papel ativo na criação da interface e o cliente se torna um ponto final mais leve, mais rápido e mais interativo.

**A primeira mudança conceitual** a antecipar é que componentes não serão mais homogêneos. A ideia de "todo componente React é igual e roda no browser" já está ultrapassada. Futuros apps React serão divididos em duas categorias fundamentais: Server components, responsáveis por buscar dados, renderizar grandes partes da UI e reduzir o bundle e Client Components, focados apenas em interatividade real (inputs, animações, eventos lógica visual).

Essa divisão exige pensar menos em "componente isolado" e mais em fluxos entre servidor e cliente, onde dados "descem" como HTML ou payloads serializados e o cliente só ativa o que é necessário. Essa abordagem reduz drasticamente o peso enviado ao navegador, melhora TTFB, LCP e torna aplicações grandes mais escaláveis.

**O segundo pilar** é a consolidação do concurrent rendering, que se torna a base do runtime. Em vez de renderizações síncronas e bloqueantes, o React passa a priorizar atualizações, interromper tarefas longas e reagendar renderizações mais pesadas para manter a UI sempre responsiva. Para apps futuros, isso significa projetar componentes com granularidade adequada, minimizar re-renders e aproveitar mecanismos como transições (startTransition) e Suspense. A concorrência deixa de ser um detalhe técnico para se tornar parte do design: componentes precisam ser "interrompíveis" e resilientes a renderizações múltiplas

**O terceiro eixo** é o streaming progressivo, já adotado em SSR com server components e suspense. Em vez de esperar tudo para renderizar a página, o servidor envia partes prontas da UI conforme os dados chegam (e o cliente já pode começar a interagir enquanto o restante ainda está sendo carregado). Isso aproxima o react daquilo que o HTML sempre fez bem (streaming natural via server-push)), mas agora com toda a inteligência do modelo declarativo. Para o desenvolvedor, isso significa que o fluxo tradicional "fetch -> loading -> render" desaparece e dá lugar ao modelo "render enquanto carrega", o que melhora UX e reduz percepção de lentidão.

Essas transformações impactam profundamente performance, bundle e UX. O bundle tende a diminuir, pois mais lógica fica no servidor. A performance melhora porque o cliente executa menos javascript, hidrata menos elementos e interage com menos componentes complexos. A UX se torna mais fluida graças ao concurrent rendering e ao streaming. Mas esses ganhos exigem que o desenvolvedor domine conceitos como boundary de components, arquitetura híbrida, suspense, acche no servidor, transições e isolamento de estado.

## Todos os conceitos estudados até agora tornam-se críticos:

- Fonte de verdade clara (estado no lugar certo: server vs client)
- Componentes coesos e bem isolados (para suportar concorrência e streaming)
- Estados derivados e lógico-arquiteturais eficientes (evitar duplicação entre server e client)

# O FUTURO DA RENDERIZAÇÃO NO REACT

- Memoização criteriosa e granularidade adequada (para minimizar custo no cliente)
- Composição e padrões de extensibilidade (para construir árvores híbridas saudáveis)

Em resumo, o futuro do react não é mais simplesmente renderizar no browser, mas orquestrar onde cada parte da aplicação deve viver para otimizar experiência e performance. Quem dominar esse novo fluxo estará preparado para construir aplicações modernas, escaláveis e alinhadas com a próxima geração do ecossistema React.

---

# MÓDULO 8

# FILOSOFIA DE ENGENHARIA

# FRONTEND

Consolidando a mentalidade e os  
princípios de decisão técnica em React.

# TRADE-OFFS: ABSTRAÇÃO VS CLAREZA

A relação entre abstração e clareza é um dos dilemas centrais da engenharia frontend moderna. Embora abstrações sejam necessárias para organizar sistemas complexos, elas também têm um custo: podem esconder a intenção original do código, dificultar leitura e aumentar o esforço cognitivo de quem mantém a aplicação. Em React, esse trade-off aparece em praticamente todas as decisões arquiteturais, desde criar um componente genérico, extrair um hook ou introduzir um contexto global, até definir a hierarquia de estados. O ponto fundamental é entender que abstrações não são um objetivo em si: elas devem resolver um problema real de repetição, acoplamento ou complexidade, jamais nascer apenas por "elegância" ou tentativa de prever cenários futuros.

Criar uma abstração é realmente necessário quando **existe repetição clara e recorrente**, quando a lógica está espalhada de forma fragmentada ou quando múltiplos componentes dependem da mesma regra de negócio. A abstração deve eliminar duplicação sem esconder o que está acontecendo. Uma boa heurística é perguntar "Se eu voltar nesse código daqui a 3 meses, essa abstração vai me ajudar ou atrapalhar a entender essa intenção?". Se a resposta for incerta, provavelmente a abstração não deveria existir ainda. Muitas vezes, é melhor repetir a lógica duas ou três vezes até que o padrão fique óbvio; só então extrair um hook, um componente composto ou um útil. Premature abstraction é trâo prejudicial quanto premature optimization: ambas criam complexidade sem retorno garantido.

Avaliar se uma abstração simplifica ou complica começa observando o fluxo mental necessário para entender o que ela esconde. Um componente altamente genérico pode exigir dezenas de props e condicionais para dar conta de múltiplos cenários, tornando o uso mais difícil do que escrever diretamente a UI específica. O mesmo ocorre com hooks que encapsulam lógica demais, tornando impossível entender de onde vem um estado ou qual é a causa de determinada atualização.

**Abstrações devem deixar o código mais previsível** e aumentar o alinhamento entre intenção e implementação; se a leitura exige saltar por vários arquivos, navegar por múltiplas camadas ou decodificar uma API complexa, a abstração provavelmente falhou.

Esse trade-off aparece frequentemente na decisão entre usar um hook customizado ou manter lógica local no componente. Um Hook é útil quando reduz repetição, isola regras de domínio e simplifica o componente. Mas quando cria uma "caixa preta" difícil de entender, a clareza é sacrificada. O mesmo ocorre com context: embora elimine prop drilling, seu uso indiscriminado pode causar re-renderizações amplas e esconder a origem do estado. Em muitos casos, passar uma prop manualmente é mais simples e mais explícito. Já padrões de composição como compound components ou slot pattern funcionam bem quando a estrutura visual é realmente compartilhada, quando usados para cenários que não pedem esse nível de abstração, resultam em APIs mais confusas do que componentes diretos.

No fim, abstração e clareza não são opostas, são forças que precisam ser **equilibradas**. Uma boa abstração revela a intenção, reduz ruído e organiza o sistema; uma abstração ruim introduz opacidade e dificulta manutenção. O critério não é "abstrair porque é mais avançado", mas "abstrair quando torna o comportamento mais claro, previsível e simples de evoluir"

# TRADE-OFFS: ABSTRAÇÃO VS CLAREZA

Um exemplo claro de abstração aplicada em um e-commerce seria a criação de um módulo de cálculo de frete.

Imagine que várias partes da aplicação precisam calcular o frete:

- Página de produto
- Carrinho
- Checkout
- Página de comparação de preços
- Sugestão de frete em um mini-carrinho no header

Se cada componente implementasse sua própria lógica, consultar CEP, consultar transportadoras, aplicar regras de frete grátis, calcular prazo, aplicar descontos regionais, etc. **A lógica acabaria copiada, divergente e difícil de manter.**

Agora vem a abstração: você cria um único mecanismo que concentra toda a lógica de frete e fornece apenas o resultado necessário. Essa abstração poderia ser representada conceitualmente como um "serviço de frete" ou um "módulo de regras de logística".

O benefício é que:

- A regra de frete se concentra em um único lugar
- Todos os pontos da interface recebem resultados consistentes
- Componentes continuam simples e focados apenas em exibir dados

# DECISÕES GUIADAS POR INTENÇÃO, NÃO POR FERRAMENTA

Decisões guiadas por intenção significam projetar a interface a partir do que o **usuário precisa** e do que o componente deve expressar, e só depois escolher ferramentas, hooks ou padrões. Em outras palavras: primeiro entende-se o propósito, depois escolhe-se o mecanismo. Isso evita que a arquitetura seja movida por modismos ("usar context para tudo", "colocar redux porque grandes empresas usam", "transformar tudo em hooks customizado") e mantém o foco em clareza, performance e manutenção.

Para identificar a intenção central, a pergunta mais importante é: "O que esse componente realmente precisa fazer e qual a responsabilidade dele?"

Se a intenção é apenas exibir os dados estáticos, adicionar estado ou efeitos desnecessários piora o design. Se o objetivo é sincronizar dados entre múltiplos componentes, usar variáveis de estado locais quebra a consistência. A intenção sempre revela quais ferramentas são adequadas e, principalmente, quais não são.

Muitos erros comuns no react surgem de **decisões tomadas por hábito, não por necessidade**. Um exemplo clássico é usar context para qualquer dado compartilhado, mesmo quando apenas dois componentes precisam dele; isso aumenta re-renderizações e dificulta manutenção. O outro hábito ruim é criar hooks customizados para lógicas simples que poderiam permanecer diretamente no componente, resultando em fragmentação desnecessária no código. Ou ainda, aplicar memoização abusiva em tudo "porque melhora performance", quando na verdade, o custo de comparação e complexidade adicional supera o benefício.

Decisões guiadas por intenção são radicalmente diferentes. Por exemplo: se a intenção é carregar produtos de um e-commerce com cache e revalidação, faz sentido usar React query, não porque "é moderno", mas porque ele resolve exatamente esse fluxo.

Se a intenção é sincronizar o tema escuroclaro da aplicação, usar context é apropriado, pois é um estado global de baixa frequência de atualização.

Se a intenção é manter o carrinho acessível de qualquer área, um store global (zustand, redux) faz sentido, porque resolve um problema de compartilhamento profundo.

Da mesma forma, se a intenção é aumentar legibilidade, às vezes o melhor é não abstrair: um componente simples, com código explícito, vale mais que uma arquitetura sofisticada. Isso mostra que boas decisões não surgem de uma caixa de ferramentas, mas da interpretação cuidadosa do problema. Em React, a maturidade técnica está menos em "saber todas as features" e mais em saber quando NÃO usar uma feature.

# EVOLUÇÃO DE CÓDIGO E CONSISTÊNCIA EM EQUIPE

A evolução de código em uma aplicação react depende diretamente da consistência e clareza nas decisões arquiteturais da equipe. Conforme o projeto cresce, múltiplos desenvolvedores passam a tocar as mesmas áreas da base de código, e aquilo que antes parecia simples rapidamente vira um emaranhado de estilos diferentes, padrões misturados e soluções conflitantes, reduzindo atrito cognitivo e evitando que cada desenvolvedor tenha que "reaprender" a arquitetura a cada novo arquivo. Ao mesmo tempo, a equipe precisa preservar espaço e flexibilidade, permitindo soluções melhores quando surgem necessidades novas, e esse equilíbrio é parte essencial do trabalho de engenharia.

O equilíbrio entre consistência e flexibilidade surge quando a equipe estabelece **padrões claros** onde importa, e mantém margem de manobra onde não há impacto estrutural. Por exemplo, definir padrões de organização de pastas, convenções de nomeação, como os hooks são criados, quando usar context vs zustand vs estado local, tudo isso reduz variabilidade e evita soluções duplicadas. Por outro lado, importar regras rígidas para cada detalhe (ex: "todo componente deve ser dividido em container e presentational, "todo estado deve usar reducer") gera desenvolvimento engessado e aumenta a curva de entrada. O guia deve ser: consistência para o essencial, flexibilidade para o incidental.

Algumas decisões arquiteturais reduzem muito a dúvida técnica no longo prazo. Entre elas, adotar um padrão claro de fluxo de dados (ex: estado global apenas quando necessário; evitar cascatas de contexts, preferir UI state local), separar regras de negócio em hooks customizados, definir boundaries entre camadas (UI, data fetching, utilidades), aplicar memoização só quando há necessidade real, e manter componentes pequenos e previsíveis. Outro ponto crítico é padronizar estruturas como Server components vs Client components em projetos modernos, decidir quando usar cada um e documentar a regra evita digergências que, no futuro, são difíceis de corrigir.

A documentação de patterns internos é o elo invisível que mantém a equipe alinhada. Ela não precisa ser extensa, precisa ser viva, clara e consultável. Um bom formato é manter um engineering playbook dentro do repositório com tópicos como: "Como criar um novo componente?", "Como organizar pastas?", "Quando criar hooks customizados?", "Padrão de nomes para estados e callbacks", "Como lidar com server components?", "Quando usar memoização?". Além disso, revisão de código (code review) funciona como mecanismo contínuo de reforço: cada PR é uma oportunidade de padronizar, explicar e aprender. Linter e formatadores (ESLint, Prettier) também eliminam debates desnecessários, eles impõem regras automáticas, liberando a equipe para discutir arquitetura e não estilo.

No final, **consistência não deve sufocar criatividade**, ela deve reduzir fricção. Quando os padrões são claros, novos desenvolvedores entendem rapidamente como contribuir, quando há flexibilidade suficiente, ideias melhores conseguem emergir sem quebrar o sistema. A evolução do código depende dessa harmonia, uma base sólida que acolhe mudanças sem se tornar caótica.

# DESIGN ORIENTADO A PREVISIBILIDADE E LEGIBILIDADE

Um design orientado à previsibilidade e legibilidade parte da premissa de que o **código deve deixar claro o que faz e como se comporta**, sem exigir que o leitor adivinhe intenções ocultas ou percorra múltiplos arquivos para entender um fluxo simples. Em aplicações react, isso significa trabalhar com componentes cujos nomes descrevem seu papel, cuja árvore de render é intuitiva e cujo comportamento é consistente independentemente do contexto. Previsibilidade reduz bugs porque reduz a superfície de surpresa: quando um componente segue padrões claros de estado, efeitos e props, o desenvolvedor consegue antecipar como ele reagirá a mudanças, e isso torna a depuração muito mais rápida.

Avaliar se um componente é previsível envolve observar se ele possui uma única responsabilidade clara, se não mistura múltiplas camadas de lógica (UI, efeitos, networking, cálculos de negócio), se não contém efeitos colaterais implícitos e se não depende de estados externos que o leitor não consegue identificar facilmente. Um componente previsível é aquele que pode ser lido de cima a baixo sem "travas cognitivas": o leitor entende onde está o estado, onde está a lógica, o que vem de ora e o que pertence a ele. Se for difícil explicar o que o componente faz em uma frase simples, há um problema de legibilidade.

Side effects mal isolados são um dos maiores inimigos da previsibilidade. Quando efeitos estão espalhados sem justificativa, quando usam dependências implícitas, quando causam re-renderizações inesperadas ou quando alteram estados externos sem deixar isso claro, a legibilidade do componente diminui drasticamente. Isso força o desenvolvedor a raciocinar sobre o tempo (quando algo acontece) ao invés de apenas ler o que acontece, quebrando o modelo mental declarativo do React. Efeitos não deveriam ser usados para derivar estado, sincronizar valores redundantes ou "consertar" algo após o render, isso ria fluxos não determinísticos e difíceis de testar. Quando os efeitos são necessários (busca de dados, subscrições, sincronização externa), eles devem ser explícitos e isolados.

Para tornar comportamento e estado mais previsíveis, algumas estratégias são essenciais. Primeiro, manter estado o mais local possível: quanto mais alto na árvore ele estivar, mais componentes ele afeta e mais difícil é rastrear suas ligações. Segundo, evitar estados duplicados e valores derivados armazenados como estado, previsibilidade vem de um único ponto de verdade. Terceiro, manter efeitos colaterais controlados, bem nomeados e limitados às dependências estritamente necessárias. Quarto, padronizar props: nomear callbacks como onSubmit, onSelect, onClose torna a semântica uniforme, nomear valores como value, items, isOpen, permite ao leitor entender rapidamente o papel de cada prop. Por fim, criar componentes puros sempre que possível: sem side effects, sem dependências externas, apenas funções que recebem props e retornam UI.

Componentes puros são naturalmente previsíveis.

**Legibilidade e previsibilidade não são "embelezamentos"**, são pilares que determinam a velocidade de evolução de uma aplicação. Código previsível facilita onboardings, reduz divergências entre membros da equipe, minimiza bugs originados por mal-entendimentos e cria uma base sustentável para escalar tanto a UI quanto a arquitetura de dados. É um investimento estrutural que produz retornos contínuos ao longo de toda a vida útil do sistema.

# PENSAMENTO SISTÊMICO NO FRONTEND

Pensar sistematicamente no frontend significa deixar de enxergar cada componente, hook ou tela como entidades isoladas e **passar a tratá-las como partes interdependentes de um ecossistema maior**. Nesse modelo mental, qualquer decisão, seja sobre estado, composição, fetch de dados ou layout, produz efeitos colaterais que se propagam pela aplicação inteira. A perspectiva deixa de ser "como faço esse componente funcionar?" e passa a ser "como essa escolha afeta performance, renderizações, UX e a evolução do sistema?". É uma mudança de postura técnica: do foco local para o foco global.

Quando adotamos pensamento sistêmico, antecipamos problemas de performance e UX porque entendemos de antemão onde o custo real será pago. Um exemplo simples: mover um estado 3 níveis acima na árvore pode parecer uma boa solução local, mas sistêmica mente significa aumentar o número de re-renderizações em componentes que não precisavam ser atualizados. Da mesma forma, decidir que cada página deve buscar seus próprios dados sem coordenação pode parecer correto do ponto de vista do componente, mas sistêmica mente leva a over-fetching, latência acumulada e inconsistências de cache. Pensar como um sistema que permite prever gargalos antes que se tornem problemas perceptíveis: saber que um contexto global causa re-renderizações amplas, que um loop de dependências em efeitos pode bloquear o thread principal, que cálculos de layout podem gerar instabilidade visual, ou que um fetch redundante degrada métricas como time to interactive. A antecipação nasce da compreensão das relações, não dos elementos isolados.

O design de componentes está diretamente ligado ao fluxo de dados global. Componentes bem isolados, com responsabilidades claras e limites explícitos, facilitam a circulação de dados sem acoplamento excessivo. Por outro lado, componentes "inteligentes demais" ou que combinam UI, lógica e networking criam pontos rígidos no sistema: qualquer alteração nesses blocos centrais produz cascatas de re-renderizações e amplia a complexidade cognitiva. Quando o sistema é visto em camadas (UI, lógica de domínio, estado global, estado local, server state) torna-se claro onde cada tipo de dado deve residir e como ele deve fluir. Essa organização reduz atrito, aumenta previsibilidade e diminui o risco de correções indiretas ("mexer aqui quebra ali").

Otimizar renderizações com pensamento sistêmico significa enxergar a árvore inteira como uma **unidade dinâmica** e não apenas como um agrupamento de componentes. Isso envolve aplicar granularidade adequada (não exagerada, reduzir re-renderizações globais movendo estado para o menor escopo possível, dividir contextos com alta taxa de atualização, usar memoização seletiva onde os valores realmente são estáveis e reorganizar fluxos de dados para minimizar cálculos custosos). Também significa considerar técnicas como server components, streaming e lazy loading não como ferramentas isoladas, mas como mecanismos de redistribuição de custo ao longo do ciclo de vida da interface. Em vez de otimizar um componente específico, o objetivo passa a ser otimizar o comportamento emergente do sistema inteiro, priorizando fluidez, redução de acoplamento e estabilidade perceptível pelo usuário.

Em essência, pensamento sistêmico transforma decisões de implementação em decisões arquiteturais conscientes. Ele produz interfaces mais consistentes, resilientes e previsíveis, onde cada componente contribui para a saúde do ecossistema em vez de competir com ele. É o que separa desenvolvimento de componentes do **desenvolvimento de sistemas**.

# REACT COMO FERRAMENTA, NÃO FIM

A ideia central deste tópico é reforçar que o React não é o objetivo final, ele é apenas uma ferramenta a serviço da experiência do usuário, da clareza do código e das necessidades reais do produto. Quando equipes começam a decidir sua arquitetura baseadas em "o que o React oferece" ao invés de "o que o problema exige", nasce uma forma de engenharia acidental: componentes complexos demais, abstrações desnecessárias, hooks usados sem propósito e otimizações prematuras que não trazem impacto concreto para o usuário final. React é poderoso, mas esse poder deve ser aplicado com intenção, não por hábito ou modismo.

Muitas features do ecossistema podem ser desnecessárias quando o problema não justifica sua complexidade. Patterns como Context API para qualquer tipo de dado, memoização generalizada, reducers para estados simples, ou criação excessiva de custom hooks são exemplos comuns. O Framework fornece mecanismos sofisticados, mas isso não implica que devam ser aplicados a toda solução. Criar um contexto global para valores que pertencem localmente a um componente, utilizar react.memo em componentes que quase não re-renderizam, ou aplicar useReducer para estados triviais introduz mais custo cognitivo do que benefício real. Esses são casos de decisões guiadas pelo framework ao invés de intenção.

Balancear inovação com simplicidade é, na prática, um exercício de engenharia madura. Significa reconhecer quando um recurso moderno resolve um problema real (como server components reduzindo o bundle, suspense simplificando fluxo de dados assíncronos ou transições concorrentes melhorando interações pesadas) mas também dizer "não" a tecnologias que acrescentam complexidade sem retorno claro. A inovação deve ser incremental e fundamentada, guiada por métricas de performance, previsibilidade do código e necessidades do usuário, não por entusiasmo técnico. Ferramentas e APIs mais recentes devem ser adotadas quando reduzem o esforço de manutenção ou melhoram a experiência, nunca apenas para "estar atualizado".

Decisões guiadas por propósito são aquelas fundamentadas no que o produto exige. Se uma tela precisa carregar rapidamente, talvez a solução seja SSG, não suspense. Se um componente exige personalização profunda, composition resolve mais elegantemente do que HOCs ou abstrações excessivas. Se o estado pertence a apenas um componente, useState é mais claro e performático do que mover tudo para um gerenciador global. Já decisões guiadas por framework são aquelas onde a escolha nasce da ferramenta e não de necessidade: usar Context para evitar prop drilling mesmo quando passar props seria mais explícito, usar Redux para um projeto simples porque "é o padrão corporativo", ou adotar serer components apenas porque são modernos, mesmo que a aplicação não se beneficie.

Em síntese, pensar react como ferramenta significa reverter a lógica tradicional: primeiro entende-se a **intenção do produto**, depois escolhe-se a solução técnica mais adequada. Esse mindset protege o time de complexidade desnecessária, aumenta a longevidade do código e garante que a experiência do usuário permaneça o objetivo principal, com React servindo, e não governando, as decisões de arquitetura.

# COMO PENSAR COMO UM ENGENHEIRO DE INTERFACE

Pensar como um engenheiro de interface significa enxergar o frontend não apenas como um conjunto de telas e componentes, mas como um sistema coerente, onde decisões de código impactam diretamente a experiência do usuário, a performance, a capacidade de manutenção e previsibilidade do comportamento da aplicação. Em vez de focar somente na solução local de um problema, você passa a considerar como cada escolha (um hook, um contexto, uma abstração ou a posição de um estado na árvore) interage com o todo. Essa mentalidade exige maturidade técnica e um compromisso constante com clareza, consistência e propósito.

O primeiro passo é aprender a fazer as perguntas certas antes de escrever qualquer linha de código. Perguntas como: este componente realmente precisa existir? Ele resolve um caso isolado ou representa um padrão repetido? Quem deve ser o dono do estado? Esse dado é local, compartilhado ou remoto? Esse hook é necessário ou é abstração prematura? Como essa decisão afeta a testabilidade, a previsibilidade e o custo de manutenção? Somente quando essas perguntas são respondidas é possível escolher conscientemente o pattern correto, seja um componente controlado, um contexto, um custom hook, uma composição mais flexível ou até uma solução fora do React. Um engenheiro de interface não se move por reflexo técnico, mas por intenção.

Avaliar o impacto de uma decisão no sistema global significa enxergar cada mudança como parte de um ecossistema. Introduzir um contexto afeta re-renderizações em múltiplos pontos da árvore. Criar um hook compartilhado afeta todos que o utilizam. Alterar a estrutura de componentes pode facilitar ou dificultar extensões futuras. Um componente com responsabilidade mal definida pode se tornar um gargalo para evolução. Por isso, decisões precisam considerar **performance** (quantas re-renderizações isso gera?), **legibilidade** (fica claro o que o componente faz?), **previsibilidade** (ele se comporta sempre da mesma forma?), **isolamento** (ele acopla partes que deveriam ser independentes?) e consistência (está mantendo os padrões adotados pela equipe?).

Aplicar essa filosofia em apps grandes e complexos exige combinar vários princípios aprendidos ao longo do estudo de React: manter uma fonte de verdade clara, evitar duplicação de estado, dividir lógica de UI da lógica de domínio, usar composição ao invés de herança, estabelecer boundaries explícitos entre client e server, dimensionar contextos com granularidade adequada, preferir hooks customizados para lógica reutilizável e manter componentes pequenos, previsíveis e fáceis de testar. Também envolve monitorar performance real, entender como renderizações se propagam pela árvore, antecipar gargalos e projetar fluxos de dados que minimizem inconsistências.

Por fim, pensar como engenheiro de interface é adotar uma postura contínua de responsabilidade técnica. Significa evitar soluções encantadoras, mas complexas, quando uma abordagem simples resolve o problema com clareza, evitar abstrações sem necessidade, projetar APIs que outros desenvolvedores compreendam facilmente e, principalmente, trabalhar guiado por intenção e impacto, não por modismos. Um engenheiro de interface experiente constrói sistemas que funcionam bem hoje, evoluem com facilidade amanhã e permanecem compreensíveis depois de anos. Essa é a essência da disciplina.

---

# PERGUNTAS

CAIO ROSSI . D E V

# MÓDULO 1

## **Paradigma Imperativo vs Declarativo**

1- Por que o paradigma declarativo é considerado um nível de abstração acima do imperativo, e como o React utiliza essa ideia para simplificar a construção de interfaces?

2- Como o React transforma uma declaração de UI em operações imperativas no DOM, e por que isso permite que o desenvolvedor pense apenas no “resultado desejado”?

3- Em quais situações o paradigma declarativo pode parecer menos intuitivo que o imperativo, e por que ainda assim ele é vantajoso para interfaces complexas?

## **React como UI derivada de estado**

4-Como o conceito “UI = f(state)” garante que a interface permaneça sempre consistente, mesmo em aplicações complexas?

5-Por que manipular o DOM diretamente pode gerar inconsistências entre a UI e o estado, e como a abordagem declarativa do React resolve esse problema?

6- Como a distinção entre estado local, derivado e global influencia a forma como a UI é calculada e mantida em sincronia no React?

## **Composição de componentes e a ideia de árvore de renderização**

7- Por que o React adota composição em vez de herança, e como esse modelo influencia a forma como a UI é estruturada e reutilizada?

8- Como a árvore de renderização é construída recursivamente no React e por que mudanças de estado podem impactar apenas partes específicas dessa árvore?

9- Por que separar responsabilidades entre componentes “pai” e “filho” contribui para previsibilidade, organização e facilidade de manutenção em aplicações React?

# MÓDULO 1

## O papel da previsibilidade e da pureza de funções

10- Por que a pureza da função de renderização é essencial para que o React consiga prever a UI resultante e garantir um processo de reconciliação consistente?

11- Como efeitos colaterais dentro da renderização podem comprometer a previsibilidade do ciclo de atualização do React?

12- De que forma o useEffect permite que aplicações interajam com o mundo externo sem quebrar o requisito de pureza da renderização?

## Abstração entre dados → estado → interface

13- Como o React atua como uma camada de abstração entre dados externos, estado interno e a interface renderizada, e por que essa separação aumenta a previsibilidade da UI?

14- Por que o fluxo unidirecional de dados (one-way data flow) é fundamental para manter consistência entre estado e interface em aplicações React?

15- De que forma o Virtual DOM funciona como uma abstração sobre o DOM real, e como isso reduz a necessidade de o desenvolvedor lidar com atualizações manuais da interface?

## O que diferencia o React de frameworks MVVM/MVC tradicionais

16- Como o modelo “UI = f(estado)” do React elimina a necessidade de camadas intermediárias como Controller (MVC) ou ViewModel (MVVM), e por que isso aumenta a previsibilidade do fluxo de dados?

17- Quais são as principais consequências práticas de adotar fluxo unidirecional de dados no React em comparação com o two-way data binding presente em frameworks MVVM?

18- De que forma o Virtual DOM do React substitui a manipulação manual do DOM presente no MVC tradicional e contribui para mais eficiência e menos complexidade na sincronização entre dados e interface?

# MÓDULO 2

## O que é o Virtual DOM e por que ele existe

19- Como o Virtual DOM permite que o React planeje atualizações antes de aplicá-las no DOM real, e por que isso reduz o custo computacional associado a reflows e repaints?

20- Por que o Virtual DOM é considerado uma camada de abstração entre o estado e o DOM real, e como isso contribui para previsibilidade na renderização?

21- De que forma o processo de diffing/reconciliação garante que apenas as mudanças necessárias sejam aplicadas no DOM real, e quais benefícios isso traz para performance em aplicações complexas?

## Reconciliação – Como o React compara árvores

22- Como o processo de reconciliação permite que o React determine de forma eficiente quais partes do DOM real precisam ser atualizadas após uma mudança de estado?

23- Por que as heurísticas baseadas no tipo dos nós (mesmo tipo vs. tipo diferente) são suficientes para que o React decida entre reaproveitar ou recriar elementos durante a comparação de árvores?

24- Qual é o papel das keys na reconciliação e por que elas são essenciais para manter identidade e evitar atualizações incorretas em listas dinâmicas?

## Diffing algorithm e heurísticas internas

25- Por que comparar duas árvores arbitrárias de forma ingênua levaria a uma complexidade de  $O(n^3)$ , e como isso tornaria inviável a renderização reativa em tempo real?

26- Como as heurísticas do React — como manter nós do mesmo tipo e recriar nós de tipos diferentes — permitem reduzir o processo de diffing para tempo linear ( $O(n)$ )?

27-Qual é o papel das keys dentro do algoritmo de diffing, e como elas evitam que o React tenha que testar múltiplas correspondências possíveis entre elementos de uma lista?

# MÓDULO 2

## **React Fiber: scheduler, prioridade e interrupção de tarefas**

28- Como o modelo do Fiber, ao dividir o trabalho em pequenas unidades, permite que o React pause e retome a reconciliação para manter a interface responsiva?

29- De que forma o Scheduler do React decide quais tarefas devem ser executadas primeiro, e por que a priorização é essencial para evitar travamentos durante interações do usuário?

30- Por que o Fiber não torna o React “mais rápido” em termos absolutos, mas sim mais inteligente e responsivo ao distribuir o trabalho entre render e commit?

## **Tempo de renderização, commit phase e mutation phase**

31- Por que a render phase do React pode ser pausada, interrompida ou refeita sem causar inconsistência na UI, e qual é a importância de ela ser completamente pura?

32- O que torna a commit phase necessariamente síncrona e não-interruptível, e por que o React separa esse momento crítico da fase de cálculo?

33- Como as subetapas da commit phase (before mutation, mutation e layout) organizam a aplicação das mudanças no DOM e ajudam a manter previsibilidade e desempenho na atualização da interface?

## **Como o React garante fluidez sob carga (Concurrent Mode)**

34- Como o Concurrent Mode permite que o React pause e retome renderizações longas por meio de time-slicing, e por que isso evita travamentos perceptíveis na interface?

35- De que forma a priorização de tarefas no Concurrent Mode impacta diretamente a experiência do usuário durante interações como digitação e scroll?

36- Por que o Concurrent Mode não significa execução paralela real, e como o modelo Fiber torna possível essa alternância cooperativa de tarefas dentro de uma única thread?

# MÓDULO 2

## **Batching de Atualizações e Event Loop**

37- Como o batching permite que o React agrupe múltiplas atualizações de estado em um único ciclo de renderização, e por que isso melhora significativamente o desempenho da aplicação?

38- De que forma o Event Loop — especialmente a distinção entre macrotasks e microtasks — determina o momento em que o React aplica os estados acumulados em uma renderização única?

39- Como o batching reduz a carga sobre o reconciler e o Virtual DOM, e qual impacto isso tem nas mutações finais aplicadas ao DOM real?

# MÓDULO 3

## **Por que os hooks foram criados (substituição de classes)**

40- Quais limitações dos componentes de classe motivaram a criação dos Hooks, especialmente em relação à verbosidade, binding de this e fragmentação da lógica de lifecycle?

41- Como os Hooks unificam a forma de criar componentes React, eliminando a divisão entre “componentes inteligentes” e “componentes burros”, que existia na era das classes?

42- De que maneira a filosofia de composição dos Hooks — principalmente através de custom hooks — resolve problemas de reutilização e organização de lógica que eram difíceis de tratar com classes?

## **O ciclo de vida funcional: render → commit → cleanup**

43- Por que a render phase é totalmente pura e independente, e como isso garante que o React possa reexecutá-la quantas vezes forem necessárias sem causar efeitos colaterais?

44- Como a separação entre useEffect (assíncrono) e useLayoutEffect (síncrono) reflete a distinção entre as fases de commit e layout, influenciando diretamente quando o navegador repinta a tela?

45- Qual é o papel da cleanup phase na prevenção de vazamentos de memória e efeitos obsoletos, e por que ela é executada antes de novos efeitos ou quando o componente é desmontado?

## **Regras dos Hooks (ordem, pureza e chamadas)**

46- Por que os Hooks dependem da ordem exata em que são chamados durante a renderização, e como essa “consistência posicional” impede que estados diferentes se misturem?

47- Por que Hooks só podem ser usados dentro de componentes React ou de outros Hooks customizados, e como isso garante que o React consiga rastrear corretamente cada chamada ao longo do ciclo de vida do componente?

# MÓDULO 3

48- Como a exigência de que componentes sejam funções puras influencia o uso de Hooks, e por que efeitos colaterais precisam ser isolados em useEffect para manter previsibilidade na renderização?

## **useState: closures e estado persistente**

49- Como o React utiliza o modelo de “gavetas internas” para manter valores de estado entre renderizações e por que essa estrutura depende da ordem em que os Hooks são chamados?

50- De que maneira closures influenciam o comportamento de callbacks em componentes React, especialmente quando esses callbacks “lemboram” do estado existente no momento da renderização em que foram criados?

51- Por que múltiplas chamadas sequenciais de setState dentro do mesmo evento podem usar valores desatualizados, e como a forma funcional de atualização (setCount(prev => prev + 1)) resolve esse problema?

## **useEffect: sincronização com o mundo externo**

52- Por que o useEffect só é executado após o commit e nunca durante a fase de renderização, e como isso preserva a pureza e previsibilidade dos componentes React?

53- Como closures criadas em diferentes renders podem levar ao problema de stale closures dentro de efeitos, e por que o array de dependências é essencial para evitar esse comportamento?

54- De que maneira o useEffect lida com montagem, atualização e desmontagem do componente, e como o cleanup garante que efeitos antigos não continuem ativos ou causem vazamentos?

## **useRef: identidade e persistência fora do ciclo**

55- Por que valores armazenados em useRef persistem entre renderizações sem disparar novos renders, e como isso diferencia refs de estados reativos como useState?

# MÓDULO 3

56- Em quais situações valores mutáveis devem ser guardados em useRef em vez de useState, e como essa escolha evita renderizações desnecessárias e melhora a performance?

57- Por que o objeto retornado por useRef é considerado estável entre renders, e como essa estabilidade permite que ref.current seja usado como uma “gaveta operacional” independente do ciclo de reconciliação do React?

## **useMemo e useCallback: memoização e reatividade controlada**

58- Como o useMemo e o useCallback utilizam “gavetas internas” semelhantes às de useState para armazenar valores e funções memoizadas entre renders, e por que isso evita recálculações desnecessárias?

59- Em quais cenários a memoização realmente melhora o desempenho — e por que aplicar useMemo ou useCallback indiscriminadamente pode, ao contrário, piorar a performance de um componente?

60- Qual é a diferença conceitual entre memoizar um valor computado com useMemo e memoizar uma referência de função com useCallback, especialmente no contexto de componentes filhos que dependem de referências estáveis?

## **useReducer e o padrão de isolamento de lógica**

61- Por que o useReducer é mais adequado que múltiplos useState quando o estado possui partes interdependentes e várias transições possíveis, e como isso melhora previsibilidade e organização da lógica?

62- Como o princípio do reducer puro (state, action) → newState contribui para testabilidade e manutenção, especialmente em fluxos complexos como carrinhos de compras ou formulários avançados?

63- De que forma o useReducer difere de gerenciadores globais como Redux ou Zustand em termos de escopo e responsabilidade, e quando faz sentido optar por um reducer local versus um estado global compartilhado?

# MÓDULO 3

## **Hooks customizados: composição e reutilização**

64- Como os hooks customizados permitem encapsular e reutilizar lógica de estado, efeitos e cálculos derivados, reduzindo duplicação e tornando componentes mais declarativos?

65- Por que custom hooks geralmente combinam vários hooks nativos (como useState, useEffect, useReducer, useMemo) para formar comportamentos de domínio mais complexos e coesos?

66- De que forma os hooks customizados substituem padrões antigos como HOCs e render props, oferecendo reutilização de lógica com menos camadas, menos acoplamento e maior clareza?

## **O problema do stale state e closures em React**

67- Como o mecanismo de closures faz com que funções criadas em um render “congelem” valores antigos de estado, levando ao problema conhecido como stale state?

68- Por que efeitos e callbacks que dependem de variáveis de estado precisam declarar corretamente suas dependências no useEffect, e como isso evita que eles operem com valores obsoletos?

69- Em quais situações o uso de useRef ou da forma funcional de atualização (setState(prev => ...)) é mais adequado para evitar stale closures, especialmente em lógicas assíncronas ou contínuas como intervalos e listeners?

# MÓDULO 4

## **Tipos de Estado: local, derivado, global e remoto**

70- Como o estado local contribui para o isolamento e simplicidade dos componentes, e por que ele deve ser preferido sempre que a informação não precisa ser compartilhada?

71- Por que o estado derivado não deve ser armazenado diretamente com useState, e como essa prática evita inconsistências e duplicação desnecessária de dados?

72- Quais são as diferenças fundamentais entre estado global e remoto, e por que cada um exige estratégias específicas de sincronização, cache e compartilhamento dentro de uma aplicação React?

## **Fonte de verdade e estado derivado**

73- Por que manter uma única fonte de verdade é essencial para evitar inconsistências de UI, especialmente quando múltiplos valores dependem de um mesmo estado base?

74- Como o uso de estado derivado evita duplicação e sincronização manual, e por que armazenar valores calculados com useState pode gerar bugs difíceis de rastrear?

75- Em que situações useMemo ajuda a derivar valores pesados sem recalculá-los desnecessariamente, e como isso se relaciona diretamente com o princípio de manter apenas o estado essencial?

## **Sincronização de estados entre componentes**

76- Como o padrão lifting state up garante que múltiplos componentes dependam da mesma fonte de verdade, evitando inconsistências quando vários precisam ler ou modificar o mesmo dado?

77- Por que o prop drilling pode se tornar um problema conforme a aplicação cresce, e em quais cenários soluções como Context API, Redux ou Zustand passam a ser mais adequadas?

78-Como o modelo declarativo do React garante que, uma vez centralizado o estado, todos os componentes que dependem dele se mantenham sincronizados automaticamente, sem necessidade de atualizações manuais?

# MÓDULO 4

## **Context API – conceito, escopo e custo de renderização**

79- Por que o Context API resolve o problema de prop drilling, e como o provider atua como uma “fonte de verdade global” para qualquer componente dentro do seu escopo?

80- Como a mudança de valores em um Provider provoca re-renderização em todos os consumidores, e por que dividir contextos em unidades menores reduz esse custo de performance?

81- Em quais cenários o Context API é suficiente para estado global e em quais casos ferramentas como Redux ou Zustand se tornam mais adequadas, considerando escala, controle e granularidade?

## **Patterns de gerenciamento**

82- Como o padrão lifting state up garante uma única fonte de verdade e em que momento ele começa a se tornar insuficiente devido ao aumento de prop drilling?

83- Por que o prop drilling é aceitável em estruturas rasas, mas se torna um problema de acoplamento e manutenção em aplicações maiores, especialmente quando a hierarquia muda?

84- O que caracteriza um bom uso de context splitting, e como encontrar o equilíbrio entre dividir contexto demais (fragmentação) e dividir de menos (re-renderizações desnecessárias)?

## **Gerenciamento externo (Redux, Zustand, React Query)**

85- Quando a aplicação começa a crescer, qual é o principal sinal de que levantar estado e usar Context API não são mais suficientes, tornando necessário adotar uma store externa?

86- Redux, Zustand e React Query resolvem problemas diferentes. Qual é a distinção fundamental entre UI state e server state, e por que o React Query não deve substituir Redux ou Zustand?

# MÓDULO 4

87- Em termos de performance, por que o Zustand consegue evitar re-renderizações em massa que ocorreriam com um Context mal dividido, e como isso influencia a escalabilidade da aplicação?

## **Cache – Sincronização e invalidação de dados**

88- Por que o cache, apesar de melhorar a performance, pode causar inconsistências visuais se não houver uma política clara de invalidação?

89- Qual é a diferença essencial entre cache de UI (como useMemo) e cache de servidor (como React Query), e por que um não substitui o outro?

90- Como ferramentas como React Query evitam que o usuário veja dados obsoletos ao navegar entre telas, e por que isso é superior a implementar fetch manual com useEffect?

## **Server state vs UI state**

91- Por que tratar dados vindos de API como estado local via useState + useEffect leva inevitavelmente a problemas de stale data e duplicação de lógica?

92- Qual é a diferença estrutural entre server state e UI state que justifica o uso de ferramentas como React Query para um, mas não para o outro?

93- Como o modelo declarativo do React Query (cache, revalidação, erro, loading) melhora a previsibilidade e evita código repetitivo em comparação com o padrão manual via useEffect?

## **Estratégias para evitar re-renderizações desnecessárias**

94- Por que a memoização excessiva (uso indiscriminado de useMemo e useCallback) pode piorar a performance em vez de melhorar?

95- Como o context splitting reduz re-renders desnecessários e qual é o problema de manter um único ApplicationContext gigante?

96- Em que situações o React.memo realmente traz benefício — e por que ele não impede renderizações quando o problema está no estado posicionado no nível errado da árvore?

# MÓDULO 5

## Ciclo de renderização e commit detalhado

97- Por que a render phase precisa ser completamente pura e sem efeitos colaterais para que o React consiga interromper, pausar e reexecutar renders de forma segura?

98- Qual é o papel da commit phase na sincronização entre estado lógico e interface visual, e por que ela não pode ser interrompida como a render phase?

99- Por que efeitos como useEffect e useLayoutEffect só rodam após o commit, e o que poderia dar errado se eles fossem executados durante a fase de renderização?

## Identificação de causas de re-render

100- Por que toda mudança no estado local força um re-render do componente, mesmo que o valor atualizado seja o mesmo da renderização anterior?

101- Como o context pode causar re-renderizações invisíveis e amplas, e por que ele é considerado um "broadcast de renders" dentro da árvore de componentes?

102- Em que situações mover um estado "para cima" (lifting state up) aumenta re-renders desnecessários e como identificar que o estado está posicionado alto demais na árvore?

## Reatividade granular e isolamento de componentes

103- Como a divisão estratégica de componentes reduz o custo de re-renderizações e por que colocar toda a interface dentro de um único componente é prejudicial para a performance?

104- Quais são os principais trade-offs ao aplicar granularidade extrema na árvore de componentes e por que componentes pequenos demais podem dificultar a manutenção?

105- Como a combinação entre isolamento de componentes e memoização seletiva (React.memo / useMemo / useCallback) cria uma reatividade mais eficiente sem comprometer a clareza do código?

# MÓDULO 5

## Suspense e streaming de dados (React 18+)

106- Como o Suspense melhora a experiência do usuário ao permitir que partes da UI “aguardem” dados sem bloquear toda a página, e por que isso é superior ao padrão tradicional com múltiplos estados de loading?

107- Qual é a diferença entre Suspense para código (lazy loading) e Suspense para dados, e como cada um afeta a render phase durante carregamentos assíncronos?

108- Por que o streaming de dados em SSR/Server Components torna o carregamento inicial muito mais rápido, e como ele interage com o Suspense para liberar partes da UI assim que ficam prontas?

## Lazy loading e divisão de código

109- Por que o lazy loading reduz o tempo de carregamento inicial da aplicação, e como o React.lazy trabalha junto com o Suspense para permitir isso?

110- Como o code splitting baseado em rotas evita que o usuário baixe componentes que ele talvez nunca visite, e por que isso melhora o tempo de pintura (paint time)?

111- Qual é o impacto do lazy loading no tamanho do bundle inicial e por que dividir o código de maneira estratégica é essencial para evitar gargalos de performance em apps grandes?

## Concurrent rendering: scheduling, interrupção e prioridade

112- Como o React 18 consegue “parecer” multitarefa mesmo rodando em single-thread, e por que o time slicing é essencial para isso?

113- Qual a diferença prática entre bloquear a UI com renderização síncrona e permitir interrupções com concurrent rendering?

114- Por que o batching automático do React 18 reduz o número de renders, enquanto o concurrent rendering reduz o custo de cada render?

# MÓDULO 5

## **Interação com Web Vitals e tempo de pintura**

115- Como grandes renderizações síncronas no React afetam métricas como LCP e INP, e por que dividir componentes ajuda diretamente na interatividade?

116- Por que reservar espaço para elementos dinâmicos evita aumento no CLS, e como isso se relaciona com o ciclo de render do React?

117- Como o concurrent rendering e técnicas como startTransition ajudam a melhorar Web Vitals ao priorizar tarefas mais urgentes na thread principal?

# MÓDULO 6

## **Padrão de composição (props.children, render props, compound components)**

118- Quais problemas de flexibilidade estrutural o padrão props.children resolve, e por que ele é ideal para componentes “invólucro”?

119- Em que cenários render props ainda fazem sentido mesmo após o surgimento dos hooks, e por que elas permitem personalização total da UI sem duplicar lógica?

120- Como compound components criam APIs mais expressivas e escaláveis, e por que o uso de contexto interno evita explosão de props em componentes complexos?

## **Inversão de controle e abstração progressiva**

121- Como a inversão de controle reduz rigidez e explosão de props em componentes, permitindo que o consumidor decida comportamento e estrutura interna?

122- Por que a abstração progressiva evita over-engineering, e como ela ajuda a evoluir componentes sem quebrar quem já os utiliza?

123- Como IoC e abstração progressiva se conectam aos princípios SRP e OCP (SOLID) no design de componentes React?

## **Componentes controlados vs não controlados**

124- Por que componentes controlados oferecem previsibilidade e seguem o princípio de “fonte de verdade única” no React?

125- Em quais cenários componentes não controlados são preferíveis, e quais limitações surgem ao delegar o valor ao próprio DOM?

126- Como a escolha entre controlado vs não controlado afeta validação, sincronização e performance em formulários grandes?

# MÓDULO 6

## **Patterns de isolamento: container/presentational, smart/dumb**

127- Como o padrão Container/Presentational (Smart/Dumb) ajuda a isolar lógica de negócios da lógica de apresentação, e por que isso melhora manutenibilidade e testabilidade?

128- Quais são as características principais de um componente Container (Smart) e de um componente Presentational (Dumb), e como essa separação reduz acoplamento na árvore de componentes?

129- Em quais situações esse padrão deixa de ser ideal — especialmente com Hooks e custom hooks — e como adaptar sua aplicação para evitar duplicação de lógica ou sobre-abstração?

## **Prop drilling e patterns alternativos**

130- Em que ponto o prop drilling deixa de ser apenas uma consequência natural da composição do React e passa a se tornar um problema real de acoplamento, manutenção e re-renderizações desnecessárias?

131- Como o Context API, hooks customizados, composição e HOCs ajudam a reduzir ou eliminar prop drilling, e quais são os trade-offs de cada abordagem?

132- Por que o prop drilling ainda é aceitável — e até desejável — em estruturas pequenas, e como decidir o momento certo de substituir esse padrão por uma solução mais global ou estratégica?

## **Patterns de extensibilidade (slot pattern, HOCs conceitualmente)**

133- Em quais situações o slot pattern oferece melhor extensibilidade do que expor dezenas de props específicas, e como isso reduz acoplamento estrutural em componentes complexos?

134- Por que os HOCs foram importantes antes dos hooks, e quais são os principais trade-offs que fazem com que hoje sejam substituídos por custom hooks na maioria dos casos

135- Como decidir entre slot pattern, HOC ou custom hook ao estender um componente, considerando estrutura, comportamento reutilizável e manutenção a longo prazo

# MÓDULO 6

## Princípios de design componível

136- Como os pilares de coesão, previsibilidade e isolamento orientam a criação de componentes escaláveis no React, e quais sintomas indicam a violação desses princípios?

137- Por que previsibilidade depende de separar logicamente render, commit e efeitos colaterais, e como práticas incorretas tornam o componente difícil de testar e de confiar?

138- Como equilibrar generalização vs simplicidade usando abstração progressiva, evitando tanto over-engineering quanto componentes rígidos demais?

# MÓDULO 7

## **Server Components**

139- Como os Server Components reduzem drasticamente o tamanho do bundle e por que isso melhora métricas como LCP e TTI em aplicações modernas?

140- Por que a regra “tudo começa como Server Component; só vira Client se precisar” cria uma arquitetura mais eficiente e previsível?

141- Em um fluxo real (como e-commerce), como decidir de forma objetiva quais partes devem ser Server Components e quais precisam necessariamente ser Client Components — e quais problemas surgem quando essa separação é feita de forma incorreta?

## **SSR, SSG, CSR e Hidratação**

142- Como SSR, SSG e CSR diferem em onde e quando o HTML é gerado — e como isso impacta diretamente métricas como LCP, TTFB e SEO?

143- Por que a hidratação é um passo crítico em SSR/SSG, e quais problemas acontecem quando o HTML do servidor não corresponde ao render do cliente (ex.: divergências, layout shifts, re-render forçado)?

144- Em quais cenários SSR é superior a SSG, e quando SSG supera SSR — e como decidir entre eles ao projetar uma aplicação React que precisa equilibrar dinamismo, performance e SEO?

## **Transmissão de dados e boundaries entre client/server**

145- Por que manter o client boundary o mais alto possível na árvore reduz drasticamente o tamanho do bundle e evita que partes desnecessárias da UI virem client components?

146- Como a separação entre estado server-only e estado client-only evita bugs estruturais e previne que lógica pesada ou dados sensíveis vazem para o navegador?

147- Por que centralizar fetching em server components reduz over-fetching, duplicação de dados e inconsistência entre requisições — e como escolher corretamente onde posicionar o boundary para garantir que o fetch aconteça uma única vez?

# MÓDULO 7

## Suspense for Data Fetching e Streaming

148- Como o Suspense para dados elimina a necessidade de useEffect + estados de loading, e por que isso resulta em uma experiência muito mais fluida e declarativa para o usuário?

149- De que forma o streaming permite que o servidor envie partes da UI conforme ficam prontas, e como isso reduz tela branca, melhora LCP e evita bloqueios causados por fetches lentos?

150- Por que combinar Suspense + Server Components + streaming cria um fluxo de carregamento superior ao modelo tradicional de fetch no cliente, e quais problemas estruturais esse trio resolve em aplicações complexas?

## Limitações e Padrões Emergentes (Next.js 14/15, React 19)

151- Quais são as principais restrições impostas aos Server Components (como impossibilidade de usar hooks client-side) e como essas limitações reforçam o princípio de “componentes puros” no servidor?

152- Como os boundaries entre server e client se tornaram mais explícitos no Next.js 15, e por que a combinação de Server Components + Server Actions exige decisões arquiteturais mais rigorosas sobre onde cada responsabilidade deve viver?

153- De que forma React 19 (com o hook use, caching integrado e Suspense/streaming nativos) muda o papel do cliente, reduz dependência de fetch em useEffect e estabelece novos padrões para aplicações híbridas?

## Limitações e Padrões Emergentes (Next.js 14/15, React 19)

154- Como a divisão definitiva entre Server Components e Client Components redefine o papel do React e muda o modelo mental de “tudo roda no navegador”?

155- Por que o concurrent rendering deixa de ser apenas uma otimização interna e passa a influenciar diretamente o design dos componentes, exigindo granularidade, interrupção e resiliência?

## MÓDULO 7

156- De que forma o streaming progressivo substitui o fluxo tradicional “fetch → loading → render” e transforma a experiência de carregamento em algo contínuo, declarativo e arquiteturalmente integrado?

# MÓDULO 8

## **Trade-offs - abstração vs clareza**

157- Como determinar o momento exato em que uma abstração realmente reduz complexidade, em vez de introduzir opacidade e esforço cognitivo desnecessário?

158- Quais sinais indicam que uma abstração está escondendo intenção e prejudicando clareza, mesmo tendo eliminado duplicação aparente no código?

159- Por que repetir lógica algumas vezes pode ser preferível a abstrair cedo demais, e como isso se relaciona com previsibilidade, manutenção e evolução incremental do sistema?

## **Decisões guiadas por intenção, não por ferramenta**

160- Como decisões guiadas por intenção ajudam a evitar o uso excessivo de ferramentas (context, hooks customizados, memoização) e mantêm o design mais simples e previsível?

161- Por que compreender a responsabilidade real de um componente é o primeiro passo para escolher corretamente entre estado local, contexto, store global ou ferramentas de server state?

162- Quais são os sinais de que uma decisão arquitetural foi tomada por hábito ou modismo e não pela necessidade explícita da interface ou do domínio?

## **Evolução de código e consistência em equipe**

163- Como padrões claros de arquitetura e fluxo de dados reduzem atrito cognitivo em equipes que evoluem uma base React ao longo do tempo?

164- Por que consistência deve ser aplicada ao que é essencial (fluxo de dados, organização, estados) enquanto a flexibilidade deve ser preservada no que é incidental?

165- Como documentação interna, code reviews e linting colaboram para manter a evolução do código previsível sem engessar a criatividade da equipe?

# MÓDULO 8

## **Design orientado a previsibilidade e legibilidade**

166- Por que previsibilidade reduz a superfície de surpresa em componentes React e como isso se relaciona diretamente com depuração mais rápida?

167- Como side effects mal isolados prejudicam a legibilidade, criam fluxos não determinísticos e quebram o modelo mental declarativo do React?

168- Quais práticas de estado, efeitos e nomeação de props ajudam a manter uma árvore de componentes intuitiva, previsível e fácil de ler?

## **Pensamento sistêmico no frontend**

169. Como o pensamento sistêmico muda a forma de avaliar decisões locais, revelando impactos globais em performance, UX e fluxo de renderização?

170. De que maneira a organização clara entre UI, lógica de domínio, estado local, estado global e server state reduz acoplamento e melhora a previsibilidade do sistema?

171. Por que otimizar um componente isoladamente pode ser ineficaz sem considerar a árvore como um sistema dinâmico, e como granularidade e fluxo de dados influenciam essa otimização?

## **React como ferramenta, não fim**

172. Como diferenciar decisões guiadas por necessidade do produto de decisões guiadas por ferramentas, evitando engenharia accidental em aplicações React?

173. Quais critérios práticos ajudam a identificar quando uma feature avançada do ecossistema (context, reducers, memoização, server components) traz benefício real versus quando adiciona apenas complexidade?

174. De que maneira tratar o React como ferramenta, e não como destino arquitetural, melhora a clareza, reduz dúvida técnica e mantém a experiência do usuário como foco central?

# MÓDULO 8

## Como pensar como um engenheiro de interfaces

175. Quais perguntas essenciais devem guiar um engenheiro de interface antes de decidir onde posicionar estado, como estruturá-lo e qual pattern aplicar?

176. Como avaliar o impacto sistêmico de uma decisão local (um contexto, um hook, um novo componente) na performance, previsibilidade e manutenção da aplicação como um todo?

177. Quais princípios estruturais (fonte de verdade, composição, isolamento, boundaries, granularidade) um engenheiro de interface deve aplicar para manter aplicações React grandes escaláveis e coerentes?



C A I O R O S S I . D E V