React Server Components e Server Actions

Oque vamos ver

01

O Problema que Resolvem

Desafios da renderização no lado do cliente.

02

React Server Components (RSC)

O que são e como mudam o jogo.

03

Server Actions

Interatividade no servidor com segurança.

04

Padrões e Aplicações

Casos de uso e melhores práticas.

05

Mão na Massa!

Demonstração prática de código.

Mas antes:

React 19 – Principais Novidades e Impactos

- **Evolução Arquitetural: Estrutura mais madura, mantendo retrocompatibilidade e facilitando migração gradual.**
- **Server Components: Separação clara entre código do client e do server, melhorando performance e organização.**
- **Suspense nativo: Carregamento assíncrono otimizado sem depender de bibliotecas externas.**
- **React Compiler: Otimizações automáticas de renderização e cache, reduzindo necessidade de código manual.**
- **Hooks Otimistas: Melhor feedback ao usuário em interações assíncronas (ex.: formulários).**
- **Integração com Frameworks: Recomendação oficial para uso com Next.js, Remix e outros.**
- **Experiência do Desenvolvedor: Mais simplicidade no código e maior foco em boas práticas.**

Release notes v19: https://react.dev/blog/2024/12/05/react-19

React Compiler

uma ferramenta que opera exclusivamente em tempo de build, analisando e otimizando seu código React antes mesmo que ele chegue ao navegador. Seu objetivo central é automatizar a memoização. Em vez de você precisar envolver manualmente valores, funções ou componentes com useMemo, useCallback ou React.memo para evitar re-renderizações excessivas, o Compiler faz isso por você.

```
function Price({ amount }) {
  return <span>{amount.toFixed(2)}</span>;
}
// Sem necessidade de memo ou useCallback — React 19 otimiza internamente
```

para mais info:
https://www.rocketseat.com.br/blog/artigos/post/react-compiler-o-que-e-como-funciona

React Suspense nativo

Carregamento assíncrono otimizado sem bibliotecas externas.

```
function Profile({ userId }) {
  const data = use(fetchUser(userId)); // Hook assíncrono
  return <h1>{data.name}</h1>;
}

export default function App() {
  return (
    <Suspense fallback={<p>Carregando...</p>}>
      <Profile userId={1} />
    </Suspense>
  );
}
```

Docs https://react.dev/reference/react/Suspense

Mas já não era nativo?

O React Suspense já existia antes do React 19, mas ele não era totalmente nativo para todos os casos de uso.
No React 18 (e até em versões anteriores, experimentalmente desde o React 16.6), o Suspense funciona, mas com limitações:

React 16.6 (2018) → Introduziu Suspense apenas para code splitting (via React.lazy), ou seja, carregamento dinâmico de componentes.

React 17 → Ainda restrito a casos de React.lazy, sem suporte oficial para dados assíncronos.

React 18 → Trouxe melhorias de concurrent rendering, mas Suspense para dados assíncronos ainda dependia de bibliotecas externas como react-query, Relay ou SWR. Ele não tinha suporte completo integrado para data fetching.

No React 19, o Suspense passa a ser nativamente integrado para carregamento assíncrono de dados, sem precisar de soluções externas para que funcione de forma oficial. Isso inclui:

Uso direto com server components e server actions.
Renderização assíncrona otimizada de forma mais simples.

Hooks Otimistas

Feedback imediato em interações assíncronas.

useFormStatus()
Feedback instantâneo: informe ao usuário quando o envio está em andamento, sem precisar criar estados manuais.

Maior controle: combine com CSS ou animações para proporcionar feedback visual mais rico

```jsx
import { useFormStatus } from 'react-dom';

function SubmitButton() {
  const { pending } = useFormStatus();

  return (
    <button type="submit" disabled={pending}>
      {pending ? 'Enviando...' : 'Enviar'}
    </button>
  );
}
```

use()
é um game-changer na forma de buscar dados. Em vez de usar useEffect e gerenciar estados para requisições, você pode simplesmente usar use()

```jsx
import { use, Suspense } from 'react';

const fetchData = async () => {
  const response = await fetch('https://api.example.com/data');
  return response.json();
};

function DataComponent() {
  const data = use(fetchData()); // Suspende até a promise ser resolvida
  return <div>{data.message}</div>;
}

function App() {
  return (
    <Suspense fallback={<div>Carregando...</div>}>
      <DataComponent />
    </Suspense>
  );
}
```

https://www.rocketseat.com.br/blog/artigos/post/hooks-no-react-19-performance

Hooks Otimistas

Feedback imediato em interações assíncronas.

useOptimistic(): atualizações em tempo real

Execução da ação real: Enquanto o request roda, a UI já exibe o resultado otimista.

Sincronização final: Quando a operação termina, você atualiza o estado real via setState, substituindo o otimista pelo dado correto.

Melhor usabilidade: casos como “like”, comentários ou formulários se tornam muito mais fluidos

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

function LikeButton() {
  const [likes, setLikes] = useState(0);

  const [optimisticLikes, setOptimisticLikes] = useOptimistic(
    likes,
    (currentState, increment) => currentState + increment
  );

  const handleLike = async () => {
    setOptimisticLikes(1);

    try {
      const response = await fetch('/api/like', { method: 'POST' });
      const result = await response.json();

      setLikes(result.likes);
    } catch (error) {
      console.error('Falha ao registrar o like:', error);
      setOptimisticLikes(-1);
    }
  };

  return <button onClick={handleLike}>Curtidas: {optimisticLikes}</button>;
}

export default LikeButton;
```

https://www.rocketseat.com.br/blog/artigos/post/hooks-no-react-19-performance

e os server components?

React Server Components

Renderização no Servidor

Components renderizam **antes** de chegar ao client.

Performance Otimizada

Menos JavaScript enviado ao navegador.

Carregamento inicial mais rápido.

Integração Transparente

Client Components e Server Components coexistem.

O Problema que Resolvem

Performance

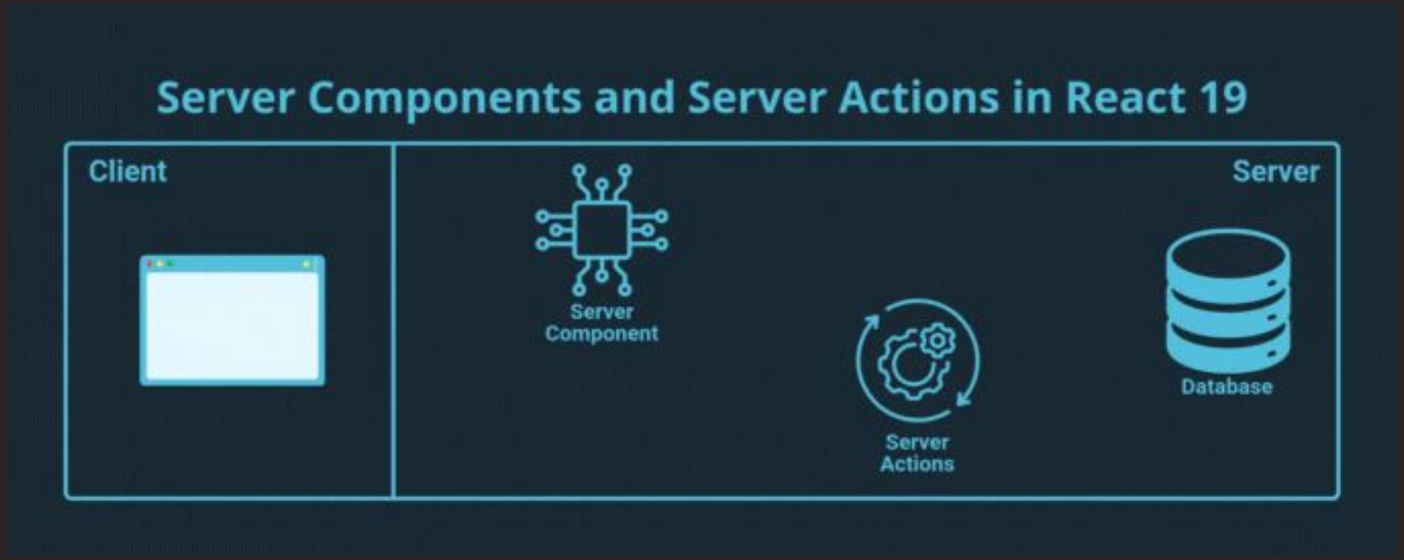
Grandes bundles JS.

Longos tempos de carregamento.

Segurança

Dados sensíveis expostos.

Lógica de negócio no client side.



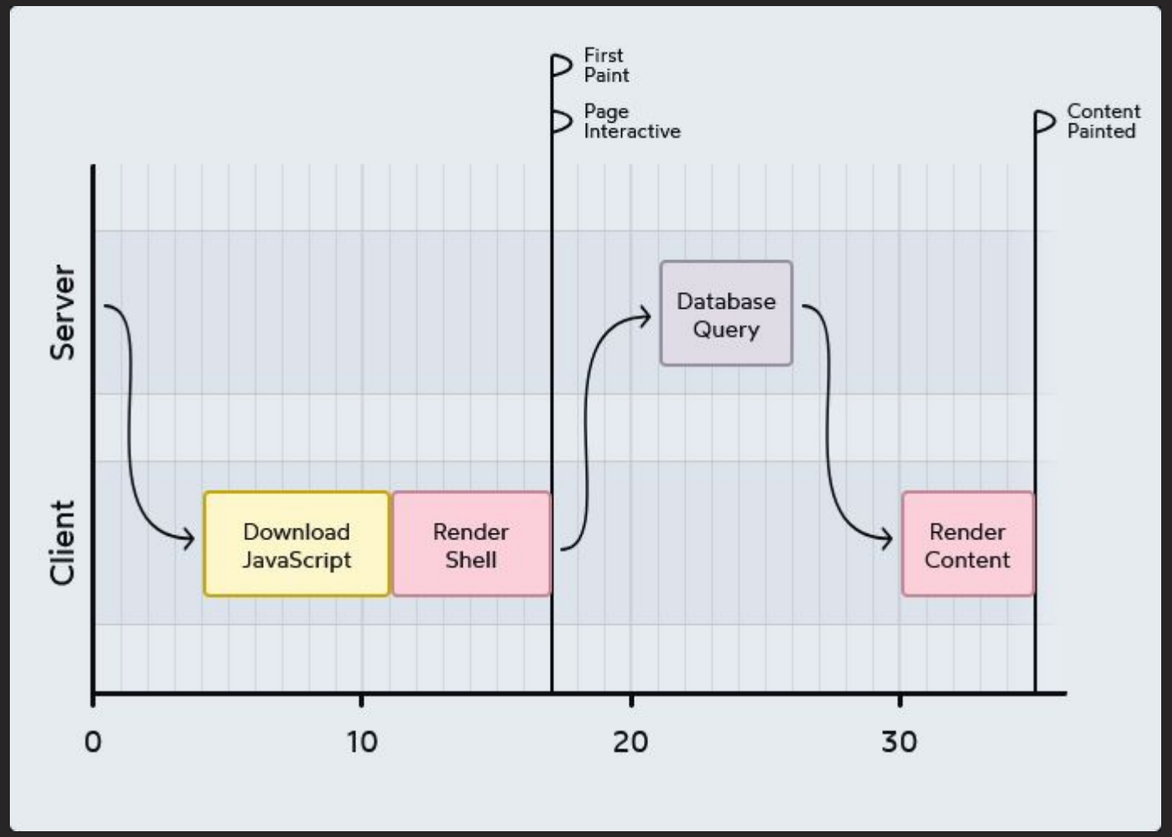
Vantagens server components:

- Sem impacto no bundle JavaScript do client (zero JS extra)
- Acesso direto a APIs ou banco de dados, mantendo segurança
- Suporte nativo a async/await e cache automático
- Streaming / Suspense — renderização progressiva para melhor UX

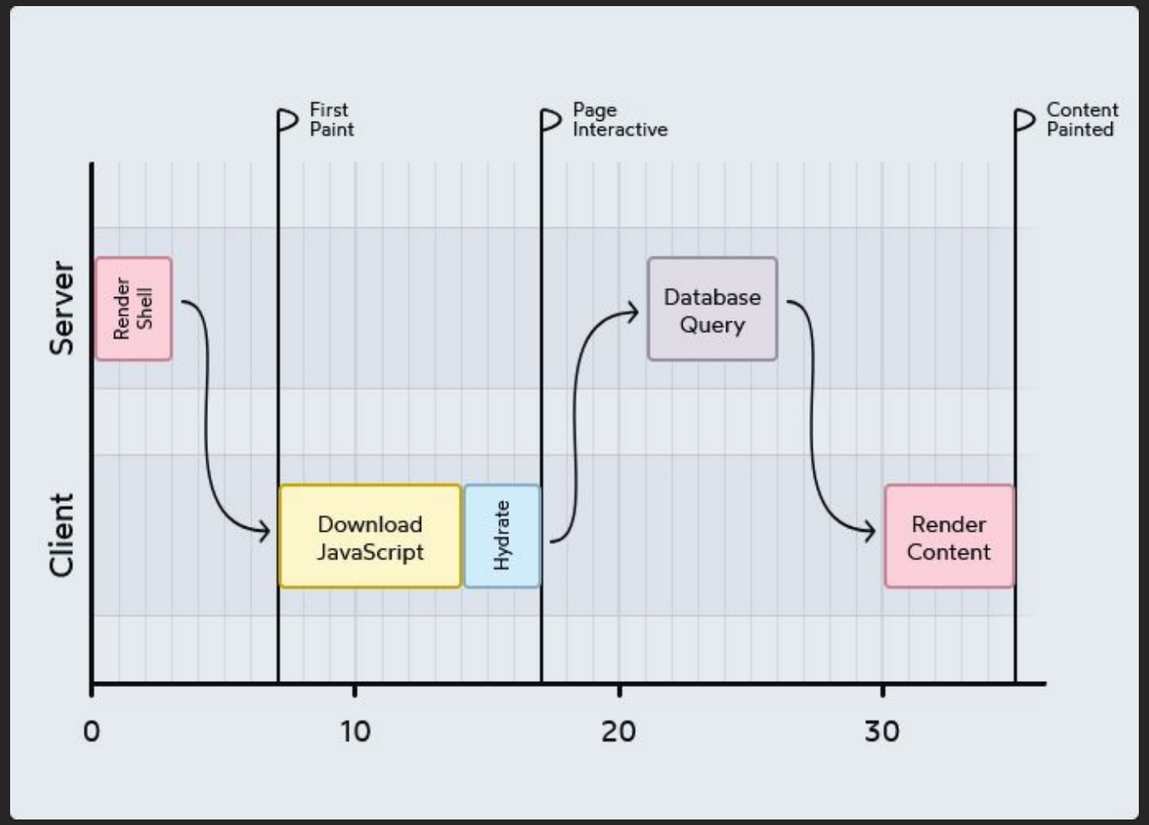
RSC: Client side(CSR) vs Server side (SSR)

Onde Renderiza	use server	use client
Acesso	Backend, Banco de Dados, APIs	Hooks de Estado, Eventos do DOM
Interatividade	Não interativos (por padrão)	Altamente interativos (re-render e hooks)
Tamanho do Bundle	Não contribui	Contribui para o bundle JS

CSR



SSR



e as server actions?

Server Actions

1

Funções Diretas no Servidor

Chamadas de API simplificadas.

2

Mutação de Dados

Atualizar dados com segurança.

3

Segurança Aprimorada

Lógica sensível executa **apenas** no servidor.

O que são Server Actions?

São funções marcadas com "use server" que rodam no servidor, chamadas diretamente pelo client side.

Funcionam como o “garçom” que busca dados ou executa ações no servidor sem que o client veja os bastidores.

Permitem chamar APIs internas ou externas sem precisar implementar toda a camada de servidor.

Simplificam envio de formulários e execução de lógica de negócios no servidor.

Facilitam atualizações em banco de dados de forma segura, mantendo tokens e dados sensíveis no servidor.

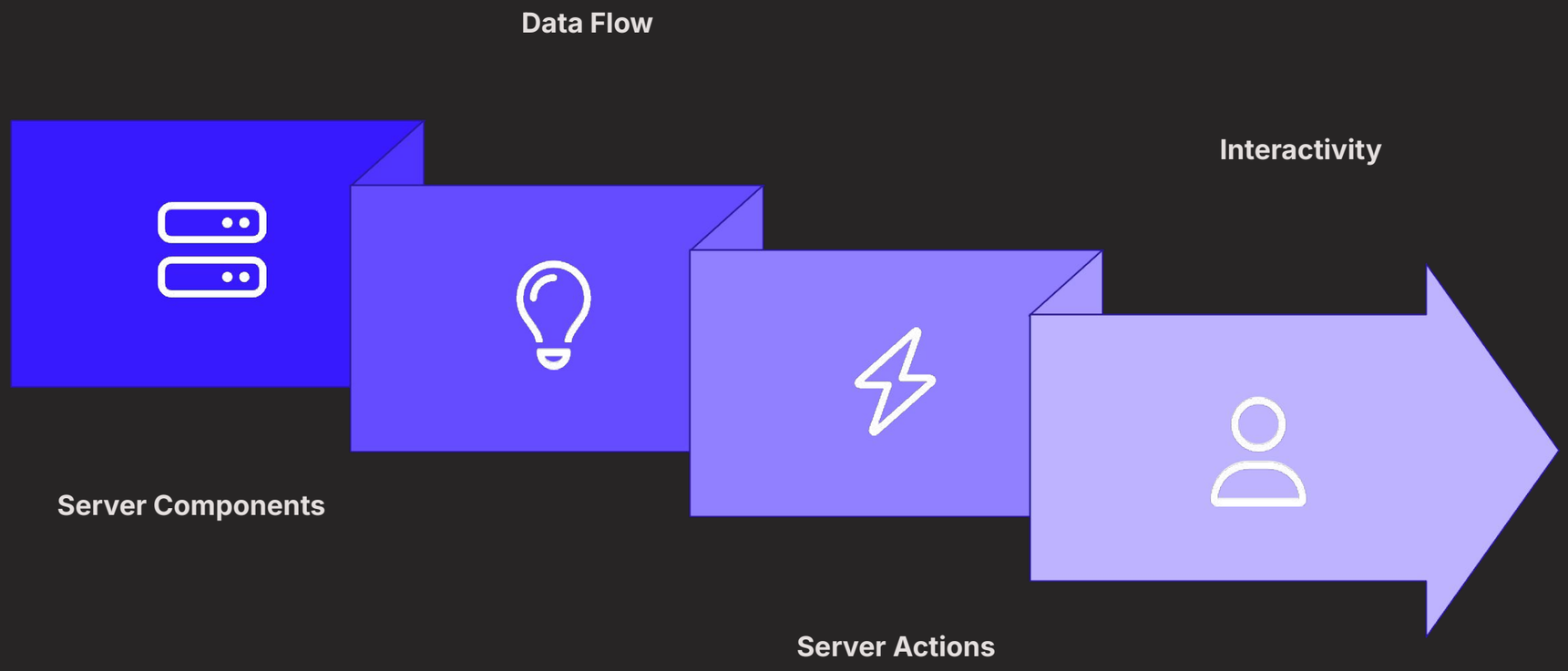
Podem residir no mesmo repositório que os componentes React, unificando front e back.

```
// Server Component
export default function ServerComponent( ) {

  // Server Action
    async function minhaAction( ) {
        'use server'
        // ... implementação
        }

   return ( … )
}
```

Padrões de Uso: RSC e Server Actions



Fluxo Simplificado

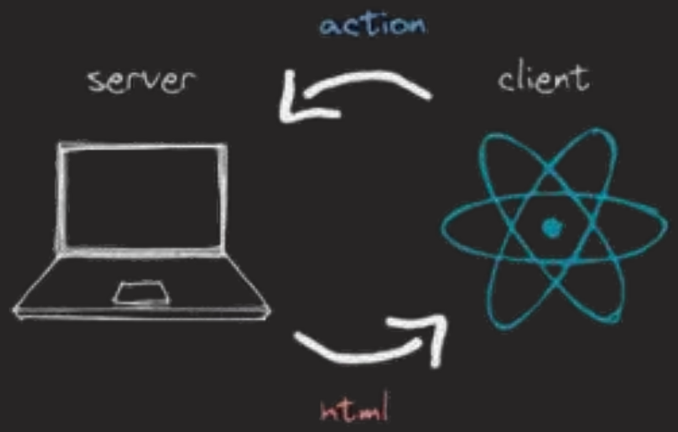
Server Component renderiza UI.

RSC Payload enviado ao client.

Client Component recebe JS e hidrata interatividade.

Formulário no client aciona Server Action.

Action é executada no server (manipula os dados)



e as novas directivas?

são as diretivas use client, use server e only-server, elas ajudam a separar o que roda no client do que fica exclusivamente no server.

use client

O que faz: Diz que um arquivo React Component deve ser tratado como Client Component.

Quando usar: Quando o componente precisa rodar no browser (acesso ao window, eventos de clique, hooks como useState e useEffect, etc.).

Regras:

- **Deve estar na primeira linha do arquivo.**
- **Tudo dentro dele é enviado como JavaScript para o client. (bundle)**
- **Pode importar outros Client Components e bibliotecas específicas de browser.**

Não se deve importar diretamente Server Components, a doc do proprio React recomenda utilizar como children caso necesário.

```
// Button.jsx
"use client";

import { useState } from "react";

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

use server

O que faz: Marca uma função como Server Action, ou seja, ela será executada no servidor, mesmo que seja chamada a partir de um Client Component.

Quando usar: Para interações que precisam acessar dados sensíveis ou executar lógica pesada no servidor (DB, API privada, etc.).

Regras:

- **Pode aparecer no topo do arquivo ou dentro da função.**
- **O código não é enviado para o cliente — só uma referência.**
- **Pode ser importada e chamada de Client Components.**

```
// server-actions.js
'use server';

export async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  return response.json();
}

// ClientComponent.js
'use client';
import { useState, useEffect } from 'react';
import { fetchData } from './server-actions';

export default function ClientComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    async function getData() {
      const result = await fetchData();
      setData(result);
    }
    getData();
  }, []);

  return <div>{data ? data.message : 'Carregando...'}</div>;
}
```

only server (nova e menos falada)

O que faz: Indica que um módulo deve existir apenas no servidor e não pode ser importado por Client Components.

Quando usar: Quando você quer garantir que um código nunca vá para o cliente (ex.: lógica sensível, chave de API, consultas DB).

Diferença para use server:

use server é para funções (Server Actions) que podem ser chamadas de um Client Component.

only-server é para módulos inteiros que não devem nunca ser enviados para o cliente.

Regras:

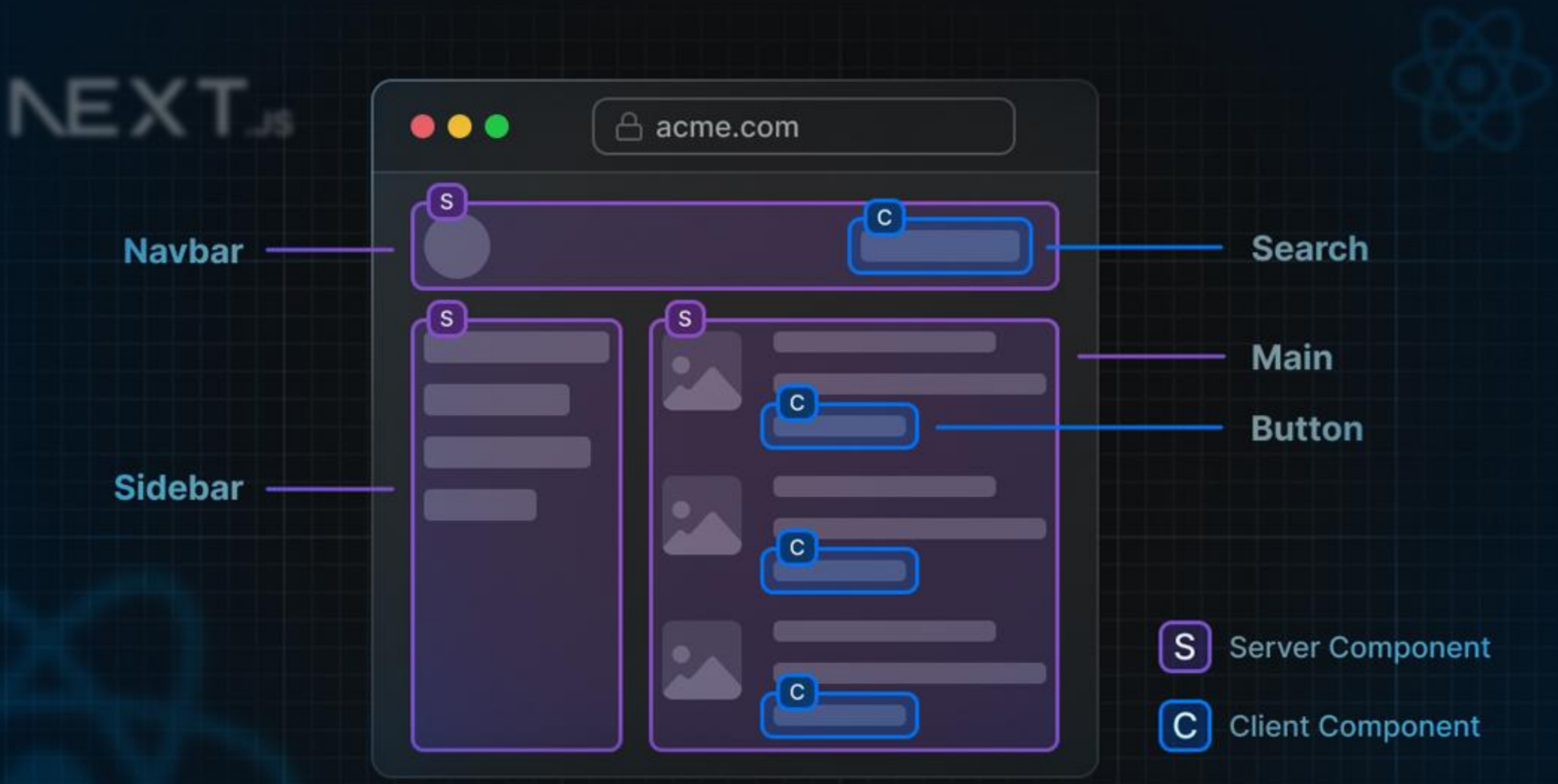
Se um Client Component tentar importar algo marcado com only-server, o build falha.

Serve como proteção de boundary entre client/server.

```
// db.js
"only-server";

import { connect } from "some-db";

export async function getUser(id) {
  return await connect().users.find(id);
}
```



Casos de Uso (para server components e server actions)

Autenticação e autorização: Valide as credenciais do usuário e controle o acesso a recursos específicos da aplicação no servidor.

Gerenciamento de dados: Buscar, atualizar e excluir dados de bancos de dados ou APIs de forma segura e eficiente, sem expor informações importantes, como tokens, etc.

Processamento de pagamentos: Implemente transações seguras sem expor informações confidenciais do cliente.

Geração de relatórios e documentos: Crie relatórios dinâmicos e personalizados no servidor.

Integração com API externas: Utilize as Server Actions para uma integração direta com APIs externas, eliminando a necessidade de desenvolver um API route interno. Com essa abordagem, é possível acessar dados de forma direta, ao mesmo tempo em que se mantêm informações críticas protegidas e seguras, realizando todas as operações relevantes no lado do servidor.

Bora ver o codigo?