

# NextJS

Prof. José Matheus

# Introdução

Next.js é um framework de desenvolvimento web React full-stack de código aberto criado pela equipe da [Vercel](#). Ele é projetado para simplificar a construção de aplicativos da web modernos, oferecendo uma experiência de desenvolvimento mais rápida e eficiente. O Next.js é baseado em React.js e adiciona recursos poderosos, como roteamento simplificado, pré-renderização, geração de páginas estáticas e muito mais. Ele é altamente configurável e flexível, permitindo que os desenvolvedores criem uma variedade de aplicativos da web, desde sites estáticos simples até aplicativos web complexos e dinâmicos.



# Por que usar Next.js em vez de outras opções?

- **Facilidade de uso:** Next.js simplifica muitos aspectos do desenvolvimento web, como roteamento, pré-renderização e gerenciamento de estado, permitindo que os desenvolvedores se concentrem mais na lógica de negócios de suas aplicações;
- **Rápido tempo de carregamento:** Com a pré-renderização e a geração de páginas estáticas, o Next.js ajuda a otimizar o desempenho da aplicação, garantindo tempos de carregamento mais rápidos e uma melhor experiência do usuário;
- **SEO (Search Engine Optimization) amigável:** O Next.js facilita a otimização para mecanismos de busca, uma vez que fornece pré-renderização estática e dinâmica, tornando o conteúdo das páginas mais acessível aos mecanismos de busca;
- **Roteamento automático:** O Next.js oferece um sistema de roteamento automático baseado em arquivos, onde cada arquivo na pasta pages corresponde a uma rota da aplicação. Isso simplifica significativamente a configuração de rotas em comparação com outras bibliotecas ou frameworks;
- **API Routes integradas:** Next.js permite criar APIs facilmente usando suas API Routes, que são arquivos na pasta pages/api. Isso simplifica a criação de serviços de back-end para aplicativos da web, eliminando a necessidade de configurar um servidor separado;
- **Grande ecossistema:** Next.js possui uma comunidade ativa e um ecossistema robusto, com uma ampla gama de bibliotecas e ferramentas disponíveis para estender suas capacidades e simplificar tarefas comuns de desenvolvimento.

# Outras opções além do NextJS

- **Create React App (CRA):** CRA é uma ferramenta oficialmente mantida pelo time do React;
- **Gatsby:** Gatsby é um framework React focado em sites estáticos e progressivos;
- **Vue.js (com Vue Router e Nuxt.js):** Nuxt.js é frequentemente usado para criar aplicativos da web universais (SSR) e aplicativos da web estáticos com Vue.js;
- **Sapper/SvelteKit:** Sapper é um framework para criar aplicativos da web com Svelte, uma biblioteca JavaScript para construção de interfaces de usuário;
- **Angular Universal:** Angular Universal é uma tecnologia oficialmente suportada pelo Angular para renderização do lado do servidor (SSR) de aplicativos Angular [agora é padrão a partir do Angular 17].

# Benefícios e recursos principais do NextJS

- **Roteamento:** Um roteador baseado em sistema de arquivos construído sobre os Componentes do Servidor que suporta layouts, roteamento aninhado, estados de carregamento, tratamento de erros e mais;
- **Renderização:** Renderização do lado do cliente e do servidor com Componentes do Cliente e do Servidor. Ainda mais otimizado com Renderização Estática e Dinâmica no servidor com Next.js. Streaming em tempo de execução Edge e Node.js;
- **Busca de Dados:** Busca de dados simplificada com `async/await` em Componentes do Servidor, e uma API de busca estendida para memorização de solicitação, cache de dados e revalidação;
- **Estilização:** Suporte para seus métodos de estilização preferidos, incluindo Módulos CSS, Tailwind CSS e CSS-in-JS;
- **Otimização:** Otimizações de Imagem, Fontes e Scripts para melhorar os Core Web Vitals e a Experiência do Usuário de sua aplicação;
- **TypeScript:** Suporte aprimorado para TypeScript, com melhor verificação de tipo e compilação mais eficiente, além de Plugin TypeScript personalizado e verificador de tipo;

# Benefícios e recursos principais do NextJS

- **Hospedagem na Vercel:** Next.js tem integração direta com a Vercel, uma plataforma de hospedagem que simplifica o processo de implantação de aplicativos Next.js.

# NextJS SSR

O NextJS utiliza uma renderização SSR (server-side rendering) e CSR (client-side rendering), ou seja, diferentemente do React, que usa somente CSR, o NextJS não compila todo o JS produzido no navegador, ele sobe, paralelo ao código enviado ao navegador, um servidor Node, que hospeda parte da aplicação, e só envia para o navegador do cliente o código necessário e já pronto para rodar, melhorando e muito a performance e otimiza o SEO (Search engine optimization), já que ele renderiza os componentes do SEO de forma rápida.

# Desvantagens do NextJS

- **Curva de Aprendizado Inicial:** O Next.js pode ter uma curva de aprendizado íngreme para desenvolvedores iniciantes em React ou desenvolvimento web avançado;
- **Complexidade para Projetos Simples:** Para projetos simples ou estáticos, o Next.js pode ser excessivamente complexo, sem necessidade dos recursos avançados que oferece;
- **Configuração Manual:** Em certos casos, pode ser necessário configurar manualmente aspectos específicos do projeto no Next.js;
- **Migração de Projetos Existentes:** A migração de projetos existentes para o Next.js pode ser trabalhosa e requerer mudanças significativas na arquitetura do projeto;
- **Bloated Bundle Size:** Dependendo da configuração e das bibliotecas adicionadas, o tamanho do pacote do aplicativo pode aumentar, afetando os tempos de carregamento;
- **Bloqueio de Versão e Atualizações:** Problemas de compatibilidade ou bugs podem surgir em determinadas versões do Next.js, exigindo cuidado na gestão das atualizações e dependências;
- **Maior Complexidade de Hospedagem:** Hospedar um aplicativo Next.js em plataformas diferentes da Vercel pode exigir mais configuração e gerenciamento.



# Fundamentos do NextJS

# Iniciando um projeto NextJS

Antes de tudo, devemos ter certeza que o Node está instalado em nossa máquina. Após isso, podemos iniciar um projeto NextJS.

**create-next-app:** Recomenda-se iniciar um novo aplicativo NextJS usando o create-next-app, que configura tudo automaticamente para você. Para criar um projeto, execute:

```
npx  
create-next-app@latest
```

Após a instalação, você verá o seguinte prompt:

TERMINAL

```
What is your project named? my-app  
  
Would you like to use TypeScript? No / Yes  
  
Would you like to use ESLint? No / Yes  
  
Would you like to use Tailwind CSS? No / Yes  
  
Would you like to use `src/` directory? No / Yes  
  
Would you like to use App Router? (recommended) No / Yes  
  
Would you like to customize the default import alias (@/*)? No / Yes  
  
What import alias would you like configured? @/*
```

# Estrutura de um projeto NextJS

**/node\_modules:** Esta pasta contém todas as dependências do projeto, que são instaladas automaticamente pelo npm ou yarn.

**/public:** Esta pasta contém arquivos estáticos, como imagens, ícones e fontes, que são servidos diretamente para o navegador. Você também pode adicionar arquivos de manifestos PWA (manifest.json) e arquivos de ícones (favicon.ico) aqui.

**src/app:** Esta é uma das pastas mais importantes em um projeto Next.js. Cada arquivo JavaScript (ou TypeScript) nesta pasta corresponde a uma rota no aplicativo. Por exemplo, `app/page.js` corresponde à rota raiz (`/`), enquanto `app/about.js` corresponde à rota `/about`.

**src/app/styles (opcional):** Esta pasta contém arquivos de estilo para o projeto. Por padrão, o Next.js usa arquivos CSS, mas você também pode usar Sass, Less ou qualquer outro pré-processador de sua escolha.

**src/app/components (opcional):** Esta pasta é comumente usada para armazenar componentes React reutilizáveis. Organizar seus componentes nesta pasta pode ajudar a manter o código mais organizado e modular.

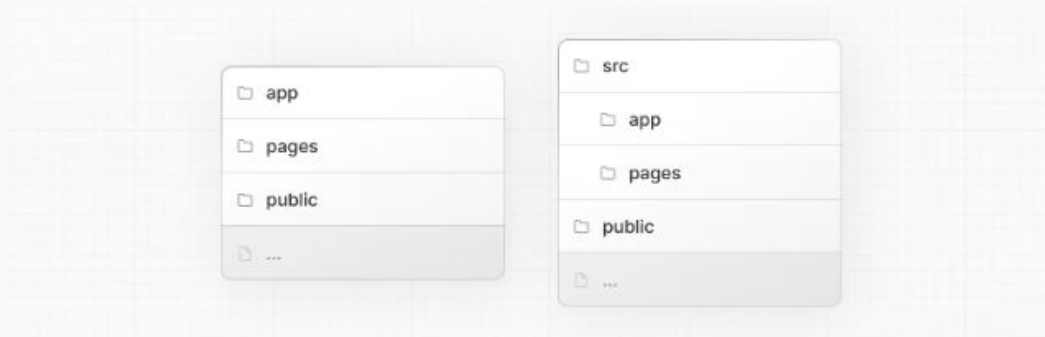
```
> node_modules
> public
> src
.eslintrc.json
.gitignore
next-env.d.ts
JS next.config.mjs
{} package-lock.json
{} package.json
JS postcss.config.mjs
i README.md
tailwind.config.ts
tsconfig.json
```

# Estrutura de um projeto NextJS

## Pastas de nível superior

As pastas de nível superior são usadas para organizar o código do seu aplicativo e os ativos estáticos.

- **app:** Roteador do Aplicativo;
- **pages:** Roteador de Páginas;
- **public:** Ativos estáticos a serem servidos;
- **src:** Pasta opcional de origem do aplicativo.

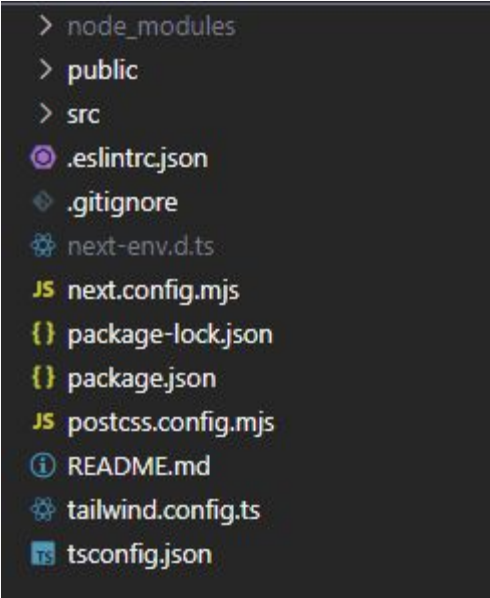


# Estrutura de um projeto NextJS

## Arquivos de nível superior

Os arquivos de nível superior são usados para configurar o seu aplicativo, gerenciar dependências, executar middleware, integrar ferramentas de monitoramento e definir variáveis de ambiente.

- **next.config.js:** Arquivo de configuração para Next.js
- **package.json:** Dependências do projeto e scripts
- **instrumentation.ts:** Arquivo de OpenTelemetry e Instrumentação (observabilidade)
- **middleware.ts:** Middleware de solicitação do Next.js



```
> node_modules
> public
> src
.eslintrc.json
.gitignore
next-env.d.ts
JS next.config.mjs
{} package-lock.json
{} package.json
JS postcss.config.mjs
i README.md
tailwind.config.ts
tsconfig.json
```

# Estrutura de um projeto NextJS

- **.env:** Variáveis de ambiente
- **.env.local:** Variáveis de ambiente locais
- **.env.production:** Variáveis de ambiente de produção
- **.env.development:** Variáveis de ambiente de desenvolvimento
- **.eslintrc.json:** Arquivo de configuração para ESLint
- **.gitignore:** Arquivos e pastas do Git a serem ignorados
- **next-env.d.ts:** Arquivo de declaração TypeScript para Next.js
- **tsconfig.json:** Arquivo de configuração para TypeScript
- **jsconfig.json:** Arquivo de configuração para JavaScript

```
> node_modules
> public
> src
.eslintrc.json
.gitignore
next-env.d.ts
JS next.config.mjs
{} package-lock.json
{} package.json
JS postcss.config.mjs
i README.md
tailwind.config.ts
tsconfig.json
```

# Roteamento em NextJS

# Roteamento do NextJS

O NextJS utiliza a estrutura de pastas para criar as rotas da aplicação. Para tal funcionalidade ser executada, precisamos então, criar dentro da pasta **src/app** as rotas para a nossa aplicação. Por exemplo, se precisarmos criar uma página para um app de “produtos”, dentro da pasta **src/app**, criamos a pasta “**products**” e dentro dela um arquivo chamado **page.tsx**. Esta página será renderizada ao acessarmos a rota **http://localhost:3000/products**.

Obs<sup>1</sup>: para páginas aninhadas, precisamos somente criar uma subpasta, com o nome desejado e dentro dela, criar um arquivo chamado **page.tsx**.

Obs<sup>2</sup>: o primeiro arquivo a ser executado num app NextJS é o arquivo **layout.tsx**, e após isso, o **page.tsx**. Ambos estão na pasta **src/app**.



# Convenções de Roteamento

As seguintes convenções de arquivos são usadas para definir rotas e lidar com metadados no roteador do aplicativo.

## Arquivos de Roteamento

- **layout .js .jsx .tsx:** Layout
- **page .js .jsx .tsx:** Página
- **loading .js .jsx .tsx:** Interface de carregamento
- **not-found .js .jsx .tsx:** Interface de não encontrado
- **error .js .jsx .tsx:** Interface de erro
- **global-error .js .jsx .tsx:** Interface de erro global

- **route .js .ts:** Endpoint da API
- **template .js .jsx .tsx:** Layout re-renderizado
- **default .js .jsx .tsx:** Página de fallback de rota paralela

## Rotas Aninhadas

- **folder:** Segmento de rota
- **folder/folder:** Segmento de rota aninhado

# Rotas dinâmicas

Também podemos criar rotas dinâmicas para acessar, por exemplo, páginas que usam o id para de alguma forma filtrar uma informação.

- **[folder]:** Segmento de rota dinâmico
- **[...folder]:** Segmento de rota de captura de todos
- **[...folder]]:** Segmento de rota de captura de todos opcional

Ex: Para acessarmos uma página que acesse o conteúdo de um produto em específico, poderíamos criar essa estrutura de pastas/arquivos: `products/[productId]/page.tsx`

Conseguimos filtrar, pegando o parâmetro da seguinte forma:

## TYPESCRIPT

```
type PageProps = {
  params: { productId: string }
}

export default function Info({ params }: PageProps) {
  return (
    <div>
      <h1>Produto { params.productId }</h1>
    </div>
  )
}
```

# Rotas dinâmicas aninhadas

Também podemos criar rotas dinâmicas para acessar, por exemplo, páginas que usam o id para de alguma forma filtrar uma informação, a partir de outra informação.

Ex: Para acessarmos uma página que acesse o conteúdo de um produto em específico e uma review específica, poderíamos criar essa estrutura de pastas/arquivos:

**products/[productId]/reviews/[reviewId]/page.tsx**

Conseguimos filtrar, pegando o parâmetro da seguinte forma:

TYPESCRIPT

```
interface PageProps {  
  params: { productId: string, reviewId: string }  
}  
  
export default function Review({ params }: PageProps)  
{  
  return (  
    <div>  
      <h1>Produto {params.productId}</h1>  
      <h2>Review { params.reviewId }</h2>  
    </div>  
  )  
}
```

# Rotas agrupadas

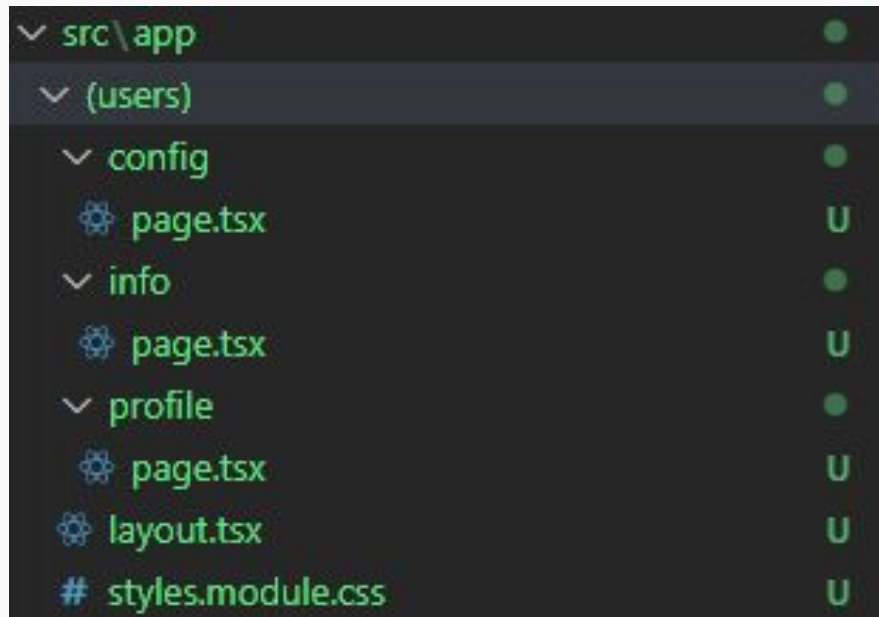
No diretório do aplicativo, pastas aninhadas normalmente são mapeadas para caminhos de URL. No entanto, você pode marcar uma pasta como um Grupo de Rotas para evitar que a pasta seja incluída no caminho da URL da rota.

Isso permite que você organize seus segmentos de rota e arquivos do projeto em grupos lógicos sem afetar a estrutura do caminho da URL.

Os grupos de rotas são úteis para:

- Organizar rotas em grupos, por exemplo, por seção do site, intenção ou equipe.
- Habilitar layouts aninhados no mesmo nível do segmento de rota:
- Criar vários layouts aninhados no mesmo segmento, incluindo vários layouts raiz.
- Adicionar um layout a um subconjunto de rotas em um segmento comum.

Um grupo de rotas pode ser criado envolvendo o nome de uma pasta entre parênteses: (nomeDaPasta).





# Layout e layouts aninhados

# Arquivos de layout e metadados

Como falado anteriormente, o arquivos de layout é o primeiro a ser carregado em uma aplicação Next, e não somente na aplicação geral, mas em todos os apps, podemos utilizá-lo. Podemos também aninhá-los. Essa estratégia serve para aplicar condições e estilos para todas as páginas subsequentes à primeira, em uma cascata hierárquica.





# Metadados

# Metadados

TYPESCRIPT

```
import type { Metadata } from "next"

export const metadata: Metadata = {
  title: {
    absolute: "", // Ignora o template
    default: "", // Nome base para qualquer template que não tenha título
    template: "%s | WEB II" // Qualquer título em outras páginas será colocado
no lugar do %s
  },
  description: "Lista de produtos",
}
```

# Links



# Estilização

# Estilos locais

O Next.js possui suporte integrado para CSS Modules usando a extensão `.module.css`.

Os CSS Modules localmente limitam o escopo do CSS, criando automaticamente um nome de classe único. Isso permite que você use o mesmo nome de classe em diferentes arquivos sem se preocupar com colisões. Esse comportamento torna os CSS Modules a maneira ideal de incluir CSS ao nível do componente.

src/app/dashboard/styles.module.css

```
.dashboard {  
  padding: 24px;  
  background: rebeccapurple;  
  color: white;  
}
```

src/app/dashboard/layout.tsx

```
import styles from './styles.module.css'  
  
export default function DashboardLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return <section  
    className={styles.dashboard}>{children}</se  
ction>  
}
```

# Estilos globais

Estilos globais podem ser importados em qualquer layout, página ou componente dentro do diretório do aplicativo.

Por exemplo, considere um CSS chamado `src/app/global.css` ou qualquer outro CSS que você queira criar neste nível. Ao importá-lo no `layout.tsx` base, que fica na pasta `src/app`, você aplicará este estilo à todas as páginas de sua aplicação.

src/app/global.css

```
body {  
  padding: 20px 20px 60px;  
  max-width: 680px;  
  margin: 0 auto;  
}
```

src/app/layout.tsx

```
// Todos estes styles serão aplicados em todas as  
páginas da aplicação  
import './global.css'  
  
export default function RootLayout({  
  children,  
}): {  
  children: React.ReactNode  
}) {  
  return (  
    <html lang="en">  
      <body>{children}</body>  
    </html>  
  )  
}
```

# Templates



# Arquivos de template

Os templates são semelhantes aos layouts no sentido de que envolvem um layout ou página filho. Ao contrário dos layouts que persistem em todas as rotas e mantêm o estado, os templates criam uma nova instância para cada um de seus filhos na navegação. Isso significa que quando um usuário navega entre rotas que compartilham um template, uma nova instância do filho é montada, os elementos DOM são recriados, o estado não é preservado nos Componentes do Cliente e os efeitos são ressincronizados.

Pode haver casos em que você precise desses comportamentos específicos, e os templates seriam uma opção mais adequada do que os layouts. Por exemplo:

- Para ressincronizar o `useEffect` na navegação.
- Para redefinir o estado de um Cliente Componentes filho na navegação.

Um template pode ser definido exportando um componente React padrão de um arquivo `template.tsx`. O componente deve aceitar uma prop `children`.

Resumindo, para re-renderizar todo o componente da página, nós devemos renomear o arquivo **layout.tsx** para **template.tsx**.



Manipulação de erros



# Componentes do cliente

# Componentes do cliente

Os Componentes do Cliente permitem que você escreva interfaces de usuário interativas que são pré-renderizadas no servidor e podem usar JavaScript do cliente para serem executadas no navegador.

## **Benefícios da Renderização do Cliente**

Existem alguns benefícios em fazer o trabalho de renderização no cliente, incluindo:

- **Interatividade:** Os Componentes do Cliente podem usar estado, efeitos e ouvintes de eventos, o que significa que podem fornecer feedback imediato ao usuário e atualizar a interface do usuário.
- **APIs do navegador:** Os Componentes do Cliente têm acesso a APIs do navegador, como geolocalização ou localStorage.

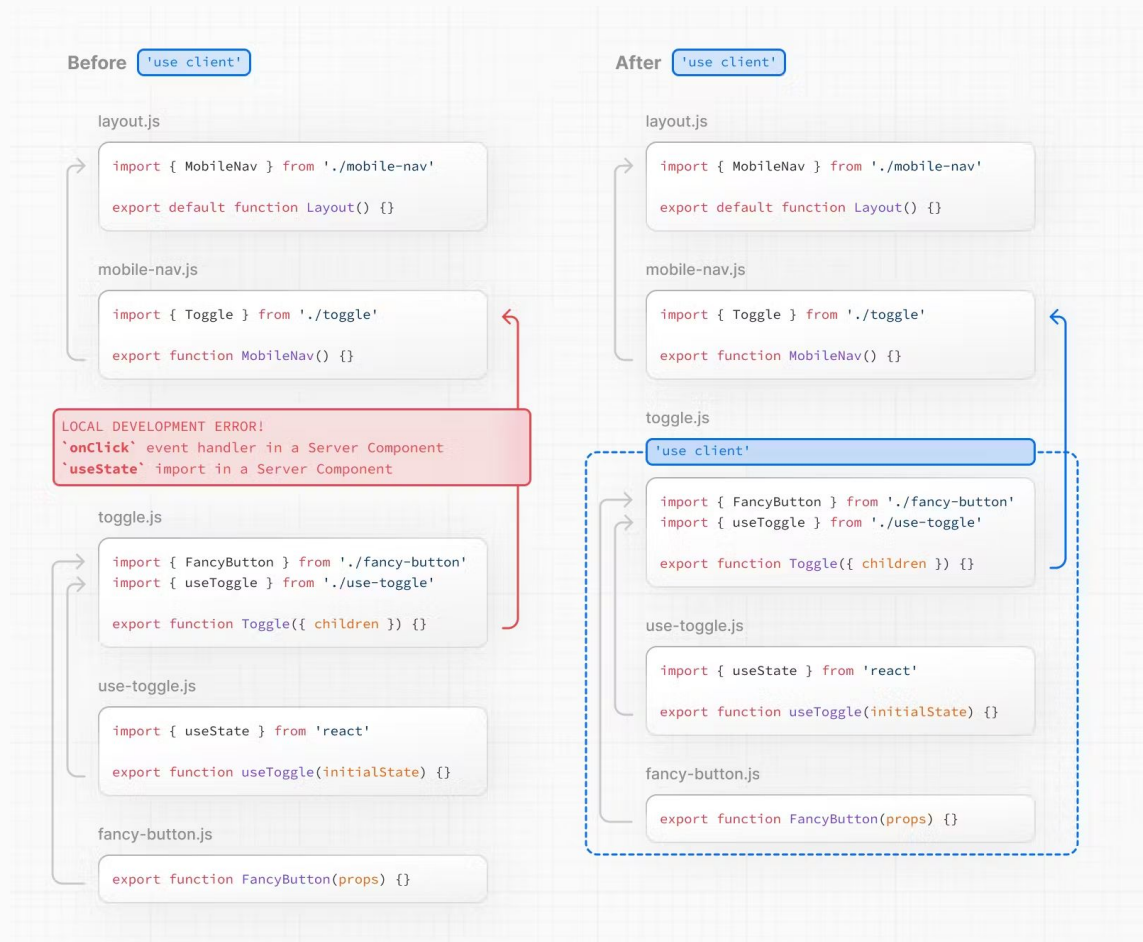
## **Usando Componentes do Cliente no Next.js**

Para usar Componentes do Cliente, você pode adicionar a diretiva "use client" do React no topo de um arquivo, acima de suas importações.

"use client" é usado para declarar um limite entre módulos de Componente do Servidor e do Cliente. Isso significa que, ao definir um "use client" em um arquivo, todos os outros módulos importados nele, incluindo componentes filhos, são considerados parte do pacote do cliente.

# Componentes do cliente

O diagrama ao lado mostra que o uso de `onClick` e `useState` em um componente aninhado (`toggle.js`) causará um erro se a diretiva "use client" não estiver definida. Isso ocorre porque, por padrão, todos os componentes no Roteador do Aplicativo são Componentes do Servidor, onde essas APIs não estão disponíveis. Ao definir a diretiva "use client" em `toggle.js`, você pode dizer ao React para entrar no limite do cliente onde essas APIs estão disponíveis.



Buscando dados



# Busca de dados do Lado do Cliente

A busca de dados do lado do cliente é útil quando sua página não requer indexação de SEO, quando você não precisa pré-renderizar seus dados, ou quando o conteúdo de suas páginas precisa ser atualizado com frequência. Ao contrário das APIs de renderização do lado do servidor, você pode usar a busca de dados do lado do cliente no nível do componente.

Se feito no nível da página, os dados são buscados em tempo de execução, e o conteúdo da página é atualizado conforme os dados mudam. Quando usado no nível do componente, os dados são buscados no momento da montagem do componente, e o conteúdo do componente é atualizado conforme os dados mudam.

É importante notar que o uso da busca de dados do lado do cliente pode afetar o desempenho de sua aplicação e a velocidade de carregamento de suas páginas. Isso ocorre porque a busca de dados é feita no momento da montagem do componente ou das páginas, e os dados não são armazenados em cache.

# Hooks do React

- **useState:** Gerencia o estado de um componente funcional. Útil para armazenar e atualizar valores de estado, como o estado de um formulário.
- **useEffect:** Executa efeitos secundários em componentes funcionais. Útil para realizar operações após a renderização do componente, como chamadas de API ou manipulação do DOM.
- **useContext:** Permite acessar o contexto global de um componente. Útil para compartilhar dados globalmente entre componentes, evitando props de perfuração.
- **useReducer:** Gerencia o estado complexo de um componente. Útil quando o estado possui uma lógica mais complexa de atualização, como em aplicações com várias ações ou estados.

# Hooks do React

- **useCallback:** Memoriza uma função, evitando recriações desnecessárias. Útil para otimizar o desempenho de funções passadas como propriedades em componentes filho.
- **useMemo:** Memoriza o resultado de uma função, evitando re-cálculos desnecessários. Útil para otimizar o desempenho de cálculos pesados dentro de um componente.
- **useRef:** Retorna um objeto ref mutável. Útil para acessar diretamente elementos do DOM ou manter referências persistentes a valores mutáveis sem causar uma nova renderização.
- **useImperativeHandle:** Permite que os componentes filhos exponham funções imperativas para o componente pai. Útil ao encapsular componentes de terceiros que precisam ser interagidos diretamente.



Dúvidas?

# O que vem depois?

O que devo estudar em seguida?

- [API Routes](#)
- [Otimização](#)
- [Cache](#)
- [Middlewares](#)
- [Testes](#)
- [Autenticação](#)
- [Context](#)

# Referências

Next.js Disponível em: <<https://nextjs.org/docs>>

Subreddit Next.js Disponível em: <<https://www.reddit.com/r/nextjs/>>