Next.js - Roteamento

Roteamento no Next.js

O Next.js possui roteamento automático baseado na estrutura de pastas. Cada arquivo dentro da pasta pages se torna uma rota.
Não precisa de configuração extra nenhuma, como no React onde precisamos instalar o pacote react-router.
No Next a estrutura do Framework já nós da o roteamento, mas ele tem uma estrutura bem definida e é isso vamos ver agora

Root

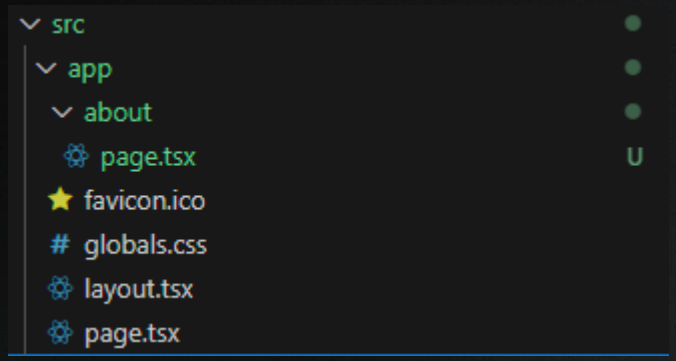
Como já vimos nas aulas anteriores, quando iniciamos um projeto em Next.js a nossa pasta raiz OU a nossa pasta app/ (de acordo com o que foi escolhido no cliente, nos exemplos anteriores usamos a pasta app/ e vamos manter esse padrão) temos dois arquivos que são a porta de entrada para a nossa aplicação: o layout.tsx e o page.tsx.

O que faz esses arquivos serem a entrada da nossa aplicação não é a nomenclatura deles (vamos ver que o nome page e layout vai se repetir muito) mas sim a localização dos arquivos na estrutura de pastas.

O roteamento do Next é automático e funciona toda pela estrutura de pastas da nossa aplicação.

Rotas estaticas

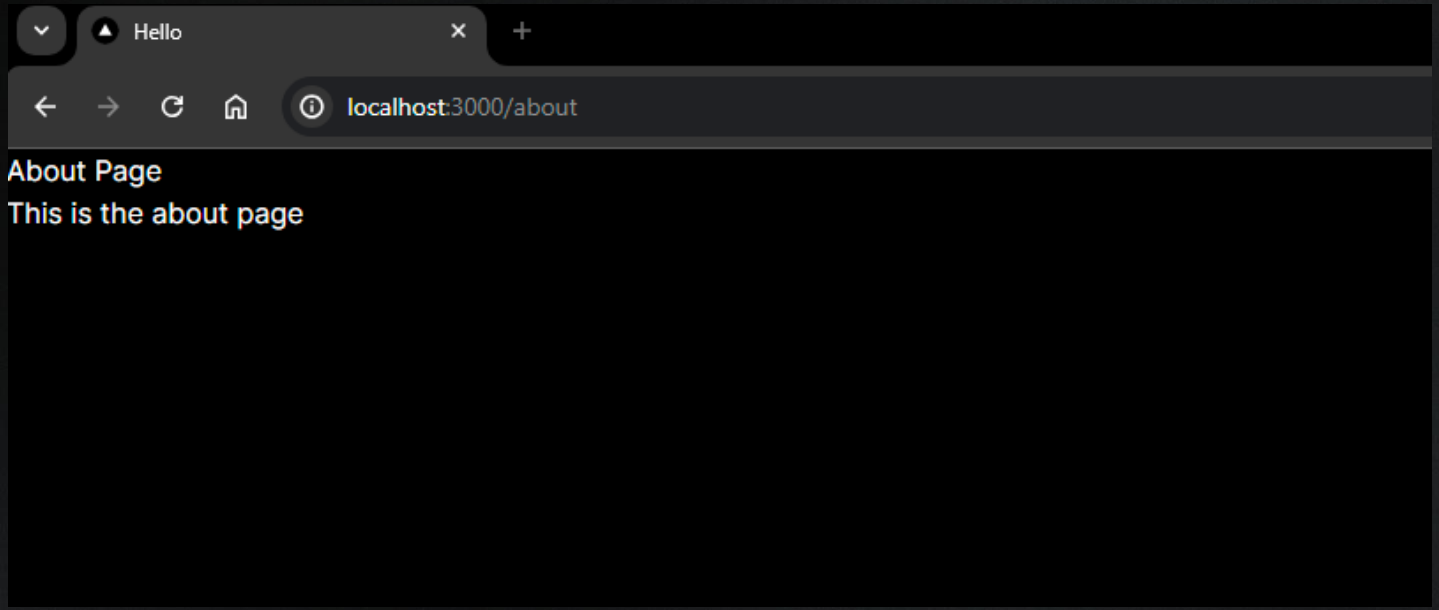
1. Assim como no React padrão, conseguimos criar notas nomeadas com um nome fixo. Vamos criar uma pasta com o nome de about, e dentro dela um arquivo page.tsx




```tsx
const AboutPage = () => {
  return (
    <div>
      <h1>About Page</h1>
      <p>This is the about page</p>
    </div>
  )
}

export default AboutPage
```

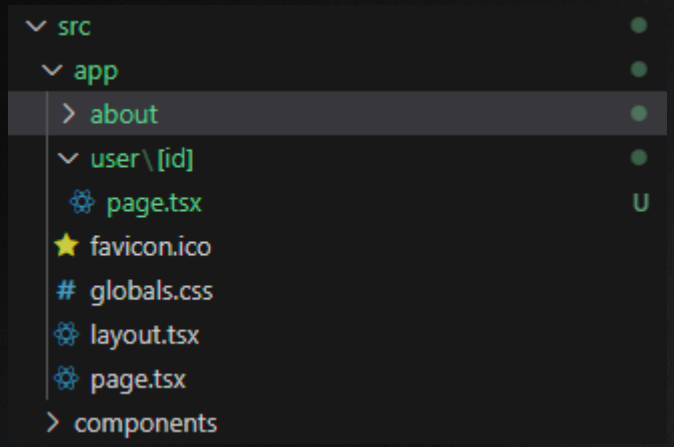
Rotas estaticas

2. Agora vamos acessar /about no nosso navegador com a aplicação rodando (lembre-se: para rodar a aplicação, rodar o comando npm run dev na pasta do projeto). Se tudo estiver certo vamos ver a nossa nova página



Rotas dinâmicas

1. Agora vamos criar uma rota com parâmetro. La no React Router nos usávamos os : (dois pontos) para indicar que uma rota poderia ter uma variavel. No Next é diferente, mas é tão simples quanto: Vamos criar uma pasta entre [] (colchetes) e colocar o nome do parâmetro dentro. Vamos criar uma pasta user, dentro dela uma pasta [id] e dentro o page.tsx

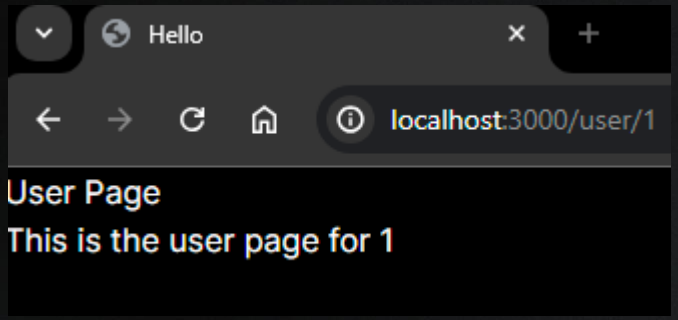


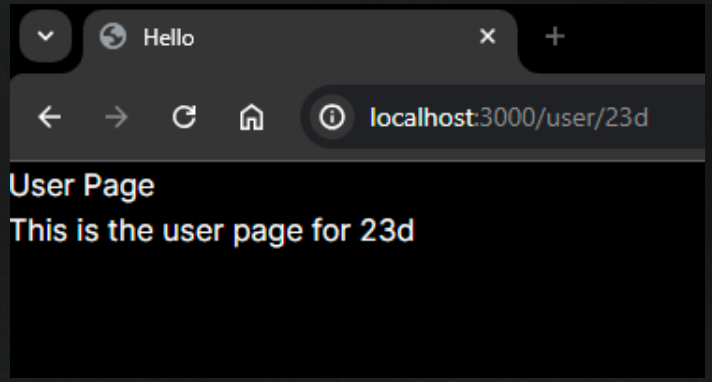
src > app > user > [id] > page.tsx > ...

```tsx
interface UserPageProps {
  params: {
    id: string
  }
}

const UserPage = ({ params }: UserPageProps) => {
  const { id } = params

  return (
    <div>
      <h1>User Page</h1>
      <p>This is the user page for {id}</p>
    </div>
  )
}

export default UserPage
```

Rotas dinâmicas

2. O que fizemos de diferente: Basicamente estamos recebendo o parâmetro da nossa URL (id) via Props. O Next passa isso de forma automática.
3. Vamos acessar /user/1 na nossa aplicação. Podemos colocar qualquer coisa no lugar do numero 1 e ir testando o nosso ID





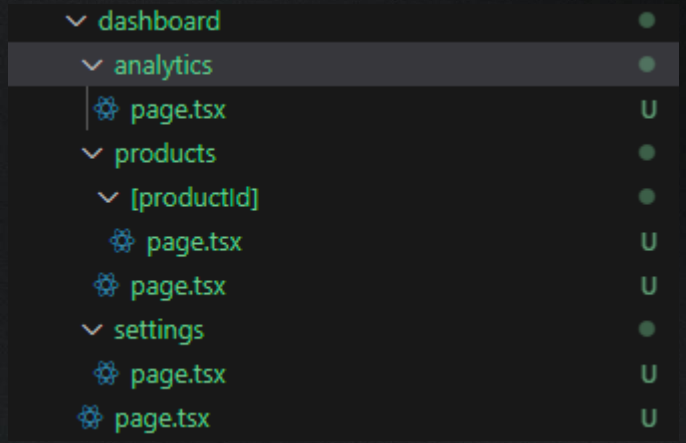
Rotas Aninhadas

Rotas aninhadas permitem a criação de uma estrutura de navegação hierárquica, onde uma rota pode ter sub-rotas. Isso é especialmente útil em aplicações mais complexas, onde diferentes seções da aplicação podem ter seus próprios conjuntos de páginas.

No exemplo anterior acabamos criando uma rota aninhada quando criamos a pasta user/ e dentro dela colocamos a outra pasta [id].

Vamos criar mais algumas páginas aninhando rotas em mais níveis diferentes.

Vamos sempre usar a estrutura básica do componente AboutPage, mudando somente o texto
e o nome do componente.



Rotas Aninhadas

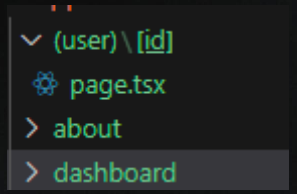
Agora vamos testas as seguintes rotas:

- /dashboard
- /dashboard/settings
- /dashboard/analytics
- /dashboard/products
- /dashboard/products/1
- /bashboard/products/abcde

Agrupadores

Mas caso queiramos usar as pastas apenas para organizar os nossos arquivos, sem alterar o roteamento padrão do Next é possível? Sim, basta criamos a pasta entre () (parênteses). Vamos voltar para a nossa página de usuário. Hoje ela é acessada como /user/id, sendo o ID o valor do ID do banco. Vamos refatorar nossa estrutura para ser /id, removendo o /user.

Nossa estrutura de pastas vai ficar assim:



Assim ao acessar /user/1 vamos cair na página de 404. E ao acessar /1 vamos cair na página de usuário.

Layout

Um layout é uma interface de usuário compartilhada entre várias rotas. Na navegação, os layouts preservam o estado, permanecem interativos e não são renderizados novamente. Os layouts também podem ser aninhados.

Você pode definir um layout por padrão exportando um componente React de um arquivo layout.jsx|js|ts|tsx. O componente deve aceitar uma propriedade children que será preenchida com um layout filho (se existir) ou uma página durante a renderização.

Layout - RootLayout

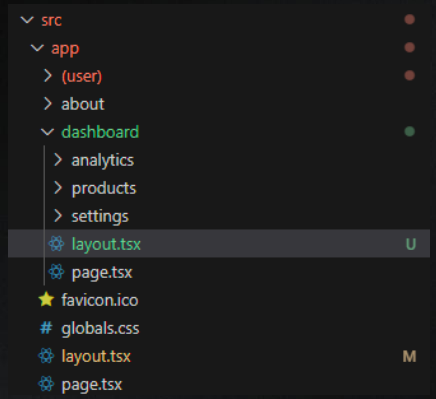
O layout raiz (Root Layout) é definido no nível superior do diretório do aplicativo e se aplica a todas as rotas. Este layout é obrigatório e deve conter tags html e body, permitindo modificar o HTML inicial retornado do servidor.

```
...
import type { Metadata } from "next"
import { Inter } from "next/font/google"
import "./globals.css"

const inter = Inter({ subsets: ["latin"] })

export const metadata: Metadata = {
  title: "Hello",
  description: "Generated by create next app",
}

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode
}>) {
  return (
    <html lang="en">
      <body className={inter.className}>{children}</body>
    </html>
  )
}
```

Layout

Por padrão, os layouts na hierarquia de pastas são aninhados, o que significa que eles agrupam os layouts filhos por meio de suas propriedades filhos. Você pode aninhar layouts adicionando layout.js dentro de segmentos de rota específicos (pastas).

Vamos criar um layout para o nosso Dashboard. Para isso vamos criar um arquivo layout.tsx dentro da pasta dashboard. Depois vamos acessar a URL /dashboard e suas rotas aninhadas



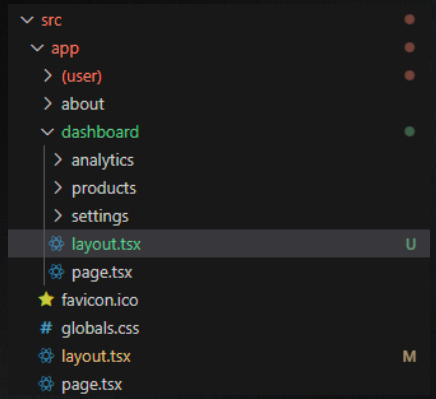
src > app > dashboard > layout.tsx > default

```tsx
interface DashboardLayoutProps {
  children: React.ReactNode
}

const DashboardLayout = ({ children }: DashboardLayoutProps) => {
  return (<div className="dashboard-layout">
    <header>
      <h1>Dashboard</h1>
    </header>
    {children}
  </div>)
}

export default DashboardLayout
```

Layout

Por padrão, os layouts na hierarquia de pastas são aninhados, o que significa que eles agrupam os layouts filhos por meio de suas propriedades filhos. Você pode aninhar layouts adicionando layout.js dentro de segmentos de rota específicos (pastas).

Vamos criar um layout para o nosso Dashboard. Para isso vamos criar um arquivo layout.tsx dentro da pasta dashboard. Depois vamos acessar a URL /dashboard e suas rotas aninhadas



src > app > dashboard > layout.tsx > default

```tsx
interface DashboardLayoutProps {
  children: React.ReactNode
}

const DashboardLayout = ({ children }: DashboardLayoutProps) => {
  return (<div className="dashboard-layout">
    <header>
      <h1>Dashboard</h1>
    </header>
    {children}
  </div>)
}

export default DashboardLayout
```

Layout

Vamos melhorar o nosso layout da Dashboard com Tailwind e criar um menu de navegação.

Vamos usar o componente Link do Next para fazer a navegação de paginas.

Também vamos criar uma página para um usuário (/:id, no caso /1) para visualizar o que acontece quando saímos de um layout aninhado

```tsx
const DashboardLayout = ({ children }: DashboardLayoutProps) => {
  return (<div className="dashboard-layout min-h-screen flex flex-col">
    <header className="bg-gray-800 text-white py-4">
      <h1 className="text-3xl text-center">Dashboard</h1>
    </header>
    <div className="flex flex-1">
      <aside className="bg-gray-800 text-white w-64 p-4">
        <nav>
          <ul className="space-y-2">
            <li>
              <Link href="/dashboard" className="block py-2 px-4 rounded hover:bg-gray-700">
                Home Dashboard
              </Link>
            </li>
            <li>
              <Link href="/dashboard/products" className="block py-2 px-4 rounded hover:bg-gray-700">
                Products
              </Link>
            </li>
            <li>
              <Link href="/dashboard/analytics" className="block py-2 px-4 rounded hover:bg-gray-700">
                Profile
              </Link>
            </li>
            <li>
              <Link href="/dashboard/settings" className="block py-2 px-4 rounded hover:bg-gray-700">
                Settings
              </Link>
            </li>
            <li>
              <Link href="/1" className="block py-2 px-4 rounded hover:bg-gray-700">
                My Profile
              </Link>
            </li>
          </ul>
        </nav>
      </aside>
      <div className="content flex-1 p-4">
        {children}
      </div>
    </div>
  </div>
```

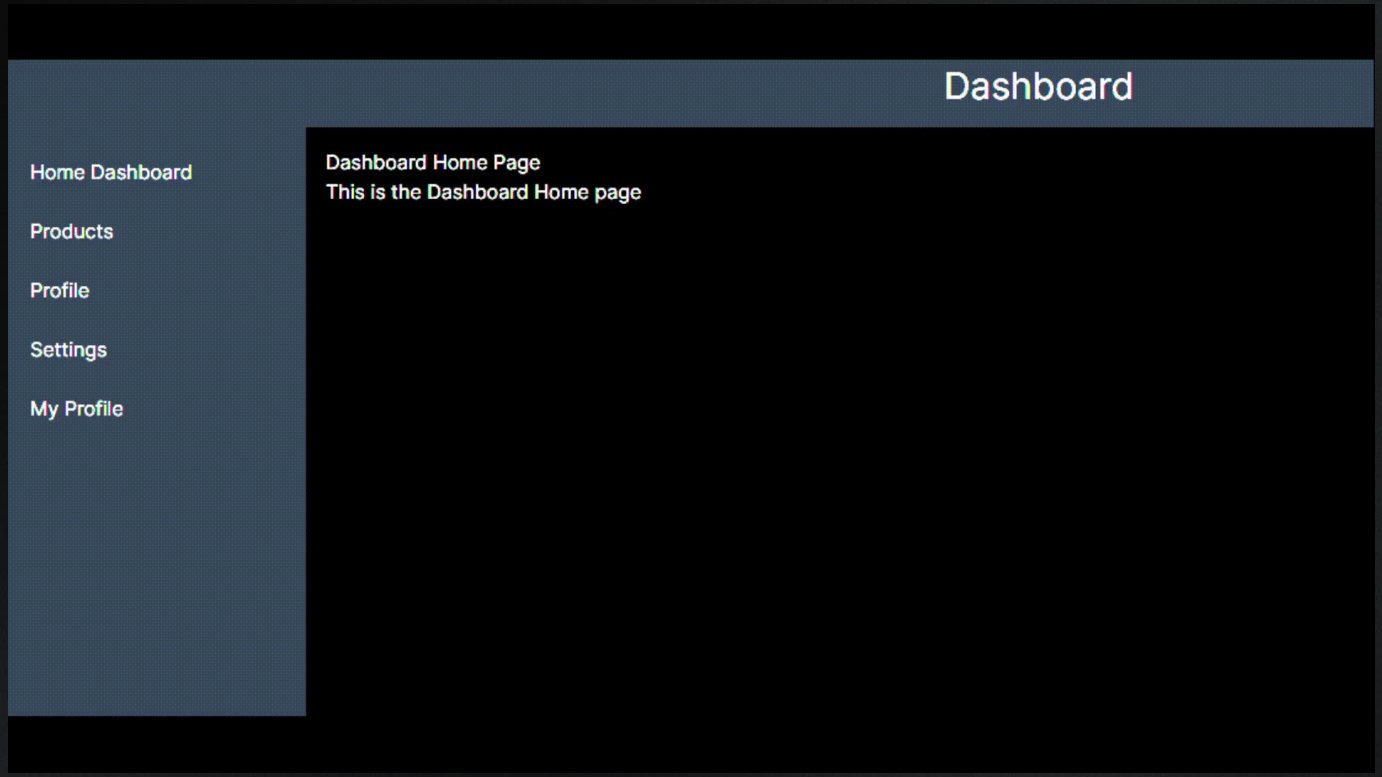
Layout

Também podemos criar um layout para a nossa tela de usuário, esse sendo diferente do Dashboard. Vamos criar um arquivo layout.tsx dentro da nossa pasta (users)/[id]/

```tsx
import Link from "next/link"

interface UserLayoutProps {
  children: React.ReactNode
}

const UserLayout = ({ children }: UserLayoutProps) => {
  return (<div className="dashboard-layout min-h-screen flex flex-col">
    <header className="bg-gray-800 text-white py-4">
      <h1 className="text-3xl text-center">User</h1>
    </header>
    <div className="flex flex-1">
      <aside className="bg-gray-800 text-white w-64 p-4">
        <nav>
          <ul className="space-y-2">
            <li>
              <Link href="/dashboard" className="block py-2 px-4 rounded hover:bg-gray-700">
                To restrict Area
              </Link>
            </li>
            <li>
              <Link href="/1" className="block py-2 px-4 rounded hover:bg-gray-700">
                My Profile
              </Link>
            </li>
          </ul>
        </nav>
      </aside>
      <div className="content flex-1 p-4">
        {children}
      </div>
    </div>
  </div>
  )
}

export default UserLayout
```

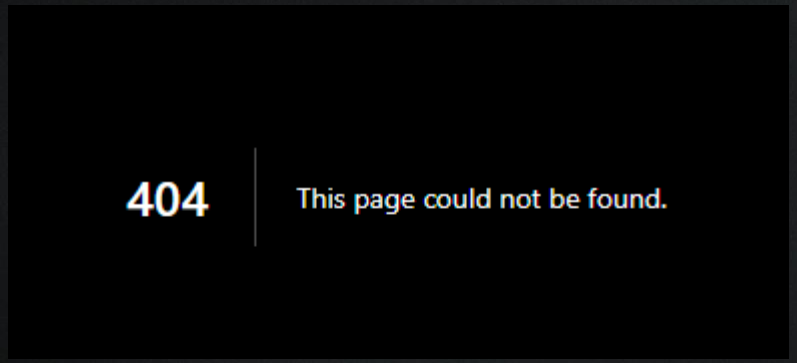
Layout

Resultado



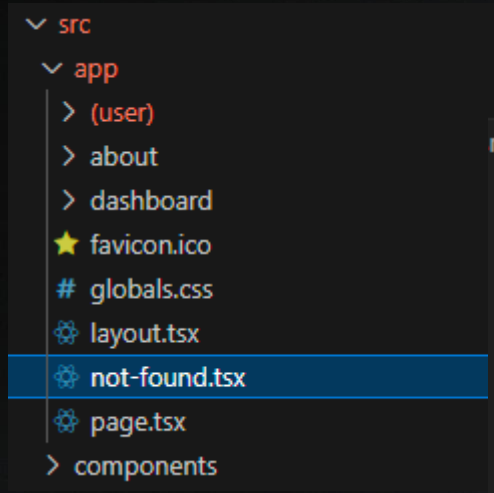
Error Page

Dentro do Roteamento automático ainda podemos criar páginas de erro. Do mesmo jeito que criamos nossas pages e layouts. Vamos primeiro ver a clássica página de 404, quando o usuário tenta acessar uma rota que não existe. Na aplicação padrão o Next já tem uma página criada por default:



Error Page

Vamos deixar criar uma nova, criando um arquivo not-found.tsx dentro da nossa pasta app/

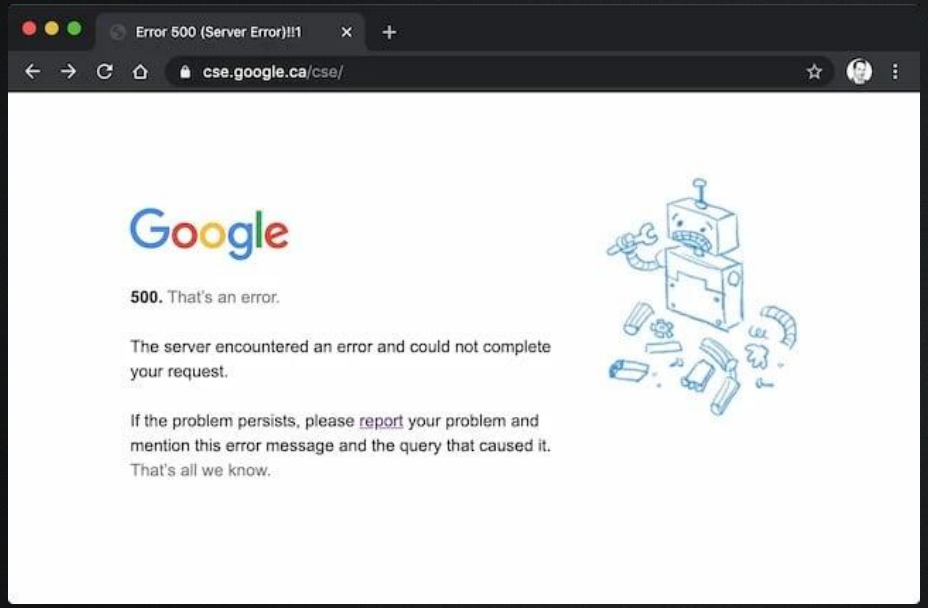


rc > app > not-found.tsx > default

```tsx
const NotFoundDashboard = () => {
  return (
    <div className="flex items-center justify-center h-screen">
      <div className="text-center">
        <h1 className="text-4xl font-bold text-white-900">404</h1>
        <p className="text-white-700">Page not found</p>
      </div>
    </div>
  )
}

export default NotFoundDashboard
```

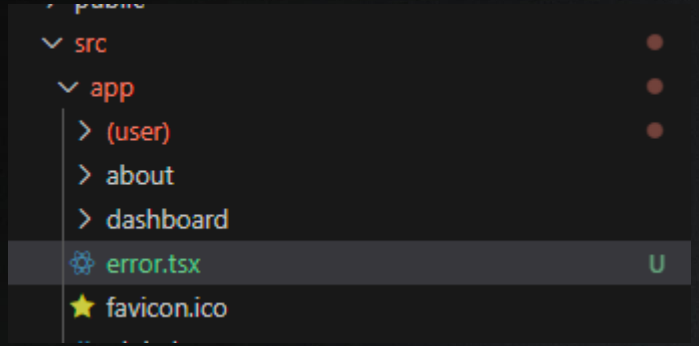
Error Page

Também conseguimos criar uma página para erros genéricos. Geralmente usamos essa página quando acontece algum erro de Javascript ou algum serviço volta uma resposta inesperada que faz com que a nossa aplicação pare de funcionar.
Um exemplo comum é quando o Google enfrenta alguma instabilidade e é exibido a página abaixo:



Error Page

Vamos deixar criar uma página de erro genérico nova, criando um arquivo error.tsx dentro da nossa pasta app/. A diferença aqui é que as páginas do tipo error só são executadas no Client Side, nunca no Server Side. Então precisamos usar a string “use client” no começo do nosso arquivo.



src > app > error.tsx > ...

```tsx
"use client"

const NotExpectedErrorPage = () => {
  return (
    <div className="flex items-center justify-center h-screen">
      <div className="text-center">
        <h1 className="text-4xl font-bold text-white-900">500</h1>
        <p className="text-white-700">Not Expected Error</p>
      </div>
    </div>
  )
}

export default NotExpectedErrorPage

```

Error Page

Agora vamos testar o nosso comportamento de erro. Vamos escrever propositalmente um componente com erro para ele quebrar nossa aplicação. Dentro da pasta components/button/button.tsx esse vai ser o nosso componente com uma variavel inexistente de forma proposital. Vamos usar ele dentro da nossa pagina settings.tsx

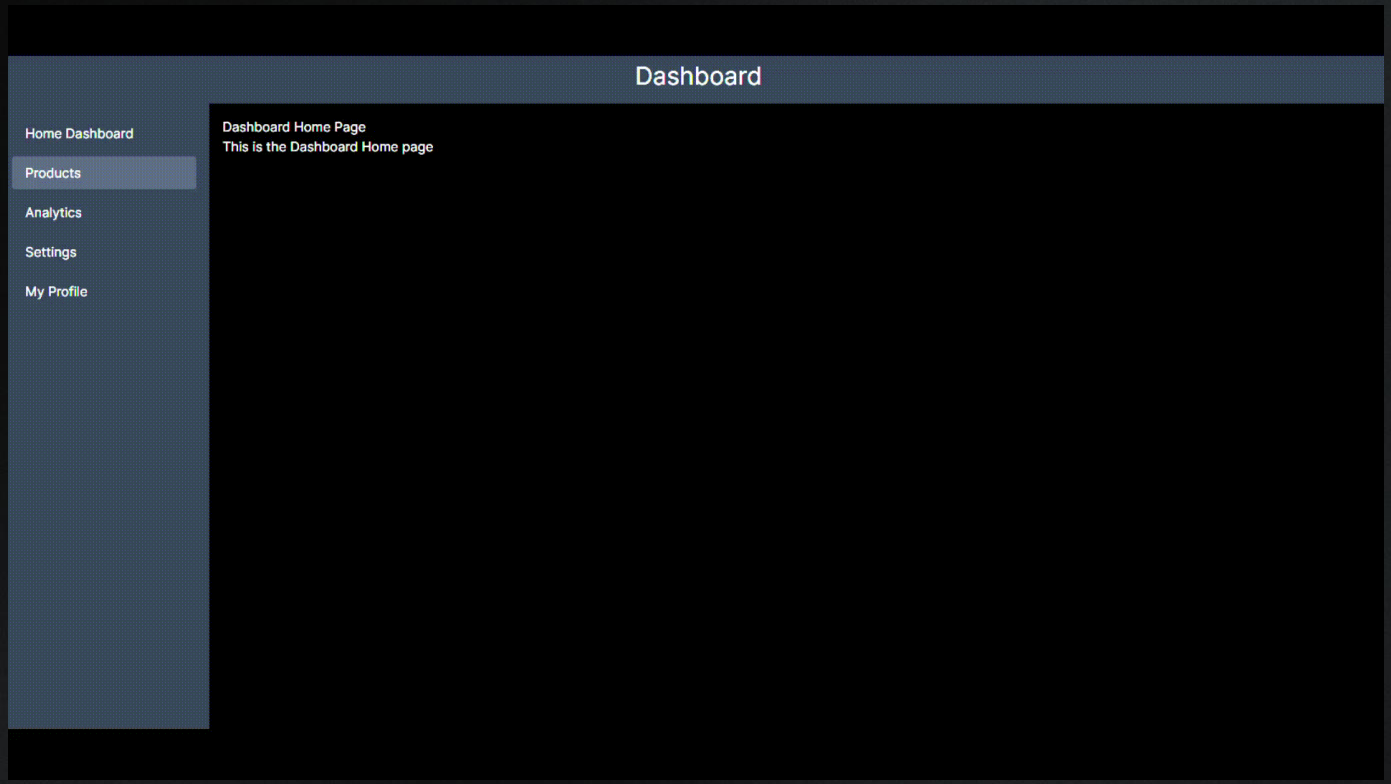
```
You, 1 second ago | 1 author (You)
1  const Button = () => {        You, 1 seco
2    try {
3      console.log(invalidVariable)
4    } catch (error) {
5      console.error(error)
6      throw error
7    }
8
9    return <button>Click me</button>
10 }
11
12 export default Button
13
```

```
You, 1 second ago | 1 author (You)
1  import Button from "@/components/Button/Button"
2
3  const SettingsPage = () => {
4    return (
5      <div>
6        <h1>Settings Page</h1>
7        <p>This is the settings page</p>
8        <Button></Button>
9      </div>
10   )
11 }
12
13 export default SettingsPage
14
```

Error Page

FIAP

Resultado



Navegação

Componente Link

<Link> é um componente React que estende o elemento HTML <a> para fornecer pré-busca e navegação do lado do cliente entre rotas. É a principal forma de navegar entre rotas em Next.js. Ele funciona tanto para o ClientSide quanto para o ServerSide

Documentação: https://nextjs.org/docs/app/api-reference/components/link

```
1   import Link from "next/link"
```

```
<Link href="/dashboard">
  Home Dashboard
</Link>
```

Navegação

redirect

A função de redirecionamento (redirect) permite redirecionar o usuário para outro URL. Você pode chamar o redirecionamento em componentes do servidor, manipuladores de rota e ações do servidor. Deve ser usado somente em elementos do lado do servidor, como componentes com a string “use server”.
Documentação: https://nextjs.org/docs/app/building-your-application/routing/redirecting#redirect-function

```tsx
'use server'
import { permanentRedirect } from 'next/navigation'

const redirectServer = ({ path }: { path: string }) => {
  permanentRedirect(path)
}

export default redirectServer
```

hook useRouter

Faz o redirecionamento do ClientSide, semelhante ao uso do useNavigate do React. Deve ser usado somente em componentes que tenham ação no client.
Documentação: https://nextjs.org/docs/app/building-your-application/routing/redirecting#userouter-hook

```tsx
You, 1 second ago | 1 author (You)
'use client'

import { useRouter } from 'next/navigation'

export default function Button() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      To Dashboard
    </button>
  )
}
```

Exercicios

Vamos continuar o nosso sistema de blog.
Vamos criar algumas páginas e melhorar a nossa estrutura.
Sugestão de Layout:

1. Vamos criar uma rota login, criando uma pasta chamada login.
2. Vamos criar uma outra rota chamada blog. Vai ser a rota responsável por listagem os posts
3. Dento da rota blog, vamos criar o [postId]. Essa será a nossa página da postagem.
4. Vamos criar uma pasta “new” dentro do blog, vai ser nossa página de nova postagem. Um post consistem em titulo, texto e autor da postagem
5. Vamos limpar o conteúdo do nosso page.tsx da raiz. Vamos redirecionar essa URL para /blog via ServerSide
6. Vamos criar a rota login/register para criar nossa página de cadastro
7. Vamos estilizar tudo conforme o Layout (pode inventar o próprio layout também e usar tailwind)

Dúvidas, críticas ou sugestões?


FIAP