

DANIEL SCHMITZ  
2024

**CRIE UM CONTROLE DE  
ESTOQUE DO ZERO**

# NEXT 14 NA PRÁTICA

INCLUI SHADCN E PRISMA

# Next.js na prática

Crie um sistema de estoque do zero com Nextjs 14, Shadcn e Prisma

Daniel Schmitz

Esse livro está à venda em <http://leanpub.com/book-nextjs-react-pt-br>

Essa versão foi publicada em 2024-04-22



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2024 Daniel Schmitz

# Conteúdo

<b>1. Introdução</b>	<b>1</b>
1.1. Sobre PIRATARIA	1
1.2. Suporte	1
1.3. Código Fonte	1
1.4. Instalação	1
1.4.1. Extensões do Visual Studio Code	2
1.5. O Sistema de Estoque	2
1.5.1. Tela de Listagem e Cadastro de Categorias	3
1.5.2. Tela de Listagem e Cadastro de Produtos	4
1.5.3. Tela de Listagem e Cadastro de Estoque	7
1.5.4. Resumo do que Vamos Ver Neste Livro	11
<b>2. O Next.js</b>	<b>13</b>
2.1. Criando o Projeto	13
2.2. App Routes	18
2.3. Limpando a Aplicação	20
2.4. Instalando o shadcn	21
2.4.1. Adicionando o Componente Button	22
2.5. Adicionando um Header	25
2.6. Banco de Dados e Prisma ORM	28
2.6.1. Criando o Schema e Tabelas	29
2.6.2. Criando o Banco de Dados Através do Prisma	32
<b>3. Tela de Categorias</b>	<b>35</b>
3.1. Criando a Tela Inicial de Categorias	35
3.2. Adicionando o Componente Card	36
3.3. Adicionando uma Categoria	40
3.3.1. Criando um Botão para Adicionar Categoria	40
3.3.2. Criando a Página para Adicionar Categoria	41

3.3.3. Estilizando o Título com @apply . . . . .	43
3.3.4. Criando o Formulário . . . . .	43
3.3.5. Criando o Método CreateCategory . . . . .	44
3.3.6. Inserindo Dados na Tabela . . . . .	46
3.3.7. Tratamento de Erros . . . . .	48
3.4. Exibindo uma Tela de “loading” . . . . .	51
3.4.1. Exibindo um “loading” no Botão “Save” . . . . .	52
3.4.2. Corrigindo o Posicionamento do Botão “Save” . . . . .	54
3.5. Exibindo as Categorias . . . . .	56
3.5.1. Instalando o Componente Table . . . . .	57
3.5.2. Exibindo Todas as Categorias . . . . .	57
3.5.3. Utilizando o Componente Table . . . . .	59
3.6. Editando uma Categoria . . . . .	61
3.6.1. Configurando a Rota . . . . .	61
3.6.2. Definido um Tipo para o Parâmetro params . . . . .	63
3.6.3. Obtendo uma Categoria pelo id . . . . .	64
3.6.4. Criando o Formulário para Editar uma Categoria . . . . .	65
3.6.5. Repassando Dados Entre Componentes . . . . .	67
3.6.6. Preenchendo os Dados do Formulário . . . . .	69
3.6.7. Editando a Categoria . . . . .	71
3.6.8. Exibindo Mensagens de Erro . . . . .	75
3.7. Excluindo uma Categoria . . . . .	75
3.7.1. Adicionando o Componente Alert Dialog . . . . .	76
3.7.2. Criando o Componente Delete Dialog . . . . .	79
3.7.3. Método para Remover uma Categoria . . . . .	83
3.7.4. Implementando o Método para Remover uma Categoria . . . . .	84
3.7.5. Implementando Erros . . . . .	85
3.8. Conclusão . . . . .	86

# 1. Introdução

O principal objetivo deste livro é ensinar o framework *Next.js* criando um sistema de estoque contendo uma variedade de telas e funcionalidades.

Em vez de mostrar apenas a teoria do framework, que pode ser facilmente acessada em sua excelente [documentação](#), já começamos no próximo capítulo o desenvolvimento do sistema. Mas antes de ir para o próximo capítulo, é importante preparar o ambiente de desenvolvimento.

## 1.1. Sobre PIRATARIA

Este livro não é gratuito e não deve ser publicado sem autorização, especialmente em sites como *scrib*. Por favor, contribua com os autores para que eles possam investir em mais conteúdo de qualidade. Se você obteve este livro sem comprá-lo, pedimos que leia o ebook e, se acreditar que o livro merece, compre-o e ajude o autor a publicar cada vez mais. Você pode comprar este livro [aqui](#).

## 1.2. Suporte

Se você tiver alguma dúvida, ou encontrar algum erro no código ou na tradução para o inglês, por favor, não hesite em abrir uma ISSUE em nosso repositório:

## 1.3. Código Fonte

Todo o código fonte deste livro está no repositório do github:

## 1.4. Instalação

Nextjs é um framework que requer apenas uma instalação básica: Node. Neste trabalho, estaremos instalando alguns programas a mais para que possamos maximizar nosso aprendizado.

- **Node.js:** Acesse <https://www.nodejs.org> e instale a versão LTS disponível.
- **Git:** Acesse <https://git-scm.com/downloads> e instale o Git. Use as opções padrão e também instale o Git Bash. Git Bash é muito útil em ambientes Windows, vamos usá-lo ao longo deste livro.
- **Visual Studio Code:** Acesse <https://code.visualstudio.com/> e instale este editor de texto que suporta várias linguagens de programação.
- A fonte que usei neste livro foi [JetBrains Mono](#).

### 1.4.1. Extensões do Visual Studio Code

Você precisa instalar algumas extensões que ajudarão no desenvolvimento de Javascript em geral.

- **ESLint:** Integra o ESLint no VS Code. ESLint encontra e corrige problemas no seu código JavaScript.
- **Material Icon Theme:** Obtenha os ícones do Material Design no seu VS Code.
- **Error Lens:** Exibe o erro, se houver, no final da linha.
- **Git Lens:** Exibe informações do git diretamente no seu código.
- **Auto Close Tags:** Fecha automaticamente as tags HTML.
- **Auto Rename Tag:** Renomeia automaticamente as tags HTML.
- **Prettier:** Formate seu código com Prettier.
- **Prisma:** Prisma é um toolkit de banco de dados para Node.js. Use esta extensão para adicionar realce de sintaxe, linting, completar código, formatação, pular para definição e mais para arquivos de Schema Prisma.
- **Tailwind CSS IntelliSense:** Tailwind CSS IntelliSense aprimora a experiência de desenvolvimento com Tailwind fornecendo aos usuários do Visual Studio Code recursos avançados como autocomplete, realce de sintaxe e linting.

Para instalar uma extensão, acesse a aba *Extensões* no Visual Studio Code (Ctrl+Shift+X), procure pelo nome da extensão e clique no botão instalar.

## 1.5. O Sistema de Estoque

Neste livro iremos criar um sistema de estoque que contém as seguintes funcionalidades:

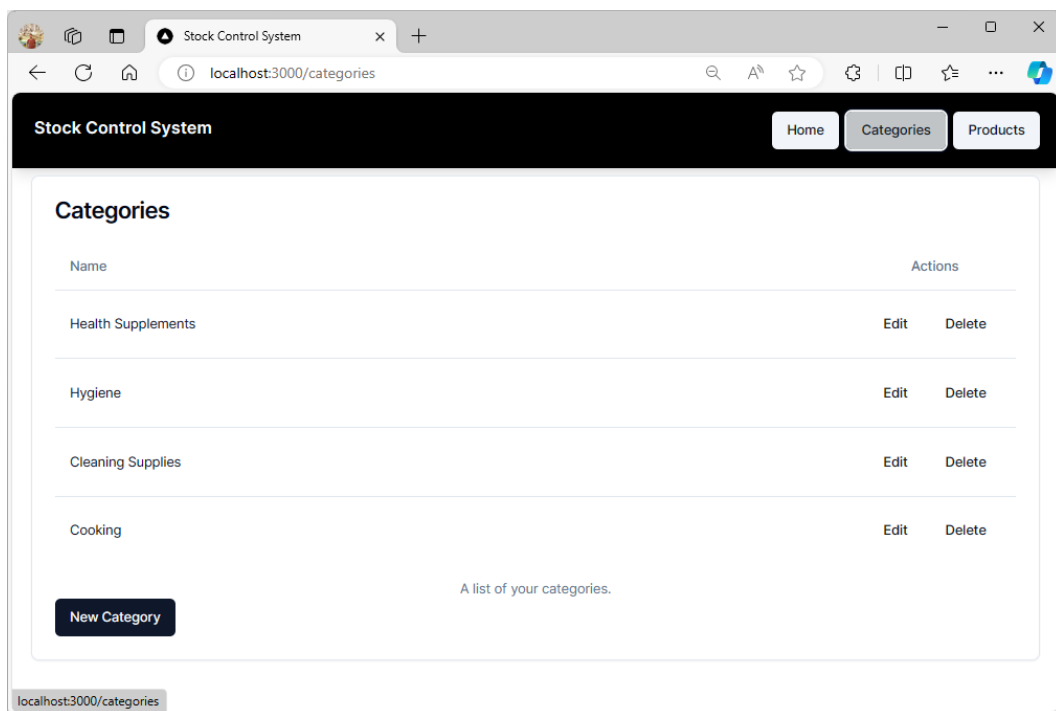
- Cadastro de categorias

- Cadastro de produtos, onde cada produto tem uma categoria
- Cadastro de estoque, onde cada estoque tem um produto
- No cadastro de estoque, iremos inserir a data de validade do produto, e na tela principal do sistema, o objetivo é mostrar o estoque de acordo com a data de vencimento, exibindo itens que estão próximos da data de validade.

Vamos exibir algumas telas que iremos construir.

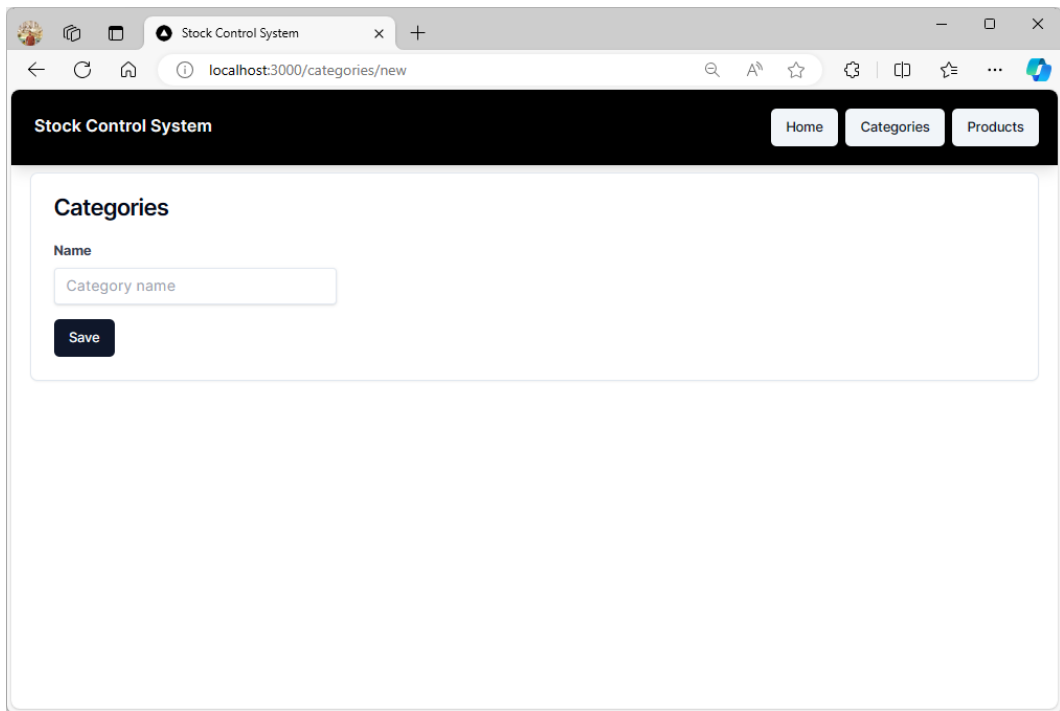
### 1.5.1. Tela de Listagem e Cadastro de Categorias

Esta tela contém apenas um campo, `name`. Ela será a primeira tela que iremos construir e iremos criar um formulário simples utilizando alguns conceitos introdutórios do Nextjs 14. A tela de listagem de categorias é exibida a seguir:



A construção da listagem, o botão para cadastrar uma categoria e a barra superior, todos estes elementos que iremos criar no sistema são do `shadcn ui`, uma biblioteca de componentes do React que permite criar telas responsivas e interativas. Também utilizaremos o Tailwind CSS para estilizar os elementos.

A tela de cadastro de categorias é a seguinte:

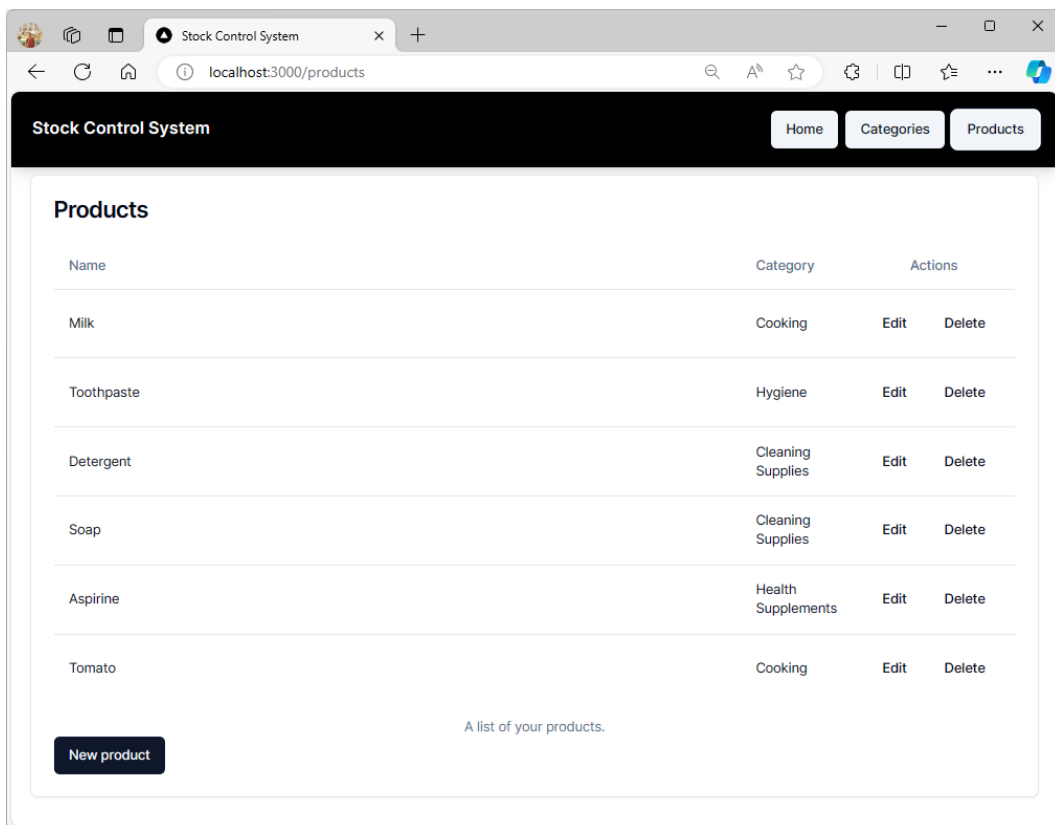


The screenshot displays a web browser window with the title 'Stock Control System'. The address bar shows the URL 'localhost:3000/categories/new'. The page features a dark header with the system name and three navigation buttons: 'Home', 'Categories', and 'Products'. The main content area is titled 'Categories' and contains a form for adding a new category. The form includes a label 'Name', a text input field with the placeholder 'Category name', and a 'Save' button.

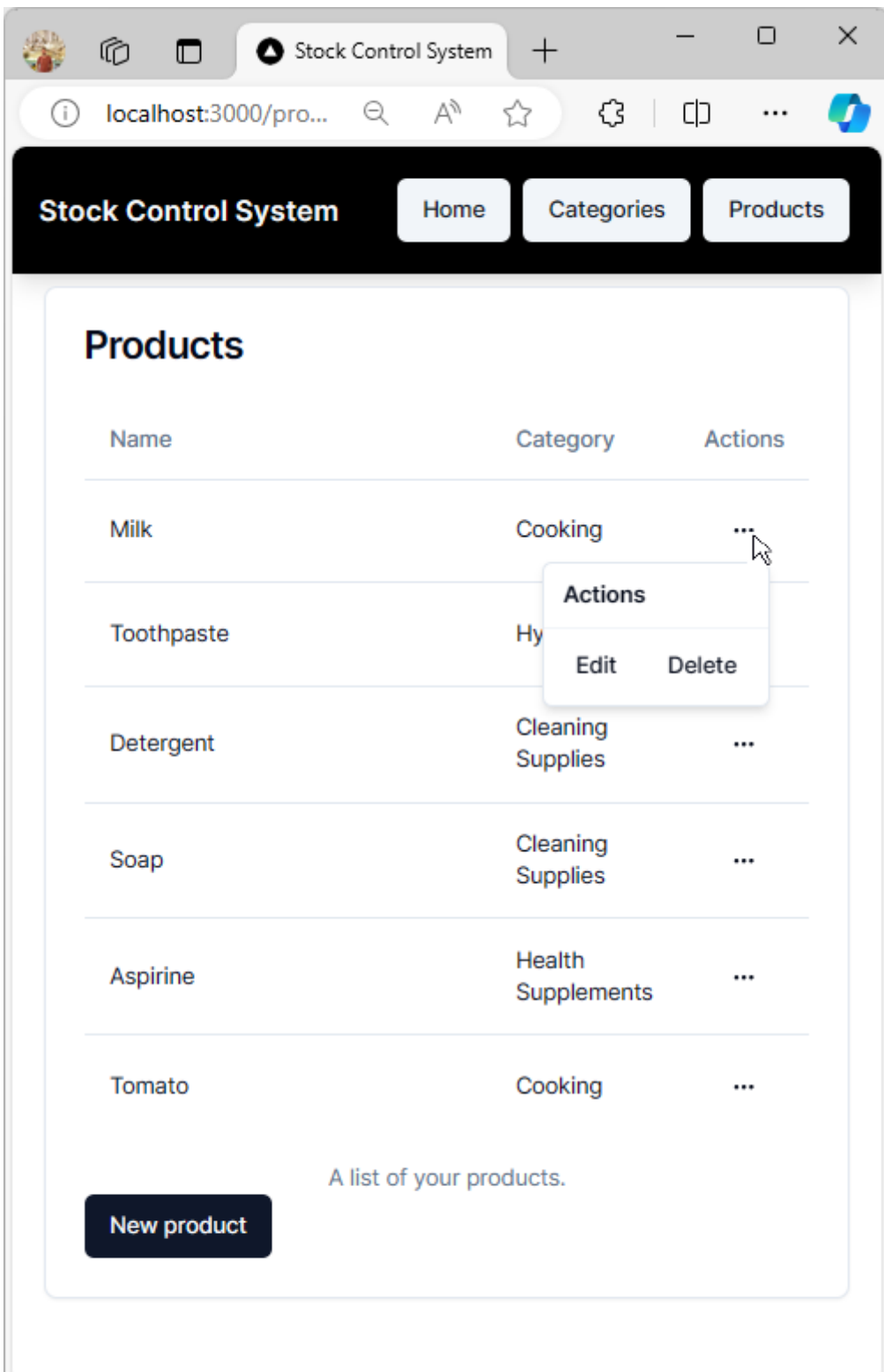
## 1.5.2. Tela de Listagem e Cadastro de Produtos

A tela de listagem de produtos é a seguinte:

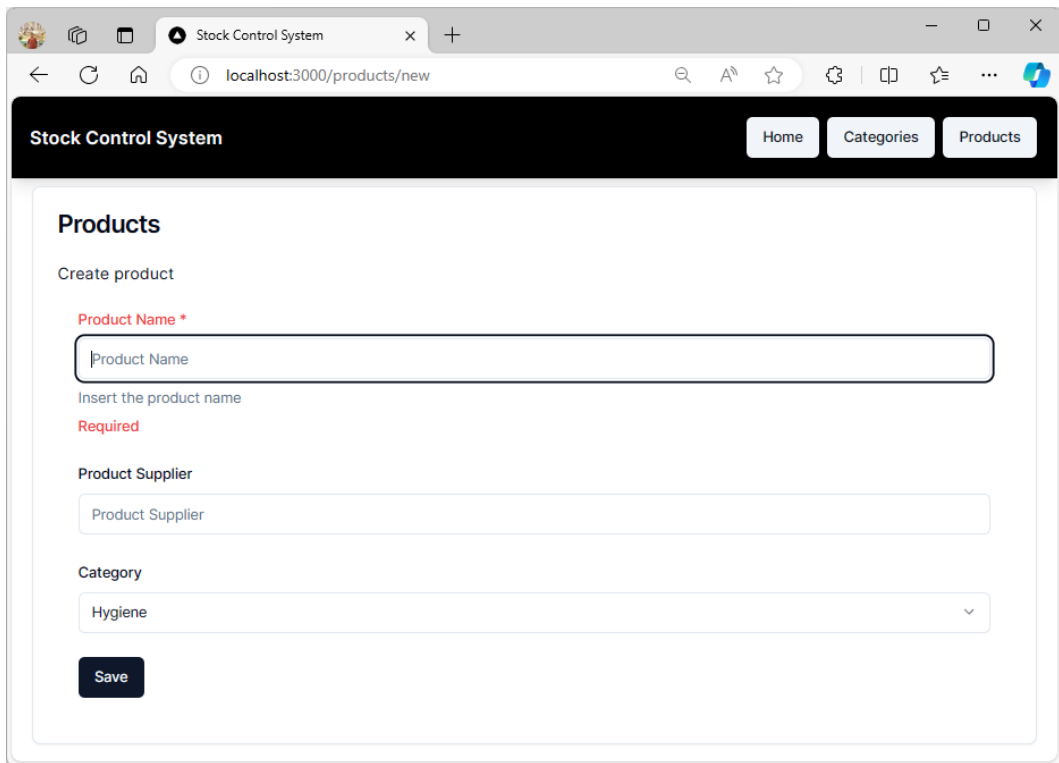




Nesta tela temos uma listagem de produtos utilizando o componente `<Table>` do `shadcn ui`. Aqui vamos exemplificar como criar um componente responsivo, que atende aos requisitos de uma tela mobile. Ao diminuirmos a tela, vemos que os botões de editar e deletar ficam ocultos, e um menu de contexto surge no lugar.



A tela de cadastro de produtos é a seguinte:

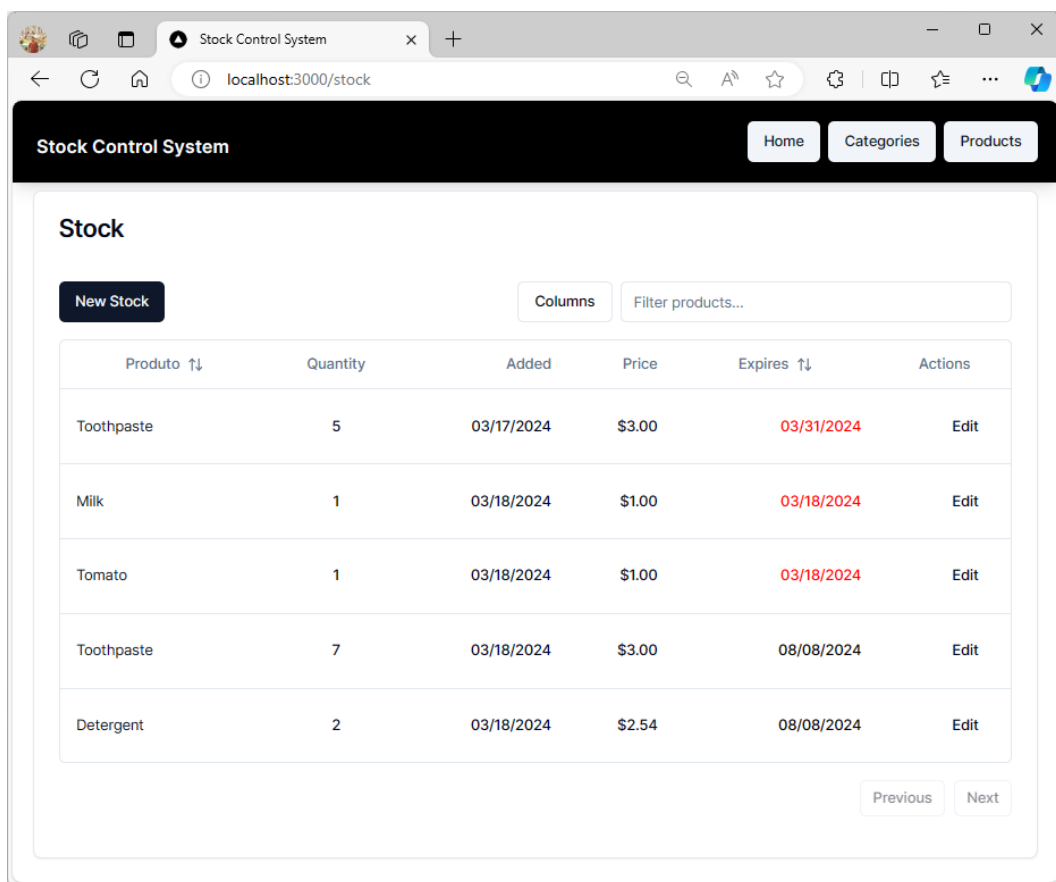


The screenshot shows a web browser window with the title 'Stock Control System'. The address bar shows 'localhost:3000/products/new'. The page has a dark header with the title 'Stock Control System' and three navigation buttons: 'Home', 'Categories', and 'Products'. The main content area is titled 'Products' and contains a 'Create product' section. This section has three input fields: 'Product Name \*' (with a red asterisk and a placeholder 'Product Name'), 'Product Supplier' (with a placeholder 'Product Supplier'), and 'Category' (a dropdown menu with 'Hygiene' selected). Below these fields is a 'Save' button.

Nesta tela de cadastro, iremos introduzir o conceito de `react-hook-forms` em conjunto com os componentes `<Form>` do `shadcn ui`, que são a forma mais completa de criar um formulário, onde os campos podem ser validados de forma automática através da biblioteca `zod`.

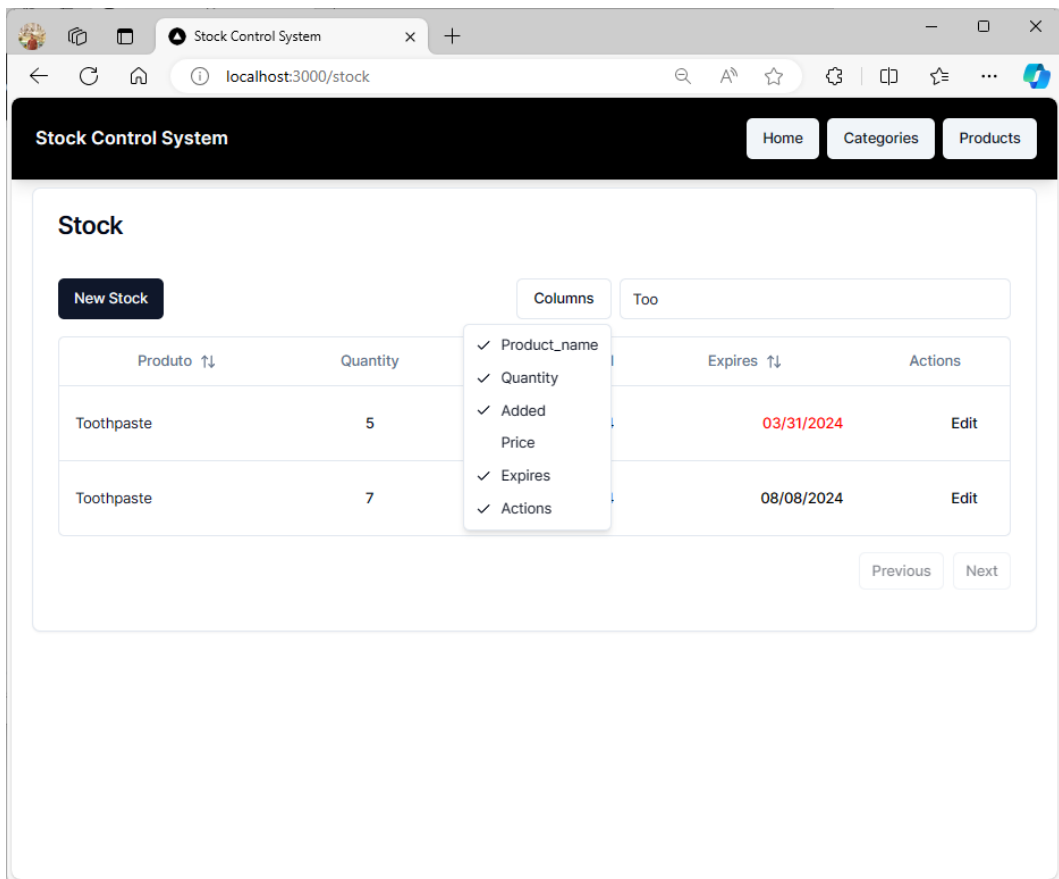
### 1.5.3. Tela de Listagem e Cadastro de Estoque

A tela de listagem de estoque é a seguinte:



Ela apresenta diversos componentes do `shadcn ui`, além do `DataTable` onde é possível implementar ordenação, validação e filtros, tudo totalmente configurável. Veja também que criamos uma coluna para mostrar a data de validade, onde itens que estão próximos da data de validade ficam em destaque.

Na figura a seguir, mostramos algumas funcionalidades do `DataTable` do `shadcn` como a pesquisa e a seleção de campos a serem exibidos na tela.



O botão New Stock é um botão que permite adicionar produtos ao estoque, através do formulário a seguir:

**Stock Control System** Home Categories Products

## Stock

Add Product to Stock

**Product:**  
Toothpaste

**Product Price:**  
3.67  
Insert the product price

**Product Quantity:**  
2  
How many items of this product will be added to stock?

**Product Expires At:**  
July 12th, 2024

Save

Este formulário usa os mesmos conceitos do formulário para adicionar produtos. Na caixa de seleção de produtos, criamos um componente que agrupa os produtos pela categoria, conforme o detalhe da figura a seguir:

Product:

Toothpaste

Health Supplements

Aspirine

Hygiene

✓ Toothpaste

Cleaning Supplies

Detergent

Soap

Cooking

Milk

Tomato

Save

Através da criação deste sistema, iremos aprender os mais diversos conceitos que o Nextjs 14 oferece. Iremos ao longo da criação das telas, arquivos e componentes, exemplificando a teoria do framework de uma forma “mão na massa”.

### 1.5.4. Resumo do que Vamos Ver Neste Livro

A seguir temos uma listagem de tudo que é visto no sistema de estoque:

- A nova versão do Next.js, a 14, traz o novo sistema de roteamento chamado de App Router onde ao invés de usarmos o diretório pages, usamos o diretório app.
- Vamos abordar o App Router em todo o sistema e veremos como criar rotas, telas de erro e de loading, em como usar outlet e children e em como criar rotas dinâmicas, além de rotas privadas.

- Vamos aprender a criar componentes e repassar propriedades para esses componentes
- Vamos aprender a criar `Server Components` e `ClientComponents` entendendo como eles se relacionam entre si.
- Vamos aprender a gerenciar dados através do `Prisma`, uma biblioteca de gerenciamento de dados para `Node.js`, utilizando o poder do `Next.js` e dos `Server Components` para chamar diretamente a consulta e manipulação de dados sem a necessidade de um outro servidor `backend`.
- Vamos utilizar o `Tailwind CSS` para estilizar o sistema de estoque e também para criar telas responsivas.
- Vamos aprender diversos componentes do `shadcn`, em como instalá-los e em como trabalhar com eles criando formulários, tabelas, botões e muito mais.
- Os formulários utilizam `zod` e `react-hook-forms` para validação e tratamento de dados, integrados ao componente `<Form>` do `shadcn ui`.



## 2. O Next.js

O next.js é um framework de desenvolvimento web com o foco em React. Além disto, eles são escritos com o objetivo de tornar o desenvolvimento de aplicativos web mais fácil e menos cansativo. Sua principal funcionalidade é unir tanto o frontend quanto o backend em somente uma única camada, através dos Server Components e Client Components. Assim podemos dizer que o Next.js é um framework full-stack.

Além desta característica, existem dezenas de outras que fazem com que o Next.js seja um framework sólido para o desenvolvimento de aplicações web.

Suas principais funcionalidades são descritas a seguir:

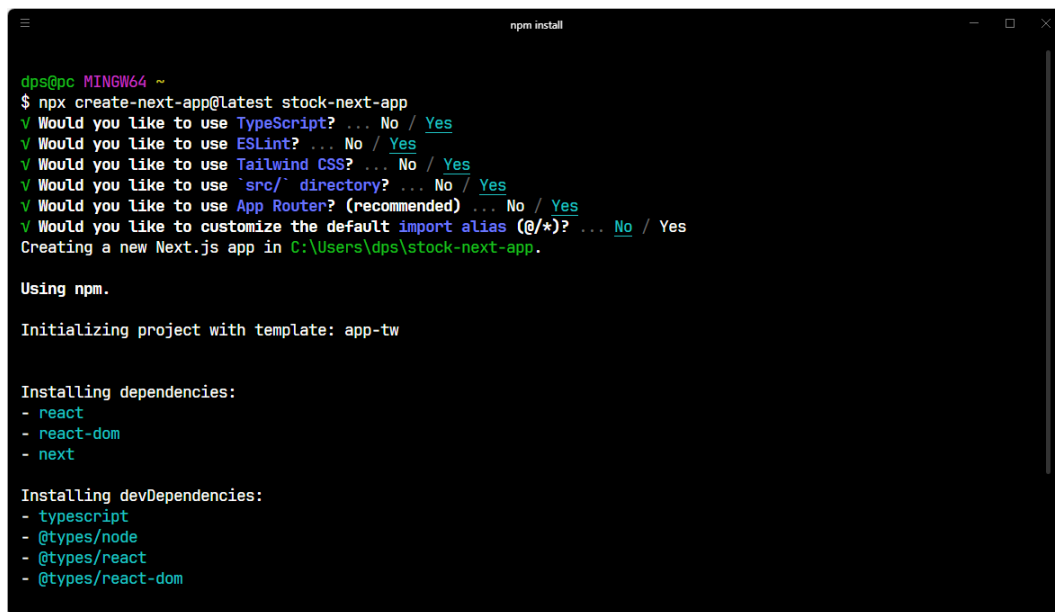
- **Routing:** Permite que os usuários naveguem entre as telas de uma aplicação web, habilitando o carregamento de conteúdos dinâmicos e otimizando a renderização, além de prover uma interface para exibir mensagem de erro, mensagens de Loading, uso de outlets, sub-rotas, rotas dinâmicas, etc.
- **Rendering:** Renderização de componentes tanto no servidor quanto no cliente, permitindo inclusive que eles interajam entre si.
- **Data fetching:** Possibilita realizar acesso a dados em Server Components possuindo inclusive controle de cache. Também possui recursos de Streaming (não abordados neste livro).
- **Styling:** Suporte a Css, Scss, CSS-in-JS Tailwind etc.
- **Optimizations:** Otimizações de imagens, fonts e scripts para melhorar a experiência do usuário da aplicação.
- **TypeScript:** Suporte completo ao Typescript

### 2.1. Criando o Projeto

Para criar um projeto para o Next.js, é necessário estar com o Node.js instalado, na versão 18.17 ou superior. Ao instalar o Node, temos disponível também o NPM que é o gerenciador de pacotes do Node e o NPX, que é o executor de pacotes do NPM.

O terminal que iremos utilizar neste livro é o Git Bash, que é instalado juntamente com o Git. Abre o terminal e execute o seguinte comando:

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

A terminal window titled 'npm install' showing the execution of 'npx create-next-app@latest stock-next-app'. The user 'dps@pc MINGW64 ~' runs the command. The terminal displays a series of prompts with green checkmarks indicating 'Yes' answers for TypeScript, ESLint, Tailwind CSS, the 'src/' directory, App Router, and customizing the default import alias. It then shows the installation of dependencies (react, react-dom, next) and devDependencies (typescript, @types/node, @types/react, @types/react-dom).

```
dps@pc MINGW64 ~  
$ npx create-next-app@latest stock-next-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  
Creating a new Next.js app in C:\Users\dps\stock-next-app.  
  
Using npm.  
  
Initializing project with template: app-tw  
  
Installing dependencies:  
- react  
- react-dom  
- next  
  
Installing devDependencies:  
- typescript  
- @types/node  
- @types/react  
- @types/react-dom
```

Após o comando, o terminal irá mostrar as instruções para criar o projeto. Selecione as opções padrão e aguarde o projeto ser criado. Após a criação do projeto, acesse a pasta `stock-next-app` e execute o seguinte comando `npm run dev` para iniciar o servidor:

A terminal window with a dark background and light-colored text. The window title is "npm config get registry". The prompt is "dps@pc MINGW64 ~/stock-next-app (master)". The user enters "\$ npm run dev". The output shows the command "stock-next-app@0.1.0 dev" and "next dev". It then displays "▲ Next.js 14.1.3" and "- Local: http://localhost:3000". Finally, it shows a green checkmark and "Ready in 3.7s" followed by a small pink progress bar.

```
npm config get registry

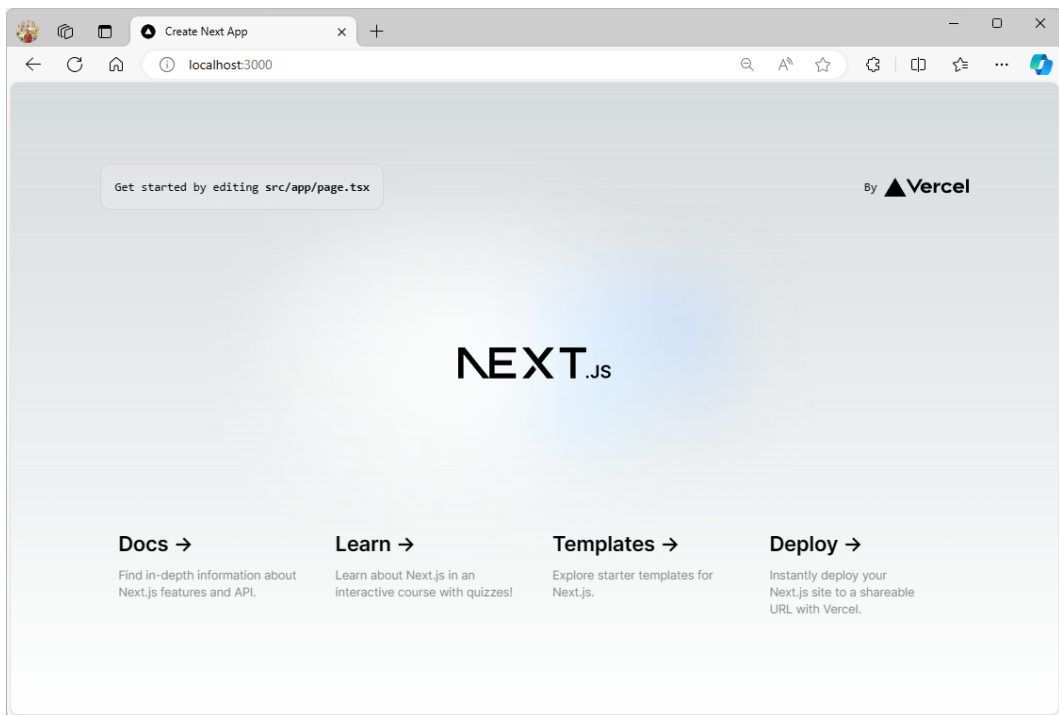
dps@pc MINGW64 ~/stock-next-app (master)
$ npm run dev

> stock-next-app@0.1.0 dev
> next dev

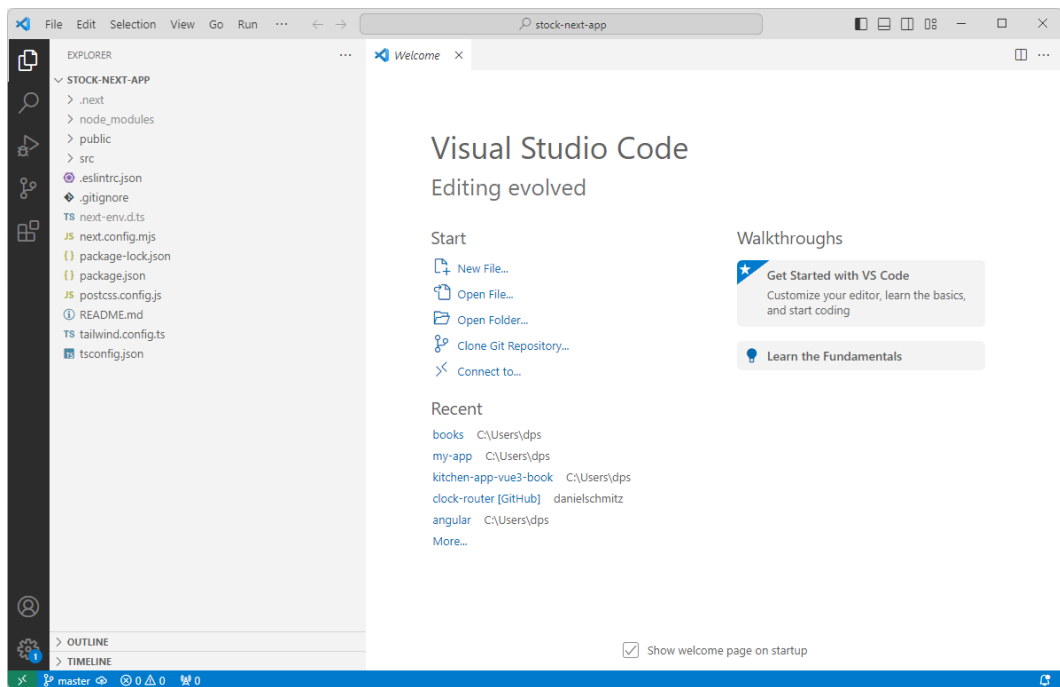
▲ Next.js 14.1.3
- Local:      http://localhost:3000

✓ Ready in 3.7s
```

Acesse <http://localhost:3000/> para ver o projeto criado:



Abra o Visual Studio Code (que chamaremos ao longo deste livro simplesmente de `vscode`) e abra o projeto `stock-next-app`:



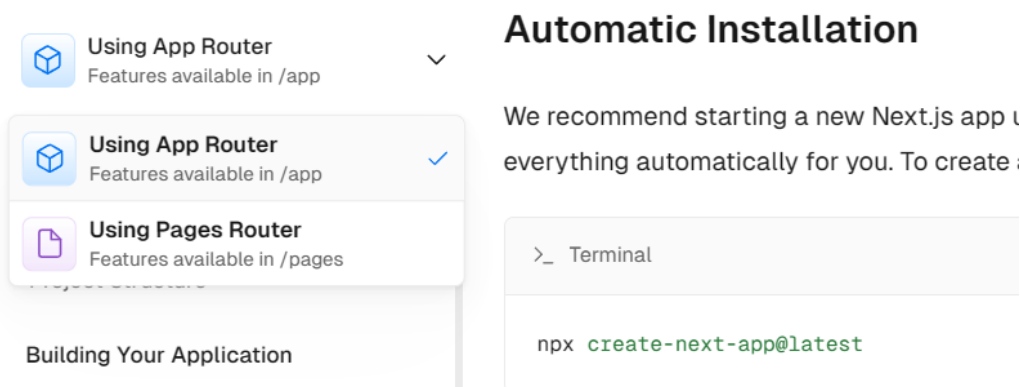
A esquerda temos os arquivos iniciais da aplicação, criados pelo comando `create-next-app@latest`. A seguir vamos compreender o que significa cada arquivo:

- └─ **stock-next-app**
  - └─ `.eslintrc.json` *# Especifica as Regras de ESLint*
  - └─ `.gitignore` *# Ignorar Arquivos e Pastas*
  - └─ `next-env.d.ts` *# Definir Variáveis de Ambiente*
  - └─ `next.config.mjs` *# Configurações do Next.js*
  - └─ `package-lock.json` *# Arquivo de Suporte Ao NPM*
  - └─ `package.json` *# Configurações do NPM e do Projeto*
  - └─ `postcss.config.js` *# Configurações do PostCSS*
  - └─ **public**
    - └─ `next.svg`
    - └─ `vercel.svg`
  - └─ `README.md`
  - └─ **src** *# Pasta Onde a Aplicação de Fato Reside*
    - └─ **app** *# Pasta de Rotas da Aplicação*
      - └─ `favicon.ico` *# Favicon da Aplicação*
      - └─ `globals.css` *# Estilos Globais*
      - └─ `layout.tsx` *# Componente de Layout da Aplicação*
      - └─ `page.tsx` *# Componente de Rota da Aplicação*

```
└─ tailwind.config.ts      # Configurações do Tailwind
└─ tsconfig.json           # Configurações do TypeScript
```

Inicialmente, o mais importante é compreender o diretório `src/app`. Esta pasta contém todas as rotas da aplicação e precisamos compreender como o Next.js trabalha com elas.

A partir da versão 14 do Next.js, o diretório `src/app` é usado para a criação das rotas. Na versão anterior, o diretório `src/pages` que era usado. É preciso ter atenção principalmente se você estiver pesquisando no Google sobre o Next.js. Você precisa saber que existem essas duas formas, que chamamos de “App Routes” e “Page Routes”. Na própria documentação do Next.js existe essa diferenciação, conforme a imagem a seguir:



Durante todo este livro, estaremos utilizando App Routes. Nunca estaremos utilizando Page Routes.

## 2.2. App Routes

### Primeira regra Do App Routes

A primeira regra do App Routes é que toda rota possui um arquivo `page.tsx` dentro de um diretório que compõe a rota. Por exemplo:

- `src/app/page.tsx` é a rota de `/`.
- `src/app/dashboard/page.tsx` é a rota de `/dashboard`.
- `src/app/categories/page.tsx` é a rota de `/categories`.
- `src/app/products/page.tsx` é a rota de `/products`.

- `src/app/user/account/page.tsx` é a rota de `/user/account`

## Segunda Regra Do App Routes

A segunda regra do App Routes é que toda rota possui um arquivo `layout.tsx` dentro de um diretório que compõe a rota. Por exemplo:

- `src/app/page.tsx` é a rota de `/`.
- `src/app/dashboard/layout.tsx` é a rota de `/dashboard`.
- `src/app/categories/layout.tsx` é a rota de `/categories`.
- `src/app/products/layout.tsx` é a rota de `/products`.
- `src/app/user/account/layout.tsx` é a rota de `/user/account`

A diferença entre `layout.tsx` e `page.tsx` é que `layout.tsx` é o componente que inclui o `page.tsx` como se fosse um outlet, e além disso, o `layout.tsx` pode ser reaproveitado para todas as sub rotas. Por exemplo, ao termos este `layout.tsx` no diretório `src/app/categories`:

```
import {Card, CardContent, CardHeader, CardTitle} from '@components/ui/card'
import {PropsWithChildren} from 'react'

export default function layout({children}: PropsWithChildren) {
  return (
    <Card>
      <CardHeader>
        <CardTitle>Categories</CardTitle>
      </CardHeader>
      <CardContent>{children}</CardContent>
    </Card>
  )
}
```

O arquivo `page.tsx` será renderizado no `{children}` do `layout.tsx`. Se por exemplo tivermos a seguinte estrutura:

```
└─ 📁categories
  └─ layout.tsx
  └─ 📁new
    └─ page.tsx
    └─ page.tsx
```

Onde `categories/new/page.tsx` não possui um `layout.tsx`, o layout utilizado será o imediatamente acima dele.

Em breve iremos instalar os componentes `<Card>`, `<CardHeader>`, `<CardTitle>` e `<CardContent>` so shadcn. Não se preocupe com eles agora.

## 2.3. Limpando a Aplicação

Agora vamos limpar a tela inicial da aplicação. Acesse o arquivo `src/app/pages.tsx` e remova todo o conteúdo, deixando o arquivo da seguinte forma:

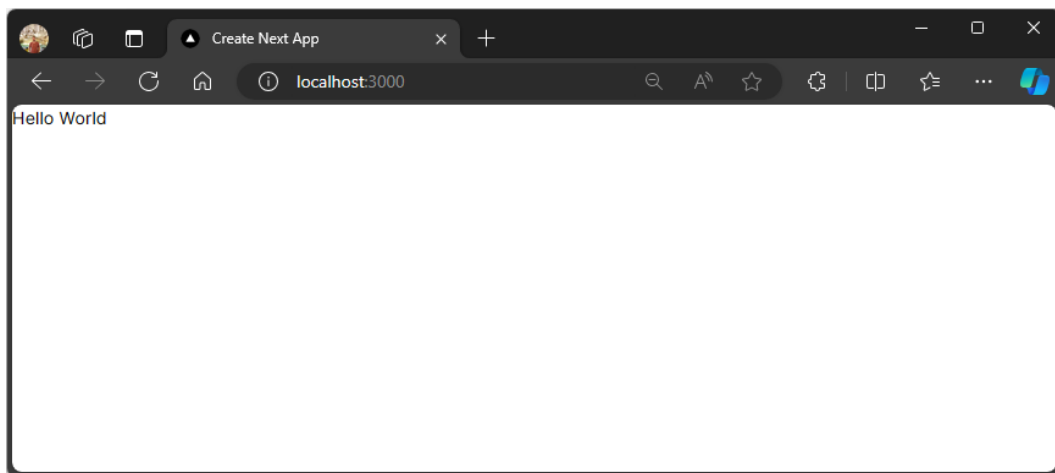
```
export default function Home() {
  return <main>Hello World</main>
}
```

Teremos então uma tela inicial onde o texto `Hello World` será exibido em branco, com um fundo preto. Para remover os estilos, acesse o arquivo `src/globals.tsx` e deixe somente o conteúdo relativo ao TailWind:

```
/* src/globals.tsx */
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Desta forma, temos o texto “Hello World” em preto com fundo branco:



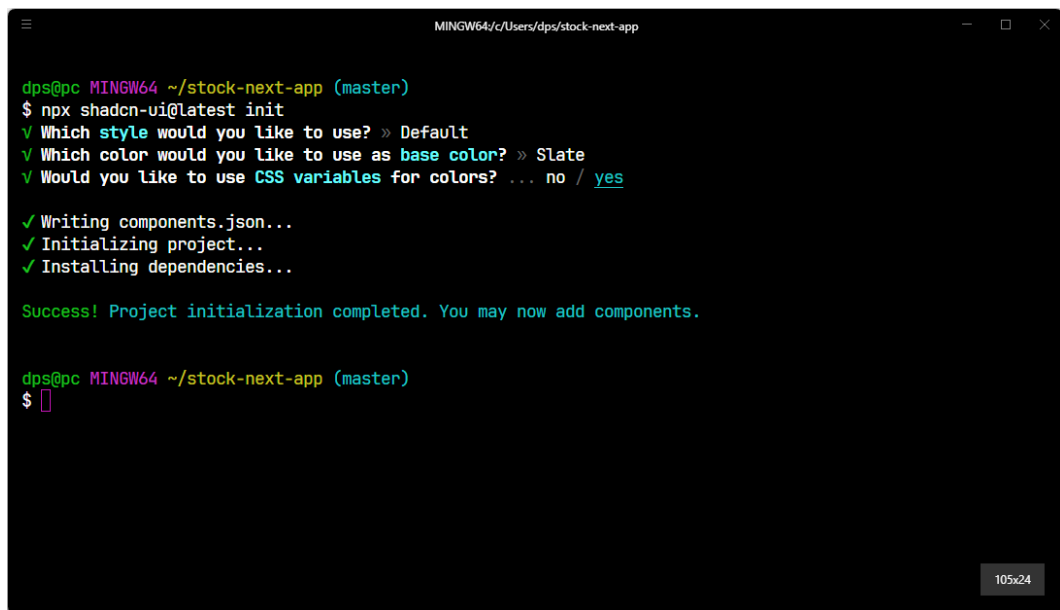


## 2.4. Instalando o shadcn

Agora vamos instalar o shadcn. Uma característica muito interessante do shadcn é que ele não é um framework completo para ser instalado. Ele é apenas um conjunto de componentes React + Tailwind que pode ser instalado componente a componente!

Para iniciar a instalação, acesse o terminal e execute o seguinte comando:

```
npx shadcn-ui@latest init
```

A terminal window titled 'MINGW64/c/Users/dps/stock-next-app' shows the command 'npx shadcn-ui@latest init' being executed. The terminal displays a series of prompts and confirmations: 'Which style would you like to use? » Default', 'Which color would you like to use as base color? » Slate', and 'Would you like to use CSS variables for colors? ... no / yes'. It then shows 'Writing components.json...', 'Initializing project...', and 'Installing dependencies...'. A success message follows: 'Success! Project initialization completed. You may now add components.' The prompt returns to '\$'.

Deixe as opções padrão do shadcn e perceba que um arquivo foi criado na raiz do seu projeto, o `components.json` contendo algumas configurações:

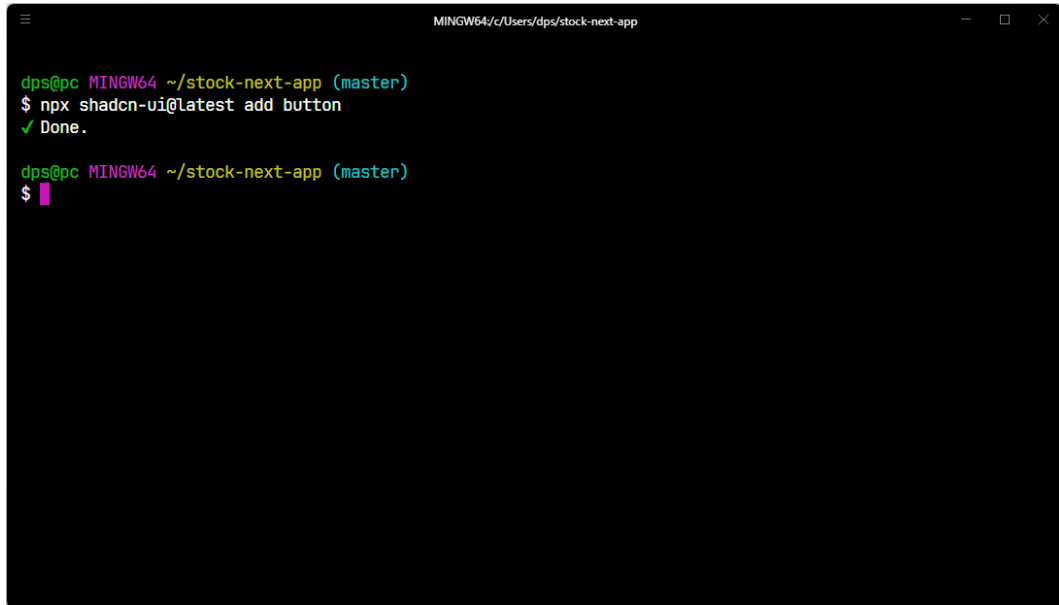
```
{
  "$schema": "https://ui.shadcn.com/schema.json",
  "style": "default",
  "rsc": true,
  "tsx": true,
  "tailwind": {
    "config": "tailwind.config.ts",
    "css": "src/app/globals.css",
    "baseColor": "slate",
    "cssVariables": true,
    "prefix": ""
  },
  "aliases": {
    "components": "@components",
    "utils": "@lib/Utils"
  }
}
```

Também foram criadas as pastas `src/components` e `src/lib` no projeto, no qual o shadcn fará modificações a medida que formos instalando novos componentes.

### 2.4.1. Adicionando o Componente `Button`

Agora vamos adicionar o componente `Button`. No terminal, digite:

```
npx shadcn-ui@latest add button
```

A screenshot of a Windows terminal window with a dark background. The title bar at the top reads 'MINGW64/c/Users/dps/stock-next-app'. The terminal shows a prompt 'dps@pc MINGW64 ~/stock-next-app (master)' followed by the command '\$ npx shadcn-ui@latest add button'. The output is '✓ Done.' followed by another prompt 'dps@pc MINGW64 ~/stock-next-app (master)' and a new line starting with '\$' and a cursor.

Ao adicionarmos o `Button`, perceba que o arquivo `src/components/ui/button.tsx` foi criado no projeto:

```
src/app/components/ui/button.tsx

// ... imports ...

const buttonVariants = cva("...tailwind styles...",
  {
    variants: {
      // ...code...
    }
  }
)
export interface ButtonProps
  extends React.ButtonHTMLAttributes<HTMLButtonElement>,
    VariantProps<typeof buttonVariants> {
  asChild?: boolean
}
const Button = React.forwardRef<HTMLButtonElement, ButtonProps>(
  ({ className, variant, size, asChild = false, ...props }, ref) => {
    const Comp = asChild ? Slot : "button"
    return (
      <Comp
        className={cn(buttonVariants({ variant, size, className }))}
        ref={ref}
        {...props}
      />
    )
  }
)

Button.displayName = "Button"

export { Button, buttonVariants }
```

Uma das particularidades do shadcn é que todos os componentes são copiados para `src/components/ui` e você tem o poder de alterar qualquer comportamento ou estilo. Tendo em mente que grandes poderes trazem grandes responsabilidades, altere estes arquivos somente se realmente for necessário. Durante a criação deste livro e da instalação de todos os componentes do shadcn, não haverá a necessidade de alterar nada nesta pasta.

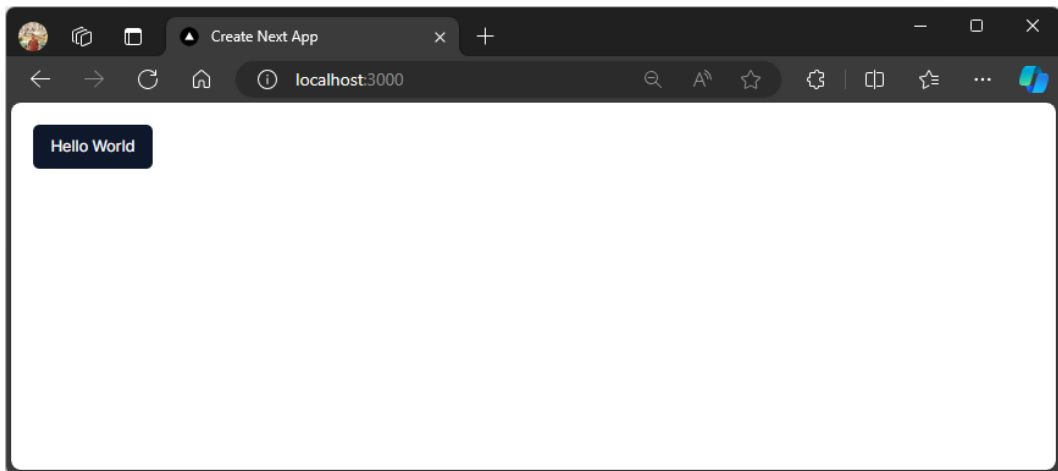
Com o componente instalado, podemos usá-lo para fazer um teste. Vamos alterar o componente `src/app/page.tsx` para mostrar o componente `Button`:

```
// src/app/page.tsx
import {Button} from '@components/ui/button'

export default function Home() {
  return (
    <main className="m-5">
      <Button>Hello World</Button>
    </main>
  )
}
```

Neste código usamos `className="m-5"` para adicionar uma margem na página. Como estamos utilizando o TailWind, podemos usar as classes dele livremente. Aliás, todos os componentes do shaden utilizam Tailwind.

A página fica semelhante a figura a seguir:



## 2.5. Adicionando um Header

O Header pode ser criado com Tailwind e o componente Button recentemente instalado. Primeiro, crie o arquivo `src/app/components/Header.tsx`:

```
import Link from 'next/link'
import React from 'react'
import {Button} from './ui/button'

export default function Header(props: {title: string}) {
  return (
    <div
      className="bg-black text-slate-50
        font-semibold text-lg p-5 shadow-lg flex
        flex-row justify-between items-center"
    >
      <h1>{props.title}</h1>
      <div className="flex flex-row gap-2">
        <Button asChild variant="secondary">
          <Link href="/">Home</Link>
        </Button>
        <Button asChild variant="secondary">
          <Link href="/categories">Categories</Link>
        </Button>
        <Button asChild variant="secondary">
          <Link href="/products">Products</Link>
        </Button>
      </div>
    </div>
  )
}
```

Neste código, criamos uma div que o `Flex Layout` para posicionar os elementos da div na forma `justify-between`. Isso garante que o primeiro elemento estará à esquerda, neste caso o `h1`, e o segundo elemento à esquerda, neste caso outra div, com 3 botões.

Os botões fazem parte do menu da aplicação, e por enquanto podemos deixar o menu da forma mais simples possível, apenas colocando 3 botões um do lado do outro e usando `Flex Layout` para posicioná-los um ao lado do outro com um pequeno espaço (`gap-2`).

Também que criamos um parâmetro `title` no componente `Header`. A forma como criamos o parâmetro `props: { title: string }` usa os conceitos de [Array Destructuring](#) para extrair a variável `title`.

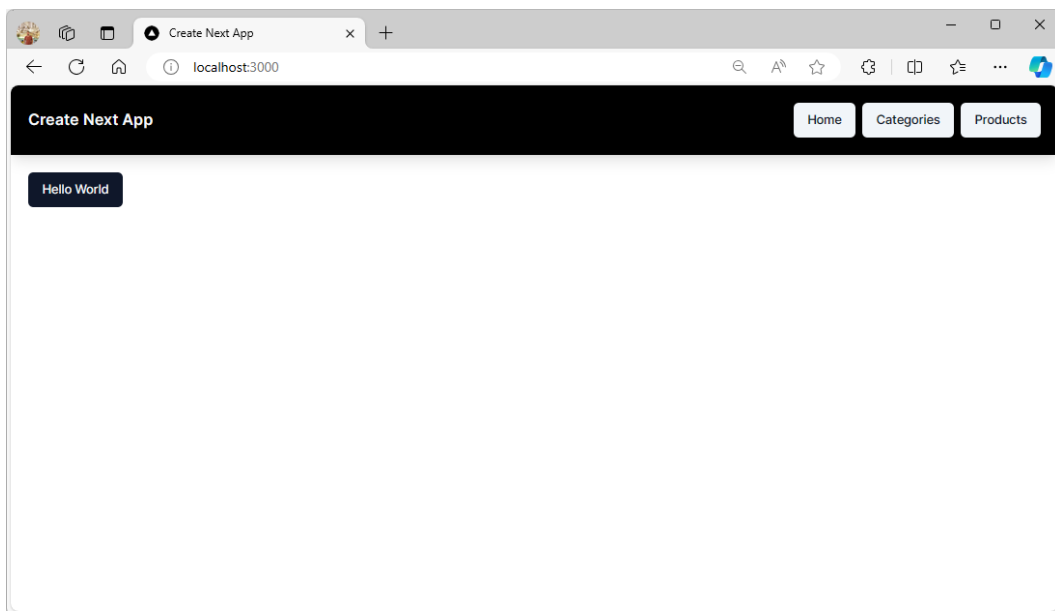
Criamos o `Button` em conjunto com o componente `<Link>` do React, desta forma, ao clicar no botão, será redirecionado para a rota correta. O `asChild` significa que o componente `Button` será renderizado como se fosse um elemento a (o componente filho dele).

Após a criação do `Header`, podemos adicioná-lo ao arquivo `src/app/layout.tsx`, que é o layout padrão para todas as páginas do sistema. Mesmo se houverem rotas e sub-rotas, o `layout.tsx` será renderizado, então o `Header` estará presente sempre.

src/app/layout.jsx

```
1  import type { Metadata } from "next";
2  import { Inter } from "next/font/google";
3  import "./globals.css";
4  import Header from "@components/Header";
5
6
7  const inter = Inter({ subsets: ["latin"] });
8
9  export const metadata: Metadata = {
10    title: "Create Next App",
11    description: "Generated by create next app",
12  };
13
14  export default function RootLayout({
15    children,
16  }: Readonly<{
17    children: React.ReactNode;
18  }>) {
19    return (
20      <html lang="en">
21        <body className={inter.className}>
22          <Header title={metadata.title as string} />
23          {children}</body>
24        </html>
25      );
26  }
```

Após incluir o `<Header>`, temos a seguinte tela:



Veja que o título da página está `Create Next App`, devido a variável `metadata`. Você pode alterar para outro título, como por exemplo `Stock Control System`.

## 2.6. Banco de Dados e Prisma ORM

Neste livro, iremos utilizar o banco de dados `SQLite` e o ORM `Prisma`. É a forma mais fácil de começar a trabalhar com dados em uma aplicação `Next`. Se num futuro mudarmos para outro banco de dados, como o `MySQL` ou o `PostgreSQL`, precisaremos apenas mudar uma configuração e o `Prisma` estará pronto para ser usado com estes respectivos banco de dados.

Para instalar o `Prisma`, execute o seguinte comando:

```
npm install prisma
```

Após a instalação, vamos configurar o `prisma` através deste comando:

```
npx prisma init --datasource-provider sqlite
```



```
MINGW64/c/Users/dps/stock-next-app

dps@pc MINGW64 ~/stock-next-app (master)
$ npx prisma init --datasource-provider sqlite

✓ Your Prisma schema was created at prisma/schema.prisma
  You can now open it in your favorite editor.

warn You already have a .gitignore file. Don't forget to add `.env` in it to not commit any private information.

Next steps:
1. Set the DATABASE_URL in the .env file to point to your existing database. If your database has no tables yet, read https://pris.ly/d/getting-started
2. Run prisma db pull to turn your database schema into a Prisma schema.
3. Run prisma generate to generate the Prisma Client. You can then start querying your database.

More information in our documentation:
https://pris.ly/d/getting-started

dps@pc MINGW64 ~/stock-next-app (master)
$
```

Após executar o comando `prisma init`, veja que o arquivo `.env` da aplicação possui a variável `DATABASE_URL`, onde iremos configurar o caminho do banco de dados. No `sqlite`, usamos o padrão `file:./dev.db` para criar um arquivo local. Em outros banco de dados, como o `MySQL`, usaríamos uma string de conexão. Neste livro, estaremos utilizando o `sqlite` e os dados serão armazenados no arquivo `dev.db`.

Veja também que o `prisma init` criou o diretório `prisma`, e o arquivo `prisma/schema.prisma`, inicialmente com o seguinte conteúdo:

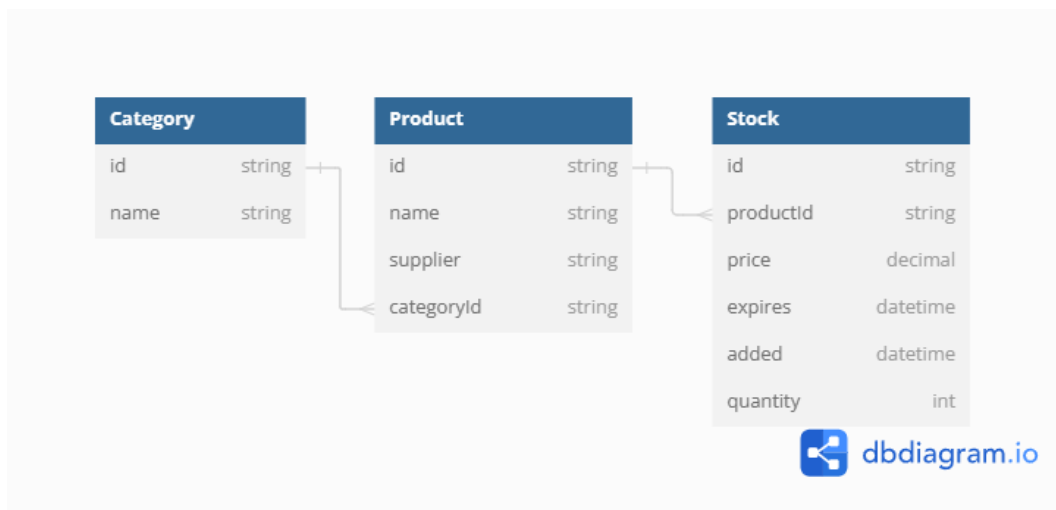
```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}
```

## 2.6.1. Criando o Schema e Tabelas

O arquivo de “schema” do Prisma é o arquivo com todas as configurações de acesso ao banco de dados. Como o Prisma é um ORM, iremos mapear as tabelas no dialeto que o prisma compreende. Para o nosso livro, iremos criar as seguintes tabelas:



Edite o arquivo `src/schema.prisma` e adicione a tabela `Categories`:

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema
```

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model Category {
  id      String @id @default(cuid())
  name    String
  products Product[]
}
```

Definimos uma tabela no Prisma através da palavra “model”, seguido do nome da tabela. O campo `id` será a chave primária da tabela, e ele será gerado automaticamente como um `cuid`, um hash de 32 caracteres que torna todos os registros únicos. O campo `name` é o nome da categoria e o campo `products` é uma tabela de relacionamento entre as categorias e os produtos. Ainda não criamos o `model Product`, então quando usamos `Products[]` será exibido um pequeno erro (se não aparecer, verifique se instalou a extensão Prisma para o vscode).

Vamos agora criar o `model Product`:

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model Category {
  id      String  @id @default(cuid())
  name    String
  products Product[]
}

model Product {
  id      String  @id @default(cuid())
  name    String
  supplier String?
  categoryId String
  category Category @relation(fields: [categoryId], references: [id])
  stock   Stock[]
}
```

A criação do `Model Product` possui o campo `Id`, novamente um `cuid()` para a geração de um hash único, além do campo `name` e o campo `supplier`. O tipo `String?` significa que o campo `supplier` pode ser opcional. Como o campo `categoryId` é uma chave estrangeira, o campo `category` é o relacionamento entre as categorias e os produtos. O `@relation fields` e `references`, representam a chave estrangeira e a chave primária da tabela. O campo `stock` é o relacionamento entre os produtos e os estoques, que será criado a seguir:

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model Category {
  id      String  @id @default(cuid())
  name    String
  products Product[]
}

model Product {
  id      String  @id @default(cuid())
  name    String
  supplier String?
  categoryId String
  category Category @relation(fields: [categoryId], references: [id])
  stock   Stock[]
}

model Stock {
  id      String  @id @default(cuid())
  productId String
  price    Decimal
  expires  DateTime
  added    DateTime @default(now())
  quantity Int
  product  Product @relation(fields: [productId], references: [id])
}
```

Finalmente criamos o `model Stock`, definido o relacionamento entre os produtos e os estoques através do campo `productId` e o campo `product`. No campo `added`, estamos usando o parâmetro `@default(now())` para definir o campo `added` como o horário atual.

## 2.6.2. Criando o Banco de Dados Através do Prisma

Com o modelo criado, podemos usar o Prisma para gerar o banco de dados e as tabelas. Ao invés de digitar o comando no terminal, vamos adicionar dois comandos ao arquivo `package.json`, na seção `scripts`:

```
{
  "name": "stock-next-app",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint",
    "db:create": "prisma db push",
    "db:studio": "prisma studio"
  },
  "dependencies": {
    ...
  },
  "devDependencies": {
    ...
  }
}
```

O comando `db:create` irá executar o comando `prisma db push` responsável em criar o banco de dados. O comando `db:studio` executará o `prisma studio`, que nos permite visualizar o esquema do banco de dados em uma página web.

Para executar `db:create`, abra o terminal e digite:

```
npm run db:create
```

```
MINGW64/c/Users/dps/stock-next-app

dps@pc MINGW64 ~/stock-next-app (master)
$ npm run db:create

> stock-next-app@0.1.0 db:create
> prisma db push

Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma
Datasource "db": SQLite database "dev.db" at "file:./dev.db"

SQLite database dev.db created at file:./dev.db

Your database is now in sync with your Prisma schema. Done in 24ms

✓ Generated Prisma Client (v5.11.0) to .\node_modules\@prisma\client in 74ms

dps@pc MINGW64 ~/stock-next-app (master)
$
```

Veja duas mensagens importantes quando executamos este comando. Primeiro, o arquivo `./dev.db` foi criado, no diretório `./prisma`. Segundo, a mensagem ✓ Generated Prisma Client (v5.11.0) to `.\node_modules\@prisma\client` indica que o Prisma utilizou o modelo que criamos no arquivo `./prisma/schema.prisma` para criar um conjunto de classes e métodos para o nosso ORM. Ou seja, no código poderemos usar, por exemplo, `prisma.category.create` para criar um registro no banco de dados ou `prisma.category.findMany` para buscar todos os registros. Todos estes métodos já estão prontos para uso e podem ser usados facilmente na aplicação.

Caso esteja utilizando Git no projeto, adicione o arquivo `dev.db` no `.gitignore` para que ele não seja comitado para o repositório. Quando estamos criando sistemas reais e com um certo nível de segurança, o arquivo `.env` também estar no `.gitignore`, pois não devemos enviar strings de conexão ao repositório.

Com o banco de dados e tabelas prontos, podemos criar a nossa primeira tela, a tela de categorias.

## 3. Tela de Categorias

A tela de categorias é uma tela simples com apenas um campo, `name`. Através dela iremos aprender a criar rotas, tabelas e formulários de uma forma mais simples, aprendendo os conceitos básicos do Next.js e preparando para o desenvolvimento de algo mais sofisticado na tela de cadastro de Produtos.

### 3.1. Criando a Tela Inicial de Categorias

Nas tela principal, temos um botão `Categories` que aponta para `/categories`, então nossa primeira tarefa será criar os arquivos `layout.tsx` e `page.tsx` em `src/app/categories`:

```
//src\app\categories\layout.tsx
import {PropsWithChildren} from 'react'

export default function layout({children}: PropsWithChildren) {
  return (
    <div>
      <h2>Categories</h2>
      <div>{children}</div>
    </div>
  )
}
```

Todos os componentes React e Next.js terão a assinatura `export default function name`, onde `name` é o nome do componente.

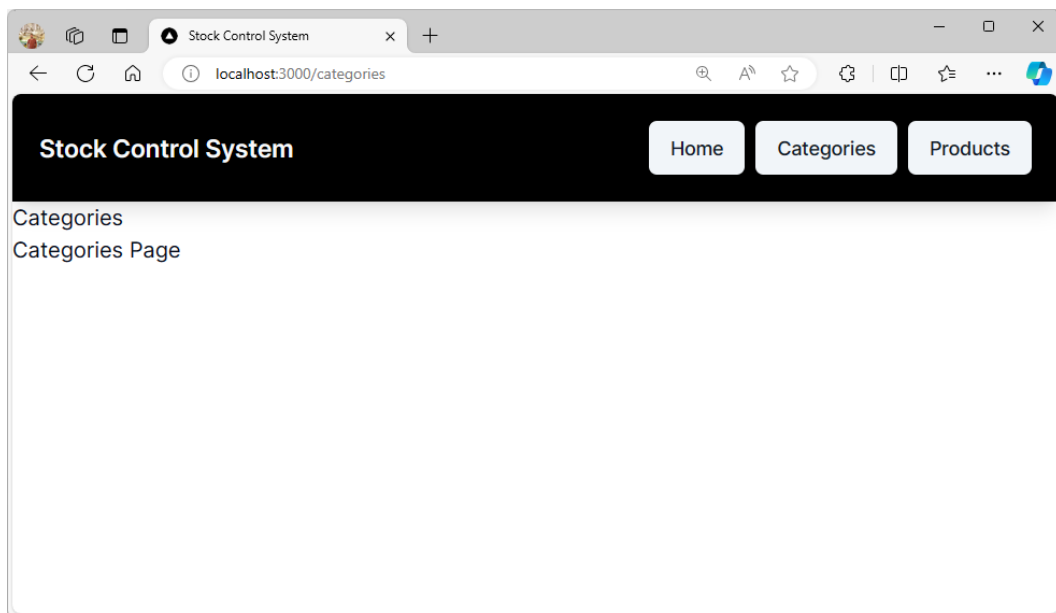
Como o Next.js é um framework que tem como base o React, então tudo que temos no React podemos usar no Next.js. Um dos conceitos fundamentais do React é o `children` que é usado para passar um conteúdo filho para o pai. Por exemplo, `layout.tsx` é o componente

pai e `page.tsx` é o componente filho, então o conteúdo de `page.tsx` será inserido dentro de `<div>{children}</div>` no `layout.tsx`.

Após criar `layout.tsx`, vamos criar `page.tsx`:

```
// src\app\categories\page.tsx
export default function page() {
  return <div>Categories Page</div>
}
```

Por enquanto, a página de categorias tem apenas o conteúdo `<div>Categories Page</div>`. Quando acessamos a página de categorias, temos:



Até o momento, temos a seguinte estrutura:

```
├── src
│   ├── app
│   │   └── categories
│   │       ├── layout.tsx
│   │       └── page.tsx
```

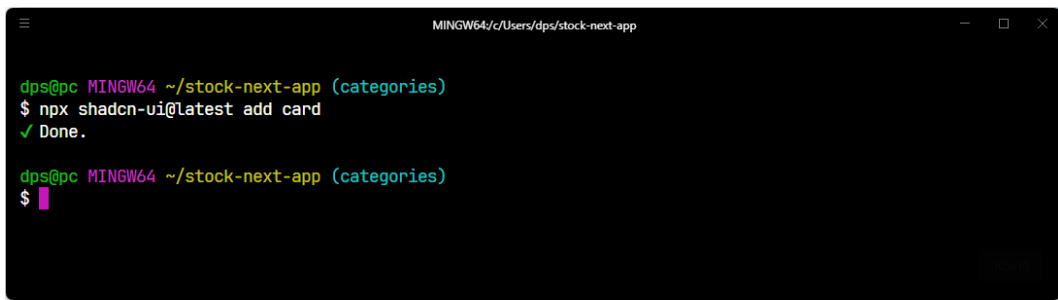


## 3.2. Adicionando o Componente Card

O componente `Card` do `shadcn` é um importante componente que iremos utilizar ao longo da aplicação.

Todo componente do `shadcn` tem um guia para instalação em sua respectiva [página](#). Para o `Card`, devemos executar o seguinte comando:

```
npx shadcn-ui@latest add card
```

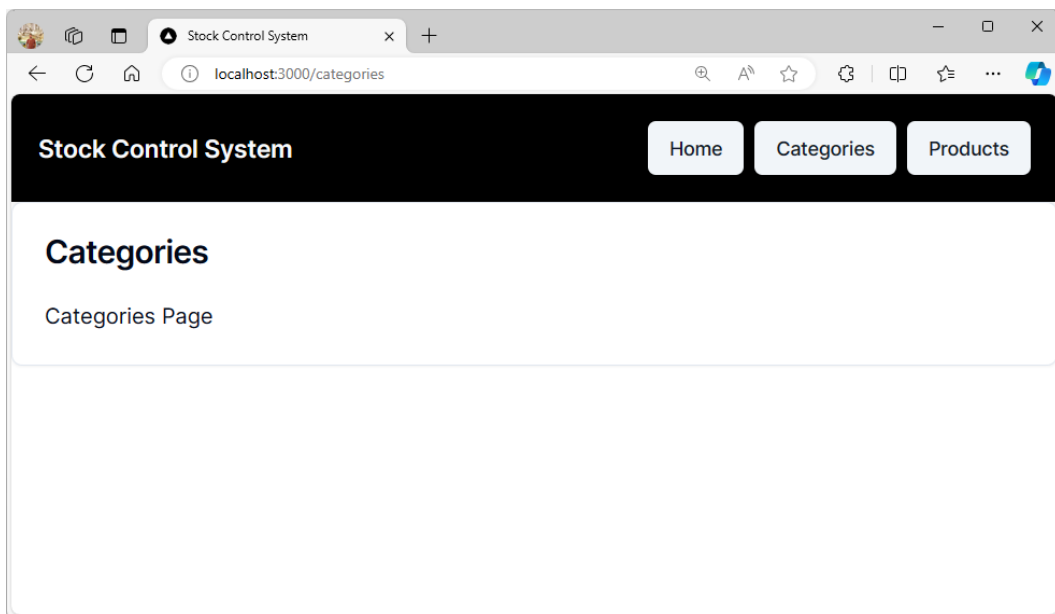
A terminal window with a dark background and light green text. The title bar shows the path 'MINGW64/c/Users/dps/stock-next-app'. The prompt is 'dps@pc MINGW64 ~/stock-next-app (categories)'. The user enters '\$ npx shadcn-ui@latest add card'. The terminal responds with '✓ Done.' on the next line. The prompt is shown again, and the user enters '\$' followed by a cursor.

Após a instalação, temos na página do componente o Usage, que é a forma como devemos usar o componente. Vamos adicionar o `Card` no `layout.tsx`:

```
import {Card, CardContent, CardHeader, CardTitle} from '@components/ui/card'
import {PropsWithChildren} from 'react'

export default function layout({children}: PropsWithChildren) {
  return (
    <Card>
      <CardHeader>
        <CardTitle>Categories</CardTitle>
      </CardHeader>
      <CardContent>{children}</CardContent>
    </Card>
  )
}
```

Realizando as devidas importações e utilizando o componente `<Card>`, criamos o título da página com o `<CardTitle>` e o `<CardHeader>`. O `{children}` foi adicionado ao `<CardContent>`. A tela de categorias fica semelhante a figura a seguir:



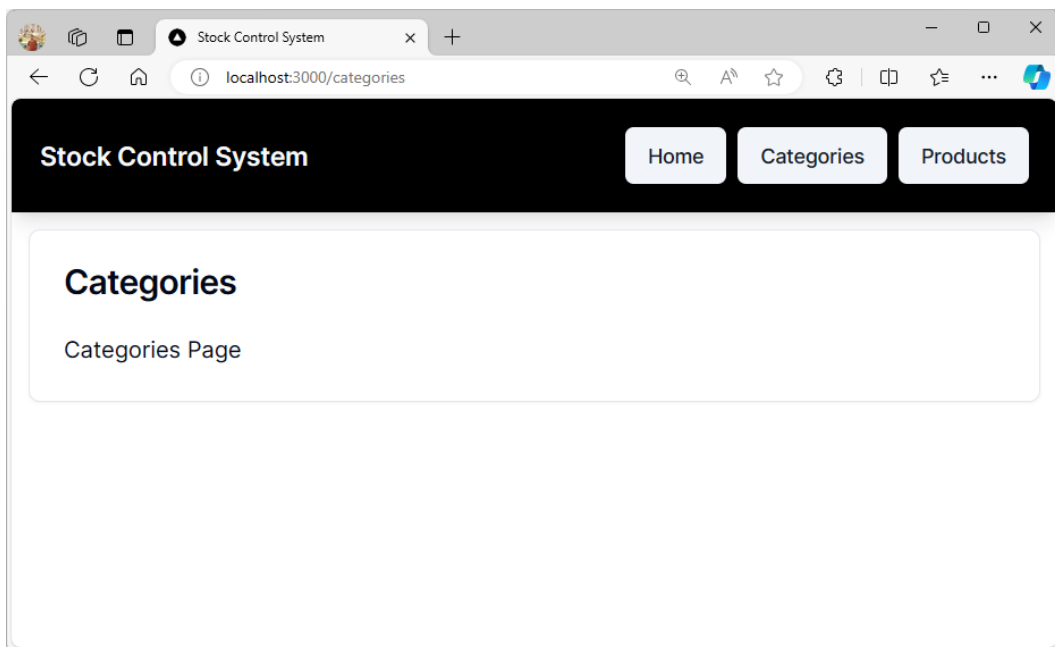
Apesar do Card estar com a margem muito pequena em relação a página, já é possível verificar que o conteúdo de `page.tsx` foi adicionando dentro do Card.

Para ajustar a margem do Card, edite o arquivo `src/app/layout.tsx` adicionando uma `div` com margem no `{children}`.

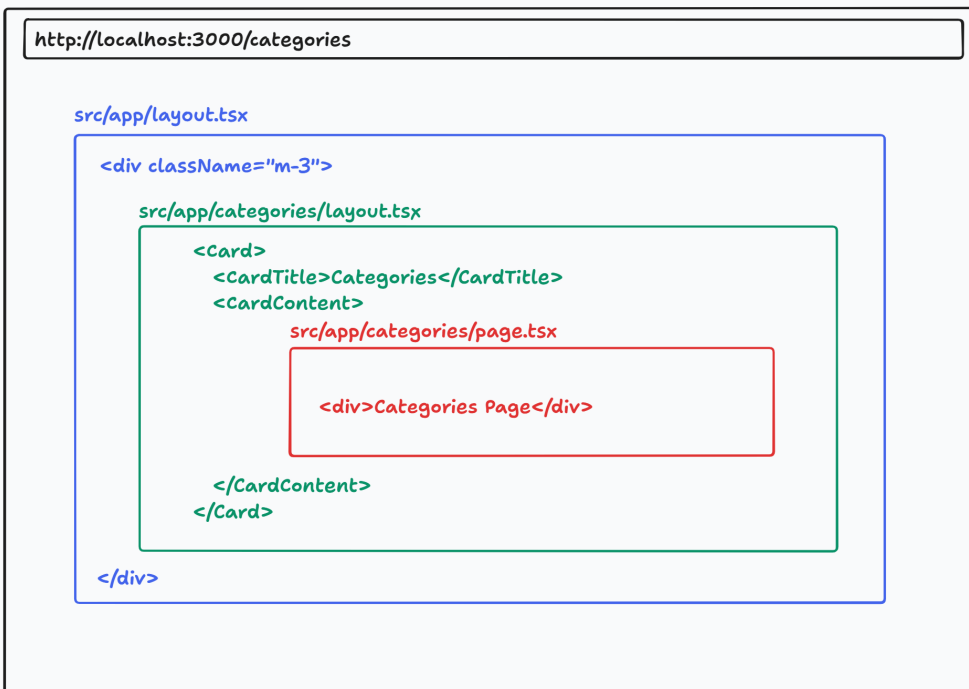
`src/app/layout.tsx`

```
1 // imports
2 // initial code
3
4 export default function RootLayout(...) {
5   return (
6     <html lang="en">
7       <body className={inter.className}>
8         <Header title={metadata.title as string} />
9         <div className="m-3">
10           {children}
11         </div>
12       </body>
13     </html>
14   )
15 }
16
```

Após alterar o `layout.tsx` de `src/app`, a página de categorias ficou assim:



Como podemos ver, o componente `/src/app/categories/layout.tsx` é renderizado dentro do componente `/src/app/layout.tsx`, obedecendo a hierarquia entre layouts e pages, conforme a imagem a seguir:



Podemos observar que o componente `src/app/layout.tsx` será renderizado em qualquer rota da aplicação, sendo global. Os componentes `layout` filhos a ele são renderizados de acordo com a rota.

O componente `page.tsx` sempre utiliza um `layout` como pai, sendo que este `layout` pode estar no mesmo diretório dele, ou em um diretório acima.

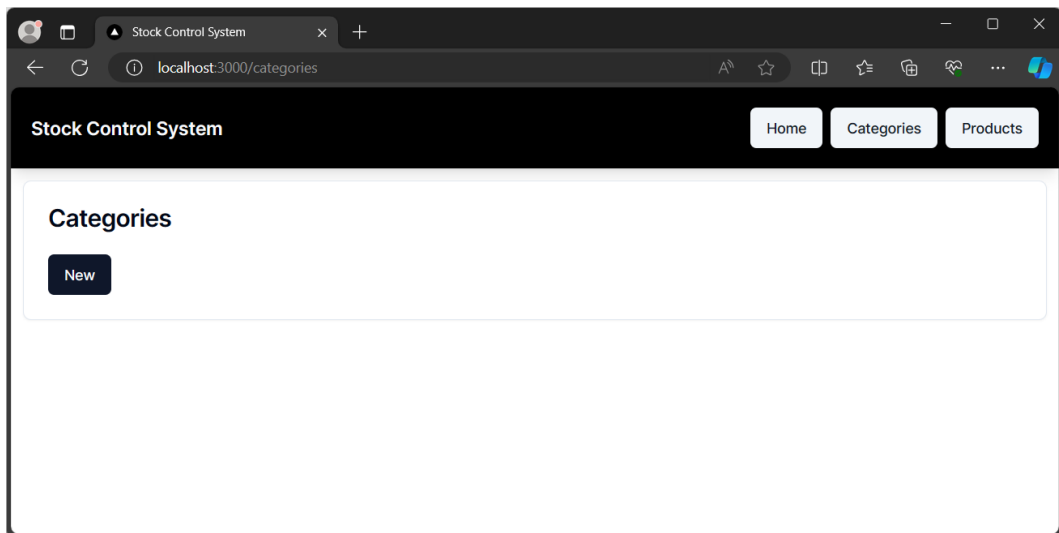
## 3.3. Adicionando uma Categoria

### 3.3.1. Criando um Botão para Adicionar Categoria

Para adicionar uma categoria, precisamos adicionar um botão para redirecionar para `/categories/new`. Em `/categories/page.tsx`, adicione um `Button`, seguido do `Link`:

```
import {Button} from '@components/ui/button'
import Link from 'next/link'

export default function page() {
  return (
    <div>
      <Button asChild>
        <Link href="/categories/new">New</Link>
      </Button>
    </div>
  )
}
```



O atributo `asChild` do `Button` diz que o botão deve ser renderizado com o elemento filho, nessa caso o `Link`, que por sua vez renderiza um elemento `<a></a>` do `html`.

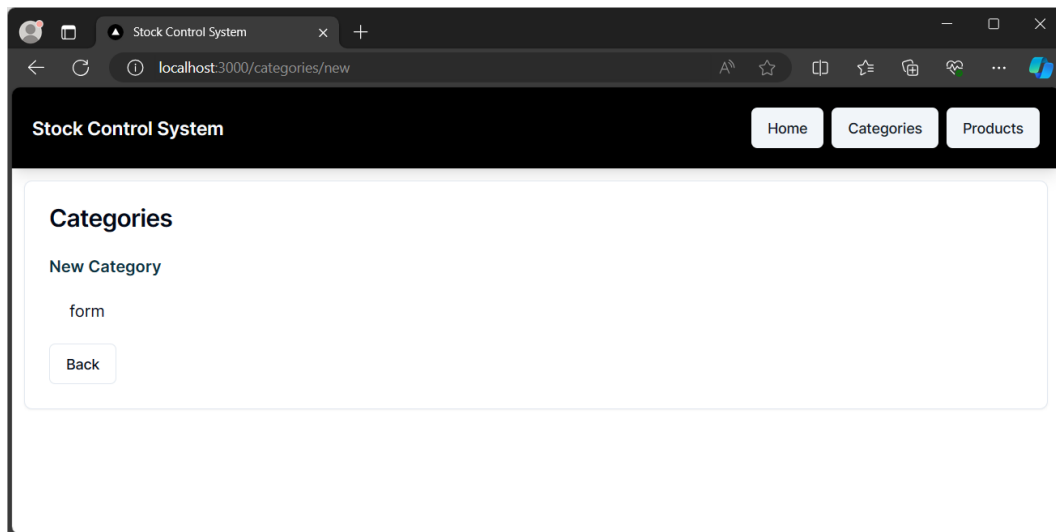
### 3.3.2. Criando a Página para Adicionar Categoria

Como a rota para definir uma categoria foi `/categories/new` precisamos inicialmente criar o arquivo `/src/app/categories/new/page.tsx`, com o seguinte conteúdo:

```
import {Button} from '@components/ui/button'
import Link from 'next/link'

export default function page() {
  return (
    <div>
      <h2 className="text-cyan-950 text-sm font-semibold">New Category</h2>
      <div className="flex flex-row w-full m-5">form</div>
      <Button asChild>
        <Link href="/categories/new">New</Link>
      </Button>
    </div>
  )
}
```

Neste código inicial, temos um título `<h2>` e logo depois uma `<div>` onde o formulário para se criar uma categoria será criado. Depois, um botão para voltar à tela de categorias.



As classes `flex flex-row w-full m-5` estilizam a `div` que conterá o formulário. A classe `flex` configura o Flex Box do CSS para que cada elemento filho a ela esteja alinhado na forma `flex-row` ou seja, linha a linha. A classe `w-full` força o `width` para 100% e `m-5` adiciona uma margem de aproximadamente 20 pixels.

### 3.3.3. Estilizando o Título com `@apply`

O título `<h2>` sem o `className` não irá estilizar o texto como um título. Isso acontece porque o Tailwind remove as estilizações de cabeçalho `h1`, `h2`, `h3` etc. Então é necessário criar uma nova estilização para ele.

Para evitar de usar `text-cyan-950 text-sm font-semibold` em todos os cabeçalhos `<h2></h2>`, podemos usar um atributo especial do Tailwind chamado `@apply`, pode que poder adicionado no arquivo `src/app/global.css`. No final deste arquivo, adicione:

```
h2 {  
  @apply text-cyan-950 text-sm font-semibold;  
}
```

Volte ao arquivo `/src/app/categories/new/page.tsx` e deixe somente o `h2`, remova o `className`, e veja que o estilo será aplicado da mesma forma, já que a tag `h2` está sendo estilizada no arquivo `global.css`.

O `@apply` é usado exclusivamente pelo Tailwind para aplicar estilos nos arquivos CSS

### 3.3.4. Criando o Formulário

Vamos criar o formulário contendo o campo `Category Name`:

```
import {Button} from '@components/ui/button'
import Link from 'next/link'

export default function page() {
  return (
    <div>
      <h2>New Category</h2>
      <div className="flex flex-row w-full m-5">
        <form>
          <div className="mb-4">
            <label
              className="block text-gray-700
                text-sm font-bold mb-2"
              htmlFor="name"
            >
              Name
            </label>
            <input
              className="shadow appearance-none border
                rounded w-full py-2 px-3 text-gray-700 leading-tight
                focus:outline-none focus:shadow-outline min-w-[300px]"
              id="name"
              name="name"
              type="text"
              placeholder="Category name"
            />
          </div>
          <div className="flex items-center justify-between">
            <button type="submit">Save</button>
          </div>
        </form>
      </div>
      <Button asChild variant="outline">
        <Link href="/categories">Back</Link>
      </Button>
    </div>
  )
}
```



### 3.3.5. Criando o Método CreateCategory

Neste código, criamos o `<form>` e usamos Tailwind para estilizar o campo. Por isso são atribuídas muitas classes como `shadow`, `text-gray` etc. A princípio, este formulário não possui nenhuma ação quando o usuário clicar no botão `save`. Podemos criar um método que será executado ao clicar no botão utilizando o atributo `action`:

```
import {Button} from '@components/ui/button'
import Link from 'next/link'

export default function page() {
  async function createCategory(formData: FormData) {
    'use server'
    console.log('Create Category with formData', formData)
  }

  return (
    <div>
      <h2>New Category</h2>
      <div className="flex flex-row w-full m-5">
        <form action={createCategory}>...form...</form>
      </div>
      <Button asChild variant="outline">
        <Link href="/categories">Back</Link>
      </Button>
    </div>
  )
}
```

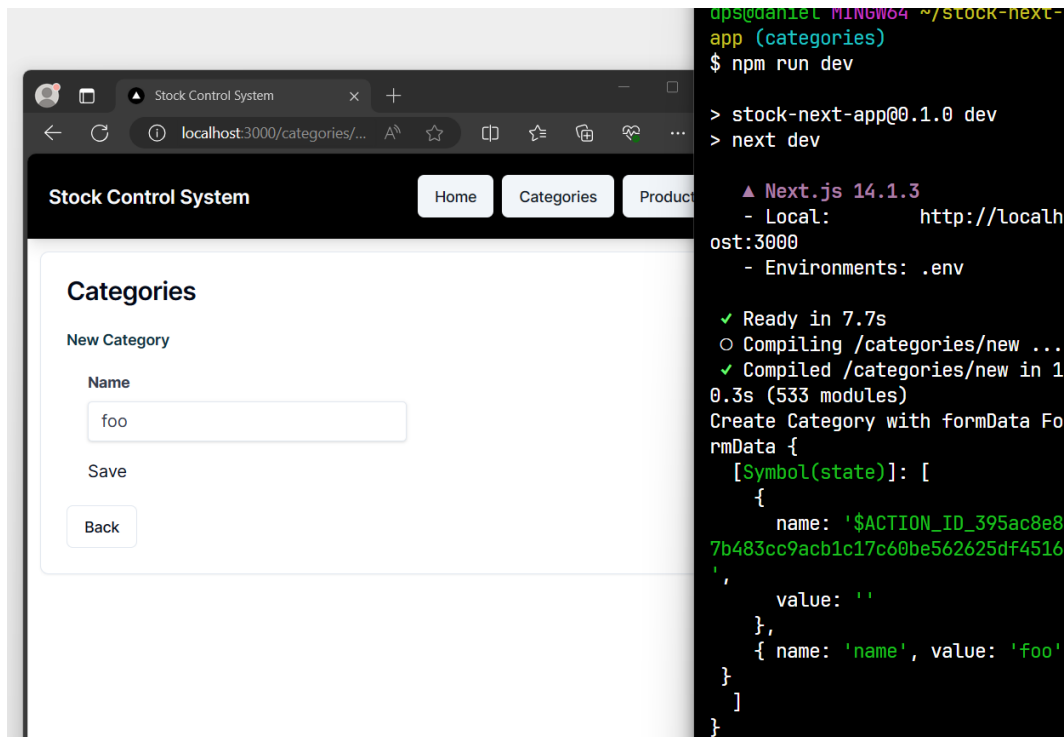
Ao criamos `action={createCategory}` no `form`, estamos dizendo que quando o formulário for submetido através do botão `Save`, o método `createCategory` será executado.

O método `createCategory` usa o `use server` para indicar que este método será executado no servidor e não no cliente. Desta forma é possível, por exemplo, executar consultas ao banco de dados.

Esta é uma característica do Next.js, ele possui partes que são utilizadas no servidor através da instrução `use server` e partes executadas no cliente através da instrução `use client`. Quando um arquivo `tsx` não possui nenhuma instrução `use server` ou `use client`, ele será

executado no servidor.

Neste formulário, por enquanto, temos apenas um `console.log`, que irá mostrar no console do navegador o que é o `formData`.



### 3.3.6. Inserindo Dados na Tabela

Agora podemos usar o Prisma para inserir os dados na tabela `categories`. No método `createCategory` no arquivo `page.tsx`, temos:

```
import { Button } from '@components/ui/button'
import { PrismaClient } from '@prisma/client'
import Link from 'next/link'
import { redirect } from 'next/navigation'

export default function page() {
  async function createCategory(formData: FormData) {
    'use server'
    // console.log('Create Category with formData', formData)

    const formObject = Object.fromEntries(formData)

    if (!formObject.name) {
      throw new Error('Name is required')
    }

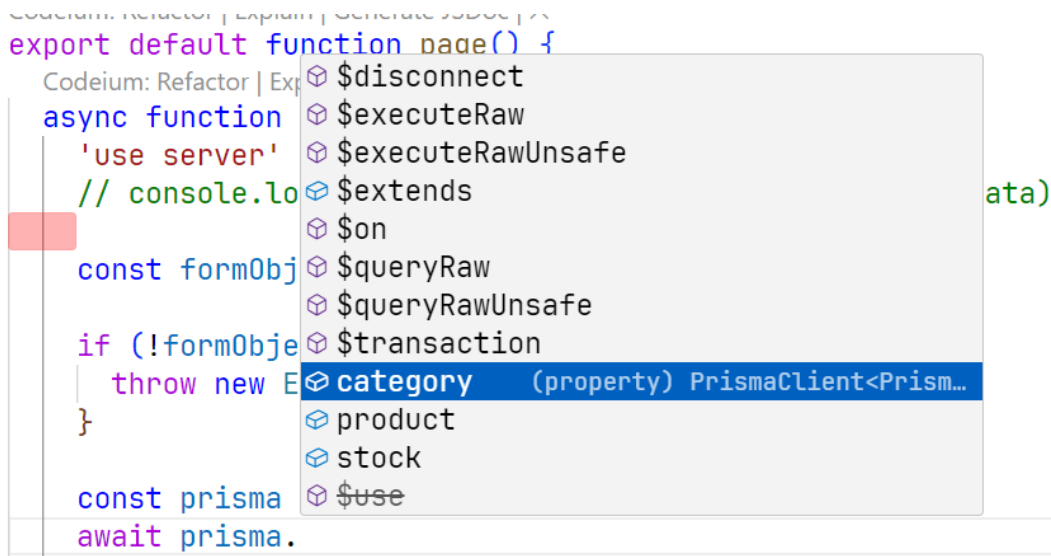
    const prisma = new PrismaClient()
    await prisma.category.create({
      data: {
        name: formObject.name as string
      }
    })

    redirect('/categories')
  }

  return (
    <div>
      <!-- code -->
    </div>
  )
}
```

O método `createCategory` possui o parâmetro `formData`, que é o que é passado para o formulário. A partir do parâmetro `formData`, podemos extrair o valor do campo `name` e verificar se o mesmo foi preenchido. Caso não, o método `createCategory` disparará um erro. Utilizamos o `prismaClient` para criar um novo registro na tabela `categories`. Ao digitar o código `prisma`, você perceberá que o código será complementado pelo Prisma com a opção `category` e depois com o método `create`, onde o atributo `data` é o que será passado para o registro. Por fim, o método `redirect` é usado para redirecionar o usuário para a tela de categorias.

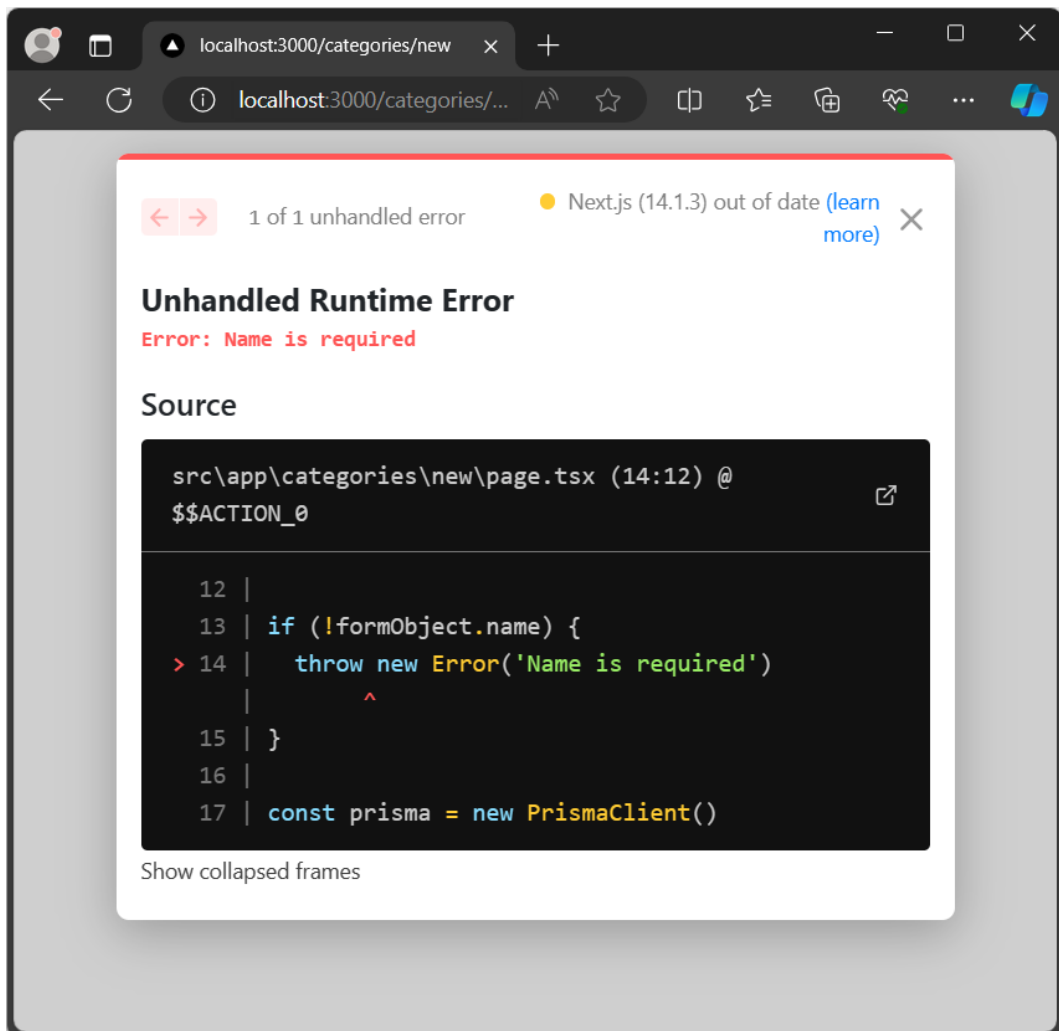
Na imagem abaixo, temos o uso do Prisma para acessar as três tabelas do sistema, que já estão previamente criadas no arquivo `schema.prisma`.



Crie as seguintes categorias: Health, Cleaning Supplies, Food, Canned Goods

### 3.3.7. Tratamento de Erros

Se não digitarmos uma categoria, o `createCategory` disparará um erro `'Name is required'`. Para testar, experimente inserir uma categoria com o campo `name` vazio e veja como o erro irá ser exibido ao usuário:



Esta seria a primeira forma de erro a ser exibida para o usuário, mas podemos concordar que não é a melhor forma. A melhor forma seria mostrar um erro em vermelho logo abaixo do campo Category Name, e vamos mostrar como podemos fazer isso no próximo capítulo, pois ele envolve algumas configurações extras.

A princípio, vamos usar um recurso nativo do Next.js. Ao criar o arquivo `error.tsx` no diretório `src/app/categories/new`, qualquer erro sendo lançado pelo `createCategory` será redirecionado para este arquivo.

Crie o arquivo `error.tsx` no diretório `src/app/categories/new` com o seguinte código:

```
'use client'

import {Button} from '@components/ui/button'

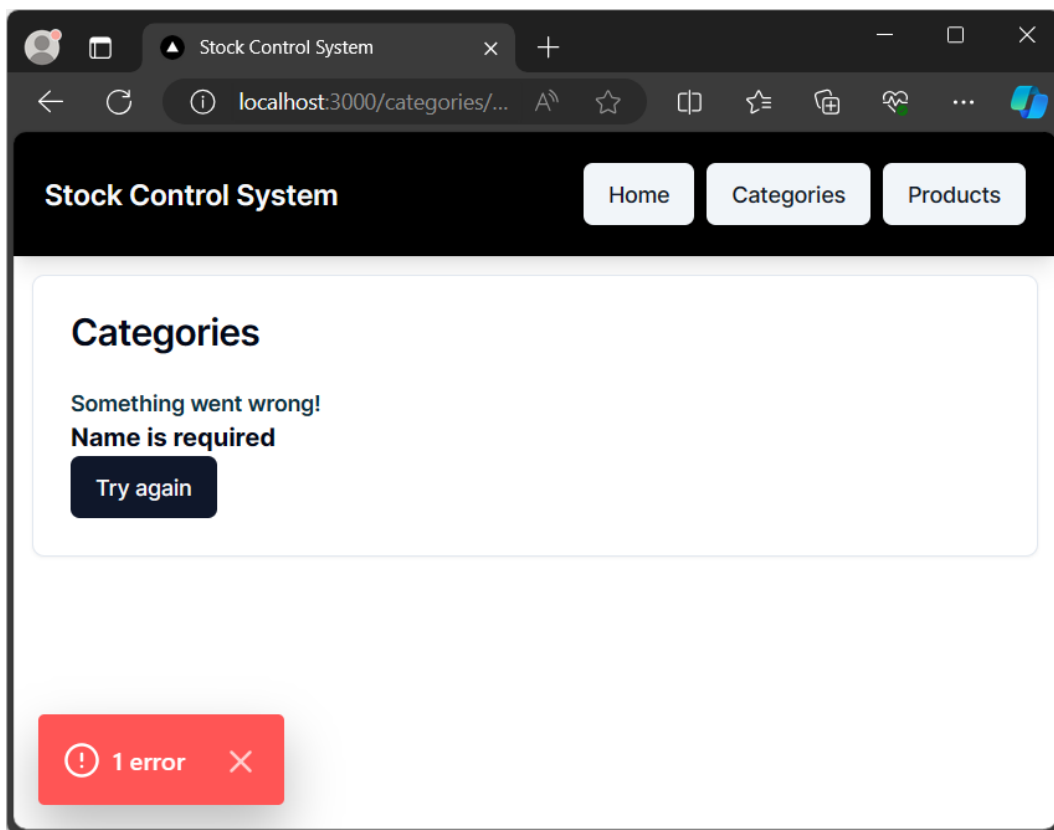
export default function Error({
  error,
  reset
}): {
  error: Error & {digest?: string}
  reset: () => void
}) {
  return (
    <div>
      <h2>Something went wrong!</h2>
      <b>{error.message}</b>
      <br />
      <Button
        onClick={
          // Attempt to recover by trying to re-render the segment
          () => reset()
        }
      >
        Try again
      </Button>
    </div>
  )
}
```

Neste código, o parâmetro `error` é o erro que foi gerado pelo `createCategory`, e o parâmetro `reset` é a função que é chamada para tentar recriar a rota. Exibimos uma mensagem amigável para o usuário através do parâmetro `error.message` que é o texto repassado no parâmetro do `Error`.

O uso do `use client` é necessário para indicar que este componente será executado no cliente, dessa forma o método `reset` pode ser chamado para tentar recriar a rota.

Como o `error.tsx` está no diretório `src/app/categories/new`, somente exceções desta rota serão exibidas neste formato. Se o arquivo `error.tsx` estiver em uma rota acima, ela poderá abranger outras rotas.

Agora, ao causar o erro, deixando o campo `Category Name` vazio, temos a seguinte mensagem de erro:



### 3.4. Exibindo uma Tela de “loading”

O Next.js possui um recurso que permite mostrar uma tela de loading ao usuário. Para isso, podemos criar um arquivo `loading.tsx` no diretório `src/app/categories` com o seguinte código:

```
export default function Loading() {  
  return <div>Loading...</div>  
}
```

Como o arquivo `loading.tsx` foi criado no diretório `categories`, ele estará configurado para ser exibido em qualquer rota dentro do `/categories`. Sempre que uma tela estiver sendo carregada será exibido o conteúdo `<div>Loading...</div>`. Para conseguir ver este recurso

em um ambiente “localhost”, onde o acesso ao servidor é bem rápido, você pode entrar no “Dev Tools” do seu navegador e na aba “Networking” usar a opção Throttling escolher o valor fast 3g.

### 3.4.1. Exibindo um “loading” no Botão “Save”

Quando clicamos no botão “Save”, a tela de “Loading...” não aparece, já que está sendo realizando um submit na página. Para que possamos contornar este problema, podemos criar um componente chamado Submit, que é um botão capaz de alterar o seu estado de acordo com o estado do formulário.

No diretório /src/app/categories, crie o arquivo Submit.tsx com o seguinte código:

```
'use client'

import {useFormStatus} from 'react-dom'
import {Button} from '../../components/ui/button'

export default function Submit({children}: React.PropsWithChildren) {
  const status = useFormStatus()
  return (
    <Button
      type="submit"
      disabled={status.pending}
      aria-disabled={status.pending}
    >
      {children}
    </Button>
  )
}
```

Neste código, o parâmetro children é o conteúdo que será exibido no botão. A propriedade aria-disabled é usada para tornar o botão desabilitado caso o formulário esteja em processo de Submit. Usamos o 'use client' para que o botão seja renderizado no cliente e desta forma podemos usar o useFormStatus() que é um hook do react-dom para saber o estado do formulário.



Sempre que usamos um hook do react, usamos o `use client` para que o componente seja renderizado no cliente.

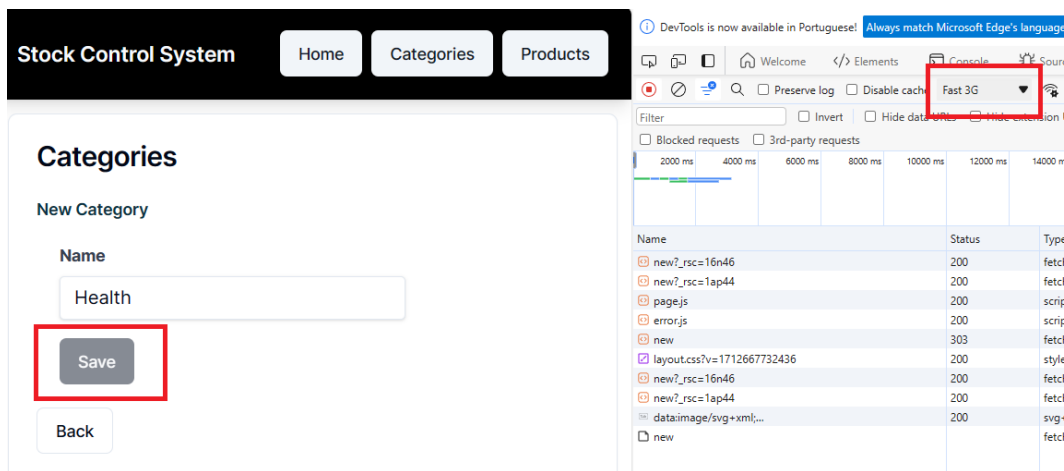
Com o componente pronto, podemos adicioná-lo ao formulário no arquivo `/src/app/categories/new/page.tsx`:

```
// imports
import Submit from '../Submit'

export default function page() {
  async function createCategory(formData: FormData) {
    'use server'
    // code
  }

  return (
    <div>
      <h2>New Category</h2>
      <div className="flex flex-row w-full m-5">
        <form action={createCategory}>
          <div className="mb-4">
            <!-- fields -->
          </div>
          <div className="flex items-center justify-between">
            <Submit>Save</Submit>
          </div>
        </form>
      </div>
      <!-- Back Button -->
    </div>
  )
}
```

Ao inserir o componente `Submit` no formulário, e adicionar uma nova categoria, percebe-se que por um instante o botão ficará em um estado de `disabled`:



### 3.4.2. Corrigindo o Posicionamento do Botão “Save”

Como podemos ver nos últimos exemplos, o botão Save não ficou posicionado corretamente com o botão Back. Para corrigir isso, vamos usar o Tailwind e também FlexBox, da seguinte forma:

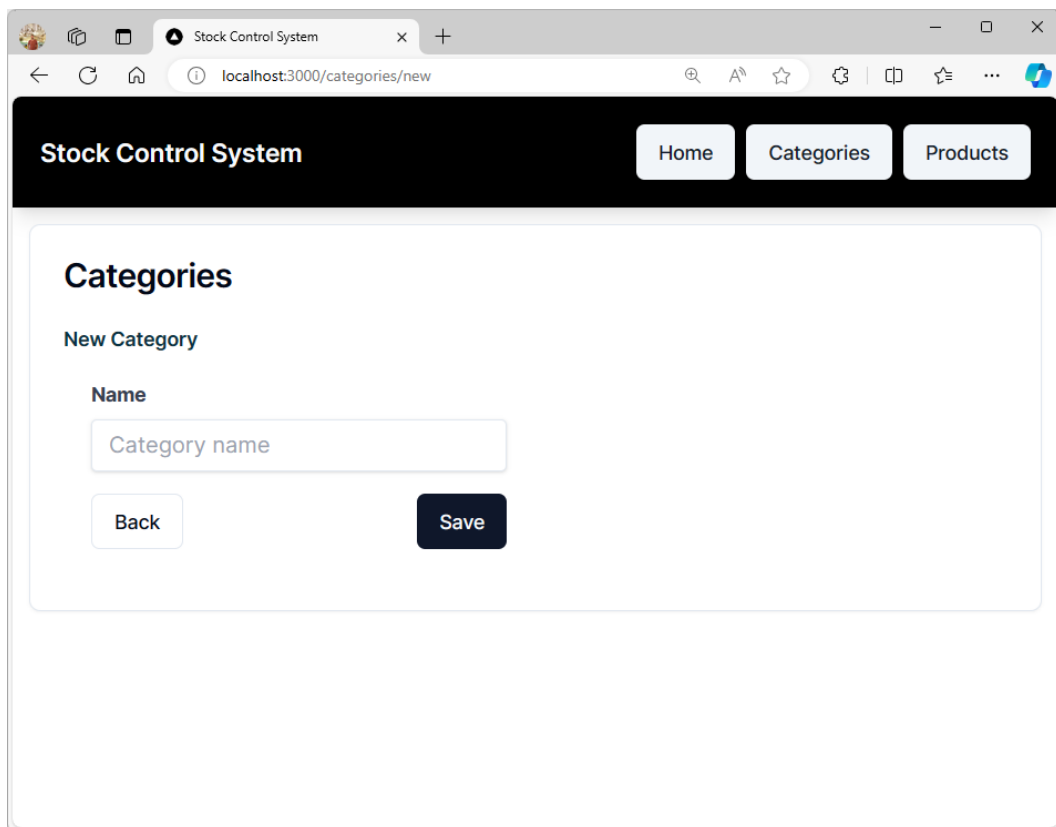
```
/src/app/categories/new/page.tsx
```

```
// imports
import Submit from '../Submit'

export default function page() {
  async function createCategory(formData: FormData) {
    'use server'
    // code
  }

  return (
    <div>
      <h2>New Category</h2>
      <div className="flex flex-row w-full m-5">
        <form action={createCategory}>
          <div className="mb-4">
            <!-- fields -->
          </div>
          <div className="flex items-center justify-between">
            <Submit>Save</Submit>
            <Button asChild variant="outline">
              <Link href="/categories">Back</Link>
            </Button>
          </div>
        </form>
      </div>
    </div>
  )
}
```

Neste código, criamos uma div utilizando `flex items-center justify-between`, e o botão Back ficou no lado esquerdo e o botão Save ficou no lado direito, graças ao `justify-between`.



The screenshot shows a web browser window with the title 'Stock Control System'. The address bar displays 'localhost:3000/categories/new'. The page has a dark header with the title 'Stock Control System' and three navigation buttons: 'Home', 'Categories', and 'Products'. The main content area is titled 'Categories' and contains a section for 'New Category'. This section includes a label 'Name' above a text input field with the placeholder 'Category name'. Below the input field are two buttons: 'Back' and 'Save'.

### 3.5. Exibindo as Categorias

Após inserir algumas categorias ainda não é possível ver elas em `/categories`. No capítulo anterior vimos o comando `db:studio` que exibe as tabelas do sistema. Você pode usá-las a princípio para ver e editar as categorias que criou.

Para exibir as categorias, podemos usar o componente `Table` do `shadcn`, que é uma tabela responsiva semelhante a imagem a seguir:

# Table

A responsive table component.

Preview Code

Style: New York ↕

Invoice	Status	Method	Amount
INV001	Paid	Credit Card	\$250.00
INV002	Pending	PayPal	\$150.00
INV003	Unpaid	Bank Transfer	\$350.00
INV004	Paid	Credit Card	\$450.00
INV005	Paid	PayPal	\$550.00
INV006	Pending	Bank Transfer	\$200.00
INV007	Unpaid	Credit Card	\$300.00
Total			\$2,500.00

A list of your recent invoices.

## 3.5.1. Instalando o Componente Table

Para instalar o componente `Table` do `shadcn`, execute o seguinte comando:

```
npx shadcn-ui@latest add table
```

A instalação irá criar o arquivo `src\components\ui\table.tsx`, que poderá ser utilizado para criar a tabela de categorias.

### 3.5.2. Exibindo Todas as Categorias

No arquivo `src\app\categories\page.tsx` iremos obter todas as categorias que já foram criadas, e isso pode ser realizado de uma forma muito simples através do prisma:

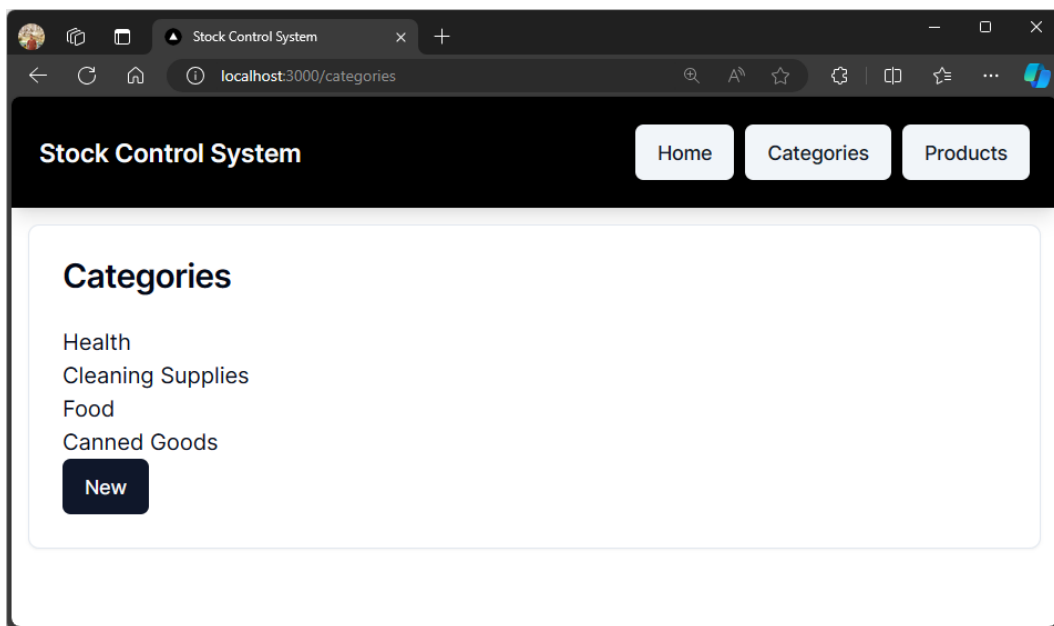
```
import {Button} from '@components/ui/button'
import {PrismaClient} from '@prisma/client'
import Link from 'next/link'

export default async function page() {
  const prisma = new PrismaClient()
  const categories = await prisma.category.findMany()

  return (
    <div>
      {categories.map((category) => (
        <div key={category.id}>{category.name}</div>
      ))}
      <Button asChild>
        <Link href="/categories/new">New</Link>
      </Button>
    </div>
  )
}
```

Ao usarmos o `PrismaClient` para obter todas as categorias, precisamos transformar a função `page` em uma função assíncrona. Utilizamos `await prisma.category.findMany()` para obter todas as categorias e as exibimos de uma forma muito simples através do `map`, uma função nativa do JavaScript para percorrer arrays.

O código acima produz o seguinte resultado:



### 3.5.3. Utilizando o Componente `Table`

Ao invés do `map`, vamos utilizar o componente `Table` do `shadcn` para exibir as categorias:

```
import {Button} from '@/components/ui/button'
import {
  Table,
  TableCaption,
  TableHeader,
  TableRow,
  TableHead,
  TableBody,
  TableCell
} from '@/components/ui/table'
import {PrismaClient} from '@prisma/client'
import Link from 'next/link'

export default async function page() {
  const prisma = new PrismaClient()
  const categories = await prisma.category.findMany()
```

```

return (
  <div>
    <Table>
      <TableCaption>A list of your categories.</TableCaption>
      <TableHeader>
        <TableRow>
          <TableHead>Name</TableHead>
          <TableHead className="w-[100px] text-center">Actions</TableHead>
        </TableRow>
      </TableHeader>
      <TableBody>
        {categories.map((category) => (
          <TableRow key={category.id}>
            <TableCell>{category.name}</TableCell>
            <TableCell className="text-right">
              <div className="flex flex-row gap-2">
                <Button asChild variant="link">
                  <Link href={`\categories/edit/${category.id}`}>Edit</Link>
                </Button>
                <Button asChild variant="link">
                  <Link href={`\categories/del/${category.id}`}>Delete</Link>
                </Button>
              </div>
            </TableCell>
          </TableRow>
        ))}
      </TableBody>
    </Table>
    <Button asChild>
      <Link href="/categories/new">New Category</Link>
    </Button>
  </div>
)
}

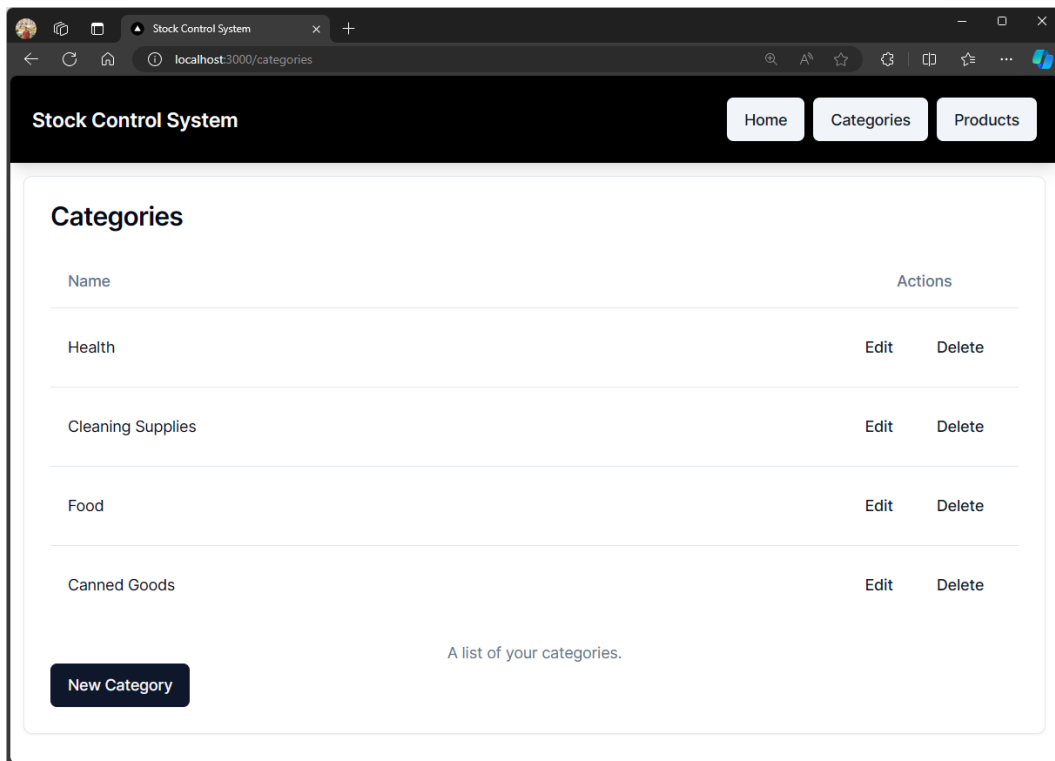
```

Neste código utilizamos o `<Table>` do `shadcn` contendo três componentes internos. O `<TableCaption>` vai renderizar uma mensagem de rodapé. O `<TableHead>` vai renderizar as linhas de cabeçalho da tabela. O `<TableBody>` vai renderizar todas as linhas da tabela.

O `TableHead` possui dois campos, sendo o segundo um campo destinado a ter botões de ações como `Edit` e `Delete`. O `<TableBody>` vai renderizar todas as linhas da tabela incluindo o nome da categoria e renderizando os dois botões para editar e remover uma categoria.



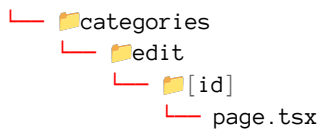
O resultado do código acima é semelhante a figura a seguir:



## 3.6. Editando uma Categoria

### 3.6.1. Configurando a Rota

Na listagem de categorias temos um botão que redireciona para o path `/categories/edit/${category.id}` onde `category.id` é o id da categoria que queremos editar. No next, quando temos uma rota dinâmica, onde o id muda constantemente, devemos usar `[id]` como uma pasta, para configurar que id é dinâmico, como no exemplo a seguir:



```
└─ categories
   └─ edit
      └─ [id]
         └─ page.tsx
```

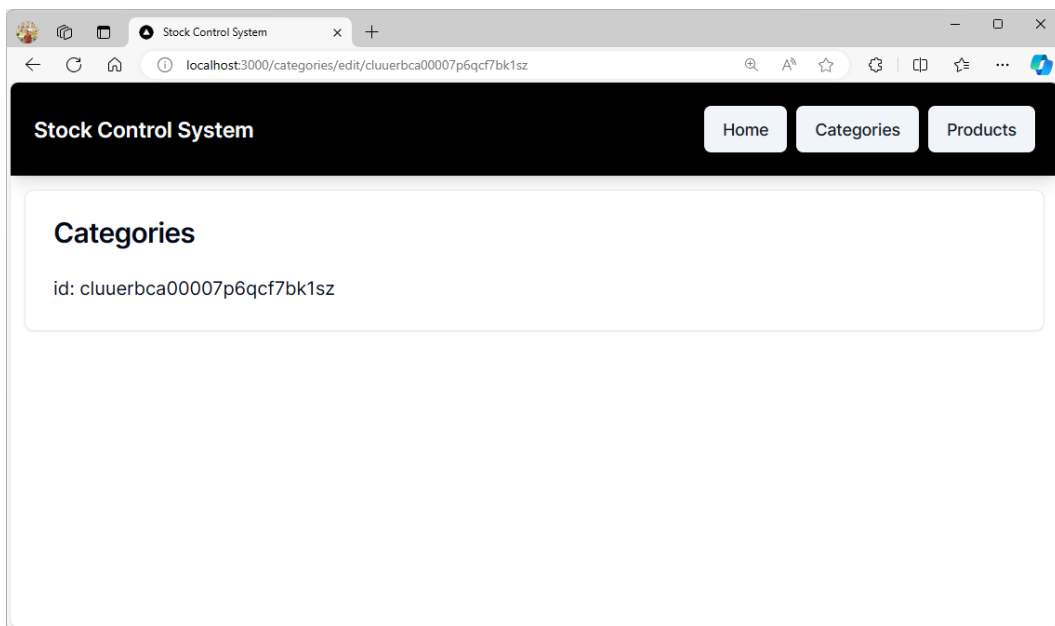
Crie o diretório `/categories/edit` e depois `/categories/edit/[id]` e depois crie o arquivo `/categories/edit/[id]/page.tsx`.

No vscode, pode-se criar tudo de uma vez só, bastando selecionar o diretório já existente `categories` e então selecionando `New File` e então digitando `edit/[id]/page.tsx`

O componente `edit/[id]/page.tsx` necessita obter o `id` da url, e isso é realizado através do `{params}` que já conhecemos do React. A versão inicial do componente é exibida a seguir:

```
export default function page({params}) {
  const id = params.id
  return <div>id: {id}</div>
}
```

Este código resulta na seguinte imagem:



Perceba que o “loading” aparece antes da página carregar, já que o diretório `edit/[id]/` reutiliza as funções do Next.js dos diretórios superiores, como `loading.tsx` e `error.tsx`.

### 3.6.2. Definido um Tipo para o Parâmetro `params`

No `vscode`, podemos perceber que existe um pequeno erro no `{params}` e o erro é `Binding element 'params' implicitly has an 'any' type`. Estamos utilizando `Typescript` por um motivo, para configurar que os parâmetros tenham um tipo definido. Neste componente, temos um parâmetro `params` com a propriedade `id` (já que temos o diretório `[id]`), então podemos criar e utilizar o seguinte tipo:

```
interface PageParams {  
  params: {  
    id: string  
  }  
}  
  
export default function page({params}: PageParams) {  
  const id = params.id  
  return <div>id: {id}</div>  
}
```

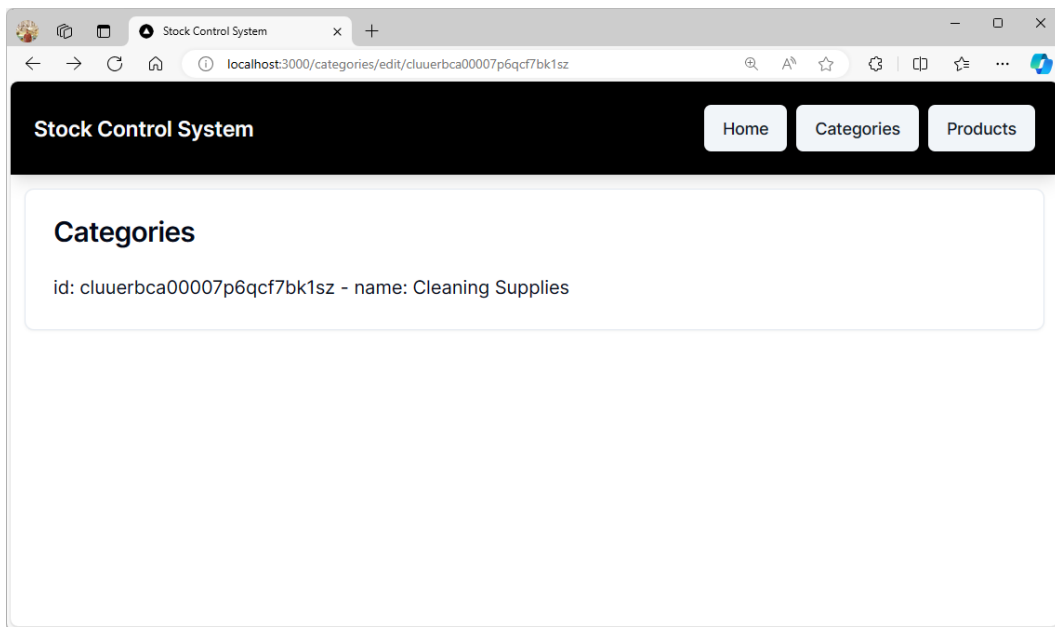
Ao criarmos o tipo `PageParams` na interface, podemos utilizá-lo no `{params}` e então deixarmos este parâmetro tipado.

### 3.6.3. Obtendo uma Categoria pelo id

Com o `id` podemos utilizar o `PrismaClient` para recuperar a categoria:

```
import {PrismaClient} from '@prisma/client'  
  
interface PageParams {  
  params: {  
    id: string  
  }  
}  
  
export default async function page({params}: PageParams) {  
  const id = params.id  
  const prisma = new PrismaClient()  
  const category = await prisma.category.findUniqueOrThrow({where: {id}})  
  
  return (  
    <div>  
      id: {id} - name: {category.name}  
    </div>  
  )  
}
```

O método `findUniqueOrThrow` irá buscar por uma categoria dado o `id` fornecido, e caso não encontre, irá disparar um erro. Caso a categoria exista, o nome será exibido na página, semelhante a imagem a seguir:



### 3.6.4. Criando o Formulário para Editar uma Categoria

No formulário anterior que criava uma categoria, nós o criamos totalmente de forma “server mode”. Desta forma, não podemos por exemplo utilizar alguns recursos que são executados no cliente (navegador). Por exemplo, não é possível utilizar React Hooks em um componente server-side.

O Nextjs lida com isso através da composição entre componentes. Um componente “server mode” como o `edit/[id]/page.tsx` pode possui um ou mais componente no modo client side.

Para compreendermos este processo, crie o arquivo `form.tsx` no diretório `src/app/categories/edit/[id]/` com o seguinte código:

```
'use client'

import {useState} from 'react'

export default function UpdateForm() {
  const [count, setCount] = useState(0)
  const handleClick = () => setCount(count + 1)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>Click me</button>
    </div>
  )
}
```

Neste código, usamos o `use client` para que o componente seja renderizado no cliente. Sem isso, o `useState` não funcionaria, pois ele é uma propriedade do cliente. Então, criamos um simples código que incrementa o valor do `useState` e renderiza o componente, algo bem simples apenas para que possamos compreender o processo de renderização do React.

Com o componente pronto, podemos adicionar o formulário ao arquivo `src/app/categories/edit/[id]/page.tsx`:

```
// imports
import UpdateForm from './form'

// code

export default async function page({params}: PageParams) {
  // code
  return (
    <div>
      <div id={id} - name={category.name}>
        <br />
        <UpdateForm />
      </div>
    </div>
  )
}
```

Adicionar um componente `client mode` a um componente `server mode` é algo totalmente possível e natural para um componente do Next. O contrário já não é possível.

Um componente Nextjs é por padrão “server mode”. Os componentes “client mode” devem possuir na primeira linha a instrução ‘use client’.

### 3.6.5. Repassando Dados Entre Componentes

Como temos o componente Page que obtém os dados da categoria através do id, e o componente UpdateForm que irá renderizar o formulário, podemos passar esses dados entre eles. Podemos criar uma interface que irá conter os dados que serão repassados do Page para o UpdateForm. Altere o arquivo form.tsx para:

```
'use client'
import {useState} from 'react'

interface UpdateFormProps {
  data: {
    id: string
    name: string
  }
}

export default function UpdateForm(props: UpdateFormProps) {
  const category = props.data
  return (
    <div>
      Category Form: {category.id} - {category.name}
    </div>
  )
}
```

Após criar a interface UpdateFormProps, atribuímos este tipo ao parâmetro props do componente UpdateForm. Então, podemos usar props.data para obter os dados da categoria. Mas, onde podemos obter esses dados? No componente page.tsx que já obteve o id da url e os dados pelo PrismaClient:

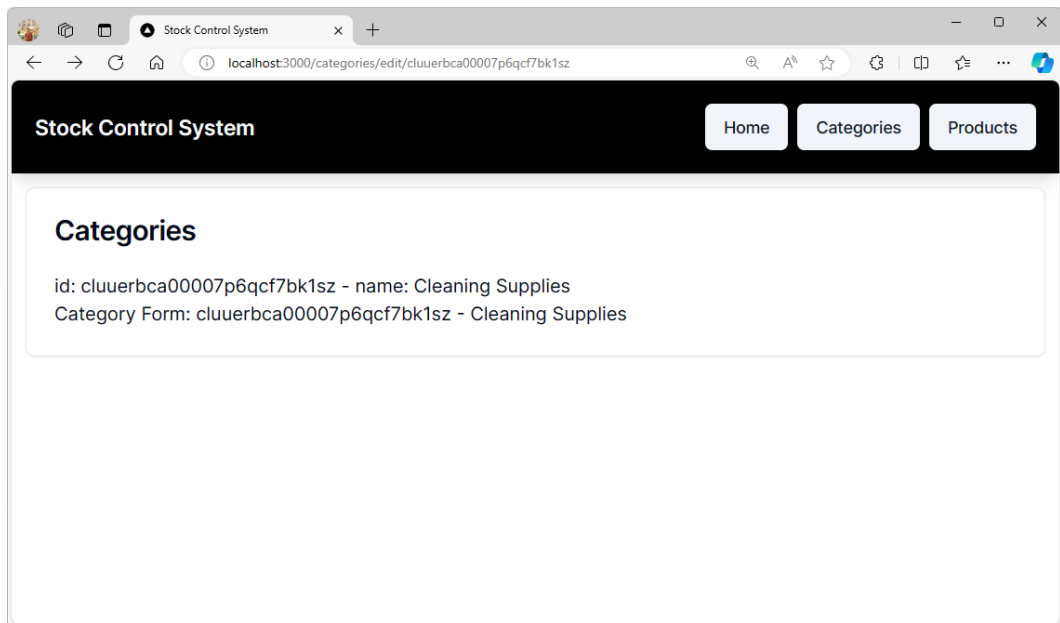
```
import {PrismaClient} from '@prisma/client'
import UpdateForm from './form'

// code

export default async function page({params}: PageParams) {
  // code
  const category = await prisma.category.findUniqueOrThrow({where: {id}})

  return (
    <div>
      id: {id} - name: {category.name}
      <br />
      <UpdateForm data={category} />
    </div>
  )
}
```

Com o `<UpdateForm data={category} />`, estaremos repassando o objeto `category` para a propriedade `data` do componente `UpdateForm`. O resultado é semelhante a imagem abaixo:



Temos então a mesma informação sendo exibida no componente `Page` e também repassada



para o componente `UpdateForm`. Para reorganizar o `page.tsx` vamos apenas deixar o `UpdateForm`:

```
import { PrismaClient } from '@prisma/client'
import UpdateForm from '../form'

// code

export default async function page({ params }: PageParams) {
  // code
  const category = await prisma.category.findUniqueOrThrow({ where: { id } })
  return <UpdateForm data={category} />
}
```

### 3.6.6. Preenchendo os Dados do Formulário

O componente `UpdateForm` agora possui os dados da categoria e podemos criar o formulário. Desta vez, ao invés de usar um formulário totalmente “server mode”, vamos criar um formulário “client mode” usando React.

```
'use client'
import {useState} from 'react'
import Submit from '../../Submit'

interface UpdateFormProps {
  data: {
    id: string
    name: string
  }
}

export default function UpdateForm(props: UpdateFormProps) {
  const [name, setName] = useState(props.data.name)
  function handleChange(event: any) {
    setName(event.target.value)
  }

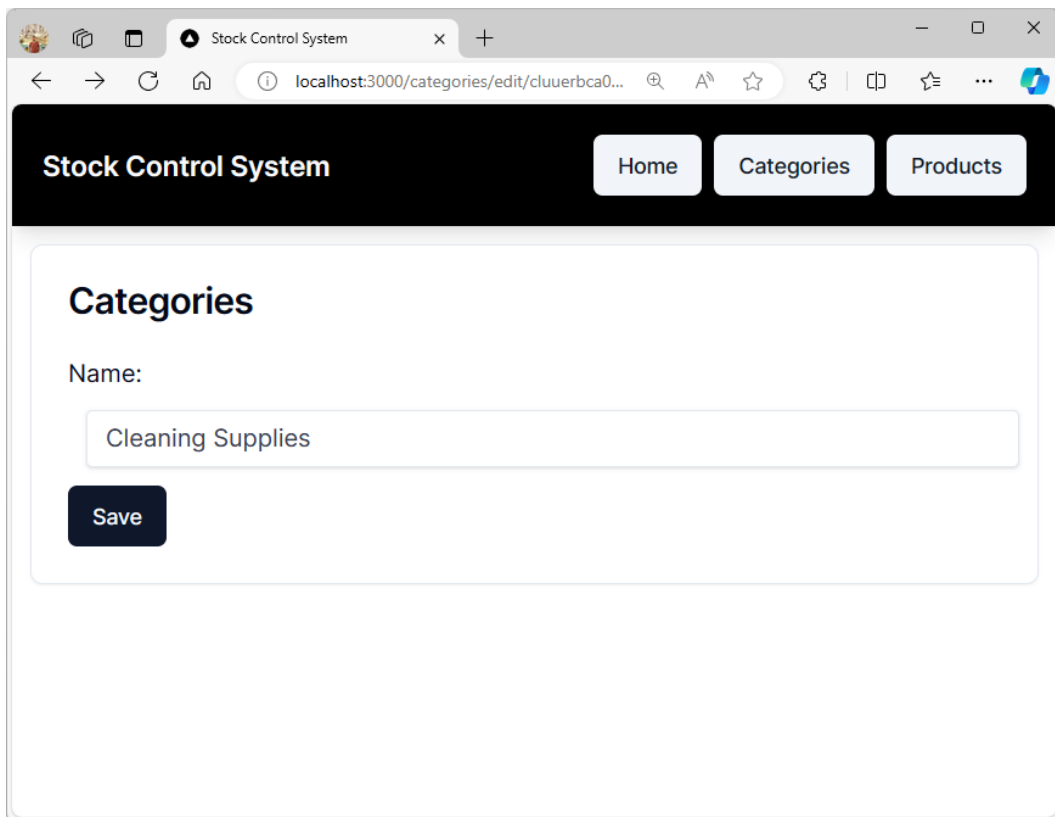
  return (
    <div>
```

```

    <form>
      <input type="hidden" name="id" value={props.data.id} />
      <label>
        Name:
        <input
          type="text"
          className="shadow appearance-none border rounded w-full
            py-2 px-3 m-3 text-gray-700
            leading-tight focus:outline-none
            focus:shadow-outline min-w-[300px]"
          id="name"
          name="name"
          value={props.data.name}
          onChange={handleChange}
        />
      </label>
      <Submit>Save</Submit>
    </form>
  </div>
)
}

```

A tela para editar uma categoria é semelhante a figura a seguir:



The screenshot shows a web browser window with the title 'Stock Control System'. The address bar displays 'localhost:3000/categories/edit/cluuerbca0...'. The page has a dark header with the title 'Stock Control System' and three navigation buttons: 'Home', 'Categories', and 'Products'. The main content area is titled 'Categories' and contains a form with the label 'Name:'. The input field contains the text 'Cleaning Supplies'. Below the input field is a dark blue button labeled 'Save'.

Graças ao React, através do `useState`, é possível alterar a propriedade `value` do campo `name` do formulário.

### 3.6.7. Editando a Categoria

O que precisamos configurar agora é o botão `Submit`. A primeira alternativa seria adicionar um método chamado `saveCategory`, como por exemplo:

```

'use client'
// imports

// code

export default function UpdateForm(props: UpdateFormProps) {
  // code

  async function saveCategory(formData: FormData) {
    'use server'
    console.log('Save Category with formData', formData)
  }

  return (
    <div>
      <form action={saveCategory}>
        <!-- code -->
      </form>
    </div>
  )
}

```

Este código não funcionará, teremos o seguinte erro: It is not allowed to define inline "use server" annotated Server Actions in Client Components. Para resolver este problema, precisamos chamar a função `saveCategory` em um componente `server mode`, e neste caso, podemos criar um evento em `UpdateFormProps` que será justamente o evento chamado `submit`, então o `saveCategory` será realizado no componente `Page`.

```

'use client'
import { useState } from 'react'
import Submit from '../Submit'

interface UpdateFormProps {
  onSubmit: string | ((formData: FormData) => void) | undefined
  data: {
    id: string
    name: string
  }
}

export default function UpdateForm(props: UpdateFormProps) {
  const [name, setName] = useState(props.data.name)

```

```

function handleChange(event: any) {
  setName(event.target.value)
}

return (
  <div>
    <form action={props.onSubmit}>
      <!-- form -->
    </form>
  </div>
)
}

```

Agora, o componente `UpdateForm` tem as seguintes mudanças:

A interface `UpdateFormProps` recebeu a propriedade `onSubmit` que é um evento do action do formulário, por isso precisa ter o tipo `string | ((formData: FormData) => void) | undefined`

O `<form>` possui o evento `onSubmit` através do `action={props.onSubmit}`, ou seja, quando o usuário submeter o formulário ao clicar no botão `Submit`, o método estabelecido em `props.submit` será executado. Mas onde está esse método? Ele está no `page.tsx`, já que é nele que adicionamos o `<UpdateForm...>`:

```

import {PrismaClient} from '@prisma/client'
import UpdateForm from './form'

interface PageParams {
  params: {
    id: string
  }
}

export default async function page({params}: PageParams) {
  const id = params.id
  const prisma = new PrismaClient()
  const category = await prisma.category.findUniqueOrThrow({where: {id}})

  async function saveCategory(formData: FormData) {
    'use server'
    console.log('Save Category with formData', formData)
  }
}

```

```

    return <UpdateForm data={category} onSubmit={saveCategory} />
  }

```

Agora, o componente page possui o método saveCategory, que deve explicitamente marcado com 'use server' para que seja executado no servidor. Por enquanto, ele apenas faz um console.log dos dados do formulário. O evento onSubmit é configurado no elemento <UpdateForm /> chamando o método saveCategory.

O código a seguir irá atualizar os dados da categoria:

```

import {PrismaClient} from '@prisma/client'
import UpdateForm from './form'
import {redirect} from 'next/navigation'

interface PageParams {
  params: {
    id: string
  }
}

export default async function page({params}: PageParams) {
  const id = params.id
  const prisma = new PrismaClient()
  const category = await prisma.category.findUniqueOrThrow({where: {id}})

  async function saveCategory(formData: FormData) {
    'use server'
    console.log('Save Category with formData', formData)

    if (!formData.get('id')) {
      throw new Error('Id is required')
    }

    if (!formData.get('name')) {
      throw new Error('Name is required')
    }

    const id = formData.get('id') as string
    const name = formData.get('name') as string

    const prisma = new PrismaClient()
    await prisma.category.update({

```

```

      where: {id},
      data: {name}
    })

    redirect('/categories')
  }

  return <UpdateForm data={category} onSubmit={saveCategory} />
}

```

Agora, o método `saveCategory` está completo. Inicialmente ele verifica se as variáveis `id` e `name` estão vindas corretamente do formulário. Se estiverem corretamente, ele tenta atualizar a categoria. Caso de certo, o redirecionamento `redirect` é chamado para redirecionar o usuário para a tela de categorias. Se por exemplo, o usuário não digitar o `name`, um erro será disparado.

### 3.6.8. Exibindo Mensagens de Erro

Ao deixar o campo nome em branco, um erro no Next.js é exibido. Já vimos no formulário para criar uma categoria que ao criar um componente `error.tsx` este erro é exibido ao usuário. Até o momento, criamos o arquivo `/src/app/categories/new/error.tsx` para mostrar o erro no formulário de criar uma categoria. Mas e para editar? Aqui temos duas saídas. Podemos criar um novo arquivo `/src/app/categories/edit/error.tsx` com o mesmo código, ou podemos então mover o arquivo `/src/app/categories/new/error.tsx` um diretório acima, dessa forma o `error.tsx` será disponível para a criação de uma categoria, ou a edição.

A configuração de arquivos e diretórios até o momento é semelhante a estrutura a seguir:

```

├── categories
│   ├── new
│   │   ├── page.tsx
│   │   └── error.tsx
│   ├── edit
│   │   ├── [id]
│   │   │   ├── form.tsx
│   │   │   └── page.tsx
│   │   └── error.tsx
│   ├── layout.tsx
│   ├── loading.tsx
│   └── page.tsx

```

## 3.7. Excluindo uma Categoria

Ao analisarmos o botão para excluir uma categoria na tabela, temos:

```
<Button asChild variant="link">
  <Link href={` /categories/del/${category.id}`}>Delete</Link>
</Button>
```

Veja que estamos redirecionando a aplicação para a rota `/categories/del/${category.id}`. Mas para excluir uma categoria, não há a necessidade de ir para uma nova rota. O que precisamos fazer é perguntar ao usuário se ele deseja mesmo excluir a categoria e, em caso afirmativo, realizar a ação.

### 3.7.1. Adicionando o Componente Alert Dialog

Para perguntar ao usuário, iremos usar um componente do shadcn chamado Alert Dialog. Primeiramente, vamos instalá-lo através do comando:

```
npx shadcn-ui@latest add alert-dialog
```

Este comando adiciona o componente `alert-dialog.tsx` em `src/components/ui`. Vamos editar o arquivo `src/app/categories/page.tsx` para incluir o componente `alert-dialog`:

```
// imports
import {
  AlertDialog,
  AlertDialogAction,
  AlertDialogCancel,
  AlertDialogContent,
  AlertDialogDescription,
  AlertDialogFooter,
  AlertDialogHeader,
  AlertDialogTitle,
  AlertDialogTrigger
} from '@components/ui/alert-dialog'

export default async function page() {
  const prisma = new PrismaClient()
  const categories = await prisma.category.findMany()
```



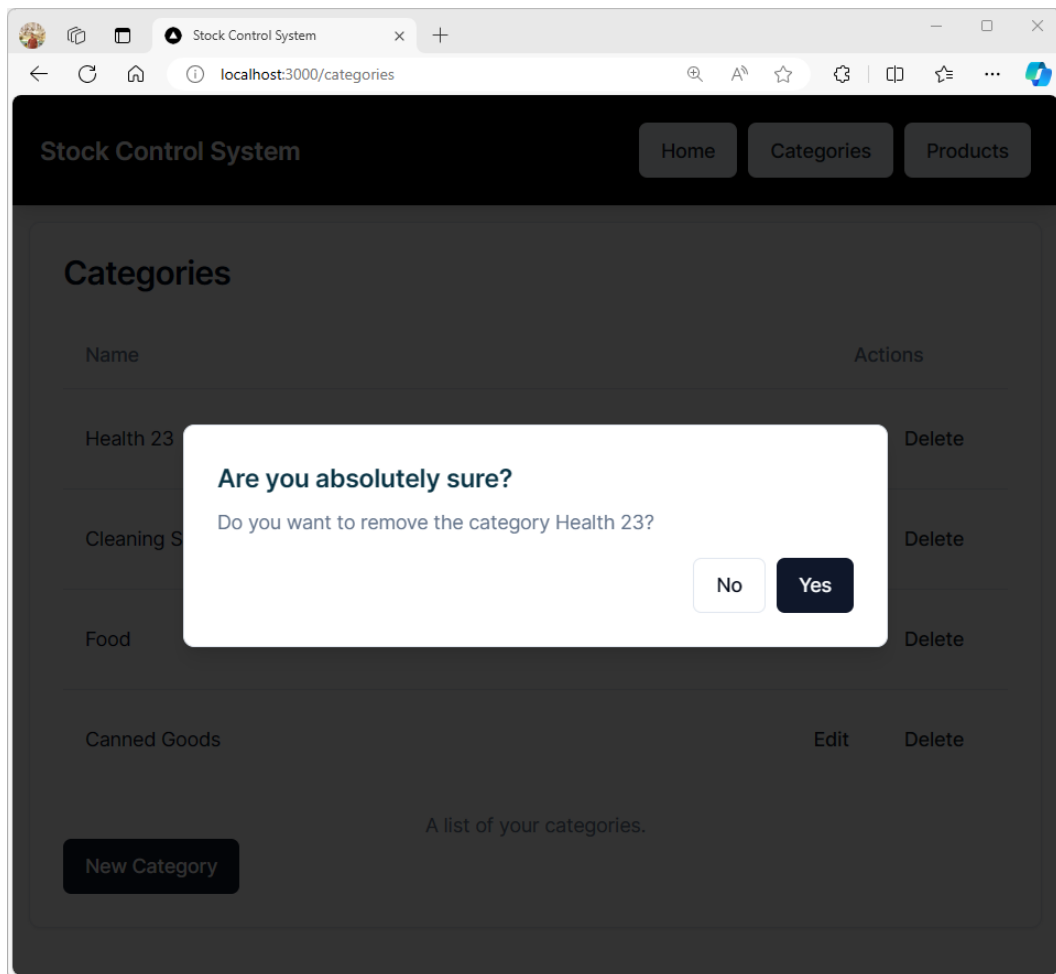
```

return (
  <div>
    <Table>
      <TableCaption>A list of your categories.</TableCaption>
      <TableHeader>
        <TableRow>
          <!-- code -->
        </TableRow>
      </TableHeader>
      <TableBody>
        {categories.map((category) => (
          <TableRow key={category.id}>
            <TableCell>{category.name}</TableCell>
            <TableCell className="text-right">
              <div className="flex flex-row gap-2">
                <Button asChild variant="link">
                  <Link href={` /categories/edit/${category.id}`}>Edit</Link>
                </Button>
                { /* <Button asChild variant="link">
                  <Link href={` /categories/del/${category.id}`}>Delete</Link>
                </Button> */}
                <AlertDialog>
                  <AlertDialogTrigger asChild>
                    <Button variant="link">Delete</Button>
                  </AlertDialogTrigger>
                  <AlertDialogContent>
                    <AlertDialogHeader>
                      <AlertDialogTitle>Are you absolutely sure?</AlertDialog\
Title>
                    <AlertDialogDescription>
                      Do you want to remove the category {category.name}?
                    </AlertDialogDescription>
                  </AlertDialogHeader>
                  <AlertDialogFooter>
                    <AlertDialogCancel>No</AlertDialogCancel>
                    <AlertDialogAction>Yes</AlertDialogAction>
                  </AlertDialogFooter>
                </AlertDialogContent>
              </AlertDialog>
            </div>
          </TableCell>
        </TableRow>

```

```
    )})  
  </TableBody>  
</Table>  
</div>  
)  
}
```

Neste código, comentamos o Link para remover uma categoria e adicionamos um AlertDialog com as opções Yes e No. O código acima é semelhante a imagem a seguir:



Uma boa prática na programação é tentar, sempre que possível, dividir o código principal de

uma página em pequenos componentes. Este componente `Alert Dialog` dentro de um laço do `categories.map` e dentro de uma célula da tabela não está bom.

### 3.7.2. Criando o Componente `Delete Dialog`

Podemos então criar um componente chamado `DeleteDialog` que vai exibir uma mensagem de `Alert` e disparar um método se o usuário clicar em `Yes`. Em `/src/components`, crie o componente `DeleteDialog.tsx` inicialmente com o seguinte código:

```
import {
  AlertDialog,
  AlertDialogAction,
  AlertDialogCancel,
  AlertDialogContent,
  AlertDialogDescription,
  AlertDialogFooter,
  AlertDialogHeader,
  AlertDialogTitle,
  AlertDialogTrigger
} from './ui/alert-dialog'
import {Button} from './ui/button'

export default function DeleteDialog(params) {
  return (
    <AlertDialog>
      <AlertDialogTrigger asChild>
        <Button variant="link">Show Dialog</Button>
      </AlertDialogTrigger>
      <AlertDialogContent>
        <AlertDialogHeader>
          <AlertDialogTitle>Are you absolutely sure?</AlertDialogTitle>
          <AlertDialogDescription>
            This action cannot be undone.
          </AlertDialogDescription>
        </AlertDialogHeader>
        <AlertDialogFooter>
          <AlertDialogCancel>Cancel</AlertDialogCancel>
          <AlertDialogAction>Continue</AlertDialogAction>
        </AlertDialogFooter>
      </AlertDialogContent>
    </AlertDialog>
  )
}
```

```
)
}
```

Então, com o componente pronto, podemos adicioná-lo no `page.tsx`:

```
/src/app/categories/page.tsx

// imports
import DeleteDialog from "../../components/delete-dialog";

export default async function page() {
  // code

  return (
    <div>
      <Table>
        <TableCaption>A list of your categories.</TableCaption>
        <TableHeader>
          <TableRow>
            <TableHead>Name</TableHead>
            <TableHead className="w-[100px] text-center">Actions</TableHead>
          </TableRow>
        </TableHeader>
        <TableBody>
          {categories.map((category) => (
            <TableRow key={category.id}>
              <TableCell>{category.name}</TableCell>
              <TableCell className="text-right">
                <div className="flex flex-row gap-2">
                  <Button asChild variant="link">
                    <Link href={` /categories/edit/${category.id}`}>Edit</Link>
                  </Button>
                  <DeleteDialog></DeleteDialog>
                </div>
              </TableCell>
            </TableRow>
          ))}
        </TableBody>
      </Table>
    </div>
  );
}
```

Veja que agora temos um código mais limpo, onde a célula tem o botão para editar e o `DeleteDialog`. Como o componente `DeleteDialog` é genérico, precisamos passar algumas propriedades para ele, tais como a mensagem que será exibida ao usuário e também o método que deverá ser chamado quando o usuário clicar em `Yes`. Como a ação de remover um item envolve um `id`, neste caso o `id` da categoria, vamos também repassar este atributo.

Sempre que pensamos em propriedades sendo repassadas através dos componentes, podemos

criar uma Interface com tais propriedades. Veja:

```
import {
  AlertDialog,
  AlertDialogAction,
  AlertDialogCancel,
  AlertDialogContent,
  AlertDialogDescription,
  AlertDialogFooter,
  AlertDialogHeader,
  AlertDialogTitle,
  AlertDialogTrigger
} from './ui/alert-dialog'
import { Button } from './ui/button'

interface DeleteDialogProps {
  message: string
  id: string
  actionYes: any
}

export default function DeleteDialog(params: DeleteDialogProps) {
  return (
    code...
  )
}
```

A interface DeleteDialogProps com estas três propriedades agora pode receber estes três parâmetros. Ao criar o DeleteDialog, podemos passar essas propriedades como parâmetro. Veja:

```
<DeleteDialog
  message={`Delete" ${category.name}?`}
  id="{category.id}"
  actionYes="{deleteCategory}"
></DeleteDialog>
```

Agora vamos implementar estas três propriedades no componente DeleteDialog:

```
'use client'

import {
  AlertDialogHeader,
  AlertDialogFooter,
  AlertDialog,
  AlertDialogAction,
  AlertDialogCancel,
  AlertDialogContent,
  AlertDialogDescription,
  AlertDialogTitle,
  AlertDialogTrigger
} from '@components/ui/alert-dialog'
import {Button} from '@components/ui/button'

interface DeleteDialogProps {
  message: string
  id: string
  actionYes: any
}

export default function DeleteDialog(props: DeleteDialogProps) {
  return (
    <AlertDialog>
      <AlertDialogTrigger asChild>
        <Button variant="link">Delete</Button>
      </AlertDialogTrigger>
      <AlertDialogContent>
        <AlertDialogHeader>
          <AlertDialogTitle>Are you absolutely sure?</AlertDialogTitle>
          <AlertDialogDescription>{props.message}</AlertDialogDescription>
        </AlertDialogHeader>
        <AlertDialogFooter>
          <AlertDialogCancel>Cancel</AlertDialogCancel>
          <AlertDialogAction
            className="bg-destructive hover:bg-destructive-hover"
            asChild
          >
            <Button onClick={() => props.actionYes(props.id)}>Yes</Button>
          </AlertDialogAction>
        </AlertDialogFooter>
      </AlertDialogContent>
    </AlertDialog>
  )
}
```

```
)  
}
```

Veja que o componente `DeleteDialog` usa o `use client` para tornar o componente renderizado no cliente. Assim, podemos capturar o evento de `click` do botão.

### 3.7.3. Método para Remover uma Categoria

Podemos agora retornar ao `page.tsx` e configurar o método que irá remover a categoria, conforme o código a seguir:

```
/src/app/categories/page.tsx
```

```
// imports
export default async function page() {
  // code

  async function deleteCategory(id: string) {
    'use server'
    console.log('Delete Category with id', id)
  }

  return (
    <div>
      <Table>
        <TableCaption>A list of your categories.</TableCaption>
        <!-- Table Header -->
        <TableBody>
          {categories.map((category) => (
            <TableRow key={category.id}>
              <TableCell>{category.name}</TableCell>
              <TableCell className="text-right">
                <div className="flex flex-row gap-2">
                  <!-- Edit Button -->
                  <DeleteDialog message={`Delete ${category.name}?`} id={category.id}
                    actionYes={deleteCategory}
                  ></DeleteDialog>
                </div>
              </TableCell>
            </TableRow>
          ))}
        </TableBody>
      </Table>
      <!-- New Category Button -->
    </div>
  )
}
```

Neste código, o `DeleteDialog` irá chamar o método `deleteCategory` quando o usuário clicar em `Yes`. O método, inicialmente, realiza um `console.log` exibindo o `id` da categoria a ser removida. Como o método é chamado de `use server`, o método será executado no servidor, e o `console.log` estará no terminal onde a aplicação foi iniciada, e não no browser.

### 3.7.4. Implementando o Método para Remover uma Categoria

O método `deleteCategory` pode ser inicialmente implementado da seguinte forma:



```
async function deleteCategory(id: string) {  
  'use server'  
  console.log('Delete Category with id', id)  
  
  const prisma = new PrismaClient()  
  await prisma.category.delete({where: {id}})  
  
  redirect('/categories')  
}
```

Além do `console.log`, usamos o `PrismaClient` para remover o item e redirecionamos para `/categories` que é a mesma página em si, mas irá realizar o refresh da página.

### 3.7.5. Implementando Erros

O método `deleteCategory` pode verificar alguns erros como uma categoria não encontrada ou uma categoria que possui productos e obviamente ela não pode ser excluída, por exemplo:

```
async function deleteCategory(id: string) {  
  'use server'  
  console.log('Delete Category with id', id)  
  
  const prisma = new PrismaClient()  
  
  const category = await prisma.category.findUnique({where: {id}})  
  
  if (!category) {  
    throw new Error('Category not found')  
  }  
  
  const hasProducts = await prisma.product.findFirst({  
    where: {  
      categoryId: id  
    }  
  })  
  
  if (hasProducts) {  
    throw new Error('Category has products, can not be deleted')  
  }  
  
  await prisma.category.delete({where: {id}})
```

```
    redirect('/categories')  
  }
```

## 3.8. Conclusão

Neste capítulo começamos a abordar os componentes do shadcn e a forma em como adicionar, editar e remover dados de uma tabela. Utilizamos a forma mais básica para adicionar e remover dados, que talvez não seja a melhor forma, mas fizemos isso para que possamos ter um primeiro contato com o shadcn.

A organização final dos arquivos ficou a seguinte:

```
├── src  
│   ├── app  
│   │   ├── categories  
│   │   │   ├── edit  
│   │   │   │   ├── [id]  
│   │   │   │   │   ├── form.tsx  
│   │   │   │   │   └── page.tsx  
│   │   │   ├── error.tsx  
│   │   │   ├── layout.tsx  
│   │   │   ├── loading.tsx  
│   │   │   ├── new  
│   │   │   │   └── page.tsx  
│   │   │   ├── page.tsx  
│   │   │   └── Submit.tsx  
│   │   ├── favicon.ico  
│   │   ├── globals.css  
│   │   ├── layout.tsx  
│   │   ├── page.tsx  
│   │   ├── components  
│   │   │   ├── delete-dialog.tsx  
│   │   │   ├── Header.tsx  
│   │   │   ├── ui  
│   │   │   │   ├── alert-dialog.tsx  
│   │   │   │   ├── button.tsx  
│   │   │   │   ├── card.tsx  
│   │   │   │   └── table.tsx  
│   │   └── lib  
│   │       └── utils.ts
```

No próximo capítulo, vamos criar a tela de Produtos, e ao invés de repetir as técnicas aprendidas, vamos utilizar novas formas de criar, editar e remover dados. Por exemplo, vamos usar `React Hook Forms` juntamente com o poderoso componente `Form` do `shadcn`. Também utilizaremos `zod` para validação de dados e implementaremos o recurso de `actions` que facilita o acesso ao servidor.

Aguarde a próxima atualização do livro !!