

APRENDA ANGULAR CRIANDO UM SIMPLES SISTEMA DE VENDAS



ANGULAR 17

DO ZERO

POR DANIEL SCHMITZ
2024

Angular 17 do zero

Crie uma simples app de vendas com o Angular.
Agora com o Angular 17!

Daniel Schmitz

This book is for sale at <http://leanpub.com/livro-angular>

This version was published on 2024-04-24



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2024 Daniel Schmitz

Contents

1. Introdução	1
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 Backend (servidor)	2
2. Olá Angular	8
2.1. Instalação	8
2.2. Criar um Workspace e Aplicação Inicial	8
2.3. As Ferramentas de Desenvolvimento do Angular	10
2.4. O Angular Material	11
2.5. É Hora de Fazer o Commit do Projeto (opcional)	12
2.6. Vamos Adicionar um Repositório Remoto (opcional)	13
2.7. Vamos Abrir o Projeto no Visual Studio Code	18
2.8. E o Módulo?	21
3. O Início	22
3.1. Vamos Limpar!	22
3.2. Os Componentes do Material	24
3.3. Adicionando Esquemáticos	28
3.4. Adicionando uma Navegação em sua Aplicação	29
3.5. Adicionando o Componente Home ao App	32
3.6. Alterando o Home	34
3.7. Componentes	36
3.8. Typescript e Interfaces	39
3.9. Componente de Categorias	42
3.10. Rotas	44

CONTENTS

3.11. Criando um Painel	48
4. Categorias	50
4.1. O Card do Angular Material	50
4.2. Adicionando um Estilo Css Global	53
4.3. Mais Estilos de Margem/Espaçamento (opcional)	56
4.4. Obtendo Dados da API de Categoria	59
4.5. Configurando HttpClient	60
4.6. Serviços	61
4.7. O Serviço de Categoria	61
4.8. Primeira Versão do Método GetAll()	62
4.9. Variáveis de Ambiente	63
4.10. Definindo o Tipo de Retorno da API	65
4.11. Versão Final do Método GetAll()	66
4.12. Usando o MatTable para Exibir Categorias	66
4.13. Adicionando a Coluna Descrição	71
4.14. Nova Categoria	73
4.15. Criar um Formulário de Categoria	73
4.16. Criando um Formulário de Categoria	76
4.17. Criando um Formulário Reativo	77
4.18. Adicionando o Campo de Descrição	80
4.19. Criando Formulários Responsivos: o Layout CSS FlexBox	81
4.20. Validação	87
4.21. Configurando Mensagens de Erro <i>{/examples/}</i>	89
4.22. Enviar Formulário <i>{/examples/}</i>	90
4.23. Revisando Alguns Padrões do Angular	92
4.24. Controlando a Visibilidade do Formulário	93
4.25. Criando um Botão de Voltar no Formulário	95
4.26. Vinculação de Evento	96
4.27. Passando Dados do Formulário Através de Eventos	98
4.28. Conversão de Tipo	101
4.29. Salvando a Categoria	102
4.30. Editando a Categoria	105
4.31. Corrigir um Pequeno Bug	109
4.32. Excluindo uma Categoria	110
4.33. O Que Aprendemos Neste Capítulo	112
4.34. Diferenças do Angular 14..15..16	113

CONTENTS

5. Categorias de Refatoração	114
5.1. Adicionando Carregamento Durante a Solicitação ao Servidor	114
5.2. Como Ver o Carregamento Funcionando	119
5.3. Pular Testes e Criação de Arquivo Css no Arquivo de Configuração Angular.json	120
5.4. Hora de Fazer o Deploy! (opcional)	121
5.5. O “Módulo Material”	122
6. Fornecedor	126
6.1. Criar os Componentes dos Fornecedores	126
6.2. Usando Rotas e Sub Rotas	129
6.3. DTO do Fornecedor	132
6.4. Serviço de Fornecedores	134
6.5. Listando Fornecedores	136
6.6. Criando um Novo Componente	140
6.7. Exibindo uma Mensagem Se @for Estiver Vazio	143
6.8. Configurando Rotas	143
6.9. Mostrando um Fornecedor	145
6.10. Editar um Fornecedor {/examples/}	149
6.11. Formulário de Fornecedor	151
6.12. Adicionando o Formulário no SuppliersEditComponent	156
6.13. Deletar Fornecedor	158
6.14. Novo Fornecedor	160
6.15. Conclusão	162
7. Produtos	164
7.1. Arquivos Iniciais	164
7.2. O Serviço de Produtos	165
7.3. Listagem de Produtos	168
7.4. Adicionar Produto ao Carrinho	171
7.5. O Botão “Adicionar ao Carrinho”	174
7.6. Criando o Ícone do Carrinho	176
7.7. Adicionar uma Página de Checkout	180
8. Carregamento Dinâmico de Arquivos e Componentes	184
8.1. Visualizações Diferíveis	184
8.1.1. Exemplo	184
8.1.2. Extraiendo o Componente	184
8.1.3. Usando @defer	187
8.1.4. Usando @viewport, @placeholder e @loading	188

8.2. Estratégias para tornar a aplicação ainda menor	190
9. Atualizações no Futuro	193

1. Introdução

O principal objetivo deste livro é ensinar o framework *Angular* criando um sistema de vendas 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: <https://github.com/danielschmitz/my-sales-app-angular/issues>

1.3. Código Fonte

Todo o código fonte deste livro está no repositório do github: <https://github.com/danielschmitz/my-sales-app-angular>

1.4. Instalação

Angular é um framework frontend 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 com o desenvolvimento em Javascript em geral.

- **ESLint**: Integra o ESLint no VS Code. O ESLint encontra e corrige problemas no seu código JavaScript.
- **TSLint**: Integra o TSLint no VS Code. O TSLint encontra e corrige problemas no seu código TypeScript.
- **Material Icon Theme**: Obtenha os ícones do Material Design no seu VS Code.
- **Opcional**: Dracula Oficial ou Nord - Um tema para o Visual Studio Code.
- **Error Lens**: Exibe o erro, se houver, no final da linha.
- **Git Lens**: Exibe informações do git diretamente no seu código.
- **Angular Language Service**: Esta extensão oferece uma experiência de edição rica para templates Angular, tanto inline quanto templates externos.
- **Angular Files**: Esta extensão permite criar rapidamente templates de arquivos angular 2 no projeto VS Code. Nos primeiros capítulos do livro, estaremos usando o comando ng para gerar arquivos, mas com esta extensão você pode automatizar esse processo.

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

1.5. O Backend (servidor)

Como já mencionado, Angular é um framework frontend. Ele não tem a responsabilidade de realizar tarefas relacionadas ao servidor backend, como persistir dados no banco de dados.

Possivelmente você tem um servidor backend de preferência, como Java, PHP, Node etc. Neste livro, estaremos usando um servidor falso, que instalaremos a seguir.

Após instalar o Node e o Git, abra o programa `git bash`. Você verá uma tela semelhante a esta:

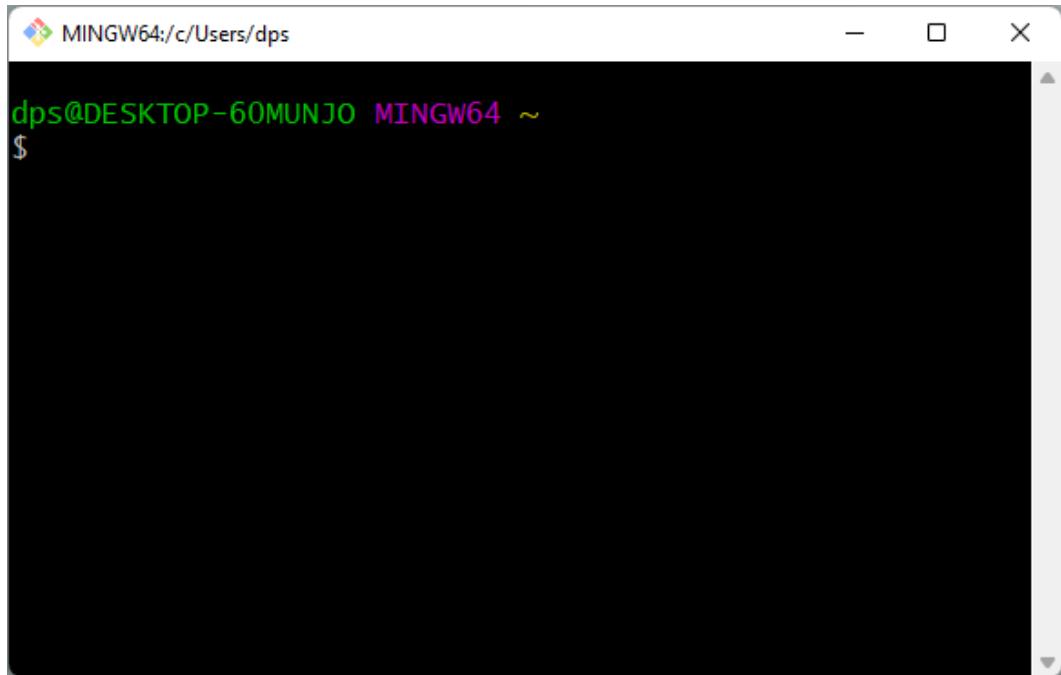


Figure 1. O Git Bash no seu diretório home

Se você está usando Linux ou Mac, assumo que está acostumado a usar o *Terminal*. Neste caso, use o *terminal* de sua escolha.

Para baixar o servidor backend, clone o seguinte projeto com este comando:

```
$ git clone https://github.com/danielschmitz/fake-server.git
```

Após o download, entre no diretório fake-server:

```
$ cd fake-server
```

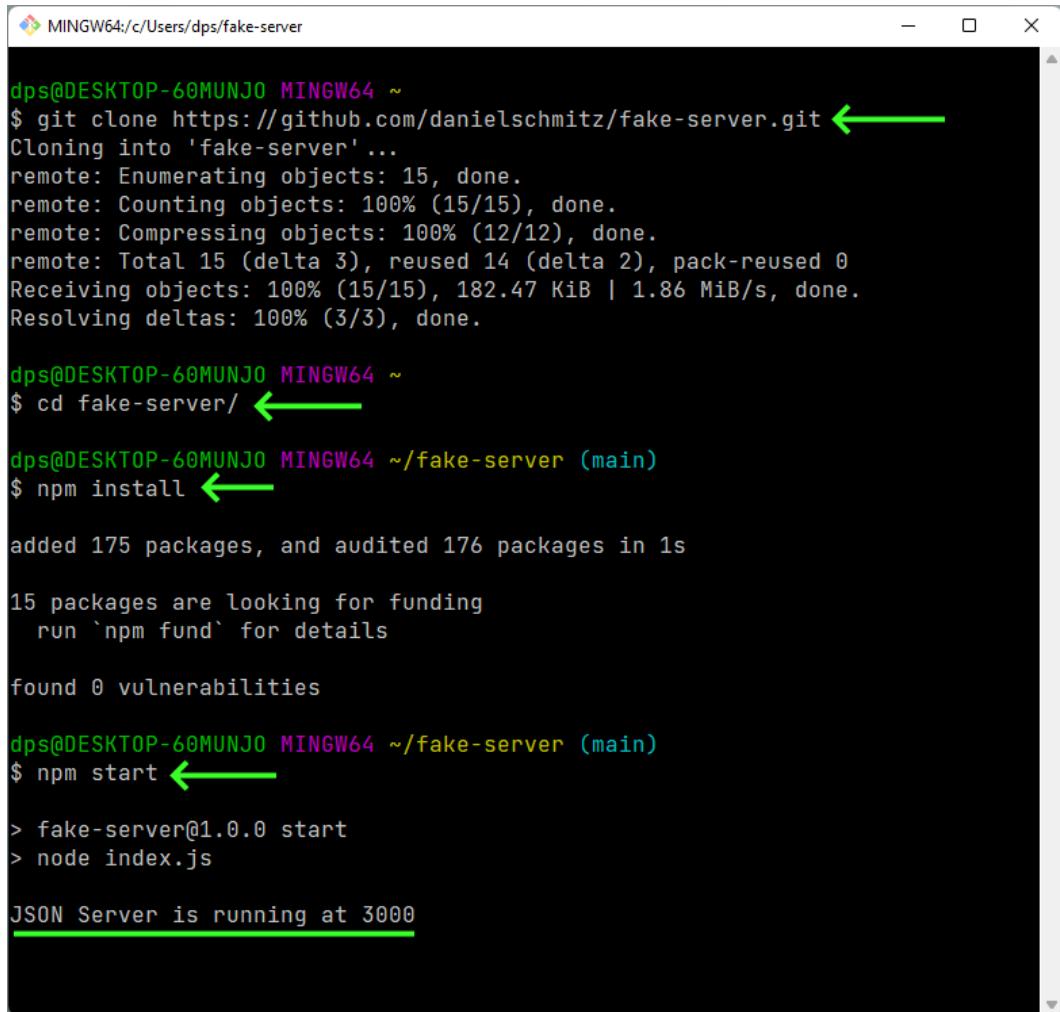
E execute os seguintes comandos:

```
$ npm install  
$ npm start
```

A seguinte mensagem aparecerá:

```
JSON Server is running at 3000
```

A imagem a seguir ilustra todo o processo:



The screenshot shows a terminal window titled 'MINGW64:/c/Users/dps/fake-server'. The terminal output is as follows:

```
dps@DESKTOP-60MUNJO MINGW64 ~
$ git clone https://github.com/danielschmitz/fake-server.git ←
Cloning into 'fake-server'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 15 (delta 3), reused 14 (delta 2), pack-reused 0
Receiving objects: 100% (15/15), 182.47 KiB | 1.86 MiB/s, done.
Resolving deltas: 100% (3/3), done.

dps@DESKTOP-60MUNJO MINGW64 ~
$ cd fake-server/ ←
dps@DESKTOP-60MUNJO MINGW64 ~/fake-server (main)
$ npm install ←

added 175 packages, and audited 176 packages in 1s

15 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

dps@DESKTOP-60MUNJO MINGW64 ~/fake-server (main)
$ npm start ←

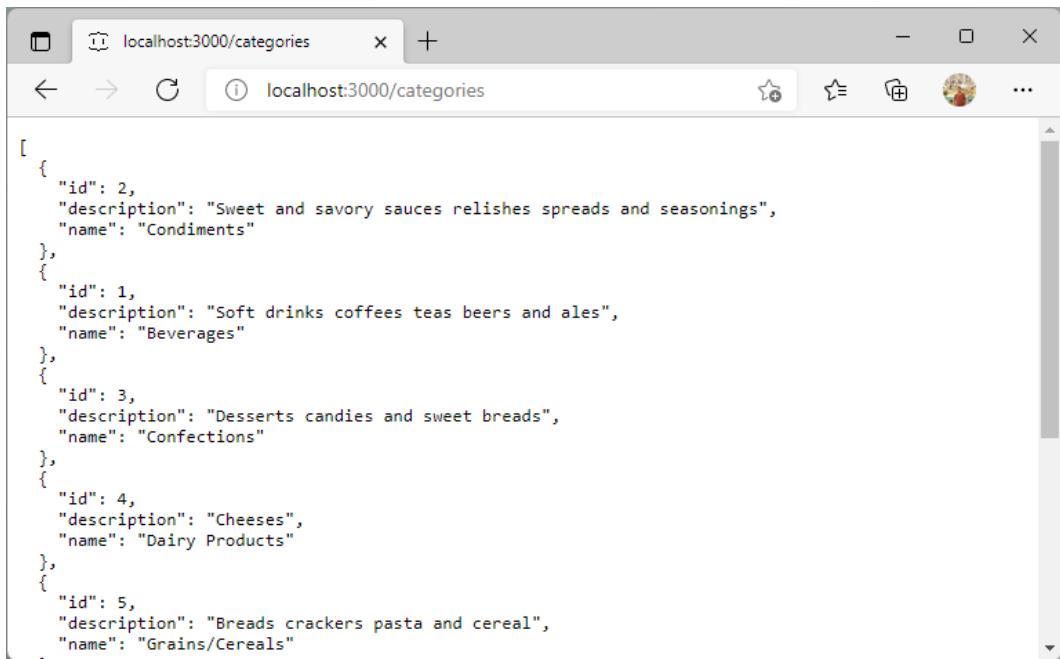
> fake-server@1.0.0 start
> node index.js

JSON Server is running at 3000
```

Three green arrows point to the command '\$ git clone https://github.com/danielschmitz/fake-server.git', the directory change '\$ cd fake-server/' and the command '\$ npm start'.

Figure 2. Todo o processo para instalar o servidor backend

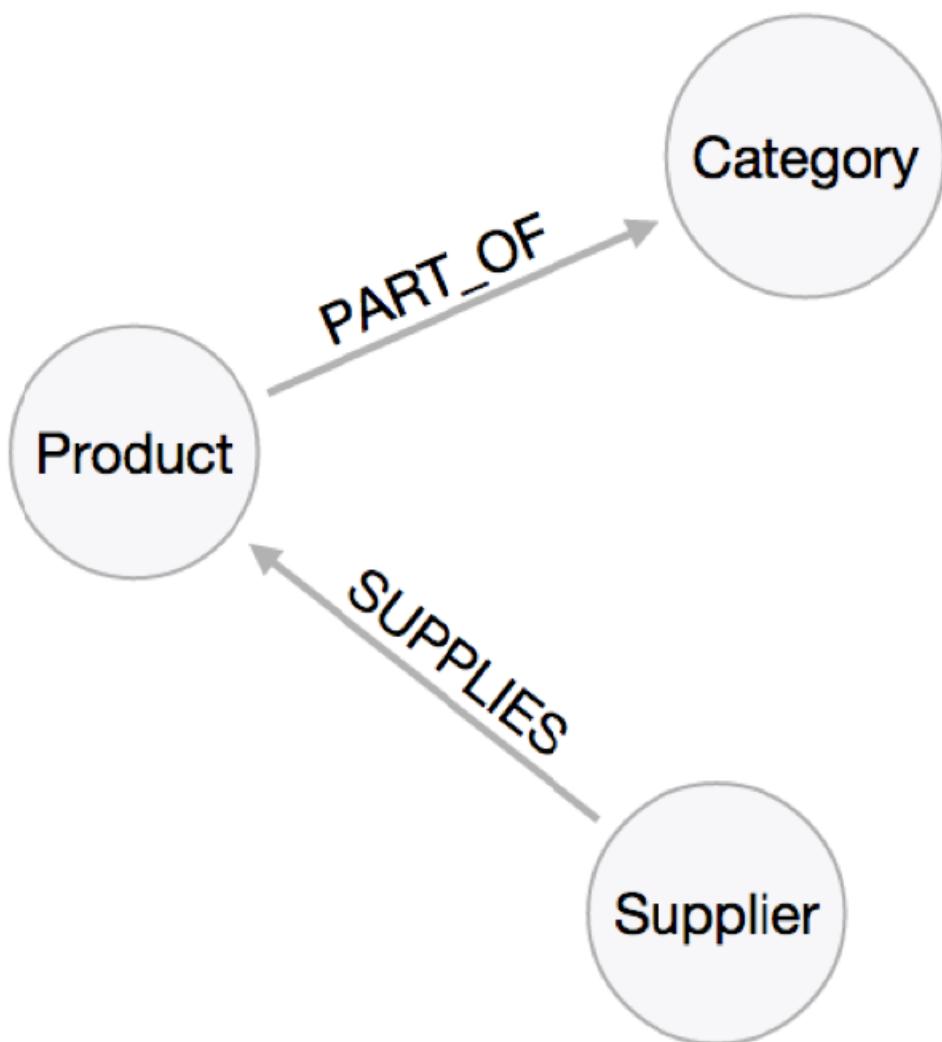
Com o backend em execução, abra seu navegador e acesse a seguinte URL: <http://localhost:3000/categories>. Você verá algo semelhante à imagem a seguir:

A screenshot of a Microsoft Edge browser window. The address bar shows 'localhost:3000/categories'. The main content area displays a JSON array of five categories. Each category object has an 'id', 'description', and 'name' field.

```
[{"id": 2, "description": "Sweet and savory sauces relishes spreads and seasonings", "name": "Condiments"}, {"id": 1, "description": "Soft drinks coffees teas beers and ales", "name": "Beverages"}, {"id": 3, "description": "Desserts candies and sweet breads", "name": "Confections"}, {"id": 4, "description": "Cheeses", "name": "Dairy Products"}, {"id": 5, "description": "Breads crackers pasta and cereal", "name": "Grains/Cereals"}]
```

Figure 3. O backend rodando em localhost:3000

O fake-server possui a seguinte estrutura:



2. Olá Angular

Neste capítulo, vamos instalar o Angular e cobrir alguns conceitos iniciais do framework.

2.1. Instalação

Existem várias maneiras de instalar o Angular, e vamos usar o *Node* para instalar o *Angular Cli*. O *Node* possui um instalador de pacotes chamado *NPM* e a partir dele vamos instalar o *Cli*. Abra um *Terminal* (Git Bash) e execute o seguinte comando:

```
$ npm install -g @angular/cli
```

[here](#)

Figure 4. Instalando o Angular Cli

A opção *-g* do comando `npm install` instala o *Angular Cli* globalmente. Com o *Cli* instalado, agora você pode criar o projeto.

2.2. Criar um Workspace e Aplicação Inicial

O Angular trabalha com o conceito de um workspace. Um workspace pode conter vários projetos Angular. Por agora, vamos ter apenas um workspace e um projeto.

Para criar um workspace e um projeto, execute o seguinte comando:

```
$ ng new my-sales-app
```

O comando `ng` é o *Cli* do Angular, instalado no tópico anterior.

Para um novo projeto, use as opções padrão se perguntado

```
dps@DESKTOP-60MUNJ0 MINGW64 ~
$ ng new my-sales-app
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
CREATE my-sales-app/angular.json (2717 bytes)
CREATE my-sales-app/package.json (1083 bytes)
CREATE my-sales-app/README.md (1092 bytes)
CREATE my-sales-app/tsconfig.json (936 bytes)
CREATE my-sales-app/.editorconfig (298 bytes)
CREATE my-sales-app/.gitignore (598 bytes)
CREATE my-sales-app/tsconfig.app.json (277 bytes)
CREATE my-sales-app/tsconfig.spec.json (287 bytes)
CREATE my-sales-app/.vscode/extensions.json (134 bytes)
CREATE my-sales-app/.vscode/launch.json (490 bytes)
CREATE my-sales-app/.vscode/tasks.json (988 bytes)
CREATE my-sales-app/src/main.ts (256 bytes)
CREATE my-sales-app/src/favicon.ico (15086 bytes)
CREATE my-sales-app/src/index.html (309 bytes)
CREATE my-sales-app/src/styles.css (81 bytes)
CREATE my-sales-app/src/app/app.component.html (20253 bytes)
CREATE my-sales-app/src/app/app.component.spec.ts (963 bytes)
CREATE my-sales-app/src/app/app.component.ts (384 bytes)
CREATE my-sales-app/src/app/app.component.css (0 bytes)
CREATE my-sales-app/src/app/app.config.ts (235 bytes)
CREATE my-sales-app/src/app/app.routes.ts (80 bytes)
CREATE my-sales-app/src/assets/.gitkeep (0 bytes)
✓ Packages installed successfully.
  Successfully initialized git.
```

Figure 5. Criando o projeto my-sales-app

Quando executamos este comando, o diretório `my-sales-app` é criado e alguns arquivos são incluídos no projeto. Após alguns minutos, o projeto está pronto. Para executá-lo, entre no diretório `my-sales-app` e execute o seguinte comando:

```
$ ng serve --open
```

O *Angular Cli* é usado novamente, agora para rodar o projeto e, através do parâmetro `--open`, abrir o navegador.

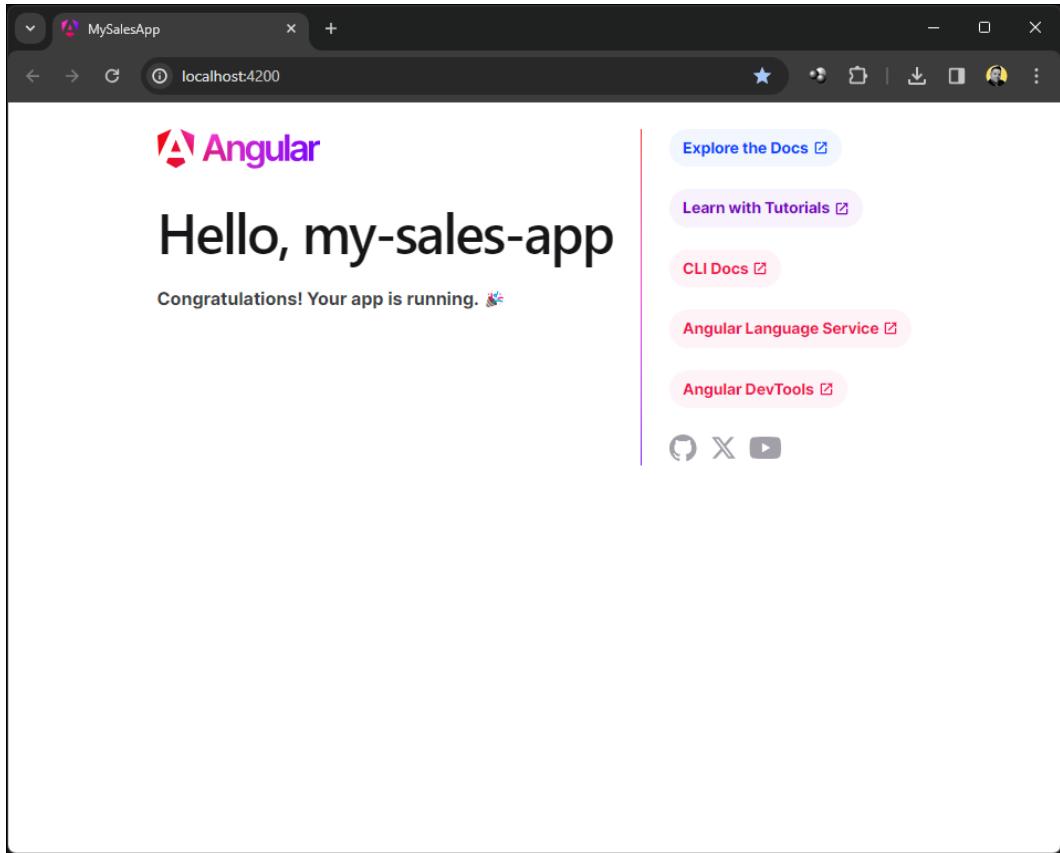


Figure 6. Executando o projeto my-sales-app

2.3. As Ferramentas de Desenvolvimento do Angular

O projeto possui muitas informações iniciais sobre o Angular. Uma delas são as *Ferramentas de Desenvolvimento do Angular*:

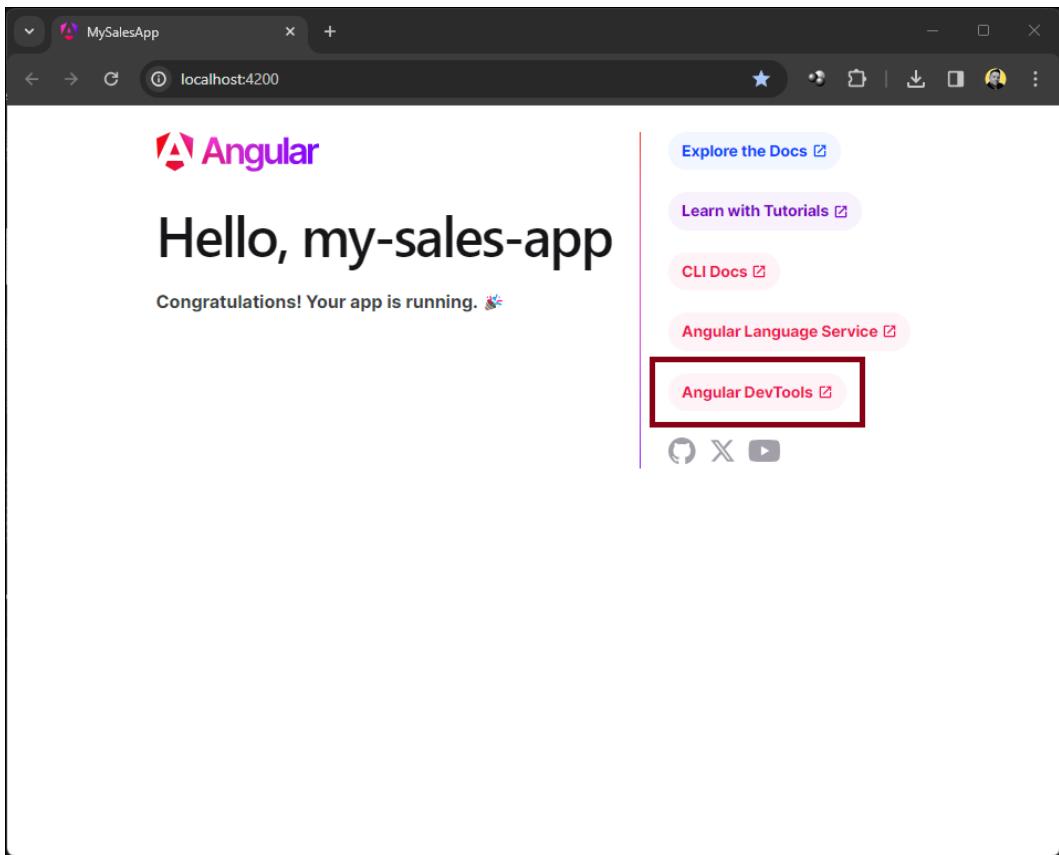


Figure 7. Ferramentas de Desenvolvimento do Angular

Angular DevTools é uma extensão do Chrome que fornece capacidades de depuração e perfilamento para aplicações Angular. Instale a extensão e volte ao projeto, recarregue a página e abra as Ferramentas de Desenvolvimento do Chrome (F12). Vá até a aba *Angular* para obter algumas informações importantes sobre a página. No projeto inicial, não temos muita coisa, mas à medida que o projeto cresce, as *Ferramentas de Desenvolvimento do Angular* serão muito úteis.

2.4. O Angular Material

O *Angular Material* é uma biblioteca de componentes que segue as diretrizes do **Material Design** do Google. Vamos usar essa biblioteca extensivamente, então nosso próximo passo

no projeto é instalar o Angular Material.

Vamos adicionar o Angular Material ao nosso projeto. Pare a execução do projeto com `Ctrl+C` e execute o seguinte comando:

```
$ ng add @angular/material @angular/cdk
```

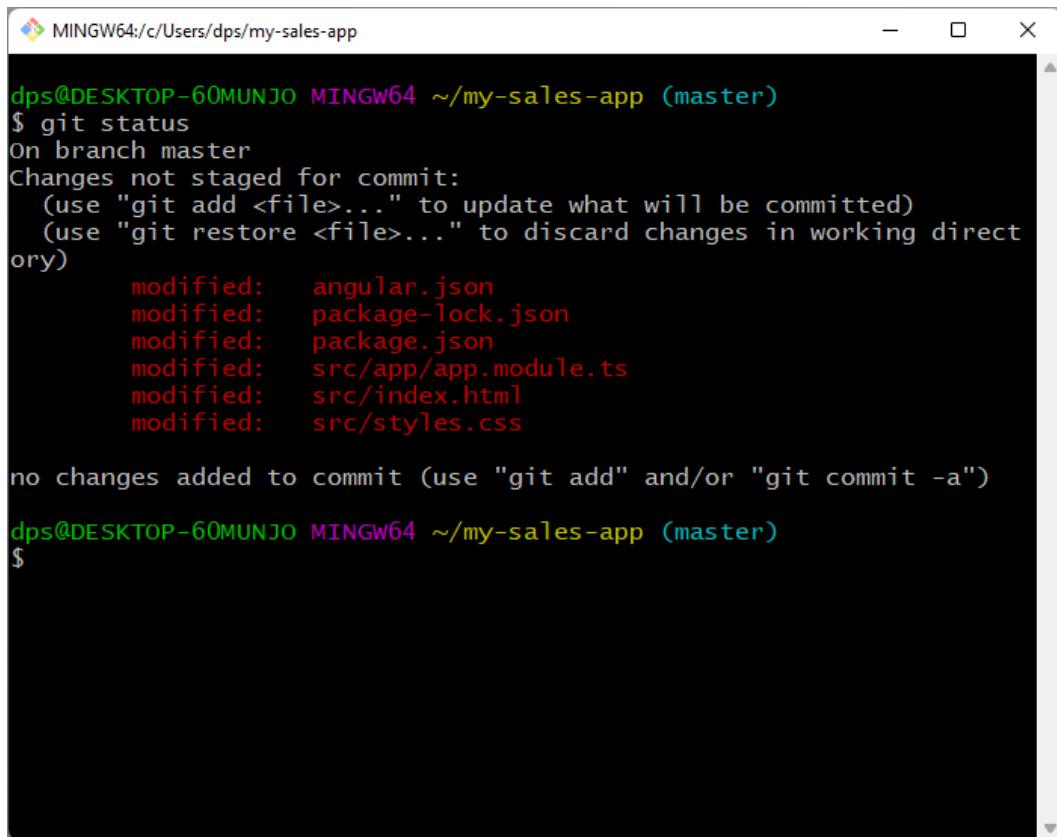
```
dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (master)
$ ng add @angular/material @angular/cdk
i Using package manager: npm
✓ Found compatible package version: @angular/material@17.0.5.
✓ Package information loaded.

The package @angular/material@17.0.5 will be installed and executed.
Would you like to proceed? Yes
✓ Packages successfully installed.
? Choose a prebuilt theme name, or "custom" for a custom theme: Indigo/Pink
? Set up global Angular Material typography styles? No
? Include the Angular animations module? Include and enable animations
UPDATE package.json (1111 bytes)
✓ Packages installed successfully.
UPDATE src/app/app.config.ts (331 bytes)
UPDATE angular.json (2853 bytes)
UPDATE src/index.html (509 bytes)
UPDATE src/styles.css (182 bytes)
```

Prossiga com a instalação escolhendo as opções padrão. Após a instalação, execute o projeto novamente com `ng serve` e recarregue a página.

2.5. É Hora de Fazer o Commit do Projeto (opcional)

Após instalar o *Angular Material*, podemos usar o comando `git status` para revisar as mudanças que foram feitas.

A screenshot of a terminal window titled "MINGW64:/c/Users/dps/my-sales-app". The window shows the output of a "git status" command. It indicates that the user is on the "master" branch and has changes not staged for commit. The modified files listed are angular.json, package-lock.json, package.json, src/app/app.module.ts, src/index.html, and src/styles.css. A message at the bottom says "no changes added to commit (use "git add" and/or "git commit -a")".

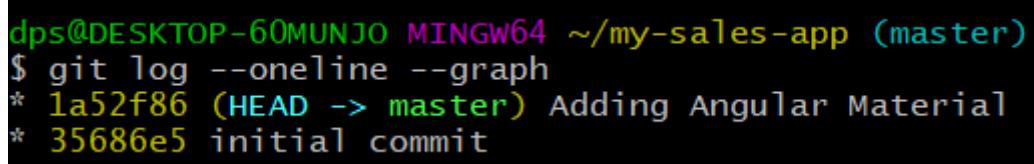
```
dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)

        modified:   angular.json
        modified:   package-lock.json
        modified:   package.json
        modified:   src/app/app.module.ts
        modified:   src/index.html
        modified:   src/styles.css

no changes added to commit (use "git add" and/or "git commit -a")

dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (master)
$
```

Para fazer o commit dessas mudanças, use o comando: `git commit -am "Adicionando Angular Material"`. Podemos ver os commits com o comando: `git log --oneline`.

A screenshot of a terminal window titled "MINGW64:/c/Users/dps/my-sales-app (master)". The window shows the output of a "git log --oneline" command. It lists two commits: one for adding Angular Material and another for the initial commit.

```
dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (master)
$ git log --oneline --graph
* 1a52f86 (HEAD -> master) Adding Angular Material
* 35686e5 initial commit
```

Ok, não vamos ficar o tempo todo realizando operações do Git pelo *Terminal*, muito em breve estaremos usando o Visual Studio Code.

2.6. Vamos Adicionar um Repositório Remoto (opcional)

Qualquer projeto de programação deve estar em um repositório remoto, e acredito que como programador você concorda com isso. Vamos configurar este projeto no Github, para que sempre que houver mudanças no projeto possamos sincronizá-lo.

Você precisa de uma conta no [site do Github](#). Se você não tem uma, é hora de criar uma. Clique no botão “New repository” e no campo “Repository name”, digite `my-sales-app`.

Importante. NÃO marque *Initialize this repository with:*

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Repository template
Start your repository with a template repository's contents.

No template ▾

Owner *  danielschmitz / **Repository name *** my-sales-aapp ✓

Great repository names are short and descriptive. [my-sales-aapp](#) is available. Inspiration? How about [ideal-goggles](#)?

Description (optional)

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore
Choose which files not to track from a list of templates. [Learn more.](#)

Choose a license
A license tells others what they can and can't do with your code. [Learn more.](#)

DON'T CHECK THIS !

Create repository

Figure 8. Criação de Projeto no Github

Após clicar no botão **Create Repository**, encontre os comandos que estão sob “...or push an existing repository from the command line” e execute-os no seu terminal.

Importante: Não execute meus comandos mostrados na imagem, mas sim os seus comandos.

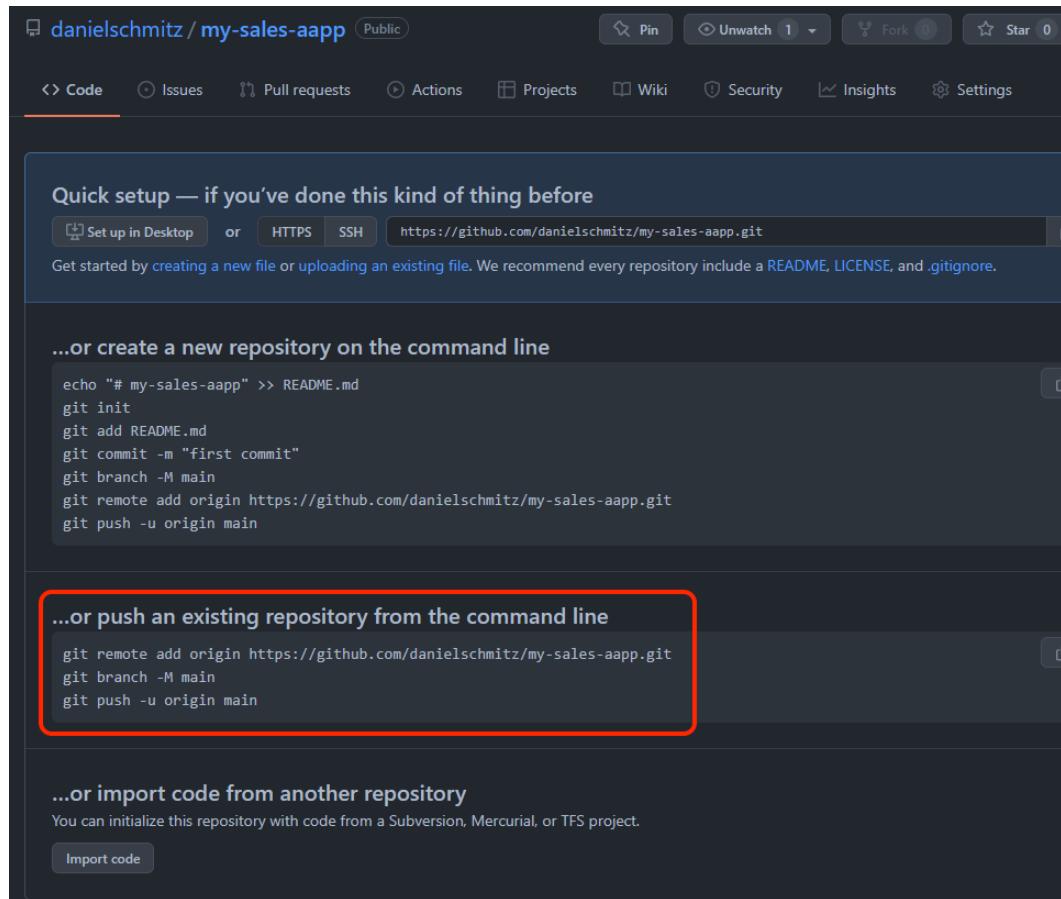
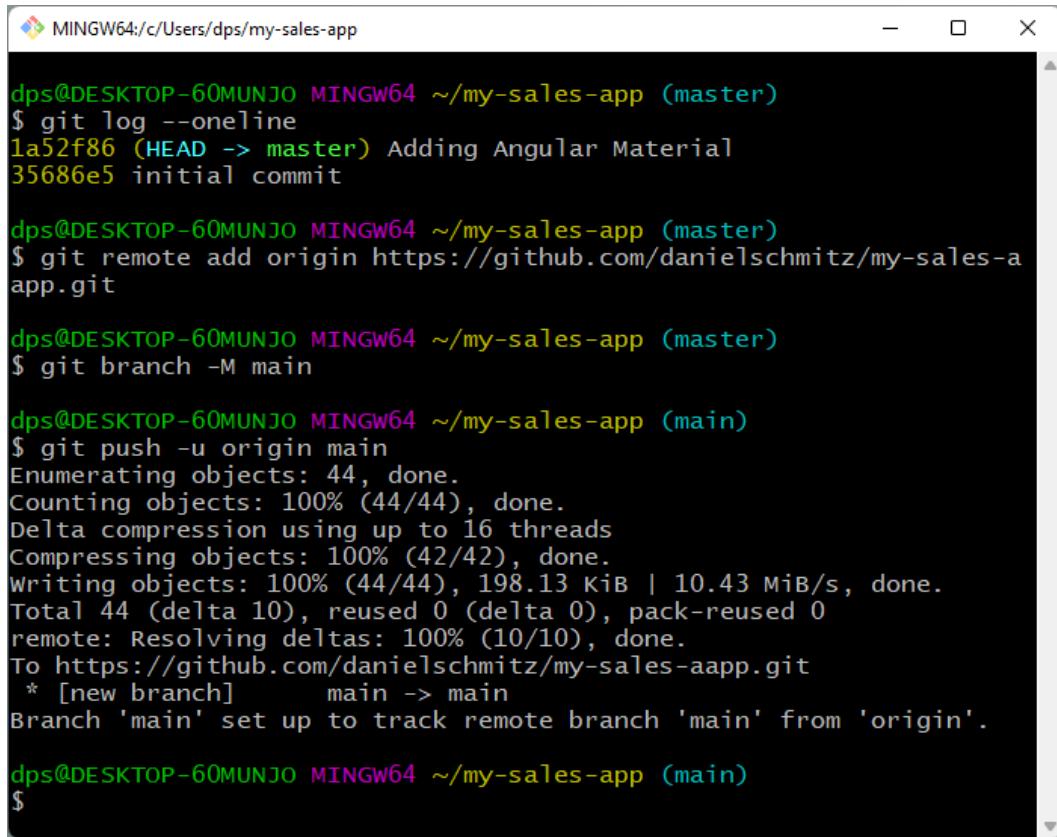


Figure 9. 3 comandos para executar no seu terminal



```
dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (master)
$ git log --oneline
1a52f86 (HEAD -> master) Adding Angular Material
35686e5 initial commit

dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (master)
$ git remote add origin https://github.com/danielschmitz/my-sales-a
app.git

dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (master)
$ git branch -M main

dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (main)
$ git push -u origin main
Enumerating objects: 44, done.
Counting objects: 100% (44/44), done.
Delta compression using up to 16 threads
Compressing objects: 100% (42/42), done.
Writing objects: 100% (44/44), 198.13 KiB | 10.43 MiB/s, done.
Total 44 (delta 10), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (10/10), done.
To https://github.com/danielschmitz/my-sales-aapp.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.

dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (main)
$
```

Após executar esses comandos, recarregue a página do Github para ver isso:

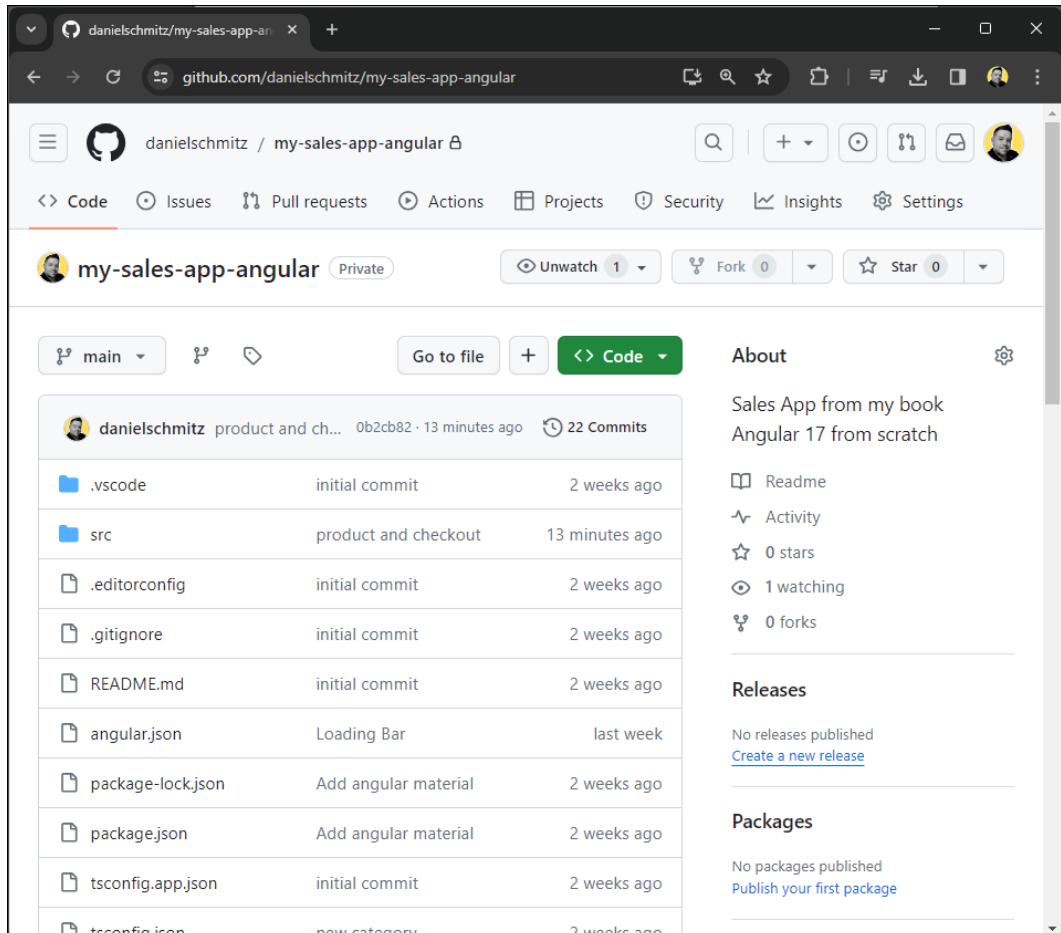
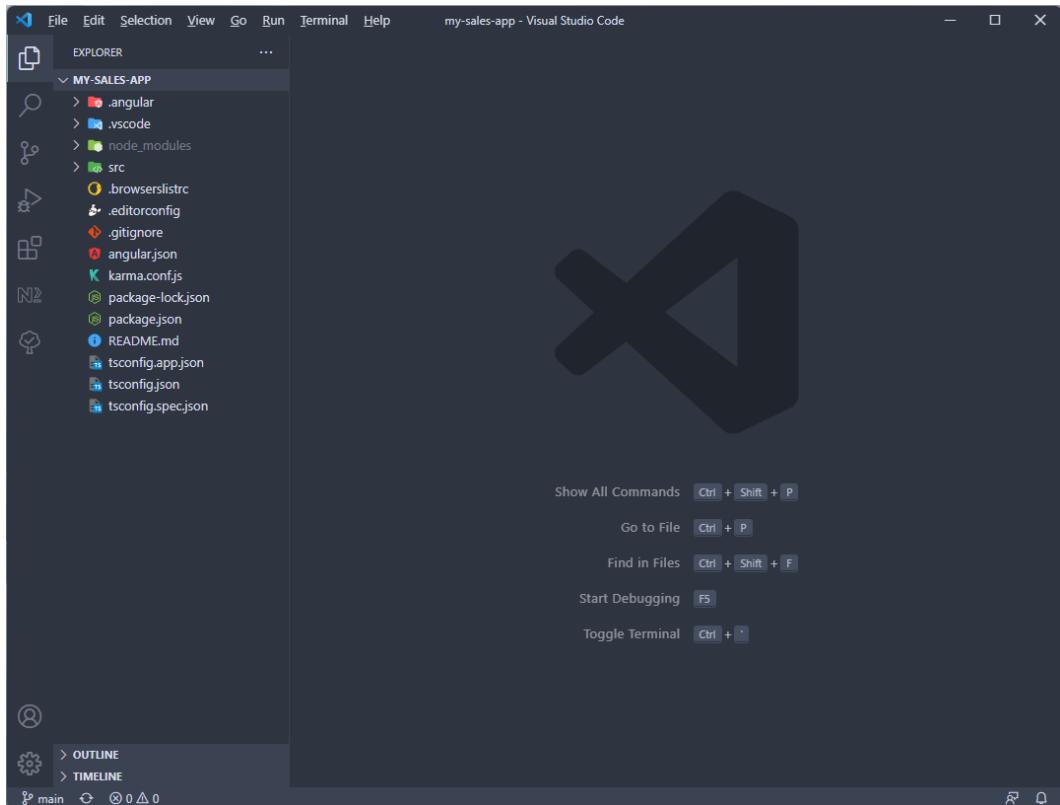


Figure 10. Os arquivos do projeto

2.7. Vamos Abrir o Projeto no Visual Studio Code

Agora vamos abrir o projeto no Visual Studio Code e olhar alguns arquivos.



Existem vários arquivos de configuração, e seria tedioso explicá-los um por um. Vamos primeiro olhar para o diretório `src/app`, que é onde estaremos criando toda a nossa aplicação.

The screenshot shows a file explorer window with the following directory structure:

- MY-SALES-APP
 - .angular
 - .vscode
 - src
 - app
 - # app.component.css
 - app.component.html
 - TS app.component.spec.ts
 - TS app.component.ts
 - TS app.config.ts
 - TS app.routes.ts
 - assets
 - favicon.ico
 - index.html
 - TS main.ts
 - # styles.css
 - .editorconfig
 - .gitignore
 - {} angular.json
 - {} package-lock.json
 - {} package.json
 - README.md
 - {} tsconfig.app.json
 - TS tsconfig.json
 - {} tsconfig.spec.json

Neste diretório, temos:

- **app.component.css** Estilos do componente App
- **app.component.html** O template HTML com muito mais funcionalidade
- **app.component.spec.ts** Testes unitários usando o framework de teste de javascript Jasmine através do executor de testes Karma

- **app.component.ts** A lógica do componente, usando a linguagem Typescript.
- **app.routes.ts** As rotas da nossa aplicação
- **app.config.ts** Um arquivo de configuração que adiciona alguma funcionalidade ao seu projeto. Por agora, apenas adicione o roteador.

2.8. E o Módulo?

Se você tem acompanhado o Angular desde versões anteriores, pode estar se perguntando sobre o arquivo `app.module.ts`. A partir da versão 17, módulos são opcionais para a sua aplicação, e aplicações Angular sem módulos são chamadas de ‘standalone’. Se o uso de módulos é importante para suas aplicações, você deve usar o atributo `--no-standalone` ao criar o projeto.

3. O Início

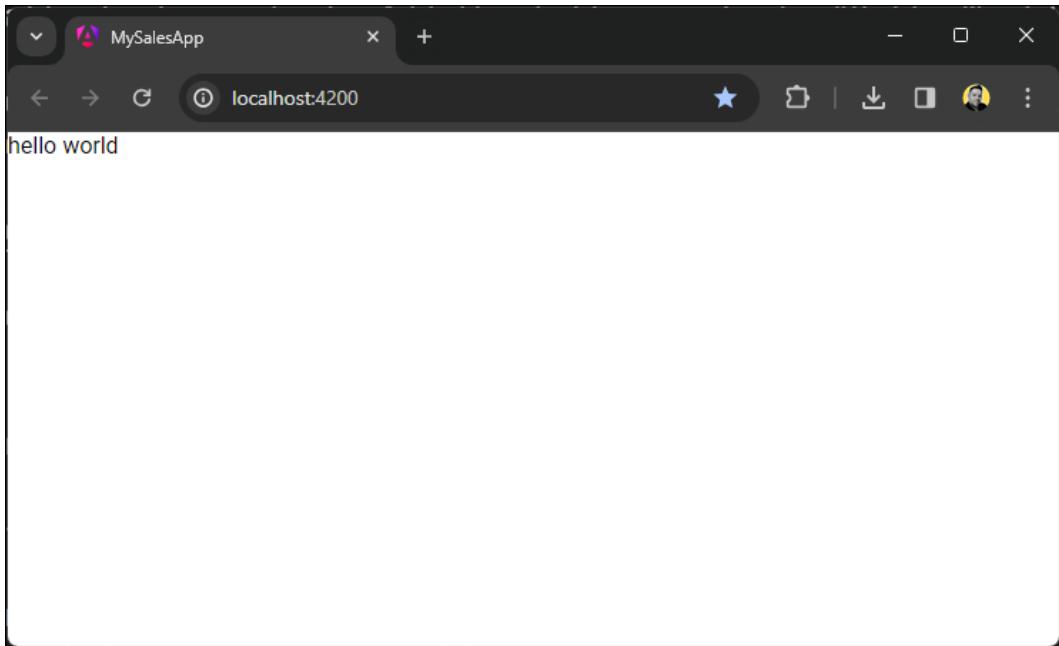
3.1. Vamos Limpar!

O projeto criado vem com a página padrão explicando um pouco sobre o Angular. Vamos agora remover este código para que possamos começar a criar o projeto real. Primeiro, abra `app.component.html` e remova todo o seu conteúdo e adicione:

```
<div>hello world</div>
<router-outlet></router-outlet>
```

Criamos uma div com um simples “hello world” e também adicionamos o `<router-outlet>` que será usado quando implementarmos algumas rotas em nossa aplicação. Se você não sabe sobre rotas, não se preocupe, vamos cobri-las em breve.

Se o projeto estiver em execução(porta 4200), ele é automaticamente recompilado e atualizado. O que temos agora é algo como a seguinte imagem:



Também vamos abrir o arquivo `app.component.ts` e refatorá-lo para:

```
import {Component} from '@angular/core'
import {CommonModule} from '@angular/common'
import {RouterOutlet} from '@angular/router'

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {}
```

Removemos a variável `title` que estava sendo usada no Template.

Uma nova funcionalidade na versão 17 é que, em vez de termos um módulo padrão, temos uma aplicação standalone, então é necessário importar o `CommonModule` e o `RouterOutlet` diretamente no componente.

3.2. Os Componentes do Material

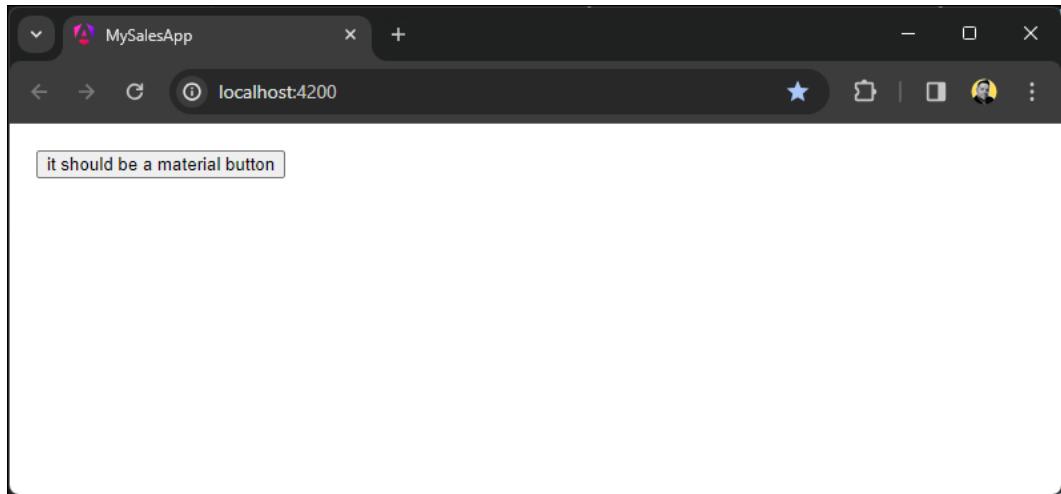
Como já adicionamos o Material à nossa aplicação usando o comando `ng add`, seria natural pensar que já é possível usar um componente como o “Botão Material”. De fato, quando visitamos a documentação do Botão Material neste [link](#), obtemos o seguinte exemplo:

```
<button id="basic" type="button" mat-button>Botão básico</button>
```

O que determina que este botão é um “botão material” é o parâmetro `mat-button`. Por exemplo, ao incluir este código em nosso `app.component.html`, temos:

```
<div style="padding: 20px;">
  <button mat-button>it should be a material button</button>
</div>
<router-outlet></router-outlet>
```

O que temos aqui é o seguinte resultado:



Note que o Botão Material não foi carregado corretamente. Isso ocorre porque precisamos importar o “MaterialButtonModule” nos imports do componente. Veja:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

...

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, MatButton],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

Após importar o MatButtonModule, o componente tem o seguinte código:

```
import {Component} from '@angular/core'
import {CommonModule} from '@angular/common'
import {RouterOutlet} from '@angular/router'
import {MatButtonModule} from '@angular/material/button'

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, MatButtonModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {}
```

Todos os componentes Material têm essa característica, e precisamos estar atentos à importação do módulo relevante para que o componente funcione corretamente.

Dica: Se a auto-completação do Material não estiver funcionando, você precisa adicionar a seguinte configuração ao arquivo tsconfig.json:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    .....
    "typeRoots": ["node_modules/@angular/material"]
  },
  "angularCompilerOptions": {
    .....
  }
}
```

Adicione a configuração a typeRoots e reinicie o vscode.

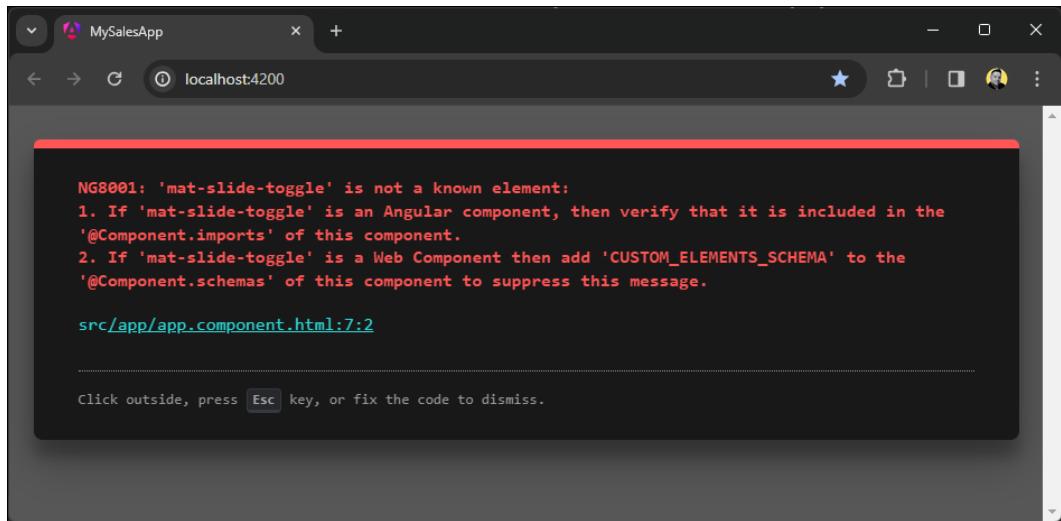
Para reforçar esse entendimento, vamos adicionar outro componente, *Material Slide Toggle*. Em app.component.html, adicione:

```
<div style="padding: 20px;">
  <button mat-button>this is a material button</button>

  <br />

  <mat-slide-toggle> Slide me! </mat-slide-toggle>
</div>
<router-outlet></router-outlet>
```

Ao adicionar este componente, uma tela de erro semelhante à seguinte figura é exibida:



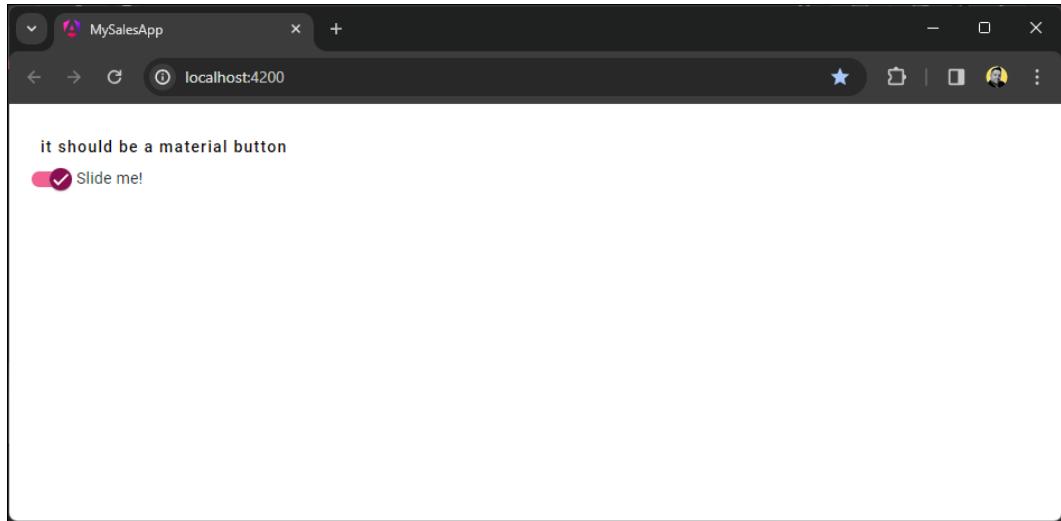
Para que o componente funcione, ele deve ser adicionado aos imports, da seguinte forma:

```
import {Component} from '@angular/core'
import {CommonModule} from '@angular/common'
import {RouterOutlet} from '@angular/router'
import {MatButtonModule} from '@angular/material/button'
import {MatSlideToggleModule} from '@angular/material/slide-toggle'

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, MatButtonModule, MatSlideToggleModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
```

```
)  
export class AppComponent {}
```

Com o `MatSlideToggleModule` adicionado, o componente funcionará perfeitamente, conforme mostrado na figura a seguir:



3.3. Adicionando Esquemáticos

A aplicação Angular é formada por um conjunto de componentes, que juntos compõem a aplicação. O Angular Material fornece algumas aplicações de exemplo que podemos usar, que são chamadas de Esquemáticos.

Esquemáticos são componentes Angular prontos para uso. O Angular Material tem alguns esquemáticos prontos para uso:

- **address-form** Componente com um grupo de formulário que usa controles de formulário do Material Design para solicitar um endereço de entrega
- **navigation** Cria um componente com um sidenav responsivo do Material Design e uma barra de ferramentas para mostrar o nome do aplicativo
- **dashboard** Componente com múltiplos cartões do Material Design e menus que são alinhados em um layout de grade

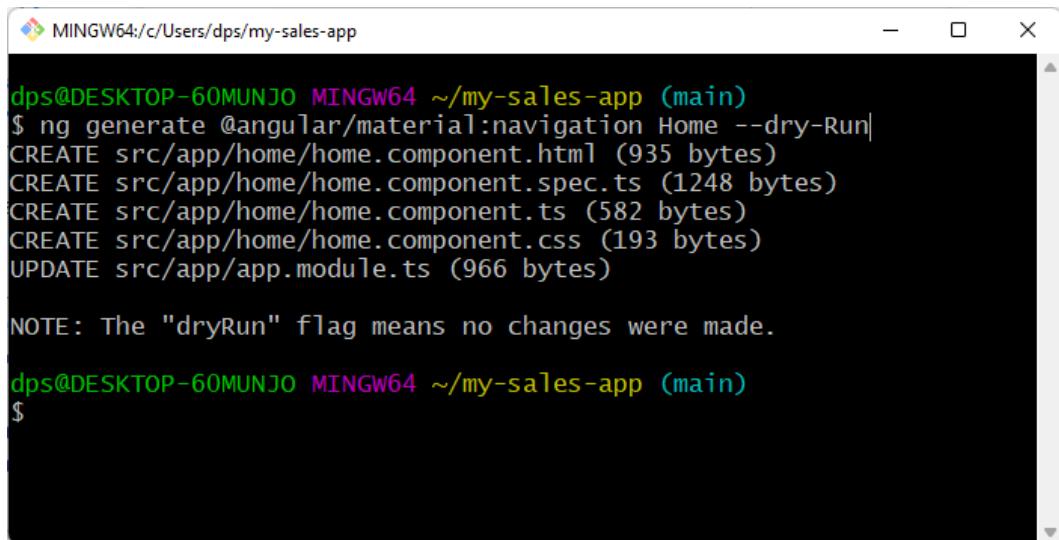
- **table** Gera um componente com uma tabela de dados do Material Design que suporta ordenação e paginação
- **tree** Componente que visualiza interativamente uma estrutura de pastas aninhadas usando o componente `<mat-tree>`

3.4. Adicionando uma Navegação em sua Aplicação

A aplicação que vamos criar precisa de uma barra de título e um menu de navegação. Para isso, usaremos a navegação dos Esquemáticos. No terminal, no diretório `my-sales-app`, execute o seguinte comando:

```
ng generate @angular/material:navigation Home --dry-run
```

A resposta para este comando é a seguinte:



```
MINGW64:/c/Users/dps/my-sales-app
dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (main)
$ ng generate @angular/material:navigation Home --dry-run
CREATE src/app/home/home.component.html (935 bytes)
CREATE src/app/home/home.component.spec.ts (1248 bytes)
CREATE src/app/home/home.component.ts (582 bytes)
CREATE src/app/home/home.component.css (193 bytes)
UPDATE src/app/app.module.ts (966 bytes)

NOTE: The "dryRun" flag means no changes were made.

dps@DESKTOP-60MUNJO MINGW64 ~/my-sales-app (main)
$
```

O *Cli* `ng` que já conhecemos, usaremos para realizar várias tarefas. Já vimos `ng new`, `ng serve` e agora temos `ng generate` que é usado para gerar “coisas” em nossa aplicação. O Esquemático “`@angular/material:navigation`” será aplicado ao novo componente chamado “Home”, ou seja, o Angular criará o componente “Home” baseado no esquemático de navegação do Angular Material.

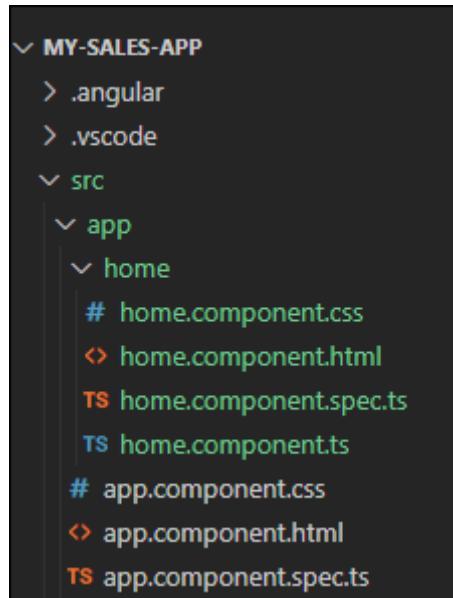
O último parâmetro do comando é `--dry-run`, que basicamente mostra o que será feito na aplicação, quais arquivos serão criados e quais arquivos serão modificados. Mas a mudança não será feita. Este parâmetro é útil para verificar se o componente está sendo criado no local correto.

O resultado é:

```
CREATE src/app/home/home.component.html (935 bytes)
CREATE src/app/home/home.component.spec.ts (1248 bytes)
CREATE src/app/home/home.component.ts (582 bytes)
CREATE src/app/home/home.component.css (193 bytes)
```

O Angular criará o diretório `/src/app/home` e criará todos os arquivos para criar o componente. Agora, vamos executar este mesmo comando sem a opção `--dry-run`.

Após a execução, os arquivos são criados:



Ao abrir o arquivo `src/app/home/home.component.html`, podemos ver o código HTML junto com alguns componentes do Material, começando com a tag `<mat-`.

O código Typescript para Home é mostrado abaixo:

```
import {Component, inject} from '@angular/core'
import {BreakpointObserver, Breakpoints} from '@angular/cdk/layout'
import {AsyncPipe} from '@angular/common'
import {MatToolbarModule} from '@angular/material/toolbar'
import {MatButtonModule} from '@angular/material/button'
import {MatSidenavModule} from '@angular/material/sidenav'
import {MatListModule} from '@angular/material/list'
import {MatIconModule} from '@angular/material/icon'
import {Observable} from 'rxjs'
import {map, shareReplay} from 'rxjs/operators'

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css'],
  standalone: true,
  imports: [
    MatToolbarModule,
    MatButtonModule,
    MatSidenavModule,
    MatListModule,
    MatIconModule,
    AsyncPipe
  ]
})
export class HomeComponent {
  private breakpointObserver = inject(BreakpointObserver)

  isHandset$: Observable<boolean> = this.breakpointObserver
    .observe(Breakpoints.Handset)
    .pipe(
      map((result) => result.matches),
      shareReplay()
    )
}
```

ATENÇÃO para as quebras de linha! Neste livro, quebras de linha podem aparecer às

vezes quando o código fonte é muito longo. Essas quebras de linha são representadas por uma barra invertida, como na linha `isHandset$` mostrada acima.

Veja que todos os componentes do Angular Material, como o Botão Material, são devidamente importados para que possam ser usados corretamente.

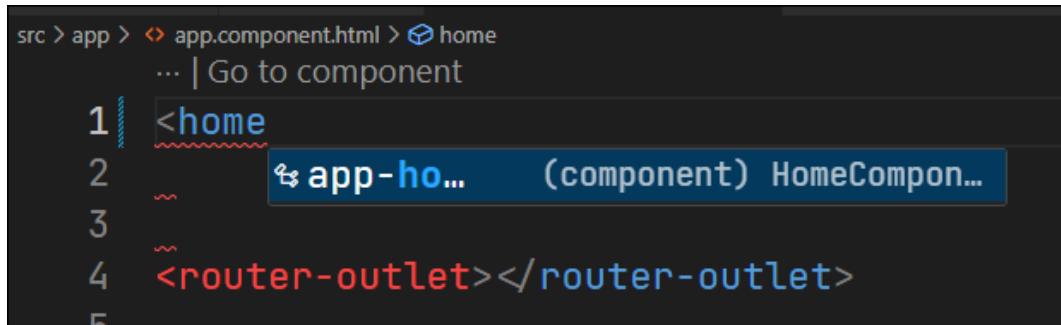
3.5. Adicionando o Componente Home ao App

Como sabemos, a aplicação está executando o componente App. Isso pode ser verificado no arquivo `main.ts`. O que precisamos fazer agora é adicionar o componente Home ao componente App. Primeiro, importe o componente no arquivo `app.component.ts`, conforme o exemplo a seguir:

```
import {Component} from '@angular/core'
import {CommonModule} from '@angular/common'
import {RouterOutlet} from '@angular/router'
import {MatButtonModule} from '@angular/material/button'
import {MatSlideToggleModule} from '@angular/material/slide-toggle'
import {HomeComponent} from './home/home.component'

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    CommonModule,
    RouterOutlet,
    MatButtonModule,
    MatSlideToggleModule,
    HomeComponent
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {}
```

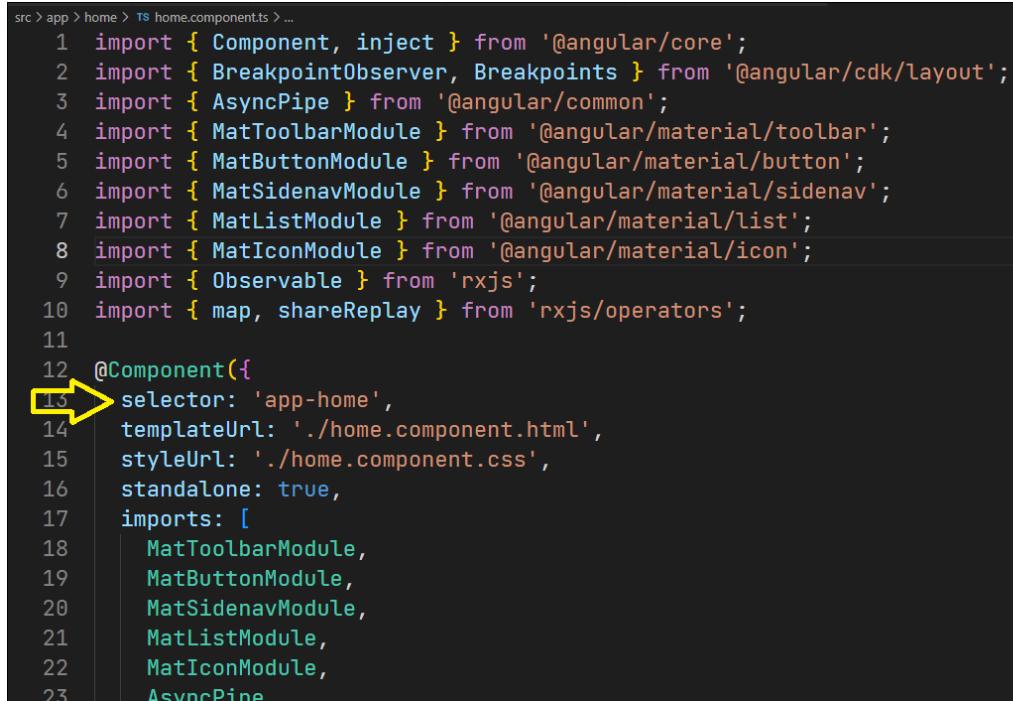
Abra o arquivo `src/app/app.component.html`, e abaixo de `Hello World`, abra a tag `<Ho...` e aguarde o Visual Studio Code completá-la com o nome do componente.



The screenshot shows a code editor with the file path `src > app > app.component.html`. A tooltip is displayed over the tag `<app-home>`, which is highlighted in red. The tooltip contains the text `app-home (component) HomeCompon...`. The code in the editor is:

```
1 <home>
2 ...
3 ...
4 <router-outlet></router-outlet>
5
```

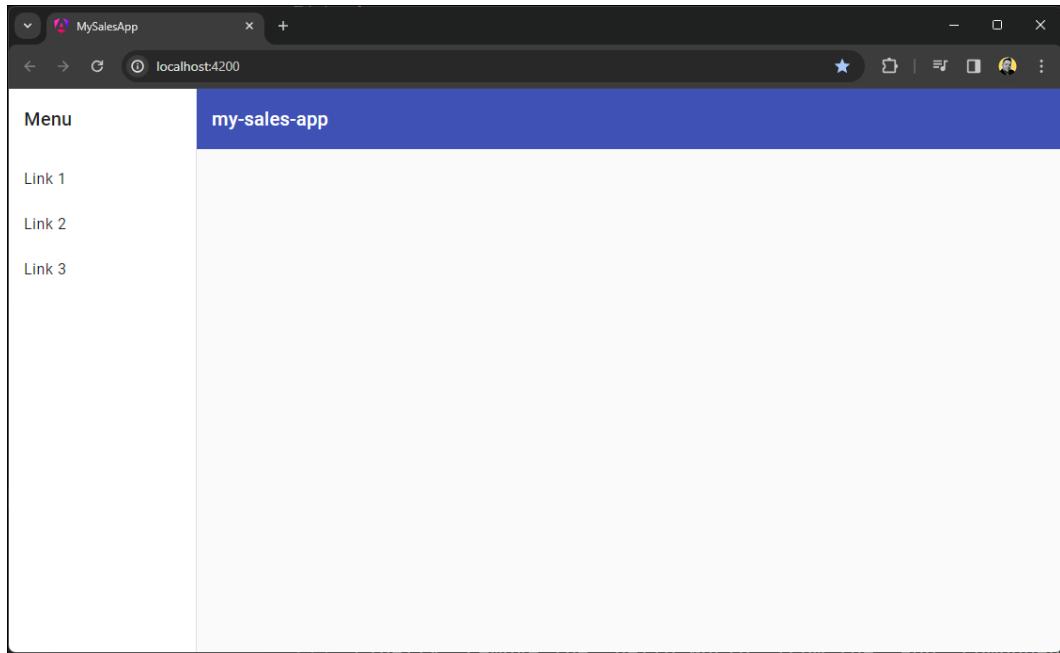
Veja que a tag html do componente é `<app-home>`, definida em `src/app/home/home.component.ts`:



The screenshot shows the file `src/app/home/home.component.ts`. A yellow arrow points to the `@Component` decorator at line 13. The code is:

```
1 import { Component, inject } from '@angular/core';
2 import { BreakpointObserver, Breakpoints } from '@angular/cdk/layout';
3 import { AsyncPipe } from '@angular/common';
4 import { MatToolbarModule } from '@angular/material/toolbar';
5 import { MatButtonModule } from '@angular/material/button';
6 import { MatSidenavModule } from '@angular/material/sidenav';
7 import { MatListModule } from '@angular/material/list';
8 import { MatIconModule } from '@angular/material/icon';
9 import { Observable } from 'rxjs';
10 import { map, shareReplay } from 'rxjs/operators';
11
12 @Component({
13   selector: 'app-home',
14   templateUrl: './home.component.html',
15   styleUrls: ['./home.component.css'],
16   standalone: true,
17   imports: [
18     MatToolbarModule,
19     MatButtonModule,
20     MatSidenavModule,
21     MatListModule,
22     MatIconModule,
23     AsyncPipe
24   ]
25 })
```

Uma vez que adicionamos o componente, podemos verificar o resultado no navegador. Verifique se o comando `ng server` está em execução. Se estiver, a página será atualizada automaticamente.



Se o ícone do menu não apareceu, significa que o menu está sendo exibido em uma tela de desktop. Como o Angular Material funciona de forma responsiva, o ícone do menu aparecerá se a tela ficar um pouco menor. Altere a largura da tela do navegador para ver as diferenças do menu.

3.6. Alterando o Home

O Esquemático é apenas o começo do componente. Após criá-lo, podemos mudar algumas informações. Abra o arquivo `src/app/home/home.component.html` e localize o seguinte texto `my-sales-app`. Mude para `Sales App`.

Vamos também mudar o menu. Ele foi criado da seguinte forma:

```
<mat-nav-list>
  <a mat-list-item href="#">Link 1</a>
  <a mat-list-item href="#">Link 2</a>
  <a mat-list-item href="#">Link 3</a>
</mat-nav-list>
```

Vamos criar um array e fazer um loop para adicionar itens de menu dinamicamente. Primeiro, crie a seguinte variável no arquivo `src/app/home/home.component.ts`:

```
// imports e @component...

export class HomeComponent {
  menuItems: Array<{path: string; label: string}> = [
    {
      path: '/',
      label: 'Home'
    },
    {
      path: '/categories',
      label: 'Categories'
    },
    {
      path: '/suppliers',
      label: 'Suppliers'
    }
  ]
}

// code ...

constructor(private breakpointObserver: BreakpointObserver) {}
```

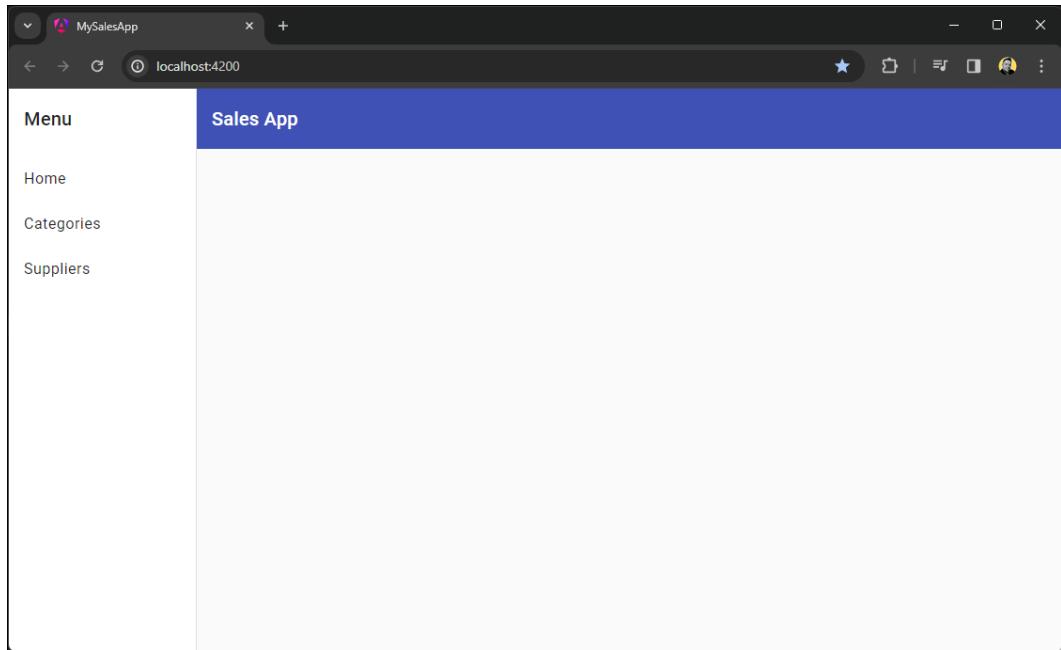
Vamos exemplificar ainda mais a criação deste Array. No template (`home.component.html`), podemos usar a diretiva `@for` para exibir os itens:

```
...
<mat-nav-list>
  <mat-nav-list>
    @for (item of menuItems; track item.path) {
      <a mat-list-item [href]="item.path">{{item.label}}</a>
    }
  </mat-nav-list>
</mat-nav-list>
...
...
```

A nova maneira de fazer um loop no Angular 17 é usar o bloco @for em vez de *ngFor.

Quando o Angular renderiza uma lista de elementos com @for, esses itens podem mudar ou mover posteriormente. O Angular precisa rastrear cada elemento através de qualquer reordenação, geralmente tratando uma propriedade do item como um identificador único ou chave.

Após criar o array e alterar o menu, temos o seguinte resultado:



3.7. Componentes

Criar o menu no componente `Home` dinamicamente foi uma boa prática de programação. Assim, para adicionar um novo item ao menu, basta adicionar um novo item ao Array. Outra boa prática de programação no Angular é componentizar cada responsabilidade na aplicação. O componente `Home` é responsável por exibir a página inicial da aplicação, e o menu da aplicação é outra responsabilidade, separada de `Home`.

Portanto, para o menu, devemos criar um novo componente, chamado `Menu`. No terminal, no diretório `my-sales-app`, digite o seguinte comando:

```
$ ng g component menu --dry-run
```

```
CREATE src/app/menu/menu.component.html (19 bytes)
CREATE src/app/menu/menu.component.spec.ts (612 bytes)
CREATE src/app/menu/menu.component.ts (267 bytes)
CREATE src/app/menu/menu.component.css (0 bytes)
```

NOTE: The "dryRun" flag means no changes were made.

O `g` é um atalho para `generate`. Você pode usar, por exemplo, `ng g c` que é um atalho para `ng generate component`. Usamos o parâmetro `--dry-run` para verificar quais arquivos serão criados. Você pode omitir a criação do arquivo `.css` e do arquivo `.spec`. Até mesmo o template, o arquivo `.html`, pode ser omitido. Desta forma, teremos o componente `menu` em apenas um arquivo. O seguinte comando adicionou parâmetros para criar um componente mais enxuto.

```
$ ng g c menu --inline-style --skip-tests --inline-template --dry-run
```

```
CREATE src/app/menu/menu.component.ts (259 bytes)
UPDATE src/app/app.module.ts (1071 bytes)
```

NOTE: The "dryRun" flag means no changes were made.

Agora que verificamos quais arquivos serão criados, execute o comando novamente sem a opção `--dry-run`. O arquivo `src/app/menu/menu.component.ts` será criado com o seguinte código:

```
import {Component} from '@angular/core'

@Component({
  selector: 'app-menu',
  standalone: true,
  imports: [],
  template: `<p>menu works!</p>`,
  styles: []
})
export class MenuComponent {}
```

Para componentes muito pequenos, é uma boa prática usar apenas um único arquivo. Entenda como um “componente pequeno”, um componente cujo template tem no máximo 5 linhas de comprimento.

Agora mova o código para o menu no componente Home para o componente Menu:

```
import {Component} from '@angular/core'
import {MatListModule} from '@angular/material/list'

@Component({
  selector: 'app-menu',
  standalone: true,
  imports: [MatListModule],
  template: `
    @for (item of menuItems; track item.path) {
      <a mat-list-item [href]="item.path">{{ item.label }}</a>
    }
  `,
  styles: ``
})
export class MenuComponent {
  menuItems: Array<{path: string; label: string}> = [
    {
      path: '/',
      label: 'Home'
    },
    {
      path: '/categories',
      label: 'Categories'
    },
    {
      path: '/suppliers',
      label: 'Suppliers'
    }
  ]
}
```

Não esqueça de adicionar o `MatListModule` aos imports, pois é necessário para que o `mat-list-item` seja renderizado corretamente.

Com o componente `Menu` pronto, retorne ao componente `Home` e adicione o componente `Menu` através da tag `<app-menu></app-menu>`:

```
<!-- src/app/home/home.component.html -->
..... code .....
<mat-sidenav
  #
  Drawer
  Class="sidenav"
  FixedInViewport
  [attr.role]="(isHandset$ | async) ? 'dialog' : 'navigation'"
  [mode]="(isHandset$ | async) ? 'over' : 'side'"
  [opened]="(isHandset$ | async) === false"
>
  <mat-toolbar>Menu</mat-toolbar>
  <mat-nav-list> <app-menu></app-menu> <<< AQUI </mat-nav-list>
</mat-sidenav>
..... code .....
```

Não esqueça de adicionar o `MenuComponent` nos imports do `home.component.ts`!

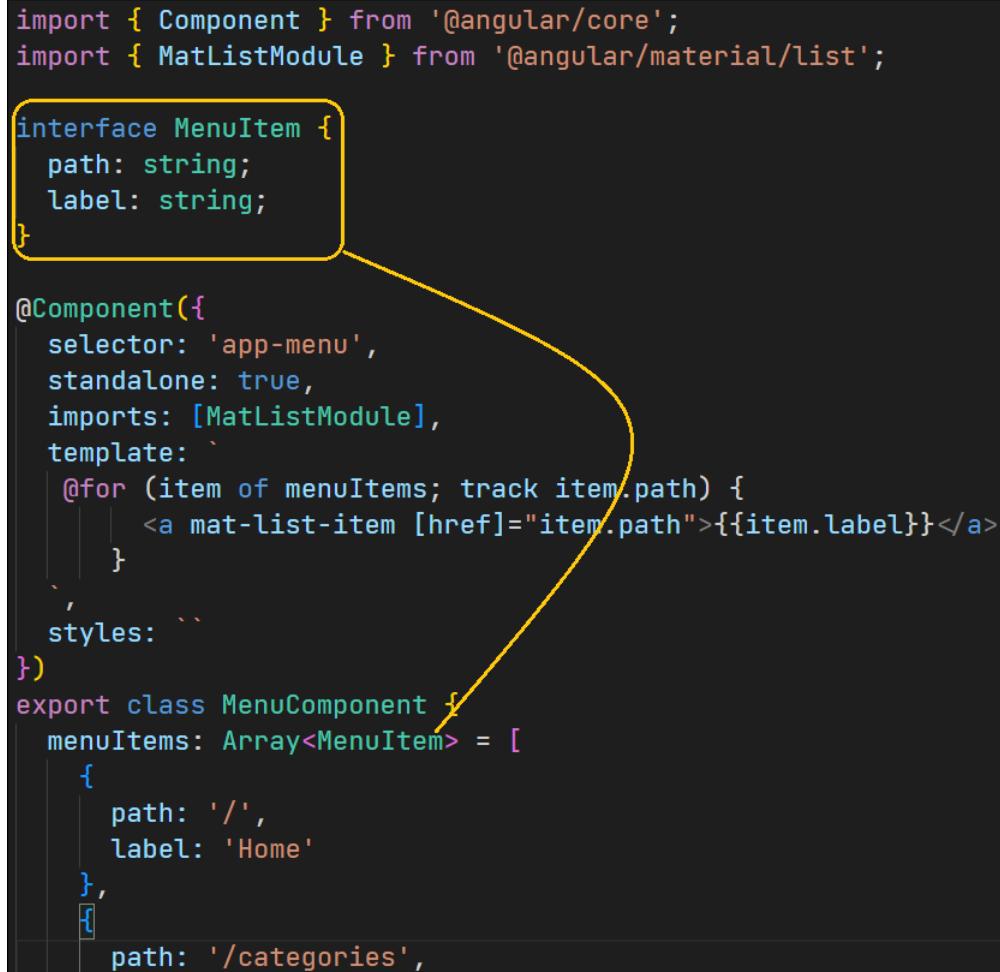
3.8. Typescript e Interfaces

Angular utiliza a linguagem Typescript e podemos fazer uso de algumas das características extras que ela tem sobre o Javascript. Typescript é basicamente JavaScript com tipos, e quando criamos a variável `menuItems` definimos seu tipo, que é um objeto que possui duas propriedades: `path` e `label`:

```
menuItems: Array<{path: string; label: string}>
```

Usando o Visual Studio Code, podemos extrair esse tipo para uma interface. Selecione o código `{ path: string; label: string }` e pressione o botão direito do mouse, vá em

Refatorar e escolha o item **Extrair para Interface**. Escolha o nome `MenuItem`, e a interface será criada da seguinte forma:



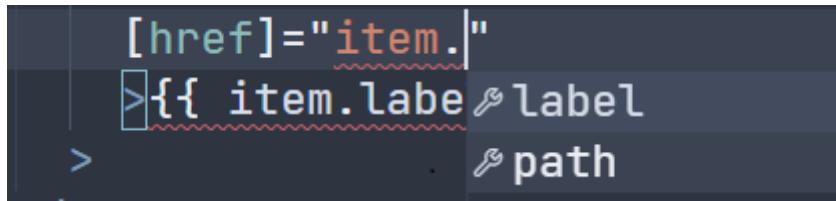
```
import { Component } from '@angular/core';
import { MatListModule } from '@angular/material/list';

interface MenuItem {
  path: string;
  label: string;
}

@Component({
  selector: 'app-menu',
  standalone: true,
  imports: [MatListModule],
  template: `
    @for (item of menuItems; track item.path) {
      | <a mat-list-item [href]="item.path">{{item.label}}</a>
    }
  `,
  styles: []
})
export class MenuComponent {
  menuItems: Array<MenuItem> = [
    {
      path: '/',
      label: 'Home'
    },
    {
      path: '/categories',
      label: 'Categories'
    }
  ];
}
```

Você também pode extrair a definição para um novo arquivo selecionando a Interface `MenuItem` e executando o comando **Refatorar** novamente e selecionando **Mover para um novo arquivo**.

Criar Interfaces ajuda a IDE a identificar corretamente os tipos de variáveis e a autocompletar o código:



Você pode adicionar comentários às propriedades da interface para torná-las mais fáceis de entender. A IDE exibirá esses comentários.

```
export interface MenuItem {  
  /**  
   * The path that will be loaded when you click on the menu  
   */  
  path: string  
  /**  
   * The text that will be displayed in the menu  
   */  
  label: string  
}
```

```
interface MenuItem {  
  /**  
   * The path that will be loaded when you click on the menu  
   */  
  path: string;  
  /**  
   * The text that will be displayed in the menu  
   */  
  label: string;  
}  
  
@Component({  
  selector: 'app-menu',  
  standalone: true,  
  imports: [MatListModule],  
  template: `  
    @for (item of menuItems; track item) The path that will be loaded when you click on the menu  
    | <a mat-list-item [href]="item.path">{{item.label}}</a>  
  `}  
}
```

3.9. Componente de Categorias

Vamos começar criando o componente que exibirá as categorias de um produto. Crie o componente da seguinte forma:

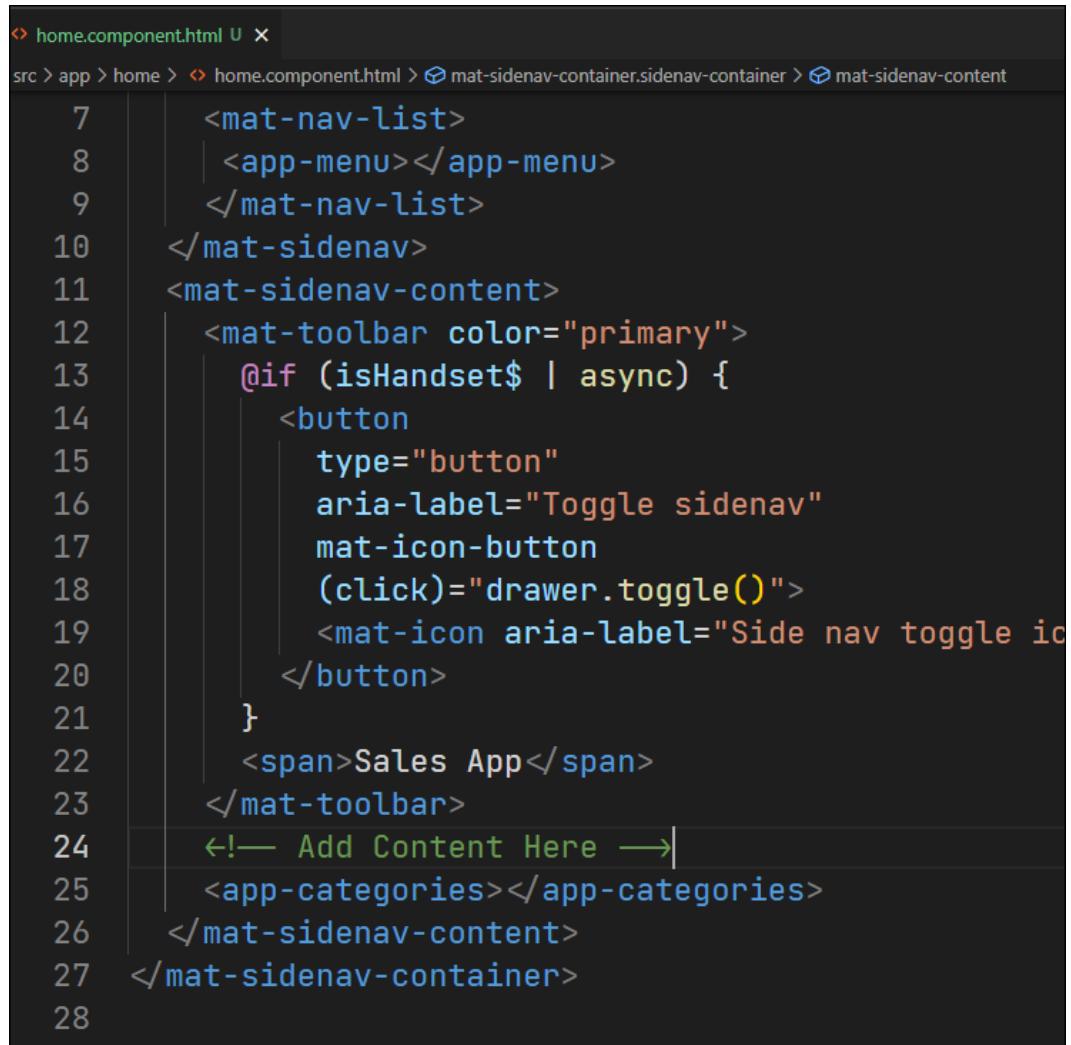
```
ng g @angular/material:table categories --inline-style --skip-tests --dry-run  
  
CREATE src/app/categories/categories.datasource.ts (3654 bytes)  
CREATE src/app/categories/categories.component.html (884 bytes)  
CREATE src/app/categories/categories.component.ts (1087 bytes)
```

NOTE: The "dryRun" flag means no changes were made.

O Schematic @angular/material:table será usado para criar este componente, exibindo uma tabela com dados. Os parâmetros `--inline-style` e `--skip-tests` já são conhecidos.

Como usamos o parâmetro `--dry-run`, os arquivos ainda não foram criados. Após verificar que os arquivos serão criados no lugar certo, podemos repetir o comando sem a opção `--dry-run`. Os arquivos serão criados em `src/app/categories`.

Para ver o componente na tela, adicione o `CategoriesComponent` aos parâmetros de importação (`home.component.ts`) e localize a tag `<!-- Add Content Here -->` no template `home.component.html`. Abaixo deste comentário, adicione o componente `Categories`:



```
home.component.html U X
src > app > home > home.component.html > mat-sidenav-container.sidenav-container > mat-sidenav-content
  7   <mat-nav-list>
  8     <app-menu></app-menu>
  9   </mat-nav-list>
 10  </mat-sidenav>
 11  <mat-sidenav-content>
 12    <mat-toolbar color="primary">
 13      @if (isHandset$ | async) {
 14        <button
 15          type="button"
 16          aria-label="Toggle sidenav"
 17          mat-icon-button
 18          (click)="drawer.toggle()"
 19          <mat-icon aria-label="Side nav toggle icon">
 20            </button>
 21      }
 22      <span>Sales App</span>
 23    </mat-toolbar>
 24    <!-- Add Content Here -->
 25    <app-categories></app-categories>
 26  </mat-sidenav-content>
 27</mat-sidenav-container>
 28
```

O resultado no navegador é o seguinte:

	Id	Name
Home	1	Hydrogen
Categories	2	Helium
Suppliers	3	Lithium
	4	Beryllium
	5	Boron
	6	Carbon
	7	Nitrogen
	8	Oxygen
	9	Fluorine

O componente Categories foi adicionado à aplicação, mas observe que não usamos o menu para carregar o componente. Esta não é a maneira correta de carregar componentes no Angular, porque se você tem 100 componentes, não pode carregá-los todos de uma vez. Para fazer isso, usamos um conceito chamado “Rotas”.

3.10. Rotas

Com o menu pronto, precisamos carregar cada componente específico conforme o usuário seleciona o item no menu.

Esta tarefa é realizada pelo Angular Router. Abra o arquivo `packages.json` e verifique se o pacote `@angular/router` está instalado na seção “dependencies”. Se você não encontrar `@angular/router`, pode instalar a biblioteca manualmente usando o comando `npm i @angular/router`.

Para adicionar uma rota, abra o arquivo `app.routes.ts` e adicione o componente de categorias na variável de rotas:

```
import {Routes} from '@angular/router'
import {CategoriesComponent} from './categories/categories.component'

export const routes: Routes = [
  {
    path: 'categories',
    component: CategoriesComponent
  }
]
```

Este array irá mudar constantemente. Quando criarmos o componente *Suppliers*, iremos adicioná-lo a este array.

Para definir onde o Router irá carregar o conteúdo do componente, usamos a tag `<router-outlet>`. Remova a tag `<router-outlet>` do template `app.component.html` e abra o arquivo `home.component.ts`. Adicione `RouterOutlet` aos imports:

```
// imports
import {RouterOutlet} from '@angular/router'

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css'],
  standalone: true,
  imports: [
    MatToolbarModule,
    MatButtonModule,
    MatSidenavModule,
    MatListModule,
    MatIconModule,
    AsyncPipe,
    MenuComponent,
    CategoriesComponent,
    RouterOutlet
  ]
})
export class HomeComponent {
  private breakpointObserver = inject(BreakpointObserver)

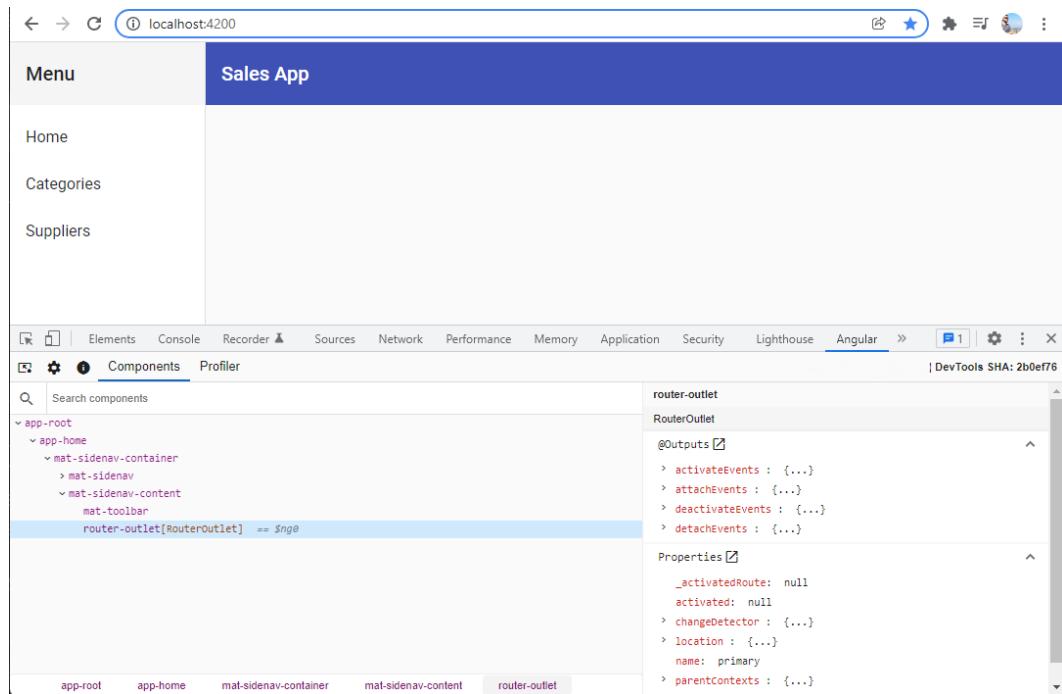
  isHandset$: Observable<boolean> = this.breakpointObserver
    .observe(Breakpoints.Handset)
```

```
.pipe(  
  map((result) => result.matches),  
  shareReplay()  
)  
}
```

Mude o `<app-categories></app-categories>` por `<router-outlet></router-outlet>`:

```
<mat-sidenav-container class="sidenav-container">  
  ... code ...  
  <span>Sales App</span>  
  </mat-toolbar>  
  <!-- Add Content Here -->  
  <router-outlet></router-outlet>  
  </mat-sidenav-content>  
</mat-sidenav-container>
```

A tag `<router-outlet></router-outlet>` irá carregar conteúdo de acordo com a variável `routes`. Por agora, temos que `categories` irá carregar o componente `Categories`. Na aplicação, quando acessamos “`http://localhost:4200/`” obtemos a seguinte tela (veja detalhe das ferramentas de desenvolvimento do angular):



Quando acessamos a url `http://localhost:4200/categories` o componente Categories é carregado, como mostrado na figura a seguir:

The screenshot shows a browser window with the URL `localhost:4200/categories`. The page title is "Sales App". On the left, there's a sidebar with "Menu" and three items: "Home", "Categories", and "Suppliers". The main content area displays a table with three rows:

	Id	Name
1		Hydrogen
2		Helium
3		Lithium

Below the browser window is the Chrome DevTools interface. The "Components" tab is selected. The left pane shows the component tree for "app-root", with "app-home" expanded, showing "mat-sidenav-container", "mat-sidenav", "mat-toolbar", and a "router-outlet" node highlighted. The right pane shows the properties for the "router-outlet" component, including methods like "@Outputs" and "Properties".

3.11. Criando um Painel

O menu “Home”, que aponta para a Url “/” está vazio e podemos configurar um novo componente para ele. Vamos criar um painel, da seguinte forma:

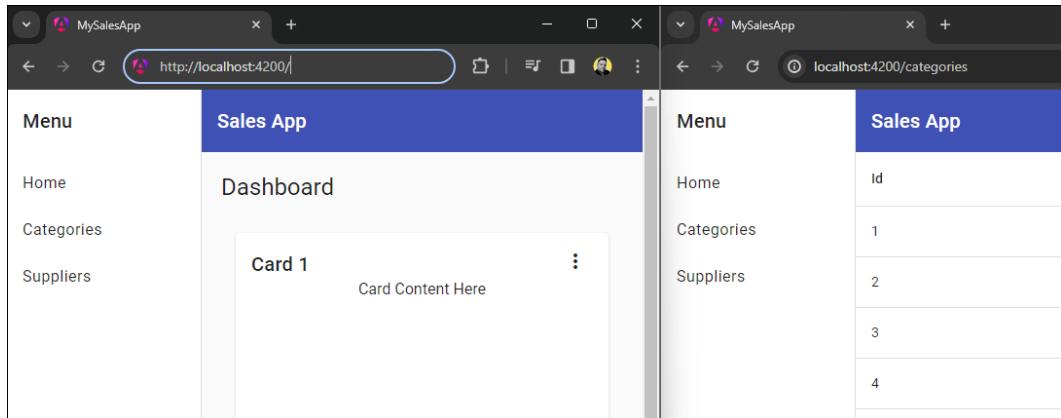
```
$ ng g @angular/material:dashboard dashboard --inline-style --skip-tests
CREATE src/app/dashboard/dashboard.component.html (936 bytes)
CREATE src/app/dashboard/dashboard.component.ts (1357 bytes)
```

Após criar o componente, altere o arquivo `app.routes.ts` adicionando o componente `Dashboard` ao caminho ‘’:

```
import {Routes} from '@angular/router'
import {CategoriesComponent} from './categories/categories.component'
import {DashboardComponent} from './dashboard/dashboard.component'

export const routes: Routes = [
  {
    path: 'categories',
    component: CategoriesComponent
  },
  {
    path: '',
    component: DashboardComponent
  }
]
```

Após adicionar a rota, volte para a aplicação e navegue entre o menu “Home” e “Categories” para ver o Router em ação.



4. Categorias

Inicialmente, a tela de categorias tem o seguinte layout:

Id	Name
1	Hydrogen
2	Helium
3	Lithium
4	Beryllium
5	Boron
6	Carbon
7	Nitrogen
8	Oxygen
9	Fluorine

4.1. O Card do Angular Material

Para tornar a tela mais organizada, usaremos o componente Card do Angular Material, que é composto pelos seguintes componentes:

- <mat-card-title> Título do card
- <mat-card-subtitle> Subtítulo do card
- <mat-card-content> Conteúdo principal do card. Destinado a blocos de texto
- <mat-card-header> Contêiner para mat-card-title e mat-card-subtitle
- <mat-card-actions> Contêiner para botões na parte inferior do card

Adiciona MatCardModule à propriedade imports em categories.component.ts:

```
// imports...

@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  styles: [
    '.full-width-table {
      width: 100%;
    }
  ],
  standalone: true,
  imports: [
    MatTableModule,
    MatPaginatorModule,
    MatSortModule,
    MatCardModule,
    MatButtonModule
  ]
})
export class CategoriesComponent implements AfterViewInit {
  // ...code...
}
```

O código inicial para o template `src\app\categories.component.html` é o seguinte:

```
<div class="mat-elevation-z8">
  <table mat-table class="full-width-table" matSort aria-label="Elements">
    <!-- Id Column -->
    <ng-container matColumnDef="id">
      <th mat-header-cell *matHeaderCellDef mat-sort-header>Id</th>
      <td mat-cell *matCellDef="let row">{{row.id}}</td>
    </ng-container>

    <!-- Name Column -->
    <ng-container matColumnDef="name">
      <th mat-header-cell *matHeaderCellDef mat-sort-header>Nome</th>
      <td mat-cell *matCellDef="let row">{{row.name}}</td>
    </ng-container>

    <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
    <tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
  </table>
```

```
<mat-paginator
  #
  Paginator
  [length]="dataSource?.data?.length"
  [pageIndex]="0"
  [pageSize]="10"
  [pageSizeOptions]=[5, 10, 20]
  aria-label="Select page"
>
</mat-paginator>
</div>
```

Adicione o componente Card com o seguinte código:

```
<mat-card>
  <mat-card-header>
    <mat-card-title>Categories</mat-card-title>
    <mat-card-subtitle>Listing all categories</mat-card-subtitle>
  </mat-card-header>

  <mat-card-content>
    <table mat-table class="full-width-table" matSort aria-label="Elements">
      <!-- Id Column -->
      <ng-container matColumnDef="id">
        <th mat-header-cell *matHeaderCellDef mat-sort-header>Id</th>
        <td mat-cell *matCellDef="let row">{{ row.id }}</td>
      </ng-container>

      <!-- Name Column -->
      <ng-container matColumnDef="name">
        <th mat-header-cell *matHeaderCellDef mat-sort-header>Nome</th>
        <td mat-cell *matCellDef="let row">{{ row.name }}</td>
      </ng-container>

      <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
      <tr mat-row *matRowDef="let row; columns: displayedColumns"></tr>
    </table>

    <mat-paginator
      #
      Paginator
      [length]="dataSource?.data?.length"
```

```
[pageIndex]="0"
[pageSize]="10"
[pageSizeOptions]=[5, 10, 20]
aria-label="Select page"
>
</mat-paginator>
</mat-card-content>

<mat-card-actions>
  <button mat-button>New Category</button>
</mat-card-actions>
</mat-card>
```

Ao incluir o Material Card, temos o seguinte layout:

The screenshot shows a web browser window titled "MySalesApp" at the URL "localhost:4200/categories". The interface is divided into two main sections: a sidebar on the left and a main content area on the right.

Sidebar (Menu):

- Home
- Categories
- Suppliers

Main Content Area (Sales App):

Categories

Listing all categories

ID	Name
1	Hydrogen
2	Helium
3	Lithium
4	Beryllium
5	Boron
6	Carbon
7	Nitrogen
8	Oxygen
9	Fluorine
10	Neon

Items per page: 1 - 10 of 20 < >

New Category

4.2. Adicionando um Estilo Css Global

Vamos adicionar um pequeno detalhe de layout para tornar a aplicação visualmente melhor. Usando o `<router-outlet>`, podemos adicionar uma pequena margem para que o efeito do Card seja visível. Abra o componente `src\app\home.component.html` e, onde a tag `<router-outlet>` está, adicione um `<div>`, conforme segue:

```
<!-- Add Content Here -->
<div class="m-10">
  <router-outlet></router-outlet>
</div>
```

A classe “m-10” ainda não foi criada, e podemos criá-la no arquivo `src\app\home.component.css`:

```
.sidenav-container {
  height: 100%;
}

.sidenav {
  width: 200px;
}

.sidenav .mat-toolbar {
  background: inherit;
}

.mat-toolbar.mat-primary {
  position: sticky;
  top: 0;
  z-index: 1;
}

.m-10 {
  margin: 10px;
}
```

O Cartão de Categoria tem a seguinte aparência:

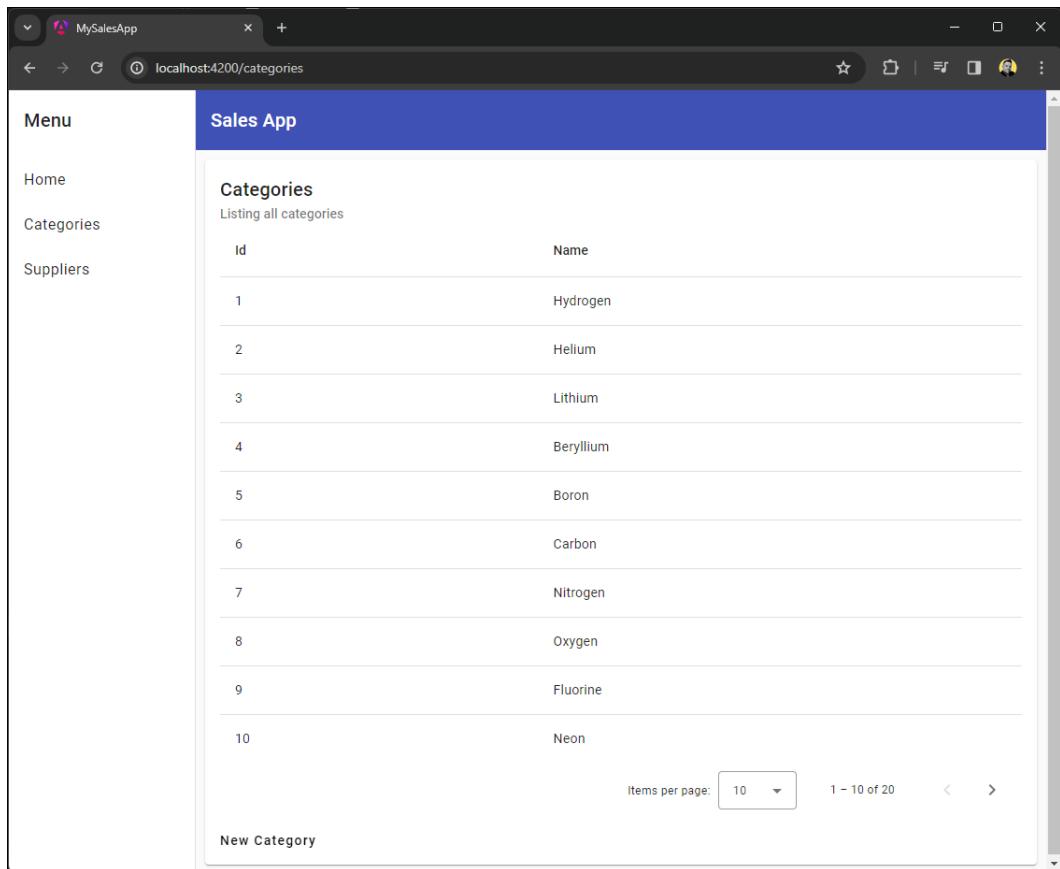


Figure 11. Cartão de categoria com margens

O problema foi corrigido, mas a classe CSS `m-10` só pode ser usada no componente `Home`, pois foi definida em `home.component.css`. Seria importante tornar a classe CSS `m-10` disponível em todo o aplicativo, caso precise ser usada novamente. Neste caso, moveremos o código CSS de `m-10` para o arquivo `src/styles.css`:

```
/* You can add global styles to this file, and also import other style files */

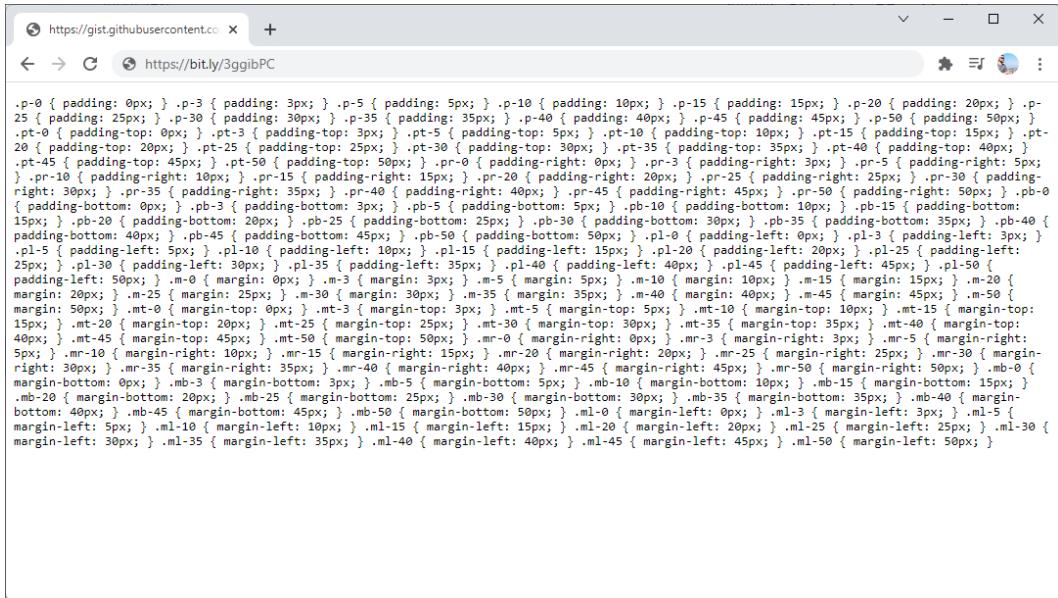
html,
body {
  height: 100%;
}
body {
  margin: 0;
  font-family: Roboto, 'Helvetica Neue', sans-serif;
}

.m-10 {
  margin: 10px;
}
```

4.3. Mais Estilos de Margem/Espaçamento (opcional)

Seria interessante ter mais estilos de espaçamento. Por exemplo, poderíamos ter `m-20` que representaria `margin:20px`, e `p-30` que representaria `padding: 30px`. Além disso, você pode ter `pt-10` que significa `padding-top:10px`.

Para implementar essa funcionalidade, o primeiro passo é obter todos os estilos. Você pode fazer isso copiando o código deste link: <https://bit.ly/3ggibPC>

A screenshot of a web browser window. The address bar shows two URLs: 'https://gist.githubusercontent.com' and 'https://bit.ly/3ggibPC'. The main content area of the browser is filled with a massive amount of CSS code, which appears to be a single, long string of characters due to the font used for the code. The code is a mix of various CSS properties like padding, margin, and font sizes, applied to classes such as .p-0 through .p-40, .m-0 through .m-40, and .mb-0 through .mb-40.

Se o link bit.ly não funcionar, vá para a [url original](#)

Após copiar o código css, crie o arquivo `src/margin-padding-helpers.css` e cole o código CSS.



```
my-sales-app > src > margin-padding-helpers.css > ...
1 .p-0 { padding: 0px; } .p-3 { padding: 3px; } .p-5 { padding: 5px; } .p-10 { padding: 10px; } .p-15 { padding: 15px; } .p-20 { padding: 20px; } .p-25 { padding: 25px; } .p-30 { padding: 30px; } .p-35 { padding: 35px; } .p-40 { padding: 40px; } .p-45 { padding: 45px; } .p-50 { padding: 50px; } .pt-0 { padding-top: 0px; } .pt-3 { padding-top: 3px; } .pt-5 { padding-top: 5px; } .pt-10 { padding-top: 10px; } .pt-15 { padding-top: 15px; } .pt-20 { padding-top: 20px; } .pt-25 { padding-top: 25px; } .pt-30 { padding-top: 30px; } .pt-35 { padding-top: 35px; } .pt-40 { padding-top: 40px; } .pt-45 { padding-top: 45px; } .pt-50 { padding-top: 50px; } .pr-0 { padding-right: 0px; } .pr-3 { padding-right: 3px; } .pr-5 { padding-right: 5px; } .pr-10 { padding-right: 10px; } .pr-15 { padding-right: 15px; } .pr-20 { padding-right: 20px; } .pr-25 { padding-right: 25px; } .pr-30 { padding-right: 30px; } .pr-35 { padding-right: 35px; } .pr-40 { padding-right: 40px; } .pr-45 { padding-right: 45px; } .pr-50 { padding-right: 50px; } .pb-0 { padding-bottom: 0px; } .pb-3 { padding-bottom: 3px; } .pb-5 { padding-bottom: 5px; } .pb-10 { padding-bottom: 10px; } .pb-15 { padding-bottom: 15px; } .pb-20 { padding-bottom: 20px; } .pb-25 { padding-bottom: 25px; } .pb-30 { padding-bottom: 30px; } .pb-35 { padding-bottom: 35px; } .pb-40 { padding-bottom: 40px; } .pb-45 { padding-bottom: 45px; } .pb-50 { padding-bottom: 50px; } .pl-0 { padding-left: 0px; } .pl-3 { padding-left: 3px; } .pl-5 { padding-left: 5px; } .pl-10 { padding-left: 10px; } .pl-15 { padding-left: 15px; } .pl-20 { padding-left: 20px; } .pl-25 { padding-left: 25px; } .pl-30 { padding-left: 30px; } .pl-35 { padding-left: 35px; }
```

Agora, você precisa configurar o Angular para carregar o arquivo `src/margin-padding-helpers.css`. Para fazer isso, abra o arquivo `angular.json` e encontre a configuração “styles”, adicione o arquivo `src/margin-padding-helpers.css` logo abaixo de “`src/styles.css`”:

```
"styles": [
  "./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
  "src/styles.css",
  "src/margin-padding-helpers.css"
],
```

ATENÇÃO: Sempre que você alterar o arquivo `angular.json`, você **precisa reiniciar o aplicativo**. Pare o processo `ng serve` em execução e execute-o novamente.

Após essa alteração, você pode usar outras margens. Por exemplo, teste “`m-50`” onde “`m-10`” estava em `home.component.html` para ver o seguinte resultado:

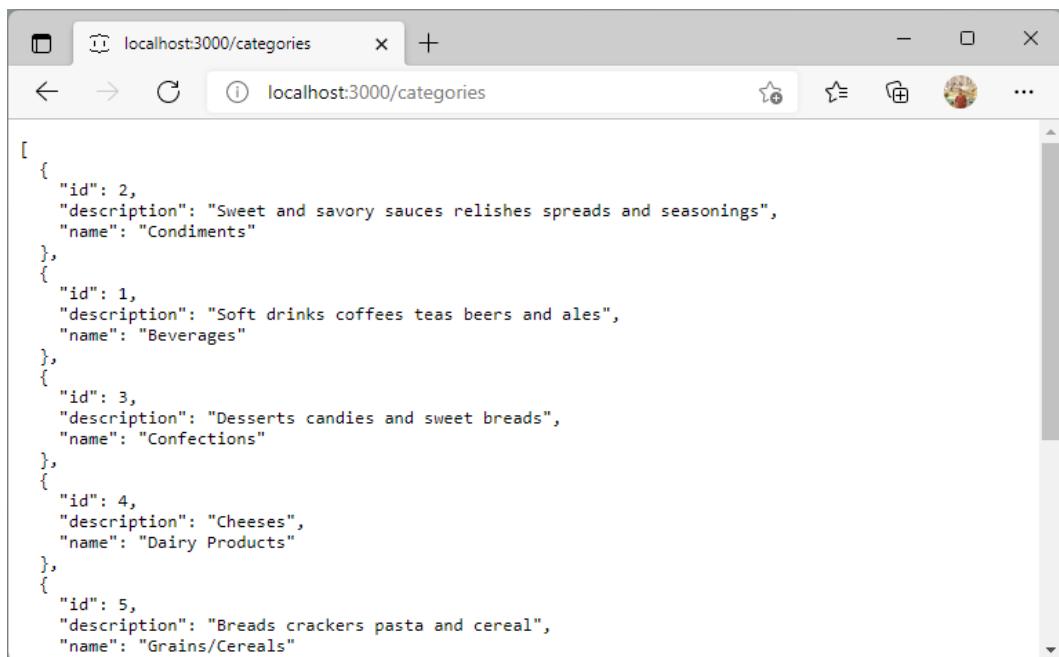
Categories	
Listing all categories	
Id	Name
1	Hydrogen
2	Helium
3	Lithium

Figure 12. The router-outlet with m-50, margin:50px

Dica final: você pode remover m-10 do arquivo `src/styles.css`

4.4. Obtendo Dados da API de Categoria

No início deste livro, aprendemos como criar um servidor backend falso para que possamos usá-lo no projeto Angular. Execute o projeto conforme mostrado no capítulo 1, e verifique que ao acessar `http://localhost:3000/categories` obtemos o seguinte resultado:



A screenshot of a Microsoft Edge browser window. The address bar shows 'localhost:3000/categories'. The main content area displays a JSON array of five categories:

```
[{"id": 2, "description": "Sweet and savory sauces, relishes, spreads and seasonings", "name": "Condiments"}, {"id": 1, "description": "Soft drinks, coffees, teas, beers and ales", "name": "Beverages"}, {"id": 3, "description": "Desserts, candies and sweet breads", "name": "Confections"}, {"id": 4, "description": "Cheeses", "name": "Dairy Products"}, {"id": 5, "description": "Breads, crackers, pasta and cereal", "name": "Grains/Cereals"}]
```

Figure 13. The backend running at localhost:3000

Esta resposta, que está em formato `json`, pode ser lida pelo Angular através da classe `HttpClient`.

4.5. Configurando HttpClient

Para configurar o `HttpClient`, você deve adicionar o método `provideHttpClient` no arquivo `app.config.ts`.

```
// src\app\app.config.ts
import {ApplicationConfig} from '@angular/core'
import {provideRouter} from '@angular/router'

import {routes} from './app.routes'
import {provideAnimations} from '@angular/platform-browser/animations'
import {provideHttpClient} from '@angular/common/http'

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes), provideAnimations(), provideHttpClient()]
}
```

4.6. Serviços

Após o HttpClient estar configurado, podemos, por exemplo, diretamente no arquivo categories.datasource.ts, usar esse recurso para chamar a api e obter dados. Como alternativa, vamos falar sobre um novo conceito no Angular, chamado **Serviço**.

Um *Serviço* é uma classe no Angular que pode ser injetada em qualquer outra classe ou componente na aplicação. Ele usa o conceito de injeção de dependência, no qual podemos injetar o serviço diretamente no componente. Podemos estabelecer que qualquer acesso à API deve ser responsabilidade do *Serviço*, não do *Componente*. Assim, se outro Componente precisar acessar a mesma API, ela sempre estará disponível na classe de *Serviço*.

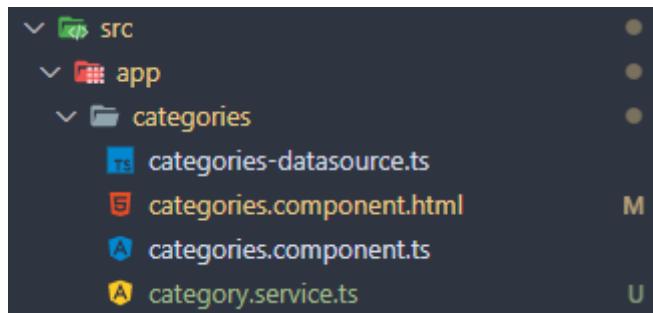
4.7. O Serviço de Categoria

Vamos agora criar o primeiro serviço da aplicação. Para isso, vamos novamente usar o Angular Cli:

```
$ ng g s categories/category --skip-tests --dry-run
CREATE src/app/categories/category.service.ts (137 bytes)
```

NOTE: The "dryRun" flag means no changes were made.

O comando ng g s é um atalho para ng generate service. Estaremos adicionando o serviço ao diretório src/categories. Após executar o comando em modo --dry-run e verificar que está sendo criado no lugar correto, podemos executá-lo para de fato criar a classe. O diretório categories agora contém os seguintes arquivos:



E o código inicial do CategoryService é o seguinte:

```
import {Injectable} from '@angular/core'

@Injectable({
  providedIn: 'root'
})
export class CategoryService {
  constructor() {}
}
```

Então, vamos criar o primeiro método do CategoryService chamado getAll, que irá obter todas as categorias. Primeiro, você precisa injetar a classe HttpClient no CategoryService. Isso é feito da seguinte forma:

```
import {HttpClient} from '@angular/common/http'
import {Injectable} from '@angular/core'

@Injectable({
  providedIn: 'root'
})
export class CategoryService {
  constructor(private http: HttpClient) {}
}
```

A variável http é declarada diretamente no construtor da classe. Dessa forma, podemos usar o HttpClient através de `this.http`. Com a classe `HttpClient` devidamente injetada no Service, podemos criar o método getAll.

4.8. Primeira Versão do Método GetAll()

De forma mais simples, o método getAll pode ser escrito como:

```
import {HttpClient} from '@angular/common/http'
import {Injectable} from '@angular/core'

@Injectable({
  providedIn: 'root'
})
export class CategoryService {
  constructor(private http: HttpClient) {}

  public getAll() {
    return this.http.get('http://localhost:3000/categories')
  }
}
```

O problema neste ponto é que, mesmo com o método funcionando perfeitamente, o uso de “`http://localhost:3000`” pode ser um problema se, por exemplo, essa url mudar. Geralmente, quando precisamos trabalhar com informações que podem ser diferentes em diferentes servidores (como o servidor de desenvolvimento e o servidor de produção final), usamos o conceito de variáveis de ambiente.

4.9. Variáveis de Ambiente

Se você nunca ouviu falar sobre isso, é hora de entender esse conceito. Uma variável de ambiente é uma informação ou um valor que pode mudar dependendo do servidor. Por exemplo, enquanto você está trabalhando no desenvolvimento da aplicação, no seu computador, você usará “`http://localhost:3000`” para acessar a API. Mas quando a aplicação está rodando no servidor de produção (como Amazon, Heroku, Digital Ocean etc) o endereço poderia ser “`http://meusiteincrivel.com:5461/api`”.

Para adicionar essa funcionalidade ao seu projeto, execute o seguinte comando:

```
ng generate environments
```

```
$ ng generate environments
CREATE src/environments/environment.ts (31 bytes)
CREATE src/environments/environment.development.ts (31 bytes)
UPDATE angular.json (3133 bytes)
```

No projeto, vá até o diretório `src/environments` e veja que existem dois arquivos:

- `environment.ts` Variáveis de Ambiente no Modo de Produção
- `environment.development.ts` Variáveis de Ambiente no Modo de Desenvolvimento

O arquivo `environment.ts` tem o seguinte código (removendo os comentários):

```
export const environment = {}
```

Por meio desses dois arquivos, podemos controlar variáveis que podem mudar de valor de acordo com o servidor. No caso da API, podemos ter:

```
// src\environments\environment.ts
export const environment = {
  api: 'http://myawesomesite.com:5461/api/'
}
```

e o ambiente de desenvolvimento:

```
// src\environments\environment.development.ts

export const environment = {
  api: 'http://localhost:3000/'
}
```

Após criar a variável API (lembre-se, o Angular cuidará de escolher o arquivo correto de acordo com o servidor), podemos mudar o `CategoriesService` para:

```
// src\app\categories\category.service.ts
import {HttpClient} from '@angular/common/http'
import {Injectable} from '@angular/core'
import {environment} from '../environments/environment'

@Injectable({
  providedIn: 'root'
})
export class CategoryService {
  constructor(private http: HttpClient) {}

  public getAll() {
    return this.http.get(environment.api + 'categories')
  }
}
```

Você vê que o objeto `environment` foi importado, e `environment.api` foi usado para acessar o endereço do servidor backend.

4.10. Definindo o Tipo de Retorno da API

Estamos usando TypeScript e podemos usá-lo para definir qual será o tipo de retorno da API. Essa funcionalidade é importante para que, quando estivermos desenvolvendo o template da tela de categorias, possamos usar o intellisense.

Gere uma nova interface com o comando `ng g i categories/category dto` Uma nova interface será criada:

```
// src\app\categories\category.dto.ts
export interface Category {}
```

O “.dto” significa *Data Transfer Object* (Objeto de Transferência de Dados), a definição de um objeto puro usado para transferência de dados entre cliente e servidor. Como a categoria possui três propriedades: *id*, *nome*, *descrição*, a interface será:

```
export interface Category {  
  id: number  
  name: string  
  description: string  
}
```

4.11. Versão Final do Método GetAll()

Voltando ao CategoryService, agora podemos definir o tipo de retorno do método getAll, que será um array do tipo Category:

```
import {HttpClient} from '@angular/common/http'  
import {Injectable} from '@angular/core'  
import {Observable} from 'rxjs'  
import {environment} from 'src/environments/environment'  
import {Category} from './category.dto'  
  
@Injectable({  
  providedIn: 'root'  
)  
export class CategoryService {  
  constructor(private http: HttpClient) {}  
  
  public getAll(): Observable<Category[]> {  
    return this.http.get<Category[]>(environment.api + 'categories')  
  }  
}
```

Após o `public getAll()` adicionamos o tipo de retorno do método, que é `Observable<Category[]>`. A classe `Observable` é uma classe especial no pacote `rxjs`, amplamente usada pelo Angular ao lidar com dados assíncronos. Quando especificamos que o retorno é do tipo `Category[]`, devemos especificar esse retorno no método `http.get`, que se torna `http.get<Category[]>`. Quando definimos um tipo para os dados que o servidor retorna, fica mais fácil trabalhar com ele, como veremos a seguir:

4.12. Usando o MatTable para Exibir Categorias

Agora podemos configurar o Mat Table do componente Categorias para exibir categorias diretamente da API. Primeiro você deve alterar o dataSource da tabela, que originalmente é

representado por CategoriesDataSource:

```
@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  styles: [
    '.full-width-table {
      width: 100%;
    }

    ,
    standalone: true,
    imports: [MatTableModule, MatPaginatorModule, MatSortModule, MatCar
  }
}

export class CategoriesComponent implements AfterViewInit {
  @ViewChild(MatPaginator) paginator!: MatPaginator;
  @ViewChild(MatSort) sort!: MatSort;
  @ViewChild(MatTable) table!: MatTable<CategoriesItem>;
  dataSource = new CategoriesDataSource();

  /** Columns displayed in the table. Column IDs can be added, removed, or reordered. */
  displayedColumns = ['id', 'name'];
}
```

Na tela de categorias, vamos realizar a paginação e ordenação dos campos no cliente. Então podemos usar um Data Source genérico, representado pela classe MatTableDataSource:

```
// imports
import { Category } from './category.dto';

... code ...

export class CategoriesComponent implements AfterViewInit {
  @ViewChild(MatPaginator) paginator!: MatPaginator;
  @ViewChild(MatSort) sort!: MatSort;
  @ViewChild(MatTable) table!: MatTable<CategoriesItem>

  dataSource = new MatTableDataSource<Category>(); // <<< altere aqui

  /** Columns displayed in the table. Column IDs can be added, removed, or reordered. */
  displayedColumns = ['id', 'name'];
```

```
... code ...
```

Para chamar a api que está no CategoryService, devemos injetar a classe CategoryService, da seguinte forma:

```
// imports

// code
export class CategoriesComponent implements AfterViewInit {
  // code

  constructor(private categoryService: CategoryService) {} // << Adicione o Ser\viço aqui

  // code
}
```

Como os dados estão sendo recuperados pelo servidor, é assíncrono. A melhor maneira de garantir que o DataSource será configurado corretamente está no método ngOnInit. Vamos instanciar o dataSource, configurar o MatTable, o Sort e o Paginator:

```
// imports

// code
export class CategoriesComponent implements AfterViewInit {
  @ViewChild(MatPaginator) paginator!: MatPaginator
  @ViewChild(MatSort) sort!: MatSort
  @ViewChild(MatTable) table!: MatTable<CategoriesItem>
  dataSource = new MatTableDataSource<Category>()

  displayedColumns = ['id', 'name']

  constructor(private categoryService: CategoryService) {}

  ngAfterViewInit(): void {
    // this.dataSource.sort = this.sort;
    // this.dataSource.paginator = this.paginator;
    // this.table.dataSource = this.dataSource;
    this.loadCategories() // << Adds new method here...
  }
}
```

```
async loadCategories(): Promise<void> {
  const categories = await lastValueFrom(this.categoryService.getAll())
  this.dataSource = new MatTableDataSource(categories)
  this.table.dataSource = this.dataSource
  this.dataSource.sort = this.sort
  this.dataSource.paginator = this.paginator
}
}
```

No método `ngAfterViewInit`, chamamos a função `loadCategories`. Como essa função utiliza `async/await`, ela deve retornar `Promise<void>`. Usamos o método `lastValueFrom`, que irá recuperar os dados da chamada assíncrona `this.categoryService.getAll`. Após recuperar esses dados do servidor, podemos criar o `MatTableDataSource` com esses dados e atribuí-lo à variável `dataSource`. Com o `dataSource` devidamente preenchido, podemos configurar a tabela (variável `table`), ordenação e paginação.

O código completo final para a classe `Categories` é mostrado abaixo:

```
// src\app\categories\categories.component.ts
import {AfterViewInit, Component, ViewChild} from '@angular/core'
import {
  MatTableModule,
  MatTable,
  MatTableDataSource
} from '@angular/material/table'
import {MatPaginatorModule, MatPaginator} from '@angular/material/paginator'
import {MatSortModule, MatSort} from '@angular/material/sort'
import {CategoriesDataSource, CategoriesItem} from './categories-datasource'
import {MatCardModule} from '@angular/material/card'
import {MatButtonModule} from '@angular/material/button'
import {CategoryService} from './category.service'
import {lastValueFrom} from 'rxjs'
import {Category} from './category.dto'

@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  styles: [
    .full-width-table {
      width: 100%;
    }
  ]
})
```

```
,  
standalone: true,  
imports: [  
  MatTableModule,  
  MatPaginatorModule,  
  MatSortModule,  
  MatCardModule,  
  MatButtonModule  
]  
})  
export class CategoriesComponent implements AfterViewInit {  
  @ViewChild(MatPaginator) paginator!: MatPaginator  
  @ViewChild(MatSort) sort!: MatSort  
  @ViewChild(MatTable) table!: MatTable<CategoriesItem>  
  dataSource = new MatTableDataSource<Category>()  
  
  /** Columns displayed in the table. Column IDs can be added, removed, or reordered. */  
  displayedColumns = ['id', 'name']  
  
  constructor(private categoryService: CategoryService) {}  
  
  ngAfterViewInit(): void {  
    this.loadCategories()  
  }  
  
  async loadCategories(): Promise<void> {  
    const categories = await lastValueFrom(this.categoryService.getAll())  
    this.dataSource = new MatTableDataSource(categories)  
    this.table.dataSource = this.dataSource  
    this.dataSource.sort = this.sort  
    this.dataSource.paginator = this.paginator  
  }  
}
```

O template de Categorias não mudou e já está totalmente funcional, com paginação e ordenação funcionando perfeitamente:

```
<mat-card>
  <mat-card-title>Categorias</mat-card-title>
  <mat-card-subtitle>Listando todas as categorias</mat-card-subtitle>

  <mat-card-content>
    <table mat-table class="full-width-table" matSort aria-label="Elements">
      <!-- Id Column -->
      <ng-container matColumnDef="id">
        <th mat-header-cell *matHeaderCellDef mat-sort-header>Id</th>
        <td mat-cell *matCellDef="let row">{{ row.id }}</td>
      </ng-container>

      <!-- Name Column -->
      <ng-container matColumnDef="name">
        <th mat-header-cell *matHeaderCellDef mat-sort-header>Nome</th>
        <td mat-cell *matCellDef="let row">{{ row.name }}</td>
      </ng-container>

      <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
      <tr mat-row *matRowDef="let row; columns: displayedColumns"></tr>
    </table>

    <mat-paginator
      #
      Paginator
      [length]="dataSource?.data?.length"
      [pageIndex]="0"
      [pageSize]="10"
      [pageSizeOptions]="[5, 10, 20]"
      aria-label="Select page"
    >
    </mat-paginator>
  </mat-card-content>

  <mat-card-actions>
    <button mat-button>New Category</button>
  </mat-card-actions>
</mat-card>
```

4.13. Adicionando a Coluna Descrição

Para adicionar uma nova coluna à Mat Table, você deve alterar a variável `displayedColumns`:

```
// src\app\categories\categories.component.ts

displayedColumns = ['id', 'name', 'description']
```

Você também deve adicionar a coluna Descrição no template:

```
<!-- src\app\categories\categories.component.html -->






```

O resultado do novo campo de descrição é semelhante à seguinte figura:

Id ↑	Name	Description
2	Condiments 2	Sweet and savory sauces relishes spreads and seasonings
1	Beverages	Soft drinks coffees teas beers and ales
3	Confections	Desserts candies and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads crackers pasta and cereal
6	Meat/Poultry	Prepared meats
7	Produce	Dried fruit and bean curd
8	Seafood	Seaweed and fish
9	foo	bar

Items per page: 10 1 ~ 9 of 9 < >

New Category

4.14. Nova Categoria

Após configurar a listagem de categorias, vamos criar um formulário para adicionar uma nova categoria. Existem várias maneiras de fazer isso. Você pode criar uma nova rota ou abrir uma janela modal com o formulário, ou colocar o formulário acima da listagem de categorias.

Neste livro, cada tela será programada de uma maneira diferente, e você pode escolher a melhor forma que desejar. Como a tabela de categoria tem apenas dois campos, é aceitável criar um formulário simples acima da listagem de categorias.

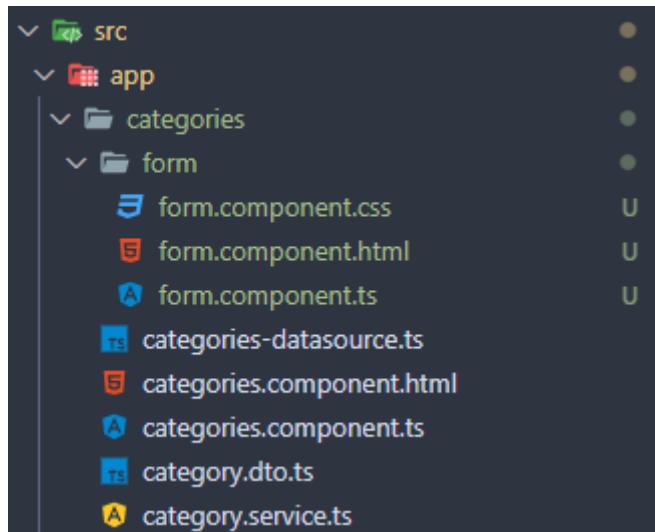
4.15. Criar um Formulário de Categoria

Primeiro, vamos criar um componente vazio chamado “”

```
ng g c --prefix 'category' --skip-tests --inline-style categories/form  
CREATE src/app/categories/form/form.component.html (3957 bytes)  
CREATE src/app/categories/form/form.component.ts (3533 bytes)
```

Renomeie a classe FormComponent para CategoryFormComponent.

O `ng g c` é um atalho para `ng generate component`. Usamos o prefixo “category” para que o componente possa ter a tag “”. O nome do componente é “categories/form”, então o componente será criado em “src/app/categories”:



Agora, podemos adicionar o formulário na Listagem de Categorias. No `categories.component.ts`, inclua o `CategoryFormComponent` na propriedade `imports`:

```
// includes
import {CategoryFormComponent} from './form/form.component'

@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  standalone: true,
  imports: [
    MatTableModule,
    MatPaginatorModule,
    MatSortModule,
    MatCardModule,
    MatButtonModule,
    CategoryFormComponent
  ]
})
export class CategoriesComponent implements AfterViewInit {
  // code
}
```

Agora, adicione o <category-form></category-form> no template:

```
<mat-card>
  <mat-card-header>
    <mat-card-title>Categories</mat-card-title>
    <mat-card-subtitle>Listing all categories</mat-card-subtitle>
  </mat-card-header>

  <category-form></category-form>

  <mat-card-content>
    <table mat-table class="full-width-table" matSort aria-label="Elements">
      <!— Id Column —>
      <ng-container matColumnDef="id">
        <th mat-header-cell *matHeaderCellDef mat-sort-header>Id</th>
```

O resultado é:

The screenshot shows a mobile application interface titled "Sales App". The main title is "Categories" and the subtitle is "Listing all categories". Below this, there is a message "form works!" followed by a yellow rectangular highlight. A table lists two categories:

Id	Name	Description
2	Condiments	Sweet and savory sauces relishes spreads and seasonings
1	Beverages	Soft drinks coffees teas beers and ales

O resultado é apenas um texto, “form works!”, pois o componente (src\app\categories\form\form.component.ts) tem em seu template apenas o seguinte conteúdo:

```
<p>form works!</p>
```

4.16. Criando um Formulário de Categoria

No Angular, existem dois tipos distintos de formulários: **Formulários Reativos** e **Formulários Orientados por Template**. A diferença básica entre eles é resumida abaixo:

	REACTIVE	TEMPLATE-DRIVEN
Setup of form model	Explicit, created in component class	Implicit, created by directives
Data model	Structured and immutable	Unstructured and mutable
Data flow	Synchronous	Asynchronous
Form validation	Functions	Directives

Ao longo da minha experiência com Angular, posso dizer com certeza que usar **Formulários Reativos** é sempre melhor, dadas as vantagens que oferece. Embora você precise escrever mais código, ele tem um melhor controle das validações do formulário.

O uso de **Formulários Orientados por Template** só é escolhido se houver campos dinâmicos no formulário, então isso se torna uma escolha melhor.

4.17. Criando um Formulário Reativo

Para criar um formulário reativo, primeiro precisamos analisar quais componentes devem ser importados para o formulário. Um Card, com um formulário tendo dois inputs (Nome e Descrição), bem como um botão. Então, já sabemos que devemos importar `MatCardModule`, `MatInputModule` e `MatButtonModule`. Como vamos criar um Formulário Reativo, também precisaremos adicionar o `ReactiveFormsModule`, da seguinte forma:

```
// src\app\categories\form\form.component.ts
import {Component} from '@angular/core'
import {ReactiveFormsModule} from '@angular/forms'
import {MatButtonModule} from '@angular/material/button'
import {MatInputModule} from '@angular/material/input'
import {MatCardModule} from '@angular/material/card'

@Component({
  selector: 'category-form',
  standalone: true,
  imports: [
    MatButtonModule,
    MatInputModule,
    ReactiveFormsModule,
    MatCardModule
  ],
  templateUrl: './form.component.html',
  styles: ``
})
export class CategoryFormComponent {}
```

Para criar um Formulário Reativo, primeiro usamos a classe `FormBuilder`, que agrupa um conjunto de campos do formulário. O `FormBuilder` é uma fábrica que ajuda a construir o “esqueleto” do formulário, incluindo campos, valor padrão e validações. Para usá-lo, você precisa injetá-lo no construtor, da seguinte forma:

```
export class CategoryFormComponent {
  constructor(private fb: FormBuilder) {}
}
```

Se preferir, você também pode injetar o `FormBuilder` desta forma:

```
export class CategoryFormComponent {
  private fb = inject(FormBuilder)
}
```

Com o `FormBuilder` pronto, representado pela variável `fb`, podemos criar os metadados do formulário usando o seguinte código:

```
// src\app\categories\form\form.component.ts
import {Component, inject} from '@angular/core'
import {FormBuilder, ReactiveFormsModule, Validators} from '@angular/forms'
import {MatButtonModule} from '@angular/material/button'
import {MatInputModule} from '@angular/material/input'
import {MatCardModule} from '@angular/material/card'

@Component({
  selector: 'category-form',
  standalone: true,
  imports: [
    MatButtonModule,
    MatInputModule,
    ReactiveFormsModule,
    MatCardModule
  ],
  templateUrl: './form.component.html',
  styles: ``
})
export class CategoryFormComponent {
  private fb = inject(FormBuilder)
  categoryForm = this.fb.group({
    id: [null],
    name: ['', Validators.required],
    description: ['', Validators.required]
  })
}
```

A categoryForm é uma variável do tipo FormGroup que representa os dados que estarão presentes no formulário. Cada campo no formulário é representado pela classe FormControl.

No template, já podemos criar um formulário, usando a tag <form> no html:

```
<!-- src\app\categories\form\form.component.html -->
<mat-card-content>
  <form [formGroup]="categoryForm" novalidate>
    <mat-form-field>
      <input matInput placeholder="Name" formControlName="name" />
    </mat-form-field>
  </form>
</mat-card-content>
```

O atributo `novalidate` é nativo do html e impede que o navegador valide os campos do formulário. Portanto, podemos usar nossas próprias validações

A propriedade `[formGroup]` aponta diretamente para a variável `categoryForm`, que já está instanciada com os campos `id`, `name` e `description`.

O primeiro `input` está relacionado ao “Nome” da Categoria, e a propriedade `formControlName` aponta diretamente para `categoryForm.name`.

4.18. Adicionando o Campo de Descrição

O campo de descrição pode ser adicionado no formulário:

```
<mat-card-content>
  <form [formGroup]="categoryForm" novalidate>
    <mat-form-field>
      <input matInput placeholder="Name" formControlName="name" />
    </mat-form-field>
    <mat-form-field>
      <input matInput placeholder="Description" formControlName="description" />
    </mat-form-field>
  </form>
</mat-card-content>
```

O resultado será:

Name	Description
Condiments 2	Sweet and savory sauces relishes spreads and seasonings
Beverages	Soft drinks coffees teas beers and ales
Confections	Desserts candies and sweet breads
Dairy Products	Cheeses
Grains/Cereals	Breads crackers pasta and cereal
Meat/Poultry	Prepared meats
Produce	Dried fruit and bean curd
Seafood	Seaweed and fish
foo	bar

Note que os campos de nome e descrição estão espaçados incorretamente. Você também pode configurar esses campos para ficarem um ao lado do outro em telas grandes, e um abaixo do outro em telas pequenas. Para fazer isso, precisamos adicionar funcionalidade responsiva à aplicação.

4.19. Criando Formulários Responsivos: o Layout CSS FlexBox

Para criar formulários que se ajustam ao tamanho da tela, usamos um Layout CSS FlexBox. Não apenas formulários, mas qualquer tipo de elemento que precisa ser responsivo pode ser gerenciado pelo CSS. Você pode estudar mais sobre FlexBox neste [link](#), será muito útil para qualquer projeto de frontend, seja com Angular ou qualquer outra tecnologia.

Para facilitar nosso projeto, criei um conjunto de classes CSS que formatarão nossos campos

automaticamente e de forma responsiva. Abra o arquivo `styles.css` e adicione o seguinte código CSS (a parte “Classes CSS FlexBox”):

```
/* You can add global styles to this file, and also import other style files */

html,
body {
    height: 100%;
}
body {
    margin: 0;
    font-family: Roboto, 'Helvetica Neue', sans-serif;
}

/* FlexBox CSS Classes */
.container {
    display: flex;
    flex-direction: row;
    gap: 10px;
}
.wrap {
    flex-wrap: wrap;
}
.center {
    align-items: center;
    justify-content: center;
}
.width-full {
    width: 100%;
}
.item-20 {
    flex: 0 0 20%;
}
.item-30 {
    flex: 0 0 30%;
}
.item-40 {
    flex: 0 0 40%;
}
.item-50 {
    flex: 0 0 50%;
}
.item-60 {
```

```
    flex: 0 0 60%;  
}  
.item-70 {  
    flex: 0 0 70%;  
}  
.item-80 {  
    flex: 0 0 80%;  
}  
.item {  
    flex: 1 1 auto;  
}  
/* tablet or mobile */  
@media (max-width: 768px) {  
.container {  
    flex-direction: column;  
    gap: 5px;  
}  
}
```

Criamos a classe “container”, que será o bloco “pai” das outras classes, e criamos a classe item-X, onde X é a porcentagem da tela que o bloco deve ocupar. A classe “item” ocupará o bloco proporcionalmente ao seu espaço. Por exemplo, se houver dois blocos com a classe “item”, cada um terá 50% do tamanho da tela. Se houver 3 blocos, 33%.

Para que os campos sejam responsivos, eles devem “quebrar” quando a tela for muito pequena (geralmente uma tela abaixo de 768 pixels). Usando o filtro CSS @media, podemos definir que abaixo de 768 pixels, os campos ficarão um abaixo do outro, através do valor column da propriedade flex-direction.

Voltando ao template `form.component.html`, podemos usar as classes css para criar o campo de nome com 30% do espaço total da tela, e o campo de descrição com 70% do espaço da tela, como no seguinte código:

```
<mat-card-content>
  <form [formGroup]="categoryForm" novalidate>
    <div class="container">
      <mat-form-field class="item-30">
        <input matInput placeholder="Name" formControlName="name" />
      </mat-form-field>
      <mat-form-field class="item-70">
        <input
          matInput
          placeholder="Description"
          formControlName="description"
        />
      </mat-form-field>
    </div>
  </form>
</mat-card-content>
```

A tag `<form>` tem um `<div>` com a classe css `container`, enquanto suas tags filhas `<mat-form-field>` têm as classes `item-30` e `item-70`. O resultado desse código é mostrado abaixo:

The screenshot shows a web browser window titled "MySalesApp" with the URL "localhost:4200/categories". The page has a blue header bar with the text "Sales App". On the left, there is a sidebar menu with options: "Home", "Categories", and "Suppliers". The main content area is titled "Categories" and displays a table with the following data:

Name	Description
Condiments 2	Sweet and savory sauces relishes spreads and seasonings
Beverages	Soft drinks coffees teas beers and ales
Confections	Desserts candies and sweet breads
Dairy Products	Cheeses
Grains/Cereals	Breads crackers pasta and cereal
Meat/Poultry	Prepared meats
Produce	Dried fruit and bean curd
Seafood	Seaweed and fish
foo	bar

Quando a tela é pequena, os campos do formulário ficam um abaixo do outro, graças ao @media no css. Veja:

The screenshot shows a web application titled "Sales App" running on "localhost:4200/cate...". The main title bar includes the application name "MySalesApp" and the URL "localhost:4200/cate...". Below the title bar, there are standard browser controls for back, forward, search, and refresh.

The main content area has a blue header bar with the text "≡ Sales App". Below this, the page title is "Categories" followed by the subtitle "Listing all categories".

There are two input fields labeled "Name" and "Description" for adding new categories.

A table lists six categories:

Id	Name	Description
2	Condiments 2	Sweet and savory sauces relishes spreads and seasonings
1	Beverages	Soft drinks coffees teas beers and ales
3	Confections	Desserts candies and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads crackers pasta and cereal
6	Meat/Poultry	Prepared meats

4.20. Validação

Adicionar validação a um campo de formulário reativo é uma tarefa muito simples. Devemos informar, por exemplo, que o campo nome é obrigatório. Ou seja, não pode ter caracteres vazios.

Para definir o campo nome, defina um validador ao criar o FormGroup:

```
// imports

// code
export class CategoryFormComponent {
  private fb = inject(FormBuilder)
  categoryForm = this.fb.group({
    id: [null],
    name: ['', [Validators.required, Validators.minLength(3)]],
    description: ['', Validators.required]
  })
}
```

Ao adicionar Validators.required, o campo de nome já exibe um erro se estiver vazio. minLength determina uma quantidade mínima de caracteres.

The screenshot shows a web application titled "Sales App" running on localhost:4200. The title bar includes the application name "MySalesApp" and the URL "localhost:4200/cate...". The main content area has a blue header with the title "Categories" and a subtitle "Listing all categories". Below this, there are two input fields: "Name" and "Description", both currently empty. A table follows, displaying a list of categories with columns for Id, Name, and Description.

Id	Name	Description
2	Condiments 2	Sweet and savory sauces relishes spreads and seasonings
1	Beverages	Soft drinks coffees teas beers and ales
3	Confections	Desserts candies and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads crackers pasta and cereal
6	Meat/Poultry	Prepared meats

4.21. Configurando Mensagens de Erro `{/examples/}`

Você pode usar o componente para determinar mensagens de erro personalizadas para a entrada, com a expressão `@if`, usada para determinar se a variável possui o valor de erro:

```
<mat-card-content>
  <form [formGroup]="categoryForm" novalidate>
    <div class="container">
      <mat-form-field class="item-30">
        <input matInput placeholder="Name" formControlName="name" />
        @if (categoryForm.controls['name'].hasError('required')) {
          <mat-error>Name is <strong>required</strong></mat-error>
        } @if (categoryForm.controls['name'].hasError('minlength')) {
          <mat-error>Name is too short</mat-error>
        }
      </mat-form-field>
      <mat-form-field class="item-70">
        <input
          matInput
          placeholder="Description"
          formControlName="description"
        />
        @if (categoryForm.controls['description'].hasError('required')) {
          <mat-error>Description is <strong>required</strong></mat-error>
        }
      </mat-form-field>
    </div>
  </form>
</mat-card-content>
```

Dois foram definidos, onde o primeiro é exibido se o usuário não inserir um valor (obrigatório) e o outro se eles inserirem menos de 3 caracteres, definido por `Validators.minLength(3)`.

Esta é a primeira vez que vemos a diretiva `@if`. Esta diretiva permite exibir ou não o componente de acordo com a expressão avaliada. Neste caso, `@if` só será exibido se `hasError()` retornar verdadeiro.

A seguinte imagem mostra os erros nos campos de nome e descrição:

The screenshot shows a web application titled "Sales App" with a blue header bar. Below it, a section titled "Categories" displays a table of categories. The table has columns for "Id", "Name", and "Description". The first row, which has an ID of 2, contains the value "Condiments 2" in the Name column and "Sweet and savory sauces relishes spreads and seasonings" in the Description column. The second row, with ID 1, contains "Beverages" and "Soft drinks coffees teas beers and ales". The third row, with ID 3, contains "Confections" and "Desserts candies and sweet breads". The fourth row, with ID 4, contains "Dairy Products" and "Cheeses". The fifth row, with ID 5, contains "Grains/Cereals" and "Breads crackers pasta and cereal". The sixth row, with ID 6, contains "Meat/Poultry" and "Prepared meats". The seventh row, with ID 7, contains "Produce" and "Dried fruit and bean curd". The eighth row, with ID 8, contains "Seafood" and "Seaweed and fish". The ninth row, with ID 9, contains "foo" and "bar". Above the table, there are two input fields: one for "Name" containing "xX" and another for "Description" containing an empty string. Both fields have red validation messages: "Name is too short" for the Name field and "Description is required" for the Description field.

Id	Name	Description
2	Condiments 2	Sweet and savory sauces relishes spreads and seasonings
1	Beverages	Soft drinks coffees teas beers and ales
3	Confections	Desserts candies and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads crackers pasta and cereal
6	Meat/Poultry	Prepared meats
7	Produce	Dried fruit and bean curd
8	Seafood	Seaweed and fish
9	foo	bar

4.22. Enviar Formulário {/examples/}

Vamos adicionar um botão para enviar o formulário. Ele executará o método `onSubmit`, que será definido no componente:

...

```
<form [formGroup]="categoryForm" (ngSubmit)="onSubmit()" novalidate>
  <div class="container">
    <!-- ... fields... -->
  </div>

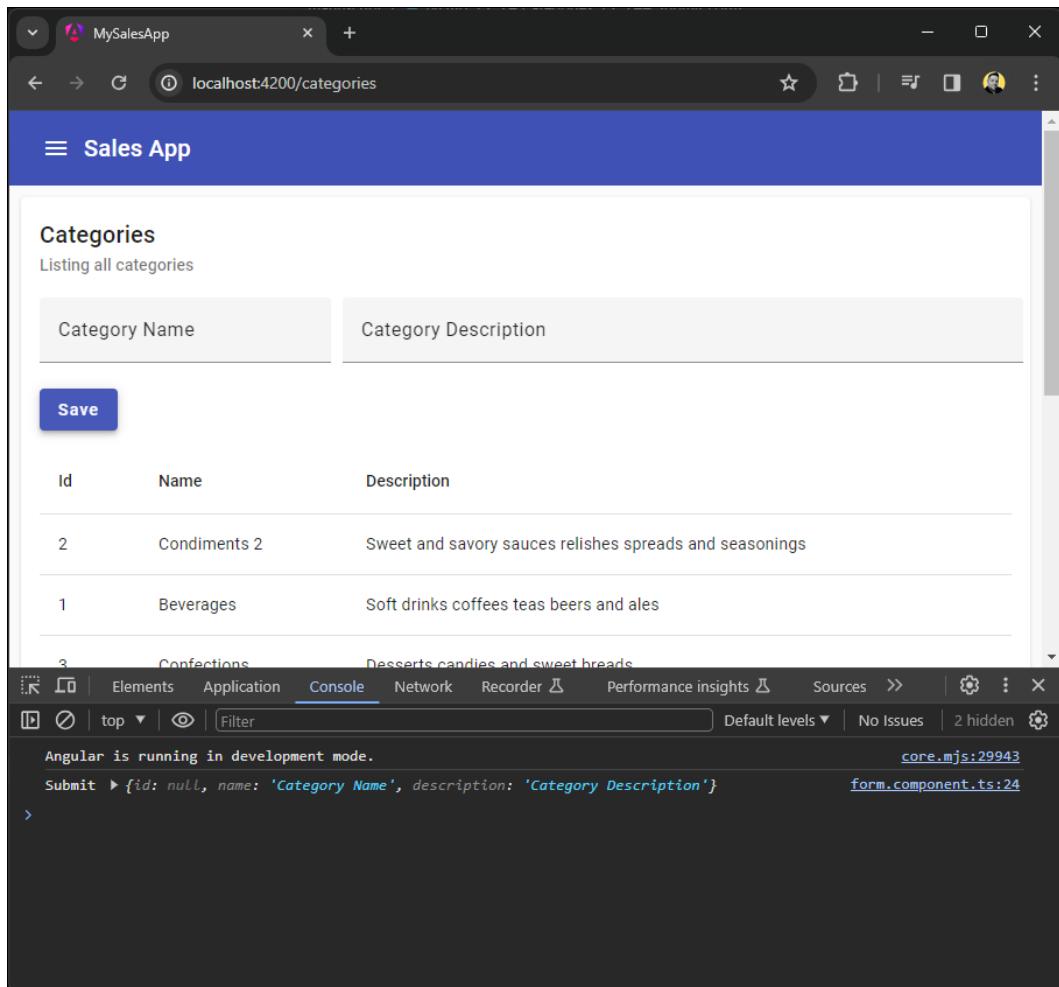
  <button
    mat-raised-button
    color="primary"
    type="submit"
    [disabled]="!categoryForm.valid"
  >
    Save
  </button>
  ...
</form>
```

No elemento `<form>`, especificamos o evento `(ngSubmit)="onSubmit()"`, que irá executar o método `onSubmit` quando o formulário for enviado. O botão `submit` possui uma diretiva `[disabled]` que desabilita automaticamente o botão se o formulário estiver inválido.

O código para o método `onSubmit` é mostrado abaixo:

```
onSubmit(){
  console.log('Submit', this.categoryForm.value)
}
```

Por enquanto, estamos apenas exibindo os dados do formulário no Console do navegador web (F12). Os dados do formulário são acessados através da variável `this.categoryForm.value`:



4.23. Revisando Alguns Padrões do Angular

Até agora, cobrimos algumas diretivas enquanto criávamos o formulário de categoria. Vamos revisar algumas delas:

- O `@if` é uma declaração de bloco que inclui ou não inclui o componente de acordo com a expressão avaliada.
- Quando usamos `[disabled]="!categoryForm.valid"`, o “`[...]`” indica que estamos

definindo uma propriedade dinâmica, que pode mudar de acordo com a expressão avaliada.

- Quando usamos `(ngSubmit)="onSubmit()"` estamos definindo que o evento `ngSubmit` executará o método `onSubmit`.

4.24. Controlando a Visibilidade do Formulário

Para controlar a visibilidade do formulário, podemos ter uma variável no componente Categorias (e não no formulário), que definirá se o formulário deve ser exibido ou não:

Com a variável definida, podemos usar `@if` para exibir ou não o formulário:

```
<!-- src\app\categories\categories.component.html -->

<mat-card>
  <mat-card-header>
    <mat-card-title>Categories</mat-card-title>
    <mat-card-subtitle>Listing all categories</mat-card-subtitle>
  </mat-card-header>

  @if (showForm) {
    <category-form></category-form>
  }

  <mat-card-content></mat-card-content>
</mat-card>
```

The New Category button will have the functionality to display the form. Let's add the (click) event and call the onNewCategoryClick method.

```
<!-- src\app\categories\categories.component.html -->  
<!-- code -->  
  
<mat-card-actions>  
  <button mat-button (click)="onNewCategoryClick">New Category</button>  
</mat-card-actions>  
  
<!-- code -->  
  
// src\app\categories\categories.component.ts  
  
... code ...  
  
export class CategoriesComponent implements OnInit {  
  
  ... code ...  
  
  onNewCategoryClick(){  
    this.showForm = true;  
  }  
}
```

O formulário só aparecerá na tela se o usuário clicar no botão. Também podemos esconder a listagem de categorias, usando o @else:

```
app > categories > categories.component.html > mat-card > mat-card-actions > button
    You, 12 seconds ago | 1 author (You)
1  <mat-card>
2    <mat-card-header>
3      <mat-card-title>Categories</mat-card-title>
4      <mat-card-subtitle>Listing all categories</mat-card-subtitle>
5    </mat-card-header>
6
7    @if (showForm) {
8      <category-form></category-form>
9    } @else {
10      <mat-card-content>
11        <table mat-table class="full-width-table" matSort aria-label="Table listing all categories">
12          <!-- Id Column -->
13          <ng-container matColumnDef="id">
14            <th mat-header-cell *matHeaderCellDef mat-sort-header>Id</th>
15            <td mat-cell *matCellDef="let row">{{ row.id }}</td>
16          </ng-container>
17
18          <!-- Name Column -->
19          <ng-container matColumnDef="name">
20            <th mat-header-cell *matHeaderCellDef mat-sort-header>Name</th>
21            <td mat-cell *matCellDef="let row">{{ row.name }}</td>
22          </ng-container>
23        </table>
24      </mat-card-content>
25    </div>
26  </div>
27
```

4.25. Criando um Botão de Voltar no Formulário

Você pode criar um botão de voltar que atribuirá o valor `false` à variável `showForm`:

```
<mat-card-content>
  ...
<form>... fields ... save button ...</form>

<mat-card-actions>
  <button type="button" mat-button color="primary" (click)="onBack()">
    Back
  </button>
</mat-card-actions>
</mat-card-content>
```

O método `onBack` pode ser definido da seguinte forma:

```
  styles:
})
export class CategoryFormComponent {
  private fb = inject(FormBuilder);
  categoryForm = this.fb.group({
    id: [null],
    name: [null, [Validators.required, Validators.minLength(3)]],
    description: [null, Validators.required]
  })

  onSubmit() {
    console.log('Submit', this.categoryForm.value)
  }

  onBack() {
    this.showForm    Property 'showForm' does not exist on type 'CategoryFormComponent'.
  }
}
```

A variável `showForm` não é definida no componente `FormCategoryComponent`, ela é definida no componente `Categorias`. A variável que precisamos acessar está definida no componente pai `Form`. Para um componente filho acessar o componente pai, precisamos usar um **Evento**, e lançar o evento no componente `Form` para que o componente `Categorias` possa capturá-lo.

4.26. Vinculação de Evento

Já usamos eventos um pouco no Angular, especialmente o evento de botão (`click`). Agora vamos criar um evento personalizado, chamado `back`. Para criar um evento no componente `FormCategoryComponent`, usamos o decorador `@Output()`:

```
export class FormComponentComponent implements OnInit {
  ...
  ...
  @Output() back = new EventEmitter();
  ...
  ...
  onBack(){
    this.back.emit();
  }
}
```

```
}
```

Agora o Componente de Formulário de Categorias tem um evento chamado back. A única tarefa que o botão back faz é executar o método onBack, que irá disparar o evento back. O componente Categorias precisa “capturar” esse evento. Então, no template do componente Categorias, crie uma referência ao evento da seguinte forma:

```
<mat-card>
  @if (showForm) {
    <category-form (back)="hideCategoryForm()"></category-form>
  } @else { ... code ... }
</mat-card>
```

Todos os eventos em Angular são referenciados pelos parênteses “(nome do evento)”. Então, o evento back é definido como: (back)="hideCategoryForm()". O método hideCategoryForm é definido da seguinte forma:

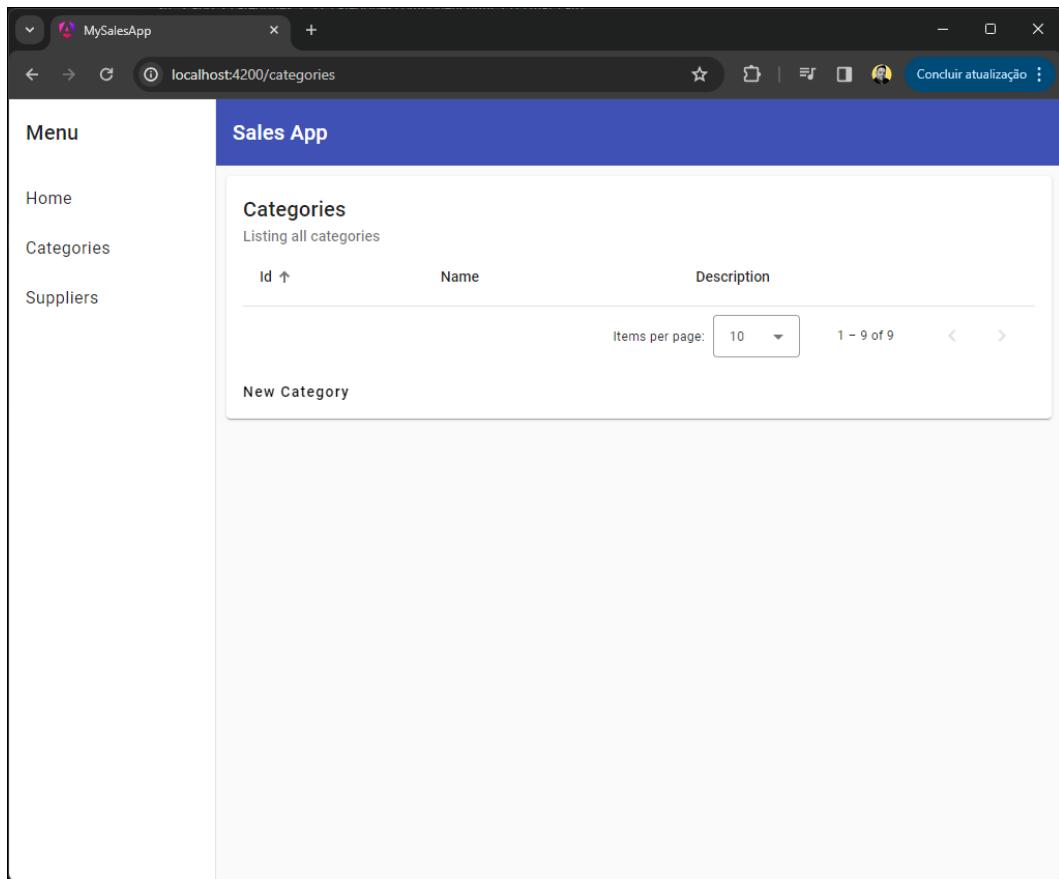
```
// imports...

export class CategoriesComponent implements OnInit {
  // ... code ...

  hideCategoryForm() {
    this.showForm = false
  }
}
```

O método hideCategoryForm mudará o valor da variável showForm para false. Assim, o @if tanto do formulário quanto da listagem de categorias será alterado.

Ao testar a aplicação, após clicar no botão voltar, obtemos o seguinte resultado:



Aqui temos um problema, porque o Mat Grid não exibe os resultados depois que é exibido novamente. Precisamos recarregar os dados e podemos fazer isso no próprio método `hideCategoryForm`:

```
// ... imports ...

export class CategoriesComponent implements OnInit {
  // ... code ...

  hideCategoryForm() {
    this.showForm = false
    this.loadCategories()
  }
}
```

4.27. Passando Dados do Formulário Através de Eventos

Agora, podemos salvar uma nova categoria. Primeiro devemos obter os dados do formulário e, através de um evento chamado “save” (o nome do evento pode ser outro, se necessário), passaremos os dados do formulário para o componente Categorias, que chamará o serviço e realizará a “inserção”.

No CategoryFormComponent, crie o evento save e use-o no método onSubmit que é executado quando o formulário é submetido.

```
// ... imports ...

// ... code ...

export class CategoryFormComponent implements OnInit {
  // ... code ...

  @Output() save = new EventEmitter()

  onSubmit() {
    console.log('Botão salvar clicado no CategoryFormComponent')
    this.save.emit(this.categoryForm.value)
  }

  // ... code ...
}
```

Neste código, criamos o evento save e atribuímos um tipo ao EventEmitter. No método onSubmit, disparamos o evento e passamos a variável this.categoryForm.value, que é um objeto contendo os dados do formulário.

No componente categories.component.html, definimos o evento de salvar da seguinte forma:

```
@if (showForm) {
<category-form (back)="onBackForm()" (save)="onSave($event)"></category-form>
}
```

O evento (save) agora tem um parâmetro, que é \$event. Você deve usar \$event para passar

automaticamente dados de um componente para outro. O método onSave não precisa de \$event, você pode simplesmente passar o objeto que contém os dados do formulário:

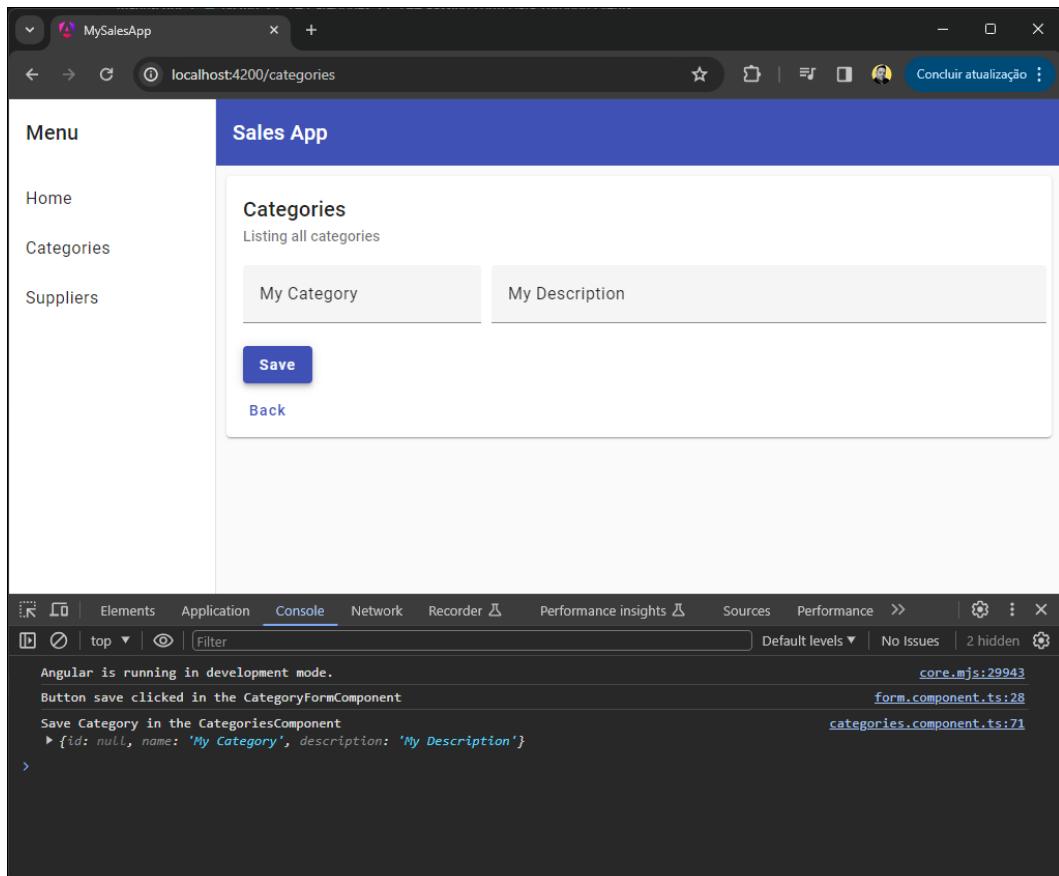
```
// src\app\categories\categories.component.ts

// ... imports ...

export class CategoriesComponent implements OnInit {
  // ... code ...

  onSave(category: Category) {
    console.log('Save Category in the CategoriesComponent', category)
  }
}
```

O resultado é mostrado abaixo:



4.28. Conversão de Tipo

Podemos criar um EventEmmiter com um certo tipo, por exemplo:

```
import {Category} from './category.dto'

export class CategoryFormComponent {
  @Output() save = new EventEmitter<Category>()
  // code
}
```

Neste caso, ao disparar o evento `this.save.emit`, é necessário realizar uma conversão de tipo, da seguinte forma:

```
export class CategoryFormComponent {
  @Output() save = new EventEmitter<Category>()

  // ...code...

  onSubmit() {
    console.log('Botão salvar clicado no CategoryFormComponent')
    this.save.emit(this.categoryForm.value as Category)
  }
}
```

Usamos o operador “as” para realizar essa conversão, também chamada de “typecast”.

Se você receber um erro nesta conversão, relacionado ao campo id sendo nulo, abra o arquivo tsconfig.json e adicione a seguinte configuração: "strictNullChecks": false em “compilerOptions”.

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "strictNullChecks": false, <<<<<<<aqui
    "outDir": "./dist/out-tsc",
    ...
  }
}
```

4.29. Salvando a Categoria

Para salvar os dados, precisamos saber se estamos inserindo dados ou atualizando dados. Através do campo id podemos obter essa informação. Se o id estiver preenchido, estamos atualizando os dados. Caso contrário, estamos inserindo.

Crie o método “save” no CategoryService:

```
// src\app\categories\category.service.ts
import {HttpClient} from '@angular/common/http'
import {Injectable} from '@angular/core'
import {Observable} from 'rxjs'
import {environment} from 'src/environments/environment'
import {Category} from './category.dto'

@Injectable({
  providedIn: 'root'
})
export class CategoryService {
  constructor(private http: HttpClient) {}

  public getAll(): Observable<Category[]> {
    return this.http.get<Category[]>(environment.api + 'categories')
  }

  public save(category: Category): Observable<Category> {
    if (category.id)
      return this.http.put<Category>(
        environment.api + 'categories/' + category.id,
        category
      )

    return this.http.post<Category>(environment.api + 'categories', category)
  }
}
```

O método `save` verifica se `category.id` está preenchido, e se estiver, uma requisição PUT é feita para a API, passando o objeto categoria como segundo parâmetro. Se `category.id` não estiver preenchido, uma requisição POST é feita para a API.

Voltando ao método `onSave` de `categories.component.ts`, podemos chamar a api da seguinte forma:

```
// ... imports ...

export class CategoriesComponent implements OnInit {
  // ... code ...

  async onSave(category: Category) {
    const saved = lastValueFrom(this.categoryService.save(category))
    console.log('Saved', saved)
    this.hideCategoryForm()
  }
}
```

Agora o método onSave é async/await. A constante criada saved será a resposta do servidor tipada para Category. Se for uma nova categoria, ela até incluirá o id.

Id	Name	Description
2	Condiments 2	Sweet and savory sauces relishes spreads and seasonings
1	Beverages	Soft drinks coffees teas beers and ales
3	Confections	Desserts candies and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads crackers pasta and cereal

```
Angular is running in development mode. core.mjs:29943
Button save clicked in the CategoryFormComponent form.component.ts:28
Saved > {id: 11, name: 'My Category', description: 'My Description'} categories.component.ts:72
>
```

4.30. Editando a Categoria

Para editar uma categoria, precisamos passar os dados da categoria (id, nome, descrição) para o formulário. Na listagem de categorias, vamos adicionar uma nova coluna chamada `actions`, e um botão para editar a categoria.

```
// src\app\categories\categories.component.ts

// ... code ...
export class CategoriesComponent implements OnInit {
  // ... code ...

  /** Columns displayed in the table.
   * Column IDs can be added, removed, or reordered. */
  displayedColumns = ['id', 'name', 'description', 'actions']

  // ... code ...

  onEditCategoryClick(category: Category) {
    console.log('edit category', category)
  }
}
```

No template html categories.component.html, adicione uma coluna extra à tabela com um botão de edição:

```
<mat-card>

  <!-- CODE -->
@if (showForm)

} @else {
<mat-card-content>
<mat-card-content>
  <table mat-table class="full-width-table" matSort aria-label="Elementos">

    <!-- COLUNAS -->

    <!-- Coluna de Ações -->
    <ng-container matColumnDef="ações">
      <th mat-header-cell *matHeaderCellDef mat-sort-header></th>
      <td mat-cell *matCellDef="let row">
        <button
          mat-button
          (click)="onEditCategoryClick(row)">
          Edit</button></td>
    </ng-container>
```

```
<tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
<tr mat-row *matRowDef="let row; columns: displayedColumns"></tr>
</table>

<!-- CODE -->

</mat-card>
}
```

O código adiciona um botão à lista, similar à seguinte imagem.

Id	Name	Description	Edit
2	Condiments 2	Sweet and savory sauces relishes spreads and seasonings	Edit
1	Beverages	Soft drinks coffees teas beers and ales	Edit
3	Confections	Desserts candies and sweet breads	Edit
4	Dairy Products	Cheeses	Edit
5	Grains/Cereals	Breads crackers pasta and cereal	Edit
6	Meat/Poultry	Prepared meats	Edit

O método `onEditCategoryClick` deve preencher os dados do formulário e exibi-lo. Para fazer isso, precisamos passar a variável `category` para o componente do formulário. Para fazer isso, precisamos criar uma variável no componente do formulário que aceite ser passada pela tag `. Isso é feito através do decorador @Input(), veja:`

```
// ... code ...

export class CategoryFormComponent implements OnInit {
  // ... code ...

  private fb = inject(FormBuilder)
  categoryForm = this.fb.group({
    id: [null],
    name: ['', [Validators.required, Validators.minLength(3)]],
    description: ['', Validators.required]
  })

  @Input()
  set category(category: Category) {
    this.categoryForm.setValue(category)
  }

  // ... code ...
}
```

A combinação de `@Input()` e `set category` executará o método sempre que houver mudanças na propriedade `category` do componente Form.

No componente Categorias, também criamos a variável `category`, da seguinte forma:

```
// src\app\categories\categories.component.ts

// ... code ...

export class CategoriesComponent implements OnInit {
  // ... code ...

  category!: Category

  // ... code ...
}
```

Esta variável será vinculada ao componente , da seguinte forma:

```
// src\app\categories\categories.component.html

<mat-card>
  <mat-card-title>Categories</mat-card-title>
  <mat-card-subtitle>Listing all categories</mat-card-subtitle>

  @if (showForm) {
    <category-form
      (back)="onBackForm()"
      (save)="onSave($event)"
      [category]="category"
    ></category-form>

    <!-- code -->
  </mat-card>
```

Dessa forma, a variável `category` no componente `Categories` será passada para a variável `category` no componente `Form`. Quando o botão `Editar` for clicado, passaremos o valor para a variável `category` no componente do formulário, e o método `setCategory` será executado, onde a atualização do formulário será feita através de `this.categoryForm.setValue(category)`.

Ao clicar no botão “`Editar`”, o formulário será preenchido, e o usuário poderá fazer a mudança dos dados. Como o método “`save`” já possui o código para atualizar a categoria, e o serviço também está pronto, o usuário agora pode atualizar os dados.

4.31. Corrigir um Pequeno Bug

Quando o usuário clica para editar uma categoria, clica em voltar e clica em nova categoria, os dados da categoria antiga estão presentes no formulário. Para resolver esse bug, devemos garantir que, ao clicar no botão “`Nova Categoria`”, a variável `category` no formulário esteja vazia.

```
// src\app\categories\categories.component.ts

// imports...

// code

export class CategoriesComponent implements OnInit {
  // code

  onNewCategoryClick() {
    this.category = {
      id: 0,
      name: '',
      description: ''
    }
    this.showForm = true
  }

  // code
}

//
```

4.32. Excluindo uma Categoria

Para remover uma categoria, adicione um novo botão ao lado do botão “editar”. Desta vez, usaremos um ícone no botão:

```
<!-- src\app\categories\categories.component.html -->

<!-- Actions Column -->
<ng-container matColumnDef="actions">
  <th mat-header-cell *matHeaderCellDef mat-sort-header></th>
  <td mat-cell *matCellDef="let row">
    <button mat-button (click)="onEditCategoryClick(row)">Edit</button>
    <button mat-icon-button color="warn" (click)="onDeleteCategoryClick(row)">
      <mat-icon>delete</mat-icon>
    </button>
  </td>
</ng-container>
```

To use <mat-icon> you need to add MatIconModule to the imports of the component:

```
// imports...
import {MatIconModule} from '@angular/material/icon'

@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  styles: [
    .full-width-table {
      width: 100%;
    }
  ],
  standalone: true,
  imports: [
    MatTableModule,
    MatPaginatorModule,
    MatSortModule,
    MatCardModule,
    MatButtonModule,
    MatIconModule, // Adds MatIconModule
    CategoryFormComponent
  ]
})
export class CategoriesComponent implements AfterViewInit {
  // code
}
```

O método `onDeleteCategoryClick` deve perguntar ao usuário se ele realmente deseja remover a categoria. Para fazer isso, primeiro usaremos o componente `confirm` do navegador:

```
// src\app\categories\categories.component.ts

// code

async onDeleteCategoryClick(category:Category) {
  if (confirm(`Deletar "${category.name}" com id ${category.id} ?`)) {
    await lastValueFrom(this.categoryService.delete(category.id))
    this.loadCategories();
  }
}

// code
```

O método `categoryService.delete` é mostrado abaixo:

```
// src\app\categories\category.service.ts

// ... code

public delete(id: número) {
  return this.http.delete(environment.api + 'categories/' + id);
}

// ... code ...
```

Após remover a categoria, usamos o método `loadCategories` para recarregar a listagem de categorias.

4.33. O Que Aprendemos Neste Capítulo

Neste capítulo, criamos uma tela de categoria, com 3 campos. Mas abordamos vários tópicos que serão utilizados até o final do livro. Vamos resumir cada um deles abaixo:

- **CSS** Quando abordamos o uso do Material Card (), vimos a necessidade de criar um estilo global com algumas classes css para posicionamento “m-10”, significando “margin: 10px”, por exemplo.
- **HttpModule** Para obter dados da API, aprendemos como criar serviços e importar o `HttpModule`. Criamos o `CategoryService` e os métodos `getAll`, `save` e `delete`. A maioria dos métodos em um serviço retorna um `Observable`. Então, criamos uma

interface chamada “DTO”, `category.dto.ts`, que representa a estrutura dos dados JSON que a API retorna. E com essa classe, podemos retornar um array tipado de categorias.

- **Variáveis de ambiente** Estas são usadas quando precisamos armazenar diferentes informações a respeito do servidor de desenvolvimento e do servidor de produção. No Angular, as variáveis de ambiente estão no diretório `src/environments`.
- **Material Table** Configuramos o `mat-table` para obter dados do Serviço. É importante lembrar que acessar a api e atribuir os dados do datasource deve ser feito no método `ngOnInit`, quando o `mat-table` está devidamente criado.
- **Componentes** Aprendemos como criar componentes e passar dados do componente pai para o componente filho através de `@Input()`, e passar dados do componente filho para o componente pai através de `@Output()`.
- **@if** Aprendemos como usar o `@if` para ocultar um componente específico, de modo que o componente de formulário alterne com o componente de listagem.

4.34. Diferenças do Angular 14..15..16

Caso você tenha comprado este livro antes do Angular 17, no ano de 2023, e chegou até aqui novamente (parabéns!), podemos destacar as principais mudanças do Angular anterior para a versão 17:

- Em vez de usar `*ngIf`, agora usamos `@if`
- Podemos importar componentes diretamente, em vez de usar módulos
- Nas chamadas ao serviço, usamos `lastValueFrom` em vez de `subscribe`
- O Angular 17 também utiliza a implementação do Material Design 3.0 (Angular 14 usava a versão 2)

5. Categorias de Refatoração

No capítulo anterior, criamos a tela de categorias. Neste capítulo, vamos refatorar algumas partes desta tela para torná-la mais funcional.

5.1. Adicionando Carregamento Durante a Solicitação ao Servidor

É necessário exibir informações ao usuário quando o angular está realizando uma solicitação ao servidor, já que o acesso ao backend é assíncrono e pode levar alguns segundos.

Podemos criar um componente chamado “loading-bar”, que conterá a barra de carregamento:

```
$ ng g c loading-bar --flat --inline-style --inline-template --skip-tests --dry-run
CREATE src/app/loading-bar.component.ts (279 bytes)
```

NOTE: The "dryRun" flag means no changes were made.

Neste comando para criar um componente, usamos vários parâmetros para determinar como o componente será criado. O parâmetro `--dry-run` é usado para verificar se tudo foi criado no lugar correto.

Os outros parâmetros são:

- `flat` Cria os novos arquivos no nível superior do projeto atual.
- `inline-style` Inclui estilos inline no arquivo component.ts.
- `inline-template` Inclui o template inline no arquivo component.ts.
- `skip-tests` Não cria “spec.ts”

Para realmente criar o componente, execute novamente o comando sem o parâmetro `dry-run`. O componente gerado é mostrado abaixo:

```
// src\app\loading-bar.component.ts

import {Component} from '@angular/core'

@Component({
  selector: 'app-loading-bar',
  standalone: true,
  imports: [],
  template: `<p>loading-bar works!</p> `,
  styles: ``
})
export class LoadingBarComponent {}
```

Este componente precisa de duas funcionalidades. A primeira é exibir a Barra de Carregamento Mat e a segunda funcionalidade é controlar se esta Barra de Carregamento Mat deve ou não aparecer na aplicação.

Vamos criar uma variável chamada `visible` que será um `@input()`, para que possa ser controlada como um parâmetro no componente:

```
// src\app\loading-bar.component.ts
import {Component, Input} from '@angular/core'
import {MatProgressBarModule} from '@angular/material/progress-bar'

@Component({
  selector: 'loading-bar',
  standalone: true,
  imports: [MatProgressBarModule],
  template: `
    @if (visible) {
      <mat-progress-bar color="primary" mode="indeterminate"> </mat-progress-bar>
    }
  `,
  styles: ```
})
export class LoadingBarComponent {
  @Input() visible: Boolean = true
}
```

Mudamos o selector de app-loading-bar para loading-bar

A variável @Input() visible é usada no @if que contém o mat-progress-bar.

Para usar este componente, no arquivo categories.component.ts podemos adicionar o LoadingBarComponent aos imports:

```
//imports
import {LoadingBarComponent} from './loading-bar.component'

@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  standalone: true,
  imports: [
    MatTableModule,
    MatPaginatorModule,
    MatSortModule,
    MatCardModule,
    MatButtonModule,
    MatIconModule,
    CategoryFormComponent,
    LoadingBarComponent // adding the <loading-bar>
  ]
})
export class CategoriesComponent implements AfterViewInit {
  // code
}
```

No template, vamos adicionar o componente:

```
<!-- src\app\categories\categories.component.html -->
<mat-card>
  <mat-card-title>Categories</mat-card-title>
  <mat-card-subtitle>Listing all categories</mat-card-subtitle>

  <loading-bar></loading-bar>

  ... continue ...
</mat-card>
```

Incluímos o <loading-bar>:

ID	Name	Description	Edit	Delete
2	Condiments	Sweet and savory sauces relishes spreads and seasonings	Edit	Delete
1	Beverages	Soft drinks coffees teas beers and ales	Edit	Delete
3	Confections	Desserts candies and sweet breads	Edit	Delete
4	Dairy Products	Cheeses	Edit	Delete
5	Grains/Cereals	Breads crackers pasta and cereal	Edit	Delete
6	Meat/Poultry	Prepared meats	Edit	Delete
7	Produce	Dried fruit and bean curd	Edit	Delete
8	Seafood	Seaweed and fish	Edit	Delete

New Category

Agora precisamos controlar a exibição do componente `<loading-bar>`. Para fazer isso, criaremos a variável `showLoading` e a usaremos ao buscar dados do servidor:

```
<!-- src\app\categories\categories.component.html -->
<mat-card>
  <mat-card-title>Categories</mat-card-title>
  <mat-card-subtitle>Listing all categories</mat-card-subtitle>

  <loading-bar [visible]="showLoading"></loading-bar>

  ... continue ...
</mat-card>
```

e:

```
// src\app\categories\categories.component.ts

// ... imports ...

@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  standalone: true,
  imports: [
    MatTableModule,
    MatPaginatorModule,
    MatSortModule,
    MatCardModule,
    MatButtonModule,
    MatIconModule,
    CategoryFormComponent,
    LoadingBarComponent
  ]
})
export class CategoriesComponent implements OnInit {
  // ... code ...

  showLoading: Boolean = false

  // ... code ...

  async loadCategories(): Promise<void> {
```

```
        this.showLoading = true
    const categories = await lastValueFrom(this.categoryService.getAll())
    this.dataSource = new MatTableDataSource(categories)
    this.table.dataSource = this.dataSource
    this.dataSource.sort = this.sort
    this.dataSource.paginator = this.paginator
    this.showLoading = false
}

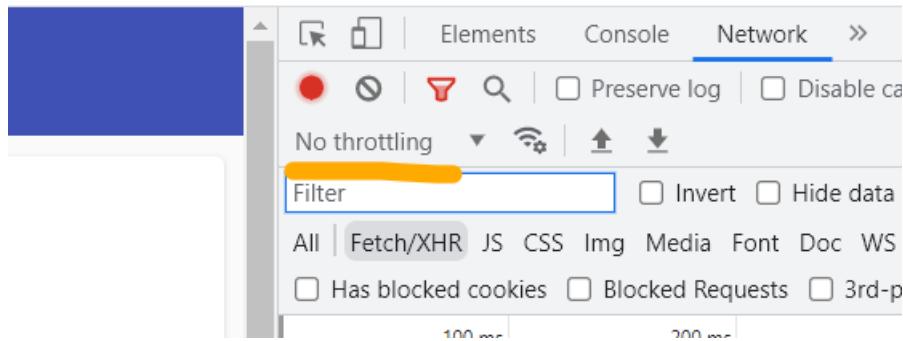
async onDeleteCategoryClick(category: Category) {
    if (confirm(`Delete "${category.name}" with id ${category.id} ?`)) {
        this.showLoading = true
        await lastValueFrom(this.categoryService.delete(category.id))
        this.showLoading = false
        this.loadCategories()
    }
}
}
```

O método `loadCategories` começa definindo `this.showLoading=true`, assim o carregamento será mostrado na tela. Após consultar o backend e obter os dados da categoria, as informações do `dataSource` são preenchidas. O método `onDeleteCategoryClick` também implementa `showLoading`.

5.2. Como Ver o Carregamento Funcionando

Se o carregamento for muito rápido, quase piscando na tela, podemos usar um recurso do Google Chrome para desacelerar o carregamento e, assim, ver o carregamento funcionando.

Pressione F12 para abrir o Dev Tools e vá para a aba Rede. Sob um botão vermelho redondo está o recurso de throttling, conforme mostrado na seguinte imagem:



Escolha 3g Fast ou 3g Slow para ver o carregamento em ação.

5.3. Pular Testes e Criação de Arquivo Css no Arquivo de Configuração Angular.json

Você pode configurar o arquivo `angular.json` na seção de projetos para adicionar um esquemático único para a geração de componentes:

```
{  
  ...  
  "projects": {  
    "my-sales-app": {  
      "projectType": "application",  
      "schematics": {  
        "@schematics/angular:application": {  
          "strict": true  
        },  
        "@schematics/angular:component": {  
          "inlineStyle": true,  
          "skipTests": true  
        }  
      },  
      ...  
    }  
  }  
}
```

Neste exemplo, `@schematics/angular:component` configurará como os componentes são criados. No exemplo, o CSS será criado “inline”, ou seja, o arquivo css não será criado, e o arquivo de teste também será omitido. Por exemplo:

```
$ ng g c foo --dry-run  
CREATE src/app/foo/foo.component.html (18 bytes)  
CREATE src/app/foo/foo.component.ts (242 bytes)
```

NOTE: The "dryRun" flag means no changes were made.

5.4. Hora de Fazer o Deploy! (opcional)

Na seção 2.6 adicionamos o projeto ao github, e inicialmente espero que você esteja commitando e fazendo push do seu projeto no github. Agora vamos aprender como fazer o deploy deste projeto para um recurso chamado “Github Pages”.

Para fazer isso, usaremos uma extensão do Angular, chamada “angular-cli-ghpages”. O primeiro passo é instalá-la no projeto da seguinte forma:

```
~/my-sales-app/ $ ng add angular-cli-ghpages
```

Após a instalação, simplesmente execute o seguinte comando:

```
ng deploy --base-href=/<repositoryname>/
```

O <repositoryname> é o nome do seu repositório, provavelmente será my-sales-app. Após o processo ser concluído, você verá uma mensagem “tenha um bom dia!” e poderá acessar o seguinte endereço:

<https://my-sales-app.github.io/>

```
~ $ cd my-sales-aapp/
~/my-sales-aapp (main) $ ng deploy --base-href=/my-sales-aapp/
[ Building "my-sales-app"
[ Build target "my-sales-app:build:production"
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files      | Names          | Raw Size | Estimated Transfer Size
main.5446f41fff02dbb5.js | main           | 729.53 kB | 156.96 kB
styles.06fd2d1babea151d.css | styles         | 75.42 kB | 7.92 kB
polyfills.7d00d561fbf9db47.js | polyfills     | 36.23 kB | 11.48 kB
runtime.ff6ce692ac3621f9.js | runtime        | 1.05 kB | 594 bytes

| Initial Total | 842.23 kB | 176.94 kB

Build at: 2022-03-12T14:51:14.834Z - Hash: 07cce609b1ca27fb - Time: 28817ms

Warning: bundle initial exceeded maximum budget. Budget 500.00 kB was not met by 342

🚀 Uploading via git, please wait...
★ Successfully published via angular-cli-ghpages! Have a nice day!
~/my-sales-aapp (main) $
```

Você verá o projeto na tela de inicialização. Quando você clicar em Categorias, possivelmente receberá um erro, porque ainda não configuramos o backend no modo de produção. O projeto fake-server pode ser usado, por exemplo, em um servidor como Heroku, AWS ou Vercel.... Para que você não tenha que fazer isso, eu já configurei um servidor que é livremente acessível. Para configurá-lo, abra o arquivo `src\environments\environment.prod.ts` e adicione a seguinte configuração:

```
export const environment = {
  production: true,
  api: 'https://northwind.vercel.app/api/'
}
```

Depois de ter feito isso, execute `ng deploy` novamente.

Para ver meu projeto, acesse: <https://danielschmitz.github.io/my-sales-app-angular/>

5.5. O “Módulo Material”

Como podemos ver ao criar a tela de categorias, às vezes precisamos importar módulos do Material, como o MatButtonModule ou MatInputModule. Essa tarefa pode se tornar um pouco tediosa após criar dezenas de telas em um sistema.

Por essa razão, acredito que podemos criar um módulo chamado “Material” que pode conter os principais componentes do Material.

Para criar um módulo, use o seguinte comando:

```
ng generate module material --flat
```

O parâmetro `-flat` cria o módulo na raiz da aplicação.

O que precisamos fazer neste módulo é importar os principais componentes do Material e, em seguida, exportá-los:

```
// src\app\material.module.ts
import {NgModule} from '@angular/core'
import {ReactiveFormsModule} from '@angular/forms'
import {MatButtonModule} from '@angular/material/button'
import {MatCardModule} from '@angular/material/card'
import {MatIconModule} from '@angular/material/icon'
import {MatInputModule} from '@angular/material/input'
import {MatPaginatorModule} from '@angular/material/paginator'
import {MatSortModule} from '@angular/material/sort'
import {MatTableModule} from '@angular/material/table'
import {MatTableModule} from '@angular/material/table'

@NgModule({
  declarations: [],
  imports: [
    MatTableModule,
    MatPaginatorModule,
    MatSortModule,
    MatCardModule,
    MatButtonModule,
    MatIconModule,
    MatButtonModule,
    MatInputModule,
    ReactiveFormsModule,
```

```
    MatCardModule,
    MatFormFieldModule
],
exports: [
    MatTableModule,
    MatPaginatorModule,
    MatSortModule,
    MatCardModule,
    MatButtonModule,
    MatIconModule,
    MatButtonModule,
    MatInputModule,
    ReactiveFormsModule,
    MatCardModule,
    MatFormFieldModule
]
})
export class MaterialModule {}
```

Nós pegamos todos os módulos do Material que foram usados para a tela de Categorias, importamos e exportamos na classe `MaterialModule`. O que precisamos fazer agora é, ao invés de importar os módulos do Material separadamente nos nossos componentes, importar diretamente o `MaterialModule`. Veja:

```
// src\app\categories\categories.component.ts
// imports
import {MaterialModule} from './material.module'

@Component({
  selector: 'app-categories',
  templateUrl: './categories.component.html',
  standalone: true,
  imports: [MaterialModule, CategoryFormComponent, LoadingBarComponent]
})
export class CategoriesComponent implements AfterViewInit {
  // ... code ...
}
```

e:

```
// src\app\categories\form\form.component.ts
// imports...
import {MaterialModule} from '../../../../../material.module'

@Component({
  selector: 'category-form',
  standalone: true,
  imports: [MaterialModule],
  templateUrl: './form.component.html',
  styles: ``
})

export class CategoryFormComponent {
  // ... code ...
}
```

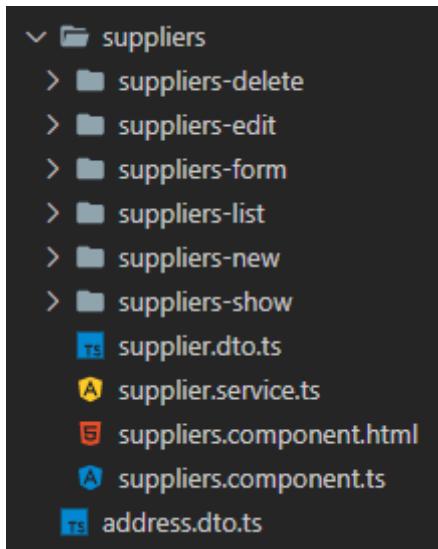
6. Fornecedor

Neste capítulo, vamos criar a tela de Fornecedores. Em vez de copiar a tela de categorias, colá-la como fornecedores e mudar os campos, vamos criar a tela de fornecedores de uma maneira diferente, para que você possa ter opções para escolher ao desenvolver seu sistema.

Em vez de criar uma variável chamada `showForm`, que controla se o formulário é exibido ou não, usaremos rotas para separar cada ação, por exemplo:

- `/suppliers` Tela Principal
- `/suppliers/list` Componente para mostrar uma lista de fornecedores
- `/suppliers/show` Componente para mostrar um fornecedor
- `/suppliers/form` Componente para mostrar um formulário de fornecedores
- `/suppliers/new` Componente para criar um novo fornecedor
- `/supplier/edit` Componente para editar um fornecedor
- `/supplier/delete` Componente para deletar um fornecedor

A estrutura final do diretório da tela de Fornecedores se parece com a seguinte imagem:



6.1. Criar os Componentes dos Fornecedores

Vamos inicialmente criar todos os componentes que fazem parte da estrutura da tela de Fornecedores. O primeiro deles é o componente “fornecedores”:

```
~/src/app/ $ ng g c fornecedores
```

Se você configurou para não criar testes e o arquivo css (capítulo anterior), apenas os arquivos `src\app\fornecedores\fornecedores.component.ts` e `src\app\fornecedores\fornecedores.component.html` serão criados.

Se você criou o MaterialModule (capítulo anterior), adicione-o ao FornecedorComponent:

```
import {Component} from '@angular/core'
import {MaterialModule} from '../material.module'

@Component({
  selector: 'app-suppliers',
  standalone: true,
  imports: [MaterialModule],
  templateUrl: './suppliers.component.html',
  styles: ``
})
export class FornecedoresComponent {}
```

Se você não criou o Módulo Material, terá que importar os componentes materiais conforme cria o template.

Altere o template do Fornecedor para:

```
<!-- `src\app\suppliers\suppliers.component.html` -->
<mat-card>
  <mat-card-content>
    <mat-card-title> Suppliers </mat-card-title>
    <router-outlet></router-outlet>
  </mat-card-content>
</mat-card>
```

Quando adicionamos o `<router-outlet>`, recebemos o seguinte erro:

```
app / suppliers / suppliers.component.html > ...
  Go to component
1  <mat-card-title>
2  |   Suppliers
3  </mat-card-title>
4  <router-outlet></router-outlet>      'router-outlet' is
5  'router-outlet' is not a known element:
  1. If 'router-outlet' is an Angular component, then v
     is included in the '@Component.imports' of this compo
  2. If 'router-outlet' is a Web Component then add
     'CUSTOM ELEMENTS SCHEMA' to the '@Component.schemas' <
```

Você precisa importá-lo no componente.

```
import {Component} from '@angular/core'
import {MaterialModule} from '../material.module'
import {RouterOutlet} from '@angular/router'

@Component({
  selector: 'app-suppliers',
  standalone: true,
  imports: [MaterialModule, RouterOutlet],
  templateUrl: './suppliers.component.html',
  styles: ``
})
export class SuppliersComponent {}
```

Adicionamos um novo `router-outlet` para que possamos criar uma subrota. Desta forma, podemos criar as seguintes URLs:

- /suppliers Mostrar uma lista de fornecedores
- /suppliers/show/:id Mostrar um fornecedor com id :id
- /suppliers/new Criar um novo fornecedor
- /supplier/edit/:id Editar um fornecedor com id :id
- /supplier/delete/:id Deletar um fornecedor com id :id

Após criar o componente, criaremos os outros no diretório `suppliers`, da seguinte forma:

```
$ ng g c suppliers/suppliers-list
$ ng g c suppliers/suppliers-edit
$ ng g c suppliers/suppliers-new
$ ng g c suppliers/suppliers-form
$ ng g c suppliers/suppliers-delete
$ ng g c suppliers/suppliers-show
```

Quando usamos o nome `suppliers/suppliers-list`, estamos configurando que o componente `SupplierListComponent` será criado no diretório `suppliers`. Para que este comando funcione, certifique-se de que você está na raiz do seu projeto Angular.

6.2. Usando Rotas e Sub Rotas

Agora, com os componentes criados, você pode configurar o roteador no arquivo `src\app\app.routes.ts`.

```
import {Routes} from '@angular/router'
import {CategoriesComponent} from './categories/categories.component'
import {SuppliersComponent} from './suppliers/suppliers.component'
import {SuppliersListComponent} from './suppliers/suppliers-list/suppliers-list.component'

export const routes: Routes = [
  {
    path: 'categories',
    component: CategoriesComponent
  },
  {
    path: 'suppliers',
    component: SuppliersComponent,
    children: [
```

```
{  
  path: '',  
  component: SuppliersListComponent  
}  
]  
}  
]
```

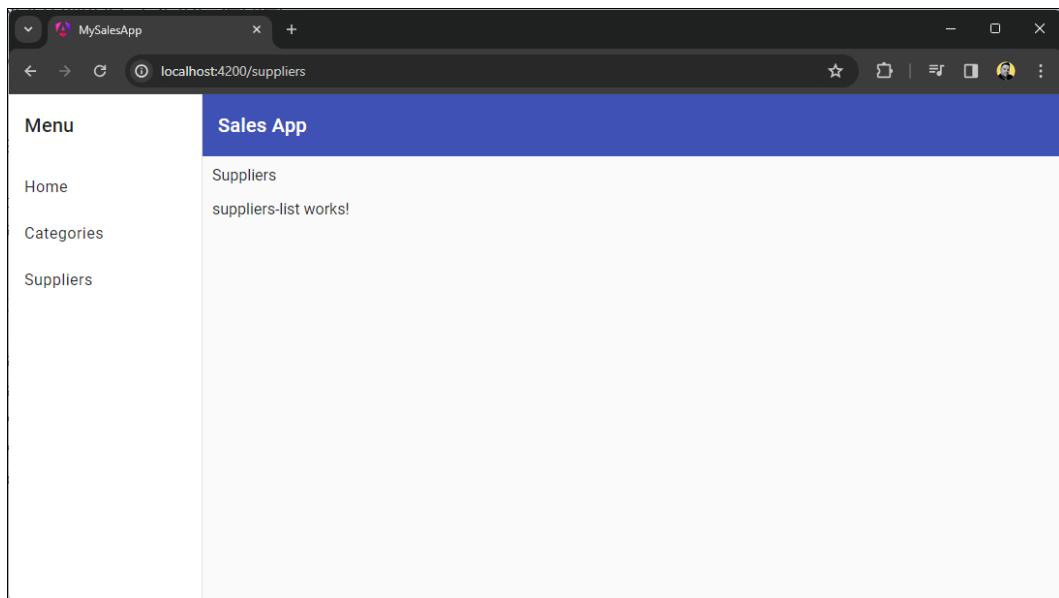
Veja que o caminho `suppliers` irá carregar o componente `SuppliersComponent`, e que a propriedade `children` possui um elemento cujo caminho é '' , e o componente a ser carregado é o `SuppliersListComponent`. Isso significa que quando você acessar o endereço `http://localhost:4200/suppliers`, o componente `SuppliersComponent` será carregado, e o `router-outlet` deste componente irá carregar o componente `SuppliersListComponent`.

Verifique no componente `src\app\menu\menu.component.ts` que o menu está configurado corretamente:

```
\\" src\\app\\menu\\menu.component.ts  
import { Component } from '@angular/core';  
import { MatListModule } from '@angular/material/list';  
  
interface MenuItem {  
  /**  
   * The path that will be loaded when you click on the menu  
   */  
  path: string;  
  /**  
   * The text that will be displayed in the menu  
   */  
  label: string;  
}  
  
@Component({  
  selector: 'app-menu',  
  standalone: true,  
  imports: [MatListModule],  
  template: `  
    @for (item of menuItems; track item.path) {  
      <a mat-list-item [href]="item.path">{{item.label}}</a>  
    }  
  `,  
  styles: ``
```

```
)  
export class MenuComponent {  
  menuItems: Array<MenuItem> = [  
    {  
      path: '/',
      label: 'Home'  
    },  
    {  
      path: '/categories',
      label: 'Categories'  
    },  
    {  
      path: '/suppliers',
      label: 'Suppliers'  
    },  
  ]  
}
```

Você vê, quando você clica no menu “Fornecedores” na aplicação, o componente SuppliersComponent é carregado, e a mensagem suppliers-list works! aparece, já que o router-outlet está configurado para carregar o componente SuppliersListComponent:



6.3. DTO do Fornecedor

A classe Data Transfer Object representando um objeto Fornecedor pode ser configurada a partir de um JSON representando um Fornecedor. Este json pode ser obtido no seguinte endereço: <https://northwind.vercel.app/api/suppliers>.

Ao acessar este endereço, podemos obter um fornecedor, que é semelhante ao seguinte código:

```
{
  "id": 4,
  "companyName": "Tokyo Traders",
  "contactName": "Yoshi Nagase",
  "contactTitle": "Marketing Manager",
  "address": {
    "street": "9-8 Sekimai Musashino-shi",
    "city": "Tokyo",
    "region": "NULL",
    "postalCode": 100,
    "country": "Japan",
    "phone": "(03) 3555-5011"
  }
}
```

O Fornecedor possui os campos id, companyName, contactName e contactTitle, e o campo address. Para “transformar” essas informações em um DTO, vá até [Json To Typescript Tool](#) e cole o json acima na caixa de texto.

Clique no botão “configurações” e desative o modo “mono”.

Primeiro, vamos ver como o resultado se parece:

```
export interface Root {  
    id: number  
    companyName: string  
    contactName: string  
    contactTitle: string  
    address: Address  
}  
  
export interface Address {  
    street: string  
    city: string  
    region: string  
    postalCode: number  
    country: string  
    phone: string  
}
```

Você pode ver que existem duas interfaces, uma chamada Root, que na verdade é Fornecedor, e outra chamada Address. Você vê que a interface Root tem uma propriedade que é do tipo Address. Então, a primeira tarefa é criar uma interface que represente Address.

```
ng g interface address/address.dto
```

O gerador ng do angular criará o diretório address (se já não existir) e o arquivo src/app/address/address.dto.ts, inicialmente com o seguinte conteúdo:

```
export interface AddressDto {}
```

Mude o nome da Interface, de AddressDto para Address, e adicione os campos de Address, conforme extraído do site [Json To Typescript Tool](#):

```
export interface Address {  
    street: string  
    city: string  
    region: string  
    postalCode: number  
    country: string  
    phone: string  
}
```

De volta ao diretório do Fornecedor, crie o arquivo `src/app/suppliers/supplier.dto.ts` com o seguinte código:

Você pode criar o Dto do Fornecedor com o comando `ng g i suppliers/supplier.dto` ou criando o arquivo diretamente.

```
import {Address} from './address/address.dto'  
  
export interface Supplier {  
    id?: number  
    companyName: string  
    contactName: string  
    contactTitle: string  
    address: Address  
}
```

A interface Root copiada do site agora é chamada `Supplier`. O campo `id` foi definido como opcional, usando `id?`, os outros campos são obrigatórios. O campo `address` é do tipo `Address`, a interface previamente criada na raiz do projeto. Observe que está sendo importada pelo comando `Import`. Como convenção, é importante que cada DTO represente um único tipo de entidade.

Com os DTOs prontos, podemos configurar o serviço que acessará os dados do fake-server.

6.4. Serviço de Fornecedores

A classe `SuppliersService` pode ser criada em `src/app/suppliers/` pelo seguinte comando:

```
$ ng g s suppliers/Supplier --skip-tests  
CREATE src/app/suppliers/supplier.service.ts
```

O arquivo `src\app\suppliers\suppliers.service.ts` será criado, e o seguinte código pode ser adicionado:

```
// src\app\suppliers\supplier.service.ts  
  
import {HttpClient} from '@angular/common/http'  
import {Injectable} from '@angular/core'  
import {Observable} from 'rxjs'  
import {environment} from 'src/environments/environment'  
import {Supplier} from './supplier.dto'  
  
@Injectable({  
    providedIn: 'root'  
})  
export class SupplierService {  
    constructor(private http: HttpClient) {}  
  
    public getAll(): Observable<Supplier[]> {  
        return this.http.get<Supplier[]>(environment.api + 'suppliers')  
    }  
  
    public getById(id: Number): Observable<Supplier> {  
        return this.http.get<Supplier>(environment.api + 'suppliers/' + id)  
    }  
  
    public save(supplier: Supplier): Observable<Supplier> {  
        if (supplier.id)  
            return this.http.put<Supplier>(  
                environment.api + 'suppliers/' + supplier.id,  
                supplier  
            )  
  
        return this.http.post<Supplier>(environment.api + 'suppliers', supplier)  
    }  
  
    public delete(id?: number): Observable<Supplier> {  
        return this.http.delete<Supplier>(environment.api + 'suppliers/' + id)  
    }  
}
```

```
public create(): Supplier {
  return {
    id: 0,
    companyName: '',
    contactName: '',
    contactTitle: '',
    address: {
      city: '',
      phone: '',
      country: '',
      postalCode: 0,
      region: '',
      street: ''
    }
  }
}
```

A classe `SupplierService` é muito semelhante à classe `CategoriesService`, e usa o DTO `Supplier` criado anteriormente. Nesta classe, temos os métodos para editar, deletar, obter uma lista de fornecedores, e também o método `create`, que retorna um fornecedor com os dados iniciais.

6.5. Listando Fornecedores

A rota padrão “/suppliers” foi definida para exibir o `SuppliersListComponent`. Neste componente, obteremos uma lista de todos os Fornecedores. Agora, usaremos uma variável `Observable` para controlar o *carregamento*.

```
// src\app\suppliers\suppliers-list\suppliers-list.component.ts
import {Component, OnInit} from '@angular/core'
import {MaterialModule} from '../../../../../material.module'
import {Observable, lastValueFrom} from 'rxjs'
import {Supplier} from '../supplier.dto'
import {SupplierService} from '../supplier.service'

@Component({
  selector: 'app-suppliers-list',
  standalone: true,
  imports: [MaterialModule],
```

```
templateUrl: './suppliers-list.component.html',
styles: ```
})
export class SuppliersListComponent implements OnInit {
  suppliers!: Supplier[]
  supplierObservable!: Observable<Supplier[]>

  constructor(private supplierService: SupplierService) {}

  async ngOnInit() {
    this.supplierObservable = this.supplierService.getAll()
    this.suppliers = await lastValueFrom(this.supplierObservable)
  }
}
```

Neste componente, criamos a variável `suppliers`, que é um array do DTO `Supplier`. E também criamos `supplierObservable`, que será o Observable do método `getAll` do serviço de Fornecedor.

O template se beneficiará de `supplierObservable` usando o `async` pipe. Dessa forma, não precisamos criar uma variável para exibir um carregamento na tela.

```
<!-- src\app\suppliers\suppliers-list\suppliers-list.component.html -->
<mat-card-subtitle> Listing all Suppliers </mat-card-subtitle>

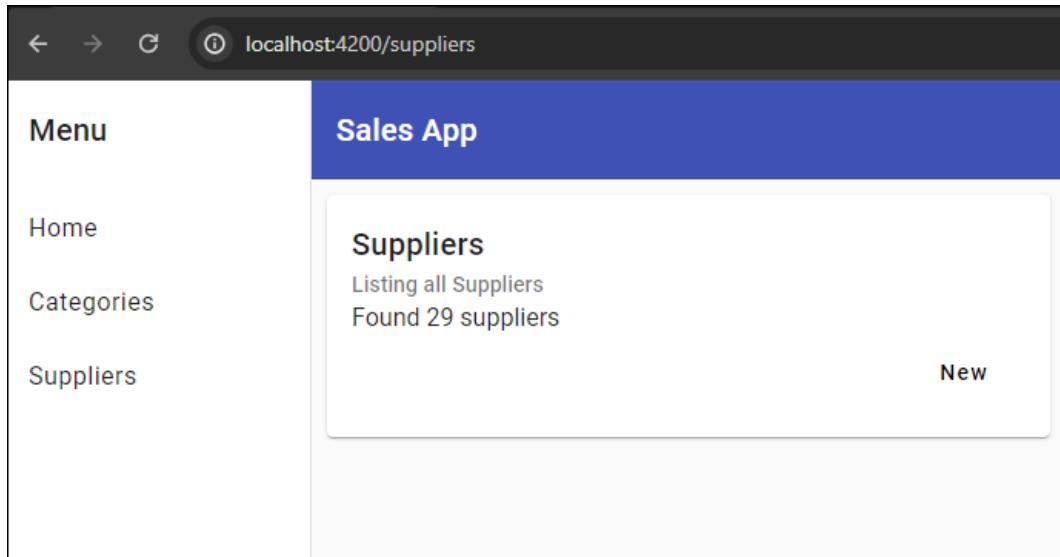
@if (supplierObservable|async) {
<div>Found {{ suppliers.length }} suppliers</div>
} @else {
<loading-bar></loading-bar>
}
<mat-card-actions align="end">
  <button mat-button [routerLink]="/suppliers/new">New</button>
</mat-card-actions>
```

O LoadingBar, AsyncPipe e RouterLink devem ser adicionados aos Imports:

```
import {Component, OnInit} from '@angular/core'
import {MaterialModule} from '../../../../../material.module'
import {Observable, lastValueFrom} from 'rxjs'
import {Supplier} from '../suppliers.dto'
import {SupplierService} from '../supplier.service'
import {LoadingBarComponent} from '../../../../../loading-bar.component'
import {AsyncPipe} from '@angular/common'
import {RouterLink} from '@angular/router'

@Component({
  selector: 'app-suppliers-list',
  standalone: true,
  imports: [MaterialModule, LoadingBarComponent, AsyncPipe, RouterLink],
  templateUrl: './suppliers-list.component.html',
  styles: ``
})
export class SuppliersListComponent implements OnInit {
  // ...code...
}
```

Neste código, o resultado esperado é inicialmente a barra de carregamento até que o SupplierObservable seja resolvido e o array suppliers seja preenchido. Quando isso acontecer, os conteúdos do @if serão exibidos, que mostra o número de Fornecedores existentes:



Se o carregamento parecer muito rápido, você pode usar a função *Throttling* das Ferramentas de Desenvolvimento do Google Chrome (F12) na aba Rede.

Em vez de exibir a quantidade total de *Fornecedores*, vamos percorrer os Fornecedores e exibir cada um em cartões:

```
<!-- src\app\suppliers\suppliers-list\suppliers-list.component.html -->
<mat-card-subtitle> Listing all Suppliers </mat-card-subtitle>

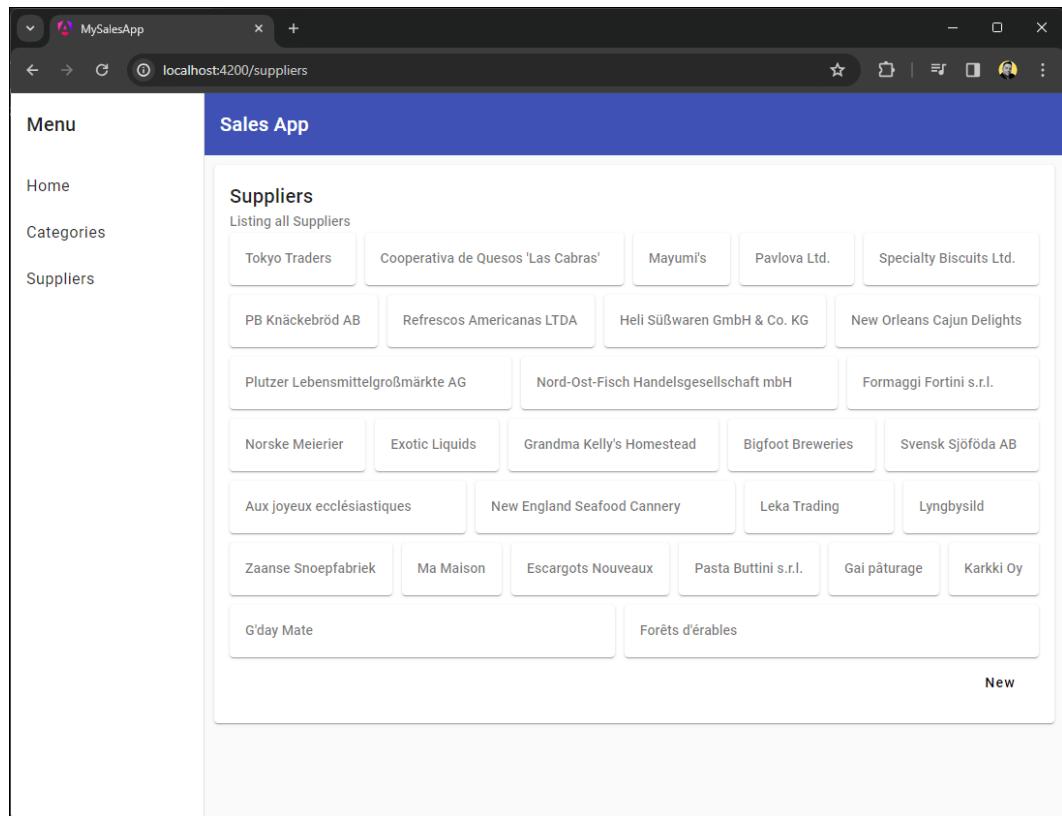
@if (supplierObservable|async) {

<div class="container wrap">
    @for( supplier of suppliers; track supplier.id) {
        <mat-card class="item">
            <mat-card-content>
                <mat-card-subtitle>{{supplier.companyName}}</mat-card-subtitle>
            </mat-card-content>
        </mat-card>
    }
</div>
```

```
} @else {  
<loading-bar></loading-bar>  
}  
<mat-card-actions align="end">  
  <button mat-button [routerLink]="/suppliers/new">New</button>  
</mat-card-actions>
```

Neste código, temos @for percorrendo os suppliers. Cada fornecedor é um *Mat Card*, cujo <mat-card-content> tem os campos supplier.companyName.

Usamos as classes css container, wrap e item para organizar os cartões dinamicamente na tela.



6.6. Criando um Novo Componente

Em vez de escrever o código para criar o cartão do fornecedor dentro do loop @for, vamos criar um componente chamado supplier-card:

```
ng g c suppliers/suppliers-list/supplier-card
```

O componente SupplierCardComponent será criado no diretório suppliers-list e tem o seguinte código:

```
import {Component, Input} from '@angular/core'
import {MaterialModule} from '../../../../../material.module'
import {Supplier} from '../../../../../suppliers.dto'

@Component({
  selector: 'app-supplier-card',
  standalone: true,
  imports: [MaterialModule],
  templateUrl: './supplier-card.component.html',
  styles: ``
})
export class SupplierCardComponent {
  @Input({required: true}) supplier: Supplier
}
```

Este componente usa o MaterialModule, e também tem o parâmetro supplier, que neste caso é obrigatório. O template inicial é mostrado abaixo:

```
<mat-card class="item">
  <mat-card-content>
    <mat-card-subtitle>{{supplier.companyName}}</mat-card-subtitle>
  </mat-card-content>
</mat-card>
```

Com o componente pronto, podemos adicioná-lo ao @for, veja:

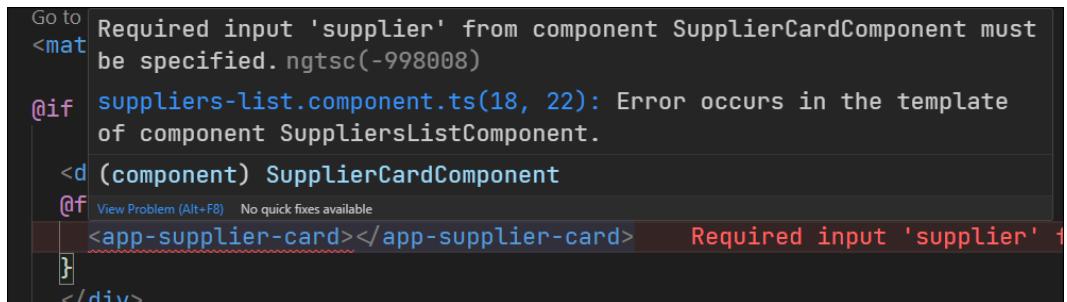
```
<mat-card-subtitle> Listing all Suppliers </mat-card-subtitle>

@if (supplierObservable|async) {

<div class="container wrap">
  @for( supplier of suppliers; track supplier.id) {
    <app-supplier-card [supplier]="supplier"></app-supplier-card>
  }
</div>

} @else {
<loading-bar></loading-bar>
}
<mat-card-actions align="end">
  <button mat-button [routerLink]="/suppliers/new">New</button>
</mat-card-actions>
```

Se o argumento supplier não for passado, um erro será exibido no componente:



The screenshot shows a code editor with the following code snippet:

```
Go to <mat> Required input 'supplier' from component SupplierCardComponent must be specified. ngtsc(-998008)
@if suppliers-list.component.ts(18, 22): Error occurs in the template of component SuppliersListComponent.
<d (component) SupplierCardComponent
@f View Problem (Alt+F8) No quick fixes available
  <app-supplier-card></app-supplier-card> Required input 'supplier' 1
}</div>
```

A tooltip for the `<app-supplier-card>` tag indicates an error: "Required input 'supplier'". The code editor interface includes standard elements like tabs, status bars, and toolbars.

Embora o resultado visual seja o mesmo, é uma boa prática de programação criar componentes que são segmentados, de modo que cada componente seja responsável apenas por uma única atribuição. Em outras palavras, se temos um componente chamado “suppliers-list”, este componente deve ser responsável apenas por exibir a lista de fornecedores. Cada cartão que representa um fornecedor pode ser outro componente.

Vamos adicionar mais funcionalidades ao componente `supplier-card`:

```
<mat-card class="item">
  <mat-card-content [routerLink]=["'show'", supplier.id]>
    <mat-card-subtitle>{{supplier.companyName}}</mat-card-subtitle>
    <span>{{supplier.contactName}}</span><br />
    <span>{{supplier.address.phone}}</span>
  </mat-card-content>
</mat-card>
```

Além de ContactName e phone, também adicionamos [routerLink] passando dois parâmetros. O primeiro é o caminho da rota que será configurado no arquivo app.routes.ts. O segundo parâmetro é o id do fornecedor.

6.7. Exibindo uma Mensagem Se @for Estiver Vazio

Podemos usar @empty para exibir uma mensagem se o número de fornecedores for 0. Veja:

```
<div class="container wrap">
  @for( supplier of suppliers; track supplier.id ) {
    <app-supplier-card [supplier]="supplier"></app-supplier-card>
  } @empty {
    <span>Without suppliers. Please create one in the "New" Button</span>
  }
</div>
```

6.8. Configurando Rotas

Quando o usuário clicar em um *Mat Card*, a url /suppliers/show será carregada, mas ainda não há uma rota configurada. A seguir, vamos configurar esta rota, e as outras, já que os componentes já estão criados:

```
// src\app\app.routes.module.ts
import {Routes} from '@angular/router'
import {CategoriesComponent} from './categories/categories.component'
import {SuppliersComponent} from './suppliers/suppliers.component'
import {SuppliersListComponent} from './suppliers/suppliers-list/suppliers-list\
.component'
import {SuppliersShowComponent} from './suppliers/suppliers-show/suppliers-show\
.component'
import {SuppliersDeleteComponent} from './suppliers/suppliers-delete/suppliers-\
delete.component'
import {SuppliersEditComponent} from './suppliers/suppliers-edit/suppliers-edit\
.component'
import {SuppliersNewComponent} from './suppliers/suppliers-new/suppliers-new.co\
mponent'

export const routes: Routes = [
{
  path: 'categories',
  component: CategoriesComponent
},
{
  path: 'suppliers',
  component: SuppliersComponent,
  children: [
    {
      path: '',
      component: SuppliersListComponent
    },
    {
      path: 'show/:id',
      component: SuppliersShowComponent
    },
    {
      path: 'edit/:id',
      component: SuppliersEditComponent
    },
    {
      path: 'del/:id',
      component: SuppliersDeleteComponent
    },
    {
      path: 'new',
      component: SuppliersNewComponent
    }
  ]
}
```

```
        }
    ]
}
]
```

Agora, todas as rotas da tela de Fornecedores foram configuradas.

6.9. Mostrando um Fornecedor

Com a rota `/suppliers/show/:id` configurada, podemos editar o arquivo `SuppliersShowComponent` para exibir um único Fornecedor.

```
// src\app\suppliers\suppliers-show\suppliers-show.component.ts

import {Component, OnInit, inject} from '@angular/core'
import {MaterialModule} from '../../../../../material.module'
import {ActivatedRoute, Route, RouterLink} from '@angular/router'
import {SupplierService} from '../supplier.service'
import {Supplier} from '../suppliers.dto'
import {Observable, lastValueFrom} from 'rxjs'
import {AsyncPipe} from '@angular/common'
import {LoadingBarComponent} from '../../../../../loading-bar.component'

@Component({
  selector: 'app-suppliers-show',
  standalone: true,
  imports: [MaterialModule, AsyncPipe, LoadingBarComponent, RouterLink],
  templateUrl: './suppliers-show.component.html',
  styles: ``
})
export class SuppliersShowComponent implements OnInit {
  route = inject(ActivatedRoute)
  supplierService = inject(SupplierService)
  supplier: Supplier
  supplierObservable: Observable<Supplier>

  async ngOnInit() {
    const id: Number = +(this.route.snapshot.paramMap.get('id') || 0)
    this.supplierObservable = this.supplierService.getById(id)
    this.supplier = await lastValueFrom(this.supplierObservable)
    console.log(this.supplier)
```

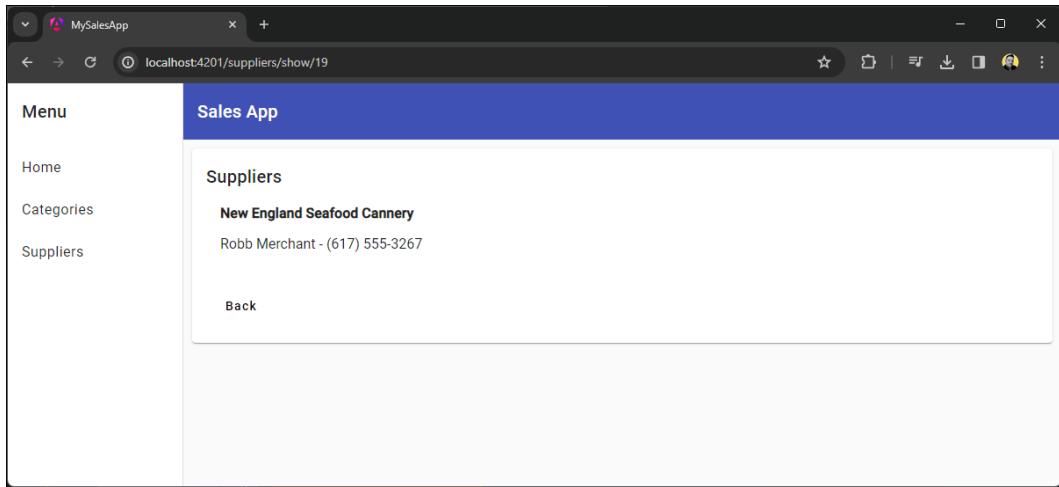
```
    }  
}
```

Neste código, estamos novamente usando `supplier` e `supplierObservable` para obter os dados do servidor. O que é novo neste código é o uso do `router` para obter o id do Fornecedor da url. A classe `ActivatedRoute` é injetada no componente e usamos a propriedade `snapshot.paramMap` para obter o Id da url. Como id é um número, precisamos usar o “+” para converter String em Número, e também precisamos dizer que se `paramMap.get` for nulo, 0 deve ser usado.

No template, vamos novamente usar o `Async` pipe para exibir o carregamento, conforme segue:

```
<!-- src\app\suppliers\suppliers-show\suppliers-show.component.html -->  
@if (supplierObservable|async) {  
  <mat-card-content>  
    <strong>{{supplier.companyName}}</strong> <br />  
    <p>{{supplier.contactName}} - {{supplier.address.phone}}</p>  
    <br />  
  </mat-card-content>  
  <mat-card-actions>  
    <button mat-button [routerLink]="'['/suppliers']'">Back</button>  
  </mat-card-actions>  
} @else {  
  <loading-bar></loading-bar>  
}
```

Quando clicamos em um cartão na Lista de Fornecedores, seremos redirecionados para este componente, que exibirá o carregamento, e então a seguinte tela:



Vamos adicionar mais informações nesta tela:

```
<!-- src\app\suppliers\suppliers-show\suppliers-show.component.html -->
@if (supplierObservable|async) {
<mat-card-content>
  <h3>{{ supplier.contactName }}</h3>
  <div>{{ supplier.companyName }}</div>
  <br /><br />
  <div>
    <dl>
      <dt>Street</dt>
      <dd>{{ supplier.address?.street }}</dd>
    </dl>
    <dl>
      <dt>Location</dt>
      <dd>
        {{ supplier.address?.city }}, {{ supplier.address?.region }} - {{ supplier.address?.country }}
      </dd>
    </dl>
    <dl>
      <dt>PO Box</dt>
      <dd>{{ supplier.address?.postalCode }}</dd>
    </dl>
    <dl>
      <dt>Phone</dt>
      <dd>{{ supplier.address?.phone }}</dd>
    </dl>
  </div>
}</mat-card-content>
```

```
</dl>
</div>
</mat-card-content>
<mat-card-actions class="container">
  <button mat-button [routerLink]="/suppliers">Voltar</button>
  <div class="width-full"></div>
  <button
    mat-raised-button
    color="primary"
    [routerLink]="/suppliers/edit/",supplier.id]>
    >
    Edit
  </button>
  <button mat-button [routerLink]="/suppliers/del/",supplier.id]>
    Delete
  </button>
</mat-card-actions>
} @else {
<loading-bar></loading-bar>
}
```

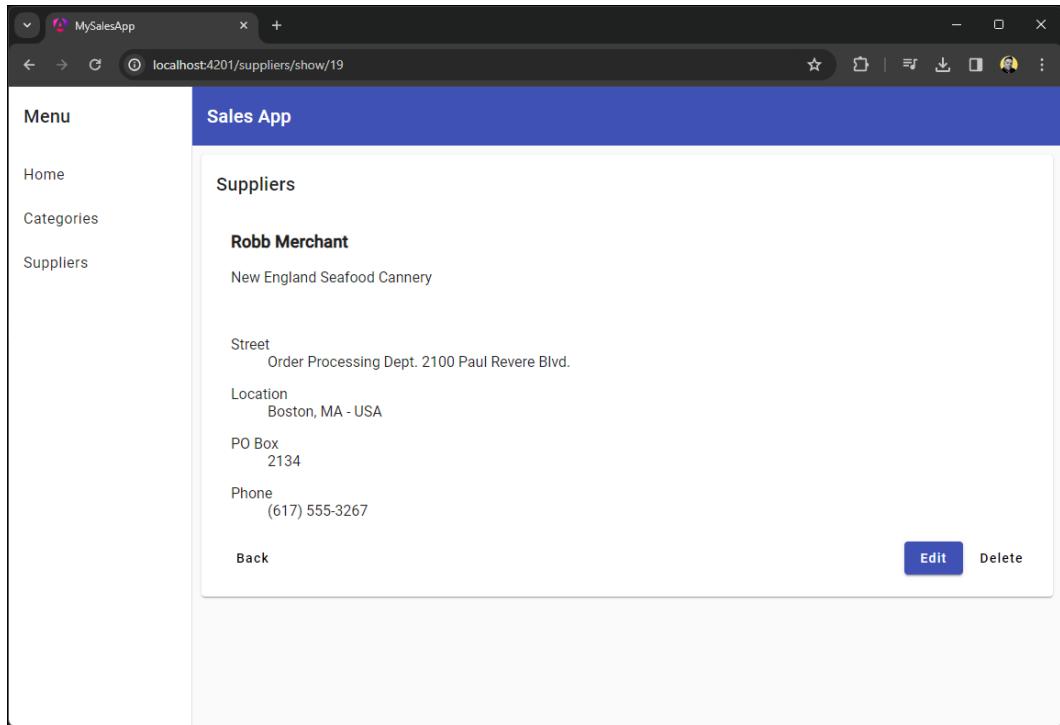
Aqui temos a tela inteira configurada. Usamos a tag html `<dl>`, `<dt>`, `<dd>` para organizar os dados apresentados na tela, já que o Fornecedor tem muitas informações.

Após exibir os dados do Fornecedor, criamos dois botões: Editar e Excluir. O botão de editar tem o `routerlink` para `['/suppliers/edit/', supplier?.id]`, ou seja, ele carregará, por exemplo, a url `/suppliers/edit/1`. O de excluir tem o mesmo comportamento, chamando `/suppliers/del/`.

A tag `<div class="width-full">` é usada para adicionar espaço de largura 100% entre os botões.

O botão *Voltar* tem o `routerlink` para `'/suppliers'` para que, se o usuário clicar no botão *Voltar*, ele seja novamente redirecionado para a listagem de Fornecedores.

A tela que exibe um Fornecedor se parece com a seguinte figura:



6.10. Editar um Fornecedor {/examples/}

O botão “Editar” do componente `SupplierShowComponent` tem o `RouterLink` para `/suppliers/edit/:id` no qual editaremos o Fornecedor.

Uma vez que as telas de editar fornecedor e criar fornecedor têm o mesmo formulário, podemos usar o componente `SupplierFormComponent` em ambos os componentes. Então, a função básica do componente `SupplierEditComponent` é carregar os dados do Fornecedor e passá-los para o formulário.

```
// src\app\suppliers\suppliers-edit\suppliers-edit.component.ts
import {Component, inject} from '@angular/core'
import {ActivatedRoute, RouterLink} from '@angular/router'
import {Observable, lastValueFrom} from 'rxjs'
import {SupplierService} from '../supplier.service'
import {Supplier} from '../suppliers.dto'
import {AsyncPipe} from '@angular/common'
import {LoadingBarComponent} from '../../loading-bar.component'
import {MaterialModule} from '../../material.module'

@Component({
  selector: 'app-suppliers-edit',
  standalone: true,
  imports: [MaterialModule, AsyncPipe, LoadingBarComponent, RouterLink],
  templateUrl: './suppliers-edit.component.html',
  styles: ``
})

export class SuppliersEditComponent {
  route = inject(ActivatedRoute)
  supplierService = inject(SupplierService)
  supplier: Supplier
  supplierObservable: Observable<Supplier>

  async ngOnInit() {
    const id: Number = +(this.route.snapshot.paramMap.get('id') || 0)
    this.supplierObservable = this.supplierService.getById(id)
    this.supplier = await lastValueFrom(this.supplierObservable)
    console.log(this.supplier)
  }
}
```

O template inicialmente tem o seguinte código:

```
<!-- src\app\suppliers\suppliers-edit\suppliers-edit.component.html -->
<mat-card-subtitle> Edit Supplier </mat-card-subtitle>

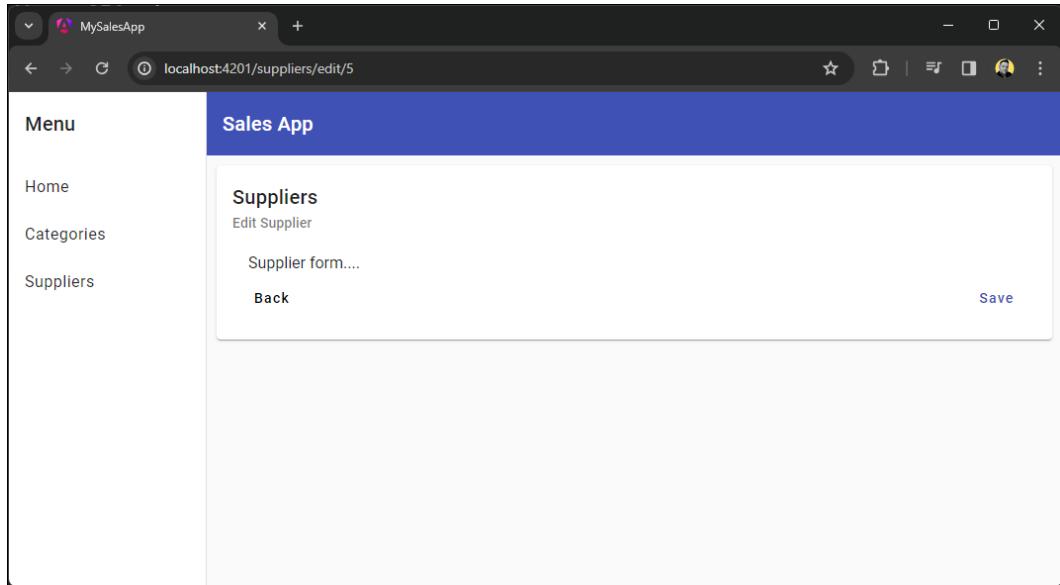
<br/>

@if (supplierObservable|async) {

<mat-card-content>Supplier form....</mat-card-content>

</mat-card-actions>

} @else {
<loading-bar></loading-bar>
}
```



6.11. Formulário de Fornecedor

O componente `SupplierFormComponent` é responsável por criar um formulário baseado nos dados do Fornecedor, e quando o usuário pressiona o botão `Salvar`, ele dispara um evento que pode ser utilizado no componente que o chamou.

```
// src\app\suppliers\suppliers-form\suppliers-form.component.ts
import {
  Component,
  EventEmitter,
  Input,
  OnInit,
  Output,
  inject
} from '@angular/core'
import {FormBuilder, FormGroup, Validators} from '@angular/forms'
import {Supplier} from '../suppliers.dto'
import {MaterialModule} from '../../material.module'

@Component({
  selector: 'app-suppliers-form',
  standalone: true,
  imports: [MaterialModule],
  templateUrl: './suppliers-form.component.html',
  styles: ``
})
export class SuppliersFormComponent implements OnInit {
  @Input({required: true}) supplier: Supplier
  @Output() save = new EventEmitter<Supplier>()
  @Output() back = new EventEmitter()
  suplierForm: FormGroup
  private fb = inject(FormBuilder)

  ngOnInit(): void {
    this.suplierForm = this.fb.group({
      id: [this.supplier.id],
      companyName: [
        this.supplier.companyName,
        [Validators.required, Validators.minLength(3)]
      ],
      contactName: [
        this.supplier.contactName,
        [Validators.required, Validators.minLength(3)]
      ],
      contactTitle: [this.supplier.contactTitle],
      address: this.fb.group({
        city: [this.supplier.address.city],
        country: [this.supplier.address.country],
        phone: [this.supplier.address.phone],
      })
    })
  }
}
```

```
    postalCode: [this.supplier.address.postalCode],
    region: [this.supplier.address.region],
    street: [this.supplier.address.street]
  )
}
}

onSubmit() {
  this.save.emit(this.supplierForm.value as Supplier)
}
onBack(event) {
  event.preventDefault()
  this.back.emit()
}
}
```

Criamos o FormGroup usando o FormBuilder, apresentado no capítulo anterior. Usamos a variável `this.supplier` para fornecer os dados iniciais do formulário, que serão passados por `@Input()`. Note que a propriedade `address` tem um novo FormGroup, com os campos de `address`.

O evento `onBack` tem `event.preventDefault()`, para evitar que o formulário seja enviado.

O método `onSubmit()` apenas dispara o evento `save` enviando os dados do formulário, que estão em `this.supplierForm.value`. O template para este formulário é:

```
<!-- src\app\suppliers\suppliers-form\suppliers-form.component.html -->
<form [formGroup]="supplierForm" (ngSubmit)="onSubmit()">
  <div class="container">
    <mat-form-field class="item">
      <input
        matInput
        placeholder="Company Name"
        formControlName="companyName"
      />
      @if (supplierForm.controls['companyName'].hasError('required')) {
        <mat-error> Company Name is required </mat-error>
      } @if (supplierForm.controls['companyName'].hasError('minlength')) {
        <mat-error> Company Name is too short </mat-error>
      }
    </mat-form-field>
  </div>
  <div class="container">
```

```
<mat-form-field class="item">
  <input
    matInput
    placeholder="Contact Name"
    formControlName="contactName"
  />
  @if (supplierForm.controls['contactName'].hasError('required')) {
    <mat-error>Contact Name is required</mat-error>
  } @if (supplierForm.controls['contactName'].hasError('minlength')) {
    <mat-error>Contact Name is too short</mat-error>
  }
</mat-form-field>
<mat-form-field class="item">
  <input
    matInput
    placeholder="Contact Title"
    formControlName="contactTitle"
  />
  @if (supplierForm.controls['contactTitle'].hasError('required')) {
    <mat-error>Contact Title is required</mat-error>
  } @if (supplierForm.controls['contactTitle'].hasError('minlength')) {
    <mat-error>Contact Title is too short</mat-error>
  }
</mat-form-field>
</div>

<br />

<mat-card-subtitle>Address</mat-card-subtitle>

<div class="container" formGroupName="address">
  <!-- Street -->
  <mat-form-field class="item">
    <input matInput placeholder="Rua" formControlName="street" />
  </mat-form-field>
</div>
<div class="container" formGroupName="address">
  <!-- City -->
  <mat-form-field class="item">
    <input matInput placeholder="Cidade" formControlName="city" />
  </mat-form-field>

  <!-- Region -->
```

```
<mat-form-field class="item">
  <input matInput placeholder="Região" formControlName="region" />
</mat-form-field>

<!-- Country -->
<mat-form-field class="item">
  <input matInput placeholder="País" formControlName="country" />
</mat-form-field>
</div>
<div class="container" formGroupName="address">
  <!-- Postal Code -->
  <mat-form-field class="item">
    <input
      matInput
      placeholder="Código Postal"
      formControlName="postalCode"
    />
  </mat-form-field>

  <!-- Phone -->
  <mat-form-field class="item">
    <input matInput placeholder="Telefone" formControlName="phone" />
  </mat-form-field>
</div>

<mat-card-actions>
  <button mat-button (click)="(onBack($event))">Back</button>
  <div class="width-full"></div>
  <button
    mat-raised-button
    color="primary"
    type="submit"
    [disabled]="supplierForm.invalid"
    (click)="(onSubmit)"
  >
    Save
  </button>
</mat-card-actions>
</form>
```

Note que o formulário tem o campo `[formGroup] = "supplierForm"` para que você possa usar o `formControlName` dos campos principais, e que quando o campo `address` é referenciado, ele deve estar dentro do `formGroupName = "address"`, assim você pode usar o `formControlName`

dos campos de Endereço.

O botão “Salvar” chama o método `onSubmit` que irá disparar o evento `save`.

6.12. Adicionando o Formulário no SuppliersEditComponent

Agora que o formulário está pronto, podemos adicioná-lo ao template do `SuppliersEditComponent`:

```
<!-- src\app\suppliers\suppliers-edit\suppliers-edit.component.html -->

<mat-card-subtitle> Edit Supplier </mat-card-subtitle>

<br />

@if (supplierObservable|async) {

<mat-card-content>
  <app-suppliers-form
    [supplier]="supplier"
    (save)="onSave($event)"
    (back)="onBack()"
  ></app-suppliers-form>
</mat-card-content>

} @else {
<loading-bar></loading-bar>
}
```

O método `onSave` e `onBack` referenciado no template é mostrado abaixo:

```
//src\app\suppliers\suppliers-edit\suppliers-edit.component.ts
// ... code ...

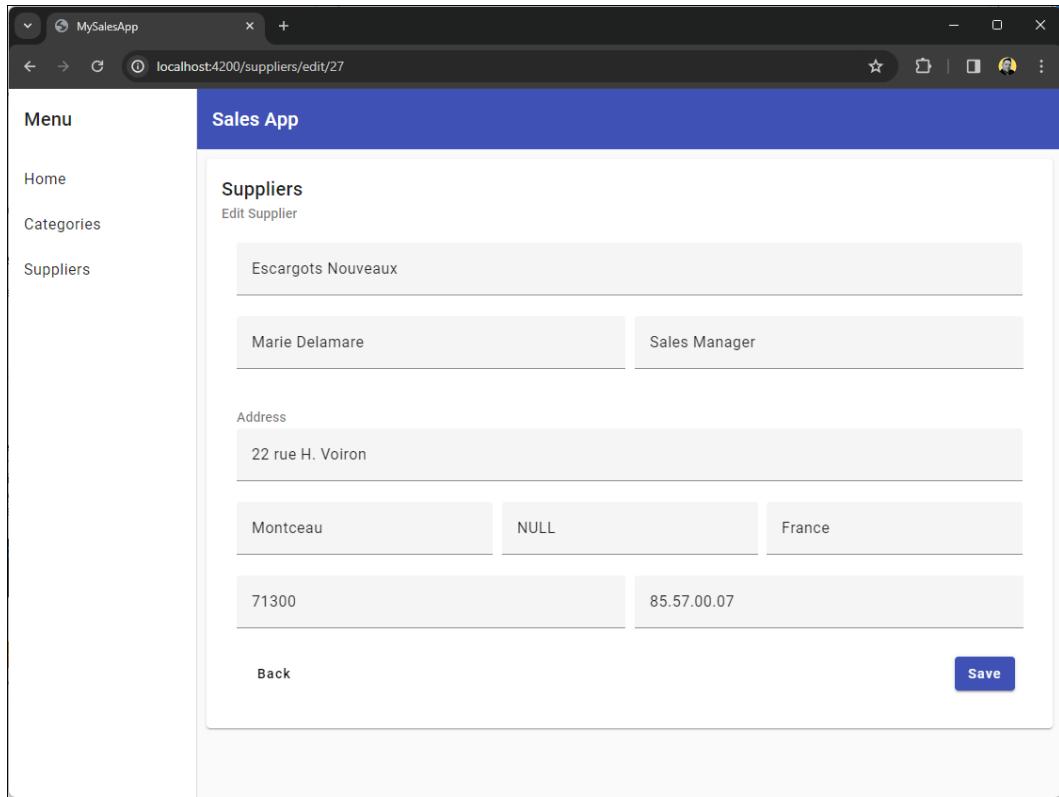
async onSave(supplier: Supplier) {
  this.supplierObservable = this.supplierService.save(supplier);
  this.supplier = await lastValueFrom(this.supplierObservable);
  this.router.navigate(['/suppliers/show/', supplier?.id]);
}

onBack() {
  this.router.navigate(['/suppliers']);
}

// ... code ...
```

O método `onSave` irá atualizar a variável `this.supplierObservable`, mas agora chamando o método `save` do `supplierService`. Isso exibirá o carregamento na tela. Note também que o método `save` aceita um objeto do tipo `Supplier`, que é precisamente o objeto passado pelo formulário.

Após fazer o `save`, usamos o `router` para navegar para a url `/suppliers/show/id`, retornando para a tela onde o Fornecedor é exibido.



6.13. Deletar Fornecedor

O botão “deletar” no SuppliersShowComponent carregará o SupplierDeleteComponent. Novamente, carregaremos os dados do Fornecedor e exibiremos uma mensagem ao usuário se eles querem remover o Fornecedor.

```
// src\app\suppliers\suppliers-delete\suppliers-delete.component.ts

import {Component, OnInit} from '@angular/core'
import {ActivatedRoute, Router} from '@angular/router'
import {lastValueFrom, Observable} from 'rxjs'
import {Supplier} from '../supplier.dto'
import {SupplierService} from '../supplier.service'

@Component({
  selector: 'app-suppliers-delete',
  templateUrl: './suppliers-delete.component.html',
  styles: []
})
export class SuppliersDeleteComponent implements OnInit {
  constructor(
    private supplierService: SupplierService,
    private route: ActivatedRoute,
    private router: Router
  ) {}

  supplier!: Supplier
  supplierObservable!: Observable<Supplier>

  async ngOnInit() {
    const id: Number = +(this.route.snapshot.paramMap.get('id') || 0)
    this.supplierObservable = this.supplierService.getById(id)
    this.supplier = await lastValueFrom(this.supplierObservable)
  }
}
```

The template has two buttons, Yes and No:

```
@if (supplierObservable|async) {
<mat-card-content>
  <h3>Delete {{ supplier.contactName }}?</h3>
</mat-card-content>
<mat-card-actions class="container">
  <button mat-button color="warn" (click)="confirmDelete()">Yes</button>
  <div class="width-full"></div>
  <button
    mat-button
    color="primary"
    [routerLink]="'[ '/suppliers/show', supplier.id ]'">
```

```
>
  No
  </button>
</mat-card-actions>
} @else {
<loading-bar></loading-bar>
}
```

O botão Não redirecionará para a url /fornecedores/show/:id, enquanto o botão Sim chamará o método confirmDelete, com o seguinte código:

```
// ... code ...
```

```
async confirmDelete() {
  this.supplierObservable = this.supplierService.delete(this.supplier.id)
  await lastValueFrom(this.supplierObservable)
  this.router.navigate(['/suppliers']);
}
```

```
// ... code ...
```

Em confirmDelete, o método delete de SupplierService é chamado. Após o registro ser removido, o roteador carrega a url '/fornecedores'.

6.14. Novo Fornecedor

O botão Novo Fornecedor pode ser inserido na tela de lista de **Fornecedores**:

```
```html
<mat-card-actions>
 <button mat-raised-button color="primary" [routerLink]=["'/suppliers/new']>
 New
 </button>
</mat-card-actions>
```

O componente SuppliersNewComponent criará um objeto Fornecedor vazio, para que possa ser fornecido ao formulário:

```
// src\app\suppliers\suppliers-new\suppliers-new.component.ts
import {Component, inject} from '@angular/core'
import {SuppliersFormComponent} from '../suppliers-form/suppliers-form.componen\
t'
import {Supplier} from '../suppliers.dto'
import {MaterialModule} from '../../../../../material.module'
import {Observable, lastValueFrom} from 'rxjs'
import {Router} from '@angular/router'
import {SupplierService} from '../supplier.service'

@Component({
 selector: 'app-suppliers-new',
 standalone: true,
 imports: [
 MaterialModule,
 SuppliersFormComponent,
 AsyncPipe,
 LoadingBarComponent
],
 templateUrl: './suppliers-new.component.html',
 styles: ``
})

export class SuppliersNewComponent {
 router = inject(Router)
 supplierService = inject(SupplierService)
 supplierObservable!: Observable<Supplier>
 supplier: Supplier

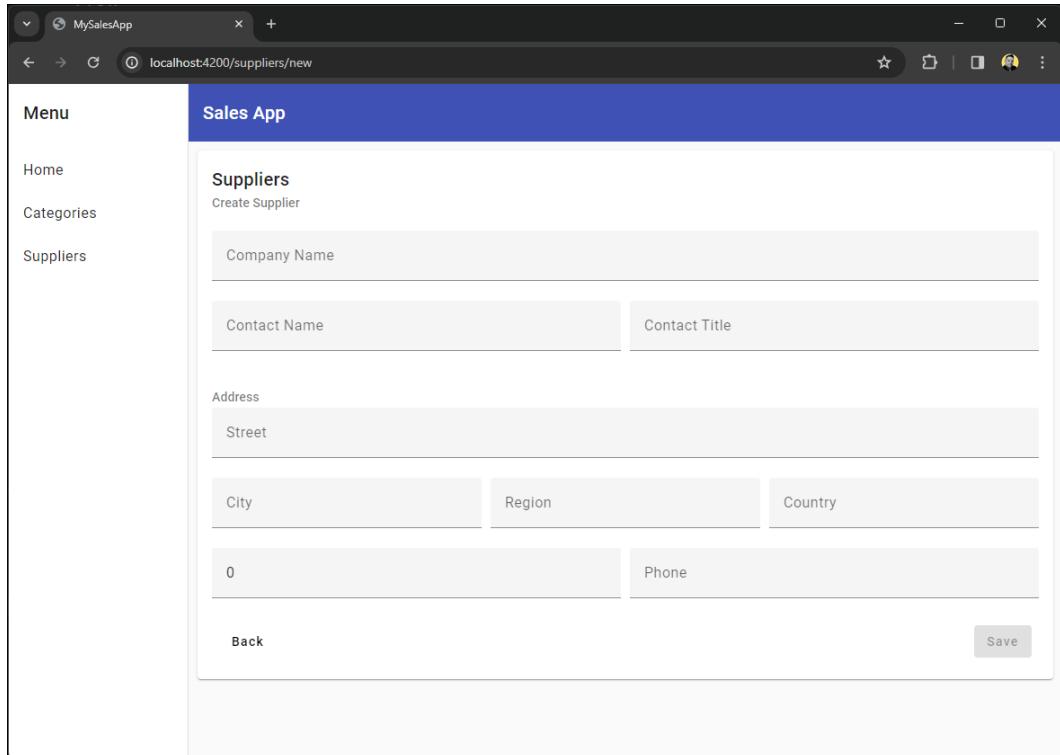
 async ngOnInit() {
 this.supplierObservable = await of(this.supplierService.create())
 this.supplier = await lastValueFrom(this.supplierObservable)
 }
 async onSave(supplier: Supplier) {
 this.supplierObservable = this.supplierService.save(supplier)
 const result = await lastValueFrom(this.supplierObservable)
 this.router.navigate(['/suppliers/show', result.id])
 }
}
```

A principal diferença com SuppliersNewComponent é que usamos supplierObservable para retornar um Fornecedor vazio. Para isso usamos o of do rxjs.

O template é semelhante ao componente para editar uma categoria:

```
<!-- src\app\suppliers\suppliers-new\suppliers-new.component.html -->
<mat-card-subtitle>Create Supplier</mat-card-subtitle>

@if (supplierObservable|async) {
<app-suppliers-form
 [supplier]="supplier"
 (save)="onSave($event)"
 (back)="onBack()">
</app-suppliers-form>
} @else {
<loading-bar></loading-bar>
}
```



## 6.15. Conclusão

O cadastro de Fornecedores trouxe uma nova maneira de programar uma tela, utilizando o recurso de sub rotas, em vez de usar ifs para determinar, por exemplo, se o formulário estava sendo exibido ou não.

Essa maneira é a sugerida pela equipe do Angular, porque quanto mais segmentados os componentes na aplicação, melhor para manter.

Na minha opinião pessoal, eu também prefiro usar sub rotas para determinar quais telas devem ser carregadas, e também usar Observables para determinar o carregamento.

No próximo capítulo, vamos programar o painel para exibir os produtos e a página de finalização de compra.

# 7. Produtos

Neste capítulo, vamos exibir uma listagem de produtos e um botão para adicionar produtos ao carrinho.

## 7.1. Arquivos Iniciais

Crie o componente products em src/app com o comando `ng g c products`.

No arquivo `src\app\products.component.html`, adicione o seguinte código:

```
<mat-card>
 <mat-card-header>
 <mat-card-title> Products </mat-card-title>
 </mat-card-header>
 <mat-card-content>
 <router-outlet></router-outlet>
 </mat-card-content>
</mat-card>
```

e o código:

```
import {Component} from '@angular/core'
import {MaterialModule} from '../material.module'
import {RouterOutlet} from '@angular/router'

@Component({
 selector: 'app-products',
 standalone: true,
 imports: [MaterialModule, RouterOutlet],
 templateUrl: './products.component.html',
 styles: ``
})
export class ProductsComponent {}
```

O componente router-outlet permite adicionar subrotas relacionadas a produtos, assim como fizemos com fornecedor.

Agora vamos criar a listagem de produtos. Primeiro, crie o componente product-list da seguinte forma: `ng g c products/products-list`. Após criar o componente, podemos alterar o roteador `app.routes.ts` para:

```
// src\app.routes.ts
// includes
import {ProductsComponent} from './products/products.component'
import {ProductsListComponent} from './products/products-list/products-list.componen't

export const routes: Routes = [
 {
 path: 'categories',
 component: CategoriesComponent
 },
 {
 path: 'suppliers',
 component: SuppliersComponent,
 children: [
 // routes
]
 },
 {
 path: '',
 component: ProductsComponent,
 children: [
 {
 path: '',
 component: ProductsListComponent
 }
]
 }
]
```

Mudamos o caminho '' para carregar o componente `ProductsListComponent` em vez do `DashboardComponent`. Quando você acessar `localhost:4200`, verá a mensagem `products-list works.`

## 7.2. O Serviço de Produtos

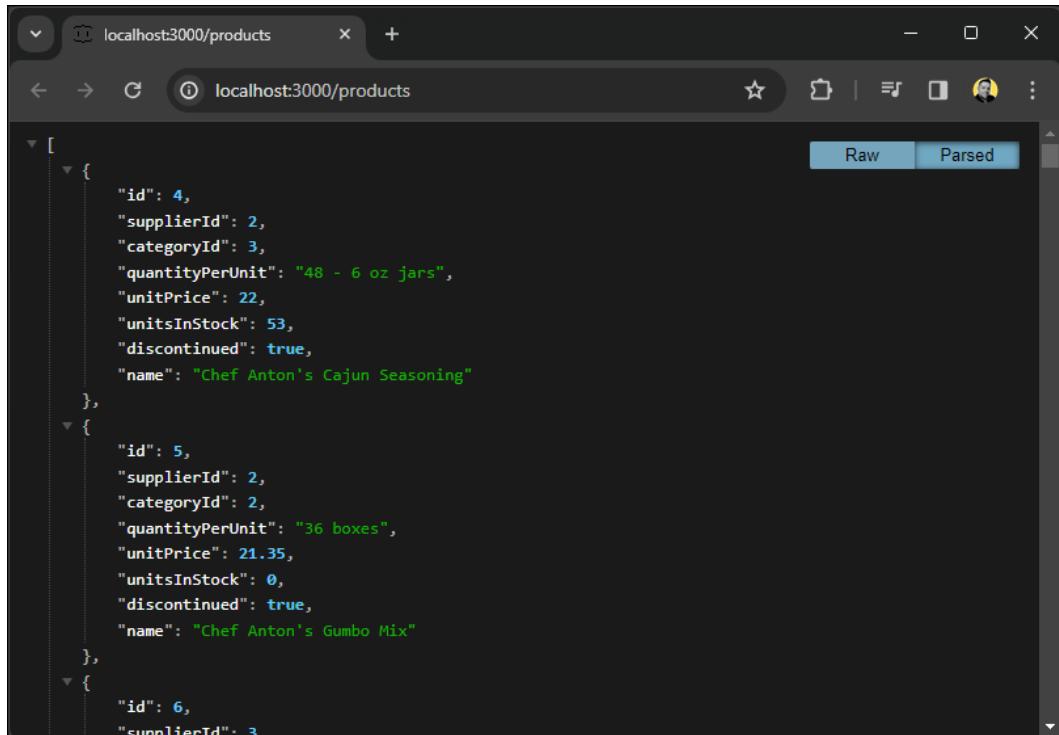
O Serviço de Produtos obterá os dados do produto do backend. Antes de criar o serviço, temos que criar o DTO (Data Transfer Object) que representa um Produto, usando o seguinte comando: `ng g i products/product.dto`. Após criar o arquivo, adicione o seguinte código:

```
// src\app\products\product.dto.ts

import {Category} from '../categories/category.dto'
import {Supplier} from '../suppliers/supplier.dto'

export interface Product {
 id?: number
 supplier?: Supplier
 category?: Category
 unitPrice: number
 unitsInStock: number
 name: string
 discontinued: Boolean
}
```

O DTO de Produto tem referências a Categorias e Fornecedores. A criação deste DTO foi configurada após observar nosso “fake-server” acessando <http://localhost:3000/products>:



```
[{"id": 4, "supplierId": 2, "categoryId": 3, "quantityPerUnit": "48 - 6 oz jars", "unitPrice": 22, "unitsInStock": 53, "discontinued": true, "name": "Chef Anton's Cajun Seasoning"}, {"id": 5, "supplierId": 2, "categoryId": 2, "quantityPerUnit": "36 boxes", "unitPrice": 21.35, "unitsInStock": 0, "discontinued": true, "name": "Chef Anton's Gumbo Mix"}, {"id": 6, "supplierId": 3, "categoryId": 1, "quantityPerUnit": "12 - 5 lb bags", "unitPrice": 123.79, "unitsInStock": 12, "discontinued": false, "name": "Gorgonzola Blue Cheese"}]
```

Agora vamos criar o serviço, usando o seguinte comando `ng g s products/product --skip-tests`, e adicionar o seguinte código:

```
// src\app\products\product.service.ts

import {HttpClient} from '@angular/common/http'
import {Injectable} from '@angular/core'
import {Observable, of} from 'rxjs'
import {environment} from 'src/environments/environment'
import {Product} from './product.dto'

@Injectable({
 providedIn: 'root'
})
export class ProductService {
 constructor(private http: HttpClient) {}

 public getAll(search: string = ''): Observable<Product[]> {
 const searchTerm = search != '' ? '&q=' + search : ''
 return this.http.get(environment.apiUrl + '/products?' + searchTerm)
 }
}
```

```
 return this.http.get<Product[]>(
 environment.api +
 'products?_expand=category&_expand=supplier' +
 searchTerm
)
 }
}
```

O Serviço de Produto tem o método `getAll`, que recupera todos os produtos de acordo com uma “busca”. A API tem o parâmetro `_expand` que também trará categorias e fornecedores.

Com o DTO e o Serviço prontos, podemos criar a listagem de produtos.

## 7.3. Listagem de Produtos

A listagem de produtos terá um formulário de busca, então precisamos criar um `FormGroup` com o nome `searchForm`. Também criamos a variável `products` que é um Array de produtos e a variável `productObservable` que usa `ProductService`.

```
// src\app\products\products-list\products-list.component.ts
import {Component, OnInit, inject} from '@angular/core'
import {FormBuilder, FormGroup} from '@angular/forms'
import {Observable, lastValueFrom} from 'rxjs'
import {Product} from '../product.dto'
import {ProductService} from '../product.service'
import {MaterialModule} from '../../../../../material.module'
import {AsyncPipe} from '@angular/common'
import {LoadingBarComponent} from '../../../../../loading-bar.component'

@Component({
 selector: 'app-products-list',
 standalone: true,
 templateUrl: './products-list.component.html',
 styles: ``,
 imports: [MaterialModule, AsyncPipe, LoadingBarComponent]
})
export class ProductsListComponent implements OnInit {
 productService = inject(ProductService)
 fb = inject(FormBuilder)
 products: Product[]
```

```
productsObservable: Observable<Product[]>
searchForm: FormGroup

async ngOnInit() {
 this.searchForm = this.fb.group({
 searchTerm: ['']
 })
 this.getProducts()
}

private async getProducts(searchTerm?: string) {
 this.productsObservable = this.productService.getAll(searchTerm)
 this.products = await lastValueFrom(this.productsObservable)
}

onSearch() {
 this.getProducts(this.searchForm.value.searchTerm)
}
}
```

O método ngOnInit cria o FormGroup com o campo searchForm e, em seguida, chama o método getProducts, que obterá todos os produtos consultando o serviço. O template html para a listagem de produtos é mostrado abaixo:

```
<!-- src\app\products\products-list\products-list.component.html -->

@if (productsObservable|async) {
<form [formGroup]="searchForm" (ngSubmit)="onSearch()">
 <div class="container">
 <mat-form-field class="width-full">
 <input matInput placeholder="Search" formControlName="searchTerm" />
 </mat-form-field>
 <div class="pt-10">
 <button mat-button type="submit">Search</button>
 </div>
 </div>
</form>
<div class="container wrap">
 @for(product of products; track product.id){
 <div style="width: 300px">
 <mat-card class="p-5">
 <mat-card-title-group>
```

```
<mat-card-subtitle> {{ product.category?.name }} </mat-card-subtitle>
<mat-card-subtitle>
 <h3>{{ product.name }}</h3>
</mat-card-subtitle>
<div>
 {{ product.unitPrice | currency }}
</div>
</mat-card-title-group>
<mat-card-content>
 Units in Stock: {{ product.unitsInStock }}
</mat-card-content>
<mat-card-actions align="end">
 <button mat-icon-button [disabled]="product.unitsInStock == 0">
 <mat-icon>add_shopping_cart</mat-icon>
 </button>
</mat-card-actions>
</mat-card>
</div>
}
</div>
} @else {
<loading-bar></loading-bar>
}
```

O template exibe um formulário com um campo de texto e uma listagem de cartões @for mostrando todos os produtos. Inserimos um botão “Adicionar ao carrinho” que será programado em seguida.

Adicione o CurrencyPipe à importação para usar o currency pipe.

Category	Name	Price	Description	Stock
Confections	Chef Anton's Cajun Seasoning	\$22.00	Chef Anton's Gumbo Mix	\$21.35
Condiments 2	Northwoods Cranberry Sauce	\$40.00	Chang	\$19.00
Condiments 2	Queso Cabrales 2	\$21.00	Dairy Products	\$38.00
			Queso Manchego La Pastora	
			Units in Stock: 96	
Condiments 2				Condiments 2
				\$15.50
				Genen Shouyu
				Units in Stock: 20

## 7.4. Adicionar Produto ao Carrinho

Para adicionar um produto ao carrinho, primeiro devemos criar um serviço que gerenciará os produtos no carrinho. Chamaremos isso de `CartService` e podemos criá-lo com o comando `ng g s cart --skip-tests`, e um dto de carrinho com o seguinte comando `ng g i cart.dto`:

```
// src\app\cart.dto.ts

export interface CartItem {
 idProduct?: number
 unitPrice: number
 quantity: number
 name: string
}
```

E:

```
// src\app\cart.service.ts

import {Injectable} from '@angular/core'
import {CartItem} from './cart.dto'

@Injectable({
 providedIn: 'root'
})
export class CartService {
 readonly CART: string = 'cart'
 readonly CART_QUANTITY: string = 'cart_quantity'

 public getItems(): Array<CartItem> {
 const cartItems = localStorage.getItem(this.CART)
 if (cartItems) {
 return JSON.parse(cartItems)
 }
 return []
 }

 public addItem(item: CartItem): void {
 let found = false
 const items = this.getItems()
 items.forEach((element) => {
 if (element.idProduct === item.idProduct) {
 element.quantity++
 found = true
 }
 })
 if (!found) {
 items.push(item)
 }
 localStorage.setItem(this.CART, JSON.stringify(items))
 localStorage.setItem(this.CART_QUANTITY, items.length.toString())
 }

 public removeItem(item: CartItem): void {
 let found = false
 const items = this.getItems()
 items.forEach((element) => {
 if (element.idProduct === item.idProduct) {
 element.quantity--
 found = true
 }
 })
 }
}
```

```
 }
 })
 const newItens = items.filter((element) => element.quantity > 0)
 localStorage.setItem(this.CART, JSON.stringify(newItens))
 localStorage.setItem(this.CART_QUANTITY, newItens.length.toString())
}

get itensInCart(): number {
 return this.getItems().length
}

get total(): number {
 let total = 0
 const items = this.getItems()
 items.forEach((element) => {
 total += element.unitPrice * element.quantity
 })
 return total
}
}
```

A ideia básica do `CartService` é armazenar os itens do carrinho de compras no `localStorage` do navegador. Criamos duas constantes chamadas `CART` e `CART_QUANTITY` que serão as chaves do `localStorage`.

O método `getItems` irá obter o texto salvo no `localStorage` que representa os itens no carrinho. Como o `localStorage` armazena apenas texto, é necessário converter o array de itens do carrinho em JSON. A string JSON é convertida em um array usando o método `JSON.parse`. O método `getItems` retorna um array de `CartItem`.

Se não houver JSON no `localStorage` cuja chave é `CART`, o método `getItems` retornará um array vazio.

O próximo método é o método `addItem`, responsável por adicionar um item do tipo `CartItem` ao carrinho. Deve-se notar que adicionar um item ao carrinho não é apenas adicionar o item ao array. Se o produto já está no carrinho, o que você deve fazer é aumentar sua quantidade:

```
public addItem(item: CartItem): void {
 let found = false;
 const items = this.getItems();
 items.forEach(element => {
 if (element.idProduct === item.idProduct) {
 element.quantity++;
 found = true;
 }
 });
 if (!found) {
 items.push(item);
 }
 localStorage.setItem(this.CART, JSON.stringify(items));
 localStorage.setItem(this.CART_QUANTITY, items.length.toString());
}
```

Se o produto não for encontrado, então nós o adicionamos ao array com o método push. Após adicionar o produto, usamos *localStorage* novamente para adicionar o Array de Produtos. Agora, usamos o método `JSON.stringify` para transformar o array em um JSON, para que possa ser armazenado como texto no *localStorage*. A quantidade de produtos no carrinho também é armazenada.

O método `removeItem` removerá o item do array se a quantidade for 1. Se a quantidade for maior, o método `removeItem` removerá apenas uma unidade da propriedade `quantidade`.

A propriedade `itemsInCart` retorna a quantidade de itens no carrinho. A propriedade `total` retorna o total do carrinho, somando o valor de todos os produtos.

## 7.5. O Botão “Adicionar ao Carrinho”

Com o `CartService` pronto, podemos usá-lo no componente `ProductList`. Primeiro, devemos injetá-lo na classe, da seguinte forma:

```
// imports...
import { CartService } from 'src/app/cart.service';

export class ProductsListComponent implements OnInit {
 // code
 cartService = inject(CartService);
 // code
```

O botão Adicionar ao Carrinho chamará o seguinte método:

```
<!-- code -->
<mat-card-actions>
 <button
 mat-button
 (click)="onAddToCart(product)"
 [disabled]="product.unitsInStock==0"
 >
 ADD to cart
 </button>
</mat-card-actions>
<!-- code -->
```

O método onAddToCart(product) é mostrado abaixo:

```
//imports...
import {CartItem} from 'src/app/cart.dto'
import {CartService} from 'src/app/cart.service'

@Component({
 selector: 'app-products-list',
 templateUrl: './products-list.component.html',
 styles: []
})
export class ProductsListComponent implements OnInit {
 /// code

 onAddToCart(item: Product) {
 const cartItem: CartItem = {
 idProduct: item.id,
 name: item.name,
 quantity: 1,
 unitPrice: item.unitPrice
```

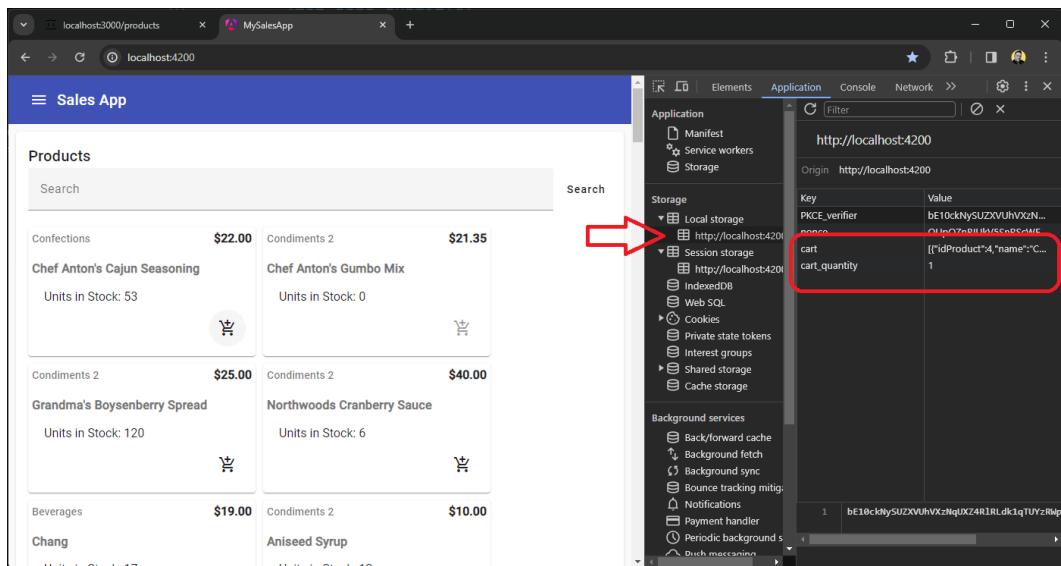
```

 }
 this.cartService.addItem(cartItem)
}
}

```

O método `onAddToCart` recebe como parâmetro um objeto do tipo `Product`. Com ele, podemos criar um novo objeto do tipo `CartItem`. Passamos o `id` do produto, nome, preço e quantidade. Com o objeto `CartItem` criado, chamamos o método `addItem` do `CartService`.

Quando testamos a aplicação e clicamos no botão, podemos inspirar as Dev Tools, na aba Aplicação, o JSON referente ao `localStorage`, conforme mostrado na seguinte imagem.



## 7.6. Criando o Ícone do Carrinho

Após o usuário adicionar alguns itens ao carrinho, podemos exibir um ícone com essa informação. Vamos adicionar esse ícone à barra superior da aplicação. Abra o arquivo `home.component.html` e após o título “Sales App” adicione o seguinte código:

```
<!-- src\app\home\home.component.html -->
<mat-sidenav-container class="sidenav-container">
 <mat-sidenav
 #Drawer
 class="sidenav"
 fixedInViewport
 [attr.role]=(isHandset$ | async) ? 'dialog' : 'navigation'"
 [mode]=(isHandset$ | async) ? 'over' : 'side'"
 [opened]=(isHandset$ | async) === false"
 >
 <app-menu></app-menu>
</mat-sidenav>

<mat-sidenav-content>
 <mat-toolbar color="primary">
 <button
 type="button"
 aria-label="Toggle sidenav"
 mat-icon-button
 (click)="drawer.toggle()"
 *ngIf="isHandset$ | async"
 >
 <mat-icon aria-label="Side nav toggle icon">menu</mat-icon>
 </button>

 Sales App

 <!-- CART BUTTON -->
 <div class="width-full"></div>
 <button mat-icon-button>
 <mat-icon [matBadge]="cartService.itensInCart" matBadgeColor="warn">
 shopping_cart</mat-icon>
 >
 </button>
</mat-toolbar>
<!-- Add Content Here -->
<div class="m-10">
 <router-outlet></router-outlet>
</div>
</mat-sidenav-content>
</mat-sidenav-container>
```

O botão possui a propriedade `[matBadge]`, que é um valor de notificação no ícone. O valor de `[matBadge]` é `cartService.itemsInCart`, que retorna o número de itens no carrinho. Inicialmente, recebemos um erro dizendo que `cartService` está indefinido. Ou seja, `CartService` não foi injetado no arquivo `home.component.ts`:

```
// src\app\home\home.component.ts
import {Component, inject} from '@angular/core'
import {BreakpointObserver, Breakpoints} from '@angular/cdk/layout'
import {AsyncPipe} from '@angular/common'
import {MatToolbarModule} from '@angular/material/toolbar'
import {MatButtonModule} from '@angular/material/button'
import {MatSidenavModule} from '@angular/material/sidenav'
import {MatListModule} from '@angular/material/list'
import {MatIconModule} from '@angular/material/icon'
import {Observable} from 'rxjs'
import {map, shareReplay} from 'rxjs/operators'
import {MenuComponent} from '../menu/menu.component'
import {RouterOutlet} from '@angular/router'
import {CartService} from '../cart.service'
import {MatBadgeModule} from '@angular/material/badge'

@Component({
 selector: 'app-home',
 templateUrl: './home.component.html',
 styleUrls: ['./home.component.css'],
 standalone: true,
 imports: [
 MatToolbarModule,
 MatButtonModule,
 MatSidenavModule,
 MatListModule,
 MatIconModule,
 MatBadgeModule, // adiciona aqui um módulo matBadge
 AsyncPipe,
 MenuComponent,
 RouterOutlet
]
})
export class HomeComponent {
 private breakpointObserver = inject(BreakpointObserver)
 cartService = inject(CartService) // adiciona aqui um serviço de carrinho

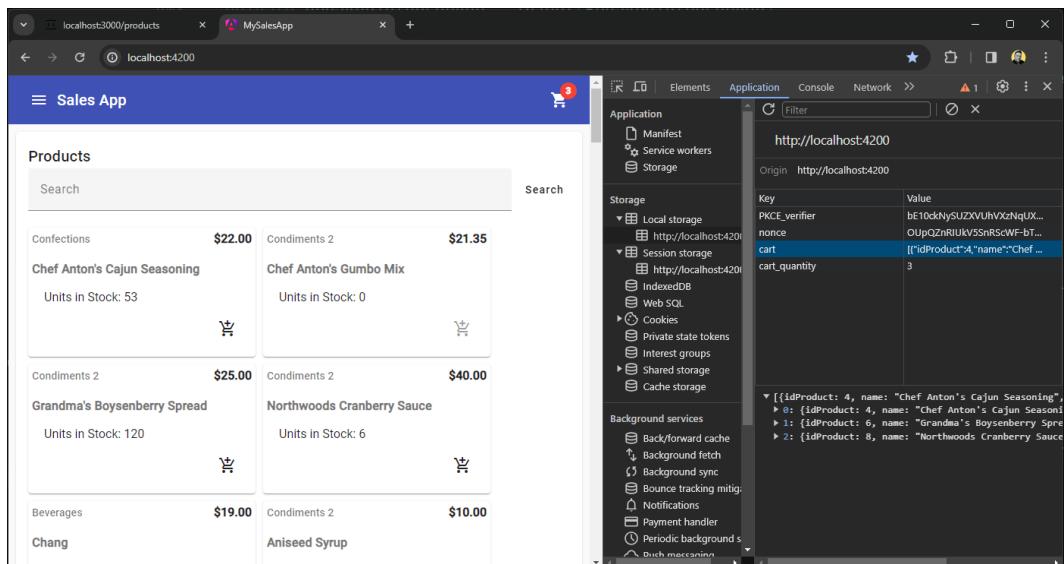
 isHandset$: Observable<boolean> = this.breakpointObserver
```

```
.observe(Breakpoints.Handset)
 .pipe(
 map((result) => result.matches),
 shareReplay()
)
}
```

O `width-full` vai puxar o botão para o lado direito da tela.

Não esqueça de adicionar o `MadBadgeModule` aos imports.

Após injetar `cartService`, temos uma aplicação que se parece com a seguinte imagem:



## 7.7. Adicionar uma Página de Checkout

Após criar o ícone do carrinho, iremos carregar uma página de “checkout”. Primeiro, adicione o parâmetro `routerLink` ao botão do carrinho, conforme segue:

```
<button mat-icon-button [routerLink]="/checkout">
 <mat-icon [matBadge]=“cartService.itensInCart” matBadgeColor=“warn”>
 shopping_cart</mat-icon>
 </button>
```

Adiciona o RouterLink à seção de importações no arquivo `home.component.ts`.

Crie o componente de checkout com o comando `ng g c checkout`. Adicione-o ao roteador da seguinte forma:

```
// imports...
import {CheckoutComponent} from './checkout/checkout.component'

export const routes: Routes = [
 // paths.....
 {path: 'checkout', component: CheckoutComponent}
]
```

O componente Checkout possui o seguinte código:

```
// src\app\checkout\checkout.component.ts
import {Component, OnInit, inject} from '@angular/core'
import {CartService} from '../cart.service'
import {MaterialModule} from '../material.module'
import {CartItem} from '../cart.dto'
import {CurrencyPipe} from '@angular/common'

@Component({
 selector: 'app-checkout',
 standalone: true,
 imports: [MaterialModule, CurrencyPipe],
 templateUrl: './checkout.component.html',
 styles: ``
})
export class CheckoutComponent implements OnInit {
 cartService = inject(CartService)
 public items: CartItem[] = []

 ngOnInit(): void {
 this.items = this.cartService.getItems()
 }

 onRemoveItem(item: CartItem) {
 this.cartService.removeItem(item)
 this.items = this.cartService.getItems()
 }
}
```

Injetamos a classe `cartService` que possui os métodos necessários para obter os dados do carrinho. No método `ngOnInit` populamos a variável `items` que será usada em uma lista no template do componente. Nesta lista temos um botão para remover um item do carrinho que chamará o método `onRemoveItem`. Este método usa o `CartService` para remover o produto do carrinho e, em seguida, atualiza a lista de itens.

O template do componente possui uma lista, onde exibimos o número de itens do produto, o nome do produto, o preço e um botão para remover o produto. Após a lista, exibimos o valor total do carrinho.

```
<!-- src\app\checkout\checkout.component.html -->
<mat-card>
 <mat-card-header>
 <mat-card-title>Checkout</mat-card-title>
 </mat-card-header>
 <mat-card-content>
 @for (item of items; track item.idProduct) {
 <div class="container center">
 <div class="item-50">{{ item.name }}</div>
 <div class="item-20">{{ item.unitPrice | currency }}</div>
 <div class="item-20">{{ item.quantity }}</div>
 <div class="item-10">
 <button mat-icon-button (click)="onRemoveItem(item)">
 <mat-icon>remove</mat-icon>
 </button>
 </div>
 </div>
 }

 Total: {{ cartService.total | currency }}
 </mat-card-content>
 <mat-card-actions align="end">
 <button mat-button>Next</button>
 </mat-card-actions>
</mat-card>
```

The screenshot shows a web browser window with two tabs: 'localhost:3000/products' and 'MySalesApp'. The main content area displays a 'Sales App' interface titled 'Checkout'. On the left, a sidebar menu lists 'Home', 'Categories', and 'Suppliers'. The main area shows a table of items in the cart:

	Item	Price	Quantity	Action
	Grandma's Boysenberry Spread	\$25.00	1	-
	Northwoods Cranberry Sauce	\$40.00	1	-
	Pavlova	\$17.45	1	-
	Teatime Chocolate Biscuits	\$9.20	1	-
	Mascarpone Fäboli	\$32.00	1	-
	Chef Anton's Cajun Seasoning	\$22.00	13	-

Total: \$409.65

Next

Com o código pronto, tente adicionar mais itens, volte para a página de finalização da compra e remova alguns. O próximo passo da finalização da compra seria obter os dados do cliente, processar o pedido, etc., o que também envolve escrever código no backend, o que não é o propósito deste trabalho.

# 8. Carregamento Dinâmico de Arquivos e Componentes

## 8.1. Visualizações Diferíveis

Visualizações Diferíveis são usadas para carregar conteúdo de forma assíncrona. Dizemos que um componente é Diferível usando `@defer`, como em:

```
@defer {
 <large-component/>
}
```

### 8.1.1. Exemplo

A tela `Home` do nosso aplicativo tem um `@for` com um componente enorme, muito completo com códigos que desenham um produto na tela, com borda, informações, botões, etc. Nossa primeira tarefa é criar um componente que isole esse código.

```
ng g c products/product-card
```

```
CREATE src/app/products/products-card/products-card.component.html (29 bytes)
CREATE src/app/products/products-card/products-card.component.ts (242 bytes)
```

### 8.1.2. Extrair o Componente

Após criar o componente, cortaremos o código que está no `@for` da `product-list` e colaremos no componente `product-card`:

```
<div>
 <form [formGroup]="searchForm" (...
 <div class="container">
 </div>
 </form>
 </div>
 <div class="container wrap">
 @for(product of products; track ...
 > <div style="width: 300px">...
 </div>
 } }
 </div>
} @else {
<loading-bar></loading-bar>
}
```

O template `src\app\products\products-card\products-card.component.html` é a parte que foi cortada do código, sem nenhuma alteração:

```
<div style="width: 300px">
 <mat-card class="p-5">
 <mat-card-title-group>
 <mat-card-subtitle> {{ product.category?.name }} </mat-card-subtitle>
 <mat-card-subtitle>
 <h3>{{ product.name }}</h3>
 </mat-card-subtitle>
 <div>
 {{ product.unitPrice | currency }}
 </div>
 </mat-card-title-group>
 <mat-card-content>
 Units in Stock: {{ product.unitsInStock }}
 </mat-card-content>
 <mat-card-actions align="end">
 <button
 mat-icon-button
```

```
[disabled]="product.unitsInStock == 0"
(click)="onAddToCart(product)"
>
<mat-icon>add_shopping_cart</mat-icon>
</button>
</mat-card-actions>
</mat-card>
</div>
```

O script do componente ProductCard tem, além dos imports adequados, o @Input do product, para que possa ser referenciado, e o método onAddToCart.

```
import {CommonModule} from '@angular/common'
import {Component, Input} from '@angular/core'
import {MaterialModule} from '../../../../../material.module'
import {Product} from '../product.dto'

@Component({
 selector: 'app-products-card',
 standalone: true,
 imports: [CommonModule, MaterialModule],
 templateUrl: './products-card.component.html',
 styles: ``
})
export class ProductsCardComponent {
 @Input() product: Product

 onAddToCart(product: Product) {
 console.log('TODO')
 }
}
```

Com o componente pronto, podemos voltar para product-list.e, no @for, inserir o componente product-card:

```
<!-- code -->
<div class="container wrap">
 @for(product of products; track product.id){
 <app-products-card [product]="product"></app-products-card>
 }
</div>
<!-- code -->
```

### 8.1.3. Usando @defer

Até agora, não usamos `@defer`, de fato, não mudamos nada para angular, apenas extraímos um pedaço de código HTML para um novo componente. Vamos agora inserir o `@defer`:

```
<div class="container wrap">
 @for(product of products; track product.id){ @defer {
 <app-products-card [product]="product"></app-products-card>
 } }
</div>
```

Ao incluir `@defer`, dizemos ao angular para carregar este componente de forma assíncrona. A primeira mudança que você pode notar está na criação dos arquivos que compõem a aplicação. Volte ao terminal, onde você executou a aplicação com `ng serve`:

```
Application bundle generation failed. [0.281 seconds]

Initial Chunk Files | Names | Raw Size
styles.css | styles | 94.17 kB |
polyfills.js | polyfills | 83.46 kB |
main.js | main | 83.18 kB |
chunk-SBROAJ46.js | - | 2.03 kB |

| Initial Total | 262.84 kB

Lazy Chunk Files | Names | Raw Size
chunk-RM66TBII.js | products-card-component | 3.39 kB |

Application bundle generation complete. [0.269 seconds]
Page reload sent to client(s).
```

Veja em `Lazy Chunk Files` que um arquivo foi criado para ser o host do componente `products-card`. Com isso, garantimos que o arquivo seja baixado apenas quando necessário, ou seja, será carregado assim que `product-list` for carregado.

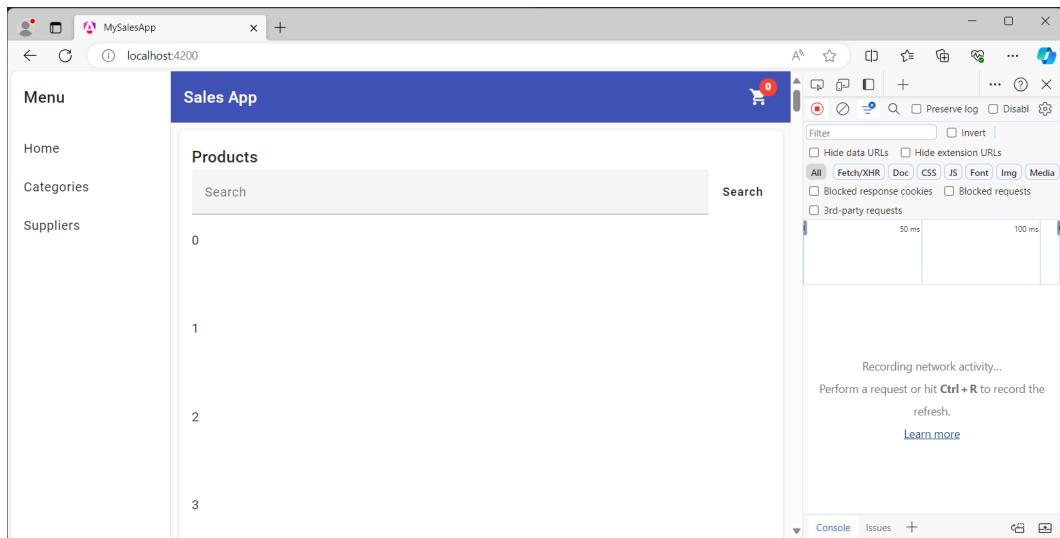
### 8.1.4. Usando `@viewport`, `@placeholder` e `@loading`

`@defer` não apenas cria um chunk separado, mas também permite o download deste chunk de forma assíncrona apenas quando o próprio componente é exibido na tela. Para isso, usamos o parâmetro `on viewport` na criação do `@defer`.

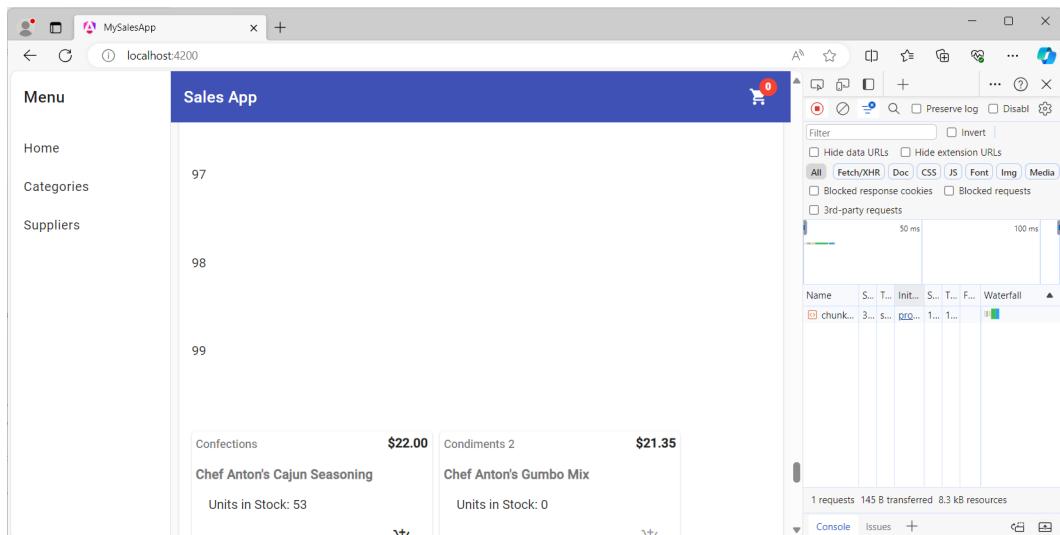
Para ver o `viewport` em ação, precisamos adicionar um `for` com um elemento visual na tela para que a lista de produtos possa ser acessada rolando a página para baixo. Veja:

```
<div class="container wrap">
 @for (item of [].constructor(100); track $index) {
 <div style="height:100px; width:100%">{$index}</div>
 } @for(product of products; track product.id){ @defer (on viewport){
 <app-products-card [product]="product"></app-products-card>
 } @placeholder {
 Placeholder
 } @loading {
 Loading
 }
 </div>
```

Usamos `@for (item of [].constructor(100)` apenas para criar 100 iterações do loop `for`, e para cada uma, adicionamos uma div com 100 de altura. Isso produzirá o seguinte efeito:



Note que, o componente `<app-products-card>` ainda não apareceu na tela. Além disso, observe que temos o DevTools aberto na aba Networking. Agora, role a página para baixo até ver a listagem de produtos:



À medida que você rola para baixo até a listagem de produtos, o `chunk` é baixado de forma assíncrona e o conteúdo é exibido. Por um curto período, você pode ver o texto Carregando... indicando que o `chunk` está sendo carregado. Se desejar, você pode configurar

o tempo de carregamento, adicione: `@loading (mínimo 5s) { ... }`. Dessa forma, a mensagem de carregamento é exibida na tela por 5 segundos.

Por meio dessa estratégia, podemos ter quase toda a aplicação Angular carregada de forma assíncrona, apenas usando `@defer` nos componentes mais complexos. Dividir a aplicação em pedaços menores é uma das melhores maneiras de tornar a aplicação menor para o usuário, o que impacta principalmente os dispositivos móveis.

## 8.2. Estratégias para tornar a aplicação ainda menor

Se você usou o `@defer` do tópico anterior, verá que a aplicação Angular é dividida nestes arquivos:

```
Application bundle generation failed. [0.281 seconds]

Initial Chunk Files | Names | Raw Size
styles.css | styles | 94.17 kB |
polyfills.js | polyfills | 83.46 kB |
main.js | main | 83.18 kB |
chunk-SBR0AJ46.js | - | 2.03 kB |

| Initial Total | 262.84 kB

Lazy Chunk Files | Names | Raw Size
chunk-RM66TBII.js | products-card-component | 3.39 kB |

Application bundle generation complete. [0.269 seconds]
Page reload sent to client(s).
```

O arquivo `main.js` tem 81kb e contém todas as telas (Categorias, Fornecedores, etc). À medida que criamos mais e mais telas, esse tamanho tende a crescer e precisamos dividir esse tamanho em tamanhos menores. Felizmente, fazer isso no Angular é extremamente fácil. Você só precisa definir o que deseja carregar dinamicamente.

Vamos supor que queremos carregar dinamicamente apenas a tela de Categorias. Ou seja, os arquivos para esta tela só serão baixados quando o usuário acessar a rota `/categories`. Para fazer isso, modificamos o `app.routes.ts` para:

```

import {Routes} from '@angular/router'
// imports

export const routes: Routes = [
{
 path: 'categories',
 loadComponent: () =>
 import('./categories/categories.component').then(
 (c) => c.CategoriesComponent
)
}
// ... another routes
]

```

Em vez de usar a propriedade `component`, estamos usando `loadComponent`, que é uma função que usará `import` para carregar dinamicamente o arquivo `'./categories/categories.component'` e, após carregá-lo, retornar `c.CategoriesComponent` que é o próprio componente! Quando analisamos os arquivos da aplicação, temos:

```

$ ng serve

Initial Chunk Files | Names | Raw Size
styles.css | styles | 94.17 kB
polyfills.js | polyfills | 83.46 kB
main.js | main | 62.86 kB
chunk-SBROAJ46.js | - | 2.03 kB
chunk-GZWSFR4H.js | - | 1.85 kB

| Initial Total | 244.37 kB

Lazy Chunk Files | Names | Raw Size
chunk-L2L67LNX.js | categories-component | 19.10 kB
chunk-RM66TBII.js | products-card-component | 3.39 kB

Application bundle generation complete. [3.876 seconds]
Watch mode enabled. Watching for file changes...
→ Local: http://localhost:4200/

```

Note que o arquivo `main.js` está um pouco menor, pois não contém mais o componente `Categories`, e que um novo chunk do `categories-component` foi criado com aproximada-

mente 19kb, que só será baixado se o usuário acessar a rota /categories.

# 9. Atualizações no Futuro

Este livro foi inicialmente concluído em Angular 14, em 2022. Sempre que houve atualizações no Angular, criamos um guia de atualização neste último capítulo.

No Angular 17, houve muitas mudanças, então decidimos reescrever o livro inteiramente com as novas funcionalidades desta versão, removendo qualquer código obsoleto. Ao fazer isso, tentamos garantir que você fique satisfeito com este livro.

Quando o Angular 18 for lançado, atualizaremos o livro novamente com as novas funcionalidades e mudanças que estão por vir.

Sinta-se à vontade para solicitar mais conteúdo escrevendo para [danieljfa@gmail.com](mailto:danieljfa@gmail.com)