



O que é Angular?

Angular é uma plataforma e framework para construção da interface de aplicações usando HTML, CSS e, principalmente, JavaScript, criada pelos desenvolvedores da Google. Ele possui alguns elementos básicos que tornam essa construção interessante.

Dentre os principais, podemos destacar os componentes, templates, diretivas, roteamento, módulos, serviços, injeção de dependências e ferramentas de infraestrutura que automatizam tarefas, como a de executar os testes unitários de uma aplicação.

Angular nos ajuda a criar [Single-Page Applications](#) com uma qualidade e produtividade surpreendente!

Alguns outros pontos dessa plataforma que merecem destaque são o fato de que ela é *open source*, possui uma grande comunidade, existem várias empresas utilizando e tem muito material de estudo para quem deseja se aperfeiçoar.

Preparando do ambiente de desenvolvimento

- Instalar a ultima versão do [Node.js](#) (sempre baixar a versão LTS), vamos utilizar o pacote [NPM](#) contido nele para baixar as ferramentas que precisamos para esse projeto.
- Baixar o [Visual Studio Code](#) que é um editor de texto [open source](#) poderoso, que nos permite criar e gerenciar projetos Angular.

Criação do nosso primeiro projeto Angular

Antes de dar incio ao nosso projeto precisamos entender que o Angular é um conjunto de diversos *frameworks*, *libs*, bibliotecas, e por aí vai. A configuração desta infraestrutura é essencial para construirmos nosso projeto, acessarmos ele em nosso navegador, empacotarmos e gerarmos o arquivo para a produção. Gastaremos um tempo considerável antes de começarmos a programar efetivamente.

- Acesse a documentação do [Angular CLI](#) (*Angular Command Line Interface*), trata-se de um projeto que nos auxilia na construção de projetos, a partir do zero, sem necessidade de nos preocuparmos com toda essa infraestrutura, bastando a execução de alguns comandos.
- Vamos abrir o Prompt de Comando no próprio VS Code apertando as teclas `Ctrl + ``, esse recurso facilita nossa administração do projeto, concentrando todos os nossos comandos, tanto CMD quando de código, no mesmo local.
- Verificaremos se o npm está instalado com `npm --version`. Será retornado a sua versão atual. Para solicitar a instalação domente, ou seja, para que esta ferramenta esteja disponível em qualquer diretório da aplicação, utilizaremos o comando `npm install -g @angular/cli`.
- Verifique se o Angular foi instalado com sucesso utilizando comando `ng --version`
- Vamos criar um onovo projeto com comando `ng new nomeDoProjeto`.
- Escolha a opção **y** aperte o enter, escolha o **CSS** e aperte enter novamente, agora é só esperar que será baixado no seu computador um projeto pré configurado.
- Agora vamos entrar na pasta do projeto pelo comando `cd nomeDoProjeto`.
- Dentro da pasta podemos rodar nosso primeiro projeto com o comando `ng serve --open`, o `--open` abre o navegador automaticamente, assim que o projeto for construído.

Entendendo o projeto criado

Para entender a estrutura dos arquivos o Angular precisamos entender primeiro que tudo no Angular é um **componente**.

Componente é um artefato que guarda, um comportamento ou “o que fazer” (TypeScript), a apresentação (CSS) e a marcação ou estrutura (HTML)

TypeScript

Para entender o que é esse TypeScript você precisa acessar o documento de TypeScript onde falamos de forma básica os principais conceitos desse superset do JavaScript.

Mas onde se encontram estes componentes, em nosso projeto?

Tudo que formos criar ao longo do curso ficará dentro de “app”, em que há `app.component.ts`. Abrindo-o, veremos que, basicamente, temos uma classe do **ECMAScript 6**, com um *Decorator* anotado com `@Component`, o qual torna a classe um componente. O *Decorator* é uma sintaxe especial do Angular, do TypeScript, em que é possível incluir uma **metainformação** sobre uma determinada classe, no caso.

O que é uma metainformação?

Ao criarmos instâncias desta classe, criamos objetos. Estamos incluindo mais uma informação desta classe, que diz respeito ao framework. Então, a classe `AppComponent` só é um componente porque está anotada com `@Component`. Nele, existe um `selector: 'app-root'`, mesmo nome encontrado em `index.html`.

Este *selector* nos permite utilizar o componente em templates em sua forma declarativa, então, todo o conteúdo de `app.component.ts`, sua apresentação, o que ele faz, seu CSS, são acessados por meio dele. Também neste arquivo, há `templateUrl: './app.component.html'`, que informa a apresentação deste componente.

Se abirmos `app.component.html`, encontraremos o código referente à apresentação que vemos na página do navegador. Então, o Angular carregará, exibindo o primeiro componente, e seu template. Voltando a `app.component.ts`, o `styleUrls: ['./app.component.css']` informa onde se localiza o CSS utilizado por este componente.

Quando uma aplicação Angular é carregada pela primeira vez, sabemos que é a `<app-root>` que será carregada, pois ela, na sua forma declarativa no *selector*, é que está no `index.html`.

É importante comentar sobre dois arquivos existentes na pasta src:

- **index.html** > Arquivo HTML principal do projeto, nesse arquivo iremos chamar a tag criada no **app-component**, pelo próprio Angular `<app-root>`, essa tag é

repositável por “chamar” o componente **app**, nesse componente temos as rotas para direcionar todo o resto da nossa demais páginas através de rotas aplicação.

- **styles.css** > Esse arquivo é responsável por criar estilos globais para todo o projeto, ou seja, tudo que você aplicar aqui como estilo css, será aplicado para toda a SPA

Primeiro contato com data binding

Antes de prosseguirmos, nos atentemos ao título que aparece na página da aplicação: “Welcome to app!”. Ao voltarmos ao nosso template, e procurarmos o template do componente que sabemos que é carregado, `app.component.html`, veremos `Welcome to {{ title }}`!. No entanto, não é “title” que vemos na tela, e sim, “app”.

O Angular se tornou famoso na época em que foi lançado por causa de um recurso chamado **Data binding**, ou “associação de dados”. Assim como em `app.component.ts` há `title = 'app'`, existe uma propriedade no componente, no caso chamada `title`. A fonte de dados é a propriedade do nosso componente, e seu valor é `app`.

O *Data binding* implica em uma associação de dados com uma fonte de dados que, no nosso caso, está no componente, com seu template (nomenclatura do Angular), ou *view*. Nele, quando encontramos esta sintaxe chamada de **Angular expression** (AE), e quando o Angular for renderizar este template do componente, ele se deparará com uma lacuna, que neste caso está apontando para a propriedade `title`.

Esta propriedade existe em `app.component.ts`, então o Angular acessará seu valor e o jogará no template. Isso é interessante pois para realizarmos uma mudança dessas em JavaScript, tradicionalmente, precisamos manipular o DOM, realizando o `document.querySelector()`, selecionando o elemento e mudando o `textContent`.

A ideia do Angular com *Data binding* é justamente evitar perda de tempo manipulando DOM, e que possamos fazer algo que realmente vá agregar ao nosso cliente. Sendo assim, se trocarmos `app` por `alurapic` em `app.component.ts`, e salvarmos o projeto, sabemos que o novo valor para o template será outro.

Teríamos que recarregar a página no navegador, porém, ao abrirmos a página, a mudança já estará feita. Esta é uma vantagem fantástica do Angular CLI: qualquer alteração na aplicação é feita instantaneamente no navegador. Isso é perfeito quando se trabalha com dois monitores, pois em um vamos desenvolvendo, e no outro vemos as modificações sendo feitas.

Adicionando Bootstrap ao projeto

No Angular, quando precisamos importar um CSS global como o Bootstrap, não adicionamos o link do bootstrap no arquivo index, como fazíamos nas aulas de desenvolvimento web básico. Isto porque esses arquivos CSS precisam estar no processo de *build*, de construção da nossa aplicação, tanto no ambiente de desenvolvimento quanto no ambiente de produção.

Sendo assim o processo para a inserção desse framework é esse:

- Pause a aplicação e no CMD, dentro da pasta nosso primeiro projeto com o comando `ng new nomeDoProjeto`.
- Escolha a opção **y** aperte o enter, escolha o **CSS** e aperte enter novamente, agora é só esperar que será baixado no seu computador do projeto iremos usar os comandos:

```
npm install bootstrap@latest --save
```

```
npm install jquery@latest --save
```

```
npm install popper.js --save
```

- Após isso vamos abrir o arquivo `angular.json`, que possui uma chave denominada `build`, dentro do qual está `styles` e adicionamos o caminho do arquivo bootstrap que baixamos via NPM.

```
"styles": [  
  "src/styles.css",  
  "./node_modules/bootstrap/dist/css/bootstrap.min.css"  
],  
"scripts": [  
  "./node_modules/jquery/dist/jquery.js",  
  "./node_modules/bootstrap/dist/js/bootstrap.js",  
  "./node_modules/popper.js/dist/umd/popper.min.js"  
]
```

Adicionando Font Awesome no projeto

```
ng add @fortawesome/angular-fontawesome
```

[Clique aqui](#) para saber como utilizar essa ferramenta em seu projeto.

Criando o primeiro componente

Um componente, nada mais é que uma classe demarcada com o decorator

`@Component()`, que define metadados para que o angular possa identificar que a classe é um componente e configurar as características para este componente.

Vamos analisar, então a estrutura do componente principal de uma aplicação angular:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {title = 'app';
}
```

Para utilizarmos o decorator `Component` deve em primeiro lugar importar o modulo `'@angular/core'`

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

Agora dentro das configurações do atributo, temos os metadados

selector: que define qual será a tag para o componente sendo criado

templateUrl: é o caminho para o template do componente, podemos substituir esta propriedade por “template”, neste caso definimos um template inline, incluindo por exemplo o nosso html do componente:

```
template:'<h1>Template do componente</h1>'
```

ou ainda podemos usar a interpolação para definirmos um código com mais de uma linha, ex.:

```
template:'<h1>Template do componente</h1>
<span>Angular é divertido demais</span>'
```

styleUrls: aqui definimos o caminho para o arquivo(s) css que serão utilizados pelo o componente.

uma coisa bem interessante é que estes css's ficam limitados ao componente, não interferindo no restante da aplicação.

podemos também definir o css inline, da seguinte forma:

```
styles:[ 'h1{font-size:10pt}' ]
```

ou com interpolação:

```
styles:[ '  
h1{font-size:18pt}  
h2{font-size:16pt}  
' ]
```

providers: outro item que é bastante utilizado no decorator @Component é o providers, que define um array com os serviços que poderão ser utilizados no construtor do componente, via injeção de dependência, iremos ver mais detalhes sobre injeção de dependências quando formos tratar de serviços.

A lista de outros metadados do decorator @Component é mostrado a seguir:

- **animations** - lista de animações do componente;
- **changeDetection** - estratégia de detecção de mudança do componente;
- **encapsulation** - tipo de encapsulamento utilizado pelo componente;
- **entryComponents** - lista componentes que são dinamicamente inseridos pelo componente;
- **exportAs** - nome pelo qual a instância do componente é exportada dentro do template;
- **host** - mapeamento de propriedades para bindings de eventos, propriedades e atributos;
- **inputs** - list de propriedades para data-bind de entrada do component;
- **interpolation** - marcadores de interpolação customizados usados no template do component;
- **moduleId** - ES/CommonJS - Id do modulo onde o componente é definido;
- **outputs** - lista de propriedade que expõem eventos aos quais outros componetes podem fazer uma subscrição;
- **queries** - consultas que podem ser injetadas no componente;
- **viewProviders** - lista de provedores disponíveis para este componente e sua respectiva view Children;

Agora vamos criar um novo componente através do terminal no próprio Vs Code, digite o seguinte comando:

```
ng generate component header
```

ou simplesmente:

```
ng g c header
```

Nesse caso a cima, *header* é o nome do nosso componente, que por sua vez pode ter qualquer outro nome.

Notamos que foram gerados os arquivos como mostrado acima, e também o modulo principal da aplicação (neste caso único), foi também atualizado.

Abrindo o arquivo `app.module` podemos perceber que o nome do nosso novo componente com adicionado ao `declarations` como `HeaderComponent`. Caso isso não aconteça será preciso adicionar manualmente, senão seu componente não será reconhecido como um modulo acessível do seu projeto.

Abra o arquivo `header.component.ts`, o básico para o funcionamento do componente já foi criado. Agora vamos editar o arquivo `html` do nosso componente, utilizando o `bootstrap` que já instalamos momentos antes.

Basta entrar na documentação do `bootstrap` e procurar por um modelo de `navbar` de sua preferencia e adicionar ao documento:

```
header.component.html
```

Após esse passo, adicione ao documento `app.component.html` a tag:

```
<app-header></app-header>
```

Essa tag é gerada no arquivo `TypeScript` do componente **header**.

Pronto, agora é só digitar no terminal o seguinte comando:

```
ng serve -o
```

O `-o` serve para o `Angular` abrir o seu `browser` automaticamente, quando você já estiver com a página aberta e precisar rodar novamente o `ng serve`, não é necessário o `-o`, senão vai ser aberta outra página.

Adicionando rotas para os componentes

Antes de trabalhar com rotas precisamos criar mais dois componentes em nosso projeto:

```
ng g c home e ng g c about
```

Componentes criados, agora abra o arquivo `app-routing.module.ts` esse é o estado inicial dele ao ser criado junto com o seu projeto:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})

export class AppRoutingModule { }

```

Foi criada uma constante (const) **routes**, ela é do tipo objeto **Routes** que vem do módulo de rotas do próprio Angular CLI. É nela que vamos adicionar nossas rotas, isso deve ser feito da seguinte forma:

```
const routes: Routes = [
  { path:'', redirectTo:'home', pathMatch:'full'},
  { path:'home', component: HomeComponent },
  { path:'about', component: AboutComponent },
];

```

Quando digitamos essas rotas o próprio Vs Code importa automaticamente esses componentes no arquivo de rotas. Ao fim desse processo nosso arquivo `app-routing.module.ts` deve estar assim:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from '../home/home.component';
import { AboutComponent } from '../about/about.component';

const routes: Routes = [
  {path:'', redirectTo:'home', pathMatch: 'full'},

```

```

    {path:'home', component: HomeComponent},
    {path:'about', component: AboutComponent}
  ];

  @NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
  })

  export class AppRoutingModule { }

```

Agora as rotas já estão criadas, porém se entrarmos em nossa página no browser, nós ainda não conseguimos acessar o conteúdo dela, pois ainda falta um detalhe, o **RouterOutlet**.

RouterOutlet

Para que possamos ver o conteúdo do nosso componente precisamos adicionar uma diretiva específica do angular que irá funcionar com um container para os componentes configurados nas rotas.

A diretiva é:

```
<router-outlet></router-outlet>
```

Vamos então alterar o arquivo `app.component.html` com o seguinte conteúdo:

```

<app-header></app-header>
<router-outlet></router-outlet>

```

Voltando novamente para o browser, podemos verificar que o conteúdo do template do nosso componente foi apresentado

Links para o componente

Para utilizar links precisamos utilizar outra importante diretiva do angular `RouterLink`. No arquivo `header.component.html` vamos incluir nossos links:

```

<li class="nav-item active">
  <a class="nav-link" routerLink="/home">Home <span class="sr-only">(págin

```

```
</li>
<li class="nav-item">
  <a class="nav-link" routerLink="/about">About</a>
</li>
```

É importante dizer que a diretiva `routerLink` é também um atributo da tag `<a>` e precisa ser escrita com o padrão `camelCase` que já estamos habituados, caso contrario não irá funcionar.

Data binding

Vamos relembrar o conceito de data binding para aplicá-lo de forma mais profunda, o *Data binding* implica em uma associação de dados com uma fonte de dados que, no nosso caso, está no componente, com seu template (nomenclatura do Angular), ou *view*. Nele, quando encontramos esta sintaxe chamada de **Angular expression** (AE), e quando o Angular for renderizar este template do componente, ele se deparará com uma lacuna. A ideia do Angular com *Data binding* é justamente evitar perda de tempo manipulando DOM, e que possamos fazer algo que realmente vá agregar ao nosso cliente.

De forma resumida os bindings em angular o *data binding* é a ,tras da ****Então** basicamente os formas de como a **View** interage com **Model/(Component)**.

Vamos entender os tipos de binding disponíveis:

Interpolation

Interpolation ou interpolação em português é o tipo mais comum de binding, ele é utilizado para transportar o valor de propriedades e retorno de métodos do componente para o template HTML.

Na interpolação utilizamos o nome da propriedade ou método dentro de duas chaves `{{}}`, em inglês *curly brackets*

ex.:

```
<h1>{{propriedade}}</h1>
```

Property binding

Aqui a coisa começa a ficar mais interessante, pois com Property binding podemos setar o valor de qualquer atributo de um elemento html. Podemos utilizar a interpolação para tanto da seguinte forma:

```
<input type="text" value="{{propriedade}}">
```

Podemos também utilizar os colchetes “[]” para envolver o atributo do elemento html (ex.: [value] = “propriedade”), desta forma o template irá receber o valor da propriedade do componente diretamente no atributo sem a necessidade de utilizar a interpolação, exemplo:

```
<input type="text" [value]="propriedade">
```

Também conhecido como *one way databinding* é utilizado para enviar informações

Event binding

Neste tipo de binding enviamos valores vindos do template HTML para o componente, este tipo de binding é importantíssimo por ser utilizado para manipular os eventos e interações com o template e enviados para o componente.

E como o próprio nome diz, event binding, utilizamos de eventos para que possamos disparar os métodos ou expressões para atualizarmos a model a partir de um elemento Html presente no template.

Por exemplo, em um input field, podemos disparar um evento enquanto o campo está sendo preenchido:

```
<input (input)="model.skype=$event.target.value" type="text" class = "form-control"/>
```

Em angular definimos o event binding utilizando os parenteses “(nomeevento)”, ex.: para o evento “OnChange” utilizamos (change), para o “OnClick” temos (click) no caso (input) representa um evento que é capturado quando uma tecla é pressionada. desta forma podemos definir uma expressão para setar a model, ex.:

```
(input)="model.skype=$event.target.value"
```

ou podemos também chamar um método, ex.:

```
(input)="metodoHandler($event)"
```

e também transferir o valor do template para o componente através do objeto \$event.

Two way binding

Diferente do angular Js, o binding de duas vias em angular algumas otimizações de performance foram implementadas, e por este motivo não existe mais este tipo de binding e sim uma combinação do *attribute binding* e *event binding* quando precisamos atualizar o template e o componente existe a diretiva ngModel, utilizada nos elementos de um formulário que simplifica o modo de fazermos o binding de duas vias.

Neste tipo de binding, como explicado, anteriormente temos a união dos dois tipos de binding em uma sintaxe conhecida como “banana in the box” ou [(ngModel)], como podemos ver esta sintaxe realmente lembra duas bananas, representando os parenteses “()”, e a caixa, representando os colchetes “[]”.

Podemos então utilizar esta diretiva da seguinte forma:

```
<div class="text-center">

  <div class="col-md-2">

    <div class="input-group">

      {{nome}}

      <input [(ngModel)]="nome" type="text" class="form-control text-

    </div>

  </div>

</div>
```

Neste exemplo ao se alterar o valor do campo nome o mesmo é refletido no html através do binding interpolação {{nome}}.

Directives

- **ngFor**

Esta é uma diretiva para processar cada item de um objeto iterável, gerando uma marcação para cada um. Ela é conhecida como uma *diretiva estrutural* porque pode alterar o layout do DOM adicionando e removendo elementos DOM de visualização.

Assim, a diretiva **ngFor** é útil para gerar conteúdo repetido, como uma lista de clientes, elementos de um menu suspenso e assim por diante. Cada item processado do iterável tem variáveis disponíveis em seu contexto de modelo.

Exemplo:

```
<div *ngFor="let item of listaItens">
  <h1> Título: {{ item.titulo }} </h1>
  <p> Descrição: {{ item.descricao }} </p>
</div>
```

- **ngIf**

Uma diretiva estrutural que inclui condicionalmente um modelo com base no valor de uma expressão coagida a Boolean. Quando a expressão é avaliada como verdadeira, Angular processa o modelo fornecido em uma cláusula `then` e, quando falso ou nulo, Angular processa o modelo fornecido em uma cláusula `else` opcional.

Exemplo:

- Sintaxe básica:

```
<div *ngIf="condition">Content to render when condition is true.</div>
```

- Sintaxe adicionando o bloco de else:

```
<div *ngIf="condition; else elseBlock">Content to render when condition is true.</div>
<ng-template #elseBlock>Content to render when condition is false.</ng-template>
```

- **[(ngModel)]**

Essa diretiva é utilizada principalmente para que o Angular consiga escutar o que está sendo digitado no template e armazenar esse dado em uma variável dentro do arquivo TypeScript, teremos muitos exemplos de utilização dessa diretiva principalmente em

formulários de login e cadastros, segue aqui um desses exemplos em um formulário de login:

```
<div class="form-group">

  <label for="email">Endereço de email</label>

  <input type="email" class="form-control" id="email" aria-describedby="emailHelp"
  placeholder="Seu email" [(ngModel)]="userLogin.usuario">

</div>

<div class="form-group">

  <label for="senha">Senha</label>

  <input type="password" class="form-control" id="senha" placeholder="Senha"
  [(ngModel)]="userLogin.senha">

</div>
```

Projeto e-commerce

Vamos criar um projeto para entender as teorias ensinadas a cima.

Como vimos basta digitar no CDM, dentro da pasta que você deseja criar o projeto o seguinte comando:

```
ng new ecommerce
```

Nesse caso o nome do projeto será ecommerce.

Criando seu primeiro componente

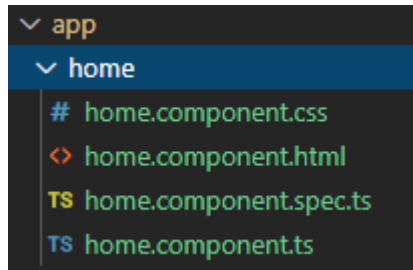
Vamos digitar o seguinte comando no CMD, porém agora dentro da pasta do projeto novo criado:

```
OUTPUT  TERMINAL  DEBUG CONSOLE  PROBLEMS

MrThi@LAPTOP-L7KTGEU5 MINGW64 ~/OneDrive/Área de Trabalho/Generation/PROJETO-GUIA/prints
$ cd ecommerce

MrThi@LAPTOP-L7KTGEU5 MINGW64 ~/OneDrive/Área de Trabalho/Generation/PROJETO-GUIA/prints/ecommerce (master)
$ ng g c home
```

Vamos verificar a estrutura desse componente em meu documento:



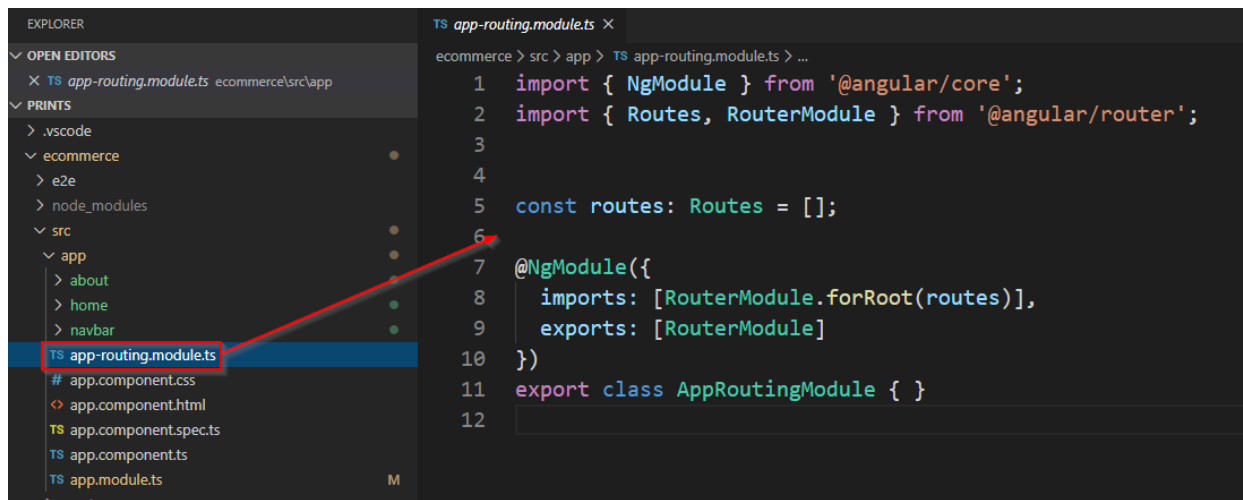
Podemos ver 4 arquivos criados, porém iremos trabalhar com 3:

- **home.component.css**: Responsável pelos estilos do componente;
- **home.component.html**: Template responsável pela estrutura da página (HTML);
- **home.component.ts**: Responsável pela lógica do meu componente.

Trabalhando com rotas entre componestes

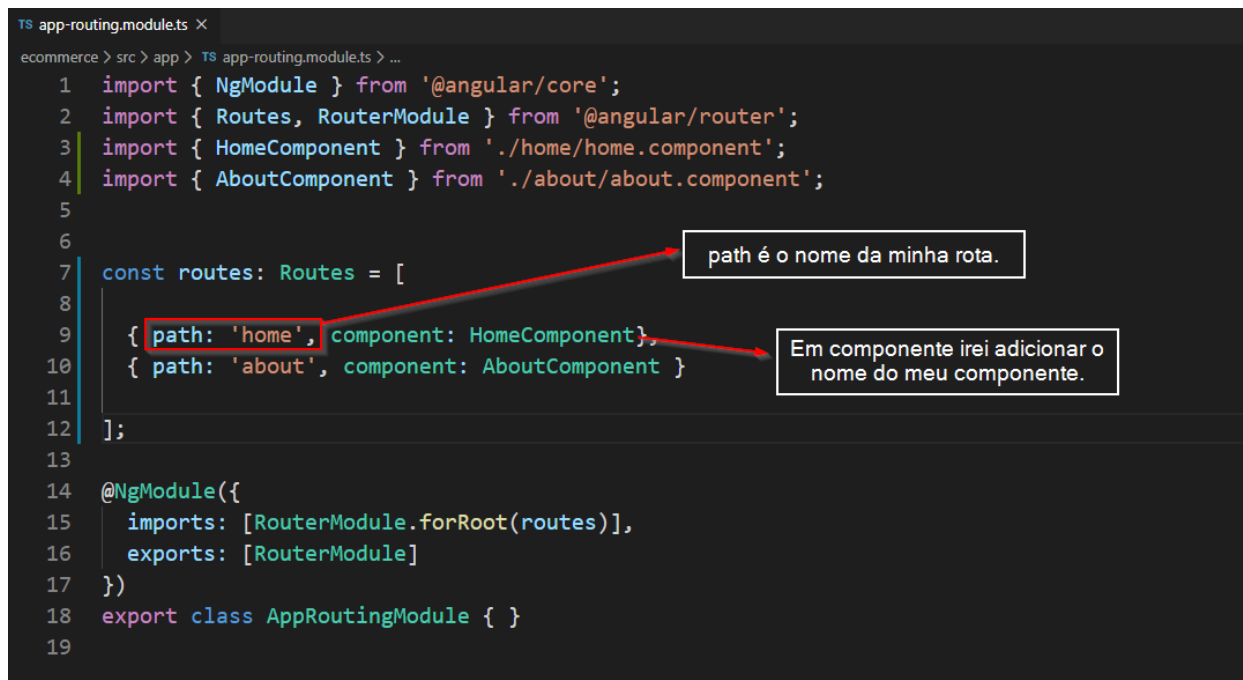
Para entender rotas de uma forma prática vamos criar mais 2 componentes, o **navbar** e **about**.

Após a criação entre no arquivo **app-routing.module.ts** que se encontra dentro da pasta **app**, esse vai ser o documento inicial que você vai encontrar:



```
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3
4
5 const routes: Routes = [];
6
7 @NgModule({
8   imports: [RouterModule.forRoot(routes)],
9   exports: [RouterModule]
10 })
11 export class AppRoutingModule { }
```

Vamos criar agora nossas rotas para acessar o componente **about**, através do componente **home** e vice versa:



```
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { HomeComponent } from '../home/home.component';
4 import { AboutComponent } from '../about/about.component';
5
6
7 const routes: Routes = [
8   { path: 'home', component: HomeComponent },
9   { path: 'about', component: AboutComponent }
10 ];
11
12
13 @NgModule({
14   imports: [RouterModule.forRoot(routes)],
15   exports: [RouterModule]
16 })
17 export class AppRoutingModule { }
```

path é o nome da minha rota.

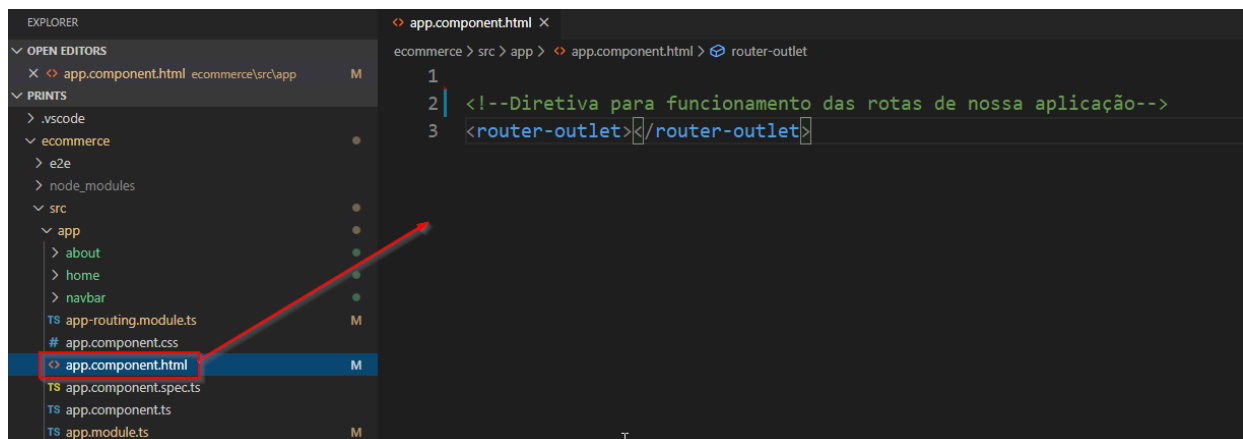
Em componente irei adicionar o nome do meu componente.

Como vocês podem observar adicionei ao Array Routes[] minhas rotas, de acordo com minha necessidade.

Agora vamos no template do componente **home** e vamos adicionar um routerLink para nossa rota **about**:

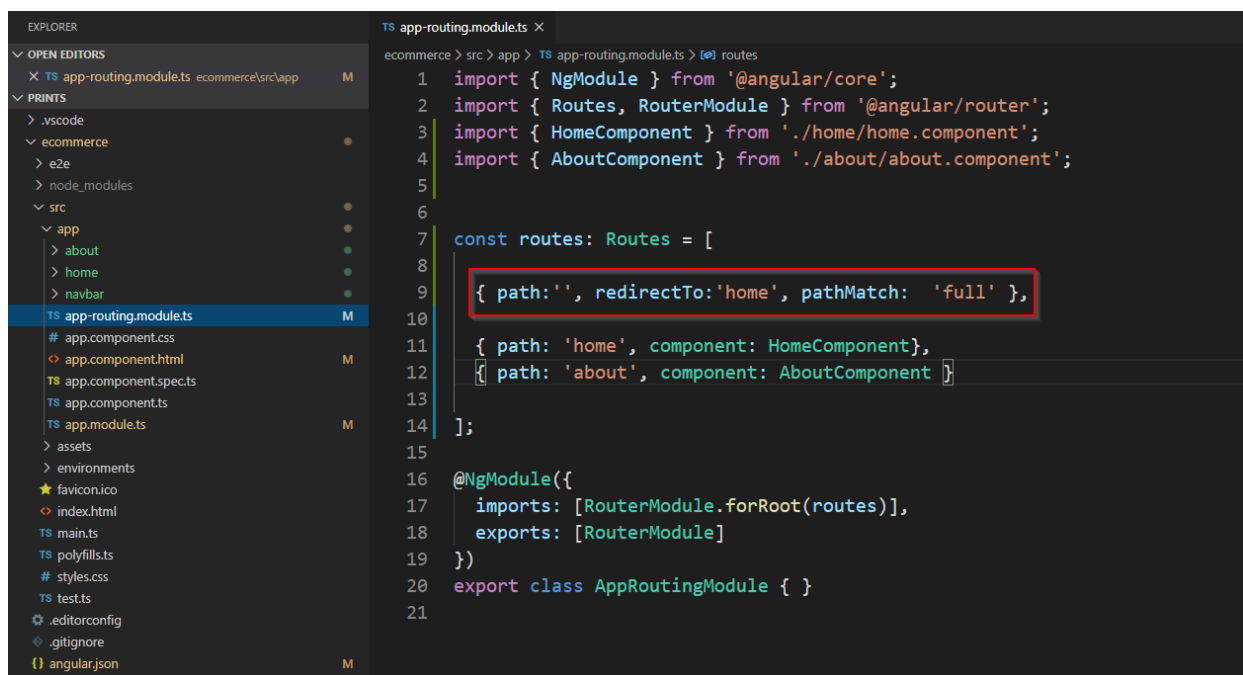


Antes de testar se deu certo precisamos colocar a diretiva `<router-outlet></router-outlet>` no nosso `app.component.html`, sem essa diretiva as rotas da nossa aplicação não funcionarão:

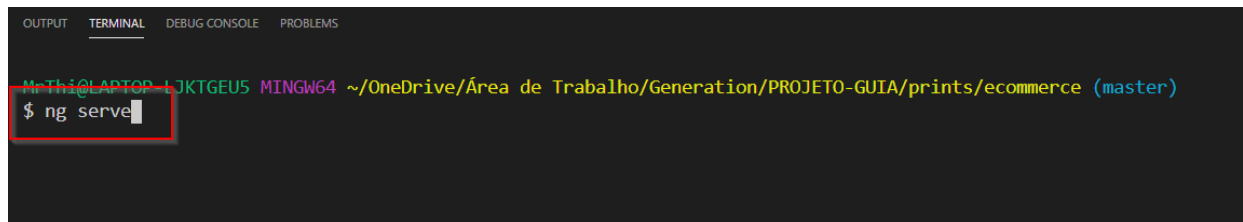


Certo, porém como nossa aplicação irá saber que ela deve inicial no componente home?

Isso é simples, basta adicionarmos a seguinte linha em nosso **app-routing.module.ts**:

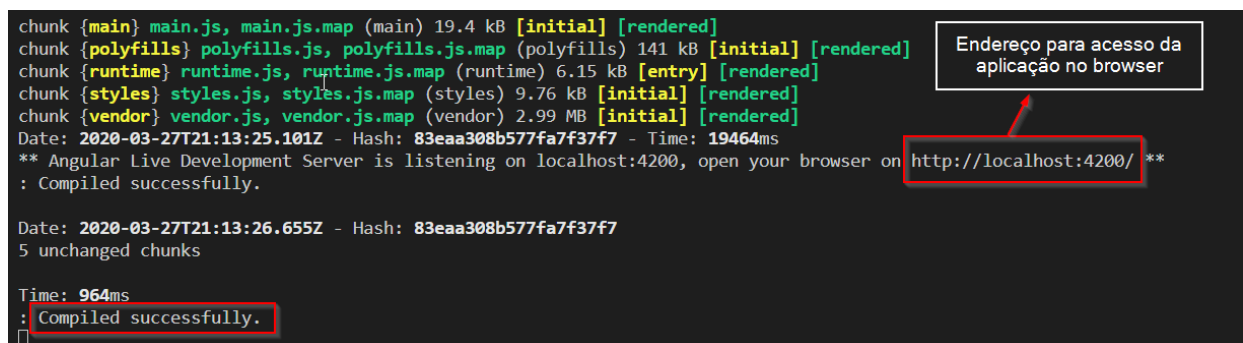


Pronto, agora estamos prontos para rodar a nossa aplicação e testar se nossa rota está funcionando, abra o CMD e digite o seguinte comando:



```
McTi@LAPTOP-LJKTGEU5 MINGW64 ~/OneDrive/Área de Trabalho/Generation/PROJETO-GUIA/prints/ecommerce (master)
$ ng serve
```

Como é a primeira vez que vocês está rodando esse comando, provavelmente irá demorar um pouco. Após a espera o CMD irá apresentar a seguinte tela:



```
chunk {main} main.js, main.js.map (main) 19.4 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 141 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 9.76 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 2.99 MB [initial] [rendered]
Date: 2020-03-27T21:13:25.101Z - Hash: 83eaa308b577fa7f37f7 - Time: 19464ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
: Compiled successfully.

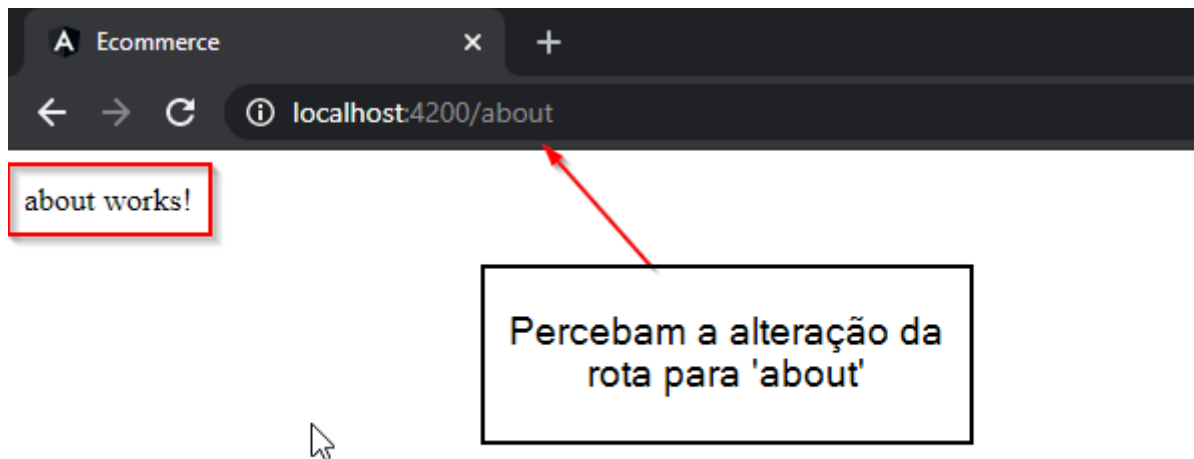
Date: 2020-03-27T21:13:26.655Z - Hash: 83eaa308b577fa7f37f7
5 unchanged chunks

Time: 964ms
: Compiled successfully.
```

Ao acessar nossa aplicação em <http://localhost:4200/> podemos encontrar o link que criamos, ao clicar nele entramos no componente about, conforme configuramos anteriormente:



Após o clique:



Perfeito! Agora conseguimos criar diversas rotas em nossa aplicação de forma estática!

O próximo passo é configurar o componente navbar para receber todos os componentes necessários para navegação em sua aplicação web. Tenha bastante criatividade!

Após essa configuração é importante colocar a tag da navbar no nosso app.component.html para que ela apareça para todos os nossos componentes, o mesmo deve ser feito para nosso footer.

Após isso, crie os componentes:

- contact: Página de contato;
- footer: Footer da aplicação.

Navbar (Menu)

Para facilitar nosso trabalho daqui para frente vamos criar uma navbar em nosso projeto, é nela que iremos colocar os links para navegação. Até aqui já criamos dois links de rotas, 'home' e 'sobre', depois você terá tempo para estilizar essas páginas.

Vamos para a documentação do bootstrap, lá precisamos buscar a seção de **navbar** ou em português [Barras de Navegação](#)

Escolha a primeira barra de navegação que aparecer, copie o código e insira no template do componente navbar que já criamos.

Após a criação da navbar, adicione no arquivo app.component.html a tag `<app-navbar>` `</app-navbar>` para que sua navbar apareça em todos os componentes:

```
src > app > <> app.component.html > router-outlet
1 | <app-navbar></app-navbar>
2 | <!--Diretiva para funcionamento das rotas de nossa aplicação-->
3 | <router-outlet></router-outlet>
```

Pronto, agora adicione dentro da lista já existente na navbar o routerLink específico para cada rota criada, dessa forma:

```
<li class="nav-item active">
  <a class="nav-link text-dark text-uppercase font-weight-bold" routerLink="/home">Home
  <span class="sr-only">(página atual)</span></a>
</li>
```

A partir daqui, todas as rotas que forem sendo inseridas em seu projeto terão que vir para a navbar para que você acesse os componentes de forma dinâmica e simples.

Integração Front/Back - Categoria

Para iniciar essa integração apenas com a tabela categoria nós precisamos criar os seguintes componentes:

- findAllCategorias;
- postEputCategorias;
- deleteCategorias;
- findByNameCategorias.

Consumo de Web service / HttpClient

O que é consumir um web service?

Consumir um Web Service e realizar as 4 operações do CRUD - Create (Criação - Post), Read (Ler - FindAll), Update (Atualizar - Put) e Delete (Deletar - Delete), no front-end se comunicando com o back-end (Web Service).

Para começar vamos consumir as informações da tabela **categoria** criada no banco de dados e no back-end.

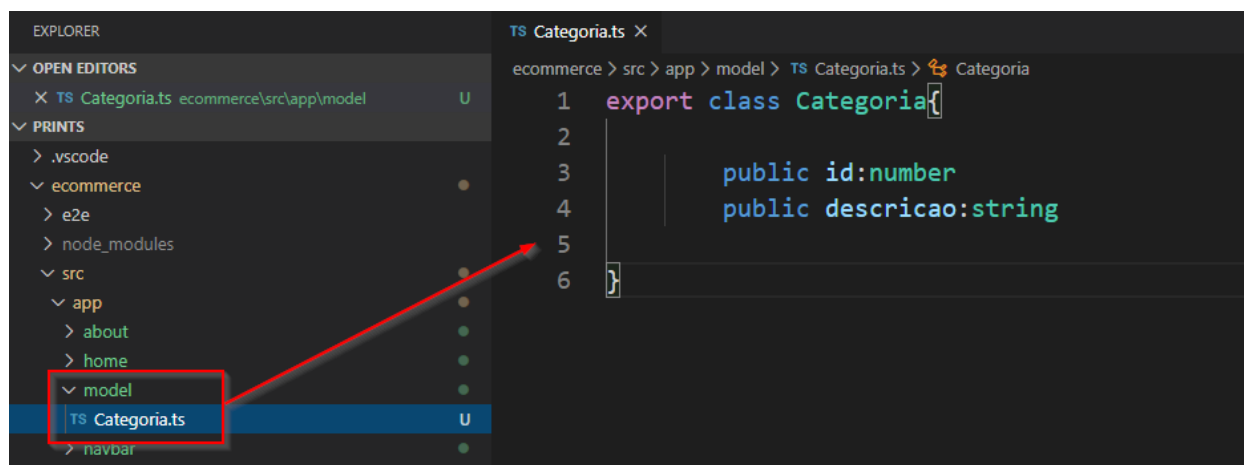
Preparação do ambiente

Para preparar o ambiente para receber os dados tempos que criar duas pastas importantes em nosso projeto, a Model e a Service, vamos falar um pouco mais sobre elas a seguir.

Criando a camada de Model

A pasta model é o local em que eu vou criar as mesmas classes de objetos criados no back-end, nesse caso inicial vamos criar a model de Categoria.

Para isso basta criar manualmente dentro da pasta **app** uma outra pasta chamada **model** e adicionar um arquivo chamado **Categoria.ts**, dentro desse documento nós criamos a mesma estrutura de atributos da entidade Categoria do back-end:



Criando a camada de Service

Agora precisamos criar os serviços que iram receber o métodos criados no back-end, esses serviços são chamados de end-points e são responsáveis pela conexão -> Front-end e Back-end.

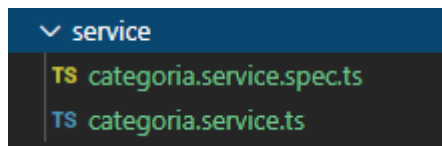
Para criar esses serviços vamos utilizar o CMD, para que o Angular-Cli crie a estrutura básica para nós, antes de digitar algum comando no CMD é preciso parar o servido Angular, para isso basta apertar **Ctrl + C** , agora digite no CDM o seguinte comando:

```
OUTPUT  TERMINAL  DEBUG CONSOLE  PROBLEMS
MITHI@LAPTOP-EJKTGE05 MINGW64 ~/OneDrive/Área de Trabalho/Generation/PROJETO-GUIA/prints/ecommerce (master)
$ ng g service service/categoria
```

Vamos entender esse comando:

- `ng g service`: Cria um **serviço**;
- `service/categoria`: Cria uma pasta **service** com um documento **categoria** dentro dela.

A nossa estrutura da pasta tem que ficar dessa forma:



Vamos abrir o arquivo **categoria.service.ts**, nele vamos encontrar a seguinte estrutura:

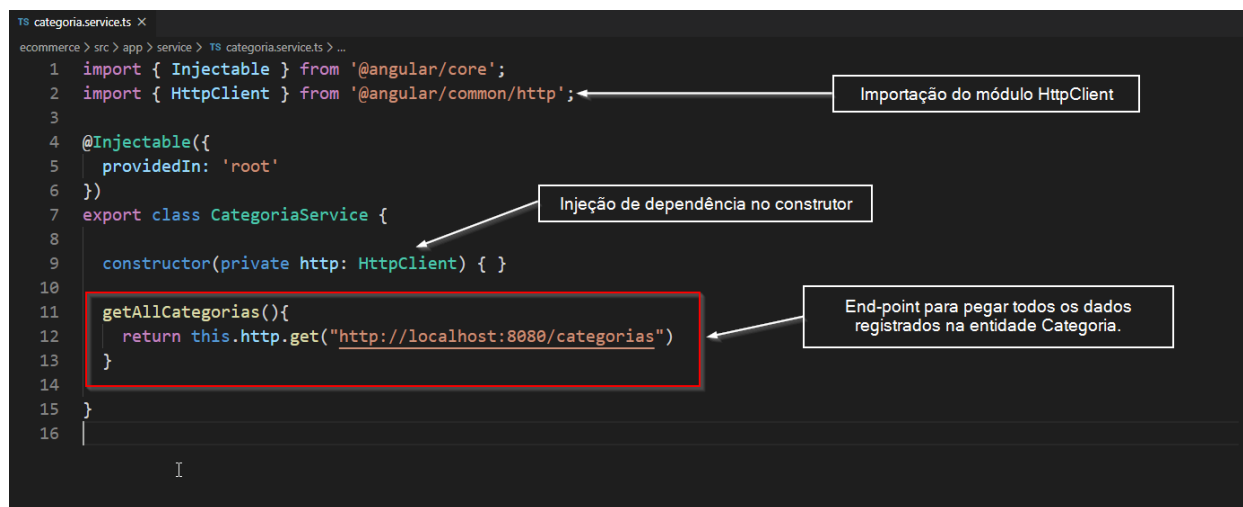
```
TS categoria.service.ts X
ecommerce > src > app > service > TS categoria.service.ts > ...
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root'
5  })
6  export class CategoriaService {
7
8    constructor() { }
9  }
10
```

FindAll Categorias

Dentro do serviço Categoria vamos criar nosso primeiro end-point, o `getAllCategorias()` que ira buscar todas as categorias cadastradas no banco de

dados e através do back-end trará ao front-end esses dados:

Porém para criar qualquer end-point é necessário injetar a dependencia do HttpClient em nosso construtor, essa dependencia é reponsável pela conexão com o servidor back-end:



Esse link passado dentro do return do método é dado pelos desenvolvedores back-end, na verdade nós recebemos todas as informações necessárias para consumir o web service criado pelo desenvolvedor(a) back-end.

Configurando o componente findAllCategorias

- TypeScript

Agora iremos configurar o componente findAllCategorias para receber as informações de nosso serviço, temos que lembrar que para que essas informações cheguem a nós precisamos ter dados inseridos na tabela consultada e nosso back-end precisa estar rodando.

Configure o arquivo `find-all-categoria.component.ts` da seguinte forma:



Vamos falar mais sobre o método findAllCategorias, linha por linha:

- `this.categoriasService.getAllCategorias().subscribe((resp: Categoria[])=>{` Essa linha é responsável por trazer o end-point criado no Service para que possamos consumir os dados em nosso back-end, o **subscribe** serve para dizer qual dados nós queremos, nesse caso é o objeto Categoria como Array, para que ele traga uma lista de objetos de categorias.
- `console.log(resp);` Serve para nos mostrar no console se estamos recebendo o dados esperado.
- `this.listaCategorias= resp` Agora nós estamos colocando em nossa variável **listaCategorias** o que foi consumido no Web Service, iremos usar essa variável no template para apresentar as categorias.
- `},err =>{ alert('Erro cod: ${err.message}'); })` Toda essas linhas servem para tratar um erro caso nossa consulta não der certo, aqui podemos colocar qualquer informação que seja a mais clara para entendermos que o consumo não funcionou. No meu caso eu preferi deixar a mensagem padrão do angular.
- HTML**

Precisamos definir no template como queremos apresentar essas informações, eu escolhi apresentar em um card, com estilos do bootstrap. Sendo assim nosso template fica da seguinte forma:

```

<div class="container">
  <div class="row">
    <div class="col-md-12 mb-5 mt-5 d-flex justify-content-center align-items-center">
      <div class="card mb-5 mr-4 ml-4" style="width: 18rem;" *ngFor="let categoria of
        listaCategorias">
        <div class="card-body">
          <h5 class="card-title">{{ categoria.descricao }}</h5>
          <p class="card-text">Código: {{ categoria.id }}</p>
          <a href="#" class="btn btn-primary" [routerLink]="['/postCategorias', categoria.id]
            ">Editar</a>
        </div>
      </div>
    </div>
  </div>
</div>

```

Depois falaremos mais sobre esse routerLink **editar**. No momento é importante entender como apresentamos os dados que foram consumidos em nosso WebService.

- **Rotas**

Agora vamos criar nossas rotas no arquivo **app-routing.module.ts**:

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { FindAllCategoriasComponent } from './find-all-categorias/find-all-categorias.component';

const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'categorias', component: FindAllCategoriasComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}

```

- **Testes**

- Para testar precisamos inserir direto no banco de dados, ou no PostMan, algumas categorias.
- Adicione sua nova rota a sua navbar.
- Após essa inserção abra o CMD e insira o comando para subir o seu servidor Angular: `ng serve`

- Agora basta entrar no browser e clicar em categorias, todas as suas categorias precisam ser apresentadas na tela.

Post/Put Categorias

V## Post/Put Categorias

Agora vamos criar categorias novas e edita-las diretamente pelo front-end. Vamos inserir essas duas funcionalidades no mesmo componente que chamamos de **postEputCategorias**, essa é a forma mais prática de realizar esses métodos.

Primeiro precisamos criar nossos end-points no nosso service de Categorias:

```
TS categoria.service.ts x
ecommerce > src > app > service > TS categoria.service.ts > CategoriaService > postCategoria
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { Categoria } from '../model/Categoria';
4
5 @Injectable({
6   providedIn: 'root'
7 })
8 export class CategoriaService {
9
10   constructor(private http: HttpClient) { }
11
12   getAllCategorias(){
13     return this.http.get("http://localhost:8080/categorias")
14   }
15
16   postCategoria(categoria: Categoria){
17     return this.http.post("http://localhost:8080/categorias", categoria)
18   }
19
20   putCategoria(categoria: Categoria){
21     return this.http.put("http://localhost:8080/categorias", categoria)
22   }
23
24 }
```

Os dois end-points são bem parecidos, o que muda é a palavra post e put, porém são diferentes do getAll, pois agora eles esperam receber um parâmetro, que nesse caso é um objeto Categoria. Isso porque iremos cadastrar e editar as Categorias, sendo assim precisamos criar novos objetos e editar os já existentes.

Observem que a categoria recebida é passada no retorno dessa solicitação de consumo, isso para que possamos “leva-las” ao back-end e consequentemente ao banco de dados gerando um novo registro e no caso do put editar um registro já existente.

Temos um ponto de atenção aqui, como saberemos qual objeto estamos editando? Poderíamos acessá-lo no banco de dados através do ID, para isso precisamos de um novo end-point, o getByIdCategoria:

```
TS categoria.service.ts x
ecommerce > src > app > service > TS categoria.service.ts > TS CategoriaService
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { Categoria } from '../model/Categoria';
4
5 @Injectable({
6   providedIn: 'root'
7 })
8 export class CategoriaService {
9
10   constructor(private http: HttpClient) { }
11
12   getAllCategorias(){
13     return this.http.get("http://localhost:8080/categorias")
14   }
15
16   postCategoria(categoria: Categoria){
17     return this.http.post("http://localhost:8080/categorias", categoria)
18   }
19
20   putCategoria(categoria: Categoria){
21     return this.http.put("http://localhost:8080/categorias", categoria)
22   }
23
24   getByIdCategoria(id: number){
25     return this.http.get(`http://localhost:8080/categorias/${id}`)
26   }
27
28 }
29
```

Esse também é um end-point que espera um parâmetro, porém agora nós iremos receber o id da categoria, e iremos passar esse id para nossa aplicação através da nossa própria URL!

- **TypeScript**

Agora vamos editar nosso arquivo **post-eput-categorias.component.ts**, ele ficou um pouco maior que o último criado, então vamos dividi-lo em partes:

- Primeiro vamos injetar as dependências necessárias para esse component:

```
constructor(private route: ActivatedRoute, private router:Router, private categoriasService: CategoriaService) { }
```

- Agora vamos criar nosso primeiro método para buscar o ID de nossa categoria, com isso conseguimos verificar se essa categoria já existe, daí iremos editar, ou se ela não existe daí então iremos criar uma nova:

```
findById(id:number){
  this.categoriasService.getByIdCategoria(this.id).subscribe((resp: Categoria)=>{
    this.categoria=resp
  }, err => {
    console.log(`Erro cod: ${err.status}`)
  });
}
```

- Esse método está esperando um ID que será buscado em nosso end-point getByIdCategoria, agora vamos verificar se ele recebeu ou não esse ID no `ngOnInit()` , mas antes temos que criar algumas variáveis de controle para facilitar nossa verificação:

```
id:number;
novo: boolean = false;
categoria: Categoria = new Categoria()
```

- Agora sim vamos criar nossa verificação:

```
ngOnInit() {
  this.id = this.route.snapshot.params["id"];
  if (this.id == undefined){
    this.novo = true;
  } else {
    this.findById(this.id);
    this.novo = false;
  }
}
```

Usamos a variável id para receber o id que foi enviado em nossa URL através desse comando.

Iremos construir nossa lógica de uma forma bem simples. Se não tiver nenhum id na URL, nossa variável booleana "novo" irá receber true, ou seja, queremos cadastrar uma nova categoria. Caso o contrário iremos passar esse id para o método findById e a variável "novo" irá receber false, ou seja, estamos querendo editar uma categoria já existente.

- Existem dois pontos importantes nesse **ngOnInit**, primeiro que nós retiramos o retorno *void* que vem por padrão, pois agora queremos retornar algo nesse método, o segundo ponto é que temos que lembrar que tudo que está dentro de **ngOnInit** será executado assim que nosso component for carregado.
- Agora estamos prontos para criar o nosso último método desse component, o método `save()` :

```
save(){
  if (this.novo) {
    this.categoriasService.postCategoria(this.categoria).subscribe((resp: Categoria)=>{
      this.categoria= resp;
      this.router.navigate(['/categorias']);
    })
  } else {
    this.categoriasService.putCategoria(this.categoria).subscribe((resp: Categoria)=>{
      this.categoria= resp;
      this.router.navigate(['/categorias']);
    })
  }
}
```

- A lógica desse método também é simples, nós iremos verificar se nossa variável booleana **novo** é true, se ela for nós iremos fazer um post em nosso WebService, caso o contrário iremos fazer um put.
- Observe que a lógica do post e do put é bem parecida com a do `getAll`, porém agora passamos parametros, pois estamos lidando com dados específicos.
- **HTML**

Vamos agora apresentar esse cadastro e essa edição em nosso template, precisamos lembrar que estamos com os dois métodos no mesmo component, isso nos indica que precisamos tratar esse componente de forma dinâmica de acordo com o que queremos fazer, veja como fica o template apenas com a opção de cadastrar:

- Primeiro eu quero que apareça dois títulos diferentes de acordo com as ações que serão realizadas, para isso vamos usar a diretiva **ngIf**:

```
<div *ngIf="id; else elseBlock">
  <h1 class="mt-5 text-center">Edição de categorias</h1>
</div>
<ng-template #elseBlock>
  <h1 class="mt-5 text-center">Cadastro de categoriais</h1>
</ng-template>
```

Essa diretiva vai verificar se a variável **id** criada no ts é true, se ela for true significa que já temos uma categoria e estamos querendo editá-la, então o título será “Editar categoria”. Para fazermos o else no ngIf, precisamos colocar após o ponto e vírgula “**else elseBlock**” e depois abrir a tag ng-template com a variável **#elseBlock** criada. Nela iremos colocar o título para cadastro de categoria.

- O próximo passo é criar um formulário que será utilizado para as duas ações:

```
<div class="form-group col-sm-12 mb-5 mt-5">
  <label for="descricao">Descrição</label>
  <input type="text" class="form-control" id="descricao" placeholder="Descrição" [(ngModel)]="categoria.descricao">
</div>

<div class="form-group col-sm-4">
  <button (click)="save()" class="btn btn-info">Salvar</button>
  <button [routerLink]="['/deleteCategoria', categoria.id]" class="btn btn-danger ml-3">Deletar</button>
</div>
```

Na entidade Categoria temos apenas um campo de preenchimento, a descrição. Sendo assim precisamos de apenas um input e dentro dele colocamos a propriedade **[(ngModel)]** que irá “escutar” o que for colocado nesse input e vai inserir no atributo **descricao** do objeto **categoria** que foi declarado no ts desse mesmo componente.

Logo em baixo colocamos dois botões, o **Salvar** que vai executar através do evento **(click)** o nosso método `save()`, lembrando que esse método realiza duas ações de acordo com resultado da condição adicionada nele.

O outro botão é o **Deletar**, vamos falar dele com mais detalhes na próxima seção desse documento.

- Rotas

Vamos criar as rotas para essas requisições:

```
const routes: Routes = [  
  { path: '', redirectTo: 'home', pathMatch: 'full' },  
  { path: 'home', component: HomeComponent },  
  { path: 'about', component: AboutComponent },  
  { path: 'categorias', component: FindAllCategoriasComponent },  
  { path: 'postEputCategorias', component: PostEputCategoriasComponent },  
  { path: 'postEputCategorias/:id', component: PostEputCategoriasComponent },  
];
```

Observem que agora precisamos declarar essas rotas de duas formas, uma apenas indo para o componente e a outra passando o ID na url, isso porque, para toda edição é necessário um ID para o `findById` conseguir enviar a alteração para o banco de dados. Então passamos esse ID via URL.

Porém, onde iremos acessar essa rota?

Vamos acessá-la de duas formas:

- A primeira é para cadastro, esse acesso pode ser direto na navbar utilizando a rota que está sem o ID.
- A segunda é para a edição, nesse caso precisamos criar um botão **editar** em nosso template do componente **findAllCategorias**, para que a nossa aplicação envie o ID da categoria em questão via URL, permitindo assim acessar a condição de ID no `ngOnInit()`, vamos construir esse botão da seguinte forma:

```
<div class="container">  
  <div class="row">  
    <div class="col-md-12 mb-5 mt-5 d-flex justify-content-center align-items-center">  
      <div class="card mb-5 mr-4 ml-4" style="width: 18rem;" *ngFor="let categoria of listaCategorias">  
        <div class="card-body">  
          <h5 class="card-title">{{ categoria.descricao }}</h5>  
          <p class="card-text">Código: {{ categoria.id }}</p>  
          <a href="#" class="btn btn-primary" [routerLink]="['/postEputCategorias', categoria.id]">Editar</a>  
        </div>  
      </div>  
    </div>  
  </div>  
</div>
```

- Testes

- Adicione uma nova categoria na rota de cadastro na navbar;

- No findAll clique no botão editar e teste a edição.

Delete Categorias

Para realizar a ultima operação do nosso CRUD precisamos criar o end-point dela em nosso service de categorias:

```
delete(id:number){  
  return this.http.delete(`http://localhost:8080/categorias/${id}`)  
}
```

Assim como o getByldCategoria, nós precisamos saber qual objeto estamos deletando, sendo assim iremos passar o ID da categoria via URL, como mostra a imagem acima.

Agora vamos construir o typescript do nosso componente, primeiro vamos criar duas variáveis em nossa classe:

```
delOk:boolean = false  
categoria: Categoria = new Categoria()
```

Agora precisamos injetar as dependências de nosso arquivo dentro do nosso construtor:

```
constructor(private route: ActivatedRoute,  
             private router:Router,  
             private categoriasService: CategoriaService){}
```

Como precisamos pegar o ID que foi enviado na URL, vamos declarar uma variável local **id** em nosso `ngOnInit()`, então vamos atribuir o comando que pega o ID na URL para essa nova variável e depois vamos pass-la como parâmetro para nosso conhecido método `findById()`:


```
ngOnInit() {

  let id:number = this.route.snapshot.params["id"];
  this.findById(id)

}
```

Criação do método findById:

```
findById(id:number){
  this.categoriasService.getByIdCategoria(id).subscribe((resp: Categoria)=>{
    this.categoria=resp
  }, err => {
    console.log(`Erro cod: ${err.status}`)
  });
}
```

Precisamos que o nosso usuário confirme se ele quer deletar sua categoria ou não, sendo assim precisamos criar dois métodos: `btnSim()` e `btnNao()` :

```
btnSim(){
  this.categoriasService.delete(this.categoria.id).subscribe(()=>{
    this.delOk = true
    this.router.navigate(['/categorias']);
  }, err => {
    console.log(err);
  });
}
```

Nós iremos passar como parâmetro o id da categoria que capturamos na URL e enviar para o nosso end-point que através da conexão com o back-end deleta nosso dado no banco de dados.

Depois enviamos nosso usuário para a rota "categorias".

Caso o usuário clique em Não, ele simplesmente será enviado de volta para a nossa rota categorias:

```
btnNao(){
  this.router.navigate(['/categorias']);
}
```

- **HTML**

Essa é uma sugestão para o nosso template:



- Rotas

Vamos criar nossa rota de deleção em nosso **app-routing.module.ts**:



Diferente da rota de post e put, a delete só tem uma “versão”, ele obrigatoriamente precisa mandar o ID via URL para o próximo componente, caso contrário é impossível deletar algo.

E onde iremos colocar a rota dele, sendo que ele não pode ficar em nossa navbar?

É simples, basta lembrar do botão de deletar dentro da edição da nossa categoria. Da uma olhada no que falamos aqui em cima:

Logo em baixo colocamos dois botões, o **Salvar** que vai executar através do evento **(click)** o nosso método `save()`, lembrando que esse método realiza duas ações de acordo com resultado da condição adicionada nele.

O outro botão é o **Deletar**, vamos falar dele com mais detalhes na próxima seção desse documento.

O botão é esse:

```
<div div class="form-group col-sm-4">
  <button (click)="save()" class="btn btn-info">Salvar</button>
  <button [routerLink]="['/deleteCategoria', categoria.id]" class="btn btn-danger ml-3">Deletar</button>
</div>
```

Ele faz a mesma coisa que o botão editar, porém agora ele chama o componente que acabamos de configurar, que por sua vez aciona o end-point de delete e gera através do back-end a deleção desse dado em nosso banco de dados.

- **Testes**

Agora basta você acessar o componente de edição e apertar em DELETAR, se a página voltar para o seu `findAll` e a categoria não estiver mais lá deu tudo certo!

Pronto, agora temos um CRUD completo construído em nossa aplicação, basta replicar o mesmo modelo para todas as outras entidades que queiramos inserir em nossa aplicação!

Bônus - FindByNameCategoria

Antes de qualquer coisa, vamos configurar o nosso end-point que será responsável pela conexão com o controller no back-end:

```
findByName(descricao:string){
  return this.http.get(`http://localhost:8080/categorias/nome/${descricao}`)
}
```

Iremos passar a **descrição** da nossa categoria como parametro e depois iremos enviá-la em nossa URL

Em nosso back-end nós construímos o controller de FindByNameCategoria, vamos consumi-lo aqui em nosso front-end. Para isso é necessário abrir o typescript do nosso

componente findByNameCategoria já criado anteriormente e realizar essa configuração:

```
import { Component, OnInit } from '@angular/core';
import { CategoriasService } from '../service/categorias.service';
import { Router } from '@angular/router';
import { Categoria } from '../model/Categoria';

@Component({
  selector: 'app-find-by-name-categoria',
  templateUrl: './find-by-name-categoria.component.html',
  styleUrls: ['./find-by-name-categoria.component.css']
})
export class FindByNameCategoriaComponent implements OnInit {

  listaCategorias: Categoria[];
  nome:string;

  constructor(private categoriasService: CategoriasService, private router: Router) { }

  ngOnInit(): void {
  }

  findByNameCategoria(){
    this.categoriasService.findByName(this.nome).subscribe((resp: Categoria[])=>{
      this.listaCategorias= resp;
    },err =>{
      console.log(err)
    })
  }
}
```

Vamos criar duas variáveis, listaCategorias que será um Array de Categorias e a nome do tipo string.

Não podemos esquecer de injetar as dependências necessárias para o nosso componente.

A lógica para esse componente é bem simples, nós iremos criar o método findByNameCategoria que irá acessar o nosso service, buscando o end-point que criamos, nele iremos passar o parametro nome que será capturado em nosso template, depois disso o back-end fará a mágica de procurar a categoria correspondente e nós a apresentaremos no template.

Vamos para o nosso template entender como será o comportamento do método criado:

```
<div class="container">
  <h1 class="d-flex justify-content-center mt-4 mb-5">Pesquisa por nome</h1>
  <div class="form-inline my-2 my-lg-0 d-flex justify-content-center">
    <input class="form-control mr-sm-2" type="search" [(ngModel)]="nome" placeholder="Pesquisar por nome"
      aria-label="Pesquisar" (change)="findByNameCategoria()">
    <button class="btn btn-outline-secondary my-2 my-sm-0" (click)="findByNameCategoria()">Pesquisar</button>
  </div>

  <div class="container">
    <div class="row">
      <div class="col-md-12 mb-5 mt-5 d-flex justify-content-center align-items-center">
        <div class="card mb-5 mr-4 ml-4" style="width: 18rem;" *ngFor="let categoria of listaCategorias">
          <div class="card-body">
            <h5 class="card-title">{{ categoria.descricao }}</h5>
            <p class="card-text">Código: {{ categoria.id }}</p>
            <a href="#" class="btn btn-primary" [routerLink]="['/postCategorias', categoria.id]">Editar</a>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

Vamos item por item:

1: Criação do nosso input de pesquisa, é nele que vamos inserir o dado a ser pesquisado, então adicionamos um ngModel para escutar o que será inserido e adicionar isso a variável 'nome' que criamos no ts.

Logo depois adicionamos um evento de (change) chamando o nosso método findByNameCategoria(), esse evento é responsável por verificar qualquer mudança do input para conseguir acionar o método de busca.

2: Inserimos em nosso botão de pesquisar o evento (click) para chamar o nosso método que irá buscar a categoria correspondente ao nome passado no input.

3: E por último, inserimos um card simples para trazer a categoria correspondente e suas informações através de interpolação.

- **Rotas**

Desafio - Entidade Produtos

Considerando tudo o que já fizemos com a entidade categoria, basta você aplicar os mesmos passos, porém agora com a entidade Produtos, lembre-se que essa entidade tem mais atributos, seja criativo!

Para te ajudar vou dar duas dicas:

- A primeira é aí vai uma dica, comece criando a model, o service e esses três componentes básicos:
 - `findAllProdutos;`
 - `postEputProdutos;`
 - `deleteProdutos;`
- A segunda dica é que você precisa se lembrar que a entidade Produto possui como atributo a categoria do tipo Categoria (como objeto). Sendo assim a criação do model de Produtos ficará dessa forma:

```
import { Categoria } from './Categoria';

export class Produto{

    public id:number
    public nome:string
    public urlImagem:string
    public qtdStoque:number
    public valor:number
    public categoria: Categoria

}
```

Lembre-se sempre de criar as rotas para os componentes criados, caso o componente precise do id, é só passá-lo pela url dessa forma: "nomeDaRota/:id"

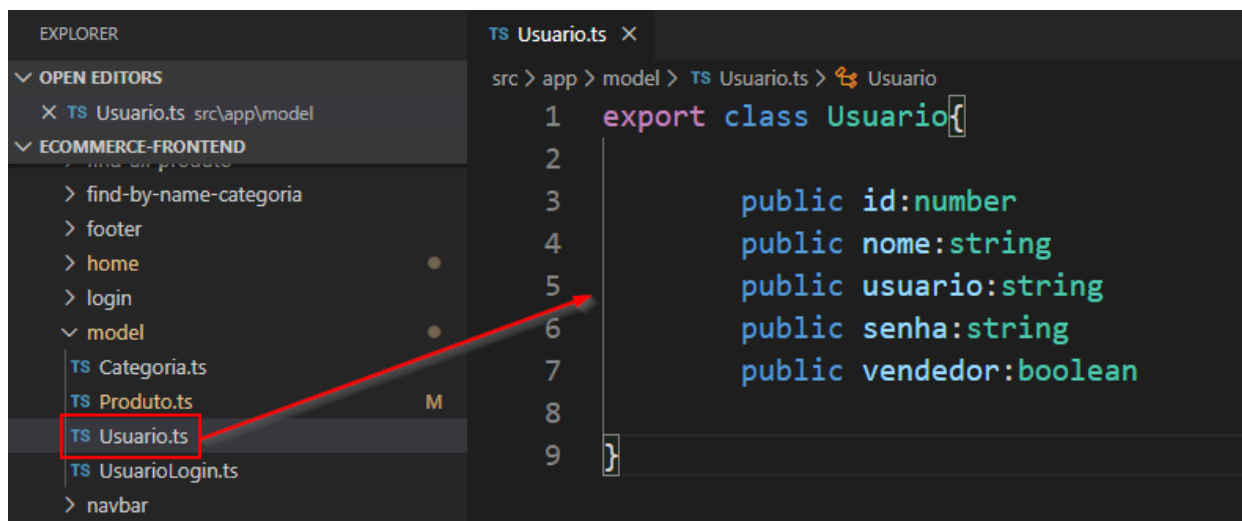
Autenticação

Para o nosso projeto ficar mais interessante vamos criar um login para a nossa aplicação, porém para trabalhar com login precisamos de autenticação. Vocês, com certeza, já viram autenticação nesse projeto no back-end, agora vamos ver como consumir tudo o que foi construído em spring!

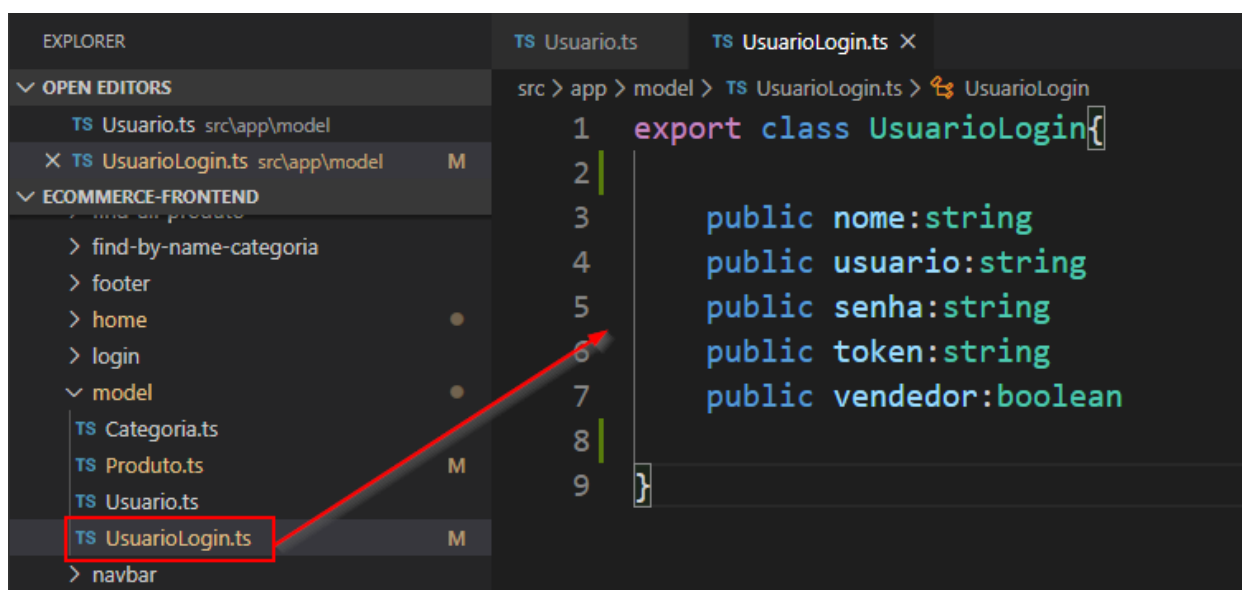
Como foi em categoria e produtos, nós iremos começar essa camada do projeto criando nossa model de autenticação.

Em nosso back-end nós temos dois tipos de entidades, o **Usuario** e o **UsuarioLogin**, sendo assim precisamos criar uma model para cada entidade.

- Model Usuario:



- Model UsuarioLogin:



A diferença entre os dois é que a Model Usuario será utilizada para o cadastro do nosso usuário novo, já a Model UsuárioLogin será utilizada para fazer o login de um usuário já existente passando o token de autenticação.

Criando a camada de Service de autenticação

Para nosso login funcionar precisamos de um service para nossa autenticação. Por questões de boas práticas de mercado vamos utilizar o nome de auth para o nosso serviço.

Vamos criar o serviço via CMD com o comando:

```
ng g service service/auth
```

Agora vamos configurar nosso service dessa forma:

```
1 import { Injectable } from '@angular/core';
2 import { HttpClient, HttpHeaders } from '@angular/common/http';
3 import { Usuario } from '../model/Usuario';
4 import { UsuarioLogin } from '../model/UsuarioLogin';
5
6
7 @Injectable({
8   providedIn: 'root'
9 })
10 export class AuthService {
11
12   constructor(private http: HttpClient) { }
13
14   token = {
15     headers: new HttpHeaders().set
16       ('Authorization', localStorage.getItem('token'))
17   }
18
19   cadastrar(user: Usuario){
20     return this.http.post("http://localhost:8080/usuarios/cadastrar", user)
21   }
22
23   login(userLogin: UsuarioLogin){
24     return this.http.post("http://localhost:8080/usuarios/login", userLogin)
25   }
26
27
28 }
```

Vamos criar o objeto token, que vai ter o atributo headers, esse atributo irá receber um módulo do Angular chamado HttpHeaders que irá inserir o nosso token no localStorage, só assim conseguiremos autenticar se o nosso usuário está autorizado a acessar nossa aplicação.

Vamos criar dois end-points, um para cadastro e outro para login, ambos são do método post, porém irão enviar dados para objetos diferentes, no back-end a API irá comparar as duas entidades e autorizar o login.

É importante focar que o nome user depois da virgula do end-point de cadastrar é o nome da variável do tipo Usuario que foi criada como parâmetro do método, e a mesma coisa acontece com a userLogin. Esses nomes podem ser diversos, basta seguir diretrizes de boas práticas de códigos para não gerar confusão.

Criando o componente de cadastro

Precisamos criar o componente de cadastro para enviar ao end-point os dados fornecido pelo usuário, a construção desse componente é bem simples. mas primeiro vamos gerar esse novo componente no CMD:

```
ng g c cadastro
```

Antes de mais nada precisamos criar nossa rota para esse componente

Após isso vamos configurar nosso arquivo typescript da seguinte forma:

- Vamos criar quatro variáveis e injetar nossas dependências necessárias para esse componente:


```
export class CadastroComponent implements OnInit {

  user = new Usuario()
  senha: string
  email: string
  vendedor: boolean

  constructor(private authService: AuthService, private router: Router) { }

  ngOnInit(): void {

  }

}
```

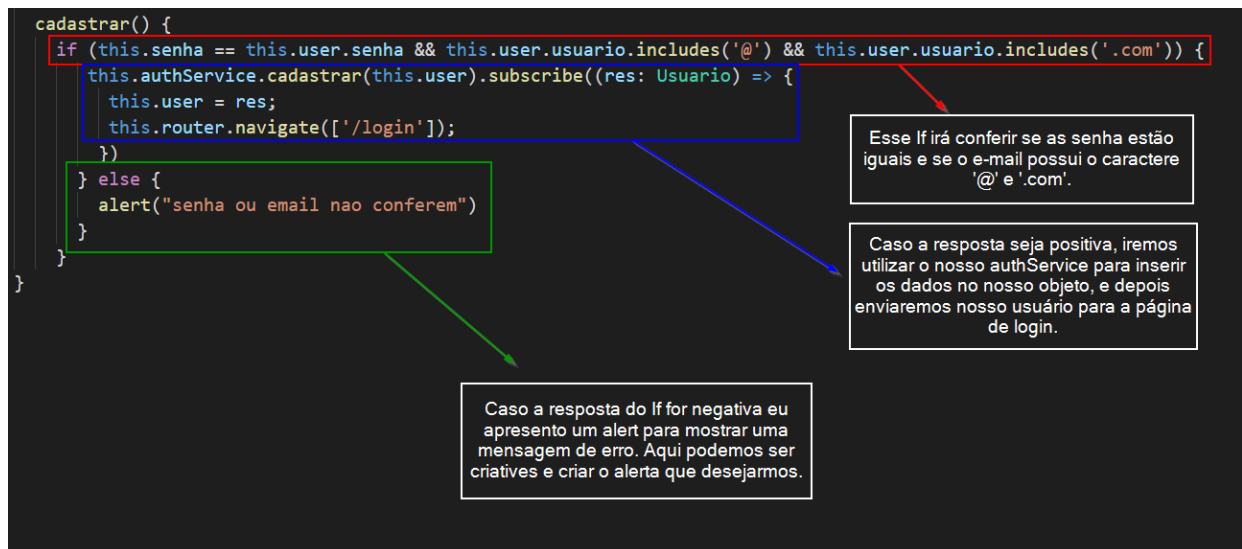
- Agora chegou a hora de criar nossos métodos, o primeiro é o de conferir senha, nele vamos utilizar um evento para capturar o valor inserido na senha, para depois compara-lo com o valor do campo conferir senha:

```
conferirSenha(event: any) {
  this.senha = event.target.value;
  console.log(this.senha)
}
```

- O próximo método servirá para verificar se o nosso usuário é um vendedor ou não, pois isso irá alterar o tipo de acesso que ele terá em nossa aplicação. Para verificar isso nós iremos adicionar no formulário do template um campo de checkbox, caso esse campo esteja selecionado ele será um vendedor, caso o contrário não será, o código fica dessa forma:

```
conferirVendedor(event: any) {
  let check: string
  check = event.target.value;
  if (check == "on") {
    this.user.vendedor = true
  } else {
    this.user.vendedor = false
  }
}
```

- O último método do nosso componente será o cadastrar, nele iremos verificar os campos digitados e logo após realizar o post no banco de dados via API, o código fica dessa forma:



- Rotas

Precisamos adicionar o nosso novo componente a uma rota em nossa aplicação:



Essa é uma rota simples, pois não precisamos passa nada pela url.

Agora basta você adicionar esse routerLink em sua navbar.

- Testes

Insira um novo usuário e vá ao banco de dados verificar se a tabela foi preenchida com os dados inseridos no front-end.

Criando o componente de login

Vamos criar o componente para autenticar a entrada do usuário em nossa aplicação:

```
ng g c login
```

Como já criamos toda a configuração de nosso service, agora vamos utilizar o end-point de `userLogin` para trabalhar com a autenticação do login.

- **TypeScript**

Essa é a configuração inicial do nosso arquivo ts:

```
export class LoginComponent implements OnInit {  
  
  userLogin = new UsuarioLogin ()  
  user = new Usuario()  
  
  constructor(private router: Router, private authService: AuthService, private  
    location: Location) { }  
  
  ngOnInit() {  
  }  
}
```

Instância dos objetos "UsuarioLogin" e "Usuario"

Injeção de dependências necessárias para esse componente.

- Vamos criar nosso primeiro e único método:

```
logar(){  
  this.authService.logar(this.userLogin).subscribe((resp: UsuarioLogin)=>{  
    this.userLogin = resp  
    localStorage.setItem("token", this.userLogin.token)  
    localStorage.setItem("vendedor", this.userLogin.vendedor.toString())  
    localStorage.setItem("nome", this.userLogin.nome)  
    this.router.navigate(['/produtos'])  
    location.assign('/produtos')  
  }, err =>{  
    alert('campos inválidos')  
  })  
}
```

Caso ocorra algum erro no preenchimento dos campos, essa mensagem aparecerá na tela, além de mostrar um alert padrão podemos personalizar essa mensagem conforme nossa necessidade ou vontade.

1 - Primeiro nós iremos acessar nosso service e utilizar o endpoint **logar()** que irá passar um **UsuarioLogin** como parâmetro, depois através do subscribe iremos inserir dentro de **userLogin** um novo usuário.

2 - O segundo passo é passar algumas informações para o localStorage do browser, isso serve para que possamos validar se o usuário está realmente logado, se ele é um vendedor e também para passarmos o nome dele, afim de apresentá-lo na tela inicial da aplicação. Atenção para o sublinhado vermelho, esse método **toString** está ali, pois o localStorage só aceita valores do tipo String, e o atributo **vendedor** é do tipo boolean,

sendo assim ele precisa ser modificado para ser enviado.

Essa técnica é muito utilizada para passar valores de variáveis entre componentes do angular.

3 - O último passo é levar o nosso usuário para a tela de produtos e na linha de baixo atualizar essa linha.

- Nosso **HTML** pode ficar dessa forma:

```
<div class="form-group">
  <label for="email">Endereço de email</label>
  <input type="email" class="form-control" id="email" aria-describedby="emailHelp" placeholder="Seu email" [(ngModel)]="userLogin.usuario">
</div>

<div class="form-group">
  <label for="senha">Senha</label>
  <input type="password" class="form-control" id="senha" placeholder="Senha" [(ngModel)]="userLogin.senha">
</div>

<div class="d-flex justify-content-center">
  <button class="btn btn-outline-secondary w-50 mt-4 text-uppercase font-weight-bold" (click)="logar()">Entrar</button>
</div>
```

De tudo que podemos adicionar em nosso template, os três principais itens estão nessas imagem:

1 - Input para o e-mail, nele inserimos um `[(ngModel)]` para escutar o valor inserido e adicionar na variável **userLogin.usuario**

2 - Input para a senha, nele inserimos um `[(ngModel)]` para escutar o valor inserido e adicionar na variável **userLogin.senha**

3 - Botão de entrar que chama através do evento de `(click)` o método **logar** que está em nosso TypeScript.

Podemos adicionar diversas configurações em nosso template para deixa-lo o mais bonito possível, porém essas são as primordiais.

- **Rotas**

Para que nosso componente seja acessado em nossa navbar basta adicionar essa linha ao nosso array de rotas que já estamos acostumados:

```
{ path: 'login', component: LoginComponent }
```

- **Testes**

Para testar utilize o usuário que você acabou de inserir no componente de cadastro e tente logar com ele.

Definido permissões de componentes

Para definir permissões de componentes nós iremos utilizar o token enviado via localStorage na hora do login, ao validarmos se esse token está no localStorage nós saberemos se o usuário está logado, com isso podemos liberar ou não componentes de nossa aplicação, outra validação que pode ser usada é se esse usuário que logou é um vendedor ou não, caso seja, terá permissões diferentes de um simples usuário. Como essas verificações precisam ser feitas no início do carregamento do componente, nós iremos coloca-la no método padrão dos componentes `ngOnInit()`, a seguir veja um exemplo dessas validações:

```
ngOnInit() {  
  
  if (localStorage.getItem('token') && localStorage.getItem('vendedor') == "false" ){  
    alert('Faça o login primeiro')  
    this.router.navigate(['/login'])  
  }else {  
    this.id = this.route.snapshot.params["id"];  
    if (this.id == undefined){  
      this.novo = true;  
    } else {  
      this.findById(this.id);  
      this.novo = false;  
    }  
  }  
}
```

Vamos entender o que essas condições estão dizendo, caso o usuário tente entrar nesse componente de cadastro de Categoria, seja pela navbar ou pela URL:

- A primeira valida se não existe um valor dentro do item token no localStorage e se o item vendedor é igual a false, caso verdade a aplicação mostra um alert e direciona o usuário para o componente de login.

- Caso o contrário a aplicação começa a validar os quesitos para cadastro ou edição de categoria como vimos anteriormente e tudo segue da forma normal.

É simples assim, você pode utilizar essa validação em qualquer parte da sua aplicação. seja para mudar uma navbar ou para impedir o acesso a um componente, use a sua lógica e principalmente sua criatividade.

