Olá, meu nome é Flávio Almeida, meu twitter é [@flaviohalmeida](http://www.twitter.com/flaviohalmeida), e muito provavelmente já nos encontramos em outros cursos da Alura. Este curso foi dividido em três partes numa tentativa de agrupar conceitos. Ao longo dessa trilogia construiremos uma aplicação completa no lado do front end, desde a criação dos primeiros componentes e boas práticas até a autenticação de usuários em uma Single Page Application.

Nesta parte 1, de **Fundamentos**, aprenderemos a organizar o projeto de maneira profissional, criar componentes, diretivas, lidar com a biblioteca **RxJS** para o benefício da nossa aplicação, entre outros.

É recomendado que se faça o primeiro exercício desta aula, por conta de toda a infraestrutura, e do que é necessário para darmos continuidade. Também é indicado o exercício sobre as motivações do Angular, em que há uma história de um projeto real, para melhor compreensão de quando o Angular passou a ser interessante para um projeto.

Bom estudo e sucesso nessa empreitada!

**Preparação do ambiente**

Desenvolveremos nossa aplicação através do **Angular CLI** (utilizaremos a versão 6.X), uma ferramenta endossada pela própria equipe do Angular. Entretanto, para que possamos utilizá-la, é necessária a instalação da plataforma **Node.js 8.X** em nossa infraestrutura de desenvolvimento.

A plataforma Node.js é apenas um requisito de desenvolvimento; não quer dizer que o produto final, nossa aplicação, deva rodar em um servidor Node.js. A aplicação final pode ser hospedada em qualquer servidor estático, independente da tecnologia utilizada.

O [Node.js](https://nodejs.org/en/) é um ambiente JavaScript multiplataforma disponível para Linux, Mac e Windows. Para instalá-lo, siga as instruções referentes à sua plataforma.

**Node 8.X obrigatório (atenção usuários Windows)**

É necessário que usemos **exatamente** a versão 8.X do Node.js, mesmo que haja versões mais recentes. Caso contrário, a API que será utilizada mais tarde no curso não funcionará, por conta de um módulo de banco de dados atrelado à versão 8.X.

Se o Node já estiver instalado na máquina, é preciso verificar se ele está pelo menos na versão 8.X com o comando node -v no terminal.

Usuários do Windows 7 e pouquíssimos usuários do Windows 10 estão tendo problemas com a execução da API. A solução segue no final da explicação.

**Instalação Node.js no Linux (Ubuntu)**

No Ubuntu, através do terminal (permissão de administrador necessária) executemos o comando abaixo:

sudo apt-get install -y nodejs

Em algumas distribuições Linux, pode haver um conflito de nomes quando o Node é instalado pelo apt-get. Neste caso específico, no lugar do binário ser node, ele passa a se chamar **nodejs**. Isso gera problemas pois, como a instrução **npm start** procura o binário node, e não nodejs, ela não funcionará. Para resolver, usemos a seguinte instrução no terminal para subir o servidor:

nodejs server

Ou no Ubuntu:

sudo ln -s /usr/bin/nodejs /usr/bin/node

Depois, o comando npm start funcionará conforme esperado.

**Instalação Node.js no Windows**

O instalador poderá ser baixado clicando-se no botão install, diretamente da página do Node.js:

* [Versão 64 bits (provavelmente seu OS é 64 bits)](https://nodejs.org/dist/v8.11.2/node-v8.11.2-x64.msi)
* [Versão 32 bits](https://nodejs.org/dist/v8.11.2/node-v8.11.2-x86.msi)

Durante a instalação, basta clicar nos botões para continuar o assistente. **Não** troque a pasta padrão do Node.js, a não ser que saiba exatamente o que está fazendo.

**Atenção**: na plataforma Windows, dependendo da configuração da sua máquina, ou do uso de uma rede corporativa (proxy), pode ser que a instrução npm install não funcione. Caso isso aconteça, pode-se tentar os passos a seguir. Caso o primeiro funcione, não é necessário fazer o segundo:

a) Problema no certificado: no terminal, deve-se executar o comando npm set strict-ssl false. Em seguida, é possível repetir a operação de instalação que falhou.

b) Redes de empresa ou proxy da rede: alguns usuários acessam a rede através de um proxy. Isso pode impedir que o npm tenha acesso ao seu repositório. Nesse sentido, é necessário saber o endereço do proxy para então configurá-lo através das instruções a seguir (troque proxy.company:8080 pelo endereço do seu proxy).

npm config set proxy http://proxy.company.com:8080

npm config set https-proxy http://proxy.company.com:8080

**Usuários de Windows 7 e Windows 10**

Se mesmo com os passos acima o comando npm install falhou, como última fronteira para tentar ajudar, recomendo que se baixe [essa versão da API](https://s3.amazonaws.com/caelum-online-public/865-angular/api-windows.zip). Ela possui o sqlite pré-compilado, sendo assim, **não é necessário executar** o comando npm install, e sim apenas o comando npm start, pois a pasta "node\_modules" já esta incluída no projeto. Mas atenção, esse recurso é indicado apenas para quem não conseguiu resolver o problema na máquina.

**Instalação Node.js no MAC**

O [homebrew](http://brew.sh/) é a maneira mais recomendada para instalar o Node.js, por meio do comando:

brew update

brew install node

Não usa homebrew? Sem problema, basta baixar o instalador clicando no botão install, diretamente da página do [Node.js](https://nodejs.org/en/).

**Editor: Visual Studio Code (gratuito)**

Recomendo extremamente o uso do Visual Studio Code (VSCode), um editor gratuito criado pela Microsoft e disponível para Windows, Linux e MAC. Sua integração com TypeScript é realizada sem qualquer esforço, e sem configurações adicionais. Além disso, haverá a paridade visual comigo, neste curso.

Você pode baixá-lo [aqui](https://code.visualstudio.com/download).

Angular foca em *ever green browsers*, isto é, suportará sempre as duas últimas versões vigentes dos navegadores do mercado. Por exemplo, se a última versão do Chrome é 65, ele suportará esta versão e a anterior, apenas.

Isso não significa que uma versão mais nova de Angular deixará de funcionar na antepenúltima versão do navegador. A equipe do Angular simplesmente removerá os testes de integração para navegadores não suportados.

Todavia, se a aplicação está fortemente vinculada a uma versão específica de um navegador, talvez essa característica do *framework* não se coadune com a estratégia

A versão 1.X do Angular é chamada de **Angularjs**, e todas as outras a partir da versão 2.X são chamadas de **Angular** apenas. A mudança do Angularjs para o Angular foi tão drástica que o primeiro não é pré-requisito para que se aprenda o segundo. O importante é entender que Angular é um framework totalmente reescrito, utilizando tecnologias que não estão mais presentes no Angularjs.

Após todos os requisitos em relação à infraestrutura, instalação do **Node.js** e do editor **Visual Studio Code**, podemos dar início à criação do nosso projeto. Há um detalhe, porém: 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, e empacotarmos e gerarmos o arquivo para a produção. Gastaremos um tempo considerável antes de começarmos a programas efetivamente.

Então, vamos acessar o site do [Angular CLI](https://cli.angular.io/) (*Angular Command Line Interface*), que é cliente de linha de comando do Angular, atualmente patrocinado pela sua própria equipe. 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 um único comando. Isso poupa bastante tempo, e é este o fluxo que vamos seguir no curso. Aliás, é o fluxo recomendado pela própria equipe do Angular.

Para instalarmos o Angular CLI, precisamos do **npm**, um gerenciador de pacotes do Node.js — sendo assim, o npm só estará disponível caso o Node esteja instalado na máquina. É uma ferramenta que baixa módulos, bibliotecas e *frameworks* da internet, para que possamos usá-los.

Vamos abrir o Prompt de Comando, sem nos preocuparmos com qual pasta estamos. Verificaremos se o npm está instalado com npm --version. Será retornado 6.1.0, que é a sua versão. Para solicitar a instalação do Angular CLI globalmente, ou seja, para que esta ferramenta esteja disponível em qualquer diretório da aplicação, utilizaremos o comando npm install -g @angular/cli@6.0.7.

Acrescentamos `@6.0.7` para indicar que usaremos esta versão específica, já que na data de gravação deste curso, esta é a versão mais recente. Deste modo garantimos paridade entre o que faremos aqui e o que poderá ser feito posteriormente. Se deixarmos o comando sem este final, a versão mais recente será instalada automaticamente, e não é isto que queremos.

Outro detalhe: para que o -g funcione, é necessário privilégio de administrador no Prompt de Comando em uso, ou no Terminal, no caso do Linux. Este comando é executado apenas uma vez por máquina de desenvolvimento, e ele irá acessar a internet e baixar o Angular CLI. O processo é rápido e, para confirmarmos sua instalação, usaremos o ng, que é o módulo do Angular CLI, colocado globalmente para uso no terminal: ng --version. Serão exibidas as versões instaladas de todos os módulos, inclusive do TypeScript do RxJS.

Para começar o nosso trabalho, de que forma criaremos um projeto?

Com o comando cd Desktop, acessaremos a área de trabalho, e solicitaremos ao ng o alurapic, que é o projeto a ser desenvolvido, um sistema de gerenciamento de fotos. O comando será ng new alurapic, e desta vez o processo costuma demorar um pouco, pois quando se cria um projeto com Angular CLI, ele vem com pré-configurações de todos os módulos que a nossa aplicação necessitará, isto é, vários downloads são feitos por meio do npm.

Após a finalização do processo, aparecerá uma mensagem que pode parecer preocupante, mas não é: "found 13 vulnerabilities <9 low, 4 high>", isto é, foram encontradas 13 vulnerabilidades, 9 baixas e 4 altas neste projeto. A partir das versões mais novas, o npm começou a ter um sistema de análise da consistência e da segurança dos módulos utilizados pela aplicação.

Isso quer dizer que os módulos baixados pelo Angular CLI possuem algumas vulnerabilidades. No entanto, o responsável pela correção delas não é a equipe do Angular CLI, e sim de quem criou estes módulos, que são numerosos, e sobre os quais não temos controle. O mais importante é que isso não nos afeta em nada, pois o que baixamos são dependências de desenvolvimento, os quais não entrarão em produção.

Se quiséssemos um servidor em Node, por exemplo, e algum módulo tivesse vulnerabilidades, pode ser que fosse algo problemático mas, no nosso caso, não há problema algum.

Criado o nosso projeto, se acessarmos a área de trabalho, encontraremos a pasta "alurapic". Então, poderemos utilizar os comandos cd alurapic, e em seguida ng serve --open. Isso subirá um servidor local configurado pelo Angular CLI usando-se boas práticas de desenvolvimento, e automaticamente abrirá o navegador em uma página padrão. Este projeto será simples porque iremos personalizá-lo utilizando a estrutura criada para o desenvolvimento do nosso projeto.

Vamos entender a estrutura que foi gerada? Inicialmente, abriremos a pasta "alurapic" no Visual Studio Code, em "File > Open Folder...", ou pelo atalho "Ctrl + K Ctrl + O". Selecionaremos a pasta, cujo conteúdo será exibido em um painel lateral esquerdo no programa.

O Visual Studio se integra muito bem à linguagem utilizada pelo Angular, que é o **TypeScript**. Mas onde é que se localiza a página gerada anteriormente, com "Welcome to app!", em nosso projeto? Neste painel lateral, não há nenhum index.html, e o importante, neste momento, é que todo código que diz respeito à aplicação está dentro de "src" (de *source*), inclusive index.html.

Pode ser tentador abrirmos diretamente a pasta "src" no programa, porém isto não é recomendado. Deve-se sempre abrir a "alurapic" como pasta raiz do Visual Studio Code, pois ele precisa encontrar alguns arquivos de configuração dentro dela. Caso contrário, teremos muitos problemas.

Vamos, então, abrir index.html, que é a página padrão criada pelo servidor local configurado pelo Angular CLI. Entretanto, a página do navegador (localhost:4200) possui uma imagem, lista de links, entre outros elementos que não aparecem no código de index.html. Além disso, o conteúdo dentro das tags <body> são tags que nem existem no mundo HTML, <app-root>.

Em Angular, **tudo é um componente**, capaz de guardar um comportamento, o CSS, e a marcação HTML, a estrutura, em um único local. Assim, a página localhost:4200 não é diferente no Angular, já que uma página também é um componente, e possui HTML, CSS e, caso haja, JavaScript, tudo vinculado em um único objeto denominado componente. Assim, <app-root> indica a existência de componentes.

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 abrirmos 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.

De que forma o Angular consegue acessar tudo isso que vimos e transformá-los em algo que o navegador consegue carregar e entender?

Internamente, o Angular utiliza um *web pack*, um *module bundle*, isto é, um empacotador de módulos famoso, utilizado por *Create React App*, *Vue CLI*, e outros frameworks *Single Page Application*. O interessante é que em nenhum momento estas configurações do *web pack* são exibidas, pois elas ficam encapsuladas pelo Angular CLI.

Com isso, fizemos o primeiro tour em relação à aplicação. Antes de começarmos a criar nossos próprios componentes, veremos mais um detalhe, a seguir.

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 lemos 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.

Para quem já utilizou ferramentas *browser sync* (sincronização de navegador) para automatizar tarefas de front end, o Angular CLI servirá muito bem. Neste momento inicial, é importante entendermos que o uso de duas chaves abertas e duas fechadas representam uma *Angular Expression*. Toda vez que o programa encontrar algo nesta sintaxe, será buscado seu conteúdo, sua **expressão**, nas propriedades do componente. Isto é *Data binding*, que funciona como se o template sugasse os dados do componente.

Vamos mudar a aplicação, e começar a adequá-la para que ela realmente se torne uma aplicação de gerenciamento de imagens?

Vamos adequar a nossa aplicação para que ela fique mais próxima de um sistema que gerencia fotos. Antes disto, vamos recapitular como subimos nosso projeto:

1. Abrimos o Prompt de Comando (ou Terminal, no Linux);
2. acessamos nossa área de trabalho (cd Desktop) e depois a pasta "alurapic" (cd alurapic);
3. iniciamos a aplicação com ng serve --open, que a abrirá no browser automaticamente.

Deste modo, a aplicação passa a rodar em um servidor local disponibilizado pelo Angular CLI. Então, poderemos abrir o Visual Studio Code, que estará com a pasta do projeto aberta. A ideia é que o primeiro componente a ser carregado pela nossa aplicação seja o <app-root>, que equivale ao app.component.ts, e sua apresentação está em templateUrl, que é ./app.component.html, pasta em que se encontra o arquivo .ts.

Vamos abrir app.component.html, deletar todo seu conteúdo, e salvar. Ao voltarmos ao navegador, teremos uma página em branco, já que não há nada no template do componente. Adicionaremos uma tag <img> cujo *source* pode ser uma imagem qualquer:

<img src="https://upload.wikimedia.org/wikipedia/commons/thumb/5/5a/Sultan\_the\_Barbary\_Lion.jpg/440px-Sultan\_the\_Barbary\_Lion.jpg" alt="Leão">

Com isso, a página do navegador passa a exibir uma imagem de leão, como gostaríamos. Por ora, não há nada de sofisticado no Angular, e este curso ficará, claro, cada vez mais complexo conforme avançamos.

Se queremos fazer um *Data binding*, e que o componente no qual este template está associado forneça os dados para o template, criaremos em app.component.ts uma propriedade com descrição Leão, e outra, url, cujo valor será o link da imagem que acabamos de incluir:

export class AppComponent {

title = 'alurapic';

description = 'Leão';

url = 'https://upload.wikimedia.org/wikipedia/commons/thumb/5/5a/Sultan\_the\_Barbary\_Lion.jpg/440px-Sultan\_the\_Barbary\_Lion.jpg';

}

Feito isso, precisaremos remover os valores de app.component.html para de fato realizarmos o *Data binding*:

<img src="" alt="">

Ao voltarmos para o navegador, nada estará sendo exibido, justamente porque ainda não passamos o src nem o alt. A primeira reação é querermos usar uma *Angular Expression*, porém, não faremos isso. Este comportamento é utilizado quando temos o conteúdo de uma tag e queremos exibir o valor de uma propriedade do componente dentro da tag de conteúdo.

Sendo assim, salvaremos o código abaixo e voltaremos ao navegador, em que teremos apenas o texto "alurapic", o title da página:

<h1>{{ title }}</h1>

<img src="" alt="">

Mas quando queremos realizar o *Data binding* de um atributo, isto ocorrerá de outra maneira — teremos que envolvê-lo em colchetes. E então, setaremos seus valores como o nome do componente correspondente:

<h1>{{ title }}</h1>

<img [src]="url" [alt]="title">

Assim, ao salvarmos e voltarmos ao navegador, teremos o título e a imagem sendo exibidos na página. O que acabamos de fazer é denominado ***One way data binding*** ("data binding unidirecional"), isto porque os dados saem do componente, da fonte de dados, e vão para o template, no entanto não fazem o caminho contrário, e é importante frisarmos isto.

Reparem que, ao usarmos os colchetes, url e title se tornam expressões que o Angular sabe que precisa buscar no componente ao qual o template está associado. Se deixarmos sem os colchetes, uma vez que não fazemos *Data binding*, isto é, não avaliamos uma expressão, o valor de src será a *string* url, enquanto o de alt será o texto title, e não o valor title.

Assim, quando formos realizar *Data binding* com atributos, usaremos os colchetes. E quando formos exibir um valor em uma tag, ou isoladamente, usaremos uma *Angular Expression*. Agora, somos capazes de evidenciar algumas convenções do Angular, assunto que veremos mais adiante.

Vamos observar a nomenclatura dos arquivos criados pelo Angular CLI, focando em app.component.ts. O nome do arquivo .ts no qual iremos programar o nosso componente é todo em *lower case*, ou seja, em caixa baixa, e seguido por .component. Trata-se de uma convenção que não podemos deixar de seguir.

Se quiséssemos criar um menubar.component.ts e, no nome do menu utilizarmos *Pascal Case*, ficaria MenubarComponent, mesma definição que ocorre em Java, C# e afins. Outro exemplo é que caso o nome do componente seja calopsita, usaremos calopsita.component.ts, e o nome da classe será CalopsitaComponent.

O template segue a mesma regra; o template do componente possui uma url, que precisa ser o mesmo nome do arquivo .ts, porém sua extensão é .html, claro, para deixar a identificação muito clara e bem definida. O mesmo ocorre com o arquivo .css.

Esta convenção está disponível no site com a sua documentação. Na época do Angular 1, existia o padrão criado por John Papa, que abdicou de definir um padrão para a versão seguinte do Angular, e contribuiu, juntamente com outros desenvolvedores, no padrão adotado pelo [site do Angular](https://angular.io/).

Existem outras convenções que veremos conforme forem aparecendo em nosso projeto. A seguir, criaremos nosso primeiro componente.

Tivemos um overview da estrutura do projeto, do Angular, e do Data binding, e podemos criar nosso primeiro componente. Antes disso, precisamos resolver uma questão: como iremos estilizar estes componentes?

Em um primeiro momento, utilizaremos o **Bootstrap**, então precisaremos carregá-lo em nossa Single Page Application usando o Angular. Vamos voltar ao Visual Code, abrir o index.html, e incluir a tag <link> do Bootstrap, certo?

Não! No Angular, quando precisamos importar um CSS global como o Bootstrap, [Normalize](https://necolas.github.io/normalize.css/), CSS Reset ou outro similar, isso não funciona desta forma. 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.

E se incluirmos a tag diretamente no código, o Angular CLI não saberá que isto é uma dependência da aplicação, e que ele precisa entrar neste processo de construção. Isso é um tanto estranho para quem está familiarizado com uma aplicação tradicional: o Angular CLI pode acessar o CSS e transformá-lo em um único código JavaScript, ou então separar o arquivo .css. Então, ele precisa saber onde se encontra o arquivo CSS, e isto não se dá por meio da tag <link>.

### Em que lugar carregamos CSSs ou scripts globais?

No caso, como estamos falando sobre CSS, há um arquivo em "ALURAPIC" chamado angular.json, com uma série de configurações que dizem respeito ao build do projeto, tanto que no meio dela há chaves referentes a styles e scripts. Estas propriedades servem inicialmente para carregarmos em styles todos os CSSs globais da aplicação, ou seja, os que serão aplicados em todos os componentes, bem como os scripts globais, os quais não pertencem a nenhum componente específico.

No caso, já existe src/styles.css em styles, o que significa que o programa busca dentro da pasta "src" o arquivo styles.css. No entanto, agora precisamos carregar o Bootstrap, e para tal precisamos baixá-lo, e indicar o caminho de onde se localiza este arquivo. Mas não precisaremos acessar o site e baixá-lo dali. Já que estamos utilizando o gerenciador de pacotes do Node para o Angular CLI funcionar, poderemos utilizá-lo para baixar todas as dependências de front end de que a aplicação necessita.

Se quisermos usar [jQuery](https://jquery.com/), [Bootstrap](https://getbootstrap.com/), [Foundation](https://foundation.zurb.com/apps/docs/#!/), e afins, conseguimos baixá-los pelo terminal.

Vamos, então, pausar o Angular CLI, usar o comando npm install bootstrap@4.1.1 dentro da pasta do projeto, isto é, iremos utilizar a versão 4.1.1 do Bootstrap. Não se preocupe caso não o conheça, pois aprenderemos seus aspectos fundamentais ao longo do curso.

Feito isso, o repositório será acessado, o Bootstrap será baixado, e a dependência do nosso projeto será incluída em package.json. Todo projeto em Node possui este arquivo, que lista todas as dependências e módulos que foram baixados pelo npm.

No passado era necessário acrescentarmos --save ou -S ao fim do comando que acabamos de utilizar, para que o arquivo fosse devidamente listado em package.json. Entretanto, já que estamos utilizando a versão Node superior à 8, no caso, 10, isto não é necessário, pois a dependência é adicionada à listagem automaticamente.

Sabemos que o Bootstrap se encontra em "node\_modules > bootstrap > dist > css", mas como o carregaremos em nossa aplicação?

Ele precisa fazer parte de uma lista de CSSs globais que o Angular CLI levará em consideração em tempo de construção da aplicação em si, seja ela para rodar em desenvolvimento ou em produção. Esta indicação é feita em angular.json, que possui uma chave denominada build, dentro do qual está styles. Alteraremos e salvaremos seu conteúdo para incluir o caminho completo do Bootstrap na pasta "node\_modules":

"styles": [

"src/styles.css",

"./node\_modules/bootstrap/dist/css/bootstrap.min.css"

],

"scripts": []

Podemos notar que em "src" já existia um arquivo styles.css, que se encontra vazio. O importante é entendermos que, ao alterarmos angular.json, somos obrigados a fechar e reabrir o Angular CLI para que a modificação seja efetiva. Se o Bootstrap for carregado corretamente, no navegador, a fonte de "alurapic" será alterada.

No terminal, tendo o Angular CLI pausado, usaremos o comando ng serve --open para que se abra uma nova aba que carrega o Bootstrap, e veremos a modificação no navegador. Com isso, partiremos para a melhoria na responsividade, já que a imagem não diminui conforme diminuímos a página.

Vamos voltar ao Visual Code, abrir o template app.component.html, que exibe uma tag <img> apontando para uma url e um title, que vêm do próprio app.component.ts, e então acrescentaremos a classe do Bootstrap, img-thumbnail:

<h1>{{ title }}</h1>

<img class="img-thumbnail" [src]="url" [alt]="title">

Assim, ao retornarmos ao navegador e manipularmos as dimensões da página, veremos que a imagem se adequará. Isso é outra prova de que o Bootstrap está funcionando. Contudo, podemos fazer muito mais por esta simples estilização, e é isso que veremos a seguir.

Imaginemos a seguinte situação em que um novo estagiário integra seu time de desenvolvimento, ou alguém que não conheça Bootstrap. Caso seja solicitada uma imagem responsiva, a pessoa terá que estudar e se inteirar dos detalhes para isso. O que aconteceria se ela componentizasse uma imagem no Angular, com <ap-photo>?

A ideia é tentarmos criar o primeiro componente que ocultará a complexidade de se criar uma imagem responsiva do Bootstrap. Vamos fazer isso?

A convenção que seguiremos é, em "app", criarmos uma nova pasta chamada "photo", nome que diz respeito ao componente que queremos criar. Nela, criaremos o arquivo photo.component.ts, do TypeScript, que terá um arquivo de mesmo nome, porém com extensão .html. Sabemos que um componente em TypeScript também é uma classe em Angular.

Em photo.component.ts, usaremos @Component, e o Visual Code exibirá, dentre outras, a opção de *autoimport*, que selecionaremos, pressionando "Enter". Isso fará com que o programa importe automaticamente o *decorator* Component do módulo @angular/core, pois toda classe e *decorator* que for importado deverá ser informado — como em um *import* em Java, C#, ECMAScript, e por aí vai.

import { Component } from "@angular/core";

@Component

export class PhotoComponent {}

Entretanto, o programa indica erro e, ao passarmos o mouse sobre Component, uma vez que o Visual Code é integrado ao TypeScript, ele nos avisa que precisamos passar uma configuração especial para o Component. Portanto, passaremos chaves para indicar a existência de um objeto JavaScript, cuja propriedade é selector.

Também informaremos que o nome do componente é ap-photo, e incluiremos um template, não diretamente no HTML, e sim em um arquivo separado. Portanto, utilizaremos a opção templateUrl.

import { Component } from "@angular/core";

@Component({

selector: 'ap-photo',

templateUrl: 'photo.component.html'

})

export class PhotoComponent {}

Prefixar todos os componentes é uma boa prática. Supondo que alguém crie um componente chamado "photo", haverá um conflito de seletores. Neste caso, utilizamos "ap" para remeter a "alurapic".

Quanto ao dado do componente, a ideia é que ele tenha a URL e a descrição. Poderemos inclusive deletar title = 'alurapic' de app.component.ts. O navegador denunciará muitos erros, mas não nos preocuparemos com isto agora. Deletaremos todo o conteúdo de app.component.html e voltaremos a photo.component.ts para incluir a URL e descrição que copiamos de app.component.ts.

export class PhotoComponent {

description = 'Leão';

url = 'https://upload.wikimedia.org/wikipedia/commons/thumb/5/5a/Sultan\_the\_Barbary\_Lion.jpg/440px-Sultan\_the\_Barbary\_Lion.jpg';

}

Assim, quando o template for renderizado, será carregada uma imagem cuja class recebe img-thumbnail, e realizaremos um *Data binding* da propriedade src com sua url, e alt com description. Em photo.component.html, teremos:

<img class="img-thumbnail" [src]="url" [alt]="description">

Com isso, a URL do componente será lida, bem como a propriedade description do mesmo. Feito isso, no template de app.component.html, que se encontra vazio, já que queremos usar o nosso componente em um template de outro, é necessário utilizarmos seu *selector*, sua forma declarativa:

<ap-photo></ap-photo>

Não esqueçamos de salvar todas as alterações feitas.

Antes de visualizarmos o resultado no navegador, pela lógica, ao carregarmos a aplicação, a ap-photo deverá ser carregada, e nosso dado deve ser exibido. Porém, nada disso acontece. E se abrirmos o console com "Ctrl + Shift", é exibido erro do Angular.

É importante entendermos tudo isso para que, quando criarmos um componente, ele seja bem executado. A razão deste erro será visto a seguir.

O Angular nos dá uma pista sobre o erro que houve anteriormente: "'ap-photo' is not a known element", ou seja, "'ap-photo' é um elemento desconhecido". O primeiro ponto que o programa pede para verificar é se ele faz parte de um módulo, caso seja um componente do Angular. **Isto porque um componente obrigatoriamente precisa fazer parte de um módulo**.

Em se tratando de componentes, para o Angular um módulo é uma maneira de agruparmos componentes afins. Vamos supor que iremos criar um *Data table*, formada por vários componentes — cabeçalho da tabela, rodapé, coluna, e por aí vai. Isso tudo se relaciona a um único *Data table*, sendo assim, poderemos ter **um** módulo na aplicação, no qual agregaremos todos estes componentes.

Se houver outra parte da aplicação que queira usar o *Data table*, basta ela importar este módulo. Isso fará com que todos os outros componentes necessários para o *Data table* também sejam importados. Se não tivermos um módulo, caso ele queira utilizar o *Data table*, terá que importar todos eles individualmente.

O módulo, então, é uma forma de **organizarmos** a nossa aplicação. Por ora, ela só possui um módulo, app.module.ts, o primeiro a ser carregado, lembram? Ao abrirmos o arquivo, perceberemos que ele importa AppComponent. No template correspondente, queremos utilizar o componente PhotoComponent.

Sendo assim, é necessário declararmos o componente neste módulo:

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

import { PhotoComponent } from './photo/photo.component';

@NgModule({

declarations: [

AppComponent,

PhotoComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Portanto, primeiro vêm os *imports* de tudo que foi criado pelo Angular, e por último os nossos *imports*. Trata-se de uma convenção proveniente do site do Angular. Assim, estando o PhotoComponent em declarations, o componente será acessível para AppComponent, isto é, se eles fazem parte do mesmo grupo, um é "enxergado" pelo outro.

Salvaremos o arquivo, e a imagem será exibida no navegador, como esperado. Inclusive, poderemos voltar a app.component.html e incluir outra imagem:

<ap-photo></ap-photo>

<ap-photo></ap-photo>

Se voltarmos ao navegador, teremos duas imagens, uma ao lado da outra, ambas responsivas. No entanto, isso não é muito útil, uma vez que queremos utilizar títulos e informações diferentes para cada imagem. Caso contrário, como poderemos reutilizar este componente?

Vamos voltar ao problema de termos duas imagens (componentes PhotoCompoment), sendo que queremos que cada uma tenha informações independentes e específicas. Em app.component.html, então, queremos passar título e descrição diferentes para cada um dos <ap-photo>. Atualmente, este dado está em photo.component.ts, e queremos algo assim:

<ap-photo url="https://upload.wikimedia.org/wikipedia/commons/thumb/5/5a/Sultan\_the\_Barbary\_Lion.jpg/440px-Sultan\_the\_Barbary\_Lion.jpg" description="Leão"></ap-photo>

<ap-photo url="https://upload.wikimedia.org/wikipedia/commons/thumb/2/2d/Lioness\_Etosha\_NP.jpg/500px-Lioness\_Etosha\_NP.jpg" description="Leoa"></ap-photo>

O segundo componente está com uma imagem qualquer, e o primeiro está com a mesma imagem de antes — assim, podemos deletar tanto o valor de description quanto de url em photo.component.ts, sem esquecermos de salvar os arquivos. Pela lógica, PhotoComponent desconhece os valores de description e url, então, se acessarmos app.component.html, indicaremos quais são estes valores.

Entretanto, se voltarmos ao navegador, nada é exibido. Isso ocorre porque em nenhum momento incluímos estes valores nas propriedades de photo.component.ts ao colocarmos url e description de componentes distintos. Assim, quando o PhotoComponent for realizar o *Data binding*, ele lerá valores de url e description que estão em branco.

O que acontece é que precisamos **explicitar** que tais propriedades podem receber um parâmetro externo, passando um valor por meio da forma declarativa do componente. Caso isto não ocorra, estes valores nunca cairão em suas respectivas propriedades. Para isso, incluiremos @Input(), que faz parte do angular/core, e o programa apontará um erro porque precisamos importá-lo. Basta clicar nele e no ícone de lupa, e então em "Add 'Input' to existing import declaration from "@angular/core"". Teremos:

import { Component, Input } from "@angular/core";

@Component({

selector: 'ap-photo',

templateUrl: 'photo.component.html'

})

export class PhotoComponent {

@Input() description='';

@Input() url='';

}

Assim, indicamos que description e url são ***Inbound properties***, ou seja, aceitam receber um valor por meio de sua forma declarativa. Ao salvarmos e retornarmos ao navegador, teremos a exibição das duas imagens desejadas. Antes de continuarmos, uma dúvida: em app.component.html, por que não colocamos url e description entre colchetes? Já não estamos acostumados a fazer *Data binding*?

Se fizéssemos assim, os links seriam buscados como se fossem propriedades, uma vez que, ao realizarmos *Data binding*, avalia-se a expressão como propriedade do componente, e não é isso que queremos. Queremos passar a *string* diretamente. Porém, se estes valores estão caindo em photo.component.ts, precisamos usar a expressão no momento de suas leituras no template, como em photo.component.html.

Se removermos o *Data binding* deste arquivo, o valor de src será a *string* url. Portanto, é importante identificarmos e entendermos bem o momento de passar e interpretar os parâmetros para os componentes. Isto é, quando usamos um componente A em um componente B, e queremos passar valores para as suas propriedades, apenas aquelas decoradas com @Input() se tornarão *inbound properties*, e poderão recebê-los.

Com isso, finalizamos nosso primeiro componente, o PhotoComponent, mas há muito mais para ser visto em termos de organização do projeto, o que faremos em breve.

Vamos começar com o uso de uma boa prática que inclusive é preconizada e indicada no site do Angular. Atualmente, temos um único componente, o qual faz parte do módulo AppModule, também denominado ***root module*** ("módulo raiz") no Angular, por ser o primeiro a ser carregado pela aplicação.

Ao longo do nosso projeto, criaremos melhorias e outros componentes que dizem respeito ao universo de imagens. E já que um componente não funciona se não pertencer a um módulo, teremos que inclui-los em declarations de app.module.ts e, eventualmente, pelo seu tamanho excessivo, acabaremos não conseguindo distinguir estes grupos com tanta clareza.

Sendo assim, existe a possibilidade de criarmos um ***feature module***, um módulo que contém vários componentes que fazem sentido de serem agrupados. Então, em "app", criaremos a pasta "photos", e tudo que se relaciona com as imagens ficarão contidos nela. Inclusive, incluiremos dentro dela a pasta "photo", e automaticamente — já que o Angular utiliza o TypeScript, e seu editor verifica a integridade do código o tempo todo —, o programa indicará que há um erro.

Caso façamos qualquer tipo de alteração, o projeto continuará com problemas no navegador. Neste caso, está sendo indicado que não é possível importar PhotoComponent em AppModule. Assim, deletaremos a linha import { PhotoComponent } from './photo/photo.component', bem como PhotoComponent de app.module.ts.

Feito isso, já que ele não pertence a nenhum módulo, criaremos um photos.module.ts em "photos", que será o *feature module*. Nele, criaremos um módulo que irá declarar não só PhotoComponent, mas todos os outros componentes que dizem respeito a imagens, e app.module.ts importará este módulo nele mesmo.

Isso tornará a aplicação muito mais organizada. Em photos.module.ts, digitaremos "@NgModule" e pressionaremos "Enter" para fazermos a importação no angular/core, e entre parênteses passaremos um objeto JavaScript. E então declararemos PhotoComponent:

import { NgModule } from "@angular/core";

@NgModule({

declarations: [ PhotoComponent ]

})

export class PhotosModule {}

Se salvarmos o arquivo da maneira em que está e acessarmos o navegador, teremos uma página em branco, e no Console veremos que o componente não foi encontrado, pois ele precisa pertencer a um módulo. Assim, em app.module.ts, importaremos um módulo, e com isso teremos acesso aos componentes que foram declarados, automaticamente. Em declarations, só entram componentes, portanto colocaremos PhotosModule em imports:

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

PhotosModule

],

providers: [],

bootstrap: [AppComponent]

})

Assim, quando o primeiro módulo da aplicação for carregado (AppModule), o programa entenderá que dependemos de PhotosModule, que será carregado, e por consequência o PhotoComponent terá que ser disponibilizado para uso.

Aproveitando, o que é que o Angular está importando em BrowserModule? Este módulo traz vários recursos que utilizaremos ao longo do curso no navegador. Por isto, por padrão, o Angular CLI já faz sua importação.

Esta alteração que acabamos de fazer deve ser o suficiente para a nossa aplicação funcionar! Salvaremos, voltaremos ao navegador e... Continuamos com o mesmo erro. É indicado no Console que ap-photo não é um componente do Angular, e que precisamos verificar se ele faz parte de algum módulo.

Se criamos photos.module.ts e adicionamos PhotoComponent no declarations, como é possível que ele não faça parte de um módulo? Além disso, não acabamos de importá-lo em app.module.ts?

Esquecemos de um passo importante: em declarations de photos.module.ts, se encontra tudo aquilo que o módulo possui. Caso tenhamos dez componentes, eles se enxergam entre si dentro do módulo. Mas para que ele seja enxergado no módulo de quem importou PhotosModule, precisaremos especificar na propriedade exports do NgModule, sendo necessário também torná-lo acessível para quem for importá-lo.

@NgModule({

declarations: [ PhotoComponent ],

exports: [ PhotoComponent ]

})

Salvaremos e voltaremos ao navegador e, desta vez, tudo continua funcionando conforme esperado.

Entendemos que um módulo pode declarar muitos componentes, mas não exportar ou dar acesso a todos eles. Comparando com linguagens como Java ou C#, é como se tudo que se encontra em declarations estivesse privado, e em exports, público.

Temos dois componentes em exibição na página do navegador, cada qual usando seus próprios dados, como gostaríamos. Mas e se tivéssemos cem, duzentas imagens, o que teríamos que fazer? Seria muito trabalhoso. Além disso, sabemos que, quando trabalhamos com uma grande quantidade de dados, no universo da programação, usualmente lidamos com **listas**.

Vamos abrir app.component.ts e substituir as propriedades de AppComponent, que não estão sendo utilizadas, pelo seguinte trecho:

photos = [

{

url: 'https://upload.wikimedia.org/wikipedia/commons/thumb/5/5a/Sultan\_the\_Barbary\_Lion.jpg/440px-Sultan\_the\_Barbary\_Lion.jpg',

description: 'Leão'

},

{

url: 'https://upload.wikimedia.org/wikipedia/commons/thumb/2/2d/Lioness\_Etosha\_NP.jpg/500px-Lioness\_Etosha\_NP.jpg',

description: 'Leoa'

}

];

Com isso, criamos a propriedade photos, em que cada item do *array* é um objeto, com aquelas mesmas informações vistas anteriormente, de url e description das imagens do leão e da leoa. A partir destes dados listados, queremos criar um <ap-photo> para cada dado proveniente do *array*.

Não queremos utilizar o forEach(), passando pelos itens e criando um <ap-photo> individualmente, fazendo loops. A boa notícia é que o Angular possui um recurso chamado **diretiva**, capaz de alterar o comportamento de um componente já existente. A diretiva ngFor receberá photos, o mesmo que aparece em app.component.ts. Entretanto, a cada passada em que se queira referenciar a este photos, usaremos o nome photo.

Isso significa que iteramos photos e, a cada passada, o valor da imagem correspondente ficará na variável photo. Não podemos nos esquecer de declarar a variável usando let. E, se photo é uma variável, em url colocamos photo.url e, em description usamos photo.description.

<ap-photo

\*ngFor="let photo of photos"

url="photo.url"

description="photo.description"

>

</ap-photo>

No entanto, desta forma não fazemos *Data binding*, e isto é um problema, pois o Angular entenderá que estamos passando a *string* photo.url, e não o valor dessa expressão. Então, o código ficará assim:

<ap-photo

\*ngFor="let photo of photos"

[url]="photo.url"

[description]="photo.description"

>

</ap-photo>

Salvaremos o arquivo, voltaremos ao navegador, e teremos as duas imagens sendo exibidas corretamente. Se adicionarmos mais uma imagem no *array*, em app.component.ts, uma terceira imagem aparecerá na página. O Angular possui esta diretiva que se encarrega de iterar uma lista e repetir o componente para nós. Isso é muito interessante, já que, em vez de termos uma lista de imagens fixa, temos uma lista cujas imagens vêm do back end.

Assim, sabemos que a partir do momento em que colocarmos um novo valor em photos, o *Data binding* do Angular disparará a renderização da nossa tela, e então teremos acesso a todas as imagens que vêm do back end, com o qual faremos a integração em breve.

Chegou a hora de realizarmos a integração da nossa aplicação Angular com uma API, um back end. Neste momento, nossa aplicação está rodando em um servidor disponibilizado pelo Angular CLI. Ainda não entraremos na questão de gerar o projeto para que seja colocado em produção, mas é importante entendermos que o projeto que criamos será transformado em alguns arquivos que eventualmente poderão ser inclusos em um servidor PHP, Apache, Java, ou qualquer outro.

No caso, o Angular CLI faz este papel para nós, e cria um servidor de desenvolvimento com todas as vantagens que vimos até aqui. Continuando, uma aplicação em Angular é **estática**, ou seja, roda no navegador, sendo assim alguém precisa fornecer os dados para ela.

Assim, do outro lado da aplicação haverá outro servidor responsável por fornecer os dados para a aplicação Angular. Em uma *Single Page Application*, este servidor normalmente é chamado de **Web API**, e funciona fornecendo-se dados, após o qual a aplicação Angular conhecerá o endereço de onde obtê-los, os acessará, trazendo e renderizando-os para a exibição dos mesmos na tela.

Então, em um fluxo natural de uma *Single Page Application*, a aplicação em Angular fica em um servidor A, e a API com os dados e regras de negócio ficam em um servidor B, totalmente independentes. A vantagem disso é que a equipe que trabalha na API, no caso do back end, pode trabalhar como preferir sem interferir no que a equipe de front end estiver fazendo, e vice versa.

Claro, deve haver uma comunicação entre as aplicações front e back/Web API. E é isso que veremos nesta aula, além das maneiras como o Angular consegue acessar outro servidor e consumir seus dados para exibi-los ao usuário final. No entanto, não entraremos no mérito de criação deste servidor, dessa Web API, porque este é um curso de Angular, e não de design e criação de Web APIs.

Em um dos próximos exercícios há um link com os endereços da API que iremos utilizar no curso, disponível para download, dentre os quais está api.zip, que precisará ser descompactada e então recortada.

**Cuidado:** caso apareça uma pasta "api" dentro de outra com mesmo nome, a que deverá ser recortada é a que contém arquivos como data.db, package, package-lock, data e server.

Colaremos a pasta na área de trabalho, onde também se encontra "alurapic". Pensando em produção, "alurapic" se relaciona ao servidor que hospeda a aplicação Angular, enquanto "api" é o servidor que hospeda a Web API.

Temos um terminal rodando "alurapic", e abriremos outro Prompt de Comando para rodar nossa API, pois **ambos precisam estar rodando simultaneamente**. Digitaremos os comandos cd Desktop, cd api e npm install, sendo que este último só deverá ser usado uma única vez, para que a API baixe todas as dependências necessárias para o funcionamento da aplicação.

Em seguida usaremos npm start, comando que executará a Web API. São exibidas algumas informações de acesso, usuário, e-mail, senhas, dados que por ora não nos serão relevantes, já que dizem respeito à parte de autenticação. O importante é que, ao abrirmos o navegador e digitarmos a URL "localhost:3000/flavio/photos", teremos acesso a dados de diversas imagens cadastradas para uso em nossa aplicação.

Neste curso, faremos então a integração da aplicação Angular com a Web API. O *array* photos virá com os dados retornados pela Web API. Lembrem-se de que os dois terminais precisam estar rodando; caso o estudo seja pausado em um dia para continuidade no próximo, é preciso abrir o terminal na pasta do Angular CLI e usar o comando ng serve --open, e ir à pasta da Web API e usar npm start.

Baixe o servidor [aqui](https://s3.amazonaws.com/caelum-online-public/865-angular/api.zip). Depois, descompacte o arquivo em sua área de trabalho e logo em seguida apague o arquivo .zip. No terminal, vá até a pasta "api", e certifique-se que dentro da pasta há o arquivo package.json. Ainda dentro da pasta, execute o comando:

npm install

Aguarde o processo até o final, e quando terminar, execute o seguinte comando dentro da pasta:

npm start

Você verá a seguinte mensagem exibida no console após o servidor ter sido iniciado:

Servidor escutando na porta: 3000

Users

{ user\_id: 1,

user\_name: 'flavio',

user\_email: 'flavio@alurapic.com.br',

user\_password: 123,

user\_full\_name: 'Flávio',

user\_join\_date: '2018-04-12 20:24:19' }

Users

{ user\_id: 2,

user\_name: 'almeida',

user\_email: 'almeida@alurapic.com.br',

user\_password: 123,

user\_full\_name: 'Almeida',

user\_join\_date: '2018-04-12 20:24:19' }

No seu navegador, verifique o resultado da API acessando o endereço:

http://localhost:3000/flavio/photos

Um JSON será exibo na janela do navegador.

Com o Angular CLI rodando de um lado, e do outro nossa Web API, poderemos consumir os dados retornados por ela e exibir, no lugar de três imagens fixas, aquelas baseadas nos dados trazidos. Como faremos esta integração?

Sabemos que o primeiro componente a ser carregado pela aplicação é AppComponent (em app.component.ts), em que temos os dados fixos das imagens, os quais substituiremos por um *array* vazio:

export class AppComponent {

photos = [];

}

Ao salvarmos o arquivo e voltarmos ao navegador, nada aparecerá na página, já que se formos a app.component.html, o ngFor pegará um *array* vazio e não iterará em dado algum. Para termos acesso à comunicação com a Web API, podemos optar pela requisição Ajax, jQuery, entre outros. No entanto, no Angular existe um serviço específico para este tipo de tarefa, integrando-se ao *framework*, e através do qual toda comunicação com o back end será feita: o HttpClient.

Criaremos uma propriedade denominada http no AppComponent, e ela guardará uma instância. Ao digitarmos new HttpClient, o Visual Studio nos dará a opção de realizarmos *auto import*. Essa é uma vantagem que o programa nos fornece, pois sem ela teríamos que fazer o *import* primeiro e, se não temos o endereço decorado mentalmente, não teríamos como fazê-lo.

Entretanto, ao pressionarmos "Enter", o programa inclui um HttpClient que não é aquele que queremos utilizar, e sim um do pacote selenium-webdriver/http, que serve para testes. Então, deletaremos a linha gerada automaticamente, clicaremos em HttpClient e no ícone de lâmpada para solicitarmos outro tipo de *import*. A única opção que o Visual Studio oferece é justamente "Import 'HttpClient' from module "selenium-webdriver/http"".

Então, incluiremos manualmente a seguinte linha no começo do arquivo app.component.ts, e o salvaremos:

import { HttpClient } from '@angular/common/http';

Isso faz com que o programa nos informe que, para criarmos uma instância de HttpClient precisamos passar um HttpHandler no construtor. Vamos tentar:

export class AppComponent {

photos = [];

http = new HttpClient(new HttpHandler());

}

Porém o programa continua acusando um erro e, ao passarmos o mouse sobre HttpHandler, teremos a mensagem de que ela é uma classe abstrata — por lidarmos com TypeScript, neste caso não poderemos usar o operador new —, e isso parece ficar cada vez mais complicado. Queremos poder trabalhar com o HttpClient e só temos dificuldades.

Não nos interessam os detalhes de criação de um HttpClient. É como se quiséssemos desparafusar uma placa da parede, e quiséssemos uma chave de fenda, sendo irrelevante sabermos como ela foi criada, quem foi o inventor, e demais detalhes.

Neste sentido, queremos que nossa aplicação faça isso por nós. No caso, queremos uma **injeção de dependências**. No Angular, precisamos incluir o constructor, o qual toda classe em ECMAScript e TypeScript possui, e em que usaremos o parâmetro de quem dependemos, neste caso, http.

O Angular não sabe identificá-lo, podendo este ser uma *string* ou um número, por exemplo, então é necessário tiparmos, **explicitar seu tipo**. Em linguagens como Java, temos algo como HttpClient http = new HttpClient(), ou seja, o tipo vem antes da variável http. No TypeScript acontece o contrário, isto é, primeiro vem a variável, dois pontos, e o tipo. E então usamos console.log() para saber se a injeção realmente está acontecendo.

photos = [];

constructor(http: HttpClient) {

console.log(http);

}

Deste modo, o AppComponent será criado pelo Angular, e então passará pelo constructor(), e solicitará o HttpClient. E o Angular, de alguma forma, pegará o HttpClient e o injetará no construtor para que possamos utilizá-lo.

Ao abrirmos o navegador, nada será exibido, pois nenhum dado foi coletado. Porém, teremos muitos erros no console. É indicado "No provider for HttpClient!", ou seja, que não há provedor para ele. O que acontece é que o AppComponent não obtém resposta em relação à solicitação do HttpClient, por sua complexidade. Ou seja, caímos no mesmo problema: não conseguimos criar um HttpClient, e o Angular também não.

Para resolvermos isto, precisamos de um *provider* para que o Angular o injete. Não o temos ainda, mas a boa notícia é que existe um módulo do próprio Angular que, ao ser importado em nossa aplicação, automaticamente terá um *provider* configurado para uso.

O Angular estava querendo fazer o *import* de selenium-webdriver, e desconhece o HttpClient que queremos, porque não importamos o módulo HttpClientModule no nosso projeto. Sendo assim, o Visual Studio com TypeScript é esperto o suficiente para entender que os artefatos e classes deste módulo estão indisponíveis. Lidaremos, então, com dois problemas:

* O Visual Studio precisa saber fazer o *import* do HttpClient correto;
* a ausência do *provider*.

Iremos ao módulo principal da aplicação, app.module.ts, e incluiremos HttpClientModule em imports:

imports: [

BrowserModule,

PhotosModule,

HttpClientModule

],

Teremos que importá-lo manualmente, digitando import { HttpClientModule } from '@angular/common/http'; no começo do código. Salvaremos e, desta vez, ao retornarmos para app.component.ts, clicarmos em HttpClient e no ícone com lâmpada, serão exibidas várias opções. Selecionaremos "Import 'HttpClient' from module "@angular/common/http"", incluiremos console.log(http) novamente, voltaremos ao navegador, e deixamos de ter quaisquer problemas.

Com isso, o HttpClient foi injetado, e poderá ser trabalhado para fazer as requisições Ajax para o back end.

Agora que temos acesso ao HttpClient, vamos realizar a integração. Voltaremos a app.component.ts e deletaremos console.log(http). Cada usuário terá um retorno de imagens, então, trabalharemos com os únicos cadastrados, flavio e almeida, este último não possuindo imagem nenhuma.

Nosso objetivo atual é acessar o endereço http://localhost:3000/flavio/photos, o qual nos retornará os dados da imagem no formato JSON (JavaScript Object Notation) para serem colocados na propriedade photos. Será retornado um observable, e precisaremos da notificação do mesmo informando que os dados provenientes do servidor chegaram.

export class AppComponent {

photos = [];

constructor(http:HttpClient) {

const observable = http.get('http://localhost:3000/flavio/photos');

observable.subscribe();

}

}

Costumamos dizer que um observable é lazy (preguiçoso), pois só vai buscar os dados se tiver alguém inscrito nele (observable.subscribe()). Mas para não termos que ficar declarando esta variável, podemos encadear uma chamada diretamente, solicitando o get() e então um subscribe().

constructor(http:HttpClient) {

http

.get('http://localhost:3000/flavio/photos')

.subscribe();

}

O Angular é um framework que utiliza TypeScript, linguagem criada pela Microsoft, que usa o RxJS — e o observable vem dele —, tecnologia muito poderosa para lidar com operações assíncronas. Por ser mais rico, no Angular 2 preferiram utilizar o Observable em vez de Promise, por padrão. Perceberemos isso ao longo do curso.

Se isso der certo, serão retornados os dados que poderemos capturar em uma function, no entanto optaremos por arrow function de nome photos, que é o que chega via back end. E então, a propriedade this.photos receberá o resultado de photos.

http

.get('http://localhost:3000/flavio/photos')

.subscribe(photos => this.photos = photos);

Isto, porém, resulta em erro de compilação, já que o resultado de tipo Object não é compatível com o tipo any[]. Quando não explicitamos o tipo de uma variável em JavaScript, adota-se o tipo any[], ou seja, "qualquer". Portanto, indicaremos que photos será do tipo Object[]:

export class AppComponent {

photos: Object[] = [];

constructor(http:HttpClient) {

http

.get('http://localhost:3000/flavio/photos')

.subscribe(photos => this.photos = photos);

}

}

Para melhor aproveitamento destas aulas, são recomendados os cursos de TypeScript, partes [1](https://cursos.alura.com.br/course/typescript-parte1) e [2](https://cursos.alura.com.br/course/typescript-parte2).

Entretanto, o problema persiste, pois Object não pertence ao tipo Object[]. Nós sabemos que trata-se de uma lista que vem do back end, mas o TypeScript não sabe disso. Para consertar, incluiremos <Object[]> no código, e com isso os tipos começam a fazer sentido entre si.

http

.get<Object[]>('http://localhost:3000/flavio/photos')

.subscribe(photos => this.photos = photos);

Salvaremos, voltaremos ao navegador, e teremos várias imagens diferentes sendo exibidas, provenientes do back end. Para confirmar o bom funcionamento, podemos abrir um bloco na arrow function, usar o console.log(), salvar o arquivo, abrir a página no navegador, e depois o console, e teremos uma lista em que cada imagem possui suas respectivas informações.

http

.get<Object[]>('http://localhost:3000/flavio/photos')

.subscribe(photos => {

console.log(photos);

this.photos = photos

});

E se houver erro, de que forma o trataremos?

O subscribe() recebe dois parâmetros, sendo o primeiro deles o callback a ser chamado caso haja sucesso, e o segundo a ser chamado passando o erro que vem do back end:

http

.get<Object[]>('http://localhost:3000/flavio/photosx')

.subscribe(

photos => this.photos = photos,

err => console.log(err)

);

Salvaremos com um endereço que não existe, nada será exibido no navegador, e no console se verifica o tipo de erro, um HttpErrorResponse. No trecho acima, ainda poderemos incluir err.message em console.log() para que o retorno fique mais sucinto:

Http failure response for <http://localhost:3000/flavio/photosx>: 404 Not Found

Mas por enquanto não lidaremos com tratamento de erros, então voltaremos o código para o que tínhamos feito antes.

http

.get<Object[]>('http://localhost:3000/flavio/photos')

.subscribe(photos => this.photos = photos);

Para melhorarmos o código, primeiro voltaremos a app.component.ts para corrigir a URL que alteramos anteriormente (http://localhost:3000/flavio/photos), e então salvaremos e iremos ao navegador para conferir que tudo funciona conforme esperado. O problema desta abordagem é que, supondo que temos muito mais componentes, e que em sete deles precisamos dar acesso à API de imagens, teremos o seguinte trecho se repetindo em sete locais diferentes:

http

.get<Object[]>('http://localhost:3000/flavio/photos')

.subscribe(photos => this.photos = photos);

Além disso, se o endereço da API for alterado, teremos que lembrar de modificá-lo em todos os outros lugares em que aparece. O componente não precisa saber de detalhes de onde os dados estão vindo. Portanto, isolaremos o acesso à API em uma classe especializada em consumi-las.

No universo do Angular, chamamos estas classes de **serviços**. Na pasta "photo", ou seja, na mesma pasta de photo.component.html e photo.component.ts, criaremos o arquivo photo.service.ts, por motivos de organização e praticidade. Este novo arquivo não é um componente, no entanto continuará sendo uma classe.

Para poder acessar a API, o PhotoService também depende de HttpClient, então acrescentaremos constructor e a propriedade http, tomando cuidado para fazermos a importação de angular/common/http. Ele terá o método listFromUser(), que receberá como parâmetro um userName.

Deste modo, quem utilizar o PhotoService chamará este método passando userName, no caso, trabalharemos com flavio, e retornará todos os dados daquela API.

import { HttpClient } from "@angular/common/http";

export class PhotoService {

constructor(http: HttpClient) {}

listFromUser(userName) {

}

}

Feito isso, voltaremos a app.component.ts, copiaremos o código referente a http, e o colaremos em photo.service.ts:

import { HttpClient } from "@angular/common/http";

export class PhotoService {

constructor(http: HttpClient) {}

listFromUser(userName) {

http

.get<Object[]>('http://localhost:3000/flavio/photos')

.subscribe(photos => this.photos = photos);

}

}

No entanto, não temos acesso ao HttpClient no método listFromUser(), por isso teríamos que incluir algo como http: HttpClient e, no construtor, this.http = http, já que ele, sim, tem acesso. No método, usaríamos this.http, e ficaríamos com o código desta forma:

export class PhotoService {

http: HttpClient;

constructor(http: HttpClient) {

this.http = http;

}

listFromUser(userName) {

this.http

.get<Object[]>('http://localhost:3000/flavio/photos')

.subscribe(photos => this.photos = photos);

}

}

Porém, não é isso que faremos. Vamos utilizar o modificador de acesso private no parâmetro do construtor. Assim, o TypeScript entende que queremos não apenas receber este parâmetro como torná-lo uma propriedade da classe. Com o private, tudo que estiver fora de PhotoService não poderá usar o http, e isto justifica o uso de this.http também. Sem o modificador, teremos um erro de compilação.

Também não podemos usar o subscribe() aqui, já que isto deverá ser feito no momento da busca de dados. O responsável por isso é aquele que for utilizar o método listFromUser(). Lembrando que, da maneira em que está, o observable, por ser *lazy*, não buscará os dados.

Há mais um detalhe: qual o tipo do userName? É any, o que quer dizer que qualquer tipo que passarmos será aceito. No entanto, sabemos que userName será sempre texto, então podemos indicar isto. E se passarmos o mouse sobre this, será exibido que o retorno é void, que equivale a nada, sendo assim acrescentaremos return:

listFromUser(userName: string) {

return this.http

.get<Object[]>(API + '/flavio/photos')

}

Precisamos tomar cuidado para não usarmos **S**tring, com a primeira letra em maiúsculo.

Deste modo, ao passarmos o mouse em this, será retornado um *Observable* do tipo Object[]. Salvaremos o arquivo, abriremos app.component.ts, deletaremos a importação de HttpClient e importaremos photoService do tipo PhotoService, tornando o código muito mais limpo:

export class AppComponent {

photos: Object[] = [];

constructor(photoService: PhotoService) {

photoService

.listFromUser('flavio')

.subscribe(photos => this.photos = photos);

}

}

Voltaremos ao navegador após salvarmos todas as alterações, e haverá um erro de ausência de provedor para PhotoService. Isto ocorre porque quando o Angular cria AppComponent e tenta injetar photoService, não consegue, por desconhecê-lo. É um pouco diferente do que vimos anteriormente: ao criarmos um serviço, usamos @Injectable().

Este *decorator*, que é automaticamente importado do angular/core, indica que photoService é injetável, ou seja, pode receber HttpClient e outros. No entanto, precisamos informar o seu escopo, se será um único PhotoService para a aplicação inteira, ou não.

No nosso caso, se tivermos trinta componentes e quisermos usar o PhotoService, e o mesmo objeto, então passaremos a configuração providedIn, um objeto JavaScript cujo valor é root. Com isso, sinalizamos que quando o Angular for criá-lo, será no **escopo raiz**, isto é, qualquer componente da nossa aplicação que precisar de PhotoService o terá disponível.

@Injectable({ providedIn: 'root' })

export class PhotoService {

constructor(private http: HttpClient) {}

listFromUser(userName: string) {

return this.http

.get<Object[]>(API + '/flavio/photos');

}

}

Salvaremos, retornaremos ao navegador, e tudo continuará funcionando muito bem. E nosso código está organizado de maneira muito mais elegante, inclusive em termos de manutenção e legibilidade. Todavia podemos melhorá-lo ainda mais, e veremos como fazer isso em breve.

Organizamos o código de maneira mais elegante, e o isolamos em uma camada de serviço. Conseguimos exibir as fotos, mas avançaremos um pouco e adquiriremos maturidade em termos de acesso à API. Quando a acessamos via navegador, recebemos uma lista de objetos com propriedades id, postDate, URL, e outras tantas informações que dizem respeito à cada imagem.

Por ora, só estamos utilizando url e description, certo? Para provar isso, em app.component.ts, criaremos um bloco na *arrow function* e, no *array* de photos, queremos acessar o id da imagem da primeira posição. Mas ao digitarmos photos[0]., não são exibidas opções de *autocomplete*. Se tentarmos incluir url logo em seguida, teremos um erro de compilação.

Quando o código está sendo executado, internamente ele roda como JavaScript, sem tipagem, e por isso funciona. Porém, do ponto de vista do TypeScript, esta lista é do tipo Object[], o qual não possui url ou outras propriedades além de constructor, hasOwnProperty(), isPrototypeOf(), e por aí vai.

Sendo assim, não é possível exibir os dados da primeira imagem, pois utilizamos esta tipagem (Object[]). Vamos tentar trocá-la por any[]:

export class AppComponent {

photos: any[] = [];

constructor(photoService: PhotoService) {

photoService

.listFromUser('flavio')

.subscribe(photos => {

photos[0].

this.photos = photos

});

}

Com a tecla "Ctrl" pressionada, e clicando em listFromUser(), acessamos o método, e indicaremos que seu retorno será do tipo any[]:

listFromUser(userName: string) {

return this.http

.get<any[]>(API + '/flavio/photos');

}

Voltaremos a app.component.ts, e o *autocomplete* em photos[0].description continua não funcionando, mas o seu acesso sim. Ou seja, não há mais erro de compilação. E como usamos any[], o tipo poderá ser qualquer um. Incluiremos um console.log() no código, salvaremos, voltaremos ao navegador, abriremos o console, em que encontraremos "Farol iluminado", descrição da primeira imagem.

photoService

.listFromUser('flavio')

.subscribe(photos => {

console.log(photos[0].description);

this.photos = photos

});

Resolvemos o nosso problema? Mais ou menos, porque se digitarmos descript em vez de description, o TypeScript não acusará erro, e no console será lido simplesmente "undefined". Além disso, também perdemos o *autocomplete* ao desenvolvermos a aplicação.

Para corrigirmos isto, tiparemos o retorno da API. Em photo.service.ts informamos que o retorno é do tipo any[], porém trocaremos para Photo[], com propriedades específicas. Então, na pasta "photo", junto ao serviço, criaremos o arquivo photo.ts, que não será um componente, tampouco uma classe, e sim uma interface chamada Photo.

No conceito do TypeScript, a ideia da interface tem a ver com encaixes específicos, e incluiremos nela todas as propriedades dos objetos retornados na lista da API, e atribuiremos um tipo para cada uma delas:

export interface Photo {

id:number;

postDate:Date;

url:string;

description:string;

allowComments:boolean;

likes:number;

comments:number;

userId:number;

}

A interface não diz em nenhum momento quais dados precisam estar em cada uma destas propriedades, e sim o *shape*, a forma que um objeto deve ter.

Em photo.service.ts, importaremos Photo[] clicando no ícone de lâmpada, e o dado será tratado como um *array* deste tipo. Nisto, app.component.ts começará a dar erro, pois quando acessamos a posição 0 do *array*, veremos que não existe a propriedade descript.

Com isso, ficamos menos suscetíveis a erros corriqueiros ao acessarmos estas propriedades, pois as padronizamos e tipamos, e somos mais produtivos com o *autocomplete*. Caso a API mude, a aplicação deixará de funcionar, pois o TypeScript não irá prevenir isso, apenas receberá os dados e tentará tratá-los, sem conseguir acessá-los.

Se o acesso do retorno da API ocorre em inúmeros locais, poderemos abrir a interface, clicar com o botão direito na propriedade alterada, escolher a opção "Rename Symbol", digitar o novo nome e pressionar "Enter". Assim, modificamos o nome da propriedade em todos os locais em que é acessada.

O código de photoService em app.component.ts ficará da seguinte forma:

photoService

.listFromUser('flavio')

.subscribe(photos => this.photos = photos);

Antes de terminarmos a aula, há mais uma melhoria que podemos aplicar. Atualmente, buscamos as imagens e acessamos a API no constructor da classe AppComponent. O que acontece quando o componente é construído? Inicialmente o Angular cria uma instância de AppComponent, e depois aplica o *decorator* @Component para tornar esta instância em um componente efetivamente.

A maneira como organizamos nosso código funciona, mas podemos padronizá-lo mantendo o constructor apenas para injeção de dependência, e qualquer lógica que queiramos executar será colocada em uma fase do ciclo de vida que todo componente Angular possui.

Todo componente Angular possui um ciclo de vida, e focaremos agora em ***ngOnInit*** ou, abreviando, ***OnInit***. Se passarmos o mouse sobre este método da maneira em que está em app.component.ts, nada será retornado, e por isso ele é void. Moveremos o código de photoService para ngOnInit(), mas sabemos que este código não funcionará, pois o photoService é acessível somente no constructor, sendo preciso acessá-lo como propriedade de classe. Para tal, usaremos private:

constructor(private photoService: PhotoService) { }

ngOnInit(): void {

this.photoService

.listFromUser('flavio')

.subscribe(photos => this.photos = photos);

}

No Java, não é necessário incluir this, mas em JavaScript e TypeScript, usamos ele para acessar a propriedade de uma classe. Salvaremos, voltaremos ao navegador, e tudo continua funcionando bem. A fase *OnInit* ocorre depois da instanciação de AppComponent, e depois do componente receber as *inbound properties*.

O mais importante, agora, é que este método nos salvará em algumas situações, mas por enquanto queremos convencionar e usar o construtor apenas para injeção de dependência, e qualquer código de inicialização de configuração será feito no ngOnInit. Porém, se escrevermos o método errado, não temos erro de compilação. No entanto, quando retornarmos ao navegador, nada é exibido.

Isso ocorre porque ao tratá-lo como um componente, o Angular espera encontrar o método ngOnInit() exatamente desta forma, ignorando e deixando de chamá-lo caso ele não seja encontrado. Seria melhor se o TypeScript pudesse nos avisar caso tenhamos algum erro deste tipo.

Podemos voltar o código para a maneira como estava antes de colocarmos o método e, logo após AppComponent, incluir implements OnInit e pressionar "Enter". O programa então fará a importação da interface OnInit de angular/core, que permitirá a definição da forma de um objeto para que possamos tratá-la de maneira tipada, além do uso do *autocomplete*, entre outros.

No entanto, uma interface também nos obriga a usar determinados métodos, então, quando implementamos a interface OnInit e passamos a mouse por cima, o programa diz que está faltando ngOnInit() na nossa classe. Ao clicarmos em AppComponent, no ícone de lâmpada, e então em "Implement interface 'OnInit'", o Angular inclui automaticamente o método ngOnInit(). O nosso código continua compilando, porém ao rodarmos a aplicação teremos um problema, já que o método lança uma exceção.

Assim, moveremos o código de photoService usando a tecla "Alt" junto com as setas para cima ou para baixo, para dentro do método. Em seguida, moveremos todo o método para após o constructor, por convenção, e tornaremos photoService privada. Incluiremos this antes de photoService, e então poderemos salvar o arquivo.

import { Component, OnInit } from '@angular/core';

import { PhotoService } from './photos/photo/photo.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent implements OnInit{

photos: any[] = [];

constructor(private photoService: PhotoService) { }

ngOnInit(): void {

this.photoService

.listFromUser('flavio')

.subscribe(photos => this.photos = photos);

}

}

Desta vez, se digitarmos algo errado, a classe não irá compilar, e nenhuma alteração do projeto será vista enquanto o ngOnInit() não for implementado. Salvaremos, voltaremos ao navegador, e tudo continuará funcionando conforme esperado. O Angular possui outros ciclos de vida, mas por enquanto vimos este para convencionar que o constructor será destinado à injeção de dependência, e qualquer inicialização que queiramos fazer posteriormente será no ngOnInit().

Por fim, acessando a definição da classe photoService (clique com "Ctrl"), sabemos que ela depende de um HttpClient, e para que ele esteja disponível, o app.module.ts faz a importação de HttpClientModule. Mas se pararmos para pensar, quem depende disso é app.module.ts ou photos.module.ts? Porque se photo.service.ts pertence a photos.module.ts, a ideia é que o segundo forneça o HttpClient de que necessita.

Caso HttpClient não esteja em app.module.ts, a aplicação não irá funcionar. Então, garantimos que a dependência necessária para este módulo funcionar esteja em photos.module.ts, e por isso incluiremos a linha imports: [ HttpClientModule ] em @NgModule, e removeremos HttpClientModule e a linha referente ao seu *import* em app.module.ts.

Salvaremos, voltaremos à aplicação, e tudo continua funcionando como antes. A grande questão é que se pegarmos a pasta photos.module.ts, ela importará tudo que for necessário para que seus componentes funcionem e, agora, sim, ela depende de HttpClientModule.

Antes de continuarmos, lidaremos com outra questão que envolve organização do código. Fazemos estas alterações aos poucos, e não tudo de uma vez desde o início, para entendermos a motivação por trás de cada um desses passos. Da maneira em que está, o componente responsável pela exibição de imagens é o AppComponent, e tanto isto é verdade que ele é o primeiro componente a ser carregado pela nossa aplicação.

Porém, se pensarmos no domínio de imagens, o módulo da nossa aplicação que guarda tudo que diz respeito a imagens é PhotosModule. Sendo assim, que tal se criarmos um componente dentro deste módulo, cuja responsabilidade será exibir as imagens?

Com isso, o AppComponent não será responsável por nada, sendo apenas o primeiro componente a ser carregado na página. Além disso, posteriormente teremos uma forma de submeter imagens, e outros recursos, como outro componente com o escopo de página (isto é, componentes que se equivalem a páginas na aplicação).

Então, criaremos na pasta "photos", que é onde se localiza PhotosModule, o componente PhotoListComponent. Entretanto não o faremos manualmente, pois sabemos que precisamos criar os arquivos photo-list.component.ts, photo-list.component.html, e outros, adicioná-los na declaração do módulo, e tudo o mais.

Usaremos o próprio Angular CLI para esta tarefa. Lembrando que estamos trabalhando com dois terminais, sendo um deles o servidor com a nossa API e o outro o Angular CLI, que é o que usaremos. Com "Ctrl + C" para pausar, executaremos dentro da pasta "alurapic" o comando ng generate component photos/photo-list.

Todo componente gerado automaticamente fica na pasta "app", mas queremos que PhotoListComponent esteja na pasta "photos". Por consequência, todos os demais arquivos serão criados nela também, inclusive alguns arquivos de testes que não usaremos por enquanto. Além disso, o comando faz um *update* em photos.module.ts.

Ao abrirmos este arquivo, veremos inclusive um .css que inicialmente não usaremos, mas que está pronto para uso, e todo o *kit*, com a boa prática de implementar o OnInit e o método correspondente para o caso de querermos incluir algum código na inicialização. Em declarations de photos.module.ts, também se encontra PhotoListComponent.

Em app.component.ts, o conteúdo de ngOnInit() junto com constructor e photos será recortado, deletaremos implements OnInit, bem como a linha referente ao *import* de PhotoService, e OnInit entre as chaves no *import* localizado logo acima deste. Assim, o código ficará simplesmente deste modo:

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent { }

Ou seja, AppComponent não fará nada. Salvaremos o arquivo, acessaremos photo-list.component.ts, e colaremos o trecho de código recortado anteriormente:

export class PhotoListComponent implements OnInit {

photos: any[] = [];

constructor(private photoService: PhotoService) { }

ngOnInit(): void {

this.photoService

.listFromUser('flavio')

.subscribe(photos => this.photos = photos);

}

}

Teremos que fazer o *import* de PhotoService novamente, e para isto clicaremos no ícone de lâmpada e optaremos por "Import 'PhotoService' from module "../photo/photo.service"". Salvaremos as alterações. AppComponent não está utilizando nada de photos.module.ts, e se observarmos este arquivo, PhotoComponent está em exports. Isto porque app.module.ts importa PhotosModule e, para que ele tenha acesso ao PhotoComponent, ele teve que ser exportado em photos.module.ts.

Mas se tudo que diz respeito a imagens está dentro de "photos" e de photos.module.ts, não é necessário exportarmos tal componente, já que ninguém externo a este módulo irá utilizá-lo. Se isso acontecer, daí o exportaremos, sob demanda. O importante é entendermos que em declarations há PhotoComponent e PhotoListComponent. Por fazerem parte do *array* de declarations, um enxerga o outro, e isso é o suficiente.

Salvaremos todo o projeto, e ao retornarmos ao navegador, nada é exibido. Claro, porque o primeiro componente a ser carregado pela aplicação é AppComponent, que não exibe nada. Temos um problema para resolver: desta vez, no lugar da exibição deste componente, queremos que PhotoListComponent seja exibido. E mais, quando tivermos um cadastro de imagem, por exemplo, queremos a possibilidade do usuário obter acesso a isso.

Então, de que forma lidaremos com a questão de ora exibirmos um componente que equivale a uma página e ora um componente que equivale a outra? É isso que veremos nesta aula, ao longo dos próximos vídeos.

Vamos continuar fazendo ajustes em nossa aplicação. Se removemos o código de AppComponent para app.module.ts, também recortaremos o template de app.compoment.ts, que portanto ficará vazio, e o colaremos em photo-list.component.html:

<ap-photo

\*ngFor="let photo of photos"

[url]="photo.url"

[description]="photo.description">

</ap-photo>

Quando o Angular CLI faz a importação de PhotoListComponent em photos.module.ts, ele foi incluído como sendo o último, porém adotamos a convenção de que as importações do Angular vêm primeiro, seguidas das importações dos componentes criados por nós, portanto ajeitaremos o código de acordo.

Outro ponto a ser resolvido é que, por padrão, o *auto import* do Visual Studio Code utiliza aspas, e sabemos que em JavaScript a preferência é por "aspas simples", ou apóstrofos. É isto que o Angular CLI fez ao adicionar o *import* de PhotoListComponent em PhotosModule. Será que existe uma maneira de ensinarmos ao programa que durante as importações devemos utilizar aspas simples?

Sim! Para configurarmos isto, clicaremos em "File > Preferences > Settings". E então, do lado esquerdo serão exibidas as configurações padrão do Visual Studio, enquanto que do lado direito estarão as configurações do usuário, em que incluiremos as seguintes linhas entre chaves:

"javascript.preferences.quoteStyle":"single",

"typescript.preferences.quoteStyle": "single"

Podemos fazer uma simulação removendo import { HttpClientModule } from "@angular/common/http", salvando o arquivo e importando-o novamente. Veremos que a configuração foi salva e funciona perfeitamente. Então, vamos substituir todas as aspas duplas existentes nos códigos por aspas simples para padronizarmos, e a partir daqui a nova configuração cuidará disso para nós.

**A Ana está trabalhando com Angular e precisa criar um novo componente. Ela está usando o Angular CLI para essa tarefa e digitou o comando:**

**ng generate component users/user-list**

**O comando executou com sucesso, mas em qual pasta o componente será criado?**

src/app/users/user-list

Vamos continuar! Ao abrirmos o projeto, a aplicação é carregada pela primeira vez, o AppComponent é exibido e, como ele não tem nada no template, a página do navegador está em branco. Em teoria, isto não implica em erro algum. Entretanto, se consultarmos o console do navegador, há a acusação de "Template parse errors", e que ngForOf (o mesmo que \*ngFor) não é uma propriedade conhecida de app-photo.

Apesar de photo-list.component.html não ter sido carregado, o Angular verificará a **integridade do template e do componente no processo de compilação**. Por que o Angular acha que \*ngFor é uma propriedade do componente?

Não tínhamos este erro quando o nosso componente de imagens era acessado pelo PhotoComponent, cujo principal módulo aplicação importa BrowserModule, que traz todas as diretivas além de outras necessidades do Angular para que a aplicação rode no navegador.

Então, a diretiva \*ngFor é carregada por meio deste BrowserModule. As diretivas estão disponíveis para app.module.ts e todos os seus componentes, que no caso é apenas AppComponent, e em nenhum momento há importação de diretivas do Angular para usarmos em photos.module.ts.

Não podemos, em hipótese alguma, importar um BrowserModule em outro local que não seja o módulo principal da aplicação. Porque além das diretivas padrão do Angular, o BrowserModule traz uma série de outros recursos para a aplicação funcionar.

De que forma, então, teremos acesso à diretiva \*ngFor em outros módulos, já que não poderemos importar BrowserModule?

A boa notícia é que, internamente, o BrowserModule possui especificidades que fazem a aplicação rodar no navegador, e também importa um módulo denominado CommonModule. É ele que possui as declarações das diretivas, e por aí vai. Sendo assim, em imports de photos.module.ts, incluiremos CommonModule, cuja importação automática do Angular colocaremos no primeiro conjunto de importes, seguindo nossas convenções:

import { NgModule } from '@angular/core';

import { HttpClientModule } from '@angular/common/http';

import { CommonModule } from '@angular/common';

//imports dos componentes PhotoComponent e PhotoListComponent

@NgModule({

declarations: [

PhotoComponent,

PhotoListComponent

],

imports: [

HttpClientModule,

CommonModule

]

})

export class PhotosModule {}

Salvaremos e voltaremos ao navegador, que não exibe nada; o erro, porém, deixa de existir.

Todo módulo a ser criado deverá ser importado como *module*; trata-se de uma boa prática para tornar as diretivas do Angular, como \*ngFor e outras, disponíveis.

Precisamos exibir photo-list.component.html para que a listagem de fotos esteja funcionando bem no navegador. Para complicar um pouquinho, criaremos um novo componente, simplesmente para que haja dois componentes em photos.module.ts. Este componente exibirá um título, e permitirá que saibamos que ora exibiremos a listagem e ora carregaremos a página do formulário para cadastro de novas imagens.

No terminal, pausaremos o *command-line interface* com "Ctrl + C", e criaremos photo-form dentro da pasta "photos" usando o comando ng generate component photos/photo-form, e com isso ele aparece automaticamente em declarations de photos.module.ts. Salvaremos o arquivo.

Nosso photo-form.component.html terá a frase "photo-form works!" que manteremos para posteriormente verificarmos se tudo funciona como queremos; poderia ser um conteúdo qualquer. Em seguida, utilizaremos o comando ng serve --open e, no Visual Studio, teremos dois componentes com escopo de página, com uma granularidade alta — "photo-list" e "photo-form".

Se verificarmos a nossa aplicação, da maneira em que ela está estruturada, o primeiro componente a ser carregado e que pode ser conferido em index.html é <app-root>. Então não faz sentido incluirmos um *selector* de "photo-form" ou de "photo-list", já que isso será fixo. Queremos que o componente adequado seja carregado de acordo com o endereço que o usuário digitar.

É neste sentido que entraremos agora na parte de **roteamento** de uma *Single Page Application*. Sendo deste tipo, a página index.html se encarrega de exibir tudo o que a aplicação possui, sem carregar outras páginas. Como funciona esta lógica?

Sabemos que, se acessarmos http://localhost:4200/, a index.html será carregada, que por sua vez carregará, por padrão, <app-root>. Se em vez deste endereço usarmos http://localhost:4200/user/flavio, a impressão é de que queremos acessar uma rota do back end. Da mesma forma, http://localhost:4200/p/add parece uma requisição para o back end.

No entanto, não é assim que funciona. Para o Angular identificar o que será a index.html, ela não carregará <app-root>, e sim funcionará como uma grande lacuna, a qual, de acordo com os endereços digitados, fará um "de/para", verificando origem da rota e tomando uma ação adequada com o componente correspondente.

Chamamos esta rota de ***path***, um caminho. Cada caminho da aplicação, do nosso sistema de rotas, estará associado a um componente. É como se o Angular, ao receber certo endereço no navegador, antes deste acessar o back end e tentar fazer uma requisição, intervirá e verificará se a rota é cadastrada por ele.

Se for este o caso, a requisição não irá ao back end, e a rota ficará com o Angular, que carregará o respectivo componente, interceptando a mudança do endereço, e verificando se é uma rota válida, e por fim carregando o respectivo componente. Caso não seja uma rota válida, a requisição segue para o back end.

O componente <app-root> continuará sendo o primeiro a ser carregado pela aplicação, e dentro de "app", criaremos o arquivo app.routing.module.ts, um módulo, sendo necessário importá-lo. Criaremos este arquivo manualmente, digitando seu conteúdo para entendermos melhor seu fundamento. Por ser um módulo, ele terá @NgModule, e uma constante denominada routes ("rotas"), que será um *array* JavaScript.

Este *array* terá a configuração de cada rota da nossa aplicação, e será responsável pelo carregando do componente adequado. Cada configuração é feita por meio de um objeto JavaScript no qual indicamos um path, isto é, para user/flavio (por enquanto fixo) carregaremos o component PhotoListComponent e faremos o *auto import*. A outra rota é para p/add, para o qual será carregado o componente PhotoFormComponent.

Porém, da maneira como está, teremos problemas, pois o Angular espera encontrar um objeto com propriedade Path. A tipagem estática do TypeScript nos ajuda nisso, então indicaremos que routes é do tipo Routes. Porém, ao tentarmos importá-lo, o Angular CLI não nos dará uma sugestão, pois seu módulo de rotas ainda não foi carregado.

Como queremos que ele não seja carregado ainda, o importaremos manualmente:

import { NgModule } from '@angular/core';

import { Routes } from '@angular/router';

import { PhotoListComponent } from './photos/photo-list/photo-list.component';

import { PhotoFormComponent } from './photos/photo-list/photo-form.component';

const routes: Routes = [

{ path: 'user/flavio', component: PhotoListComponent },

{ path: 'p/add', component: PhotoFormComponent }

];

@NgModule({

})

export class AppRoutingModule { }

Ao fazermos isto, o programa entende que cada item do *array* precisa ser um objeto JavaScript com tais propriedades, e tudo está muito bem tipado. Continuando, em NgModule precisamos importar o RouterModule, que é o módulo de rotas do Angular. Entretanto, deste modo, o AppRoutingModule importa o RouterModule, o qual não está linkado ao arquivo de rotas que acabamos de criar (routes), ou seja, não há uma relação entre eles.

Então, não importaremos RouterModule diretamente em imports, e utilizaremos forRoot(). Sendo nossa aplicação localhost:4200, tudo o que for 4200/ terá relação com a rota raiz. Então, o resultado de forRoot() será importado, e levará em consideração esta lista de rotas (RouterModule).

@NgModule({

imports: [ RouterModule.forRoot(routes) ]

})

É como se tivéssemos um RouterModule pré-configurado que conhece as rotas da aplicação de antemão, em vez de passarmos um RouterModule que não sabe de nada. Salvaremos o arquivo e acessaremos app.module.ts, e importaremos AppRoutingModule. Com isso, o BrowserModule virá junto, com as rotas pré-configuradas.

Dica: às vezes o Angular CLI fica um pouco perdido quando há importação de um arquivo de módulo de rotas. Então, antes de tudo, é recomendado pausar o terminal do Angular e em seguida rodá-lo novamente.

Abriremos o navegador e consultaremos o console. Não há nenhuma mensagem de erro, então vamos tentar acessar a rota digitando localhost:4200/user/flavio na barra de navegação. Nada acontece, nenhuma requisição ao back end é realizada, e nenhum dado é exibido. Por que?

O primeiro componente a ser carregado pela aplicação é app.component.html, certo? Sendo assim, precisaremos indicar nele que, de acordo com um determinado endereço, o componente correspondente deverá ser carregado, e que isso deve ser exibido neste mesmo arquivo. Para isso, usaremos a diretiva <router-outlet>, deixando o arquivo da seguinte forma:

<router-outlet></router-outlet>

É dentro desta *tag* que deverá se exibir o componente. Salvaremos, retornaremos ao navegador, e nos depararemos com uma mensagem de erro indicando que <router-outlet> é um elemento desconhecido, e que não é um componente, solicitando a verificação de seu pertencimento a algum módulo.

O que acontece é que AppModule importa AppRoutingModule, bem como os arquivos de configuração de rotas. No entanto, não consegue importar as diretivas de rotas do módulo RouterModule. Para resolvermos isso, poderíamos simplesmente importá-lo em app.module.ts, porém optaremos por uma alternativa: em @NgModule de app.routing.module.ts, exportaremos RouterModule.

@NgModule({

imports: [

RouterModule.forRoot(routes)

],

exports: [ RouterModule ]

})

Assim, informamos que quem for importar AppRoutingModule também receberá RouterModule, o que fará com que não haja responsabilidade de importá-lo. Com esta alteração, nada será modificado em app.module.ts, voltaremos ao navegador e acessaremos localhost:4200/user/flavio mais uma vez, e as imagens serão carregadas com sucesso. E se acessarmos localhost:4200/p/add, o outro componente também terá o carregamento bem sucedido.

Nosso sistema de rotas funciona de acordo com o esperado, o que pode ser confirmado se digitarmos o endereço localhost:4200/user/flavio na barra de navegação. Isto fará com que a página contendo a lista de imagens seja carregada. Do mesmo modo, se digitarmos localhost:4200/p/add, o photo-form será carregado. Mas e se colocarmos um endereço que não existe?

Nada será exibido, e se consultarmos o console, haverá um erro indicando que não há nenhuma rota que se relacione com o endereço digitado. Podemos melhorar isso, e uma solução é incluir uma terceira rota em routes de app.routing.module.ts que, ao ser carregada, para qualquer endereço que não sejam estas duas opções, carregará PhotoListComponent:

const routes: Routes = [

{ path: 'user/flavio', component: PhotoListComponent },

{ path: 'p/add', component: PhotoFormComponent },

{ path: '""', component: PhotoListComponent }

];

No entanto, não queremos utilizar esta abordagem. Queremos exibir ao usuário uma página indicando que o endereço digitado não existe, ou não foi encontrado. A questão é: onde estará esta página não encontrada? No módulo da pasta "photos", ou em app.module.ts?

Nenhum dos dois. Como esta questão não possui relação direta com "photos", e também não queremos importá-lo em app.module.ts, a ideia é que ele contenha apenas as importações dos módulos que a aplicação realmente precisa utilizar. Criaremos, então, um módulo chamado "errors" diretamente na pasta "app".

Faremos isso pausando o Angular no terminal, e utilizando o comando ng generate module errors. Automaticamente, uma subpasta em "app" será criada, dentro da qual será criado o módulo, além do arquivo errors.module.ts, com o importe de CommonModule.

Então, vamos criar o componente not-found, voltando ao terminal e digitando ng generate component errors/not-found. Ele será criado em "app > errors", sendo o primeiro erro que teremos. Qualquer outro tipo de erro, como acesso não autorizado ou algo assim, também será criado ali. Além disso, foi declarado NotFoundComponent no *array* de declarations automaticamente.

Em not-found.component.ts há sua estrutura básica, com implementação do OnInit e tudo o mais. Em not-found.component.html, substituiremos todo o conteúdo por:

<div class="text-center">

<h2>This page is not avaliable</h2>

<p>

The link you have acccessed may be broken or

the page may have ben removed.

</p>

</div>

Trata-se de um texto estático que indica que a página não foi encontrada. Utilizamos a classe text-center do Bootstrap para que a estrutura (ou seja, o subtítulo e o parágrafo) fique centralizada. Salvaremos o arquivo, e com isto o componente estará pronto, possuindo apenas um template, uma apresentação. Agora, basta incluí-lo em nossa rota.

Em app.routing.module.ts, onde está PhotoListComponent no último path, colocaremos NotFoundComponent. Salvaremos e testaremos seu funcionamento retornando ao terminal, subindo novamente o Angular CLI por meio do comando ng serve --open. O ambiente de desenvolvimento será construído, após o qual o navegador será aberto.

Ocorre um erro já visto anteriormente, indicando que NotFoundComponent não faz parte de NgModule, sendo que, em realidade, isto não é verdade. Em nosso projeto, ele faz parte de errors.module.ts e, se consultarmos app.module.ts, em nenhum momento ErrorsModule está sendo carregado. Assim sendo, o componente da rota, NotFoundComponent, não estará disponível na aplicação, então faz sentido o console informar que o que estamos tentando usar em app.routing.module.ts não pertence ao módulo, pois este módulo do qual ele faz parte não foi carregado.

Para resolvermos isto, em app.module.ts faremos a importação de ErrorsModule:

imports: [

BrowserModule,

PhotosModule,

AppRoutingModule,

ErrorsModule

]

Assim, o Angular saberá que este componente pertence ao módulo, e tudo funcionará bem. Desta vez, ao abrirmos o navegador, se testarmos uma página inexistente, será exibida a mensagem "This page is not available" ("Esta página está indisponível").

Ainda sobre rotas, temos um ponto interessante em nossa aplicação: se acessarmos localhost:4200/user/flavio, significa que queremos ir à nossa API, em photo-list.component.ts, e exibir as imagens referentes a este cadastro. No entanto, queremos que, ao digitarmos localhost:4200/user/almeida, isto é, utilizarmos outro usuário, também seja uma rota válida para o Angular.

Portanto, precisaremos passar o nome da URL para o PhotoService. Buscamos, então, uma **rota parametrizada**, que não dependa de URLs fixas, variando de acordo com o usuário cadastrado. Da maneira em que está, se tentarmos qualquer outro usuário, cairemos na página de erro.

Em AppRoutingModule, queremos tornar este segmento da URL uma variável, algo genérico, como se funcionasse como um coringa. Para tal, basta colocarmos :userName no lugar de flavio:

const routes: Routes = [

{ path: 'user/:userName', component: PhotoListComponent },

{ path: 'p/add', component: PhotoFormComponent },

{ path: '""', component: NotFoundComponent }

];

Com isso, embora o sistema de rotas não aceite erros de digitação em "user" na barra de endereços, passará a aceitar qualquer usuário. Isso significa que a rota pode ser localhost:4200/user/ seguido de qualquer userName para ser válida. O mais importante neste momento é que queremos ter acesso a este parâmetro da rota no componente, para passarmos ao nosso serviço, porque não faz sentido passarmos almeida como userName na barra de endereços do navegador, mas estar flavio em ngOnInit(), em photo-list.component.ts.

Usaremos uma ferramenta do Angular, disponibilizada pelo próprio serviço (módulo) de rotas. Em photo-list.component.ts recebemos photoService no construtor, e queremos activatedRoute, o qual indica a rota ativada em determinado momento. Em ngOnIt(), incluiremos const userName = this.activatedRoute.snapshot.params.userName, cujo snapshot equivale a uma fotografia do momento desta rota:

constructor(

private photoService: PhotoService,

private activatedRoute: ActivatedRoute

) { }

ngOnInit(): void {

const userName = this.activatedRoute.snapshot.params.userName;

this.photoService

.listFromUser(userName)

.subscribe(photos => this.photos = photos);

}

Clicaremos em listFromUser no trecho acima para acessarmos photo.service.ts, em que substituiremos flavio, e deixaremos o código assim:

listFromUser(userName: string) {

return this.http

.get<Photo[]>(API + '/' + userName + '/photos');

}

Salvaremos todas as alterações e, quando acessarmos a rota passando almeida, o userName, que é o segmento da rota, o receberá. Extrairemos este valor, passando-o ao método listFromUser(), o qual o concatenará para montar a **URL dinâmica**. É deste modo que conseguimos ter um segmento de rotas parametrizado e, em um componente, por meio de activatedRoute, conseguimos acessar o valor que foi passado à rota.

Por mais que a página exibida seja responsiva, as imagens não estão ordenadas de forma visualmente agradável. Já que estamos utilizando o Bootstrap, seria interessante aplicarmos seu *grid* para exibirmos uma imagem ao lado da outra.

No template de photo-list.component.html, por motivos de semântica, utilizaremos uma lista ordenada, pois o back end retorna as imagens das mais recentes às mais antigas. Essa <ol> não terá *bullets* nem nada do tipo, portanto usaremos a classe list-unstyled, e conterá itens <li>, cada um sendo uma imagem.

Assim, a diretiva \*ngFor deixará de ficar em ap-photo, e sim em <li>. Em relação ao *grid*, é necessário definir quais são as linhas, portanto utilizaremos a classe row, isto é, a lista, que é <ol>, equivalente a uma linha, e indicaremos que cada <li> terá a classe col-4. Isso porque o *grid* do Bootstrap possui 12 colunas, então, neste caso, cada <li> guardará 4 delas, somando 12 no total.

Não poderemos passar deste valor, caso contrário teremos problemas no layout. Salvando o código abaixo, voltaremos ao navegador e, aparentemente, tudo funciona conforme esperado.

<ol class="list-unstyled row">

<li \*ngFor="let photo of photos" class="col-4">

<ap-photo

[url]="photo.url"

[description]="photo.description">

</ap-photo>

</li>

</ol>

Entretanto, se inspecionarmos o elemento do DOM, temos 12 <li>s, cada um com tamanho 4. Não faz sentido, isto logo ficará sobrecarregado. Como resolveremos isso respeitando o Bootstrap, e sem ter problemas futuros no layout da aplicação? Criaremos um novo componente com a responsabilidade de receber a lista de imagens e renderizá-la na forma correta para o *grid* do Bootstrap.

Criaremos o componente photos, a ser utilizado por photo-list, que diz respeito à página. Por ser um componente muito atrelado a photo-list, o criaremos dentro desta pasta. Assim, se quisermos saber quais componentes dividimos para ajudar na organização de exibição do maior componente, que é photo-list, encontraremos nesta pasta.

Pausaremos o Angular CLI no terminal e digitaremos ng generate component photos/photo-list/photos. Se checarmos photos.module.ts, o PhotosComponent foi automaticamente adicionado em declarations. Para evitarmos que tudo fique em um mesmo componente, dividimos elementos em photo-list e photos, sendo este especializado em exibir as imagens de acordo com o *grid* do Bootstrap.

Em seguida, abriremos photo-list.component.html, recortaremos seu conteúdo e o colaremos em photos.component.html, para que o componente photos.component.ts receba uma lista de imagens do tipo Photo, um *array* inicialmente vazio. Clicaremos no ícone de lâmpada para que o tipo seja importado. Sabemos que para passarmos um dado para este componente, ele precisa ser um *inbound property*, portanto também importaremos @Input.

export class PhotosComponent implements OnInit {

@Input() photos: Photo[] = [];

constructor() { }

ngOnInit() {

}

}

No momento de criação do componente, app-photos está prefixado com "app". Precisamos tomar cuidado, pois a nossa aplicação é denominada "alurapic". Assim, usaremos "ap-photos" no selector de @Component em photos.component.ts, "ap-photo-form" em photo-forms.component.ts e "ap-not-found" em not-found.component.ts.

Recapitulando, em photos.component.ts passamos uma lista de imagens em Photo, e seu selector é ap-photos. Precisaremos utilizar este componente em photo-list.component.html, então incluiremos as tags <ap-photos> e o *Data binding* de photos:

<ap-photos [photos]="photos"></ap-photos>

Em photo-list.component, onde está photos: any[] = [], trocaremos por photos: Photo[] = []. E no arquivo photo-list.component.html usaremos o componente ap-photos, e associaremos a *inbound property* photos à propriedade photos, que guarda as imagens provenientes do back end.

Vamos testar tudo no navegador? Antes disso, voltaremos o Angular CLI pelo terminal.

Em suma, retiramos a responsabilidade de apresentação do componente de photo-list para photos. E toda vez que acessarmos o navegador cairemos na página de erro, pois precisamos agora especificar o usuário. Se testarmos com localhost:4200/user/flavio, as imagens serão exibidas como anteriormente, porém ainda com o problema de termos um único *row* (linha) com várias *columns* (colunas).

É necessário ajustarmos estes dados no componente photos, no ato de recebê-los, para que o *grid* do Bootstrap funcione corretamente.

Precisamos entender que nosso template (photos.component.html) não pode se basear na lista de imagens recebidas, e sim em uma lista de dados que criaremos e começará em branco. Sabemos que se não tiparmos, este *array* será any[]. Precisaremos adequar esta estrutura para uma nova, com um *array* cujo limite de itens será 3. Assim, cada *array* corresponderá a uma linha com 3 colunas.

Em photos.component.ts, teremos:

@Input() photos: Photo[] = [];

rows = any[] = [];

Neste sentido, em photos.component.html definiremos como row a nossa <li>. Isto porque a primeira lista será composta de três imagens, a segunda também, e por aí vai. E não iteraremos mais de photos, mas sim de rows. Também incluiremos uma <div> de classe col-4, pois queremos 3 itens para totalizar 12, e assim utilizarmos todo o *grid* do Bootstrap.

No entanto, não precisaremos fazê-lo para cada item da coluna, então teremos outro \*ngFor dentro deste. A diferença é que neste iteraremos de cols, e trocaremos cols de "let cols of" por photo:

<ol class="list-unstyled">

<li \*ngFor="let cols of rows" class="row">

<div \*ngFor="let photo of cols" class="col-4">

<ap-photo

[url]="photo.url"

[description]="photo.description">

</ap-photo>

</div>

</li>

</ol>

Falta implementarmos a lógica para a construção da lista rows de photos.component.ts. Em ngOnInit() indicaremos que this.rows receberá o resultado de this.groupColumns(), um método a ser criado e que recebe a lista de photos. Criaremos um *array* chamado newRows que começa vazio, e sabemos que teremos que retorná-lo em algum momento.

Faremos uma iteração de 3 em 3, então utilizaremos um for:

ngOnInit() {

this.rows = this.groupColumns(this.photos);

}

groupColumns(photos: Photo[]) {

const newRows = [];

for(let index = 0; index < photos.length; index+=3) {

newRows.push(photos.slice(index, index + 3));

}

return newRows;

}

Isso é um exemplo clássico — quando se quer resolver um problema, o *One framework* não ajuda em nada, sendo necessário aplicar uma lógica de JavaScript e, claro, conhecer esta linguagem.

O slice() sempre recebe a posição inicial que queremos considerar, e a final **não inclusiva**, "fatiando" o *array*. Ou seja, quando o primeiro index é 0, o outro vale 3, e o slice() pegará a fatia de 0 a 2. Esta segunda posição não é inclusiva, e se tivéssemos colocado a posição final como 2, seriam pegos 0 e 1, por isto utilizamos index + 3. Na passada seguinte, o primeiro index será 3, e o final, 6, sendo pegos 3, 4 e 5.

Não há problema se no final sobrarem dois ou apenas um elemento, pois o slice() só trará a quantidade existente. Vamos testar? No navegador, nada é exibido, e no console não há nenhuma mensagem de erro. Ao começarem a criar componentes no Angular, muitas pessoas passam por esta dificuldade, e trataremos dela a seguir.

Vamos entender nosso problema: quando photo-list.component.ts é carregado, o valor de photos é um *array* vazio. Porém, a busca de dados é uma operação assíncrona, e demora milésimos de segundo até ser executada e lançar um novo valor. Então, quando photos começa com um *array* vazio, no template (photo-list.component.html) o *Data binding* para ap-photos passa um *array* vazio também.

Significa que photos está vazio e, na inicialização do componente, feita uma única vez, ele tentará acionar o groupColumns(). Será lançado um novo valor para o *array* em photo-list.component.ts, o qual automaticamente irá para photo-list.component.html, por conta do *Data binding*.

Ou seja, apesar de o *array* de @Input() photos: Photo[] = [] em photo-list.component.ts está sendo atualizado, em nenhum momento estamos chamando this.groupColumns(). Ele só estava sendo chamado na inicialização do componente, e apenas uma vez. Precisamos chamá-lo toda vez que a propriedade da *inbound property* mudar.

Sendo assim, ngOnInit() não nos servirá, e trocaremos a implementação do OnInit por OnChanges, não esquecendo de o deletarmos da lista de importes no começo do código de photos.component.ts. Usando a interface OnChanges, quando clicamos em PhotosComponent, o Visual Studio não nos permite implementar o método, mas se logo abaixo de constructor digitarmos ngOnChanges(), ele nos dará a opção de adicionarmos um método com este nome.

Este método recebe como parâmetro todas as possíveis mudanças das *inbound properties* do nosso componente. Tais mudanças são do tipo SimpleChanges, que importaremos de angular/core. Caso haja alguma mudança, uma propriedade com mesmo nome da *inbound property* que sofreu a mudança será adicionada dinamicamente. Se não houver mudança, tampouco haverá propriedade.

Vamos testar isso implementando if para o caso de haver mudanças especificamente na *inbound property* photos e, caso positivo, executaremos this.groupColumns() passando os novos dados das imagens. Testamos com photos pois poderemos ter várias propriedades, porém apenas uma delas sofrer alteração. É necessário testar cada propriedade da *inbound property*.

export class PhotosComponent implements OnChanges {

@Input() photos: Photo[] = [];

rows: any[] = [];

constructor() { }

ngOnChanges(changes: SimpleChanges) {

if(changes.photos)

this.rows = this.groupColumns(this.photos);

}

groupColumns(photos: Photo[]) {

const newRows = [];

for(let index = 0; index < photos.length; index+=3) {

newRows.push(photos.slice(index, index + 3));

}

return newRows;

}

}

**Cuidado**: existe o tipo SimpleChange. Aqui, usamos o "Changes", no plural!

Vamos salvar, voltar ao navegador, em que temos as imagens sendo exibidas. Inspecionaremos esta estrutura no DOM, e teremos a <ol> com cada <li> contendo três <div>s, exceto pela última, que contém duas. Conseguimos resolver a questão do *grid*, e evoluímos um pouco em termos de criação de um componente, lidando com *edges cases* (problemas que raramente acontecem, porém não são impossíveis) como este.

Vamos melhorar a experiência do usuário em nossa aplicação. Atualmente temos 12 imagens em exibição na página; e se futuramente tivermos 100, 200? Seria interessante darmos ao usuário a opção de filtrar, por exemplo. Sabemos que cada objeto utilizado para a construção destes componentes possui a propriedade description, a ideia é colocarmos um campo de filtro no qual digitaremos parte do texto que está nesta descrição, e todas as imagens que a possuírem serão exibidas, enquanto as demais serão omitidas da página.

Em photo-list.component.html, que envolve o componente que traz os dados e os disponibiliza para photos.component.html, componente que os renderiza usando o *grid* do Bootstrap, acrescentaremos o seguinte trecho:

<div class="text-center mt-3 mb-3">

<form>

<input

class="rounded"

type="search"

placeholder="search..."

autofocus>

</form>

</div>

Com isso, o <input> ficará centralizado na <div> e terá bordas arredondadas (rounded), será do tipo search, e possuirá foco assim que a página for carregada. E as classes mt-3 e mb-3 implicam nas margens de topo e de baixo, ambas com valor 3, valor arbitrário do Bootstrap. Salvaremos e voltaremos ao navegador, com o campo de busca implementado, porém ainda não pronto para uso.

De que maneira capturaremos o dado digitado neste campo de busca? E depois, como será realizada a filtragem deste valor? Em algum momento sabemos que este valor deverá cair em uma propriedade de PhotoListComponent, portanto criaremos em photo-list.component.ts a propriedade filter, do tipo string, que começa inicializada com uma *string* em branco. À medida em que formos digitando no *input* do navegador, o valor será atribuído a esta propriedade.

Qual é o evento do JavaScript podemos trabalhar para lidar com isso? Uma das opções é o keyup, disparado à medida em que digitamos. Para entendermos melhor, vamos supor que trabalhamos com JavaScript puro, e sem *arrow function* para facilitar a legibilidade:

h1.addEventListener('keyup', function(event) {

alert('oi')

});

Com este código, a cada valor digitado pelo usuário no h1, acessado em DOM, será exibido um alerta. Por meio do event conseguimos acessar o valor de alert('oi'), sendo possível utilizarmos event.target.value para saber o que foi digitado.

Voltando ao código que estávamos escrevendo, indicaremos em keyup que filter receberá $event.target.value, o que se assemelha ao JavaScript puro que usamos de exemplo, com a diferença de que, no lugar de event, usamos $event do Angular, tratado e modificado.

<form>

<input

class="rounded"

type="search"

placeholder="search..."

autofocus

keyup="filter = $event.target.value"

>

</form>

Então, a cada valor que for digitado no campo de busca, o valor do input será acessado usando-se a propriedade filter. No entanto, nada acontecerá, pois se mantermos o código assim, o Angular entenderá que estamos usando a *string* filter = $event.target.value dentro do evento keyup.

Já aprendemos que existe um tal de *Data binding*, então podemos supor que usar [keyup]="filter = $event.target.value" resolverá nosso problema, o que também não é o caso, já que este *Data binding* buscará a expressão entre aspas no componente, e este atributo não existe.

Para resolvermos isso, precisaremos realizar um *Data binding* que é um ***Event binding***. Isto, no Angular, é feito colocando-se o nome do evento entre parênteses — (keyup)="filter = $event.target.value". Ou seja, para o evento keyup, a expressão "filter = $event.target.value" será avaliada.

Porém, esta associação de eventos se difere ao uso de colchetes, como em um *Data binding* regular, cujo dado vem da fonte de dados (componente) para o template, nunca o caminho inverso. Já quando utilizamos os parênteses, fazemos exatamente o oposto, isto é, o evento é disparado, indo da *view* do template para o componente.

É importante entendermos que estes *bindings* são **unidirecionais**, cada qual percorrendo um caminho diferente. Vamos salvar o projeto, abrir a página no navegador, consultar o console, em que não teremos nenhum erro. E para nos assegurarmos de que o campo de busca captura o termo digitado, incluiremos uma *Expression language* para a propriedade filter.

O código de photo-list.component.html ficará assim:

<div class="text-center mt-3 mb-3">

<form>

<input

class="rounded"

type="search"

placeholder="search..."

autofocus

(keyup)="filter = $event.target.value"

>

</form>

</div>

{{ filter }}

<ap-photos [photos]="photos"></ap-photos>

Ao salvarmos tudo, voltarmos ao navegador e digitarmos "calopsita" (ou qualquer outro termo), à medida em que vamos digitando, a palavra vai se formando no layout antes da exibição das imagens. Deste modo temos certeza de que tudo está sendo adicionado por meio do *binding* de evento, e modificado no filter, atualizado conforme nossa digitação.

Quando abrimos a aplicação, um *Data binding* unidirecional ocorre, de filter para o template, e nada é exibido. Para que isso fique claro, podemos atribuir um valor inicial (flavio) temporariamente em filter: string = 'flavio' de photo-list.component.ts, e reabrir a página, e teremos "flavio" sendo exibido antes das imagens.

Agora, precisamos usar o valor que está no filter para de fato filtrarmos a lista. Faremos isto a seguir!

Precisaremos usar o valor digitado no campo de busca para filtrar a lista de imagens. Antes disso, vamos fazer uma introdução ao que chamamos, no Angular, de ***Pipes***. Em photo-list.component.ts criaremos uma propriedade temporária nome = 'flavio', apenas para efeito de teste, logo acima da parte de constructor.

E em photo-list.component.html, incluiremos antes das tags <ap-photos> um *Data binding* para nome, com {{ nome }}. Salvaremos, voltaremos ao navegador e veremos "flavio" acima das imagens. Caso queiramos este texto em caixa alta, poderíamos alterar o dado, mas sabemos que esta parte de estilização tem a ver com o template, a camada de visualização.

O Angular nos permite incluirmos um *Pipe*. Em photo-list.component.html deixaremos assim:

{{ nome | uppercase }}

O uppercase é uma espécie de transformação pela qual nome passará, por meio de |, que serve de "tubo de passagem". Com esta alteração, após salvarmos e voltarmos ao navegador, o nome passará a ser exibido em caixa alta.

Podemos aplicar *Pipes* em expressões, e existem vários feitos para Angular. No entanto criaremos o nosso, pois queremos aplicar um *Pipe* denominado filterByDescription em photos, o qual precisará levar o filter em consideração, e por isto o incluiremos após o *Pipe*. Em photo-list.component.html:

<ap-photos [photos]="photos | filterByDescription: filter"></ap-photos>

Se tentarmos traduzir o trecho acima, o *Pipe* filterByDescription recebe dois parâmetros, a lista de imagens (photos) e o critério (filter). Se salvarmos o arquivo da maneira em que está e abrirmos o navegador, nada é exibido. No console, encontramos um erro indicando que o filterByDescription não é encontrado. É claro, pois ainda não o implementamos.

Como este filtro está muito atrelado a photos, à própria listagem de imagens, criaremos em "photo-list" o arquivo filter-by-description.pipe.ts. Para que seja um *Pipe*, é necessário utilizarmos um *Decorator*, que será automaticamente importado, e que possui a propriedade name, informando o nome do filtro para esta classe específica, exatamente o mesmo que existe em photo-list.component.html.

A classe FilterByDescription precisa implementar um método transform(), com a lógica de aplicação do *Pipe*, mas em vez de digitarmos o método, implementaremos PipeTransform, com *auto import*. Com isto, clicaremos em FilterByDescription e no ícone de lâmpada para implementarmos a interface.

Teremos:

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'filterByDescription'})

export class FilterByDescription implements PipeTransform {

transform(value: any, ...args: any[]) {

throw new Error('Method not implemented.');

}

}

O primeiro parâmetro do método transform() é sempre aquilo em que queremos aplicar a transformação. No caso, value será a lista de imagens, portanto será substituído por photos, e aproveitaremos para trocar o tipo any para Photo[], o qual importaremos. ...args: any[] é um *array* com todos os parâmetros que forem passados. E já que temos apenas um parâmetro, não o colocaremos como sendo um *array*, e o chamaremos de descriptionQuery, que será do tipo string.

Precisaremos retornar o mesmo tipo de dado de photos: Photo[]. Indicaremos que descriptionQuery receberá o seu valor, o qual chama trim(), para a invalidação da digitação de espaços em branco, e o passaremos a toLowerCase(), para deixar em letras minúsculas e assim conseguirmos comparará-lo com a descrição, pois ambos estarão em caixa baixa.

Então, testaremos se (if) há descriptionQuery, caso positivo, queremos a filtragem, se não, retornaremos o próprio *array* de photos. Usaremos filter(), do JavaScript para solicitar que, para cada imagem, tenhamos sua description. Deixaremos em minúsculo, e verificaremos se o que foi digitado faz parte desta *string*.

import { Pipe, PipeTransform } from '@angular/core';

import { Photo } from '../photo/photo';

@Pipe({ name: 'filterByDescription'})

export class FilterByDescription implements PipeTransform {

transform(photos: Photo[], descriptionQuery: string) {

descriptionQuery = descriptionQuery

.trim()

.toLowerCase();

if(descriptionQuery) {

return photos.filter(photo =>

photo.description.toLowerCase().includes(descriptionQuery)

);

} else {

return photos;

}

}

}

Salvaremos, voltaremos ao navegador e encontramos o mesmo erro indicando que filterByDescription não pode ser encontrado. Um *Pipe* também precisa ser declarado, assim como componentes, portanto, em photos.module.ts, incluiremos FilterByDescription em declarations. Agora, sim, ao salvarmos, tudo estará funcionando bem no navegador, com as imagens sendo filtradas de acordo com o que for digitado no campo de busca.

Agora, queremos exibir uma mensagem para quando o usuário digitar algum termo que não pode ser filtrado por não existir dentre as descrições. Atualmente isto faz com que a tela fique simplesmente em branco. Para exibirmos uma tela mais amigável, voltaremos ao template, photos.component.html, componente que faz a listagem, e colaremos o seguinte parágrafo no início do código:

<p class="text-center text-muted">

Sorry, no photos

</p>

No Bootstrap, text-muted se refere à fonte de corpo um pouco menor, em tom acinzentado. Da maneira em que está, porém, a mensagem é fixa na página, sendo que ela deverá surgir apenas se não houver nenhuma imagem a ser mostrada. Usaremos a diretiva \*ngIf, que espera avaliar uma expressão como verdadeira ou falsa. Já que em photos.component.ts há um *Data binding* para photos, indicaremos que se o comprimento (length) deste for 0, exibiremos a mensagem — sabemos que em JavaScript 0 é falso, enquanto qualquer outro número é considerado verdadeiro.

Salvaremos a alteração abaixo e voltaremos ao navegador.

<p class="text-center text-muted" \*ngIf="!photos.length">

Sorry, no photos

</p>

Desta vez tudo funcionará conforme esperado, e continuamos melhorando a experiência do usuário.

Nosso filtro está funcionando como gostaríamos, porém há um detalhe a respeito da última mensagem que implementamos, "Sorry, no photos", que apenas alguém mais atento conseguirá notar. Ao recarregarmos a página, a mensagem é exibida durante alguns poucos segundos, antes da listagem ser carregada. Isso é meio estranho, não?

Não fizemos uma requisição à API para que os dados das imagens fossem trazidos? Sendo assim, esta mensagem deveria aparecer somente em decorrência do filtro. Isso pode acabar deixando o usuário frustrado, ainda mais se for uma operação que acabar demorando.

Em photo-list.component.ts, a lista de imagens começa com um *array* vazia (photos: Photo[] = []), então, quando o componente é carregado, a lista será considerada no \*ngIf e a mensagem será exibida. Depois, na inicialização ngOnInit(), os dados serão trazidos da API e colocados na propriedade photos, que então terá dados, fazendo com que a mensagem deixe de ser exibida.

O problema é que a busca destes dados está sendo feita no componente, então, para que a mensagem não apareça, o ideal é que o componente receba a lista de imagens pronta antes de navegarmos a ele. Em suma, entraremos na rota e, antes do componente ser criado e renderizado, resolveremos e disponibilizaremos os dados de que ele precisa. Deste modo, o componente receberá os dados prontos, sem precisar buscá-los, e o *array* de imagens será preenchido.

Podemos resolver este tipo de problema com o Angular, por meio do ***Resolver***, capaz de lidar com dados durante a navegação de uma rota para disponibilizá-los a um componente antes deste ser carregado.

Por estar muito atrelado ao "photo-list", criaremos nosso arquivo nesta pasta, e o chamaremos de photo-list.resolver.ts. Precisamos do serviço PhotoService no construtor desta classe para resolvermos esta dependência. E sabemos que, para ser injetável, termos acesso ao PhotoService e podermos utilizar o *Resolver*, teremos um @Injectable().

Para ser um *Resolver*, a classe precisará implementar a interface Resolve, genérica, e que devolverá o tipo de dado retornado pelo listFromUser(), um Observable<Photo[]>. Ambos precisarão ser importados. Para conseguirmos trabalhar com o PhotoListResolver, implementaremos o método resolve(), e o Visual Studio está se recusando a adicioná-lo automaticamente, mas não tem problema, faremos isto manualmente.

Como parâmetro, este método receberá ActivatedRouteSnapshot, que tirará um *screenshot* do que estiver acontecendo no momento, e um state, RouterStateSnapshot. Teremos uma variável const userName, cujo resultado será route.params.userName. Depois, usaremos o return com this.service.listFromUser().

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

import { Resolve } from '@angular/router';

import { Observable } from 'rxjs';

import { PhotoService } from '../photo/photo.service';

import { Photo } from '../photo/photo';

@Injectable({ providedIn: 'root'})

export class PhotoListResolver implements Resolve<Observable<Photo[]>>{

constructor(private service: PhotoService) {}

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

const userName = route.params.userName;

return this.service.listFromUser(userName);

}

}

Assim, se passarmos o mouse sobre resolve(), será indicado que o retorno é do tipo Observable<Photo[]>, o que também pode ser explicitado usando-se dois pontos:

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Photo[]> {

const userName = route.params.userName;

return this.service.listFromUser(userName);

}

Entretanto, ainda precisaremos acessar o arquivo de rotas, app.routing.module.ts. Como o *Resolver* é acionado no momento em que a rota está sendo resolvida, é ali que se encontra sua configuração. Portanto, incluiremos resolve passando um objeto JavaScript para disponibilizar a propriedade photos com o resultado de PhotoListResolver:

path: 'user/:userName',

component: PhotoListComponent,

resolve: {

photos: PhotoListResolver

}

E em photo-list.component.ts, deletaremos o seguinte trecho, que passa a ser desnecessário:

const userName = this.activatedRoute

.snapshot

.params

.userName;

this.photoService

.listFromUser(userName)

.subscribe(photos => this.photos = photos);

Em consequência, removeremos também private photoService: PhotoService em constructor(), e o seu respectivo importe. Em ngOnInit(), incluiremos this.photos = this.activatedRoute.snapshot.data.photos, cujo snapshot.data nos permite acessar o valor contido em photos de app.routing.module.ts.

Também é possível usar this.activatedRoute.snapshot.data['photos'].

Com isso indicamos que é preciso buscar dados em photos, nome da propriedade usada no Routes, para esta rota que foi ativada. Salvaremos tudo, voltaremos ao navegador e acessaremos localhost:4200/user/flavio. As imagens são buscadas, e então recarregaremos a página e, em nenhum momento, a mensagem "Sorry, no photos" é exibida, pois os dados são resolvidos no sistema de rotas.

Esta é a motivação por trás do *Resolver* — a resolução de dados assíncronos dos quais o componente depende antes de ser ativado, no momento em que ativamos a rota, antes mesmo dela avaliar tal componente.

Melhoramos a experiência do usuário, agora vamos fazer o mesmo em relação à performance da aplicação. Como estamos disparando o evento keyup, a cada preenchimento do campo de busca o valor digitado será passado para filter, e deste para o *Pipe*. Então, a cada caractere digitado é feita a aplicação do filtro, e isso prejudica a performance, ainda mais se a quantidade de imagens é muito grande. Pior ainda se estivéssemos realizando requisição AJAX ao servidor.

Seria melhor se, ao digitarmos "farol", fizéssemos uma pausa de 300ms, e aí sim o *Pipe* fosse aplicado. A vantagem disto é que evitamos a execução de uma série de operações. Para tal, usaremos um *Pattern* muito comum no JavaScript, para que o filtro seja atualizado somente se pararmos de digitar durante um determinado período de tempo.

A chave para o sucesso desta operação é o ***Subject*** do RxJS. Em photo-list.component.ts, criaremos inicialmente a propriedade debounce na classe PhotoListComponent, de tipo Subject, por sua vez do tipo string, que receberá um novo Subject, os quais precisarão ser importados.

Por meio de next(), é possível emitirmos um valor para Subject, que acessamos caso o tenhamos inscrito. Diferentemente do *Observable*, com o qual podemos inscrever e obter valores, com o Subject podemos, além disso, emitir um valor e escutá-lo, como em:

export class PhotoListComponent implements OnInit {

photos: Photo[] = [];

filter: string = '';

debounce: Subject<string> = new Subject<string>();

constructor(private activatedRoute: ActivatedRoute) { }

ngOnInit(): void {

this.photos = this.activatedRoute.snapshot.data['photos'];

this.debounce.next('f')

this.debounce.subscribe(value => alert(value));

}

}

Do trecho acima, deletaremos o que se segue:

this.debounce.next('f')

this.debounce.subscribe(value => alert(value));

E em photo-list.component.html, no lugar de colocarmos o valor digitado em filter, solicitaremos ao debounce para que seja feito o next(). A cada keyup será emitido um valor:

<input

class="rounded"

type="search"

placeholder="search..."

autofocus

(keyup)="debounce.next($event.target.value)"

>

Voltaremos a photo-list.component.ts, em ngOnInit() nos inscreveremos em debounce, e o valor a ser emitido ali será chamado de filter, o qual receberemos, e então indicaremos que this.filter receberá filter.

ngOnInit(): void {

this.photos = this.activatedRoute.snapshot.data['photos'];

this.debounce.subscribe(filter => this.filter = filter);

}

Com isso, em vez de jogarmos o valor digitado diretamente em filter, emitiremos um valor de RxJS, a ser escutado pelo subscribe(), o qual atualizará o filtro. O subscribe() será chamado enquanto o valor estiver sendo emitido. Ele é um tanto diferente do HttpClient pois este emite um único valor, e o completa, algo que não ocorre com Subject, por termos criado-o.

Tanto isto é verdade que, se voltarmos ao navegador e recarregarmos a página, o filtro funcionará bem, mesmo sem vantagens, já que estamos tendo mais trabalho, pois pegamos o valor digitado pelo usuário, passando-o para o debounce para que o Subject emita o valor por meio de next(). Tivemos que fazer o subscribe() para então levarmos o valor adiante.

Assim, em photo-list.component.ts importaremos debounceTime de rxjs/operators, junto ao qual uma série de operadores poderá ser importada. A ideia é que, antes do subscribe(), pediremos para o debounce aplicar tal operação, com a estrutura pipe(), em que incluiremos debounceTime, a receber o período de tempo. E desta operação faremos o subscribe().

ngOnInit(): void {

this.photos = this.activatedRoute.snapshot.data['photos'];

this.debounce

.pipe(debounceTime(300))

.subscribe(filter => this.filter = filter);

}

A grande sacada é que, com esta alteração chamada de ***Lettable operators*** no RxJS, por usarmos o debounceTime, quando emitimos um valor no evento keyup, todas as emissões serão ignoradas, sendo consideradas após 300ms. E é isso que será repassado ao subscribe().

O *Observable* é engenhoso para lidar com situações deste tipo, com fluxos e eventos, e colocamos *threshold*, um *Pipe* no que chamamos de debounce, para limitar a quantidade de operações.

Resolvemos o problema de performance, porém ganhamos outro se não tomarmos cuidado: uma vez que o *Observable* em .subscribe(filter => this.filter = filter) nunca se completa, ele ficará guardando um espaço na memória, e se saímos deste componente e vamos a outra página, a área da memória continuará ocupada, ocasionando em *memory leaking* (vazamento de memória).

Então, toda vez que houver algo que fique emitindo valores infinitamente, é necessário implementar uma interface, OnDestroy, que também precisa ser implementada em photo-list.component.ts. Ao fazermos isto, o método ngOnDestroy() é acrescentado. Ele faz parte do ciclo de vida de um componente do Angular, sendo chamado toda vez que um objeto é destruído.

Significa que quando sairmos de PhotoListComponent, e ele for destruído, o método será chamado, e faremos o unsubscribe():

export class PhotoListComponent implements OnInit, OnDestroy {

/\* código omitido

\*/

ngOnInit(): void {

this.photos = this.activatedRoute.snapshot.data['photos'];

this.debounce

.pipe(debounceTime(300))

.subscribe(filter => this.filter = filter);

}

ngOnDestroy(): void {

this.debounce.unsubscribe();

}

}

Trata-se de uma boa prática, portanto não deixemos de fazê-lo.

Vamos continuar melhorando a performance da aplicação? Atualmente, a integração que temos com o back end retorna todas as imagens existentes, que consistem em cerca de 15. Se tivéssemos 1000, lidaríamos com uma quantidade exorbitante de dados, o que não é bom para o back end, pois ele teria que fazer uma consulta, trazê-los, convertê-los para JSON, e enviá-los de volta. Isto também prejudica o front end, pela quantidade enorme de dados armazenados na memória do navegador, um ambiente hostil, pois nunca se sabe onde a aplicação estará rodando.

Uma das formas de resolvermos este problema do carregamento excessivo de dados é a **paginação**. Carregaremos as primeiras 12 imagens, e incluiremos ao fim da página um botão ou link para seguirmos à próxima. Quando o usuário clicar nele, serão carregadas as 12 imagens seguintes, e por aí vai. Para conseguirmos isso, porém, tanto o back end quanto o front end precisam nos ajudar.

Assim, em photo.service.ts, em que temos o listFromUser(), criaremos o método listFromUserPaginated(), com o primeiro parâmetro sendo userName de tipo string, e o segundo, page, de tipo number. Deste modo, ao escrevermos localhost:3000/user/flavio/photos na barra de navegação, e depois localhost:3000/user/flavio/photos?page=2, a API estará preparada para trazer apenas as imagens da segunda página, algo definido pelo back end.

listFromUserPaginated(userName: string, page: number) {

return this.http

.get<Photo[]>(API + '/' + userName + '/photos');

}

Passaremos um QueryParam na URL, com new HttpParams(), módulo do HttpClient, encadeando uma chamada com append() de page, com valor page, a página que recebemos como parâmetro. No entanto, o append() espera receber uma *string*, e estamos passando um number, então é necessário incluir toString().

Agora que temos o params, o get receberá um segundo parâmetro do HttpClient, um objeto JavaScript que espera receber a propriedade params. E são estes parâmetros que serão colocados na QueryString.

listFromUserPaginated(userName: string, page: number) {

const params = new HttpParams()

.append('page', page.toString());esta `page`

return this.http

.get<Photo[]>(API + '/' + userName + '/photos', { params: params });

}

Entretanto, sabemos que se uma chave possui o nome de alguma variável, no JavaScript, podemos omitir os dois pontos e manter apenas um params:

.get<Photo[]>(API + '/' + userName + '/photos', { params });

Feito o método de paginação, vamos atualizá-lo indo a photo-list.resolver.ts. Por padrão, o resolve() sempre pegará as 12 primeiras imagens, então passamos 1:

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Photo[]> {

const userName = route.params.userName;

return this.service.listFromUserPaginated(userName, 1);

}

Salvaremos, acessaremos o navegador, e teremos apenas as 12 imagens iniciais, como desejado. Precisamos agora criar o botão de "Load More" ("Carregar mais") para passarmos à página seguinte.

Criaremos o botão "Load more" como sendo um componente em "photo-list", portanto voltaremos ao terminal e pausaremos o Angular CLI. Usaremos o atalho ng g c photos/photo-list/load-button, em que c se refere a *component* e g a *generate* para gerarmos o "load-button".

A primeira propriedade que ele terá é hasMore, de tipo boolean, que começará com false. Isto nos ajudará a saber se há mais elementos a serem exibidos, ou não. Em load-button.component.ts, teremos:

export class LoadButtonComponent implements OnInit {

hasMore: boolean = false;

constructor() { }

ngOnInit() {

}

}

Apagaremos o conteúdo de load-button.component.html e incluiremos o seguinte código:

<div class="text-center">

<button class="btn btn-primary">Load more</button>

</div>

<p class="text-center text-muted">No more data to load</p>

Com isso, teremos uma <div> centralizada, que exibirá nosso botão, e usamos a classe btn do Bootstrap, bem como btn-primary, para que ele tenha destaque, e os textos "Load more" e "No more data to load". Quem for utilizar o componente em load-button.component.ts receberá o hasMore, um booleano. Em selector, não podemos esquecer de remover um "p" de app-load-button, mantendo simplesmente ap-load-button.

Se queremos passar um parâmetro via forma declarativa do componente, precisaremos utilizar @Input(), que importaremos. E em photo-list.component.html, já que incluiremos o botão após a listagem de imagens, adicionaremos:

<ap-load-button></ap-load-button>

Salvaremos, subiremos a aplicação no terminal e verificaremos o resultado no navegador, um botão azul de cantos arredondados contendo o texto "Load more" centralizado, junto a "No more data to load" logo abaixo, sendo que isto deveria ser condicional — se há dados a serem exibidos, o botão deve aparecer. Caso contrário, apenas o texto de baixo.

Para lidarmos com esta situação, incluiremos na <div> de load-button.component.html a diretiva \*ngIf, que já conhecemos, e a condição será hasMore. Significa que, se hasMore ("há mais"), que começa com valor falso, será exibido o botão. Se não, o texto.

O Angular permite uma estrutura *if/else* no template, e aquele que estiver no else precisará estar dentro da tag <ng-template>. Assim, adicionaremos nele uma **variável de template** capaz de identificá-lo, podendo ter um nome qualquer. Neste caso, será utilizado messageTemplate.

Poderíamos colocar esta variável no botão, mas optaremos por fazê-lo no template.

<div class="text-center" \*ngIf="hasMore; else messageTemplate">

<button class="btn btn-primary">Load more</button>

</div>

<ng-template #messageTemplate>

<p class="text-center text-muted">No more data to load</p>

</ng-template>

Salvaremos e retornaremos ao navegador, que exibe "No more data to load" ao final da página, pois o valor padrão de hasMore é false, então, o trocaremos por true e testaremos novamente. Desta vez, o texto será substituído pelo botão.

Em photo-list.component.html será necessário realizar *Data binding* em uma propriedade, para que seja indicado se há mais ou não. Em photo-list.component.ts, também incluiremos a propriedade hasMore, que começará recebendo false:

photos: Photo[] = [];

filter: string = '';

debounce: Subject<string> = new Subject<string>();

hasMore: boolean = false;

E então, faremos em photo-list.component.html uma associação de hasMore com a expressão homônima:

<ap-load-button [hasMore]="hasMore"></ap-load-button>

Ou seja, a *inbound property* do botão fará um *Data binding* com o hasMore de photo-list.component.ts. Cada vez que clicarmos no botão, um *load* ("carregamento") será realizado, e então verificaremos se os dados chegaram. Caso positivo, significa que hasMore se mantém true. Caso contrário, e se tivermos um *array* vazio, é porque fomos além de sua capacidade.

Portanto, precisaremos do photoService de volta, para que seja acessado a cada clique do botão. Além disso, criaremos a propriedade currentPage de tipo number, que começará com 1, e userName de tipo string, pois precisaremos saber disso toda vez que os dados forem lidos.

hasMore: boolean = false;

currentPage: number = 1;

userName: string = '';

constructor(

private activatedRoute: ActivatedRoute,

private photoService: PhotoService

) { }

Agora, no ato de pegarmos os dados, indicaremos que this.userName recebe this.activatedRoute.snapshot.params.userName, e em breve entenderemos o porquê de guardarmos tantos valores. Também criaremos o método load(), que chamará this.photoService.listFromUserPaginated. Usaremos ++ pois, quando carregamos a página pela primeira vez, o *Resolver* já carregou a primeira página, então quando damos *load* é porque queremos carregar a segunda. Trata-se de um pré-incremento, e em seu subscribe() receberemos photos.

Também em subscribe() fazemos uma **desestruturação**, ou ***Destructuring***, na verdade, um ***Spread Operator***, para informar que photos será desmembrado como se passássemos os dados de cada item retornado do back end. Testaremos se photos.length é diferente de this.hasMore e, neste caso, seu valor será false.

ngOnInit(): void {

this.userName = this.activatedRoute.snapshot.params.userName;

this.photos = this.activatedRoute.snapshot.data['photos'];

this.debounce

.pipe(debounceTime(300))

.subscribe(filter => this.filter = filter);

}

ngOnDestroy(): void {

this.debounce.unsubscribe();

}

load() {

this.photoService

.listFromUserPaginated(this.userName, ++this.currentPage)

.subscribe(photos => {

this.photos.push(...photos);

if(!photos.length) this.hasMore = false;

});

}

Feito isso, vamos substituir false de hasMore: boolean = false para true. Ao carregarmos a página, o botão aparecerá mas, quando clicado, o método load() será chamado, o que alterará o estado do componente, podendo ou não alterar o hasMore. Precisaremos acessar photo-list.component.html e chamar este método por meio de um *binding* para o evento click, usando parênteses.

<ap-load-button

(click)="load()"

[hasMore]="hasMore">

</ap-load-button>

Salvaremos tudo e voltaremos ao navegador para testarmos. Quando clicamos no botão, a mensagem "No more data to load" é exibida, e aparentemente funciona bem. Na tag <ap-photos> de photo-list.component.html realizamos um *Data binding* de photos para a propriedade photos de photo-list.component.ts, da classe PhotoListComponent.

O problema é que quando fazemos o push(), a cada item incluso ali dentro, o outro componente não sabe que photos é alterado. Isto porque não modificamos a referência this.photos. Estamos mudando os itens da lista de imagens, entretanto o photos.component.ts não sabe que photos mudou. Isso só aconteceria se fosse atribuído outro valor nesta propriedade.

Para resolvermos isso, em load() de photo-list.component.ts, usaremos this.photos = this.photos.concat(photos) em vez de this.photos.push(...photos). O concat() acessará a lista de imagens existente (this.photos) e a concatenará com a lista photos, gerando outra, que será uma nova referência. Com isto, o mecanismo de *Change detection* do Angular funcionará.

Se salvarmos e voltarmos ao navegador, ao clicarmos no botão, o restante das imagens da lista será apresentado. E se clicarmos mais uma vez, o texto "No more data to load" aparecerá.

Entendemos que precisamos nos atentar a estes casos de *Data binding*, em que o Angular só detecta alteração em uma *inbound property* quando há uma nova atribuição nela.

Todos os componentes contidos na pasta "photos", e nas suas subpastas, como "photo-form" e "photo-list", fazem parte de photos.module.ts. No entanto, eventualmente começaremos a ter uma quantidade muito grande de artefatos em declarations deste arquivo, que precisarão ser importados à medida em que formos criando componentes que dizem respeito a imagens.

Por exemplo, sabemos que LoadButtonComponent, PhotoListComponent e PhotosComponent fazem parte do mesmo conjunto. Neste sentido, criaremos **submódulos** para deixar o nosso código ainda mais organizado. Manualmente, criaremos o módulo photo-list.module.ts em "photo-list", em que teremos a seguinte estrutura:

import { NgModule } from '@angular/core';

import { PhotoListComponent } from './photo-list.component';

import { PhotosComponent } from './photos/photos.component';

import { LoadButtonComponent } from './load-button/load-button.component';

@NgModule({

declarations: [

PhotoListComponent,

PhotosComponent,

LoadButtonComponent

]

})

export class PhotoListModule {}

Em "photo-form" criaremos o módulo photo-form.module.ts:

import { NgModule } from '@angular/core';

import { PhotoFormComponent } from './photo-form.component';

@NgModule({

declarations: [PhotoFormComponent]

})

export class PhotoFormModule { }

Da mesma forma, em "photo" criaremos photo.module.ts:

import { NgModule } from '@angular/core';

import { PhotoComponent } from './photo.component';

@NgModule({

declarations: [PhotoComponent]

})

export class PhotoModule { }

E em photo-list.module.ts acrescentaremos FilterByDescription em declarations. Em seguida, abriremos photos.module.ts, removeremos todo o conteúdo de declarations, e alteraremos imports. Desta forma, ele funcionará como um agregador dos elementos importados para que outro módulo também possa importá-los.

@NgModule({

imports: [

PhotoModule,

PhotoFormModule,

PhotoListModule,

HttpClientModule,

CommonModule

]

})

Se voltarmos ao navegador, o console acusará um erro em PhotosComponent, pois as diretivas \*ngFor e \*ngIf estão sendo utilizadas em photos.component.html, porém seu módulo (photo-list.module.ts) precisará fazer a importação de CommonModule:

@NgModule({

declarations: [

PhotoListComponent,

PhotosComponent,

LoadButtonComponent,

FilterByDescription

],

imports: [ CommonModule ]

})

Sabemos que é uma boa prática importar CommonModule em todos os módulos, então o importaremos também em photo-form.module.ts. Por fim, o removeremos dos importes em photos.module.ts, juntamente com HttpClientModule, pois seu único dependente até o momento é "photo". Iremos a photo.module.ts e o acrescentaremos dentre os importes:

@NgModule({

declarations: [PhotoComponent],

imports: [

CommonModule,

HttpClientModule

]

})

Esse trabalho todo é necessário porque, agora, ao lidarmos com os módulos e submódulos, sabemos com clareza do que cada módulo precisa para funcionar. Vamos testar no navegador? Ao carregarmos, teremos outro problema — photo-list.component.html usa <ap-photos>, enquanto photos.component.html usa <ap-photo>, que está indisponível por não ter sido carregado em seu módulo.

Então, em photo-list.module.ts importaremos CommonModule, sabendo que um dos componentes que temos dependerá de PhotoComponent, portanto precisamos importar PhotoModule também. Salvaremos as alterações, voltaremos ao navegador e, ao rodarmos, continuamos tendo problemas. Isso porque o módulo que estamos importando não exporta PhotoComponent.

Abriremos photo.module.ts e incluiremos em @NgModule:

exports: [ PhotoComponent ]

Será que isso é o suficiente? No navegador, tudo parece funcionar bem, e o console já não acusa nenhum erro. Esta parte de organização é uma revisão do que aprendemos sobre módulos, além de melhorarmos a compreensão do funcionamento do projeto como um todo. Entendemos que cada componente criado em "photos" muito provavelmente será um módulo, e não é ruim fazermos isso, mesmo que tenhamos apenas um deles.

Outra curiosidade é que photo-form.module.ts não possui nada em declarations. Mas não usamos PhotoFormComponent no arquivo de rotas? Não precisamos exportá-lo? Não, pois para declarations, só precisaríamos incluir PhotoFormComponent em exports caso ele fosse utilizado no template de outro componente. E photo-list.module.ts não exporta nada, e por ora não há nenhum outro local da aplicação que esteja interessado em usar qualquer um dos elementos em declarations isoladamente.

O TypeScript não remove automaticamente os importes inutilizados, portanto precisamos lembrar de fazê-lo manualmente.

Vamos incluir um ícone de lupa ao lado do campo de busca na nossa aplicação. Até a versão 3 do Bootstrap, o **Glyphicons**, um conjunto de fontes disponíveis para uso, já vinha embutido. A partir da versão 4 do Bootstrap, não houve indicação sobre qual pacote de fontes poderia ser utilizado, sendo necessária sua instalação manual.

Contudo optaremos pelo [Font Awesome](https://fontawesome.com/), que possui as versões paga e gratuita. A gratuita supre muito bem nossas necessidades, portanto pausaremos o Angular CLI via terminal, e com npm install font-awesome@4.7.0 instalaremos a versão 4.7.0. Já que usamos a versão 8 (ou superior) do Node, não precisamos incluir -s no comando, mas é possível fazê-lo para que ele seja salvo no pacote JSON também.

Confirmaremos se a instalação foi bem sucedida abrindo package.json, com todas as dependências do projeto, e buscando por "font-awesome" na listagem. Ele tem um CSS que precisa ser carregado para que as fontes sejam de fato disponibilizadas para a aplicação. Como dito anteriormente, CSSs globais como o Bootstrap, que são carregados e disponíveis para todos os componentes da página, precisam ser registrados em angular.json.

Neste arquivo, temos o caminho do Bootstrap, "./node\_modules/bootstrap/dist/css/bootstrap.min.css", em styles. Precisaremos do caminho referente ao Font Awesome: "./node\_modules/font-awesome/css/font-awesome.css".

"./node\_modules/bootstrap/dist/css/bootstrap.min.css" pode ficar sem o min, pois no processo de build do Angular, ao ser gerado para produção, ele será minificado de qualquer forma.

Salvaremos, e não podemos esquecer que ao alterarmos angular.json temos que reiniciar o Angular CLI pelo terminal. Mas como ele já está parado, não há problema algum. Vamos subí-lo novamente com ng serve --open e verificar se há algum erro no console do navegador.

Quando trabalhamos com Angular CLI, é preferível trabalhar com uma máquina com HD SSD, para facilitar o build.

Abriremos photo-list.component.html de "photos", usaremos a tag <i> e a classe fa fa-search mr-1, proveniente da documentação do Font Awesome, em que fa remete a "Font Awesome" e search, ao nome do ícone. Além disso, podemos acrescentar aria-hidden="true", pois é boa prática indicar que leitores de tela ignorem ícones, também recomendado na documentação do próprio Font Awesome.

<i aria-hidden="true" class="fa fa-search mr-1"></i>

Vamos continuar mexendo na estética da nossa página diminuindo o espaçamento entre as imagens, pois quando diminuímos a tela, o espaço fica ainda maior, e nada harmonioso. Para resolvermos isto, acessaremos photos.component.html e, ao lado de row incluiremos no-gutters ("sem espaçamento"), deixando a linha assim:

<li \*ngFor="let cols of rows" class="row no-gutters">

Isso torna a página ainda mais agradável. Vamos continuar?

Assim como várias redes sociais possuem compartilhamento e exibição de fotos, seria interessante se houvesse um número de curtidas para cada uma delas. A boa notícia é que, se checarmos a interface que define a forma da API, na qual acessamos os dados retornados por ela, teremos likes e comments.

Eles se encontram vazios, mas se tivéssemos interações na nossa aplicação, estariam disponíveis. Queremos que tais informações sejam exibidas logo após as imagens. Como faremos isso?

O componente responsável pela renderização das imagens é photos.component.ts e, em seu template, colaremos as seguintes linhas logo após a *tag* de fechamento de <ap-photo>:

<i aria-hidden="true" class="fa fa-heart-o fa-1x mr-2"></i>{{ photo.likes }}

<i aria-hidden="true" class="fa fa-comment-o fa-1x mr-2 ml-2"></i>{{ photo.comments }}

Com isso, acrescentamos dois ícones de Font Awesome, um de coração e outro de balão de fala (para comentário), ambos com tamanho 1x. Também fazemos o *Data binding* por meio de interpolação, para photos.likes e photo.comments. Se salvarmos e voltarmos ao navegador, eles aparecem abaixo de cada imagem, como gostaríamos.

Mas isso pode ficar ainda melhor se incluirmos as linhas entre <div>s centralizadas:

<div class="text-center">

<i aria-hidden="true" class="fa fa-heart-o fa-1x mr-2"></i>{{ photo.likes }}

<i aria-hidden="true" class="fa fa-comment-o fa-1x mr-2 ml-2"></i>{{ photo.comments }}

</div>

Além disso, podemos melhorar a exibição de cada imagem e seus respectivos botões colocando-os em um *card*.

No Bootstrap 3, era preciso utilizar *panels*, mas a partir da versão posterior eles se tornaram *cards*.

A estrutura de um *card* com borda suave e conteúdo centralizado é a seguinte:

<div class="card border-light text-center">

<h4 class="card-header">TITULO DO CARD</h4>

<div class="card-block text-justify">

<!-- aqui entra o conteúdo -->

</div>

</div:

Um *card* pode ter cabeçalho com título, e um corpo com texto justificado. Iremos **componentizar** este *card* para podermos utilizá-lo em qualquer módulo, independentemente do projeto. Não podemos criá-lo dentro de "photos", pois assim o atrelaremos ao universo de imagens, e não é este o caso.

No entanto, *cards* são compartilháveis, podendo ser utilizados em outros locais e com outros módulos. Sendo assim, criaremos em "app" a pasta "shared", e nela, a pasta "components", dentro da qual teremos subpastas de todos os componentes compartilháveis pela aplicação. No caso, criaremos "card", que conterá card.component.ts e card.component.html.

No primeiro arquivo, teremos:

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

@Component({

selector: 'ap-card',

templateUrl: './card.component.html'

})

export class CardComponent {

@Input() title: string = '';

}

E em seu template:

<div class="card border-light text-center">

<h4 class="card-header">{{ title }}</h4>

<div class="card-block text-justify">

</div>

</div>

Nele, o title é pego da *inbound property* de card.component.ts e, na <div> com card-block, deve-se pegar o conteúdo do *card* em si. Em seguida, também criaremos dentro de "card" o módulo card.module.ts, pois se este *card* será compartilhado em diversos locais, ele deverá ser importado. Porém, **um componente não pode fazer parte da declaração de mais de um módulo**.

Isso significa que, se "errors" e "photos" incluírem o *card* em declarations, teremos problemas, e o componente não poderá ser exportado ali por não ser um módulo. Então, mesmo que tenhamos um único componente, o transformaremos em módulo para conseguirmos importá-lo em todos os locais que precisarem utilizá-lo.

Assim, em card.module.ts, digitaremos e salvaremos o seguinte:

import { NgModule } from '@angular/core';

import { CardComponent } from './card.component';

@NgModule({

declarations: [CardComponent],

exports: [CardComponent]

})

export class CardModule { }

Como queremos utilizar este *card* em "photos > photos > photo-list.module.ts", teremos que acessar photos.module.ts e importar CardModule:

imports: [

CommonModule,

PhotoModule,

CardModule

]

E por falar nisso, já vimos que todo módulo criado precisa importar CommonModule para que as diretivas do Angular funcionem corretamente. Em card.module.ts, então, teremos:

@NgModule({

declarations: [CardComponent],

exports: [CardComponent],

imports: [CommonModule]

})

Recapitulando: o CardModule depende de CommonModule e exporta CardComponent. E em "photos > photo-list > photo-list.module.ts" indicamos que precisamos de CardModule, pois este componente exportado pelo *card* será utilizado por PhotosComponent.

Vamos acessar photos.component.html, incluir <ap-card> e mover as tags <ap-photo> e <div> para dentro dela. Por ora, nosso *card* não terá título, pois estamos utilizando-o por motivos de organização. Então, dentro do *card* exibimos a imagem, e incluiremos um *padding* de valor 1 para todos os lados.

<div \*ngFor="let photo of cols" class="col-4">

<ap-card>

<ap-photo

[url]="photo.url"

[description]="photo.description">

</ap-photo>

<div class="text-center p-1">

<i aria-hidden="true" class="fa fa-heart-o fa-1x mr-2"></i>{{ photo.likes }}

<i aria-hidden="true" class="fa fa-comment-o fa-1x mr-2 ml-2"></i>{{ photo.comments }}

</div>

</ap-card>

</div>

Ao salvarmos e voltarmos ao navegador, teremos uma surpresa: nada é exibido. Podemos adicionar title="Flavio" em <ap-card>, e o texto será exibido em todos os *cards*, porém não seu conteúdo. Isto porque em nenhum momento indicamos que no componente criado em card.component.html deve estar o conteúdo que for colocado em <ap-card>. Para isso, utilizamos a diretiva ng-content:

<div class="card-block text-justify">

<ng-content></ng-content>

</div>

Isso resolverá nosso problema e, se quisermos, podemos voltar à listagem em photos.component.html e incluir title="photo.description" com *Data binding* (caso contrário a *string* é que aparecerá): <ap-card [title]="photo.description">. Salvaremos as alterações, e teremos as imagens sendo exibidas com sucesso, cada qual com o título correspondente. Podemos deletar o título, que incluímos apenas para fins de teste.

Conseguimos criar nosso primeiro componente que aceita receber conteúdo entre abertura e fechamento de tags.

O template de photo-list.component.html possui a primeira <div> correspondendo ao topo da página com o campo de busca, e na página temos <ap-load-button>, <ap-photos>, e a parte de *search* (busca), certo?

Com o Angular, a ideia é que tenhamos o mínimo possível de HTML para trabalhar. Sendo assim, esta primeira <div> é um forte candidato a se tornar um componente. É o que faremos — componentizaremos o máximo que pudermos para melhorar a legibilidade e manutenção do nosso código.

Já que esta área de busca será exclusiva a "photo-list", criaremos nela uma pasta denominada "search", com o componente search.component.ts e o template search.component.html. O primeiro terá o seguinte conteúdo:

@Component({

selector: 'ap-search',

templateUrl: './search.component.html'

})

export class SearchComponent { }

Recortaremos de photo-list.component.html a <div> referente à caixa de busca, e a colaremos em search.component.html, salvando-o em seguida:

<div class="">

<form>

<i aria-hidden="true" class="fa fa-search mr-1"></i>

<input

class="rounded"

type="search"

placeholder="search..."

autofocus

(keyup)="debounce.next($event.target.value)"

>

</form>

</div>

Em photo-list.component.ts há o debounce para lidar com a questão da digitação do *input*, não mais necessário, portanto moveremos a linha debounce: Subject<string> = new Subject<string>(); e deletaremos import { Subject } from 'rxjs';, e deixaremos SearchComponent de search.component.ts do seguinte modo:

export class SearchComponent {

debounce: Subject<string> = new Subject<string>();

}

Em seguida, removeremos o trecho a seguir de ngOnInit() do photo-list.component.ts:

this.debounce

.pipe(debounceTime(300))

.subscribe(filter => this.filter = filter);

E faremos o mesmo com

ngOnDestroy(): void {

this.debounce.unsubscribe();

}

Com isso, simplificamos o componente. Poderemos deletar também os importes de debounceTime e OnDestroy, e a exportação deste. E precisaremos adicionar o debounce no ngOnInit() de search.component.ts, não esquecendo de importá-lo. Incluiremos implements OnInit na clase SearchComponent, e removeremos as demais linhas de ngOnInit().

Então, implementaremos OnDestroy, e em ngOnDestroy() incluiremos o unsubscribe de debounce. O arquivo ficará assim:

import { Component, OnInit, OnDestroy } from '@angular/core';

import { Subject } from 'rxjs';

import { debounceTime } from 'rxjs/operators';

@Component({

selector: 'ap-search',

templateUrl: './search.component.html'

})

export class SearchComponent implements OnInit, OnDestroy {

debounce: Subject<string> = new Subject<string>();

ngOnInit(): void {

this.debounce

.pipe(debounceTime(300));

}

ngOnDestroy(): void {

this.debounce.unsubscribe();

}

}

Salvaremos, abriremos photo-list.component.html e incluiremos, no topo do código, de modo a termos três componentes:

<ap-search></ap-search>

Salvaremos tudo, abriremos o navegador e obteremos um erro que aponta que faltou acrescentarmos SearchComponent no NgModule de photo-list.module.ts. Além disso, ao testarmos o campo de pesquisa, notaremos que o filtro não funciona. Temos um componente isolado que precisa se comunicar com outro. Isto é, para interagirmos com o <ap-search> de photo-list.component.html, ele precisa se comunicar com photo-list.component.ts, pois ele precisa atualizar a propriedade filter para que o filtro de fato funcione.

Como faremos essa comunicação entre componentes hierarquicamente organizados?

É importante entendermos que, à medida em que digitamos no campo de busca no navegador, o evento keyup está sendo disparado. Da mesma forma, o debounce é executado, porém não há nada inscrito nele. Antes, realizávamos o subscribe() passando o termo buscado à propriedade filter de PhotoListComponent, para filtrarmos a lista.

Entretanto, desta vez estamos em um componente, e o valor filter precisa chegar a outro. O ngOnInit() de search.component.ts ficará assim:

ngOnInit(): void {

this.debounce

.pipe(debounceTime(300))

.subscribe(filter => console.log(filter));

}

No navegador, conforme vamos digitando o termo que queremos buscar e consultamos o console, confirmaremos que os dados estão sendo armazenados corretamente. Precisamos fazer com que o valor de console, neste componente específico, de alguma forma caia em filter de PhotoListComponent.

Já aprendemos a trabalhar com a *inbound property*, então, na tag <ap-search> de photo-list.component.html, teremos um evento customizado ("*custom event*") chamado onTyping. Trata-se de um evento do componente que nós definiremos da maneira que preferirmos. Sabemos que para fazermos um *Data binding* de eventos é necessário colocarmos a propriedade entre parênteses, então indicaremos que neste <ap-search> que irá disparar o evento onTyping, filter receberá $event.

Com isso, informamos que, conforme vamos digitando no campo de busca, o componente irá disparar este evento cujo $event será o *input* digitado, o qual será levado a filter de PhotoListComponent. Para criarmos um evento customizado, em search.component.ts adicionaremos um @Output() chamado onTyping, mesmo nome que colocamos no evento. Além disso, precisamos indicar que @Output() é um new EventEmitter, tomando cuidado para que isto não seja importado do pacote errado — neste caso, queremos que seja do próprio Angular.

Podemos definir um tipo genérico em EventEmitter(), e como usaremos um valor que é um texto, o evento emitirá um valor de tipo string.

export class SearchComponent implements OnInit, OnDestroy {

@Output() onTyping = new EventEmitter<string>();

debounce: Subject<string> = new Subject<string>();

// código omitido

}

O TypeScript automaticamente infere o tipo, porém é possível enfatizar isso usando dois pontos seguido de EventEmitter<string>, o que não faremos neste caso.

Salvaremos as alterações e, no subscribe() de ngOnInit(), trocaremos console.log(filter) por this.onTyping.emit(filter), pois toda vez que passarmos pelo debounce o filter será gerado:

ngOnInit(): void {

this.debounce

.pipe(debounceTime(300))

.subscribe(filter => this.onTyping.emit(filter));

}

Com isso, a instrução filter => this.onTyping.emit(filter) só será pausada quando atingirmos 300ms sem digitarmos nada no campo de busca. Daí, será pego o valor digitado (filter), e o evento onTyping será emitido com o mesmo valor. Em photo-list.component.html, o $event (que nós programamos) é o valor que passamos pelo filter. Vamos salvar tudo e conferir seu funcionamento no navegador.

Desta vez, o filtro funciona corretamente! Aprendemos a lidar com a comunicação entre um elemento filho e um elemento pai. E podemos trabalhar com variações. Porém, há algo que pode confundir o usuário: após a busca e filtragem de imagens que correspondem a "farol", por exemplo, o clique no botão "Load more" não acarretará em nada.

Isso porque o filtro está em ação, e o usuário não saberá se as imagens estão sendo carregadas com sucesso, ou não. Podemos convencionar que, ao clique de "Load more", o filtro deverá ser limpo e as imagens serão recarregadas. Sendo assim, voltaremos a photo-list.component.ts e, em load(), com a chegada de novos dados, indicaremos que filter recebe uma *string* em branco:

load() {

this.photoService

.listFromUserPaginated(this.userName, ++this.currentPage)

.subscribe(photos => {

this.filter = '';

this.photos = this.photos.concat(photos);

if(!photos.length) this.hasMore = false;

});

}

E então retornaremos ao navegador e repetiremos a ação de digitarmos "farol" e clicarmos em "Load more". Isto fará com que as imagens sejam recarregadas e o filtro seja limpado, porém o componente de busca não está ciente disso, portanto não é atualizado. Uma das formas de resolvermos isso é incluir @Input() com valor de tipo string:

@Output() onTyping = new EventEmitter<string>();

@Input() value: string = '';

debounce: Subject<string> = new Subject<string>();

Em photo-list.component.html, faremos o *Data binding* com filter para que ele seja limpo:

<ap-search

(onTyping)="filter = $event" [value]="filter">

</ap-search>

E em search.component.html, incluiremos a linha [value]="value" dentro da tag <input>. Salvaremos tudo, recarregaremos a página no navegador. Atingimos nosso objetivo, e esta é uma das maneiras de conseguirmos lidar com a comunicação entre componentes que possuem alguma hierarquia entre si.

Pensando em outra melhoria, ao passarmos o mouse sobre cada *card*, podemos fazer com que ele ganhe um tom mais escuro para dar destaque. Poderíamos utilizar a *pseudo class* hover do CSS, não é? Mas e se quisermos aplicar o mesmo efeito na lupa, ou no *placeholder* "search" no campo de busca, ou em qualquer outro componente existente em nossa aplicação?

Teríamos que repetir o CSS, mas a ideia é pegarmos o comportamento desejado, o qual não pertence a nenhum componente em específico, e isolá-lo em um único local. Para isso, criaremos nossa primeira **diretiva**, a ***Custom Directive***.

Todo componente é uma diretiva. A diferença é que @Component, por exemplo, está atrelado a um template, o que não ocorre com a diretiva.

Então, criaremos uma pasta em "shared" chamada "directives", em que colocaremos todas as diretivas. Nela, também criaremos uma pasta denominada "darken-on-hover", dentro da qual criaremos a primeira diretiva. Se tivermos outra diretiva, criaremos outra pasta, e assim por diante, isso porque ainda não estamos trabalhando com arquivos de testes, ou similares.

Criaremos o módulo desta diretiva, darken-on-hover.module.ts, em "darken-on-hover", no qual também criaremos darken-on-hover.directive.ts — isso não é muito diferente do que fizemos até então.

No arquivo da diretiva, teremos:

import { Directive } from '@angular/core';

@Directive({

selector: ''

})

export class DarkenOnHoverDirective { }

Neste caso, a diretiva será utilizada da seguinte forma: se tivermos um componente <a>, por exemplo, e queremos aplicar o efeito de escurecimento ao passarmos o mouse sobre ele, poderemos digitar algo como <a apDarkOnHover></a>. Isto é, usaremos a diretiva como se fosse um atributo, diretamente na tag ou componente. No entanto, para tal, teremos que colocar seu selector entre colchetes:

@Directive({

selector: '[apDarkenOnHover]'

})

Enquanto isso, o conteúdo de darken-on-hover.module.ts será:

import { NgModule } from '@angular/core';

import { DarkenOnHoverDirective } from './darken-on-hover.directive';

@NgModule({

declarations: [ DarkenOnHoverDirective ],

exports: [ DarkenOnHoverDirective ]

})

export class DarkenOnHoverModule { }

Em "photo-list > photos > photos.component.html" incluiremos a diretiva apDarkenOnHover na <div>, que é onde queremos aplicar este efeito. Veremos mais detalhes sobre esta implementação a seguir.

Por utilizarmos a diretiva como sendo um atributo de um elemento desejado, precisaremos saber se, quando ela estiver "pendurada" em um elemento como a <div>, teremos acesso ao elemento do DOM no qual ela foi associada. Já entendemos o conceito de injeção de dependências, portanto em darken-on-hover.directive.ts incluiremos constructor(), cujo el se refere a *element*, de tipo ElementRef.

Com isso, o Angular entenderá que deverá acessar o elemento do DOM no qual a nossa diretiva está adicionada, e será dada uma referência para que possamos manipulá-la.

No entanto, o ElementRef toca superficialmente neste elemento do DOM, agindo como uma última camada de proteção antes de atingirmos o elemento nativo. Criaremos os métodos darkenOn() e darkenOff(), e ambos exibirão suas respectivas mensagens para indicar que foram chamados. Mas de que forma conseguiremos chamar estes métodos por meio da diretiva?

Sabemos que em JavaScript há os eventos mouseover e mouseleave, e é aí que entra o *decorator* @HostListener, do pacote core. Passaremos a ele o evento do elemento hospedeiro ao qual queremos responder, no qual a diretiva está sendo colocada.

import { Directive, ElementRef } from '@angular/core';

@Directive({

selector: '[apDarkenOnHover]'

})

export class DarkenOnHoverDirective {

constructor(private el: ElementRef) {}

@HostListener('mouseover')

darkenOn() {

console.log('darkenOn');

}

@HostListener('mouseleave')

darkenOff() {

console.log('darkenoff');

}

}

Então, o darkenOn() será chamado quando passarmos o mouse por cima do elemento no qual a diretiva está, e o darkenOff(), quando tirarmos o mouse dele. Entretanto, precisaremos acrescentar o módulo DarkenOnHoverModule na lista de importes de photo-list.module.ts, pois queremos disponibilizar a diretiva em PhotosComponent, que faz parte de PhotoListModule.

Vamos voltar ao navegador para verificar se há algum erro ao fazermos testes. No console, veremos as mensagens anteriormente configuradas, confirmando que o código está sendo executado como esperado. Então, na chamada dos métodos da diretiva, manipularemos os elementos do DOM. Se incluirmos this.el.nativeElement em darkenOn(), teremos acesso ao elemento nativo do DOM no qual a diretiva foi adicionada.

Daí, basta manipularmos o DOM como já sabemos, mas o mais importante é que não iremos fazê-lo diretamente, e sim por meio de uma ferramenta do Angular chamada ***Render***, um módulo que nos permite manipular o DOM sem termos que digitar o que queremos fazer.

Quando fazemos uma renderização do lado do servidor, não há DOM. Portanto, se formos utilizar o *Render* para manipulação do DOM, garantimos que o código não dará problemas durante a renderização de back end, e assim por diante. Iremos importá-lo e, agora que temos uma dependência do Renderer, indicaremos em darkenOn() para que render chame setElementStyle() em quem o estilo será modificado, no caso, this.el.nativeElement. A propriedade CSS a ser alterada é filter com valor brightness, cujo *default* é 100%, mas cairá para 70%.

E quando tirarmos o mouse de cima do *card*, indicaremos que o brightness passará a 100%:

export class DarkenOnHoverDirective {

constructor(

private el: ElementRef,

private render: Renderer

) {}

@HostListener('mouseover')

darkenOn() {

this.render.setElementStyle(this.el.nativeElement, 'filter', 'brightness(70%)');

}

@HostListener('mouseleave')

darkenOff() {

this.render.setElementStyle(this.el.nativeElement, 'filter', 'brightness(100%)');

}

}

Salvaremos, abriremos o navegador, e confirmaremos que o efeito funciona como gostaríamos. Para vermos o efeito em outro local, vamos incluir um style fixo em <h1> de photo-list.component.html:

<h1 style="background: red" apDarkenOnHover>Photos</h1>

<ap-search

(onTyping)="filter = $event" [value]="filter">

</ap-search>

Assim, conseguimos um código isolado para aplicação de efeitos em elementos diversos!

Mas e se quisermos um valor diferente de escurecimento, ou brightness? Só precisamos acessar a diretiva e indicar que ela possui uma *inbound property* @Input(), de forma similar ao que fazemos com componentes. Usaremos um valor padrão de 70%, e o brightness será utilizado em darkenOn() com a ajuda do *Template String*.

export class DarkenOnHoverDirective {

@Input() brightness = '70%';

constructor(

private el: ElementRef,

private render: Renderer

) {}

@HostListener('mouseover')

darkenOn() {

this.render.setElementStyle(this.el.nativeElement, 'filter', `brightness(${this.brightness})`);

}

@HostListener('mouseleave')

darkenOff() {

this.render.setElementStyle(this.el.nativeElement, 'filter', 'brightness(100%)');

}

}

Salvaremos, voltaremos ao navegador, e verificaremos que tudo funciona conforme esperado. Podemos testar um valor mais destoante no brightness do <h1> de photo-list.component.html, como 20%:

<h1 style="background: red" apDarkenOnHover brightness="20%">Photos</h1>

Ao retornarmos ao navegador e passarmos o mouse sobre o texto "Photos", o escurecimento é bem maior. Se trocarmos 20% por 5%, passará a ser quase preto. Entendemos, portanto, que a **diretiva pode receber parâmetros, assim como componentes**.

Em seguida, podemos deletar o <h1>, criado apenas para fins de teste. E para finalizar, vamos resolver o espaçamento superior nas imagens, correspondente ao title, pois em card.component.html exibimos o <h4>, no entanto ele não possui valor algum.

Poderemos utilizar a diretiva \*ngIf="title". Caso o title esteja em branco, seu valor será falso e nada será exibido.

<div class="card border-light text-center">

<h4 class="card-header" \*ngIf="title">{{ title }}</h4>

<div class="card-block text=justify">

<ng-content></ng-content>

</div>

</div>

No navegador, não teremos mais aquele espaçamento "sobrando", deixando o layout ainda mais elegante.