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.