**40. Angular CLI**

É uma ferramenta de linha de comando, que auxilia a criar projetos em Angular, criar projetos, diretivas e etc. Se já tiver o npm instalado, no console, digite:

npm cache clean [*enter*]

npm install –g @angular/cli

Após concluir a instalação, ainda no console, passaremos a utilizar o comando "ng", que é um comando do Angular. Digitando então,

ng -v [*enter*]

Voce poderá ver qual a versão do Angular CLI instalado.

**41. Angular CLI – criando um novo projeto** *(aula 8.7)*

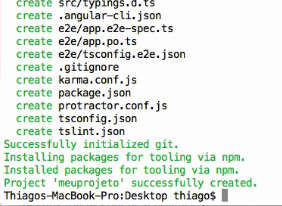
No console, digite ng new meuprojeto [*enter*]

Se precisar configurar Proxy, o comando é:

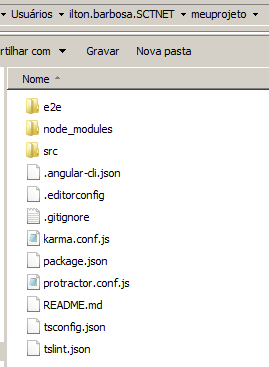
npm config set proxy http://usuario:senha@host\_proxy:porta\_proxy

Para maiores esclarecimentos sobre proxy: <https://jjasonclark.com/how-to-setup-node-behind-web-proxy/>

Após executá-lo, ele levará um tempo para instalar os pacotes, até que exiba, no final da lista de arquivos a mensagem de sucesso:



O projeto foi então criado na pasta do seu usuário do Windows.



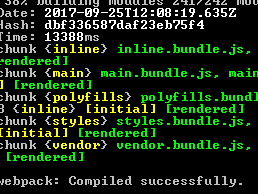
Arquivos e subpastas do projeto.

O nome da pasta é o nome do projeto

 O próximo passo é, ainda no console, acessar a pasta do projeto e digitar o comando abaixo para inicializar o servidor que rodará o projeto:

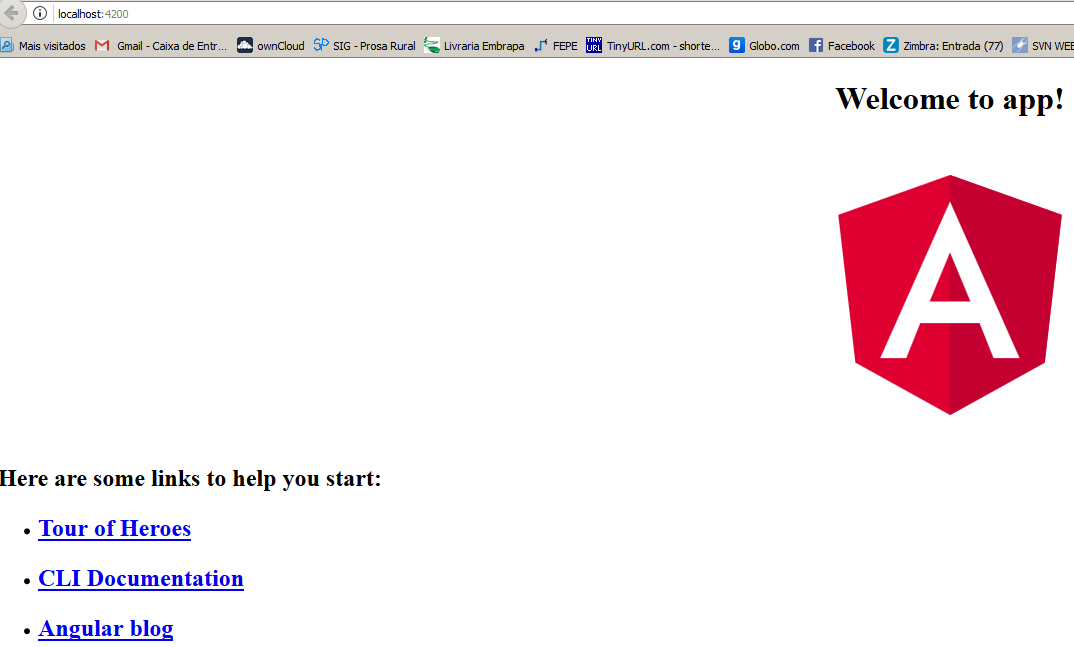
ng server [enter]

Após este comando, o Angular CLI executará uma série de comandos para criar o build do projeto. E no final exibirá a mensagem de sucesso.



A partir de então, basta digitar no browser "localhost:4200" (O Angular utiliza a porta 4200).

Está é a tela inicial do projeto:

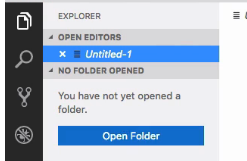


Caso não apareça esta tela para você, pode ser que a porta 4200 já esteja sendo utilizada no seu computador. Nesse caso será necessário informar uma outra porta, no comando "ng server", da seguinte forma:

ng server –port 8181 [*enter*] Caso queira utilizar a porta 8181.

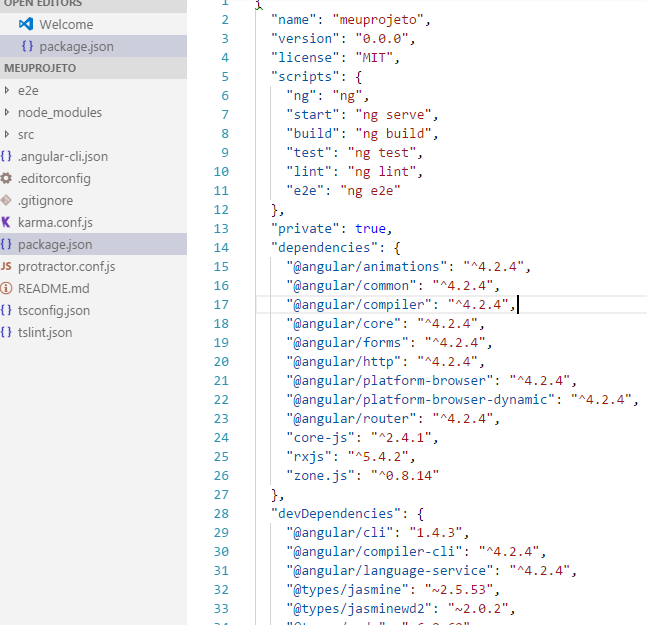
**42. Utilizando o editor Visual Studio Code** *(aula 8.8)*

Clique em Explorer -> **Open Folder** para abrir a pasta do projeto criado pelo Angular CLI.



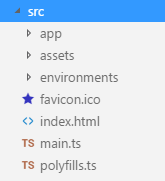
O servidor agora poderá ser inicializado a partir do Visual Studio. Clique no menu **View -> Integrated Terminal**. Será exibida uma tela onde você poderá digitar o comando de inicialização do servidor.

No arquivo **package.json** está o nome do projeto e a lista de arquivos das dependências do projeto. Nele estão as dependências necessárias para executar o projeto como também as dependências necessárias somente para o ambiente de desenvolvimento, que são as "**devDependencies**". Ao compilar o código, o Angular CLI não acrescenta estas dependências.

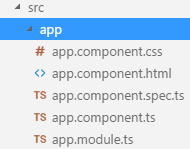


Os arquivos das dependências ficam na pasta **node\_modules**.

Os arquivos do nosso projeto deverão ficar na pasta "**src->app**". O arquivo **index.html** é aquele que foi exibido na tela inicial do projeto criado pelo Angular CLI. No entanto o conteúdo deste arquivo está no arquivo **app.component.html**, o qual é carregado no index.html, pelo **app.component.ts**.



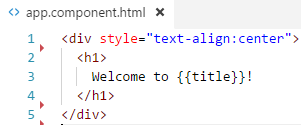
O componente principal do nosso sistema é o **app.component.ts**, que fica na pasta APP. Ele recebe o nome de "**app-root**".



O arquivo **app.component.spec.ts** é utilizado para testes unitários. Este assunto não será tratado neste curso.

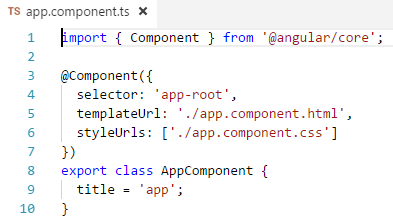
O Angular trabalha com uma técnica chamada de **interpolação**, que possibilita capturar valores de variáveis do JavaScript e exibir na tela das páginas do sistema.

Somente para testar, substitua o conteúdo do nosso arquivo app.component.html, de forma que fique conforme abaixo:



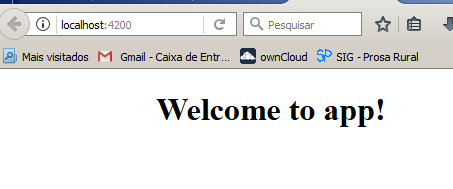
Interpolação (uso de chaves duplas). O nome da variável é "title", mas poderia ser qualquer outro nome.

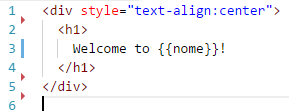
No caso do código acima, o Angular buscará o valor da variável "**title**" no arquivo o **app.component.ts**.

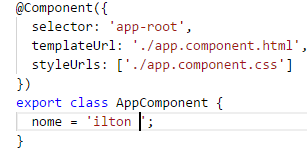


Aqui ele insere no index.html, o conteúdo do arquivo **app.component.html**.

Da forma como está, ao acessarmos nosso sistema pelo browser (localhost:4200), agora teremos a tela principal da seguinte forma:



Agora, se alterarmos o nome da variável "title" para "**nome**" e colocarmos nosso nome no valor da variável, teremos então que alterar o nome da variável também no arquivo **app.component.html**. Logo teremos o seguinte resultado:





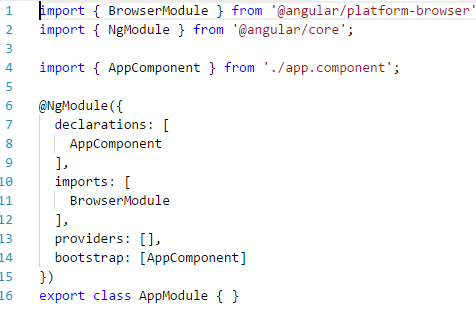
Caso você copie algum outro projeto, seja do GitHub ou de algum amigo seu, ele poderá vir sem as dependências. Nesse caso, ao abrir o projeto no Visual Studio, será necessário digitar um comando na console, para que as dependências possam ser baixadas da internet para a pasta node\_modules do seu projeto. Caso a pasta não exista, o próprio Angular CLI criará.

O comando para baixar as pendências deve ser "npm install", o próprio Angular lhe dará esta dica, caso tente executar o projeto sem antes ter baixado as dependências.

Este comando deverá ser executado dentro da pasta do projeto.

**43. Bootstrapping e AppModule** *(aula 9.1)*

O arquvo **app.module.ts** é o módulo principal do nosso sistema. É a partir dele que a Angular faz a ligação entre o index.html e o arquivo app.component.html, que é o componente principal do sistema.



É nesta linha que o Angular define o app.component, como sendo o componente principal da aplicação.

É no arquivo **main.ts**, que o Angular define o "app.module.ts" como sendo o módulo principal. Tudo começa por este arquivo main.ts.

**44. Componentes** *(aula9.2)*

Um sistema criado com Angular utiliza uma combinação (um conjunto) de componentes, onde cada componente tem o seu template HTML, estilos CCS e sua lógica de negócio. Se tivermos elementos da nossa tela que poderão ser reutilizados em outras telas, recomenda-se implementá-los como componentes.

Os componentes controlam partes da página. Por exemplo, suponha que você tenha uma barra de navegação que deverá estar em todas as telas do seu sistema, esta barra então poderá ser um componente.

**45. Criando Componentes** *(aula9.3)*

Existem duas formas de criar componente no Angular. Uma delas é manualmente, criando o arquivo e digitando o código. A outra é utilizando o Angular CLI.

 Para criar manualmente, no Visual Studio, dentro da pasta "app", crie uma nova pasta com o nome do componente que deseja criar. É uma boa prática colocar seus componentes dentro de pastas apropriadas. Se, por exemplo quiser criar um componente com o nome de "hello", o nome do arquivo deverá ser **hello.component.ts** que deverá estar dentro da pasta "hello". Esta formatação do nome é somente uma convenção de mercado.

Para que o Angular reconheça este arquivo como um componente é necessário um "Decorator", conforme está nas linhas 3 a 10.

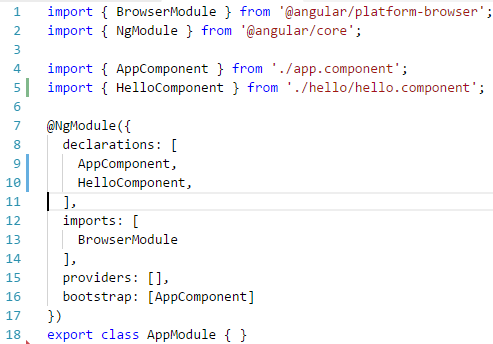
Dentro do Decorator temos os metadados (linhas 4 a 8).

O "decorator" pertence à biblioteca "core" do Angular, por isso a necessidade do import na linha 1.

O termo "export" é para que o componente possa ser visto externamente.

* A propriedade "**selector**" é para definir o nome da TAG que será utilizada no html para chamar este componente.
* A propriedade "**template**" é o código HTML que será exibido na página. Observe que o conteúdo do template fica entre crases ( ` ) – linha 5 e linha 9. Se este conteúdo for muito extenso, deve-se então utilizar a propriedade "templateUrl", que referenciará um arquivo html, como conteúdo do template.
* A variável nome que está na linha 7, recebe valor na linha 13.

Agora, é necessário registrar o novo componente como sendo parte da estrutura do nosso sistema. Isso é feito no arquivo **app.modules.ts** onde deverão estar todos os componentes que utilizaremos.



Registrando/declarando o módulo "hello"

Para mostrar o uso deste componente, poderemos implementá-lo no nosso componente principal que é o **app.component.html**, ao invés de implementá-lo diretamente no nosso arquivo index.html, que não é recomendado fazer.



Esta TAG é a referência ao nosso módulo "hello".

No browser então, ficará da seguinte forma:



**Para criar um componente utilizando o Angular CLI** é necessário abrir uma nova instância do terminal do Visual Studio. Basta clicar no sinal de **+** que fica no canto inferior da tela.

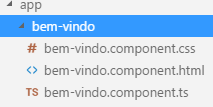


No terminal digite: ng generate componente bem-vindo --spec=false [enter]

A forma abreviada deste comando seria: ng g c bem-vindo --spec=false [enter]

O "—spec=false" é para o Angular não gerar o arquivo com extensão "spec", que é utilizado para testes unitários.

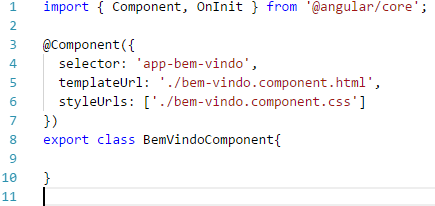
Ao executar este comando, o Angular cria a pasta "bem-vindo" e dentro dela os arquivos do componente.



O arquivo **bem-vindo.component.ts** deverá estar com o seguinte conteúdo:

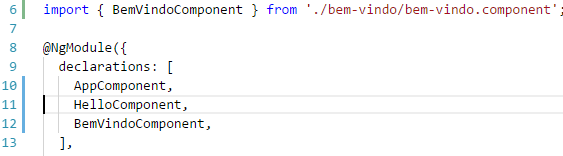


... mas para simplifica-lo poderemos retirar o "implements OnInit" e as linhas 10 a 13, por enquanto.



**templateUrl** referenciando o arquivo com o conteúdo html a ser exibido na página.

A partir deste ponto, o componente já poderá ser utilizado. O próprio Angular já registra este novo módulo no arquivo **app.modules.ts**.



**46. Bootstrap** *(aula9.4)*

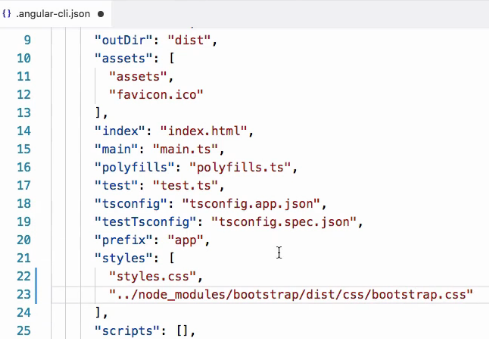
Framework CSS para aplicações web. Não é uma biblioteca para Angular. O curso não é focado em Bootstrap, será utilizado apenas para dar uma boa aparência ao sistema que estamos desenvolvendo.

Para instalá-lo no nosso projeto utilizaremos o **npm**, que é o nosso gerenciador de pacotes. No console, digite:

npm install bootstrap --save

O "--save" é para incluir o bootstrap nas dependências do nosso projeto (arquivo **package.json**).

Após instalar, verifique se ele aparece na pasta **nodes\_module**. Na pasta "**dist/css**" do Bootstrap, tem um arquivo chamado "**bootstrap.css**", é este arquivo que utilizaremos, portanto é necessário referenciá-lo dentro do arquivo ".angular\_cli.jason", da seguinte forma:



Incluir esta referência ao arquivo css do Bootstrap.

Para testar se o bootstrap está funcionando, digite o código abaixo no arquivo "**app.component.html**" e verifique no browser se está tudo ok.



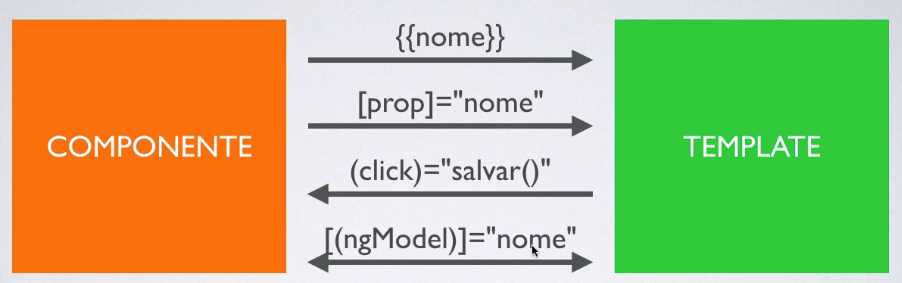
**47. Introdução a data binding** *(aula 9.5)*

Data binding são formas utilizadas para vincular componentes a templates e vice e versa, da forma como já vimos no exemplo do "Alô Mundo", em que trabalhamos com a variável "nome". A variável que está no template, buscou o valor dela que estava na classe/componente, que é o que chamamos de interpolação.

Existe também o recurso "propertie binding" que é uma outra forma de fazer este vínculo ( [prop]="nome" ).

Temos também o "event binding", que é utilizado para executar métodos a partir do click em um botão do formulário ( (click) = salvar()).

Ainda temos o "two-way data binding"que é uma mistura dos dois acima. Este será explicado posteriormente.



Para implementar um event binding do Angular é necessário acrescentar ao botão do formulário o termo "on-click". Não confundir com o "onClick" do JavaScript.

Exemplo:

<button type="button" class="btn btn-primary" on-click="adicionar()">Adicionar</button>

Observe que o nome do método não precisar estar entre chaves duplas (interpolação).

Nesse caso, o método "adicionar()" deverá estar implementando no componente da página html que implementa o formulário. A forma mais utilizada do event binding é a que chamamos de "sintaxe sugar", em que, ao invés do on-click, escreve-se (click) entre parênteses mesmo.

Exemplo:

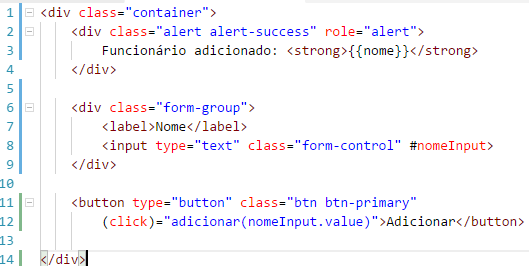
<button type="button" class="btn btn-primary" **(click)**="adicionar()">Adicionar</button>

**48. Data binding em formulários** *(aula 9.8)*

Em formulários trabalhamos com o que chamamos de "variáveis de referência de template" ou "Templates reference variables". Em um campo <input> por exemplo, esta variável será representada da seguinte forma:

<input type="text" class="form-control" #**nomeInput**>

Veja, o código html abaixo como exemplo:

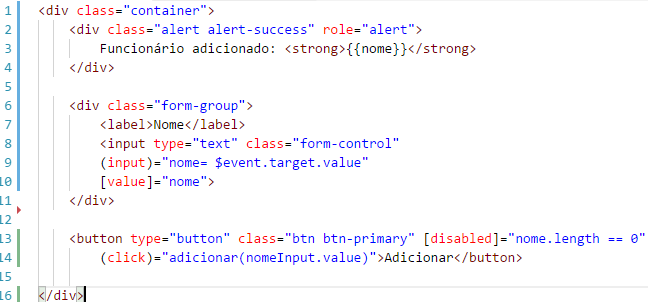


O valor da variável "**nomeInput**" está sendo passada como parâmetro no método adicionar() do botão "Adicionar". Dessa forma o componente desta página deverá ter implementado o método "adicionar()" que receba como parâmetro este valor do input. Então no arquivo "app.component.ts" teremos o método **adicionar()** da seguinte forma:



Ao digitar um nome e clicar no botão "Adicionar", o nome digitado será exibido no topo da tela, pois ao alterar o valor de uma variável do componente, este valor é automaticamente atualizado na página html, sem que aconteça um reload da página.

Caso queira desabilitar o botão Adicionar, de modo que ele somente fique ativo após o usuário digitar o nome no campo <input>, o código html ficará da seguinte forma:



Ao digitar algum valor (ação input), a variável "nome" deixa de ter tamanho zero.

Esse recurso é o que chamamos de "two-way data binding", pois o valor é passado ao mesmo tempo da página (template) para o componente e vice-versa.

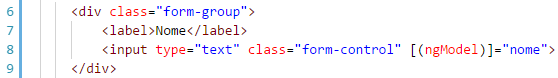
Os uso dos parênteses, como nas linhas 9 e 14, é o que chamamos de "**event binding**". E o uso de colchetes, como na linha 10, é o que chamamos de "**property binding**".

Podemos também utilizar o "event binding" e o "property binding" juntos de uma só vez, deixando assim nosso código mais limpo. Sendo uma forma mais adequada de utilizar o "two-way data binding". Para utilizar este recurso é necessário importar o "**forms module**" no arquivo "**app.module.ts**", da seguinte forma:



As linhas de 8 a 10 do nosso template (**app.component.html**), fica então, da seguinte forma:

Uso do "event binding" e do "property binding" juntos de uma só vez (two-way binding).



Nome do atributo no componente.



Agora o método adicionar() não precisa mais de parâmetro.

Retire o parâmetro do método adicionar também no componente.

**49. Diretivas** *(aula 9.11)*

Diretivas é algo fundamental em aplicações Angular. São instruções que passamos no template html. Existem três categorias de Diretivas: Diretivas de componentes, Diretivas estruturais e Diretivas de Atributos.

* Exemplo de diretiva de componente:



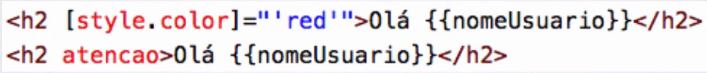
Quando no nosso código html definimos o local em que um componente deverá ser adicionado.

* Exemplo de diretiva estrutural:



Quando o conteúdo de uma tag do nosso código html sofre alteração conforme implementação/comportamento do componente. A diretiva "**ngIf**" implementa uma condição. Nesse caso o nome do usuário somente será exibido na tela se o usuário estiver logado. Quando implementamos diretivas estruturais no Angular é necessário acrescentar o **asterisco**.

* Exemplo da diretiva de atributo:



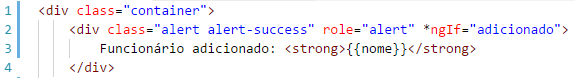
Esta diretiva modifica o comportamento ou a aparência do elemento html. Na segunda linha temos uma diretiva customizada que também pode ser uma maneira de implementar a alteração da cor do elemento html.

Um exemplo da implementação da diretiva **ngIf**, podemos ver implementado no **app.component.html**, de forma que a "<DIV> das linhas 3 a 4 somente seja exibida após adicionar um novo nome, clicando no botão adicionar().

Nesse caso acrescentaremos um novo atributo ao componente **app.component.ts**. Este atributo será do tipo booleano e terá valor inicial **false**, passando para **true** após um novo nome ser adicionado pelo formulário.



Agora adicionaremos a diretiva "**ngIf**" na <DIV> do app.component.html, da linha 3 a 4.



Ao invés da diretiva **ngIf**, podemos utilizar também a diretiva **hidden**.



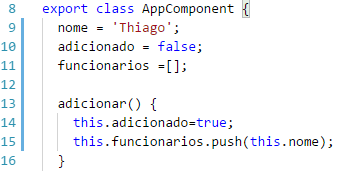
A diferença entre as duas é que utilizando a ngIf a <div> é literalmente excluída do html, sendo acrescentada somente se o valor de "adicionado" for "true". Enquanto que a diretiva "hidden", mantem a <DIV> no código html, mas ocultada pelo atributo hidden que é definido no CSS com valor oculto. O desenvolvedor deverá utilizar a que for mais conveniente dependendo do caso, considerando questões de segurança.

No campo <input> também podemos incluir um **event binding** que oculta a <div> do cabeçalho sempre que o usuário clicar novamente neste campo.

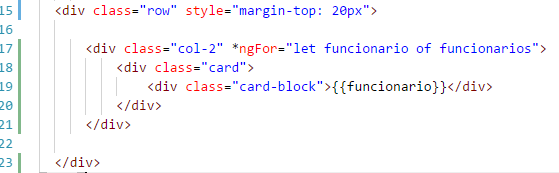


**50. O uso da diretiva ngFor** *(aula 9.13)*

Utilizada para trabalhar com listas. No caso do nosso sistema, o primeiro passo é criar uma array no nosso componente principal, de forma que nos possibilite trabalhar com listas. Agora cada novo nome que for adicionado, via formulário, este será armazenado num array. O código do componente então deverá ter o array implementado da seguinte forma:



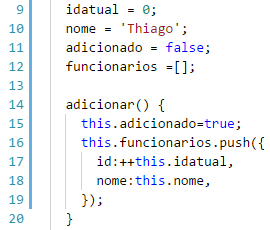
Dentro da <div class='container'> do nosso código html, acrescentaremos o código abaixo:



Veja que na linha 17 estamos implementando o uso da diretiva "\*ngFor". Ela percorre o array funcionários, do componente, utilizando uma variável auxiliar que aqui recebeu o nome de "funcionário", mas poderia ser qualquer outro nome.

Este array de funcionários também poderia armazenar um objeto Funcionário, de forma que possibilite armazenar não só o nome do usuário, mas também outras informações dele. Podemos, por exemplo, criar um ID de funcionário, que terá como valor, o número do funcionário registrado. Este valor será incrementado automaticamente à medida que adicionarmos novos funcionários à lista.

No componente, implementaremos uma variável a mais e faremos com que o array passe a armazenar não mais somente um nome, mas um objeto, uma vez que objetos podem ter vários atributos. Ficando então da seguinte forma:

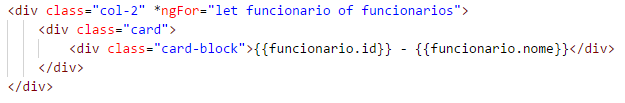


Para armazenar um objeto num array utiliza-se as chaves.

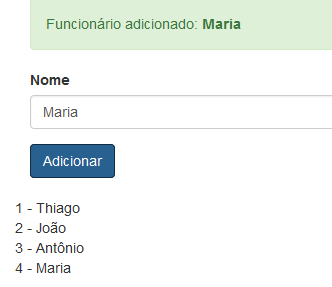
Criamos a variável idatual que inicia com valor zero.

Esta variável é incrementada a cada novo usuário cadastrado.

No nosso código html agora teremos que referenciar o **id** e o **nome** do funcionário dentro do **ngFor**, da seguinte forma:



Exemplo do resultado do nosso código html no browser:

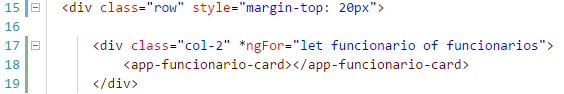


**51. Binding de propriedades customizadas com @Input** *(aula 9.14)*

Já aprendemos como mostrar na página html os valores dinâmicos advindos de um componente. Mas em alguns casos será necessário fazer o caminho oposto – passar o conteúdo de um objeto, que está na página html, para um outro componente.

No exemplo anterior vimos como listar os funcionários a partir de um objeto de funcionários advindo de um componente. Mas, caso quiséssemos que a <div class="card">, do arquivo **app.component.html**, fosse também um componente, como ficaria o nosso código html? Lembrando que se transformarmos esta <div> em um componente, poderemos reutilizá-la sempre que precisarmos exibir os dados do funcionário.

Supondo que, ao transformar esta <div class="card"> em um componente, o nome deste componente seria "<app-funcionario-card>", logo, esta <div> no nosso código html, passa a ser uma "tag", da seguinte forma (linha 18):



Veja que no lugar da <div class="card"> agora temos a "tag" que referencia o componente <app-funcionario-card>.

Nesse caso, teremos que criar este componente. Lembrando que o comando a ser digitado no terminal é:

ng g c funcionario-card --spec=false [*enter*]

OK. Agora temos o componente já criado e referenciado no arquivo **app.component.html**.

Nos arquivos gerados para este novo componente, coloque o conteúdo da <div class="card"> dentro do arquivo "**funcionario-card.component.html**".



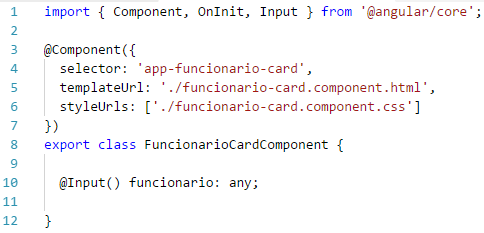
A novidade agora é, como passar o objeto "funcionário" para dentro do componente app-funcionario-card.

A "tag" do componente **<app-funcionario-card>** deverá receber um parâmetro que possa ser acessado pelo seu componente correspondente. Isso se faz da seguinte forma:



Perceba que na linha 17, ao invés de "**let funcionário**..." colocamos "**let f**...", para evitar confundir a variável "funcionario" da iteração com a variável "[funcionário]" do property binding.

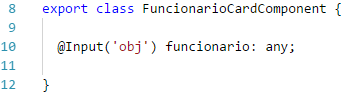
Esta variável [funcionário] da linha 18, é a que será lida pelo componente <app-funcionario-card>, no arquivo "**funcionário-card.component.ts**". Para que ele consiga ler esta variável que está vindo do html, seu código deverá ficar da seguinte forma:



* Linha 1 estamos importando o elemento @Input que será utilizado na linha 10.
* Linha 4 – temos o nome do componente, conforme referenciado no arquivo html.
* Linha 5 – temos o arquivo html (template) que contém o conteúdo da div card.
* Linha 10 – O @Input() serve para definir que a variável "funcionario" poderá ser acessada fora deste componente. O "any" é para indicar que ela é de um tipo genérico.

Podemos ainda atribuir um nome 'alias' (apelido) para nossa variável "funcionario", da seguinte forma:

Nesse caso, o nome do nosso 'alias' será "obj". Um apelido para a variável "funcionário"



Este 'alias' agora poderá ser utilizado no nosso código html, da seguinte forma:



**52. Binding de eventos customizados com @Output e EventEmitter** *(aula 9.15)*

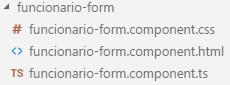
Até este ponto, da forma como está implementado, nosso formulário, que está no arquivo **app.component.html**, envia dados para o seu componente correspondente, que é o arquivo **app.component.ts**, que por sua vez, armazena estes dados em um array. O conteúdo deste array é então exibido em uma lista, logo abaixo do formulário, por meio do componente **funcionario-card.component.ts**.

Vejamos agora, como ficaria se transformássemos nosso formulário em um outro componente. A implementação fica um pouco mais complexa. O novo componente do formulário, terá que enviar seus dados para o componente que exibe a lista de funcionários na tela.

A forma de enviar dados de um componente para outro é utilizando o recurso **EventEmitter**.

O primeiro passo é criar o componente do formulário. O nome será **funcionário-form**.

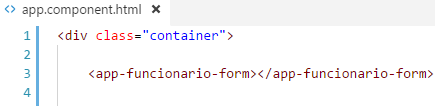
ng g c funcionario-form --spec=false [*enter*]



Retire o conteúdo do formulário do arquivo **app.component.html** e cole no arquivo   
**funcionario-form.component.html**.



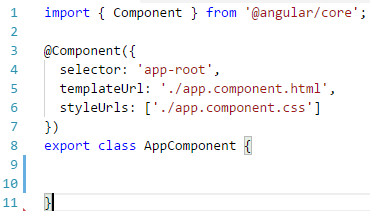
O formulário agora deverá ser representado somente pela "tag" do novo componente criado (funcionario-form). Mantendo ainda, abaixo dele, a <div> que exibe a lista de funcionários.



Agora que o formulário está dentro do componente "**funcionario-form**", ele não consegue mais acesso à função "**adicionar()**" que ainda está no componente principal "**app.component.ts**". A princípio um componente não consegue acessar uma função que esteja em outro componente.

Para solucionar este problema, o primeiro passo é mover a função "adicionar()" do componente principal para o componente "**funcionario-form**" arquivo "**funcionario-form.component.ts**".

O componente principal fica sem a função "adicionar()":



O componente "funcionario-form" (funcionario-form.component.ts) fica com o seguinte conteúdo:

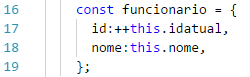


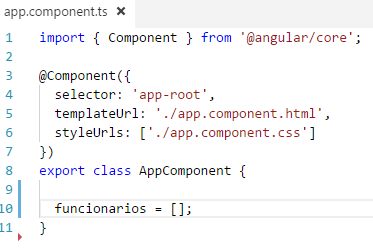
Lembre-se de remover o conteúdo padrão que já vem dentro da "**class FuncionarioFormComponent"** e também o "**implements OnInit**", conforme já fizemos nos componentes criados anteriormente.

Agora o formulário já consegue acesso à função "adicionar()", mas isso ainda não é o suficiente para funcionar perfeitamente. O sistema ainda não conseguirá exibir a lista de funcionários. Isso porque o array de funcionários agora está no componente "funcionario-form", enquanto que a exibição da lista de funcionários está no componente principal.

Para solucionar este problema podemos **mover para componente principal o array de funcionários** que está na linha 13. Esta ação gerará um erro na linha 17. Então precisamos agora transformar este array da linha 17 para uma variável, da seguinte forma:

Agora precisamos fazer com que o conteúdo desta variável "funcionario" possa ser adicionado ao array "funcionários[]" que está no componente principal.





Componente principal agora somente com o array de funcionários.

Para possibilitar a comunicação entre os dois componentes, precisaremos utilizar os recursos **@OutPut** e o **EventEmitter** (emissor de eventos), para implementar o "**binding de eventos**" .

No componente "**funcionario-form**" criaremos uma variável que será o **emissor de eventos**, de forma que, toda vez que um novo funcionário for cadastrado, ele enviará mensagem ao outro componente.



Emissor de eventos.

Agora no template do componente principal (app.component.html), implementaremos as tags que possibilitam realizar um binding de eventos. A tag <app-funcionario-form> deverá ficar da seguinte forma no arquivo app.component.html:



Observe que esta tag agora faz referência à variável "**funcionarioAdicionado**", que criamos no componente "funcionario-form", conforme pode ser visto no código do componente que está mais acima.

Mas para possibilitar acesso à variável "funcionarioAdicionado" do componente, é necessário acrescentar a esta variável no componente, o recurso que faça com que ela possa ser acessada ou fique visível externamente. Portanto acrescente a ela o elemento **@Output()**, da seguinte forma:



Para tanto, é necessário acrescentar o **Output** na lista de imports deste componente:



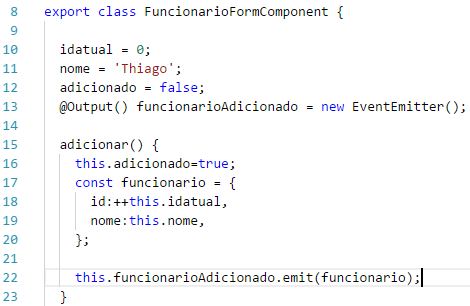
Este recurso na verdade, é para possibilitar saída de dados do componente. Observe que a variável "funcionarioAdicionado", instancia a classe "EventEmitter()", que possibilita a saída de dados a partir de um evento. O evento será executado pelo método "adicionar()". Ao executar o método adicionar, este método deverá enviar para fora do componente os dados do novo funcionário adicionado, por meio do EventEmitter.

Ainda com relação à tag "<app-funcionario-form>", observe que ela faz referência ao método "**aoAdicionar()**". Este é então o método que receberá os dados enviados para fora do componente "**funcionario-form**", de forma que estes dados possam ser adicionados ao array "funcionários" que está no componente principal.

Nesse caso o método "aoAdicionar()", deve então receber do componente "funcionario-form" um objeto do tipo "**event**", ficando então da seguinte forma:

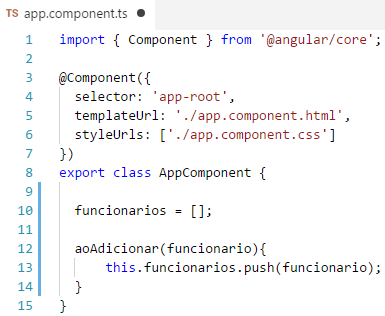


Para fazer com que os dados do novo funcionário sejam enviados para fora do componente "funcionario-form", é necessário acrescentar a linha de **comando 22**, ao componente "funcionario-form":



Observe que ela utiliza o método "emit()", da classe "EventEmitter" para enviar o objeto "funcionario" para fora do componente, por isso que na tag <app-funcionario-form> temos o método "aoAdicionar($event)", recebendo um evento como parâmetro.

Agora falta somente implementar o método "aoAdicionar()" no componente principal, que deverá adicionar o novo funcionário ao array de funcionários. Ele deverá ficar da seguinte forma:



Execute testes no servidor "localhost:4200" e verifique se está funcionando corretamente.

**53. Adicionando estilos CSS em componentes** *(aula 9.16)*

A cada novo componente criado, o AngularCli gera um três arquivos importantes: o arquivo ".html" que é o template do componente, o arquivo ".ts" que é o componente propriamente dito e o arquivo ".css" que é onde implementaremos nossos estilos CSS para o template.

Se por exemplo, quisermos aplicar a cor vermelha a todas as tags <label> do nosso template, bastaria criar o estilo no arquivo css, da seguinte forma:



Observe que no arquivo do componente (.ts), temos a referência para o arquivo do template e para o arquivo css. A diferença da referência para o arquivo css, é que o nome dele vem entre colchetes. Isso quer dizer que a referência ao arquivo css é um array de referências, possibilitando então referenciar vários arquivos css para um mesmo template. Esse recurso pode ser utilizado para melhor organizar os estilos css.

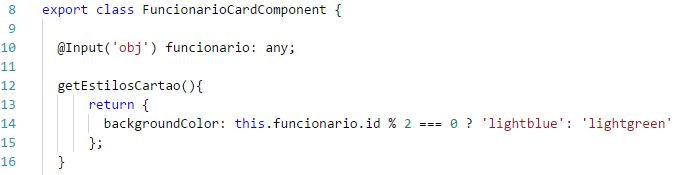
**54. Estilos CSS dinâmicos com ngStyle** *(aula 9.17)*

O recurso **ngStyle** é uma diretiva do Angular. Diretiva são recursos que possibilitam alterações no conteúdo de elementos Html. O uso desta diretiva possibilita criar código CSS dinâmico como por exemplo, para criar cores alternadas na linha de uma tabela (efeito zebra).

Vejamos por exemplo, o template "**funcionario-card**" (arquivo: funcionario-card.component.html). Nele temos uma <div> com a classe "card". A esta <div> poderemos adicionar uma diretiva que implementará a alternância de cores na lista de funcionários, da seguinte forma:



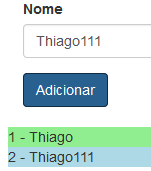
O termo ngStyle deve ficar entre colchetes. O método "**getEstiloCartao()**" deverá ser implementado no componente "funcionario-card" (arquivo: funcionario-card.component.ts), da seguinte forma:



Se quiser utilizar hífen no nome da propriedade então, deverá utilizar aspas: 'background-color'.

Linha 14: a cor da linha é definida pelo valor do ID do funcionário. Se número par pega a cor "lightblue".

O resultado deste exemplo ficará da seguinte forma:



**55. Estilos CSS dinâmicos com ngClass** *(aula 9.18)*

Como alterar a classe CSS de um elemento dinamicamente? Utilizaremos como exemplo a classe "badge" do Bootstrap, com a diretiva ngClass do Angular, conforme abaixo:



Criamos a tag <span> e adicionamos a ela a diretiva [ngClass], que recebe como conteúdo um array de classes do Bootstrap. Da forma como está o resultado será exibido conforme abaixo, mas esta ainda não é uma implementação de CSS dinâmico com ngClass:

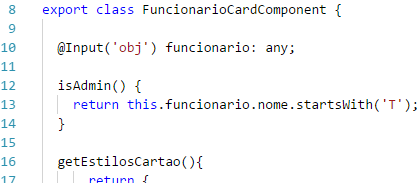


Efeito da classe "badge" do Bootstrap.

Vamos supor agora, que todo nome que iniciar com a letra "T", terá uma formatação diferente dos outros nomes. Nesse caso, o código html ficará assim:



Agora o conteúdo da [ngClass] está entre chaves, para indicar que se trata do uso de conteúdo em JavaScript. Neste conteúdo temos o método "isAdmin()", que precisará ser implementado no componente "funcionario-card", da seguinte forma:



**56. Iniciando o projeto do curso (AlgaMoney)** *(aula 10.3)*

Conforme já vimos nas aulas do SpringBoot, o projeto é um cadastro de pessoas e lançamentos.

O primeiro passo é acessar o console, criar a pasta do projeto e dentro dela digitar o comando que cria o projeto em Angular, conforme já vimos nas aulas anteriores: ng new algamoney-ui

O "-ui" é opcional e somente para indicar que se trata do frontend.

**57. Utilizando bibliotecas de componentes** *(aula 10.2)*

Existem várias bibliotecas que podem ser integradas a projetos com Angular. A que utilizaremos é a PrimeNG ([www.primefaces.org/primeng](http://www.primefaces.org/primeng)). É considerada uma das bibliotecas mais completas.

Após criar o projeto conforme indicado no item 56, acesse a pasta do projeto pelo Visual Studio Code e instale o "**primeng**", conforme comando abaixo a ser digitado no console do Visual Studio Code:

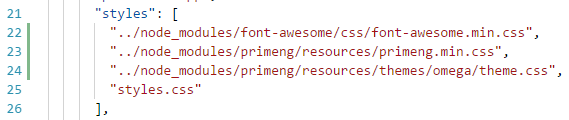
npm install primeng --save

Lembrando que o "—save" é para incluir o primeng no arquivo "package.json"

Além da "PrimeNg" incluiremos também no nosso projeto a biblioteca "fontawesome" (<http://fontawesome.io>). O PrimeNg utiliza esta biblioteca para trabalhar/adicionar ícones. A instalação é da mesma forma que o PrimeNg:

npm install font-awesome --save

Falta ainda adicionar três referências para arquivos CSS, necessárias para o AngularCLI. No arquivo "**angular-cli.json**" (está na pasta raiz do projeto), localize a propriedade "styles" e acrescente a ela as linhas de código, conforme está abaixo:

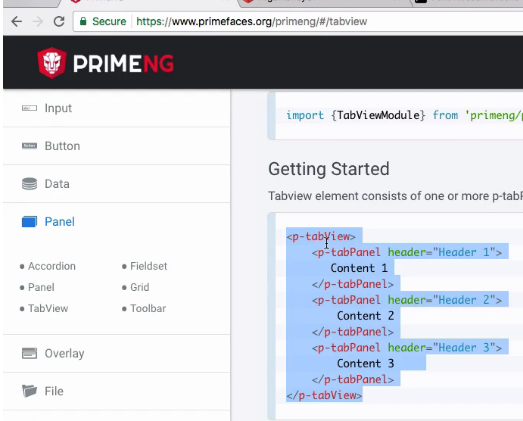


Estas linhas de código css podem ser encontradas no site do primeng ([www.primefaces.org/primeng/#setup](http://www.primefaces.org/primeng/#setup))

Na página que trata da integração com Angular CLI.

Conforme podemos ver na linha 24, utilizaremos o tema "ômega" do primeng.

Agora vamos testar a aplicação, para verificar se o primeng está funcionando. Implementaremos uma tabview do primeng para testar. Acesse o site do primeng e copir o código relativo a uma tabview, conforme indicado abaixo:



Agora cole este código no arquivo "app.component.html". Apague todo o conteúdo dele e cole o código abaixo:



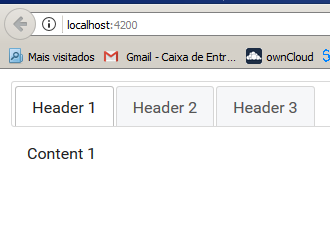
Em seguida acrescente a linha de importação do "tabview" no componente principal da aplicação (app.module.ts), da seguinte forma:

|  |
| --- |
| import {TabViewModule} from 'primeng/primeng'; |
|  |

Além desta linha é necessário acrescentar também o Tabview no array de imports deste mesmo arquivo:



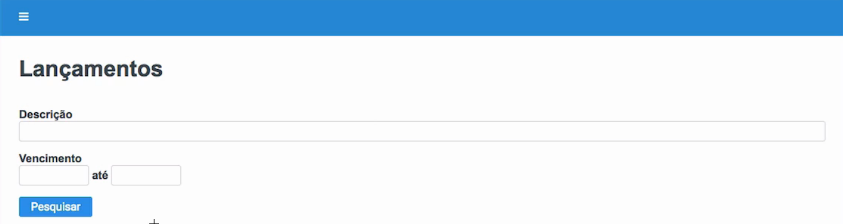
Salve todos os arquivos, aguarde a compilação automática e em seguida, o resultado então será este:



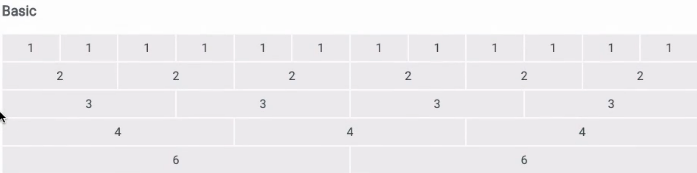
Fique atento para adicionar os imports necessários para cada novo componente que precisar adicionar ao seu proejto. No site do primeng tem o código de cada componente, inclusive dos seus imports necessários.

**58. Criando as primeiras telas do projeto** *(aula 10.4)*

Iniciaremos pela tela de lançamentos, que deverá ficar conforme imagem abaixo:

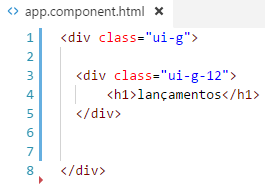


Recomenda-se começar estudando sobre grids CSS, que é uma forma de organizar os elementos na tela para que esta fique responsiva.



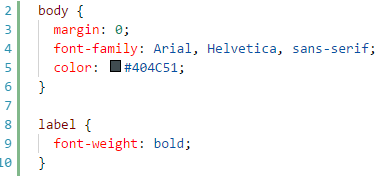
No site do primeng há um artigo sobre Grid CSS, na aba Panel. O Primeng já disponibiliza algumas classes que implementam recursos para tela responsiva. São as classes em que o nome inicia-se por "ui-....". Por exemplo, no caso da classe "ui-g-8" quer dizer que o elemento que utilizar esta classe, ocupará 8 colunas na tela.

Para criar a tela de lançamento, começaremos criando uma <DIV> com uma classe que indique que trabalharemos com grids, conforme está na linha 1 abaixo:



Aqui indicamos que o conteúdo da DIV irá ocupar 12 colunas da grid.

Na pasta raiz da nossa aplicação (fora da pasta app), tem o arquivo "styles.css". Ele serve para definição de estilos globais, isto é, que servirão para toda a sua aplicação. Coloque o seguinte código nele:



Todo o texto da nossa tela estará na fonte Arial e toda tag <label> terá texto em negrito.

Agora à nossa página html, acrescentaremos os campos do formulário:



O botão também utiliza uma diretiva do PrimeNg, portanto o import dele também deverá estar no "module.ts".

O sytle="**display:block**" é para quebrar a linha.

A classe "ui-fluid" é para fazer com que o elemento ocupe as 12 colunas do grid.

Utilizamos a diretiva "**pInputText**" da biblioteca do PrimeNg, que é um campo input, já com formatação adequada ao tema que estamos utilizando.

Para utilizar este elemento é necessário acrescentar o "import" referente a ele no nosso arquivo app.module.ts, conforme está na página seguinte.

Nosso arquivo app.modules.ts, ficará da seguinte forma:



Linha 5 – estamos importando somente o elemento "inputtext", de maneira a evitar que na compilação do projeto sejam incluídos outros elementos do PrimeNg que não estaremos utilizando.

Linha 16 – não esquecer de acrescentar o InputTextModule neste array de imports.

Linhas 6 e 17 temos o import do botão do PrimeNg.

Observe que nosso formulário de lançamento está dentro de uma <div> que utiliza a classe "container". Esta <div> serve para centralizar o conteúdo na tela. Para que fique centralizado é necessário acrescentar a classe container no arquivo "**styles.css**".



Aqui, no caso, nosso conteúdo ocupará um espaço de 1170 px centralizado. O parâmetro "auto" é para o browser colocar o mesmo espaço na direita e esquerda.

A instrução "@media" define uma condição para a aplicação da classe "container". Esta classe só será aplicada à nossa página, no caso em que a largura do Browser seja no mínimo de 1200px. Isto é assunto de estudo sobre Designer Responsivo, que não é o foco deste curso.

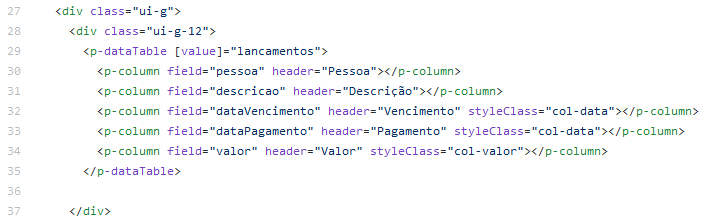
**59. Adicionando a tabela de dados do projeto** *(aula 10.5)*

O primeiro passo é acrescentar o import do "dataTable" ao arquivo "app.module.ts", conforme abaixo:

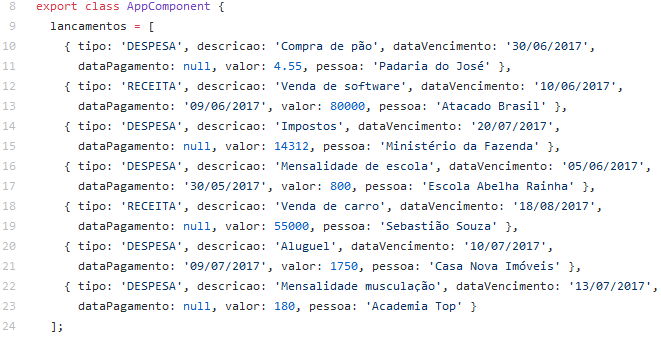
import { DataTableModule } from 'primeng/components/datatable/datatable';

Sem esquecer de acrescentá-lo também no array de imports.

Em seguida, no arquivo html (app.component.html), implementaremos a <div> para o "dataTable", que deverá ficar da seguinte forma, dentro da <div class="container"> :

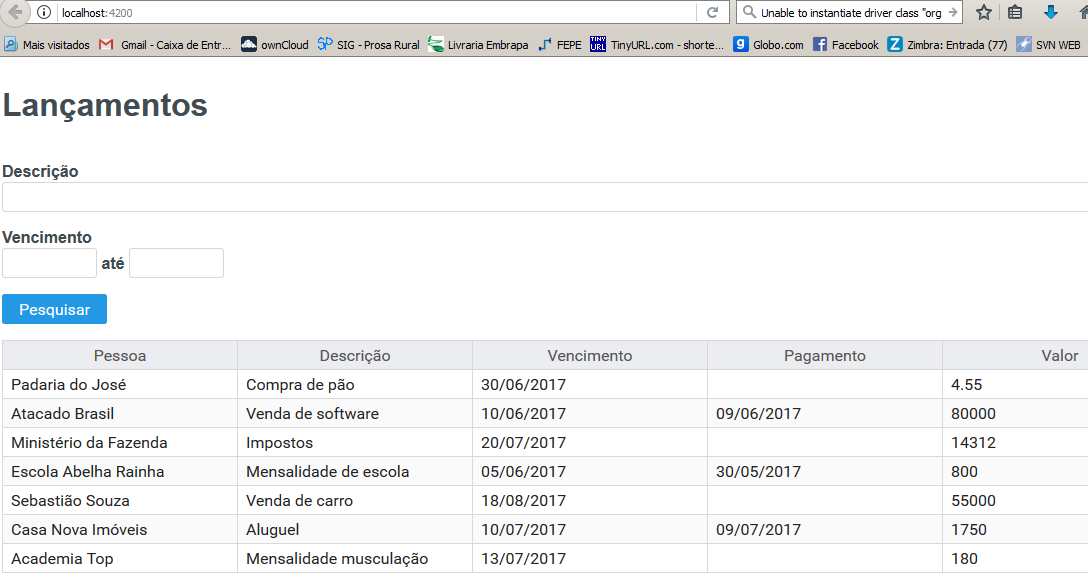


Linha 29 – Todo "dataTable" deve ter como valor um array. No nosso caso temos um array de lançamentos. Como ainda não estamos implementando a integração com nosso backend, utilizaremos um array para teste, que deverá ser implementado no arquivo "**app.component.ts**", da seguinte forma:



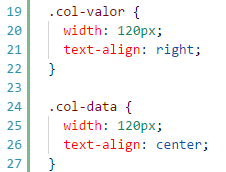
No nosso código html, observe que nas linhas 30 a 34, colocamos o nome dos atributos dos objetos do array de Lançamentos que queremos exibir na lista. Na linha 30 por exemplo, estamos exibindo o nome da pessoa, por isso utilizamos o termo "pessoa", conforme está na linha 11 do arquivo "app.component.ts".

O resultado exibido no browser será então, o seguinte:



Nas linhas 32 a 34 do nosso código html, estamos utilizando os estilos "**col-data**" e "**col-valor**". Estes estilos devem ser implementados no nosso arquivo "styles.css" que está na raiz do nosso projeto.

Os dois estilos que acrescentaremos são os seguintes:



**60. Customizando colunas da nossa tabela de dados, com ng-template** *(aula 10.6)*

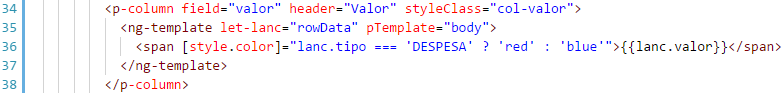
Vamos supor que seja necessário incluir o "R$" nos valores financeiros da nossa tabela. Para fazer isso precisaremos utilizar o recurso do ng-template.

A nossa coluna de valor na tabela html, ficará da seguinte forma:

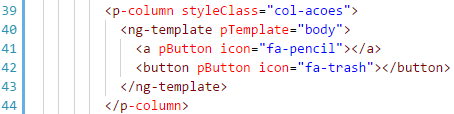


Na linha 35 temos o elemento "**let-lanc**". Esta é a forma de criarmos a variável "**lanc**" que está sendo utilizada entre as chaves duplas, para exibir o valor. O "**rowData**" é para se referir às linhas do objeto "lançamento" que estamos listando. O único elemento aqui que percente ao AngularCli é o "pTemplate".

Considerando que na nossa tabela de lançamentos, temos valores referentes a créditos e valores referentes a débitos, precisamos agora, exibir os créditos na cor azul e os débitos na cor vermelha. O código desta coluna de valores, nesse caso, ficará assim:

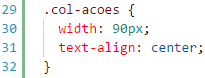


O próximo passo é acrescentar mais uma coluna, para exibir os botões de edição e exclusão dos registros. Ela deverá ficar da seguinte forma:

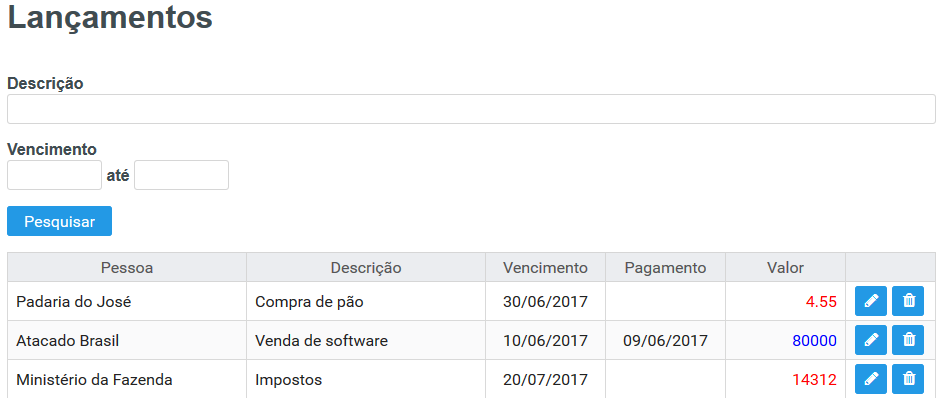


Os elementos "icon='fa-pencil'" e "icon='fa-trash'", são da biblioteca "font-awesome" que instalamos no nosso projeto anteriormente.

Na linha 39 utilizamos mais um estilo para customizar a largura desta coluna. No arquivo styles.css, ficará da seguinte forma:



O resultado agora, está da seguinte forma:



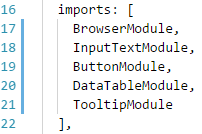
Ainda falta formatar os valores financeiros. Faremos isso mais adiante.

Nos botões de "Editar" e "Excluir" podemos incluir também uma mensagem a ser exibida ao usuário quanto passar o mouse sob eles. Basta acrescentar dois elementos do PrimeNg, à tag do botão:



Nesse caso o "**pTooltip**" precisa ser exportado no componente principal (**app.module.ts**).





**61. Implementando paginação na tabela de dados de lançamento** *(aula 10.7)*

Basta adicionar o elemento [paginator]="true" na tag <p-dataTable> e acrescentar também o elemento [rows]=10, para definir o número de linhas por página. Pronto!

O resultado será este:



**62. Implementando responsividade à tabela de dados de lançamento** *(aula 10.9)*

A melhor forma de fazer isso é utilizando o recurso do PrimeNG, que aplica regras CSS sem que precisemos trabalhar com código CSS. Basta acrescentar na tag <p-dataTable>, a propriedade [responsive]=true.

**63. Implementando o componente de pesquisa de lançamentos** *(aula 10.10)*

O "app.component" é nosso componente principal. Neste componente não é recomendável implementar funcionalidades muito complexas, estas devem ser componentes auxiliares, cabendo ao componente principal trabalhar com chamadas a estes outros componentes.

Sendo assim, transformaremos o formulário de pesquisa e o resultado de pesquisa em um compomente. Para tanto, o primeiro passo é criar o novo componente. No terminal digite então, o seguinte comando dentro da pasta do seu projeto:

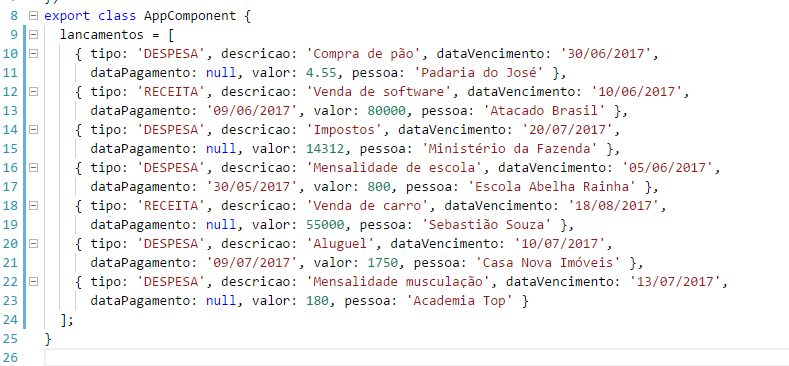
ng g c lancamentos-pesquisa --spec=false

g – abreviatura de "generate"

c – abreviatura de "component"

Agora, recorte todo o conteúdo do arquivo "app.component.html". Deixe ele vazio por enquanto. E cole este conteúdo dentro do novo arquivo criado, que é o **lancamentos-pesquisa.html**.

Recorte também do arquivo "**app.component.ts**" a propriedade "lacamentos", com todo o seu conteúdo e cole na classe do novo componente " **lancamentos-pesquisa.ts**":



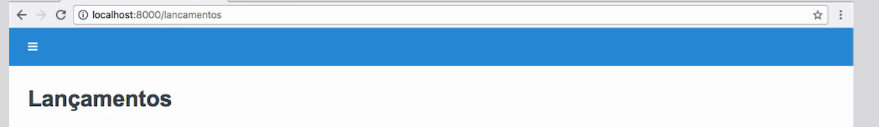
Remova o construtor e o "implements OnInit" da classe do componente "lancamentos-pesquisa.ts" e inclua nele a propriedade lançamentos que foi recortada do componente principal.

No arquivo "**app.component.html**", que havia ficado vazio, coloque nele somente a chamada para o novo componente criado, da seguinte forma:



**64. Implementando a barra de navegação** *(aula 10.11)*

A barra azul no topo da nossa tela será a barra de navegação.



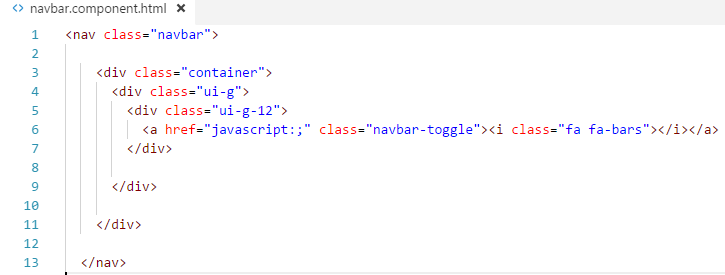
Para implementar esta barra de navegação, iniciaremos com o comando de criação deste componente, digitando no terminal o comando a seguir:

ng g c navbar --spec=false

Criado então o novo componente "navbar", acrescente ao arquivo "navbar.component.css", o seguinte conteúdo:



O arquivo "navbar.component.html", deverá ter o seguinte conteúdo:



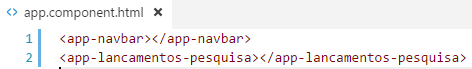
Lembrando que a tab "container" é para colocar o espaçamento lateral da forma como já fizemos no formulário de lançamentos. Lembrando também que a div da linha 4 serve para indicar que utilizaremos a grids para distribuir nossos elementos na tela e que a div da linha 5 divide o grid em 12 colunas.

Na linha 6 temos o código que gera as barrinhas do nosso menu de navegação.



Ainda na linha 6, perceba que estamos utilizado o estilo css "**navbar-toggle**" que já incluímos no arquivo "navbar.component.css". Ele serve para colocar a cor branca nas barrinhas do nosso menu de navegação.

No arquivo do componente principal "**app.component.html**", acrescentaremos a "tag" deste novo componente criado:



**65. Adicionando menu intercambiáve** *(aula 10.12)*

Ao clicar nas barrinhas do menu de navegação, deverá ser exibida na tela, o menu intercambiável. O primeiro passo é criar a lista <UL>, no nosso arquivo "**navbar.component.html**", conforme abaixo:



Na linha 14, colocamos um nome fictício, para representar o nome do usuário logado.

Na linha 13, estamos utilizando o estilo "navbar-menu", portanto temos que implementá-la no nosso arquivo "**navbar-component.css**", da seguinte forma:

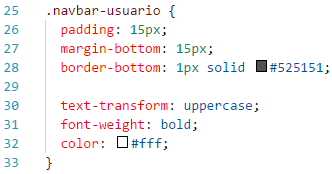


Linha 11 – o position:fixed, mantem o menu visível, mesmo que a tela seja rolada para baixo.

Linha 18 – o "z-index" está com um valor bem alto, para garantir que nenhum outro elemento sobreponha o menu.

Linha 22 – o list-style:none, é para não exibir as bolinhas (bullets), default da tag <UL>.

No código html, linha 14, estamos utilizando o estilo "navbar-usuario" e nas linhas seguintes o estilo "navbar-menuitem". Precisamos implementar estes estilos também no arquivo css do componente navbar, da seguinte forma:

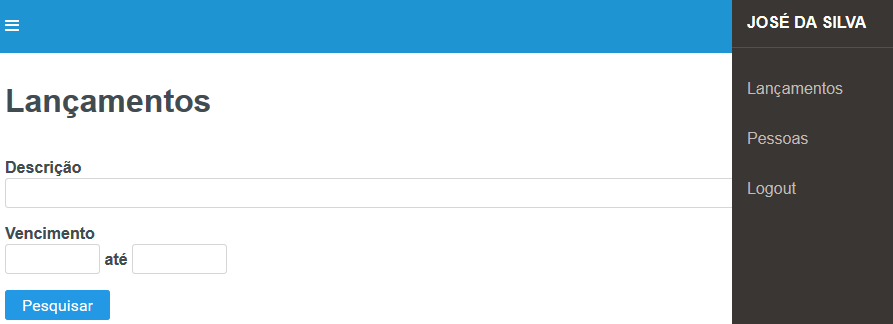
Agora precisamos implementar a função que exibirá o menu quando clicarmos nas barrinhas brancas.

Na linha 6 do arquivo "navbar-component.html" acrescente o recurso (click), conforme abaixo:



A variável "exibindoMenu" ficará intercalando seu valor entre "true" e "false", sempre que clicarmos no menu das barrinhas. Para tanto, no nosso arquivo "**navbar-component.html**", linha 13, foi acrescentado o elemento [hidden]="!exibindoMenu".

O resultado, ao clicar no menu das barrinhas, deverá ser o seguinte:



Este menu deverá ser exibido ao clicar nas barrinhas brancas.

**Obs.:** este recurso também pode ser implementado utilizando o "slidemenu" do PrimeNG (https://www.primefaces.org/primeng/#/slidemenu.) ou o "tieredMenu".

**66. Criando diretivas customizadas** *(aula 11.1)*

Como já foi dito, diretivas alteram a aparência ou o comportamento de um componente ou mesmo de uma outra diretiva. Além das diretivas já disponibilizadas, podemos criar nossas próprias diretivas.

Para criar uma diretiva, basta digitar o comando abaixo no console, dentro da pasta do seu projeto:

ng g d campo-colorido --spec=false

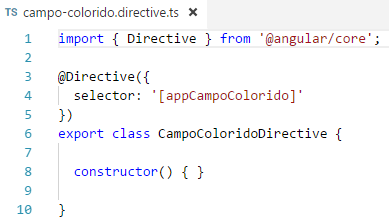
g = generate

d = diretiva

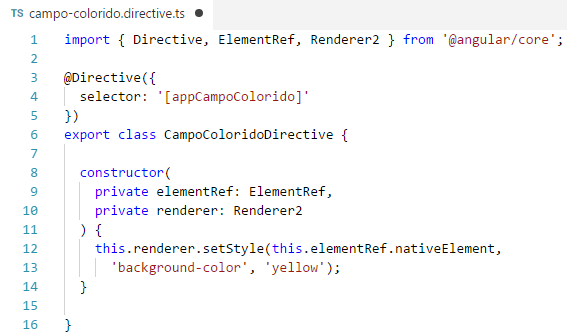
Voce verá na lista de arquivos, o novo arquivo criado para a diretiva: "campo-colorido.directive.ts". O Angular já inclui também automaticamente esta nova diretiva ao arquivo "**app.module**".

A princípio implementaremos esta diretiva somente para alterar a cor de fundo de um campo no formulário, o que poderia ser feito utilizando CSS, no entanto, veremos mais adiante que com uma diretiva pode-se fazer bem mais que somente mudar a cor. Este é somente um exemplo para começarmos a entender a função de uma diretiva.

Inicialmente, o arquivo "campo-colorido.directive.ts", deverá estar da seguinte forma:



No construtor desta diretiva injetaremos os elementos que implementarão a alteração da cor de fundo de um campo <input> do nosso formulário. Ele ficará então da seguinte forma:



**Linha 9** – declaramos a variável "elementRef" como "private", para que o Angula possa interpretá-la como atributo da classe da nossa diretiva. Este atributo é do tipo "ElementRef" (veja que ele está sendo importado na linha 1).

O "ElementRef" é o recurso do Angular que possibilita acesso a um elemento HTML da nossa página.

**Linha 10** – temos o atributo "renderer", que é do tipo "Renderer2", recurso do Angular para, neste caso, aplicar um estilo ao elemento HTML.

**Linhas 12 e 13** – É a maneira de aplicar um estilo a um elemento HTML da nossa página, utilizando Diretivas customizadas. Nesse caso estamos aplicando a cor amarela ao fundo do campo.

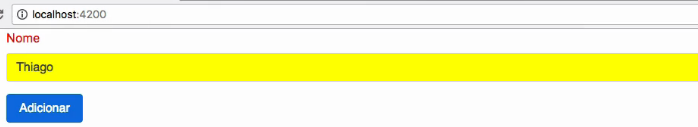
Na **linha 4**, temos o nome do seletor da nossa diretiva (**appCampoColorido**). Este nome deverá ser adicionado ao campo <input> do nosso HTML, que deverá ficar com o fundo amarelo. Este nome deve vir entre colchetes mesmo, no entanto, se a opção for aplicar o fundo amarelo a todos os campos do formulário, você pode colocar o nome "input", como sendo o próprio nome do seletor da nossa diretiva, só que, nesse caso o nome não deverá ficar entre colchetes. Exemplo: **selector: 'input'**. Apesar de funcionar, o Angular recomenda não utilizar dessa forma.

Para testar utilizaremos o arquivo "**funcionario-form.component.html**".

Adicionaremos a diretiva "**appCampoColorido**", ao campo <input> do formulário deste arquivo, da seguinte forma:



O resultado será este:

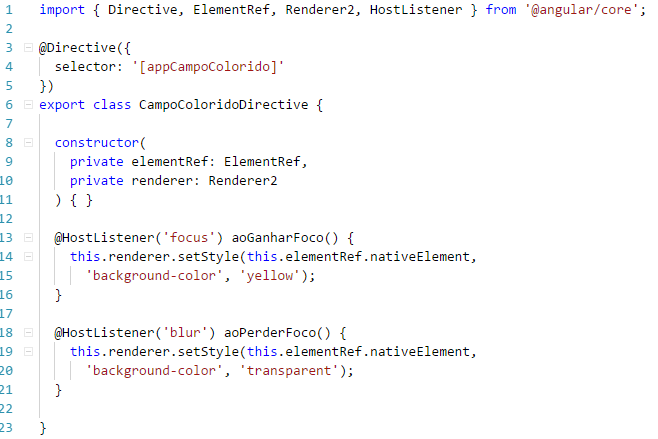


**66. Aplicar o CSS dinamicamente, por evento** *(aula 11.2)*

Agora queremos que a cor amarela seja exibida somente quando o mouse passar sob o campo <input>. Para conseguir este efeito, deveremos implementar na nossa diretiva o recurso @HostListener. Este recurso é utilizado para acionar funções da nossa diretiva a partir de eventos JavaScript, como 'focus', ou 'blur', ou 'click' etc.

Utilizaremos o evento 'focus'. Quando o foco do mouse estiver no campo <input>, ele ficará na cor amarela e quanto o foco sair do campo, ele deverá voltar à cor branca. Para funcionar precisaremos criar duas funções na nossa diretiva, uma para aplicar a cor amarela e outra para retirá-la.

Agora o código da nossa diretiva ficará da seguinte forma:



Linha 8 – o construtor da diretiva agora não tem mais a implementação do estilo para o fundo amarelo. Ele agora está na função "aoGanharFoco" (linhas 13 a 16).

Linha 13 – estamos utilizando o recurso @HostListener, para capturar o momento em que o foco do mouse estiver no campo "<input>".

A função da linha 18 é para retirar a cor amarela, quando o foco sair do campo <input>.

Uma outra forma, até mais simples, seria utilizando a diretiva @HostBinding. Nosso códido então fica assim:



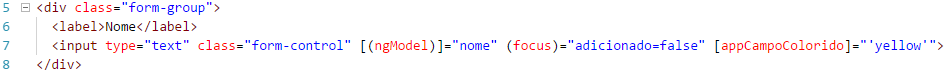
Não temos mais o construtor e nem as classes "ElementRef" e "Renderer2" do Angular. Bastou criar um atributo com o nome "corDeFundo" (linha 8) (poderia ser qualquer nome) e nas linhas 10 e 14 estamos implementando o recurso "@HostListener" para efetuar a alteração da cor de fundo do campo.

Podemos também passar a cor, como um parâmetro da diretiva.

Na linha 8 acrescentaremos uma nova linha de comando, que inicializa com a cor 'gray', que poderá ser outra a partir da cor que definirmos no nosso código html:

@Input('appCampoColorido') cor = 'gray';

Agora no campo <input> do nosso formulário, podemos atribuir a cor a ser exibida.

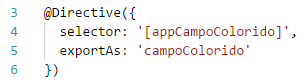


Caso, esteja acusando erro na linha 8 do arquivo "campo-colorido.directive.ts", então será necessário fazer uma alteração de valor no arquivo "tslint.json" – linha 132. O parâmetro "no-input-rename" deverá ter o valor "false"

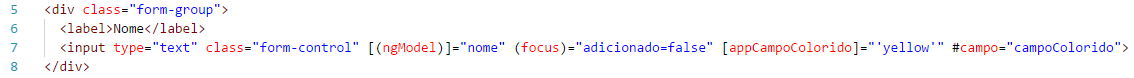
no-input-rename": false

Podemos perceber neste caso, que a diretiva "appCampoColorido" está disponível somente para o campo <input>. Se quiséssemos fazer com que a cor fosse aplicada ao campo, a partir do click em um botão, por exemplo, não teríamos como fazer isso, uma vez que a diretiva não estaria disponível para este botão. O escopo de ação dela está somente para o campo <input>.

A solução para este problema é utilizar o recurso "expostAs" do Angular, que possibilita o acesso externo às funções internas da diretiva.



Na linha 5 temos a implementação deste recurso, ao qual atribuímos o nome de "campoColorido". Este será então o nome que atribuiremos como valor a uma **variável local** no nosso código html, dentro do campo <input> do nosso formulário "funcionario-form.component.html".



Dessa forma esta variável local poderá ser utilizada por outros elementos html, com por exemplo um botão para colorir o campo input, quando clicado.

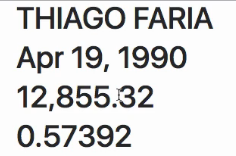
<button class="btn btn-info" (click)="**campo.aoGanharFoco**()">Colorir</button>

Observe que a variável local "campo", que declaramos dentro do campo "<input>", está sendo utilizada agora dentro do elemento <button>, desta forma é possível acessar as funções da diretiva "campoColorido". Veja que a variável "campo" está utilizando a função "aoGanharFoco()". Assim torna-se possível criar funções, como por exemplo, para cálculo, que possibilite serem utilizadas por toda a página.

**67. Conhecendo e usando pipes** (*aula 11.6*)

Pipe é um recurso que possibilita formatar valores de saída, para que sejam melhor apresentados ao usuário. Por exemplo, valores decimais, datas, etc.

Neste exemplo abaixo, estamos utilizando um pipe nativo do Angular para exibir o nome em letras maiúsculas, outros para formatar a data, o preço e o troco:



<div class="container">

<h1>{{ nome | **uppercase** }}</h1>

<h1>{{ dataAniversario | **date** }}</h1>

<h1>{{ preco | **number** }}</h1>

<h1>{{ troco | **currency** }}</h1>

</div>

Mais à frente veremos como melhorar ainda mais a exibição da data e dos valores.

No site "angular.io/", clique no menu "API" e digite "pipe" no filtro de pesquisa, para ver os pipes nativos do Angular. Neste site você poderá ver como formatar os valores para serem exibidos da forma que necessitar.

No caso da formatação da data, poderíamos fazer da seguinte forma:



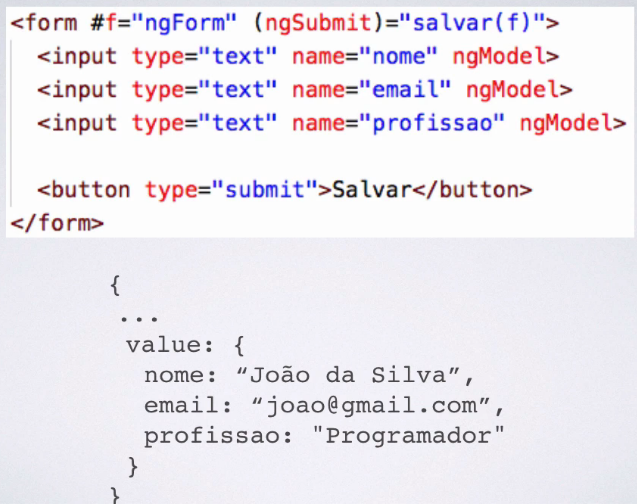
Para exibir o R$, e valor com duas casas decimais, a princípio ficaria assim:



Mais à frente veremos como formatar valores decimais.

**68. Introdução aos formulários** (*aula 12.1*)

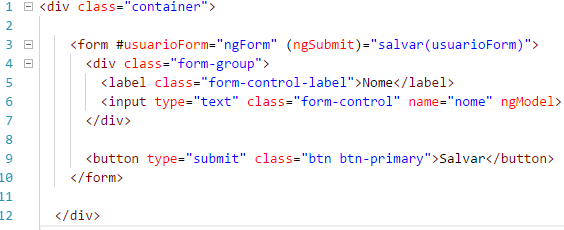
Já conhecemos a maneira tradicional de se trabalhar com formulários em programação web. No Angular, o diferencial é que os dados enviados a partir de um formulário já são tratados como sendo um objeto JavaScript. Na imagem abaixo, é o objeto "value" que terá os valores advindos do formulário (chave/valor).



O módulo de formulários do Angular também possibilita fazer validações de dados das informações digitadas nos formulários, destacando campos de preenchimento obrigatório ou preenchidos incorretamente.

A partir do momento em que no módulo "app.module" exportamos o elemento "formModule" a tag html <form> passa a ser um componente gerenciado pelo Angular e não mais uma simples tag html. No entanto, caso você queira que o seu formulário não seja gerenciado pelo Angular, terá então que incluir na <form> o parâmetro "ngForm", da seguinte forma: <form **ngNoForm**>

**69. Criando formulários** (*aula 12.3*) (utilizando a abordagem Template Driven-forms)



**Linha 3** – criamos uma variável local chamada "usuarioForm". Esta varável está recebendo como valor uma instância da classe "**ngForm**". Essa variável é então passada como parâmetro para o método "salvar()", que por sua vez está associado ao botão "submit". O método "salvar()" deverá estar implementado no módulo principal (app.component.ts), veremos mais à frente.

**Linha 6** – utilizamos a diretriz "**ngModel**" no campo <input> para indicar que o campo "nome" deverá ser um atributo do objeto javascript do formulário, que será gerado após clicarmos no botão submit.

A função "salvar()", deverá ser implementada no módulo principal (app.component.ts), da seguinte forma:



**Linha 1** – importamos o NgForm.

**Linha 11** – A função salvar receber o NgForm como parâmetro. Nele deverá estar o campo "nome" do formulário, como já sendo um atributo de classe.

**Linha 13** – estamos somente exibindo o valor do atributo nome na tela (no console do Browser).

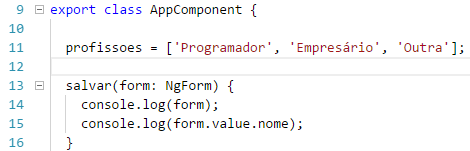
No arquivo "**app.module.ts**" também deverá ser feito o import do **FormModule**.



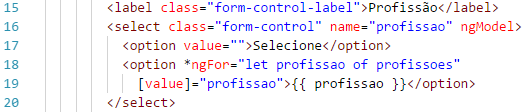
**69. Colocando valores dinâmicos em um combo** (*aula 12.4*)

Ainda não veremos como trazer valores do banco de dados para popular um combo. Mas para mostrar como popular um combo a partir do componente principal. Faremos o seguinte:

Primeiro criaremos um array no componente principal (app.component.ts):



O combo no nosso formulário ficará da seguinte forma:



**Linha 18** – em \*ngFor = "let profissão of **profissoes**", estamos implementando um "loop", onde a variável "profissoes" está se referindo ao array "profissoes" que criamos no componente principal.

**Linha 19** – estamos fazendo a interpolação para exibir os valores advindos do array.

**70. Validando campos do formulário** (*aula 12.7*)

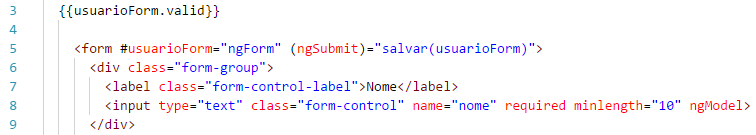
A ideia é que o botão de submissão do formulário somente fique ativo após todos os campos obrigatórios terem sido preenchidos corretamente.

O objeto gerado pelo formulário no Angular, possui um atributo chamado "valid", que recebe o valor "false" se o formulário for considerado "inválido" e valor "true" se válido. Portanto no caso de campos obrigatórios não preenchidos, o atributo "valid" do objeto do formulário, terá o valor "false". Este atributo pode ser visualizado no console do browser após submeter os dados de um formulário.

Vejamos um exemplo:

Se na linha 3 implementarmos a exibição do valor do atributo "valid" e na linha 8 implementarmos o "required minlength='10'", perceberemos que no browser aparecerá o valor "false" no topo da nossa página, mas, ao digitarmos o nome da pessoa, com mais de 10 caracteres, o valor exibido no topo da página passa a ser "true".

Campo de preenchimento obrigatório e que deverá ter pelo menos 10 caracteres.



Para validar um endereço de e-mail, basta colocar assim:



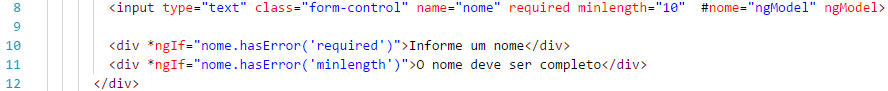
Para desabilitar o botão de "salvar", de forma que ele somente fique habilitado após todos os campos obrigatórios estarem preenchidos corretamente, é necessário colocar a seguinte instrução no botão "salvar":



Para ler sobre as diretivas de validação consulte:

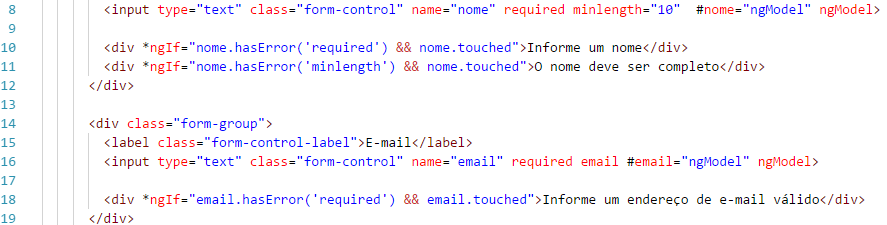
https://angular.io/api?query=valid&type=directive

**71. Exibindo mensagens de validação nos campos do formulário** (*aula 12.9*)



No caso do campo "nome", incluímos na linha 8, uma nova variável local, para referenciar o atributo "nome" do nosso formulário. Esta variável local é então utilizada nas linhas 10 e 11 para fazer a verificação de preenchimento obrigatório e se tem mais de 10 caracteres.

Observe que as mensagens ficam visíveis já na exibição da tela do formulário. O ideal é que apareçam somente após clicar no campo. Para este caso, existe um recurso chamado "**touched**". Se o incluirmos nas linhas 10 e 11, as mensagens somente serão exibidas após o usuário clicar no campo.



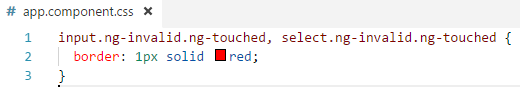
Outra forma é exibir a mensagem somente se o usuário, após clicar no campo, digitar alguma coisa nele. Para utilizar dessa forma ao invés de "touched", utilize "**dirty**" ou "**pristine**".



Este recurso poderá ser utilizado também na tag <form> para exibir mensagens quando o usuário clicar no formulário ou fazer alguma alteração.

**72. Como exibir borda vermelha na validação dos campos do formulário** (*aula 12.11*)

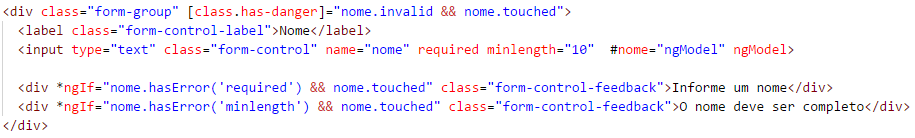
Este recurso CSS poderá ser implementado somente para o formulário que estamos trabalhando ou para toda a aplicação. Se somente para o formulário, a implementação deverá ser feita no arquivo "**app.component.css**", se para toda a aplicação, o arquivo deverá ser o "**styles.css**".



Este efeito também pode ser implementado utilizando a classe CSS "has-danger" do Bootstrap nos campos do formulário e também a classe "form-control-feedback" nas mensagens de erro a serem exibidas .

A classe "has-danger" deverá ficar na <div> que agrupa o campo do formulário com as mensagens de erro.

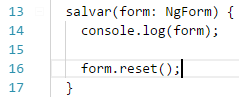
Para testar este recurso, primeiramente apague o CSS que foi colocado no arquivo "app.components.css".



Para funcionar, a versão do Bootstrap deverá ser 4.0.0 ou superior.

**73. Limpar os campos do formulário** (*aula 12.13*)

Basta incluir no método salvar(), que está no componente principal (app.component.ts) a instrução "form.reset()".



Nesse caso, o que ocorre é que o campo "profissão", que é um combo, não fica com o seu valor padrão que é o termo "selecione". Ele fica em branco.

Para que ele fique com o valor "selecione" após clicar no botão salvar, acrescente um parâmetro à função reset(), da seguinte forma:



Se quiser manter o valor padrão de mais algum campo, basta incluí-lo dentro das chaves e separados por vírgula, como parâmetro nesta função.

Agora poderemos voltar para o nosso projeto inicial e criar o formulário de lançamentos.

Na pasta que está este arquivo, coloquei também os arquivos com os códigos html.

**Observações**

* Para auxiliar no desenvolvimento utilizando o Visual Studio Code, recomenda-se instalar a extensão TSLint for Visual Studio. Basta clicar no ícone de extensões do Visual Studio, digitar este nome e clicar em instalar. Esta extensão orienta sobre erros no código.
* Instale também o "auto import". Ele baixa automaticamente as dependências que precisar utilizar no seu projeto.

1. **Componente Calendário**

O PrimeNG tem o componente DatePicker, que precisa do módulo CalendarModule. Acrescente o import no arquivo app.module.ts e no array de imports deste mesmo arquivo.

import {InputTextareaModule} from 'primeng/primeng';

Este recurso, para funcionar, precisa também importar o BrowserAnimationModule.

O campo input, ficará da seguinte forma: <p-calendar dateFormat="dd/mm/yy"></p-calendar>

No caso de aparecerem mensagens de erro como estas abaixo:

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@^1.0.0 (node\_modules\chokidar\node\_modules\fsevents):  
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

Instale o 'fsevents', conforme commando abaixo:

$ npm install fsevents@^1.0.0 –force

1. **Máscara para valores financeiros**

Até o momento, o PrimeNG não tem o recurso específico, então poderá ser utilizado instalando o componente abaixo:

npm install ng2-currency-mask --save

Necessário acrescentar o import no arquivo "app.module.ts":

import {CurrencyMaskModule} from 'ng2-currency-mask';

1. **Como criar um componente com apenas um único arquivo**

ng g c [nome do componente] --spec=false --inline-template --inline-style

Dessa forma o Angular não criará o arquivo de template e nem o arquivo de css. Este recurso pode ser utilizado quando precisar criar um componente simples, que não tem muito código.

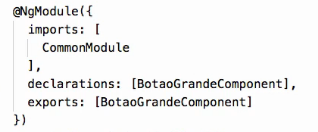
1. **O que é um módulo no Angular, como compartilhar elementos entre módulos?**

Toda aplicação Angular, tem pelo menos um módulo, que é o módulo raiz (root-module), que normalmente recebe o nome de "**app-module**".

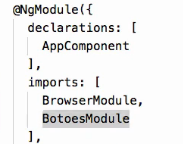
À medida que o projeto de um sistema vai crescendo, vai ficando cada vez mais difícil administrar a quantidade de componentes, diretivas, pipes etc. O recurso de módulos no Angular é uma forma de organizar estas diretivas, componentes e pipes que são relacionados.

No caso do nosso projeto "Lançamentos", podemos ter um módulo de lançamento e um de pessoa, como também um módulo "utilitário" que conterá todos os recursos que poderão serem utilizados por diversos outros módulos.

Opcionalmente os módulos podem exportar seus componentes para poderem ser utilizados por outros módulos. Exemplo: Se precisarmos utilizar um componente do módulo 1 dentro do módulo 2, será necessário que no módulo 1 utilizemos o recurso "export" no determinado componente, enquanto que no módulo 2 precisaremos utilizar o recurso "import" para este determinado módulo.

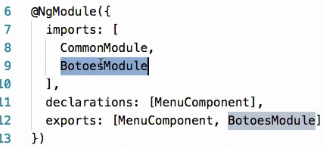


Exemplo de exportação de um componente do módulo.



Exemplo de importação de um componente do módulo.

Elementos que são importados de outros módulos também podem ser exportados. Utiliza-se para isso o termo "reexportar"



Para criar um módulo digite: ng g m [nome do modulo]

Uma pasta será criada e dentro dela um arquivo com o nome do módulo com extensão .ts

Para criar um novo componente já dentro de um módulo específico digite:

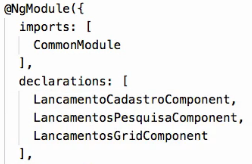
ng g m [nome do módulo] / [nome do componente]

Dentro da pasta do módulo serão então criada uma nova pasta para o novo componente. Este novo componente então estará automaticamente já declarado no arquivo do módulo, pelo próprio Angular.

1. **Organizando as funcionalidades em módulos**

Ao criar um módulo, mova os arquivos das funcionalidades já criadas para dentro da pasta do novo módulo.

Declare estas funcionalidades no arquivo do novo módulo e retire a declaração deles e os imports do módulo anterior em que estavam, no caso o "app.module" (módulo principal).

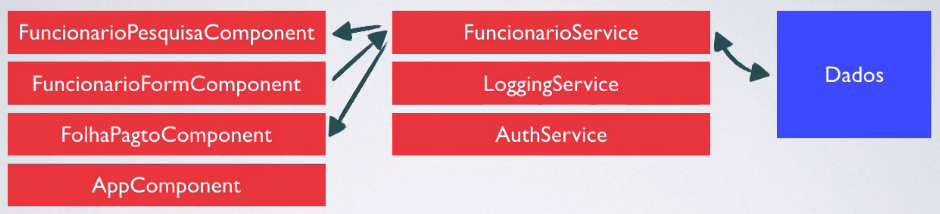


Exemplo de declaração de componentes dentro de um módulo.

1. **Criando o Shared Module**

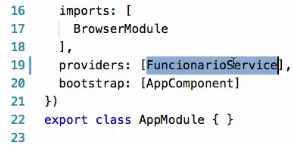
São módulo que não possuem funcionalidades. Server apenas para compartilhamento de módulos.

**74. Introdução aos serviços**



Os serviços são os acessos a funcionalidades externas à nossa aplicação Frontend, que pode ser feito por meio de REST.

A melhor maneira de implementar a acesso a serviços é utilizando injeção de dependências. A maneira proporcionar este recurso no Angular é informando no seu módulo principal, a classe de serviço a ser injetada.



O "providers" serve para informar as classes de serviço a serem injetadas. É como passar a classe "FuncionarioService" como parâmetro para o construtor da classe "app.component" do Angular.

O providers também pode ser da seguinte forma, que é a mesma coisa:



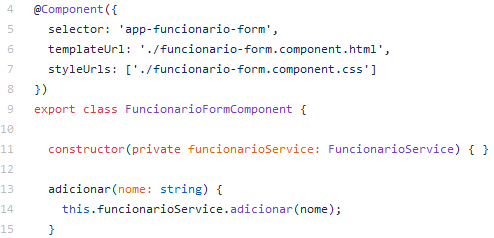
Ou desta outra forma, em que ao invés do "useClass", foi implementado uma função, que instancia a classe "FuncionarioAbreviadoService". Esse modo é para quando for necessário passar parâmetros para o construtor.





Outra forma de passar parâmetros para o construtor é por meio da implementação de um INJETOR.

Todos os componentes passam a ter acesso a uma mesma instância da classe FuncionarioService. Isso é injeção de dependência.



**74. Conversão de formato de datas** (aula 17.4)

Foi utilizada a biblioteca Moment.js (https://momentjs.com). A instalação é simples:

npm install moment --save

Na aplicação, o import desta biblioteca ficará da seguinte forma:

Import \* as moment from 'moment';

No código de um componente ou serviço, a implementação ficará da seguinte forma:

Moment(dataVencimento).format(YYYY-MM-DD);

Na documentação desta biblioteca estão os formatos possíveis.

**75. Como implementar requisições Http utilizando Angular** (aula 17.2).

Iniciamos criando uma classe de serviço para implementação deste recurso.

ng g s lançamentos/lançamento --spec=false

A classe será criada dentro da pasta "lançamentos". Lembrando que esta classe de serviço deverá estar declarada nos provides do app.module.

Nesta classe é necessário importar um serviço de Http, no app.module: HttpModule

É necessário importar também o operador "toPromisse":



Linha 4 – import do toPromise.

Linha 11 – injeção do serviço de Http.

Linha 13 – definindo o retorno da pesquisa para o formato Promise.

Linhas 14 e 15 - No backend, já implementamos um recurso de segurança de acesso. Por isso nos testes com o POSTMAN foi necessário informar no cabeçalho o "Authorization", com usuário e senha na base 64. Basta copiar a string criptografada que geramos lá no POSTMAN e colar aqui.

Observe que na linha 17 estamos passando o headers já com estas informações de acesso.

Posteriormente esta implementação será alterada para utilizarmos uma outra forma de autenticação.

Linha 17 – utilizamos a URL da nossa API e passamos como parâmetro o 'resumo', que, conforme implementamos na nossa API, ela retorna um resumo dos lançamentos. Observe o uso de ( ` ) para informar a URL como sendo um "template literal"

Linha 19 – o ".then" serve para estabelecer que os próximos comandos somente serão executados após a conclusão do processo da requisição GET. Nesse caso o conteúdo da resposta está sendo retornado no formato JSON.

Criaremos agora o componente de pesquisa de lançamentos (lancamento-pesquisa.component.ts),



Linha 13 – injeção do serviço de lançamentos no construtor. A variável "private lancamentoService" é a variável local desta classe ("LancamentoPesquisaComponent").

Linha 15 – O ngOnInit() é para garantir que o método "pesquisar()" somente será executado após a execução dos comando que forem implementados no construtor. No caso a instância da classe LancamentoService.

Linha 21 – recebe a resposta de "LancamentoService" e atribui o conteúdo ao array de lançamentos.

**76. Comunicação entre o backend e o frontend**. (*aula 17.1*)

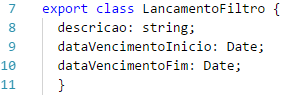
Primeiramente é necessário gerar um arquivo JAR da API que desenvolvemos com SpringBoot. Faremos isso já passando o "username" e a senha de acesso ao banco de dados. É necessário também informar qual a porta em que está rodando o frontend. Com este comando, iniciamos o servidor e geramos o JAR:





**77. implementando pesquisa com parâmetros no backend a partir do frontend**.

Dentro do arquivo da classe de serviço "lançamento.service.ts", implementaremos uma outra classe que será o objeto a ser enviado para o backend, contendo os parâmetros de busca e seus respectivos valores (linhas 7 a 11).





Linha 21 – a classe "URLSearchParams()" se encarregará de passar os parâmetros para a API.

Linhas 26 a 38 – "setando" em "params", os valores a serem pesquisados.

Linhas 32 e 37 – conversão do formato de data, para o formato compatível com a API.

Linha 41 – passando para a API, via URL, o header, com os parâmetros de pesquisa (search: params).

O componente de pesquisa de lançamentos (LancamentoPesquisa), agora ficou da seguinte forma:



Linhas 12 a 14 – variáveis locais que receberão valor por meio da página HTML (template).

Linhas 24 a 27 – Utilizando a classe LancamentoFiltro, criada no arquivo "lançamento.service", para atribuir os valores oriundos da página HTML

O arquivo (template) "lançamentos-pesquisa.component.html", terá um formulário:

<form autocomplete="off" (ngSubmit)="pesquisar()">

O <input> do campo "Descrição", ficará da seguinte forma:

<input pInputText type="text" name="descricao" [(ngModel)]="descricao">

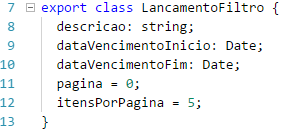
E os dois campos de datas deverão estar neste formato:

<p-calendar dateFormat="dd/mm/yy" [inputStyle]="{ width: '95px' }"

name="vencimentoInicio" [(ngModel)]="dataVencimentoInicio"></p-calendar>

**78. implementando paginação a partir do backend**. (*aula 17.5*)

Na nossa classe "LancamentoFiltro" acrescentamos mais dois parâmetros: pagina e itensPorPagina.



Na classe "LancamentoService", acrescentamos estas duas novas variáveis ao nosso atributo "params":



**Lancamento-pesquisa.component.ts** agora fica da seguinte forma:



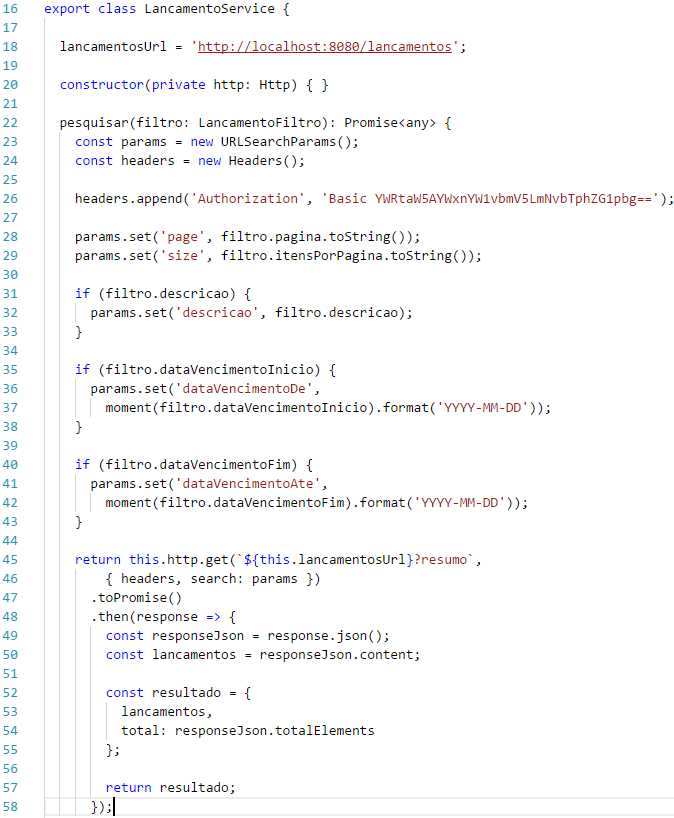
Aqui não precisaremos mais chamar o método pesquisar. Ele agora será acionado pelo recurso Lazy implementado na tabela que lista os lançamentos.

Os campos no arquivo (template) "lançamentos-pesquisa.component.html", agora referenciarão as variáveis da seguinte forma:

O <input> do campo "Descrição", ficará da seguinte forma:

<input pInputText type="text" name="descricao" [(ngModel)]="**filtro.**descricao">

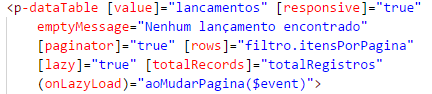
A classe "LancamentoService" agora tem novos atributos para controle da paginação:



Linha 54 – está pegando o valor do total de elementos trazidos do banco de dados, para fazer o cálculo da quantidade de páginas que teremos na paginação. Este valor total foi implementado na nossa API, com SpringBoot.

Linha 57 – o objeto "resultado" que está sendo retornado é o que está sendo capturado nas linhas 26 e 27 de "lancamento-pesquisa.component.ts".

Na tabela que lista os lançamentos, que está no arquivo "lancamentos-pesquisa.component.html", teremos que implementar a paginação "Lazy". Quer dizer que agora os registros não serão trazidos todos de uma só vez, mas sim à medida que o usuário for clicando nas setas de paginação.

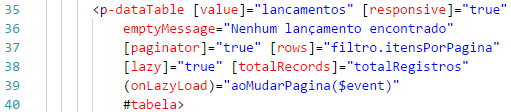


O "onLazyLoad" acionará o método "aoMudarPagina()" sempre que o usuário clicar nas setas de paginação.

**79. Implementando exclusão de lançamentos**. (*aula 17.8*)

Implementar a função de exclusão é simples, o que é um pouco mais complexo é a atualização da tabela que lista os lançamentos.

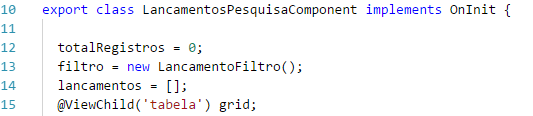
No cabeçalho da tabela acrescentaremos a variável **#tabela**.



Na coluna de ações acrescentamos o "let-lanc='rowData'", para capturar o objeto "lançamento". Na linha 63 estamos passando este objeto para a função "excluir()".



Em "lançamento-pesquisa.component.ts", acrescentamos a variável que referenciará a variável #tabela que criamos na página html. O nome dela será "grid", podendo ser qualquer outro nome.

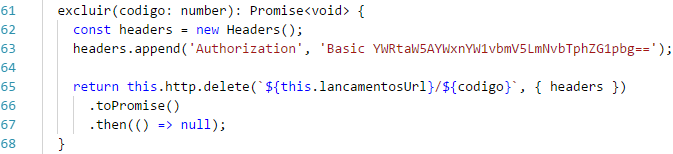


Neste mesmo arquivo, implementamos a função de exclusão:



A linha 40 faz a atualização da lista de lançamentos, chamando o método "pesquisar()". A função de pesquisa é acionada automaticamente e posiciona a tabela na página 1.

Em "lançamento.service.ts", a função de exclusão fica da seguinte forma:



Fica faltando agora implementar a mensagem de "exclusão" e a confirmação de exlcusão.

**79. Implementando a mensagem de exclusão de lançamentos**. (*aula 17.9*)

Foi utilizado o recurso "ng2-toasty", no entanto, atualmente já é possível implementar utilizando o recurso "growl" do próprio "primeng".

Se ainda assim, optar por utilizar o "ng2-toasty", basta instalar por meio da console:

npm install ng2-toasty --save

No app.module.ts faça a importação do toasty: 

No array de imports inclua: ToastyModule.forRoot(),

Utilize o "forRoot", somente se a importação for implementada dentro do app.module.

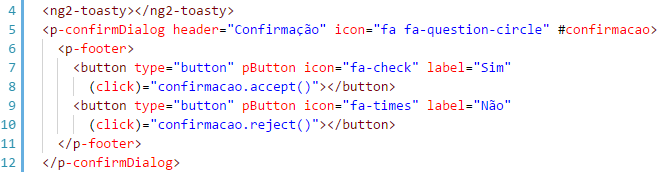
No arquvo "app.component.ts", podemos configurar o tema do nosso "toast", da seguinte forma:



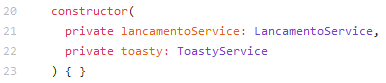
No arquivo "angular-cli.jason" é necessário acrescentar o arquivo CSS do "Toasty". Na propriedade "styles" deste arquivo adicione a seguinte linha:



No arquivo app.component.html (template principal), precisamos adicionar as seguintes linhas:

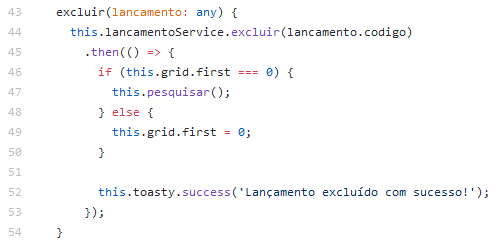


No construtor de "lançamento-pesquisa.component.ts", adicionamos a variável "toasty":



A classe ToastyService precisará ser importada aqui.

Na função de exclusão que temos neste arquivo acrescentaremos a mensagem de exclusão, da seguinte forma:



**80. Implementando a mensagem de confirmação de exclusão de lançamentos**. (*aula 17.10*)

Utilizaremos o componente "confirmDialog" do PrimeNG.

Em app.module.ts faremos a importação do "ConfirmDialog":

import { ConfirmDialogModule } from 'primeng/components/confirmdialog/confirmdialog';

No array de imports, incluímos o módulo: ConfirmDialogModule

No array de providers, precisamos incluir o serviço "ConfirmationService"

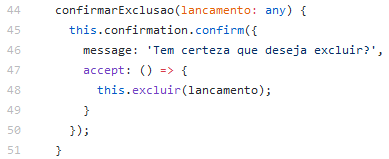
Que terá o seguinte import:

import { ConfirmationService } from 'primeng/components/common/api';

No constructor "lancamentos-pesquisa.component.ts", acrescentamos a variável local "confirmation"



Agora implementamos neste arquivo, a função de confirmação de exclusão:

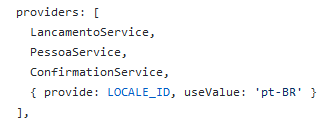


**81 – Configurando o Locale da aplicação para pt-BR**

No arquivo app.module.ts, adicione um provider:

import { NgModule, LOCALE\_ID } from '@angular/core';

(…)



A partir deste novo provider, os números decimais passarão a serem exibidos com vírgula ao invés de ponto.

**82 – Criando um serviço de tratamento de erros** (*aula 17.12*)

Caso ocorra algum erro de acesso ao backend (pode estar fora do ar no momento), ou caso o backend retorne alguma mensagem de erro, ela precisará ser tratada, para que possa ser exibida ao usuário, de forma mais amigável.

Para criar o serviço, digite o seguinte comando no console:

ng g s core/error-handler --spec=false

O modulo de serviço será criado na pasta "core" e no "app.module.ts" já poderemos vê-lo na lista de imports. - Agora precisamos adicionar o "provider" deste serviço no arquivo "core.module.ts".

import { ErrorHandlerService } from './error-handler.service';

(…)

providers: [

ErrorHandlerService

]

Alguns dos "imports" que estão no "app.module.ts", poderão serem migradas para o "core.module.ts", para melhor organização.

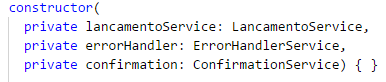


Agora, para implementarmos o tratamento de mensagem de erro na pesquisa de lançamentos, poderemos fazer da seguinte forma:

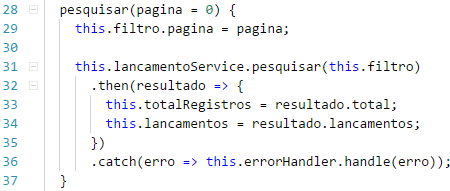
No arquivo " lancamentos-pesquisa.component.ts", import o "ErrorHandlerService"

import { ErrorHandlerService } from './../../core/error-handler.service';

Acrescentá-lo no construtor:



E agora, no método "pesquisar()", adicionaremos um "catch" com a implementação do tratamento de mensagem de erro:



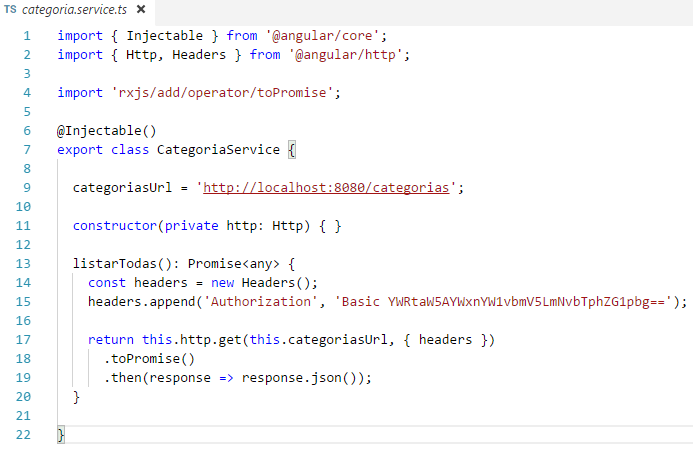
O mesmo poderá ser feito com as outras funcionalidades/métodos.

**83 - Listando as categorias cadastradas no dropdown** (*aula 17.17*)

No formulário de cadastro de lançamentos, temos um combo que lista as categorias a partir de um array de categorias que criamos em "lancamento.cadastro.component.ts". Agora precisamos buscar as categorias no Banco.

Primeiro precisamos criar o serviço de categorias, da mesma forma como fizemos com "lancamento.service", só que, para este caso, não precisaremos dos recursos de paginação.

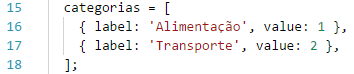
Basta criar a pasta "categorias" e dentro dela o arquivo "categoria.service.ts". O conteúdo deste arquivo deverá ser o seguinte:



Veja que é bem parecido com o "lancamento.service".

Precisamos agora, zerar nosso array de categorias, que está em em "lancamento.cadastro.component.ts".

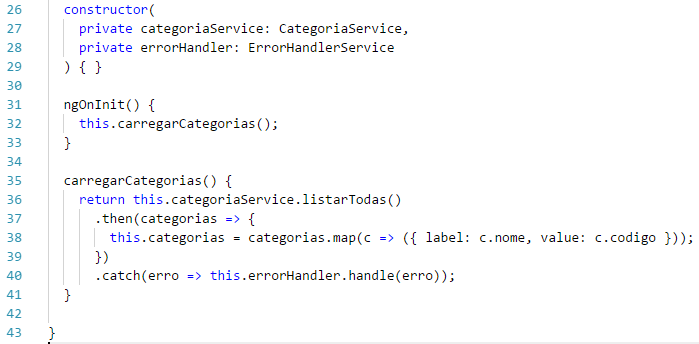
Estava assim:



Agora deverá ficar assim:

Categorias = [];

O construtor de "lancamento.cadastro.component.ts", deverá ficar da seguinte forma:



Lembre-se de fazer os imports de "categoriaService" e "errorHandler".

**Linha 38** – O recurso "map" percorre todo o conteúdo do array recuperado do banco de dados e coloca-o no formato padrão que o Angular utiliza para conseguir popular o combo de categorias no formulário. O "map" cria um novo array contendo as propriedades "label" e "value".

**Nota**: O símbolo "=>" (conhecido como "arrow function"), é uma forma abreviada de criarmos uma função, que retorna um valor para uma variável, no caso a variável "categorias".

O mesmo deverá ser feito para o combo que lista as pessoas no formulário de cadastro de lançamentos.

**84 – Implementando o serviço de cadastro de lançamentos** (*aula 17.20*)

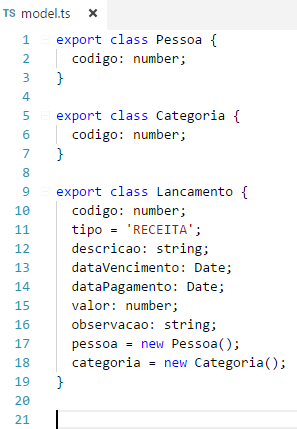
Para cadastrar um novo lançamento por meio do nosso backend, basta fazermos uma requisição do tipo POST, pela URL <http://localhost:8080/lancamentos>, em que no corpo da requisição, tenhamos o conteúdo, em formato JSON, do lançamento a ser gravado.

 Exemplo:

Este é o fomato que implementamos no nosso backend.

Inicialmente, precisaremos criar uma classe de modelo, que represente o objeto a ser enviado para o backend para ser gravado no BD.

Criaremos então dentro da pasta "core" um arquivo que conterá as classes de modelo que precisaremos utilizar. Ele deverá ficar inicialmente da seguinte forma:



Veja que os nomes das propriedades de lançamento são os mesmos que estão no exemplo do formato JSON. Se não forem os mesmos nomes, poderá ocorrer erro.

Em "lancamento-cadastro.component.ts", criaremos o atributo que instanciará a classe "Lancamento" que acabamos de criar.

lancamento = new Lancamento();

Agora, é necessário fazer o "binding" dos atributos do formulário com o atributo "lançamento" do componente de lançamentos. No formulário, pegando como exemplo o campo "tipo lançamento", acrescentaremos a este campo o "binding" de lançamentos:



Necessário acrescentá-lo a todos os campos do formulário, fazendo as devidas correspondências de campos.

A tag "form" do formulário de cadastro de lançamento, ficará da seguinte forma:

<form #f="ngForm" autocomplete="off" (ngSubmit)="salvar(f)">

A variável "f" é a instância do formulário, que é passada como parâmetro para o método salvar().

Em "lancamento-cadastro.component.ts" teremos o método "salvar()", que por enquanto somente imprimirá o conteúdo do formulário na console, para que possamos verificar se tudo está correndo bem.

import { FormControl } from '@angular/forms';

(…)

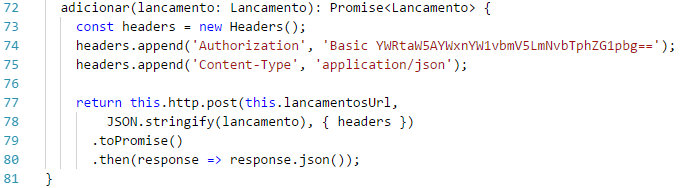


O próximo passo é criar o método "adicionar()", em "lancamento.service.ts". Este método deverá retornar o próprio lançamento que será gravado no BD, conforme foi implementado no backend.

Este método terá que enviar para o backend, o lancamento em formato JSON.

import { Lancamento } from './../core/model';

(...)



Linha 75 – necessária para informar que o objeto enviado está no formato JSON.

Linha 78 – convertendo o objeto "lancamento" para JSON.

Em "lancamento-cadastro.component.ts" agora teremos o nosso método "salvar()", da seguinte forma:

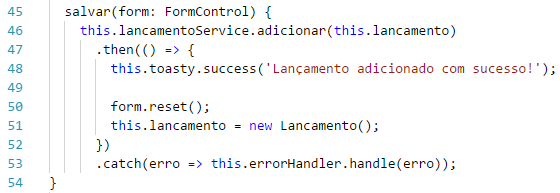
import { LancamentoService } from './../lancamento.service';

import { ToastyService } from 'ng2-toasty';

(...)

No **construtor** incluir:

private toasty: ToastyService,



Linha 50 – com aquele parâmetro "f" que informamos na tag "<form>", conseguimos limpar o formulário.

**Nota**: o SpringBoot que utilizamos para desenvolver o backend, pega o objeto em formato JSON e converte para um objeto em formato Java, para então, gravar os dados no banco.

Repita esta mesma lógica para o cadastro de pessoas.

**85 – Configurando a rotas e navegação** (aula 18.1)

Como implementar a navegação entre as telas do sistema? Essa navegação é feita com implementações/configurações do Angular.

Quando a nossa url for "http://localhost/lancamento", o sistema deverá exibir a tela de consulta de lançamentos e quando for ".../pessoa", deverá exibir a tela de consulta de pessoas.

Quando for ".../lancamento/novo", exibirá a tela de cadastro de lançamentos. O mesmo para "pessoas".

Quando for ".../lancamento/22", por exemplo, abrirá a tela de edição do lançamento número 22.

Estas URLs precisam ser configuradas no Angular.

O elemento do Angular que gerencia estas rotas é a diretiva chamada "<router-outlet>".

Começamos criando um array no arquivo "app.modules.ts", que poderemos chamar de "routes", onde armazenaremos todos os caminhos (URLs) que nossa aplicação fará uso. Ficará da seguinte forma:

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

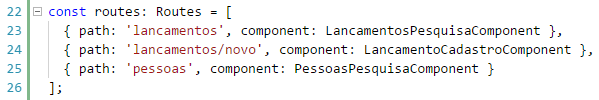
(…)

import { PessoasPesquisaComponent } from './pessoas/pessoas-pesquisa/pessoas-pesquisa.component';

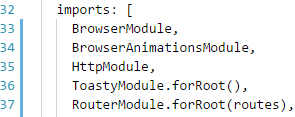
import { LancamentoCadastroComponent } from './lancamentos/lancamento-cadastro/lancamento-cadastro.component';

import { LancamentosPesquisaComponent } from './lancamentos/lancamentos-pesquisa/lancamentos-pesquisa.component';

(...)



Acrescente o "RouterModule" no array de imports do app.module.



Perceba que o "RouterModule", recebe como parâmetro a variável "router" que declaramos na linha 22. A partir daí o módulo de rotas do Angular, passa a saber quais são nossas rotas de navegação.

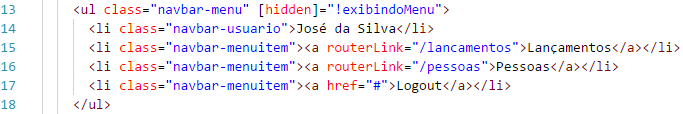
Agora, no arquivo "app.component.html" precisamos retirar a definição do template que estava fixada na linha 2.



Agora passará a ser da seguinte forma:



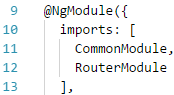
Agora precisamos configurar os links no nosso componente NAVBAR. Ao invés de utilizarmos a tag "<a href", utilizaremos a diretiva do Angular "routerLink". Esta diretiva evita ter que recarregar todos os arquivos javascripts, CSS e demais arquivos relacionados à tela, toda vez que um novo link for clicado.



Para o "routerLink" funcionar corretamente, é necessário ainda, importar o "RouterModule" no arquivo "core.module.ts", uma vez que o nosso NAVBAR está dentro do módulo "core".

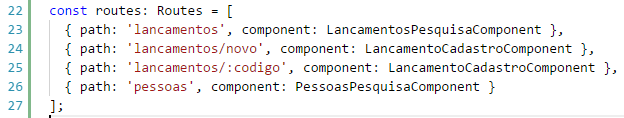
import { RouterModule } from '@angular/router';

(…)



**86 – Configurando a rotas com parâmetros** (aula 18.5)

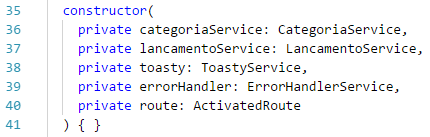
Quando o código de um lançamento, por exemplo, é informado via URL, a aplicação deverá abrir a tela de edição do referido lançamento. Para tanto é necessário configurar na lista de URL do array que criamos anteriormente no arquivo "app.module.ts", uma URL que contenha um token, da seguinte forma:



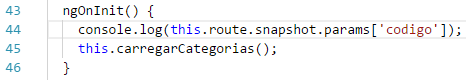
Agora, em "lancamento-cadastro.component.ts", precisamos implementar uma forma de capturar o código do lançamento informado na URL. Necessário utilizar o **ActivateRoute**:

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

(…)



(…)



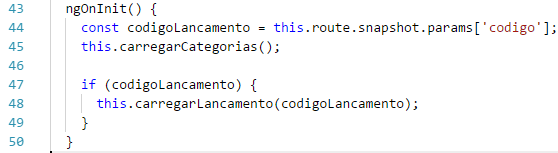
Aqui, ele consegue capturar o valor do código de lançamento, informado pela URL e exibir na console.

Em "**lancamento-pesquisa.components.html**", precisamos configurar o **routerLink** no botão "Editar", para conseguir passar o código do lançamento clicado.

(...)

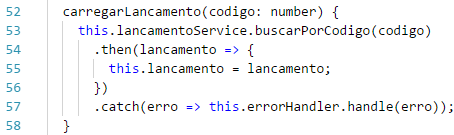
**87 – Preenchendo os campos do formulário para edição** (aula 18.7)

Em "lancamento-cadastro.component.ts", ao invés de exibir o código na console, armazenaremos o valor numa variável.

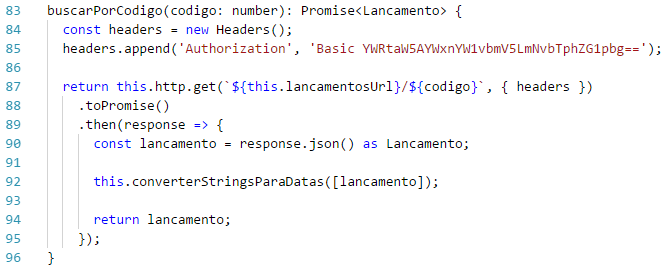


**Linha 47** – Se um valor for passado via URL, carregar lançamento.

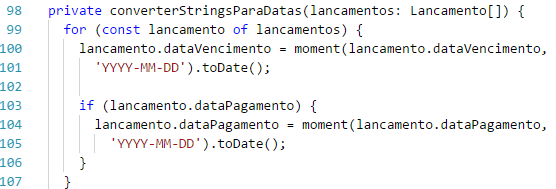
(...)



Linha 53 – Necessário implementar o "buscarPorCodigo" em **Lancamento.Service**.



Ele utiliza um método para converter a data :



**88 – Salvando lançamento editado** (aula 18.8)

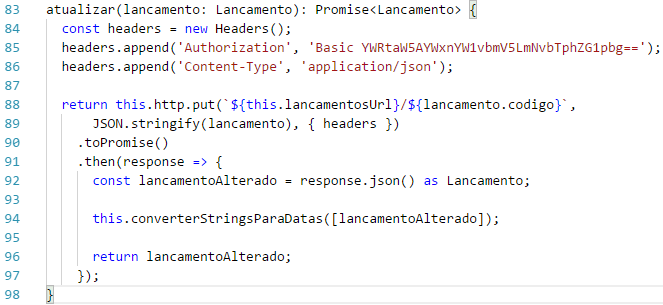
Em "**lancamento-cadastro.component.ts**" precisamos agora verificar se estamos editando ou criando um novo lançamento. O primeiro passo é criar o método abaixo:



O método salvar agora passa a verificar se estamos no modo de edição ou criando um novo lançamento.



Em "**lancamento.service.ts**" criamos o método "atualizar()":



**89 – Redirecionamento ou navegação imperativa** (aula 18.9)

Se após gravar um lançamento, for necessário redirecionar o usuário para a tela que lista os lançamentos, basta implementar o "router" no método adicionarLancamento()", do arquivo "lancamento-cadastro.component.ts", da seguinte forma:

|  |
| --- |
| import { ActivatedRoute, Router } from '@angular/router'; |
| (…) |

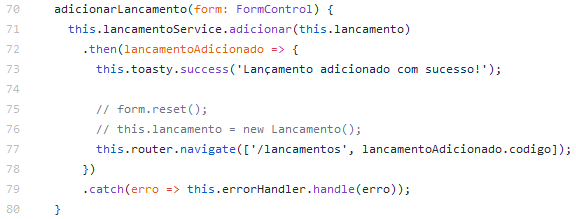
adicionarLancamento(form: FormControl){

(...)

This.router.navigate(['/lancamentos'])

}

Caso seja necessário direcionar o usuário para a tela edição, logo após efetuar o cadastro de um lançamento, o código do método "adicionarLancamento()", ficaria da seguinte forma:



**Linha 72** – Sempre após gravar um novo Lançamento, o método do serviço retorna o Lançamento que foi gravado. Nesse caso, estamos armazenando na variável "lancamentoAdicionado", o Lançamento que acabou de ser gravado.

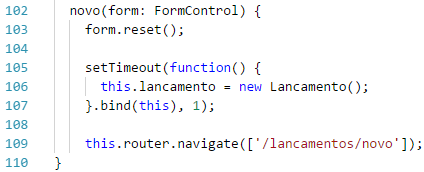
**Linha 77** – Estamos redirecionando o usuário para a tela de edição já preenchendo o formulário com os dados do Lançamento que acabou de ser gravado.

**- Botão Novo Lançamento**

O botão "novo" da tela de Lançamentos, precisa direcionar o Usuário para a tela de cadastro de novo lançamento. Acrescente a este botão o comando "(click)='novo(f)'"



Em "lancamento-cadastro.component.ts", implemente o método "novo()", da seguinte forma:



Lembrando que é necessário fazer o import do Router e declará-lo no construtor, da seguinte forma:

private router: Router

**Linha 105** – foi necessário implementar o setTimeout para fazer com que o "this.lancamento = new Lancamento()", fosse executado somente após um milésimo de segundo (observe o número 1 na linha 107).

**Linha 107** – temos um "bind(this)". Este comando serve para fazer com que o "this.lancamento" seja reconhecido como uma variável de scopo de "lancamento-cadastro.component.ts". Uma vez que está sendo utilizada dentro da função setTimeout(), ela somente será reconhecida como sendo do scopo da classe, se utilizarmos o bind().

**90 – Redirecionamento da página inicial** (aula 18.10)

Se digitarmos somente "http:localhost", o sistema atualmente não mostra nada. Poderemos definir a tela de lançamentos como sendo a página inicial, adicionando uma nova rota com direcionamento, no array de rotas que criamos em "app.module.ts", da seguinte forma:

{ path: '', redirectTo: 'lancamentos', pathMatch: 'full' },

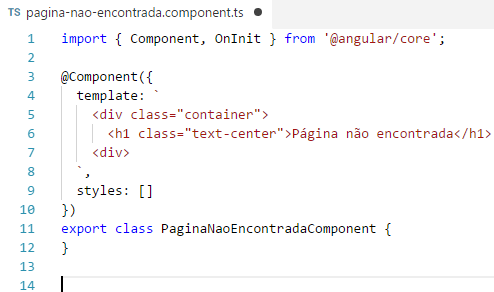
**90 – Tratando páginas não encontradas (rotas inexistentes)**(aula 18.11)

Necessário criar um componente para "página não encontrada". Criaremos este componente dentro da pasta "core" digitando o comando abaixo no console:

ng g c core/pagina-nao-encontrada --inline-style --inline-template --flat --spec=false

O nome do arquivo será "pagina-nao-encontrada". Não será criado um arquivo de estilo CSS e nem de template. O parâmetro "--flat" é para evitar criar uma nova pasta para este arquivo.

O conteúdo deste arquivo será o seguinte:



Em "app.module.ts" criaremos duas novas rotas no array de rotas:

{ path: 'pagina-nao-encontrada', component: PaginaNaoEncontradaComponent },

{ path: '\*\*', redirectTo: 'pagina-nao-encontrada' }

A segunda rota é para redirecionar qualquer outra coisa que seja digitada na URL que não seja uma URL real para uma página do sistema.

Necessário o seguinte import neste arquivo:

import { PaginaNaoEncontradaComponent } from './core/pagina-nao-encontrada.component';

**91 – Refatorando o array de rotas** (aula 18.13)

Se sua aplicação tiver muitas rotas, o array de rotas tende a ficar muito extenso, acabando por deixar poluído o código do arquivo "app.module.ts". A solução para este problema é criar um módulo somente para configuração de rotas**.** Chamaremos este módulo de "routing module", que é uma convenção dos desenvolvedores Angular.

Começaremos então, criando o arquivo "app-routing.module.ts" na pasta "app". Dessa forma ele será reconhecido como sendo o módulo de roteamento do app-module.

O conteúdo deste arquivo, extraído do "app.module.ts", será o seguinte:



Agora, já podemos apagar o array de rotas do arquivo "app.module.ts". E, neste arquivo, retiraremos também o import do RouterModule. Agora faremos nele, o import deste módulo que acabamos de criar.

O arquivo "app.module.ts" ficou agora com o seguinte conteúdo:



Cont...



* Podemos também criar módulos de rotas específicos para cada grupo de funcionalidades da nossa aplicação. É uma forma de organizar ainda mais as rotas.