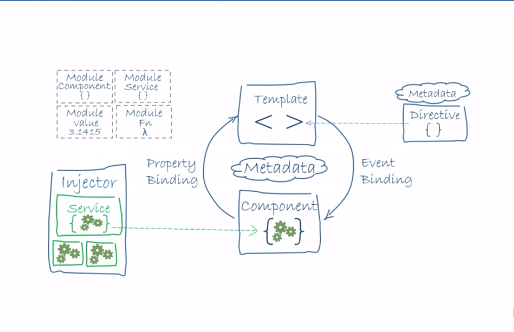
Angular

O Framework possui 8 blocos principais:

1. Componentes
2. Diretivas
3. Roteamento
4. Serviços
5. Template
6. Metadata
7. Data Binding
8. Injeção / Dependência



* **Componentes**: (view), ela encapsula:
  + Template
  + Metadata (processamento das classes)
  + Dado a ser mostrado na tela (Data Binding) (uma div por exemplo)

Quando criamos aplicação em Angular, criamos aplicações que são orientadas à componentes. Temos então: pai/mãe (que vai ser o componente raiz) e partir dele, deriva-se novos componentes.

Sendo que cada componente também pode ter um componente filho (como uma árvore binária)

O componente pode interagir com o back-end em inúmeras linguagens, como: NodeJS, Java, .NET, Ruby, Python... Quem faz o intermédio, entre o componente e o back-end é o **“Serviço”** ele pode ser injetado em outras classes

Por mais que o Angular trabalhe como SPA (Single Page Application), isso não impede que tenha várias abas dentro da mesma aplicação, o responsável pela navegação entre as páginas é o **Router.**

Além disso, temos uma Diretiva, a qual é responsável por modificar elementos DOM (Document Object Model) e/ou seu comportamento, exibir uma div, modificar valores, etc.

Utilizando Angular CLI para criar novos componentes:

$ ng generate componente <name>

Ou:

$ ng g c <name>

Para inicializar o projeto:

$ npm start

**TYPESCRIPT**

Para trabalhar com ecma script, precisaremos de um transpiller... um dos mais conhecidos é o babel.js.  
mas também é possível usar o ‘tsc’ no console para transpillar para js

$ tsc <nome-arquivo.ts>

Hoje em dia, temos os decorators para fazer essa parte. Depois procurar saber melhor

As variáveis podem ser tipadas, ou seja, definir o tipo primitivo dela, assim como C e Java. para isso: funciona dessa forma:

Var n1: string = “teste string recebida”;

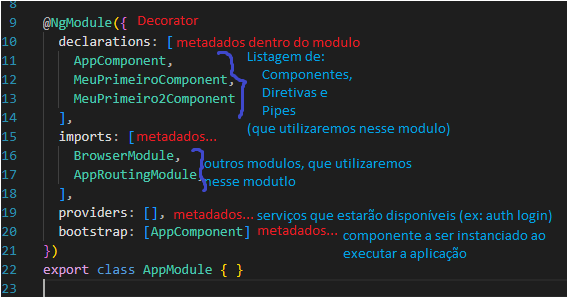
 n1 = 4 (causará erro, pois só aceita tipo string)

var n2: any = “outro teste”;

n2 = 5 (não causará erro, pois a n2 aceita qualquer tipo)

**Módulo**

Por que separar em módulos? Organizar melhor o código, escolher o que cada modulo expõe, etc.



Para gerar um modulo pelo CLI:

$ng generate modules <nome>

Ou

$ng g m <nome>

a

**Templates**

Com os templates é possível utilizar os dados dinamicamente. No component.ts, precisa declarar os atributos (de forma tipada) que utilizará no html. Por Exemplo:

Para fazer a interpolação nessa linha:

<p>Lista de cursos do portal: {{ nomePortal }}</p>

O atributo nomePortal já precisa estar declarado na classe do componente, e sua declaração é feita dessa forma:

export class CursosComponent {

  nomePortal: string;

  constructor() {

    this.nomePortal = 'https://loiane.training';

  }

}

**Serviços (injeção de dependências)**

$ng g s <nome do serviço>

A injeção de dependência serve para que o angular automaticamente forneça uma instância de uma classe de serviço, ao invés de usar um HttpRequest, isso é feito através de um decorator: @Injectable(); esse decorator fica acima da declaração da classe. Para usá-lo, seguem os passos:

* Precisa receber a classe como argumento do método construtor:

  constructor(private cursosService: CursosService) {

* (após isso, consigo utilizar um método dessa classe dentro do bloco do constructor):

    this.cursos = this.cursosService.getCursos();

* E também, preciso passar como um provider no “index” do modulo:

  providers: [

    CursosService

  ]

* Não esquecer de importar:

import { CursosService } from './cursos.service';

Plugins / Extensões:

Angular Snippets

Auto Import

**Novo projeto: Data Binding**

Data Binding -> associar informações que estão no componente para o template, e vice-versa

No angular existem 4 formas

**Interpolação:** “ {{ valor }} “

**Property Binding:** “ [propriedade]= “valor”; ”

**Event Listener:** “ (evento) = “handler”; “ -> (oposto do property binding)

**2 way database:** “[(ngModel)] = “propriedade” “ -> mantem os 2 componentes atualizados ao mesmo tempo.

**Class e Style Binding**

Para pegar o valor de um combo (um select) precisa criar uma variável local dentro do template, e para isso, precisa “escutar” quando o valor muda. Nesse caso:

<select name="" #classe (change)="0">

Class Binding pode ser usado com property Binding, mas também é possível usar a interpolação.

1. Class Binding com property binding:

<div class="alert" role="alert" [class.alert-success]="classe.value == 'alert-success'">Sucesso</div>

1. Class binding com interpolação:

<div class="alert {{ classe.value }}" role="alert" >Texto colorido conforme valor do combobox</div>

Event Binding

No property Binding, usamos colchetes para indicar qual a propriedade a ser alterada. No caso do evento, usamos parênteses. Ex:

<button (click)="botaoClicado()" class="btn btn-primary">Me Clique!</button><br><br>

**Anotações para se lembrar:**

Eventos:

onclick == (click)

onmouseover == (mouseover)

onmouseout == (mouseout)

Variáveis locais em elementos HTML:

Basta usar o ‘#’ e indicar o nome dela. Ex:



Inclusive no exemplo acima, usa-se a ‘#’ para atribuir um id, e utiliza o EVENTO onchange como == (change)

Diferença entre **Propriedade** (que via acima, no property-binding) e **Atributo**.

Uma tag, exemplo: <p id=’aoba’>Teste</p>

Possui um **atributo:** **ID**, de descrição ‘aoba’, que por sua vez, também possui uma propriedade de mesmas características, e a maioria dos atributos também geram propriedades de mesmo nome.   
 Algumas excessões são: o **atributo: for**. Neste caso, a propriedade gerada, é de nome htmlFor, pelo que vi, é uma das únicas excessões, pesquisar futuramente caso precise.

Então, no resumo, o atributo, é uma indicação do html. Quando isso é convertido pelo DOM, é Gerado uma propriedade. Que é como se fosse o “”objeto”” real. Até por isso, quando eu faço algo como:

<input id=’teste’ value=’TESTANDO’></input>

O input gerado já vem com o texto “TESTANDO” nele. Mas se eu deletar, e digitar novas coisas por cima, o Atributo value continuar sendo “testando”, mas a propriedade value será alterada para o valor que eu digitar.

Two Way Data-Binding

O angular possui uma propriedade chamada ngModel, ela é a representação de uma entidade, essa entidade pode ser um campo simples, como uma string, ou um objeto. (entidades essas, estão no arquivo .ts)

Além dessa propriedade, também possui um evento: ngModelChange, que é para quando tal campo for atualizado

Basicamente, deixa os valores atualizados entre template e componente

NgModel é uma diretiva, que pertence ao FormsModule

Input-Modules

Para passar um componente entre modulos, precisa antes no arquivo component.ts declarar que vai receber um input de tal nome, e precisa importar a biblioteca Input. Fica dessa forma:

@Input() nome: string = '';

@Input é um decorator

No caso de nomes diferentes para essa variável. Por exemplo, renomeando a variavel acima para nomeCurso, mas ainda pasando o parâmetro como nome la no html do template anterior. Esse decorator pode receber um argumento, o qual possibilita, expor externamente a variavel com outro nome

@Input('nome') nomeCurso: string = '';

Ex: externamente -> nome, internamente -> nomeCurso

Output modules

Ele possui decorators similares ao input (precisa importar também)

E tem um detalhe em um emissor de eventos. Declarado dessa forma no modulo

  @Output() mudouValor = new EventEmitter();

E para utilizar, utilizamos o nome da variavel.emit(), e pode se emitir varios tipos de dados, incluindo objetos, como está no exemplo a seguir:

    this.mudouValor.emit({novoValor: this.valor})

isso é executado dentro de uma função

Ciclo de Vida de Componentes:

Existe uma lista de lifecycle hooks (ou eventos disponiveis durante o ciclo de vida de um componente), link:

<https://angular.io/guide/lifecycle-hooks#:~:text=kinds%20of%20operations.-,HOOK%20METHOD,-PURPOSE>

ao mudar o valor de um componente, são disparados em sequencia:

ngOnChanges (mais usado)

ngDoCheck

ngAfterContentChecked

ngAfterViewChecked

Existe também o ngif

Veremos melhor no futuro

Acesso ao DOM / Template com ViewChild

  @ViewChild('campoInput')

  campoValorInput!: HTMLElement;

Utilizando o decorator @ViewChild, e passando como parametro a variavel local dentro do component.html é possivel visualizar o input diretamente, sem uso de html, etc.

Declarando a variável como HTMLElement foi possível visualizar no console.log qual o tipo do elemento correto

    console.log(this.campoValorInput)

 que nesse caso é ElementRef

Dessa forma é possivel modificar apenas o html.

Módulo de **ANGULAR CLI** (ferramenta de linha de comando)

Instalação Angular: (requer Node.js > v4)

$npm install -g angular-cli

Criação de novo projeto:

Opção 1:

$ng new <nomeProjeto>

Opção 2:

$mkdir <nomeProjeto>

$ng init

Rodar o projeto:

$npm start

OBS:

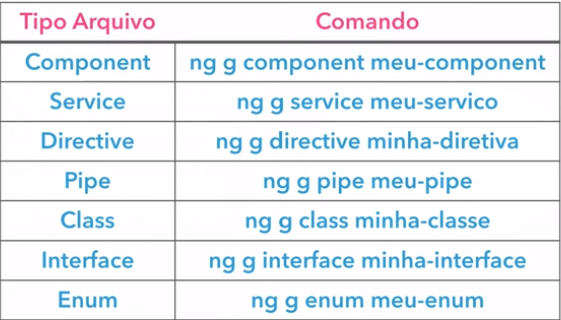
24/01/24 -> os novos projetos que criei, começaram a vir com uma estrutura diferente (por ex: sem o módulo app)

Então achei essa solução para vir da forma que acostumei:

$ng new <my-app> --no-standalone --routing --ssr=false

Generate:

$ng generate <classe>

**Classes**: 

Criar novo componente:

$ng generate component <nome>

$ng g c <nome> (aliases)

Criar novo serviço:

$ng generate service <nome>

$ng g s <nome> (alias)

Comandos ng lint, ng test and ng e2e

$ng lint: escaneia o projeto, e verifica se está de acordo com as boas práticas AND se há erros

$ng test

os arquivos .spec.ts seguem o padrão do Jasmine do JS, um framework? De testes unitários (Interessante procurar saber como o Jasmine funciona, para realizar os testes futuros)

$ng e2e

End to end, para testar a integração com outros apps

**Gerando o Build de Desenvolvimento (Deploy)**

4 opções para build de dev:

$ng build –target=development –environment=dev

$ng build –dev -e=dev

$ng build –dev

$ng build

Para facilitar, rodar apenas o ultimo.

O build será gerado dentro da pasta ./dist/

Buildando para dev, o codigo não é ofuscado, nem minificado, então é possível debugar, ler nomes de variáveis, fazer uma rodada de testes manuais.

3 opções para build de produção:

$ng build –target=production –environment=prod

$ng build –prod –env=prod

$ng build –prod

Obs: para rodar a versão buildada (como teste) é necessario utilizar uma ferramenta do node, chamada http-server, para instalá-lo:

$npm install http-server -g

E aí para rodar a versão teste: mude para a pasta, com $cd dist, e rodar:

$http-server

Mais fácil para testar codigos simples, já que não precisa de um servidr apache.

Para instalar bibliotecas externas:

Basicamente rodar:

$npm install –save bootstrap@next (ou procurar um codigo equivalente no npm lib)

E depois adicionar no campo de scripts



**Diretivas**

Diretivas Estruturais: interagem com a view e modificam a estrutura do DOM, exemplos: \*ngFor, ou \*ngIf

Diretivas de Atributo: interagem com o elemento que foi aplicado, exemplo: ng-class, ou ng-style

**Diretiva \*ngIf**

Tem o mesmo comportamento da estrutura condicional if da programação.

Não possui else

**Diretiva \*ngSwitch-Case**

Funciona como o switch case de outras linguagens.

Exemplo:

div class="container" [ngSwitch]="aba">

        <p \*ngSwitchCase="'mapa'">Modo Mapa</p>

        <p \*ngSwitchCase="'lista'">Modo Lista</p>

        <p \*ngSwitchDefault>Home</p>

      </div>

Analisa a variavel “aba” (declarada no component.ts)

E caso seu valor seja: mapa, lista, ou qqr coisa diferente disso, exibe o respectivo case.

**\*ngFor**

For funciona bem parecido com um let do javascript/Python.

Por exemplo:

    <ul>

        <li \*ngFor="let curso of cursos, let i = index">

        {{i}} - {{ curso }}

        </li>

    </ul>

Variavel cursos (vetor) declarada no component.ts

Para cada curso, em cursos, ele vai reimprimir o <li>.

E o index é apenas outro exemplo

Diretivas de atributo: \*ngClass

Como visto anteriormente possivel manipular os atributos com o property binding, ele fica dessa forma:

    <h1 (click)="onClick()"

    class="bi"

    [class.bi-star]="!meuFavorito"

    [class.bi-star-fill]="meuFavorito"></h1>

Nesse caso, há uma classe “estática” que é a ‘bi’, e caso meuFavorito == false, bi-star será adicionado à lista de classes. E caso meuFavorito == true, bi-star-fill será adicionada às classes.

E a proposta do ngClass é ser mais eficiente, usar menos linhas, etc.

A mesma operação acima, fica dessa forma com ngClass:

 <h1 class="bi" [ngClass]="{

        'bi-star': !meuFavorito,

        'bi-star-fill': meuFavorito

        }" (click)="onClick()"></h1>

Ainda possui ‘bi’ como estático, mas modifica apenas a propriedade ngClass.

Perceba que a exibição é como se fossem objetos (ou JSON)

{

‘bi-star’ :              !meuFavorito ,

‘bi-star-fill’ : meuFavorito

}

A leitura ficaria dessa forma:

Para ‘bi-star’ estar na lista de classes, a expressão: !meuFavorito precisa existir. Ou seja. Bi-star será exibida apenas quando a variável meuFavorito for igual a false.

O mesmo acontece com o bi-star-fill.

ngStyle

Similar à diretiva anterior, o ngstyle é uma alternativa ao property binding, mas focado na parte de css. Exemplo:

<button

    [style.backgroundColor]="ativo ? 'blue' : 'gray'"

    [style.color]="ativo ? 'white' : 'black'"

    [style.fontWeight]="ativo ? 'bold' : 'normal'"

    [style.fontStyle]="ativo ? 'italic' : 'normal'"

    [style.fontSize]="tamanhoFonte + 'px'"

    (click)="mudarAtivo()"

     >Mudar atributo 'ativo'</button>

No codigo acima, ele vai alterar os estilos com base na variável ‘ativo’, utilizando ‘[]’ property binding.

O mesmo código, utilizando ngstyle, ficaria dessa forma:

<button

    [ngStyle]="{'backgroundColor' : (ativo ? 'blue' : 'gray'),

                'color': (ativo ? 'white' : 'black'),

                'fontWeight': (ativo ? 'bold' : 'normal'),

                'fontStyle': ativo ? 'italic' : 'normal',

                'fontSize' : tamanhoFonte + 'px'

            }"

    (click)="mudarAtivo()"

     >Mudar atributo 'ativo'</button>

O layout, é bem similar a um json / objeto.

Operador elvis – navegação segura entre os objetos

é um operador igual ao ternário, mas menos verboso:

    <!-- Utilizando operador ternário: -->

    <p>Responsável: {{tarefa.responsavel != null ? tarefa.responsavel.nome : 'nenhum'}}</p>

    <!-- Utilizando operador Elvis: -->

    <p>Responsável: {{tarefa.responsavel?.nome}}</p>

**ngContent**

o ngContent é como se fosse uma variavel, que espera receber o conteudo através da chamada da tag em questão, por exemplo:  
<div class="border rounded-3 p-5" style="width: 1200px; margin: auto;">

    <ul class="nav nav-tabs" id="myTab" role="tablist">

        <li class="nav-item" role="presentation">

            <ng-content select=".titulo"></ng-content>

        </li>

    </ul>

    <div class="tab-content" id="myTabContent">

        <ng-content select=".corpo"></ng-content>

    </div>

</div>

Nas divs acima, espera-se receber 2 ng-content, aonde ele vai buscar como seletor as classes ‘titulo’ e ‘corpo’.

Os conteúdos são passados para essas tags da seguinte forma:

    <app-diretiva-ngcontent>

      <h5 class="titulo">Título Dinâmico</h5>

      <p class="corpo">Conteúdo passado para o componente</p>

      <app-operador-elvis class="corpo"></app-operador-elvis>

    </app-diretiva-ngcontent>

Por exemplo: app-diretiva-ngContent é o componente que espera receber o conteúdo passado de forma dinâmica

E ai, dentro da tag, passo um h5 com a classe “titulo”, e um p com a classe “corpo”, e posso passar até outro componente dentro do p “corpo”.

Resumão:

Diretivas Estruturais: ngIf, ngFor, ngSwitch

Diretivas de Atributo: ngClass e ngStyle

ElementRef e Renderer:

Criar novas diretivas de atributos. Geralmente elas são compartilhadas por vários componentes, (normalmente dentro de shared)

$ng g d shared/<nome>

A forma não recomendada é utilizar dessa forma:

  constructor(private \_elementRef:ElementRef) { **//obs: injection**

    console.log(this.\_elementRef)

    this.\_elementRef.nativeElement.style.backgroundColor = 'yellow';

  }

Não é recomendado pelo próprio angular, para evitar vulnerabilidades (ex: ataque xxs)

A melhor prática para fazer isso acima, é utilizar o Renderer.

Para utilizá-lo, também precisa fazer a injeção de dependência:

  constructor(private \_elementRef:ElementRef, private \_renderer:Renderer2) {

no fim das contas. Fica dessa forma:

export class FundoAmareloDirective {

  constructor(private \_elementRef:ElementRef, private \_renderer:Renderer2) {

    console.log(this.\_elementRef)

    // this.\_elementRef.nativeElement.style.backgroundColor = 'yellow';

    this.\_renderer.setStyle(this.\_elementRef.nativeElement, 'background-color', 'yellow'); } }

HostListener e Hostbinder

Dois conceitos novos de diretivas customizadas

Hostlistener é uma classe do Angular (que precisa ser importada) que vai escutar do elemento host da diretiva, e executar certa operação.

No exemplo, vou usar o HostListener como se fosse um :hover do css:

  @HostListener('mouseenter') onMouseOver() {

    this.\_renderer.setStyle(this.\_elementRef.nativeElement, 'background-color', 'yellow');

  }

  @HostListener('mouseleave') onMouseLeave() {

    this.\_renderer.setStyle(this.\_elementRef.nativeElement, 'background-color', 'white');

  }

  constructor(private \_elementRef: ElementRef, private \_renderer:Renderer2) {

   }

Percebe-se que utiliza uma annotation, que quando for executada, chama um método.

Como está muito repetitivo, é melhor usar o Hostbinder nesse caso

  @HostListener('mouseenter') onMouseOver() {

    this.backgroundColor = 'yellow';

  }

  @HostListener('mouseleave') onMouseLeave() {

    this.backgroundColor = 'white'

  }

  @HostBinding('style.backgroundColor') backgroundColor: string = '';

O Host Listener ficará escutando evento no hospedeiro da diretiva (nesse caso a tag html que o chamou)

E o host binding, permite que faça associação de atributo/classe para uma variável.

É possivel também utilizar métodos getters e setters para atribuição desses valores (mas só é util caso precise fazer alguma alteração dentro do hostBinding (no comentário “//codigo extra” do print abaixo)

Apenas para registro, com os métodos getters/setters, fica dessa forma:



Diretivas com @Input e Property Binding

Usando o mesmo highlight que o anterior, modificaremos para deixar a cor dinâmica, ou seja, vai ser passado como parâmetro para o método.

Nesse caso precisaremos importar a classe do Angular Input.

E usá-lo para criar 2 variáveis, dessa forma:

  @Input() defaultColor: string = 'white';

  @Input() highlightColor: string = 'yellow';

Obviamente, nos metodos principais, agora recebo o valor dessas variáveis:

  @HostListener('mouseenter') onMouseOver() {

    this.backgroundColor = this.highlightColor;

  }

  @HostListener('mouseleave') onMouseLeave() {

    this.backgroundColor = this.defaultColor

  }

Porém, não está inicializando como o defaultColor recebido. Nesse caso, como uma directive é igual a um componente, mas sem template, ele também possui ligação com os lifecycle-hooks, então é possível usar dessa forma:

  ngOnInit(){

    this.backgroundColor = this.defaultColor;

  }

(o ngOnInit não pode ser declarado dentro do método construtor.)

Caso queira usar o input property, com o mesmo nome da diretiva, da pra fazer dessa forma: (na declaração da variável ficará dessa forma):

@Input('hightlight') highlightColor: string = 'yellow';

E na chamada fica dessa:

    <p [highlight]="'red'"  defaultColor="grey">Texto com highlight com cores customizadas</p>

Criando uma diretiva de estrutura: ngElse

Para essa diretiva, reutilizei o código da ngIf.

Para a parte lógica dessa diretiva, tem 2 novas classes do Angular:

TemplateRef, ViewContainerRef.[

O Código completo está assim:

@Input() set ngElse(condition: boolean){

  if(!condition){

    this.\_viewContainerRef.createEmbeddedView(this.\_templateRef);

  }else{

    this.\_viewContainerRef.clear();

  }

}

  constructor(private \_templateRef:TemplateRef<any>, private \_viewContainerRef:ViewContainerRef) { }

}

Explaining:

Utiliza o decorator @Input para receber o valor como parâmetro do método ngElse. E o ‘set’ antes de ngElse, é o método setter.

Lógica:

Se a condição for verdadeira:

Utiliza o viewContainerRef para criar o template, método: createEmbeddedView. E passa como argumento o conteúdo da div.

Caso não for:

Limpa o conteúdo do template.

E no constructor precisa fazer a injeção de dependência dos dois métodos novos. Isso é feito dentro dos parenteses do método

Serviços: DRY – Don’t Repeat Yourself

Services do padrão MVC.

Mas não são apenas um intermediador para o CRUD. Também são muito úteis para não repetir códigos na aplicação.

O ideal é que a toda a lógica esteja nas classes de serviços.

O componente é apenas para exibição dos dados

Injeção de Dependência

Para deixar os valores das classes atualizadas entre sí, precisa fazer essa injeção de dependencia, no Angular 2, ela é feita através do método construtor (passada como parâmetro)

Para utilizar, é necessario utilizar o decorater @Injectable(), e para usar ele, é necessario importá-lo, ele pertence ao pacote @angular/core

Aula 40: Escopo de instância de serviços e módulos (singleton e várias instancias)

Confesso que fiquei beeeeeeeeeeeem perdido. Mas pelo pouco que entendi, não tem necessidade de instanciar várias vezes o mesmo serviço, e quem controla essa quantidade de execução, é o “provider” dos módulos. E aí a depender do escopo, voce controla esse provider de certos modulos... (app é o geral), isso é o singleton

E não funcionou muito bem os códigos que fiz, provavelmente terei que refazer essa aula quando compreender melhor... ta dando bastante erro, principalmente na hora de descomentar os providers para usar o escopo geral.

Comunicação entre componentes:

Podemos usar o Input e Output property, como vimos anteriormente, mas apenas funciona de componente pai para componente filho.

Caso sejam dois componentes independentes, podemos usar os services para tal:

Ao criar um event emitter, fui no componente que vai receber o evento. E usei a função subscribe() para receber

this.cursosService.emitirCursoCriado.subscribe

usando o método acima, ainda assim só conseguirei acessar de outro componente, quando o serviço for instanciado.

Para fazer uma espécie de broadcast entre varios componentes, utilizar métodos static:

    static criouNovoCurso = new EventEmitter<string>();

nesse caso, para utilizar em um método da mesma classe, não utilizarei como this.criouNovoCurso, mas sim dessa forma:

        CursosService.criouNovoCurso.emit(curso);

E todos os componentes, que possuirem link com esse service, conseguirá utilizar esse mesmo método, sem precisar instanciá-lo (injetar dependência) no componente.

Exemplo: o cursos.component utilizando esse método:

    CursosService.criouNovoCurso.subscribe(

     curso => this.cursos.push(curso) // adiciona à lista

    );

**Injetando um serviço dentro de outro + gerando serviço com Angular CLI**

$ng g s shared/<nome-serviço>

Mas o serviço não é adicionado em nenhum provider, isso precisa ser feito manualmente

Fiz o import no modulo pricipal, e adicionei no provider.

Em seguida, fui no curso.service (já que vou inserir um serviço no outro), também fiz o import, instanciei no construtor

    constructor(private logService: LogService){}

E para utilizar nos métodos, dessa forma:

getCursos() {

        this.logService.consoleLog('Obtendo lista de cursos')

        return this.cursos;

    }

Obs: o logService possui apenas esse método:

  consoleLog(msg:string){

    console.log(msg);

  }

Pipes (filtros)

Criei o novo projeto utilizando aqueles argumentos para que seja criado também os módulos:

$ng new <my-app> --no-standalone --routing --ssr=false

O pipe serve para transformar um valor, e exibir o resultado dessa transformação no template, e o Angular 2 tem alguns pipes embutidos, Ex:

Pipe para mostrar data formatada na tela, para tratar numeros, decimais, etc.

Não é necessario importar nada para os modulos, pois já estão embutidos no BrowserModule, e CommonModule, e como eles já estão importados no modulo, não é necessario importar no componente.

Para utilizá-los, é dessa forma:

<p>Título: {{ livro.titulo | uppercase | lowercase }}</p>

<p>Estrelas: {{ livro.rating | number:'1.1-1'}}</p>

<p>Páginas: {{ livro.numeroPaginas | number }}</p>

<p>Preço: {{ livro.preco | currency:'BRL':true }}</p>

<p>Data Lançamento: {{ livro.dataLancamento |date:'dd/MM/yyyy' }}</p>

<p>Url: {{ livro.url }}</p>

<br>

<p>Livro: {{ livro | json }}</p>

Criando novos pipes:

$ng g p <nome-pipe>

Como foi criado pelo Angular CLI, é necessário importá-lo no módulo que o utilizará, e eles vão dentro das declarations

**(!IMPORTANT: componentes, diretivas e pipes sempre são dentro das declarations)**

(estranhamente, no meu caso já declarou e importou automaticamente)

Criei o pipe CamelCase, a logica utilizada foi essa:

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

    let values = value.split(' ');

    let result = '';

      for (let v of values) {

        result += this.capitalize(v) + ' ';

      }

    return result;

  }

    capitalize(value: string) {

        return value.substr(0 , 1).toUpperCase() + value.substr(1).toLowerCase();

        value.substr(1).toLowerCase();

    }

E o jeito de usar no código, é exatamente igual às pipes vistas anteriormente.

**Locale (internacionalização)**

Aplicar um locale, para que os pipes sejam convertidos para o padrão correto do país,

Por exemplo, no Brasil utilizamos vírgula para separar os decimais da moeda, mas por padrão, os pipes já vem com o padrão americano. Para fazer essa localização, nas versões mais novas do angular precisa importar esses 2 library:

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

import ptBr from '@angular/common/locales/pt';

ja no codigo, antes do @NgModule, deixei:

registerLocaleData(ptBr);

e dentro do NgModule, nos providers, deixei dessa forma:

  providers: [

    {

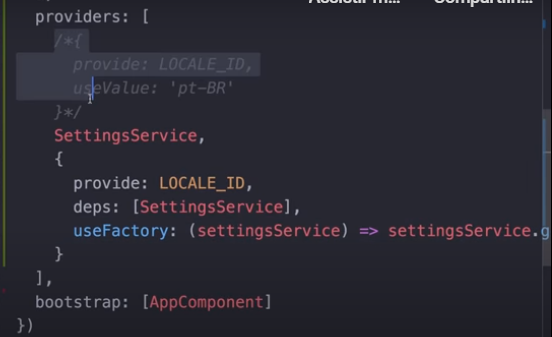
      provide: LOCALE\_ID, useValue: 'pt'

    }

  ],

E ai funcionou.

Caso precise utilizar um serviço para puxar essa localização, funciona dessa forma:



Esse serringsService possui um método getlocale() {return ‘pt-BR’}

Na verdade isso era da versão antiga, caso precise fazer isso um dia, procurar saber como fazer no angular mais recente

**Pipe puro:**

Para isso, criei uma lista no template

**Disclaimer: não filtrar array em projeto real**

Aqui filtraremos apenas pra aprender a diferença entre pipe puro e pipe impuro

Pipe puro basicamente, não atualiza, quando o valores que recebe é alterado, apenas faz a transformação. E também no caso do exemplo da lista com filtro, ele não atualizou a lista ao adicionar algo com um filtro já selecionado.

A solução para isso, é utilizar um pipe impuro, para fazer essa verificação, e atualizar corretamente a lista

**Pipe Impuro:**

Basicamente, ele atualiza em tempo real o valor recebido.

No exemplo da lista (o qual não deve ser usado em projetos reais), funciona dessa forma:

export class FiltroPipeImpuroPipe extends FiltroArrayPipe {

}

A classe apenas herda os atributos e métodos da classe FiltroArrayPipe

E a outra diferença, é que adiciona um argumento a mais na annotation @Pipe, aonde indica que pure == false.

Apenas isso.

**Pipe Async:**

Funciona para casos de requisições que demoram alguns milissegundos para serem executadas.

Seu uso, é tao simples quanto os primeiros pipes vistos:

<p>{{ valorAsync | async}}</p>

Nesse caso, fizemos uma Promise para atribuir o valor à variável valorAsync após 2 segundos. Mas também é possível utilizar um Observable, para utilizar esse pipe async.

Nas versões mais recentes do Angular, a sintaxe ficou dessa forma:

  valorAsync2 = interval(2000).pipe(map(valor => 'Valor Asíncrono 2'));