**Básico Angular 9**

-> Baseado em componentes (Angular 2 em diante)

-> Baseado em TypeScript (JavaScript ‘tipado’ e com algumas funcionalidades, como decorators).

-> A Aplicação é 'modularizada', trazendo maior nível de encapsulamento.

-> A aplicação é inicializada pelo **AppModule** (Chamado no **main.ts**)

-> O **AppModule** vai apontar para o componente padrão (**AppComponent**) e a partir do AppComponent toda a árvore vai ser chamada.

-> Dentro do AppComponent será feita a referência para **AppHeader**, **AppFooter**, **AppContent**, etc...

-> Componente Angular (Ex: Home)

\* HTML (home.component.html)

\* CSS (home.component.css)

\* TS (home.component.ts)

De cada componente vai ser gerado uma tag encapsulando todos os componentes filhos de App: <app-home></app-home>

Nesse componente todo o html, css e js(gerado a partir do ts) vai estar contido.

A partir do TypeScript (home.component.ts) que o Angular vai encontrar o html e css, sendo necessário importar dentro

decorartor desse componente.

Colocar o template dentro do arquivo TypeScript (*inline template*) **????**. Caso queira os arquivos html e css separados, ajustar em angular.json (inlineTemplate = false e inlineStyle = false).

-> Angular é um SPA (single page application), o componente principal src/app/app.component.ts vai ter como seletor a tag ‘app-root’, que é a tag que vai ser renderizada dentro do body da nossa única página (src/index.html).

-> Todos os componentes vão estar dentro de um módulo (Usar tudo dentro de um único módulo, AppModule por exemplo) ou criar mais módulos **????** (Figura 1).

-> Quando criamos um componente dentro de um módulo podemos deixar esse componente visível apenas dentro do módulo, ou disponibilizar para fora do módulo.

Figura 1. Modularização dos componentes (A até M).

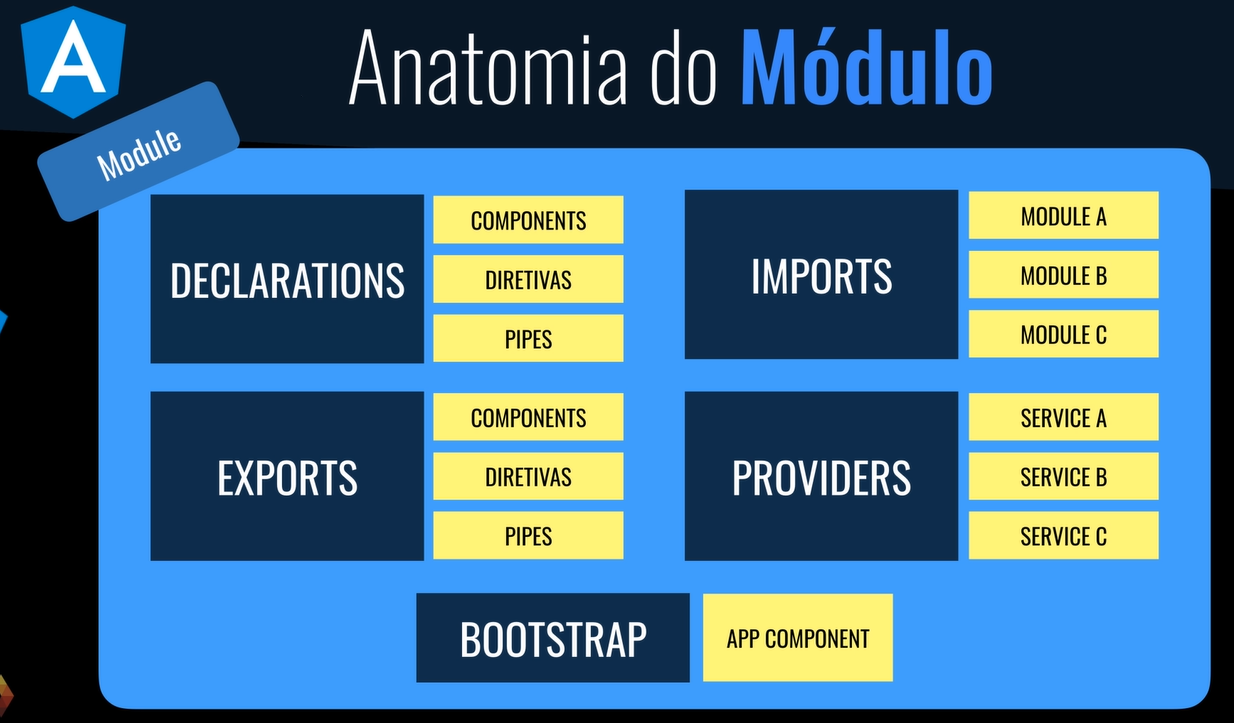
Diagrama, Linha do tempo

Descrição gerada automaticamente com confiança média

-> O Componente H é o único que pode ficar visível apenas dentro do módulo dele (**YModule**), pois não comunica com nenhum componente de outro módulo.

-> Dentro de um módulo, temos cinco atributos (Figura 2), **DECLARATIONS**, **EXPORTS**, **IMPORTS**, **PROVIDERS** e **BOOTSTRAP**.

Figura 2. Estruturação de um módulo em Angular.



* DECLARATIONS: Aqui vamos declarar todos os componentes que fazem parte do módulo (visível ou não), para ficar visível temos que colocar dentro de exports. Diretivas **????** Pipes **????**
* EXPORTS: Aqui vamos definir os componentes, diretivas e pipes que vamos deixar visível em outros módulos.
* IMPORTS: Aqui vamos fazer os imports de módulos dentro do nosso módulo, que podem ser módulos da própria aplicação ou externo (libs).
* PROVIDERS: Aqui vamos declarar os services **????**
* BOOTSTRAP: Aqui vamos definir o componente que será carregado no módulo. O bootstrap vai ser necessário apenas no módulo carregado inicialmente (AppModule). É o atributo bootstrap que vai fazer com que o componente principal seja carregado e esteja disponível na SPA(*Single page application*).

**Elementos do Angular #01**

* Componentes

Componente é formado por um html, um css e um TypeScript. Cada componente tem seu ‘mundo’ interno, ou seja, tudo que for aplicado de estilo por exemplo, vai ficar ser apenas usado no componente em questão.

* Diretivas (**????**)

Existem dois tipos de diretivas, a de atributo e a estrutural. A de atributo foca em alterar a aparência ou o comportamento (Ex: colocar diretiva em botão que vai fazer uma chamada ajax para o backend) de um elemento, de um componente ou de outra diretiva.

Para definir uma diretiva, precisamos informar na classe com o decorator @Directive. A diretiva estrutural altera o layout, adicionando e removendo elementos da dom.

Ex: Setar um form caso product esteja setado(true)

**<form \*ngIf=’produtc’**

**class=’product-form’>**

**</form>**

Ex: Exibir uma lista de produtos quando existir

<ul>

<li \*ngFor=’let product of products’>

{{ product.name }}

</li>

</ul>

**<ul>**

**<li \*ngFor=’let item of lista>**

**<b>**

**{{ item.nome ? item.nome : ‘Não existe’}}**

**</b> - {{ item.cod ? item.cod : ‘Não existe’}};**

* **Carne** – 564;
* **Peixe** – 856;
* **Não existe** – Não existe;

**<button ngIf=”adm=’true’”> Botão </button>**

**<button ngIf=”adm”> Botão </button>**

**</li>**

**</ul>**

Para comunicar nosso TypeScript com o html, fazemos isso por referência **(Property Binding).**

Ex:

HTML

<table

[dataSource] = ‘**products**’>

</table>

TS:

@Component({

selector: ‘app-product-read’,

templateUrl: ‘./product-read.componente.html’,

styleUrls: [‘./product-read.componente.css’]

})

export class ProductReadComponent implements OnInit {

**products**: Product[];

}

A tag table do html, vai procurar na nossa classe (componente ProductReadComponent) a parâmetro ‘product’ para setar na DOM, ou seja, [dataSource] recebe products.

Para ligar um evento no html a um método no nosso TypeScript, fazemos isso também por referência (**Event Binding**).

Ex:

HTML

<button mat-raised-button

(click)=”**createProduct()**”

color=”primary”>

</button>

TS

@Component({

selector: ‘app-product-read’,

templateUrl: ‘./product-read.componente.html’,

styleUrls: [‘./product-read.componente.css’]

})

export class ProductReadComponent implements OnInit {

**createProduct()**: void { ... }

}

Se quisermos alterar o valor de um input no html, quando a variável de referência é alterada no TypeScript, podemos usar o **One Way Data Binding.**

Ex:

HTML

<input value=”nome”>

TS

nome: string

Se for necessário que essa alteração seja em via dupla (alterou ts altera html, alterou html, altera ts), podemos usar o **Two Way Data Binding.**

Ex:

HTML

<input [(ngModel)]=”nome”>

TS

nome: string

As rotas servem para exibir em uma ‘mesma tela’ diversos componentes conforme seleção do usuário.

Esse mecanismo vai ser possível, pois vamos ter um mapeamento entre a rota e o componente, e conforme a rota selecionada, o componente correspondente será renderizado no Router Outlet.

Os pipes são processamentos que fazemos em cima de variáveis

Ex:

<p>

O vencimento é

{{ produto.vencimento | date }}

</p>

O ‘| date’ vai formatar a data passada (produto.vencimento).

Alguns pipes recebem parâmetros.

Ex:

<td mat-cell \*matCellDef=”let product”>

{{ product.price | currency: ‘BRL’ }}

</td>

Podemos também ter uma cadeia de processamento.

Ex:

{{ produto.vencimento | date: ‘fullDate’ | uppercase }}

**Elementos do Angular #02**

Programação Reativa (menor uso de processamento)

O Angular utiliza um framework chamado ‘ReactiveX’, que trabalha com programação reativa, ou seja, só é chamado se algo externo acontecer.

O ReactiveX trabalha em cima do padrão de projeto ‘**Observer**’.

O observer é um padrão orientado a evento, ou seja, temos um Subject, que é quem tem a capacidade de detectar um determinado evento, e temos os observers, que podem se registrar nesse subject de forma a sinalizar o interesse em determinado evento que esse subject consegue detectar. Quando o evento ocorrer, o subject vai sinalizar (call-back) para todos os observers registrados que esse evento aconteceu. Essa chamada foi reativa, pois os observers estão livres para outras tarefas.

Observables (subjects) diferente das promises podem ser reutilizáveis, conseguem receber strings de dados (Ex: Manipular andamento de apuração de votos, além disso possuem operadores (O que seria o then as promises), como map, filter , etc..., como por exemplo, listar produtos do backend, temos como realizar essas diversas operações nesses dados (Figura 3).

Figura 3. Exemplo de funcionamento de um *observable* com chamada ao backend.

Texto

Descrição gerada automaticamente

O criarNoBackend vai registrar um produto via método post no backend, porém, esse registra leva algum tempo (Asyncrono), para isso, criarmos um observable, que é o nosso criarProduto (Usando o .subscribe).

**Elementos do Angular #03**

Services são classes que tem como objetivo organizar e compartilhar métodos e dados entre componentes. Services podem ser usados dentro de componentes e diretivas também.

Services podem ser criados para determinadas responsabilidades, como por exemplo determinada chamadas ao backend, compartilhamento de informações entre componentes que não estão ligados.

Como boa prática, determinadas lógicas, principalmente as duplicadas em componentes, podem ser associadas a um service.

Ex: Essa classe vai ser possível de ser injetada em outras classes.

ng g s services/product

@Injectable({

providedIn: “root”,

})

export class ProductService {

}

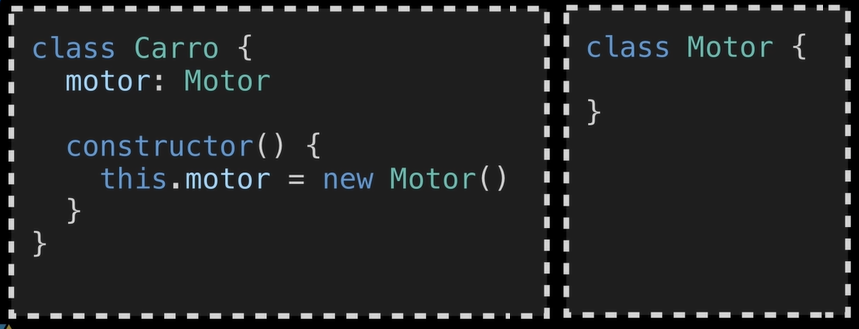
Para usar um service dentro de um componente precisamos fazer a injeção de dependências. Sempre que injetamos um service dentro de um componente, o Angular vai devolver a mesma instância. O Angular também vai ser responsável gerenciar a criação dos componentes, ou seja, instanciar as classes que representam os componentes, diretivas e services.

Quando um service é injetado em um componente, sobre alguma modificação de algum atributo, ao injetar esse service dentro de outro componente, esse atributo é mantido como o que foi alterado, pois trabalhamos sempre com a mesma instância do service.

Essa ‘injeção de dependência’ é um padrão no qual a classe recebe as dependências de uma fonte externa ao invés de criar por conta própria (A classe correspondente ao service, não é criada novamente, já foi instanciada).

Uma classe pode receber outra através de um construtor (Figura 4) ou da injeção de dependência (Figura 5).

Figura 4. Absorção de uma classe via construtor.



Se a classe motor tiver algum parâmetro variável (Ex: cilindrada), nossa classe Carro vai ter que ser alterada sempre que o motor for alterado (Ex: Variação da cilindrada).

Para resolver isso, é a injeção de dependência, passar o motor como parâmetro, de forma que uma fonte externa constrói o motor e nossa classe Carro, recebe o motor ‘pronto’.

O Angular tem um framework para injeção de dependência, esse framework vai verificar se uma classe (componente), é ‘injetável’ (@Injectable), feito isso, a angular instância essa classe para nós, criando uma instancia dessa classe. Ao criar um outro componente que vai receber nosso injetável (classe instanciada), o angular vai injetar (passar como parâmetro), nosso componente injetável para a classe.

No service, temos o parâmetro provideIn, que ao receber root, fazemos um ‘alias’ para o módulo app (app.module). Associado ao app.module, existe o RouteInjector, responsável pela injeção de dependência. Sempre que criamos um service com provide ‘root’, estamos dizendo que esse nosso service vai ter uma única instância em toda aplicação.

OBS: Services são singletons dentro do escopo de um injector.

Temos basicamente 2 grupos de injetores:

* + ModuleInjector: Ao passar provide igual a ‘root’, o root aponta para app.module, logo estamos usando o Injetor de módulo.
  + ElementInjector: Podemos criar um módulo de autenticação (auth-module), se quisermos um serviço dentro desse escopo, passaremos para o provide o ‘auth-module’

Resumindo: Temos 2 injetores, o injetor de módulo e o injetor de elemento. Sempre que usamos @NgModule e dentro do módulo definimos os providers, referenciado determinado service (dentro do escopo do módulo teremos apenas uma única instância do serviço).

Para acrescentar nossas rotas entre views(componentes menores), precicamos seguir alguns passos:

1ª Setar a tag router-outlet dentro do nav.component.html, onde será incluída dinamicamente nossos componentes

2ª Incluir essas rotas dentro do objeto Routes em app-routing.module.ts

3ª Dentro de nav.component.html, nos links, precisamos incluir o parâmetro routerLink e passar os caminhos (‘/’ e ‘products’).

**Ex de Diretiva de atributo**

ng g d directives/red – Comando para criar uma diretiva. Que ao criar vai importar em app.module.ts

red.directive.ts:

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

@Directive({

selector: '[**appRed**]'

})

export class RedDirective {

constructor(private el: ElementRef) { //Aqui injetamos o el

el.nativeElement.style.color = '#e35e6b'

}

}

Em footer.component.html:

<mat-toolbar class="footer">

<span>

<p class="texto" **appRed**>Copyright © 2021 Crud-Angular. Todos os

direitos reservados.</p>

</span>

</mat-toolbar>

Na paragráfo passamos o seletor, que vai aplicar a o estilo setado na diretiva.

Ex de diretiva estrutural

ng g d directives/for

Em for.directive.ts

import { Directive, Input, OnInit, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({

  selector: '[myFor]'

})

export class ForDirective implements *OnInit* {  //implements OnInit, vai chamar a diretiva sempre que houver mudança

  @Input('myForIn') numbers ? : *number*[]

  //myForEm, vai pegar o que vem depois da palavra 'in', no caso [1,2,3]

  constructor(

    private *container*: ViewContainerRef,

    private *template*: TemplateRef<*any*>

  ) {}

  ngOnInit(): *void* {

    for (let number of *this*.numbers) {

*this*.container.createEmbeddedView(

*this*.template, { $implicit: number }

      )

    }

  }

}

Em [home.component.html](http://home.component.html)

  <ul>

    <li *\*myFor*="let n in [1,2,3]">{{ n }}</li>

  </ul>

Para fazer um binding de uma função (event binding), em um button, por exemplo, precisamos usar a sintaxe (click)

Em product.create.component.html:

<button (click)=”fazAlgo()”>Faz algo</button>

Em product-create.component.ts:

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

@Component({

  selector: 'app-product-create',

  templateUrl: './product-create.component.html',

  styleUrls: ['./product-create.component.css']

})

export class ProductCreateComponent implements *OnInit* {

  atributo = 'qualquer'

  constructor() { }

  ngOnInit(): *void* {

  }

  fazAlgo(): *void* {

    console.log("Fazendo algo")

  }

}

Para fazer um binding de id

Em product-create.component.ts:

export class ProductCreateComponent implements *OnInit* {

  propLegal = 'qualquer'

  constructor() { }

  ngOnInit(): *void* {

  }

  fazAlgo(): *void* {

    console.log("Fazendo algo")

  }

}

Em product-create.component.html

<p *[id]*="propLegal">product-create works!</p>

O id da tag p agora não é mais ‘propLegal’, e sim ‘qualquer’