**Criar uma nova aplicação:**

ng new ***nomeDoProjeto*** --prefix=***prefixoDosComponentes***

**Inicia um servidor Angular:**

ng serve

**Módulos Angular**: Um módulo Angular é responsável por agrupar vários componentes da aplicação ou do framework.

**Decorator**: Responsável por aplicar metadados em uma classe, atributo, etc. Alguns exemplos são o @NgModule e @Component.

@Input(): Usado para entrada de dados.

@Output(): Usado para saída de dados.

@Component: Usado setar o seletor, template e o estilo de um componente.

@NgModule(): Usado para inserir metadados sobre um módulo.

**Componente**: São classes que possuem ciclo de vida, template e um seletor(tag). São elementos personalizados. Se a declaração de um componente for feito no módulo raiz, toda a aplicação tem acesso ao mesmo.

**Criar componente**:

ng g c ***nomeDoComponente*** --spec

g: generate; c: componente; --spec: true para gerar arquivos de teste ou false para não gerar.

Podemos colocar um componente dentro do outro apenas usando as tags do mesmo.

**Property Binding**: é a ligação de uma propriedade do componente ao uma expressão angular que pode avaliar para um método, classe ou uma expressão mais elaborada.

Sintaxe: <input [value]=”user.name”>

Se for usado pra tipos que não sejam string, é obrigatório usar a sintaxe de property binding [] no atributo.

**Diretivas**: Existem 3 tipos. Componentes que são as mais comuns, estruturais que mudam o template/estrutura do DOM, e as de atributo que associa um atributo a um elemento do DOM e um determinado comportamento é atribuído ao elemento.

**NgIf:** Usado para trabalhar com a visibilidade do elemento. Se for true, o elemento fica visível.

<div \*ngIf=”***expressãoBooleana***”></div>

**NgFor**: Usado para iterar arrays.

<div \*ngFor=”let ***elemento*** of ***array***”></div>

**Operador de navegação segura**: Usada para não deixar a aplicação quebrar, caso o objeto que nós utilizarmos no componentes seja undefined. Utiliza-se interrogação a frente do objeto ou propriedade que pode ser undefined.

<div>{{***objeto***?.***propriedade***}}</div>

Aplicando eventos a um elemento do template:

No template:

<a href=”#” (***eventoDoDOM***)=”***nomeDoEvento***()”>Texto</a>

Na classe do componente:

***nomeEvento***(){

//instruções

}

Referenciar elementos:

No template:

<textarea #***nomeDoMarcador***></textarea>

<button (click)=” ***nomeDoMarcador.métodoDeFoco()***”>Focar Textarea!</button>

**Rotas:**

**Arquivos com todas as rotas:**

Exports const ROUTES: Route = [

{path: ‘***rotaDisponível1***’, componente: ***nomeDoComponente1***},

{path: ‘***rotaDisponível2***’, componente: ***nomeDoComponente2***},

{path: ‘***rotaDisponível3***’, componente: ***nomeDoComponente3***},

]

**Inserir o RouterModule na classe AppModule da raiz:**

nNgModule({

declarations: [...],

imports: [..., RouterModule.forRoot(ROUTES)],

})

Export class AppModule

Como acessar essas rotas:

1ª: <a routerLink=”***rotaDefinida***”>Link</a>

2ª: <a [routerLink]=***”[‘rotaDefinida’]***”>Link</a>

**Injetando dependências**

Usa-se a dependência como parâmetro do construtor para evitar a criação constante de chamadas de outros módulos:

import { RestaurantsService } from './restaurants.service';

export class RestaurantsComponent implements OnInit {

constructor(private restaurantsService: RestaurantsService) { }

ngOnInit() {

this.restaurants = this.restaurantsService.restaurants();

}

}

Neste caso a classe RestaurantsService importada é um Serviço e passamos ela por parâmetro no método construtor. Devemos iniciar todos os objetos que usam esse serviço injetado no método ngOnInit, que é chamado quando a classe é executada. Por fim, devemos dizer ao módulo que queremos utilizar quem é esse provider. Pra isso devemos importar a classe e coloca-la na lista de providers do módulo.

import { RestaurantsService } from './restaurants/restaurants.service';

@NgModule({ ...

providers:[RestaurantsService], ...

})

**Adicionando HTTP ao projeto**

Adicionamos o Decorator @Injectable() para especificar que o nosso serviço usará outro serviço. Importamos o módulo HTTP e o Observable para trabalharmos com requisição. A requisição retorna um Observable e não um retorno com os dados solicitados. Para isso usamos o método map() para iterar o Observable e trazer um objeto do tipo Response. Para utilizar o map, temos que importa-lo também. Isso tudo é feito na classe do serviço que estamos trabalhando.

import { Restaurant } from "./restaurant/restaurant.model";

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

import { Http } from "@angular/http";

import { Observable } from "rxjs/Observable";

import 'rxjs/add/operator/map';

@Injectable()

export class RestaurantsService {

constructor(private http: Http) {}

restaurants(): Observable<Restaurant[]> {

return this.http.get(`${MEAT\_API}/restaurants`)

.map(response => response.json())}

}

No classe do componente que iremos utilizar os dados, devemos fazer uma chamada para o método em que está a requisição desejada e utilizar o método subscribe(). Só após usar esse método que a requisição é feita. Dentro do método subscribe estamos setando um valor para o atributo restaurants.

ngOnInit() {

this.restaurantsService.restaurants()

.subscribe(restaurants => this.restaurants = restaurants)

}

**Tratamento de erros no serviço**

Há duas maneiras de tratar erros. Uma com a classe Error Hadling e outra com o operador catch. Para trabalhar com o catch, temos que importa-lo.

Import ‘rxjs/add/operator/catch’

Se no método catch() houver algum erro, ele fecha o Observable corrente. O método receberá uma função e retornará um Observable.

Crie uma classe para tratar os erros com métodos próprios.

import { Response } from "@angular/http";

import "rxjs/add/observable/throw";

import { Observable } from "rxjs/Observable";

export class ErrorHandler {

static handleError(error: Response | any) {

let errorMessage: string

if (error instanceof Response) {

errorMessage = `Erro ${error.status} ao acessar a URL ${error.url} - ${error.statusText}`

} else {

errorMessage = error.toString()

}

console.log(errorMessage)

return Observable.throw(errorMessage)

}

}

Importamos a classe criada na classe de serviço e adicionamos o método catch() após o método map com a função desejada como parâmetro.

import { ErrorHandler } from "../app.error-handler";

.catch(ErrorHandler.handleError)

**Trabalhando com parâmetros nas rotas**

Existem duas maneiras para se trabalhar com parâmetros. A primeira que é a SNAPSHOT é usada quando você não precisa se preocupar com a mudança de rotas e tempo de vida de um componente, pois ela busca a rota ativa no momento. O outro é o SUBSCRIBE, que é usado quando você tem dois ou mais componentes que se conversam. Por exemplo, um link clicado em um componente que muda o conteúdo de outro.

Snapshot:

Primeiro injetamos o objeto ActivatedRoute na classe do componente.

constructor(private route:ActivatedRoute){}

Buscamos o parâmetro id da seguinte forma:

ngOnInit(){

const id = this.route.snapshot.params[‘id’}

Subscribe:

Primeiro injetamos o objeto ActivatedRoute na classe do componente.

constructor(private route:ActivatedRoute){}

Buscamos o parâmetro id da seguinte forma:

ngOnInit(){

this.route.params.subscribe(params => {

const id = params[‘id’})

**Trabalhando com rotas filhas**

As rotas filhas são rotas pertencentes a outras rotas. Por exemplo a rota “restaurants/:id/menu” é filha da rota “restaurants/:id”. Para isso adicionamos o elemento children ao path mãe no arquivo de rotas, especificando o path e o componente:

Export const ROUTES: Routes = [

{

path: ‘restaurants/:id’, component:RestaurantDetailComponent,

children:[

{path: ‘menu’, componente: MenuComponent },

{path: ‘reviews’, componente: ReviewsComponent }

{path: ‘’, componente: MenuComponent }]

}]

A última rota vazia, indica que quando a rota for igual a rota mãe, porém não ter nada após a barra da rota mãe, ela será redirecionada para o componente mencionado após a rota vazia.

No HTML do componente devemos acessar a rota da seguinte forma:

<a [routerLink]=”[‘menu’]”>Menu</a>

Para aplicar uma classe no link ativo:

<a [routerLink]=”[‘menu’]” routerLinkAtive=”***classe-css***”>Menu</a>

**Pipes**

São responsáveis por transformação de dados para uma apresentação diferente. Parecidos com os filtros do Angular 1.x. Se você quer transformar uma string para maiúsculo, basta fazer o seguinte:

<div>{{user.name | uppercase}}</div>

Ou minúscula:

<div>{{user.name | lowercase}}</div>

Para utilizar porcentagem:

<div>{{0.5 | percent}}</div>

Para limitar a quantidade de caracteres:

<div>{{user.name | sile: 0:4}}</div>

Data:

<div>{{birthday | date: ‘dd/MM/yyyy’}}</div>

Ou moeda:

<div>{{price || currency: ‘BRL’: true}}</div>

Os parâmetros do pipe são separados por dois pontos “:”. O primeiro parâmetro é o formato ISO, o segundo é o que indica se usa String ao invés do símbolo ISSO e o terceiro indica o formato do número.

Os pipes são iguais aos usados no Node.js, onde um resultado de um processamento é usado como entrada de outro processamento.