**Primeira aula, baixando o projeto inicial**

Por algum motivo, não iremos fazer o projeto completamente do 0, o que me deixa puto xD

Deve ser baixando algumas coisas prontas do repositório

Serão baixados os repos:

# meat-app-starter

Depois de baixado, é preciso dar um npm install na pasta raiz dessa aplicação para baixar as dependências

Tive que rodar esse comando pois estava tendo um erro bolado

npm config set registry http://registry.npmjs.org/

xD

Após isso atualizei o angular cli com o comando

npm install --save-dev @angular/cli@latest

=======================================================================

Em outra aula, foi componentizado o header (que já estava no projeto) e foi incluído um corpo da index do projeto de template no projeto meat construído em angular.

Nada novo xD

=======================================================================

Rotas e navegação entre componentes

Na maioria das aplicações existe algo que é fixo, e algo que muda de acordo com a navegação do usuário

Ilustrando isso

Na maioria das páginas existe um header (cabeçalho) fixo, e tem um conteúdo que muda de acordo com a navegação

De acordo com q a url muda, só muda o corpo A cada url, um corpo novo

Para colocar um conteúdo dinâmico na página, é necessário uma tag chamada router-outlet que marca uma região de conteúdo dinâmico.

Para isso é necessário mapear cada componente em uma rota.

Rotas são basicamente um array que contém um mapeamento de cada caminho pra um component, rotas aceitam parâmetros usando a sintaxe de dois pontos seguido do nome do parâmetro

Dando exemplo de uma rota:

export const ROUTES: Route = [

{ path: ‘’, component: RestaurantsComponent },

{ path: ‘restaurant/:id’, component: RestaurantComponent },

{ path: ‘about’, component: AboutComponent }

];

Por fim, é preciso indicar que rotas são usadas em um módulo atráves da função forRoot na raiz ou forChild através de outros módulos (será visto em outro momento no curso).

Exemplo:

@NgModule({

declarations: [...],

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

})

export class AppModule

Como os caminhos são acionados? Como consigo navegar?

O módulo de roteamento disponibiliza uma diretiva chamada routerLink onde podemos passar um caminho a ser acionado ou um conjunto de parâmetros.

Podem ser usadas das seguintes maneiras

**<a routerLink=”/restaurants”> Restaurantes </a>**

ooou

**<a [routerLink]=”[‘/restaurants’]”> Restaurantes </a>**

Criando as primeiras rotas

Para começar a trabalhar com roteamento, nesse momento é preciso criar mais um component, que será o component da página **sobre** (about), que terá um conteúdo fixo e depois disso será visto como se define rotas e como navegar. É importante isso ser feito agora pois uma aplicação angular depende muito de rotas e roteamento, pois tem vários components que serão usados

Criando o component about

Será copiado o conteúdo do projeto de template para o component criado.

Feito isso, é hora de começar a criar as rotas

Vamos começar a criar as rotas em um arquivo separado (o cara não vai falar sobre módulos ainda porque não está afim)

Será criado um arquivo chamado app.routes.ts na pasta app, onde serão declaradas as rotas.

Para usar as rotas, é necessário importar o módulo de rotas: Segue import:

**import { Routes } from "@angular/router";**

Feito isso pode ser definido o conteúdo do array Routes.

O documento nesse momento fica assim:

*import { Routes } from "@angular/router";*

*import { HomeComponent } from './home/home.component';*

*export const ROUTES: Routes = [*

*{path: '', component: HomeComponent}*

*];*

Quando um componet é exportado, ele precisa ser importado, então por isso que importei o componente (obviamente)

Depois de criado esse arquivo, deve-se importar ele no **modulo principal**.

Depois de importar isso, deve-se declarar esse ROUTES como argumento de um método forRoots do import RouterModule, ficando assim:

RouterModule.forRoots(ROUTES)

Depois disso, deve-se ir no template do componente principal, e deve-se usar a diretiva <router-outlet>, para marcar a região dinâmica do template.

Para usar links que darão acesso ao template dos outros componentes, deve-se trocar os links por uma diretiva que acessa aos routers. Como é essa diretiva?

<a routerLink=”/restaurants”> Restaurantes </a>

ooou

<a [routerLink]=”[‘/restaurants’]”> Restaurantes </a>

Também temos uma diretiva para indicar que quando o link estiver ativo, será ativado alguma classe

A diretiva é o routerlinkActive, que pode ser usado da seguinte forma. (link para router ativado? Se sim, ative a seguinte classe, se não, deixe desativada)

<li routerlinkActive="active">

Quando esse routerLink estiver ativo, a classe active também será ativada sobre o list item.

Criando o componente de Restaurantes

Nessa aula, será criado o componente de restaurants, será criado outros componentes individuais.

Nessa aula será criado o component de restaurants. O conteúdo também será copiado do projeto template.

Primeira coisa a se fazer é arrumar as rotas. Existe um link na página principal que leva para o componente de restaurants e também há um link no header que leva para esse componente.

Então, depois de criado o componente, esse é cadastrado nas rotas e é adicionado os links com routerLink nos componentes header e home.

Próximo passo é criar um componente individual para cada item da página restaurants, porque, se observado, as informações se repetem entre os itens.

Por ter informações repetidas, esses itens dessa página são fortes candidatos a ser componentes

Vai ser criado um componente individual que representa os dados individuais de cada restaurant.

Esse componente será criado na mesma pasta de restaurants, porque o professor quer que os componentes estejam próximos aos componentes relacionados.

Como isso será feito?

Para as versões recentes do angular cli, pode ser feito da seguinte maneira:

ng g c restaurants/restaurant

Será criado um componente dentro do outro da pasta de outro componente xD

Criado o componente de um restaurant individual, será observado o componente restaurants.

Lá, pode ser visto que para cada restaurante as informações link da imagem, nome, estrelas, detalhes e tempo de espera sempre se repetem. Esse pedaço vai ser tirado do componente de restaurants e será levado para o componente restaurant. (Atenção se existe ‘s’ no final)

Esse componente individual precisa dos dados do restaurant para poder trabalhar, então o que pode ser feito? Criar uma propriedade com nome de restaurant e passar as propriedades desse elemento para o componente individual, em algum momento essas informações serão passadas para esse componente.

Nesse momento, será criado uma model que identifique os dados que serão passados, e essa model será utilizada no componente com o decorator input, para que ele possa receber esses dados de algum lugar.

Primeiro, será criado o model, que será um arquivo chamado restaurant.model.ts, que irá declarar uma interface definindo um restaurante. (basicamente uma definição do objeto)

Esse modelo que será usado como tipo do array de restaurantes criado no componente restaurants. Lembrando que precisamos importar o tipo lá em cima do arquivo, por que estamos lidando com es2015, exportou de um lado, importa do outro.

Após isso, é preciso mexer com o template do componente, usando de template interpolation ou property binding para vincular as propriedades do restaurante que serão passadas para o componente ao template.

O caminho ainda não será criado.

Agora é preciso mudar a página de restaurants, pois os restaurants serão colocados lá usando um \*ngFor. Como ainda não aprendemos sobre serviço, esses restaurants devem vir da memória, então deveremos mexer na classe do componente.

Depois de criado o array com alguns restaurants (no caso 2, que pegamos do arquivo db.json no projeto) será modificado o template, agora será usado o componente restaurant no lugar dos vários restaurants que foram usados de maneira fixa.

Será usado de fato o \*ngFor.

Algumas considerações importantes. O ngFor deve ser declarado na tag que irá se repetir, usando uma iteração com of (recurso ES6).

Aquela página que antes estava bem poluída ficou simples, pois agora só é preciso usar essas linhas para repetir tudo

<div \*ngFor="let restaurant of restaurants" class="col-sm-6 col-xs-12">

<mt-restaurant [restaurant]="restaurant"></mt-restaurant>

</div>

Lembrando de uma coisa

No componente restaurant, ele espera receber um input, um objeto que siga a interface Restaurant. Nessa sintaxe, o [restaurant] é um meio de definirmos aquela propriedade usando property binding, e o argumento passado é um item da iteração do ngFor. Veja bem que o nome é o mesmo no ngFor e na declaração da property binding. Não se pode usar sem o property binding nesse caso, pq o input espera um objeto, e dessa forma que é passado um objeto, de outra forma (sem property binding), somente uma string seria passada.

**O que é injeção de dependência?**

Injeção de dependência **é um padrão de projeto** onde é possível **deixar de instanciar os objetos manualmente** e passa a depender do framework para obter os objetos que a aplicação precisa (**automatizar instanciação de objetos necessários**)

O framework gerencia a instanciação dos objetos e também suas dependências e disponibiliza isso para os componentes da aplicação

Segue exemplo:

Se um componente precisa utilizar um serviço, podíamos instanciar isso manualmente. Mas com injeção de serviços, podemos passar isso no construtor do componente, e o framework vai instanciar esse serviço quando ele quiser ou precisar xD

Ao inverter essas responsabilidades, o código fica mais testável e mais limpo

Para deixar um serviço disponível para ser injetado em um componente, esse deve ser declarado na lista de providers e devidamente importando em um componente ou modulo

Se for em um componente, o serviço fica disponível para o componente ou seus filhos.

Se for declarado no modulo principal, esse serviço fica disponível para todos componentes da aplicação.

**O que é um serviço?**

Serviços **são classes comuns** em angular que podem ser injetados em componentes ou em outros serviços. Serviços são geralmente usados em uma aplicação para encapsular o acesso a api de backend, isso vai acontecer muito em uma aplicação angular.

Serviços podem ser singletons, ou seja, só ter uma instancia deles, e são ótimos candidatos para guardar dados compartilhados que servem toda a aplicação (cuidado com isso), se quiser, também pode ser usado serviços para suprir somente parte de uma aplicação.

No geral, são 3 escopos que podem ser utilizados para declarar um serviço.

~> Módulo angular ~> onde serão declarados na lista de providers. Se declarado no modulo raiz, ficará disponível para todas as classes disponíveis no mesmo módulo, componentes e outros serviços também. Todos eles irão compartilhar a mesma instancia desse serviço.

Para outros módulos que não sejam o módulo raiz existe algumas particularidades que serão discutidas à frente.

~> Componentes e seus filhos ~> se declarado na lista de providers de um componente, um serviço vai ser instanciado e compartilhado apenas para o componente e seus componentes filhos. Mesmo se um serviço de tipo idêntico existir como provider em um modulo, o componente receberá uma outra instancia.

~> Somente Component ~> se declarado na lista de viewProviders de um componente, esse serviço será usado somente só no componente, não podendo ser usado nos seus filhos.

Serviços também podem ser solicitar injeção de outros serviços, para isso ele deve declarar o decorator @Injectable(), mas atenção, esse decorator não é necessário para um serviço que será injetado em outro objeto, apenas para receber injeções do framework.

Esse serviço é importado da seguinte maneira:

import { Injectable } from ‘@angular/core’

Segue um exemplo

import { Injectable } from ‘@angular/core’

import { Http } from ‘@angular/http’

@Injectable()

export class MyService{

constructor (private http:Http){ }

list(){

return this.http.get(‘/url’);

}

}

Nesse exemplo vemos um serviço marcado com injectable recebendo um serviço do framework, que é o Http.

Alguns serviços que o próprio framework fornece:

Title, http e router.

http ~> encapsula o acesso a http

router ~> realiza a navegação de forma programática

Será falado sobre eles em outro momento

Title ~> serviço para obter e alterar o título de uma página

O componente pode requisitar a injeção e usar o método setTitle para substituir para o título que quiser.

Esse serviço é necessário pois não é possível utilizar expressões angular na página html inteira, pois o título fica no header, e essa parte não faz parte do bootstrap, foi criada essa alternativa como serviço.

Exemplo:

import { Title } from '@angular/platform-browser';

@Component({

viewProviders: [Title]

})

export class MyPageComponent{

constructor(title: Title){

title.setTitle(' PÁGINA DO GERALDÃO');

}

}

Criando a classe de serviços para restaurante

Continuando o projeto, será criado uma classe de serviço

Um serviço geralmente é utilizado para acessar o backend de uma aplicação

Pode ser declarado em vários lugares como já visto, no modulo ou no componente.

Dependendo do lugar onde foi declarado, altera a visibilidade e também como a instancia vai ser feita pelo framework.

A classe de serviço pode ser gerada com o angular cli, mas o professor vai fazer na mão, dentro da pasta de restaurants, para uma melhor organização.

O arquivo terá o nome restaurant.service.ts (o padrão deve ser com esse ‘service’ no nome).

Essa classe de serviço não terá nada de especial, apenas será criada a classe e será injetada em algum componente. (Provavelmente restaurants xD)

Essa classe foi criada normalmente, utilizando apenas um export.

É importante ter um construtor nela, então já foi criado um construtor vazio.

Essa classe terá um método que retornará os restaurantes, sendo possível tirar aquele array de restaurantes do componente Restaurants.

Pq vai ser colocado aqui? Pq é a classe que fica mais próxima dos dados, é a classe que vai fazer o consumo da api.

Por enquanto, a classe de serviço está ok somente retornando dados sobre alguns restaurantes.

Mas como vou consumir isso no componente?

Primeiro devo injetar minha classe de serviço no meu componente. (obs, um atributo de deve haver um atributo de restaurants ainda no componente Restaurants)

Como vou injetar isso? Utilizando o construtor que no caso será utilizado com o modificador de acesso private. Lembrando que, ao utilizar modificador de acesso na frente de um parâmetro do construtor, o ts gera um atributo com aquele nome. Então o que vai acontecer com esse construtor? Foi declarado uma propriedade e será instanciada a classe de serviço, tudo em uma linha.

Aproveitando para falar sobre os métodos do ciclo de vida do componente:

Sempre que é criado um componente com o angular cli, este implementa a interface OnInit e o método necessário ngOnInit, esse método vai ser chamado uma vez no ciclo de vida do componente.

Fiquei em dúvida pela contradição. Segundo o professor, toda vez que o componente entrar na tela, será instanciado um novo componente e irá executar o construtor e após isso o ngOnInit.

Então esse ponto é um ponto ideal para fazer uma inicialização do componente com algo que dependa do construtor.

Então o que iremos fazer nesse método?

Popular o atributo restaurants

Como?

this.restaurants = this.restaurantService.restaurants();

Com essa linha, mas falta alguma coisa

É preciso dizer ao angular quem é o restaurantService, por enquanto só foi criada uma classe e essa foi posta no construtor

O que falta?

Nesse ponto, não deu nenhum erro no console do compilador do webpack, porém, se tentarmos acessar a página restaurants na página, dará um erro bem doidera no console.

Sem providers para um serviço é o erro, então adivinhe...

É o que falta, então declaramos esse service no providers do modulo principal

Declarou, funcionou

**Reactive programming e http**

A ideia de programação reativa é simples:

Um evento acontece, os que estão interessados são notificados e reagem a ele.

Baseado no design patter observer, (também baseado em eventos)

Nesse padrão, tem o objeto foco e objetos interessados em mudanças que ocorrem nesse objeto, os “*listeners”.* Esses objetos se inscrevem esperando uma mudança, ou evento. Quando esse evento acontece, o listener realiza uma alguma ação.

Esta é a base.

Em reactive programming, os eventos vêm em forma de streams, ou seja, sequência de eventos q podem ser modificados ou transformados em uma nova cadeia de eventos.

Cada evento que acontece gera uma notificação nos objetos interessados

Um exemplo:

Imagine que tem 5 objetos que realizam alguma ação e tem um outro objeto que está escutando esses eventos.

Os eventos dos 5 objetos vão acontecendo 1 a 1, e a cada evento, o listener é notificado e realiza algo.

Então, pode-se dizer também que reactive programming é a junção de iterator + observer.

Pq iterator? Pq os eventos vão sendo iterados item a item em uma stream(Isso me lembra algo xD), e observable pq a cada evento algum outro objeto ouvinte vai realizar algo.

Em angular, várias partes do framework são implementados em cima desse conceito, por exemplo:

Ao realizar uma chamada http ao backend, parte da aplicação se inscreve e espera pela resposta, que é o evento, quando o evento acontecer (resposta chegar), a aplicação reage a ela e extrai os dados necessários para preencher um componente ou alguma informação na tela.

Outro exemplo:

Quando você precisa saber quando os parâmetros das rotas mudam, mesmo padrão, se inscreve e espera pela resposta!

A biblioteca que o angular usa para reactive programming é o rxjs (Reactive Extensions for JavaScript). Nessa biblioteca o objeto principal é observable. No dia a dia não vai ser tão criado objeto desse tipo, pq na maioria dos casos a aplicação vai estar ouvindo eventos produzidos pelo próprio angular.

O observable possui uma série de funções, ou operadores para inscrição e manipulação de eventos.

Existe uma similaridade dessas funções e operadores com as funções auxiliares de array, segue exemplo:

[1,2,3,4,5].map(x => x+ 3 ).filter(x => x/ 2 === 0).forEach(x => console.log(x)); // 4,5,6

Observable.from([1,2,3,4,5]). map(x => x+ 3 ).filter(x => x/ 2 === 0).forEach(x => console.log(x)) // 4,6,8

No Exemplo, pode ser visto métodos comuns a array e observables, como map e filter, que contém o mesmo comportamento. De diferente, tem o subscribe, que é muito similar ao forEach, pois cada item do array será passado para a função do argumento do método subscribe.

Por que não promisses?

Grande diferença entre promisses e observable é que observable continua disparando eventos até que sejam explicitamente fechados e promises são resolvidas depois do primeiro exemplo. Observable + flexível então como para usar web sockets por exemplo.

Exemplo

this.http.get('/url')

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

.subscribe(data => this.mydata = data);

Se for olhado esse exemplo da api http do angular, pode se observar que os métodos retornam objetos observable<response>, um dos principais objetos da lib rxjs

Na maioria dos casos iremos transformar a resposta e recolher o dado a requisição é feita sempre depois que a aplicação se inscreve usando o método subscribe (lembre-se que poderemos usar todos os verbos rest)

rxJs permite coisas interessantes como:

facilmente refazer as chamadas http com o método retry como no exemplo

this.http.get('/url')

.retry(2)

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

.subscribe(data => this.mydata = data);

Ou fazer vários mapeamentos para deixar a resposta da forma esperada

this.http.post('/url', JSON.stringfy( mydata ))

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

.map(response => result.id)

.subscribe(id => this.id = id);

Quando um objeto se inscreve em um observable, é necessário remover a inscrição posterior para evitar memories leak, mesmo que um componente saia de cena, ou seja, a página muda e o componente é descartado, o listener inscrito pode continuar sendo chamado, por isso devemos cancelar a inscrição, mas o angular traz algo positivo.

Os observables retornados da api http, pelos parâmetros do router e pelo pipe async não precisam de unsub

Geralmente quem acessa a api http do angular são serviços da aplicação, então é uma boa prática tratamento de erro básica nessa parte da aplicação com o operador catch, depois de um erro o observable vai parar de enviar eventos, o método catch recebe o objeto de um erro e deve devolver um outro observable , segue exemplo:

save(myData){

return this.http.post('/url', JSON.stringify(myData))

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

.map(result => result.id)

.catch(error => {

console.log(error);

Observable.throw(`Error posting ${myData}`);

})

}

Nesse exemplo o observable propaga um erro para cima.

Configurando a API de Backend (json-server )

Como o próximo passo é utilizar HTTP, será necessário mockar a API de backEnd.

Para isso, será usada uma biblioteca para o node chamado json-server, que pega um arquivo chamado db.json e simula uma api rest.

Como assim? Nesse arquivo, terá um objeto JSON com propriedades apontadas para arrays, cada array será um backend.

Como instalar?

Npm install –g json-server

Depois de instalado, precisamos startar o processo, que vai carregar aquele arquivo e simular uma api.

Como?

Json-server db.json ~> db.json é o nome do arquivo

Se tudo der bom, vai ser mostrado no terminal umas mensagens e alguns endereços, abra um deles para conferir se está tudo sob controle.

Com isso, o “backend” ta pronto.

Próxima aula, modificar o serviço criado no projeto para consumir essa “api”

Adicionando Http ao serviço de restaurantes

Com o back end no ar, vamos consumir isto.

Da paara perceber pelo endereço da “api” que ela fica em uma porta diferente. Ela fica na porta 3000.

Primeiramente, será criado uma constante com esse endereço, para caso seja modificado futuramente fique fácil de alterar, já que estará em um lugar e basta mudar neste.

Será criado um arquivo na pasta raiz do projeto, com nome app.api.ts apenas com o propósito de de exportar uma constante com o endereço da api. No caso, o professor nomeou como MEAT\_API

Após isso, será modificado o serviço

Primeiro passo: importar o endereço de consulta a API.

Agora não será mais necessário o array em memória para simular o acesso a uma api

Assim a gente pode criar serviços e injetar em módulos ou componentes, o angular também disponibiliza alguns serviços para serem injetados na nossa aplicação, tanto nos componentes, módulos ou serviços.

Um deles é o serviço HTTP, que será utilizado.

Um porém, para uma classe de serviço receber outro serviço via injeção de dependência, precisamos marcá-la com o decorator @injectable. Lembrando que não é preciso esse decorator caso a classe não acesse um serviço (como o http) mas como vamos receber outro serviço, precisamos marcar com esse decorator.

Como vamos utilizar o serviço HTTP, também iremos importar este do angular.

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

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

O topo da página nesse momento irá ficar assim

Em termos de import do angular, será isso.

Próximo passo é receber a injeção do serviço Http, para isso será criado um campo no parâmetro do construtor da classe de serviço utilizando o modificador de acesso private (sintaxe reduzida naquele padrão já conhecido)

constructor(private http: Http) { };

E agora? Consumimos a api

O método que retornava aqueles restaurantes vai ser refatorado.

Vamos utilizar o serviço Http fornecido pelo angular para consumir a API.

O serviço http tem uma série de métodos que representam os verbos do protocolo Http (get, put, post...).

Para o nosso exemplo, será utilizado o get passando uma url para retornar os restaurantes. A resposta vai vir o json com os dados dos restaurantes que estão guardados em um backend mockado. Há de se lembrar que todos os métodos da api HTTP retornam um objeto Observable que é um objeto da lib rxJS que já foi discutido o que significa.

Então, serão necessárias algumas adaptações para funcionar, até pq essa api trabalha de forma assinc, não sendo possível realizar verificação de tipo no momento da chamada do método.

Então pora, em vez de retornar Restaurant, iremos retornar um Observable de Restaurants (Observable pode ser tratado como uma classe genérica (lembrar de generics do java))

Obviamente que, ao adaptar o método, precisamos também importar, e parece que deve ser importado direto da lib rxjs.

Mas coé, continua dando erro, qual foi agora?

Checando o log do próprio visual code, pode ser visto que o que está sendo retornado é um Observable<Response> e não um Observable<Restaurant[]>, E agora josé?

Por que deu isso? Pq ainda não fizemos uma fodendo mapeação da resposta (muito importante)

Quando fizermos uma requisição, realmente será retornado um observable<Response>, só que response representa uma resposta crua, com todas as informações que uma resposta tem, como corpo, cabeçalho, statusCode, essas paradas que você viu com ajax. Só que não precisamos de tudo isso, precisamos apenas do objeto Json que essa requisição tinha como objetivo.

Precisamos usar o operador map para transformar essa resposta em um objeto Restaurant, BEM NICE.

Devemos transformar o objeto JSON que essa response está fornecendo para um array de Restaurants com o método MAP. Mas ae, meu editor de código super robusto com seu auto complete não ta mostrando a disgrama do map, qual foi? É pq devemos importar essa porá manézão sem teta, bora bora, agilidade, a linha é esta:

import 'rxjs/add/operator/map';

Caralho em, poggers, mas ae, como vou usar essa birosca? Falou falou e não falou nada

Com essa fodendo linha desgraça (A explicação anterior foi para chegar nesse momento):

restaurants(): Observable<Restaurant[]> {

return this.http.get(`${MEAT\_API}/restaurants`).map(response => response.json());

}

Gostozin no azeite.

Então, pq devemos mapear? pq a fodendo requisição com esse serviço retorna um response, e response tem dados dms, não queremos todos eles, então mapeamos essa resposta e extraímos somente o que queremos!

Agora o bagulho parou com os erros.

Mas ae, a chamada vai ser feita com esse fodendo map?

Não mané, ta cantando de galo a toa, tu tem que fazer o subscribe. Lembra do conceito de Observable? Então (se não lembra, volte uma ou duas aulas).

Apenas depois do subscribe que a fodendo requisição vai ser feita.

Onde fazemos o subscribe?

La aonde vai ser consumido esse método, até pq lá ta reclamando pra caralho.

Esse lá é no componente restaurants.

Então, voltando na classe componente, não iremos mais atribuir diretamente como fazíamos antes.

Como o método foi modificado na classe de serviço e agora retorna um Observable, precisamos fazer um subscribe e precisamos também passar uma função pra esse método, essa função será chamada sempre que algum dado chegar.

Como vai ficar esse método?

restaurants: Restaurant[];

ngOnInit() {

this.restaurantService

.restaurants()

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

}

Copiei o restaurants para fins explicativos.

Então, vamos analisar, aqui estamos chamando o método da classe de serviço que retorna um Observable<Restaurant[]>, então, como retornou um observable, estamos adicionando algum comportamento ouvinte a esses eventos que irão ocorrer (no caso, a chegada da resposta).

No método subscribe, passamos uma função que recebe um parâmetro e então atribui esse parâmetro ao atributo da classe restaurants, declarado acima.

Então qual a ordem do bagulho?

A sacada é que o negoço é feito de modo async, então é meio baixaria.

Depois do método subscribe que será chamado o serviço que busca informações daquela outra url. Na chegada dessas informações, elas serão mapeadas, pegaremos só o que importa dessa resposta, e então utilizaremos isso, atribuindo essa resposta mapeada a um atributo usável.

Acessando a página, pode ser visto mais restaurantes agora. Então tá funcionando, OBA OBA. É ISTO por HOJE.

**Tratamento erros com o operador Catch**

Essa aula tratará a manipulação de erros e o operador catch dos observables

Quando é trabalhado com observable geralmente não fazemos uma ação e sim uma configuração.

Por exemplo, se der uma olhada no serviço criado, o método get não executa uma ação, ele só configura o que será acessado e como será mapeado, a ação só irá ocorrer quando alguém se inscrever.

Os outros serviços serão basicamente iguais a esse. A ação só é executada quando os componentes ou outros em alguma outra parte da aplicação chamar o método subscribe do observable.

Existem duas formas de tratar erros no angular. Uma é implementado a interface ErrorHandler e criando uma classe que faz um tratamento de erro global e declarando essa classe como um provider da aplicação. A outra é utilizar o operador catch do observable

Para utilizar esse operador, primeiro ele deve ser importado da seguinte forma:

import 'rxjs/add/operator/catch'

E usar esse método catch na classe de serviços. Esse método recebe uma função que recebe um response e retorna um observable. Quando houver um erro, esse observable irá ser fechado. Na hora de tratar o erro, o desenvolvedor irá ter uma chance de logar o erro, devolvendo um outro observable, decidindo se irá ser trocado para uma outra sequência ou vai ser disparado o erro.

Como essa função irá ser usada em vários pontos, será criada uma classe estática em outro arquivo, pois nos outros métodos e serviços também será usado esse tratamento de erros básico.

Será criado uma classe no diretório raiz de nome app.error-handler.ts, com o seguinte código:

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

import { Observable } from "rxjs/Observable";

import 'rxjs/add/observable/throw';

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);

}

}

Nessa classe, é possível ver um método estático e essa mensagem é logada.

E possível observar que esse método retorna um observable com a mensagem construída.

É possível testar essa funcionalidade trocando o endereço no serviço.

**Parametrizando as Rotas**

Já é sabido como navegar utilizando rotas, pois já foi abordado em aulas passadas, mas em algum momento será preciso uma rota parametrizada onde parte dela vai identificar algum objeto armazenado no backend. Como pode-se ser feito isso?

Nas aulas passadas já foi informado que pude passar um parâmetro variável para uma rota utilizando a seguinte sintaxe:

export const ROUTES: Routes = [

{ path: 'restaurant/:id', component: RestaurantComponent }

];

Para passar o valor para a rota, é utilizado a diretiva routerLink, que aceita um array com os parâmetros, exemplo:

<a [routerLink]="['/restaurants', restaurant.id]">Ver Restaurantes</a>

Nesse exemplo, é possível ver que é passado o caminho /restaurants, e o angular deve concatenar com o id, sendo assim, cada parâmetro separado por virgula é separado por barra.

Existem duas formas de obter o valor do parâmetro:

Snapshot: fotografia dos parâmetros no momento que for pedido

Subscribe: se inscrevendo na rota e escutando a mudança de parâmetros.

No curso, será abordado os dois modos.

No caso do Snapshot é preciso injetar o objeto ActivatedRoute (já sabemos que pra injetar algo em um objeto precisamos do @Injectable() bla bla bla...). Esse objeto representa a rota ativa no momento

Para obter o parâmetro id a partir do snapshot, é da forma mostrada no exemplo, e a partir da posse desse valor, é possível fazer uma busca no backend para obter informação:

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

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

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

@Injectable()

export class MyComponent implements OnInit {

myObj: any;

constructor(private router: ActivatedRoute) { }

ngOnInit(): void {

const id = this.router.snapshot.params['id'];

this.myObj = // ... obtendo os dados baseado no id

}

}

O snapshot é muito útil quando não ligamos de a rota mudar, e o componente pode ser descartado e recarregado, sendo destruído e criado novamente. Esse é o caso mais comum e que será mais utilizado.

Porém, há casos que o componente não pode ser destruído e criado novamente, será tipo que mudado sem recarregar, estilo ajax, parada doidera.

Imagina o seguinte caso:

Existe uma página no site com uma div ocupando a parte esquerda da tela com alguns itens e outra div na parte direita da tela com itens que serão carregados de acordo com os itens da div anterior. É isto! Para isso será utilizado o subscribe. Se for reparado, as duas divs serão components e não poderão sair da tela.

Mas como saber que o item mudou?

Pra isso, devemos relembrar os princípios.

Muitas coisas no angular são baseadas em reactive programming, aquele estilão que mistura iterator com observable. Sendo assim, você se inscreve para ouvir um evento e quando ele acontece você é notificado e responde a ele. Quem fornece essa capacidade ao angular é a lib rxjs. Partes muito importantes do angular como http e rotas são implementadas em cima do padrão de reactive programming.

Então, voltando a pergunta. Como um componente que não sai da tela sabe que o parâmetro de rota mudou?

Um componente se inscreve, a cada mudança na rota, ele reage. xD

Como vai ser implementado isso? Quase da mesma forma do snapshot, também será utilizado o serviço ActivatedRoute, mudará apenas a forma de capturar o id, que será com outro método.

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

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

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

@Injectable()

export class MyComponent implements OnInit {

myObj: any;

constructor(private router: ActivatedRoute) { }

ngOnInit(): void {

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

const id = params['id'];

this.myObj = // ... obtendo os dados baseado no id

});

}

}

Através do objeto da classe ActivatedRoute, será utilizado o método params.subscribe para notificar qualquer mudança obtida da rota. Ao invés de snapshot, é utilizado o subscribe pois toda vez que a rota mudar é possível fazer uma busca pelo dado através do id gerado de forma “mais dinâmica”.

**Criado componente de detalhe de restaurante**

Essa aula só vai ser só criar detalhe de restaurante

Vamos criar um componente chamado restaurant-details, copiar o conteúdo html e colar no componente, padrão.

Comando utilizado no terminal:

**ng g c restaurant-detail --spec=false**

Padrão, já conhece isso.

Criei o componente, o que fazer?

Cadastrar rota, mudar links, vamo então.

Como cadastrar? Modelo dinâmico, daquele jeitão que a rapaziada gosta e foi aprendido na última aula

{ path: 'restaurants/:id', component: RestaurantDetailComponent },

Cadastrei a rota, agora vamos mudar o componente de restaurant, em um href que ficou vazio.

Como vai ficar?

Vamos ver se funciona até aqui, se sim, vamos componentizar.

SUAAAAVE MEU LINDO, agora temos que componentizar o menu e as avaliações.

Então vamos criar um componente dentro desse componente de detalhes chamado menu.

Qual comando meu chegado?

**ng g c restaurant-detail/menu --spec=false**

Depois disso, ritual padrão de tirar o css e o metadado styleUrls

Retira o menu do componente restaurant-detail e coloca no template do menu.

Depois disso também vamos criar o carrinho de compras e dos itens do menu, que se bem observado, são informações que se repetem, coisa liandra.

**ng g c restaurant-detail/shopping-cart --spec=false**

**ng g c restaurant-detail/menu-item --spec=false**

Com isso feito, primeiro vamos componentizar a parte do carrinho de compras, porque é mais fácil.

Ritual padrão, copiar e colar do componente restaurant-detail. Não esquecer de usar a tag do carrinho de compras dentro do componente do menu. Se funcionar humilde, go next!

**Serviço http para detalhe restaurante**

Como essa parte de detalhes vai ser o inferno, vamos começar pela parte de cima, preenchendo os detalhes.

Já temos um serviço para consultar os detalhes de um restaurante em especifico.

Vamos voltar no nosso serviço e vamos criar um novo método, porém, dessa vez iremos retornar um restaurant só, uma busca por id.

Como será feita essa busca?

Por essa url:

${MEAT\_API}/restaurants/${id}

Por que isso pode ser feito? Pq o backend mocado funciona de forma igual a uma api rest, como ela deve se comportar.

Se eu tenho um recurso chamado restaurants, deve ser retornado todos os restaurants. A partir do momento em que é passado um /id (barra ai di), onde esse id vária, deve ser retornado um elemento em especifico que corresponde a esse id.

Como ficará nosso método feito?

restaurantById(id: string): Observable<Restaurant>{

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

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

.catch(ErrorHandler.handleError);

}

Só de bater o olho no método já dá para saber do que se trata, não vou ficar explicando de novo.

Então, agora vamos consumir esse serviço criado no componente de detalhes do restaurant.

Como faremos isso? Primeiro injetamos esse serviço no componente de detalhes do restaurante (nada novo). Obs.: Como esse serviço não é novo, não precisamos declarar ele no modulo principal da aplicação (aquela parte de providers tão conhecida).

Então vamos injetar via parâmetro do construtor nosso serviço no componente restaurantDetails.

Após isso, iremos utilizar esse serviço no método ngOnInit, que é o método ideal para esse tipo de situação. Nele iremos fazer a consulta baseado na situação.

Para isso, iremos adicionar uma propriedade chamada restaurant do tipo Restaurant(criamos esse tipo faz alguns dias).

Após isso realizaremos a consulta via subscription no método criado. O corpo ficará assim:

this.restaurantService.restaurantById('').subscribe(restaurant => this.restaurant = restaurant);

Mas como iremos passar o id para esse método? Temos que capturar isso da url e mandar como parâmetro.

Aí é que lembramos da nossa aulinha passada, onde falou sobre algo onde escutamos as mudanças da url, algo com snapshot ou subscribe.

Para isso, utilizaremos um objeto do angular chamado ActivatedRoute, que prevê informações sobre as rotas como url acionado e parâmetros passados para essa url. Vamos precisar desses parâmetros para saber sobre id que vamos passar para esse método.

Então vamos injetar esse objeto também.

Podemos acessar por snapshot ou subscribe, dessa vez cabe utilizar pelo snapshot, não estamos ligando se o componente vai ser destruído e depois construído novamente.

Como isso será feito?

let id = this.route.snapshot.params['id'];

Com essa linha, deixei desse modo para ficar mais fácil de visualizar.

Esse id que será utilizado como argumento do método restaurantById.

Esse params utilizado é um objeto que contém valores sobre os parâmetros da rota utilizada. Parece ser um array ou um array like onde pegaremos o parâmetro pelo “nome” dele. Ou seja, pegaremos o parâmetro id.

Agora temos que olhar o console para ver se não vai dar nenhum erro. Nada de errado, podemos continuar.

Nesse momento que começamos a colocar as informações de maneira dinâmica no componente restaurant details.

Mudando a informação sobre o nome do estabelecimento, já pipoca um erro falando que não podemos ler a propriedade name de undefined. Mas por que está retornando um undefined?

Por que tem todo um delay entre o carregamento da página e o consumo da api. Digamos que a página é renderizada antes do retorno do observable, que vai sofrer um mapeamento e etc.

O que fazemos para resolver isso?

Podemos utilizar o operador de navegação segura, que é a interrogação após o parâmetro da interpolation.

Por que funciona agora?

Pq agora o angular sabe que em um primeiro momento, aquela informação realmente pode ser undefined, mas está sendo entendido pelo operador que não é para ele ligar pra isso, carrega depois ai quando chegar, se chegar.

Tendo a seguinte interpolação:

{{restaurant?.name}}

É falado o seguinte para o angular: Depois da interrogação, só carregue se tiver dado, se undefined, espere.

Agora faremos isso para o resto das informações. (Acho meio moios deixar tudo com interroga, mas é o framework, fazer o que né?).

Toda vez que muda de restaurant, também está dando um erro, pq? Sempre que é mudado de página, está sendo buscado de maneira rápida por um id vazio da imagem, por isso rola esse erro.

Para resolver isso, será utiliza utilizado a diretiva ngIf, renderizando a div que apresenta as informações do restaurant apenas se existir um restaurant. Agora sim, o erro foi embora. Ihul!!

Com isso funcionando, go next!!!