

Angular de cria

PROJETO USADO PARA EXPLICAÇÃO

Assim que você cria um projeto Angular é gerado algumas pastas e arquivos, hoje vamos entender cada um desses elementos abaixo:

node_modules = nessa pasta fica todas as dependências que foram instaladas no projeto.

package.json = fica listado as dependências do node modules.

Tudo que fica em dependencies é o que precisamos para rodar o projeto em produção depois.

"rxjs" = Para fazermos a parte de observables e chamadas HTTP.

"tslib" = Biblioteca do TypeScript já que usaremos o TS para codificar no nosso projeto Angular.

```
"dependencies": {  
  "@angular/animations": "^14.2.0",  
  "@angular/common": "^14.2.0",  
  "@angular/compiler": "^14.2.0",  
  "@angular/core": "^14.2.0",  
  "@angular/forms": "^14.2.0",  
  "@angular/platform-browser": "^14.2.0",  
  "@angular/platform-browser-dynamic": "^14.2.0",  
  "@angular/router": "^14.2.0",  
  "rxjs": "~7.5.0",  
  "tslib": "^2.3.0",  
  "zone.js": "~0.11.4"  
},
```

devDependencies fica todas as dependências que usamos somente no ambiente de desenvolvimento (não são adicionadas no nosso pacote quando vai para produção).

"Karma" = Biblioteca de testes.

"Jasmine" = Biblioteca de testes.

```
"devDependencies": {
  "@angular-devkit/build-angular": "^14.2.8",
  "@angular/cli": "~14.2.8",
  "@angular/compiler-cli": "^14.2.0",
  "@types/jasmine": "~4.0.0",
  "jasmine-core": "~4.3.0",
  "karma": "~6.4.0",
  "karma-chrome-launcher": "~3.1.0",
  "karma-coverage": "~2.2.0",
  "karma-jasmine": "~5.1.0",
  "karma-jasmine-html-reporter": "~2.0.0",
  "typescript": "~4.7.2"
}
```

angular.json = fica as configurações do nosso projeto e lá colocamos o **styles** principal (global), **scripts** no caso de import, **assets**.

`app.component.ts` = arquivo principal da aplicação.

```
app.module.ts = modulo principal.
```

`app-routing.module.ts` = tem o roteamento que faz possível criarmos outros modules.

`app.component.spec.ts` = arquivo de especificação para testes unitários.

PARA CADA COMPONENT CRIADO TERÁ UM ARQUIVO TS, HTML, SCSS.

/////////////////ANOTAÇÕES DO PROJETO

SENDO FEITO \\\\\\\

MODULE:

Modulo no Angular é como a gente organiza de forma lógica nossa aplicação, todos os componentes que a gente gera e cria dentro desse modulo eles estão visíveis apenas dentro do modulo, caso a gente queira utilizar um componente em outro modulo a gente precisa expor (vamos ter de exportar o componente) que é o caso do Toolbar que a gente importa o módulo do tool bar e o Angular exporta o materialtoolbar pra gente poder utilizar nos nossos componentes como diretiva por exemplo (mat-toolbar).

```
angular.json U  app.component.html U  app.component.ts U  app.module.ts U X
src > app > app.module.ts > AppModule
1  import { NgModule } from '@angular/core';
2  import { MatToolbarModule } from '@angular/material/toolbar';
3  import { BrowserModule } from '@angular/platform-browser';
4  import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
5
6  import { AppRoutingModuleModule } from './app-routing.module';
7  import { AppComponent } from './app.component';
8
9
10 @NgModule({
11   declarations: [
12     AppComponent
13   ],
14   imports: [
15     BrowserModule,
16     AppRoutingModuleModule,
17     BrowserAnimationsModule,
18     MatToolbarModule
19   ],
20   providers: [],
21   bootstrap: [AppComponent]
22 })
23 export class AppModule { }
```

Router:

Path no routes é para indicarmos o caminho.

pathMatch é para verificar o caminho.

```
const routes: Routes = [
  { path: '', component: CoursesComponent }
];
```

router-outlet é usado para falarmos para nosso HTML que nós estamos utilizando o roteamento.

```
courses-routing.module.ts U  app-routing.module.ts U  courses.component.html U  app.component.html U X
src > app > app.component.html > router-outlet
Go to component
1  <mat-toolbar color="primary">
2    <span>CRUD Angular + Spring</span>
3  </mat-toolbar>
4  <router-outlet></router-outlet>
```

Quando o Angular ver **router-outlet**(está vendo **localhost:4200**) que vai fazer o redirecionamento vai carregar o **app-routing.module.ts** (**MOSTRADO ABAIXO**) que irá redirecionar para **localhost:4200/course** e irá renderizar nossos **CoursesModule**. (AQUI ESTAMOS FAZENDO UM LAZY LOADING e nele damos um **loadChildren** que carrega o arquivo a ser mostrado no caso no import colocamos o caminho do arquivo (**./courses/courses.module**) onde está nosso modulo filho e dentro desse arquivo temos o module declarado que colocaremos após o importe (**then(m => m.CoursesModule)**)).

AQUI DEPOIS DE CRIAMOS OS NOVOS MÓDULOS/COMPONENTES NO NOSSO PROJETO COLOCAREMOS OS OUTROS MODULOS AQUI COM OUTROS LOADCHILDREN.

```
const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'courses' },
  { path: 'courses',
    loadChildren: () => import('./courses/courses.module').then(m => m.CoursesModule)
  }
];
```

Dentro do `courses-routing.module.ts` a gente colocou que quando for apenas `/courses` (`localhost:4200/courses`) iremos renderizar o `CoursesComponents`.

(MOSTRANDO ABAIXO)

```
const routes: Routes = [
  { path: '', component: CoursesComponent }
];
```

Lazy Loading:

O Angular faz o download do seu código inteiro de uma só vez no navegador. Esse é seu comportamento padrão, chamado de **Eager Loading** (ou carregamento "ansioso").

Lembra dos vários módulos que você projetou?

Exato, ou ele baixa todos os módulos, ou não baixa nada...

E antes que perceba, você encontra problemas de **lentidão** e **dificuldades** no uso do aplicativo.

Porém você pode trocar o comportamento padrão do Angular pelo **Lazy Loading** (ou carregamento "preguiçoso").

Onde você **divide seu código** em partes menores. Para que a parte mais importante seja carregada primeiro e depois todas as outras secundárias são carregadas **sob demanda**.

Assim vamos baixar apenas uma fração do pacote do aplicativo em vez do pacote inteiro. O que ajuda a melhorar seu **desempenho**.

Principalmente para os usuários que usam dados móveis ou conexões de Internet muito lentas.

TypeScript:

Como o Angular utiliza o TS ao iniciar um variável temos que **tipar** ela, se não fizermos isso ela vai ficar com tipo never, ou seja não terá ela.

Tipado do tipo **any** significa que estamos tipando a variável de um jeito "qualquer" que não sabemos o tipo ainda, mas sempre é bom não fazer essa tipagem com **any** quando possível (sabermos o tipo).

Ex de tipagem aonde estamos tipando a variável como do tipo **any**:

```
courses: any[] = [];
```

Para resolvermos isso vamos criar o tipo course para declarar nossa array do tipo course, então iremos criar uma pasta model dentro da pasta courses e criar uma interface dentro dela com o código:

```
ng g interface courses/model/course
```

Com esse código será criado a interface e poderemos colocar dentro dela todas as variáveis que vão ter nessa interface, LEMBRANDO QUE NO TypeScript essa interface só existe no ambiente de desenvolvimento depois que for compilado para o JavaScript que vai realmente rodar no browser não irá ter uma classe course que vai ser transpilada para JavaScript, ou seja isso é somente para ajudar a gente em tempo de desenvolvimento, para termos intelicense, para caso a gente modificar o nome da nossa propriedade vermos em quais lugares ela está sendo usada, caso você tenha alguma lógica de negócio que precise ser dentro do seu modelo aí sim usamos uma classe que vai ser transpilada para uma classe normal do JavaScript que na verdade tudo acaba sendo um objeto no JavaScript.

Vale lembrar também que no TypeScript temos 2 formas de indicar o String ou string.

String: Objeto

string: tipo

Não precisamos tipar quando já fazemos inferência de tipos:

```
displayedColumns = ['name', 'category']
```

?:

Operador que verifica se o valor que está chegando não é undefined ou null.

No Angular temos que inicializar as variáveis ou então inicializar ela no construtor.

Data Source:

No **data source** do html informamos o **array** com os dados e nas colunas a gente troca o nome para as colunas que tem nesse **array** (Nome, idade, etc).

Displayed Column: Informamos a **displayed columns** no nosso **component TS** para quando formos iterar de cada coluna fazer o link com **data source**, e também quando estiver no **let row** estará fazendo com que a cada linha (registro) que tivermos na nossa lista nós iremos mostrar as colunas que foram declaradas nessa variável de **displayedcolumns**.

```
<tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
<tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
```

Services:

Como principal objetivo do component é mostrar dados, podemos fazer a integração com o Backend (servidor) e não importa a linguagem que usaremos no backend. Como é de bom tom e padronizado, não escrevemos códigos de **lógica de negócio** nos nossos components e sim no **SERVICE** que será responsável por ter toda **lógica de negócio** e se comunicar com o **backend**, além de poder ser injetado em outras classes e para isso usamos **injeção de dependência** do **Angular**.

Como o **Angular** é um **framework** que define muito bem a arquitetura que teremos de usar na nossa aplicação e já fornece todas as ferramentas para podermos utilizar, temos o **Angular CLI** que vai cuidar de todo nosso projeto (todo ciclo de vida do projeto) e o usaremos para gerar o **service**.

```
ng g s caminho/services/nomedoservice
```

O **service** também é uma classe e podemos ver isso porque temos uma anotação no Services o **@Injectable** que vai fazer parte da injeção de dependências das classes dentro do **Angular**, esse **decorator** (Anotação) irá adicionar informações essenciais para o ciclo de vida daquele tipo de classe que existe no **Angular**, um exemplo é o **OnInit** que faz parte do ciclo de vida de um component do **Angular**.

@Component:

selector: Esse seletor é justamente o nome do nosso component (diretiva), é a tag HTML que a gente vai poder utilizar caso a gente exporte esse component.

`templateUrl`: Fica o caminho do nosso HTML.

`styleUrls`: Fica o caminho do nosso CSS.

O `service` em si é uma classe independente, a gente utilizará o `service` dentro do nosso component para ajudar na nossa `lógica`, por exemplo mostrar os dados que serão renderizados para o component que sem o `service` não saberia, o component não precisa saber de onde está vindo esses dados só que vai renderiza-los, quem cuida dessa parte do backend é o `service` e também nele iremos `manipular os dados` que chegam para deixar da forma que o component espera.

Para usarmos o `service` dentro do component se tivermos os dados em memória precisaríamos de apenas instanciar um novo objeto do tipo `Services` e atribuir o `service` a lista do component. (EXEMPLO ABAIXO MOSTRANDO O COMPONENT)

```
courses: Course[] = [];  
displayedColumns = ['name', 'category'];  
  
coursesService: CoursesService;  
  
constructor() {  
  // this.courses = [];  
  this.coursesService = new CoursesService();  
  this.courses = this.coursesService.list();  
}
```

Porém se tivermos na nossa aplicação dados vindo de fora da memória no caso da API, para fazermos isso precisaremos de uma chamada `Ajax` que será uma `chamada assíncrona para o servidor`, no `Angular` temos uma classe utilitária que fornece todos os métodos para podermos fazer essa conexão com nossa `API` que é o `HTTP CLIENT`. (IMAGEM DO CONSTRUTOR DO SERVICE ABAIXO)

```
constructor(private httpClient: HttpClient) { }
```

Quando fazemos isso o `Angular` vai fornecer o `httpClient` para nós automaticamente simplesmente por temos `declarado ele no constructor` isso é uma `injeção de dependência`.

Uma maneira de falarmos para o `Angular` que queremos que ele `instancie a classe automaticamente` é através da anotação `@Injectable` que nesse caso têm dentro do `HttpClient` que por isso ele está sendo instanciado automaticamente.

Por causa do `providedIn` dentro do `@Injectable` definido como `root` acaba que o `service` fica como `global`, ou seja pode ser utilizado em qualquer lugar da nossa aplicação, e como tudo no `Angular` precisamos importar o módulo do `HttpClient` e faremos isso no `app.module` para ele ficar disponível globalmente e o `Angular` irá conseguir fornecer a instância dessa classe para nós.

```
imports: [  
  BrowserModule,  
  AppRoutingModule,  
  BrowserModule,  
  BrowserAnimationsModule,  
  MatToolbarModule,  
  HttpClientModule  
],
```

Para darmos a nossa classe do `component` a instância do `HttpClient` basta instanciarmos o `service` no `constructor` dela, isso é possível por conta do `@Injectable` e como não usaremos esse `service` no nosso `HTML` iremos declará-lo como `private`, pois sempre criaremos métodos no nosso `component` para podermos fazer esse acesso.

```
constructor(private coursesService: CoursesService) { }
```

Como fazemos a declaração dessa variável no nosso `constructor` ela se torna uma variável da instância, uma variável daquele `component` e com isso conseguimos acessar o `coursesService`.

E para o `courses` receber a lista do `service` daremos ao `courses` o valor da chamada do `list()` que criamos no `coursesService`.

```
this.courses = this.coursesService.list();
```

Podemos fazer isso tanto no `constructor`, quanto no `ngOnInit` que com isso será feito o `list` apenas quando o `component` for inicializado no `HTML`

Angular conexão:

O que o `Angular` nos permite fazer é criar um `proxy` que vai funcionar como se estivéssemos fazendo chamadas para o mesmo domínio com domínios diferentes, então no caso vai ser do `front-end` para o `back-end` e não vai importar que estão em domínios diferentes, para desenvolvimento local é uma melhor opção do que o `CORS`.

Proxy:

Criamos então o `proxy` na pasta raiz do projeto como arquivo `"proxy.conf.js"` assim gerando um arquivo `JS` e nele colocamos nossa configuração.

```
const PROXY_CONFIG = [  
  {
```



```

context: ['/api'],
target: 'http://localhost:8080/',
secure: false,
logLevel: 'debug'
}
];
module.exports = PROXY_CONFIG;

```

O `context` sendo `/api` ele vai ajudar nós a diferenciarmos quando é uma chamada do roteamento do **Angular** para nossa **API** do **Spring**, então esse `context: ['/api']` ajuda nisso, então sempre que fizermos `/api` no **Angular** o **Angular** irá automaticamente irá redirecionar para o `target`.

A segurança está falsa, pois não estamos usando SSL local, na produção a gente habilita isso.

Como é uma constante vamos exportar essa config com `module.exports = PROXY_CONFIG;`.

Para dizer para o **Angular** que queremos iniciar isso sempre que nosso projeto também iniciar colocamos no `package.json` no `"start"` o nosso `proxy`:

```

"scripts": {
  "ng": "ng",
  "start": "ng serve --proxy-config proxy.conf.js",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test"
},

```

Com isso o **Angular** entende que sempre que for usar o `ng serve` usar este `proxy`, para funcionar a partir de agora não iniciaremos nosso projeto com `ng serve` e sim com `npm run start` para sempre utilizarmos esse `proxy`. Não podemos esquecer de alterar também no nosso `service` o link para apenas:

```

private readonly API = 'api/courses';

```

Chamada HTTP:

Usamos chamadas `http` para obtermos os dados de forma dinâmica, para mostrar isso agora não temos a API então criamos um `json` na pasta `assets` que contem os dados da "lista" que é mais ou menos o formato que esperamos do nosso servidor, então no `service` no lugar de passar os dados no `return` da lista no `list()` iremos usar o `httpClient` e chamaremos os métodos que queremos usar do **HTTP** nesse caso que queremos somente pegar as informações de uma lista usaremos o `get`

```
this.httpClient.get
```

, e no **get** precisamos informar a URI de onde pegaremos esses dados, uma recomendação é que na classe de **service**, principalmente aonde iremos fazer um CRUD é bom usarmos a mesma **url** e fazer com que os métodos **HTTP** que informem qual tipo de ação queremos fazer (obter lista, atualizar, deletar, etc.), para isso criamos uma **variável readonly** para evitar termos modificações nesse **valor** e daremos um **valor** que é o caminho da nossa API (endpoint).

```
private readonly API = '/assets/courses.json';
```

Após isso no nosso **get** poderemos informar de onde ele irá fazer o **get** que no caso é da variável **API** que criamos agora que tem o **endpoint** da **API**, nesse caso é retornado um **observable** e por isso será gerado um erro por conta de nossa **list()** não estar esperando um **observable**, para resolver é só tirarmos a tipagem que fizemos nela, só que para deixarmos o **observable** retornando um **array de courses** usaremos o operador de **diamante <>** no **get** para indicar qual tipo de dados está vindo do **get**.

```
return this.httpClient.get<Course[]>(this.API)
```

Porém no **courses component** também será gerado um erro por que o **courses** é um **array de Course**, então teremos de modificar isso também informando que o **courses** é um **observable** e usaremos novamente **<>** colocando o **Course[]** dentro para informar que o **observable** é um **array** de **course**, porém com isso não conseguiremos iniciar mais a variável e irá gerar um erro por conta do **strict**.

```
courses: Observable<Course[]>;
```

E para resolvermos o erro é só tirarmos o **this.courses = this.coursesService.list()** do **ngOnInit** e colocar no **constructor**.

```
constructor(private coursesService: CoursesService) {  
  this.courses = this.coursesService.list();  
}
```

Para verificarmos se a informação está chegando corretamente usaremos o **pipe()** que irá fazer com que antes de retornarmos a informação final possamos **manipular ela de maneira reativa utilizando programação reativa** onde podemos utilizar operadores do **rxjs** para que possamos fazer essa manipulação da maneira que for necessária, ou seja os dados irão passar por esse "**cano**" que é o **pipe** eles podem ser manipulados e no final retornaremos o resultado final, porém como queremos saber apenas se está chegando tudo ok a gente não vai mudar os dados, mas iremos fazer o uso do **operador tap** dentro do **pipe** que vem do pacote **rxjs/operators** e irá receber uma lista de **courses** e quando

receber essa lista irá fazer algo com a informação nesse caso apenas um `console.log`.

```
list() {  
  return this.httpClient.get<Course[]>(this.API)  
    .pipe(  
      first(),  
      tap(courses => console.log(courses))  
    );  
}
```

E já podemos rodar isso que irá funcionar, porém se você parar para perceber a gente não teve que modificar nada no nosso arquivo `html` e sabe por quê? Porque no `material table` o `datasource` pode ser de 2 tipos: `array de informações` ou um `observable`, o próprio `Angular Material` sabe tratar se essa informação está vindo de um `observable`, se tivéssemos que fazer isso de forma manual teríamos que fazer o `pipe` que faz o `subscribe` automaticamente para nós e se chama `async`.

```
<table mat-table [dataSource]="courses" <code>async</code> class="mat-elevation-z8">
```

Como nosso servidor não é um `stream` (não envia toda hora algo para nós) iremos obter a lista de courses e fechar a conexão, isso é uma boa prática já que fecha uma conexão que estava sendo inutilizada, para isso iremos usar o operador `first()` dentro do `pipe()` que é para quando estamos interessados em obter a 1ª resposta que o servidor nos enviar e irá finalizar a inscrição do `rxjs` nesse nosso endpoint.

TRATAMENTO DE ERRO:

No Angular quando acontece algum erro temos o objeto `HttpErrorResponse` que retorna o erro juntamente com o código, mensagens, todos os detalhes de erro para o Angular.

E para isso iremos tratar o erro no `component`, pois é o `component` que vai ter que tratar esse erro para poder mostrar algo para o usuário, e para isso usaremos o `pipe(catchError)` no nosso `Observable` que irá fazer o tratamento de erros, e no `rxjs` também temos um operador para isso que no caso se chama `catchError` que nele iremos receber o erro e iremos fazer algo com este erro e como precisamos retornar algo iremos abrir e fechar `{}` e o `catchError` ele espera um `observable` também de alguma informação e podemos criar `observable` de qualquer `informação estática` (está na memória) com outro operador chamado `of`.

```

constructor(private coursesService: CoursesService) {
  this.courses$ = this.coursesService.list()
    .pipe(
      catchError(error => {
        return of([])
      })
    );
}

```

Com isso retornamos um **array observable** vazio que será entregue para o front e acabaria com o carregamento infinito do **spinner**, porém temos de informar ao usuário que aconteceu algum erro e para isso usaremos um **pop-up** do **Angular Material** (tem outras maneiras de informar o erro, mas nesse caso usaremos isso). Criaremos um módulo **shared** para isso, pois usaremos essa **pop-up** de erro em outros locais e dentro dele criaremos os **components** de erro, e como usaremos esse **component** em outros modules iremos fazer o **exports** desse **component**. Após feito os preparos normais (**imports**, **exports**, etc) traremos um **constructor** de algum **dialog** para dentro do nosso **component** (EXEMPLO ABAIXO).

```

constructor(@Inject(MAT_DIALOG_DATA) public data: string) {}

```

Isso faz com que automaticamente **injetamos o tipo** do **MAT-DIALOG-DATA** e chamamos essa variável de data com tipagem de **string** e após isso iremos trazer do site do **Angular Material** os **dialog** e os métodos de **onError**.

```

onError(errorMessage : string) {
  this.dialog.open(ErrorDialogComponent, {
    data: errorMessage
  });
}

```

E também precisaremos **injetar o MatDialog**, iremos fazer isso no constructor.

```

constructor(
  private coursesService: CoursesService,
  public dialog: MatDialog
) {
  this.courses$ = this.coursesService.list()
    .pipe(
      catchError(error => {
        this.onError('Erro ao carregar cursos');
        return of([])
      })
    );
}

```

Na imagem acima temos o **pipe** fazendo com que **ao encontrar um erro** será carregado o método **onError** passado a mensagem **ERRO AO CARREGAR**

CURSOS, que será mostrado no pop-up e retornado um array vazio.

Ícone para cada item através do pipe:

Como temos essa classe especial no Angular que transforma valores, ou seja dado um valor iremos ter uma lógica que irá retornar esse valor transformado.

Vamos então criar um pipe (no caso iremos colocar ele no módulo compartilhado (shared), pois outros módulos podem querer utilizar deste pipe também)

```
ng g pipe shared/pipes/category
```

Pelo método transform do pipe que podemos passar o valor e argumentos adicionais para retornar algo de algum tipo, geralmente os pipes são os tipos mais fáceis que temos na hora de escrever testes unitários por causa que são bem diretos (recebo um valor, faço a lógica e retorno outro valor).

No nosso caso iremos receber um valor string e retornar um valor string (ícone que exportamos do Icon do Angular Material (MatIconModule) e nossa lógica é que se o valor for 'front-end' o pipe irá retornar a string 'code' (ícone de html).

```
transform(value: string): string {
  switch(value) {
    case 'front-end':
      return 'code';
    case 'back-end':
      return 'computer'
  }
  return 'code';
}
```

Para fazer esse módulo ficar visível a outros components além do shared iremos coloca-lo no exports do shared-module.ts, pois iremos querer utiliza-lo também no nosso módulo de cursos.

```
exports: [
  MatDialogComponent,
  CategoryPipe
]
```

Para usar este pipe basta somente pegarmos o nome do pipe e colocar dentro do icon copiado no Angular Material.

```
<ng-container matColumnDef="category">
  <th mat-header-cell *matHeaderCellDef>Categoria</th>
  <td mat-cell *matCellDef="let course">{{ course.category }}
    <mat-icon aria-hidden="false" aria-label="Example home icon" fontIcon="home">{{ course.category | category }}</mat-icon>
  </td>
</ng-container>
```

relativeTo:

Faz com que ao ser acionado leve para a rota atual + /desejada por causa do `relativeTo` que faz com que seja relativo a rota atual (ativa) por ter um valor de `this.route`, gerando assim uma melhor organização do nosso código, pois se mudarmos a rota não afetará o código inteiro e ficará mais fácil de se arrumar.

FormBuilder:

Tem toda a lógica que vai nos ajudar a criar o **FormGroup**, para criar um Formulário com o **FormBuilder** já fazemos a **tipagem** do mesmo, isso para não haver inconsistência nos dados ou erros admitidos e temos todos os erros e verificações no tempo de compilação ao invés do dado com erro ir para o Back-end, então damos a eles o tipo com o `group()` de acordo com o que queremos dos atributos.

```
form = this.formBuilder.group({
  name: [''],
  category: ['']
});
```

Isso faz com que a nossa aplicação não aceite do usuário um name, category a não ser uma **string**, e também podemos acessar os métodos/funções do **JavaScript** da classe em questão (string nesse caso).

E para não permitirmos também um valor **null** (nulo) usamos uma 2ª configuração que é o `nullable`

```
email: new FormControl('', {nullable: true}),
```

Ou então para não precisar repetir isso em cada dado a gente usa no **constructor** ao invés de **FormBuilder** a classe **NullableFormBuilder** que é o **FormBuilder** só que não aceitamos valores nulos, isso acaba que todos os dados (campos) irão ter essa característica, então vai de necessidade a questão de qual usar.

Porém ao fazermos todos esses processos temos que saber que o método `save()` irá ocasionar um erro, pois nosso **formulário** tem um valor **parcial (partial)** de **Course** e não um valor de **Course**.

```
(property) AbstractControl<Partial<{ name: string; category: string; }>, { name: string; category: string; }>.value: Partial<{ name: string; category: string; }>
```

The current value of the control.

- For a `FormControl`, the current value.
- For an enabled `FormGroup`, the values of enabled controls as an object with a key-value pair for each member of the group.
- For a disabled `FormGroup`, the values of all controls as an object with a key-value pair for each member of the group.
- For a `FormArray`, the values of enabled controls as an array.

```
(this.form.value)
```

No retorno dele então temos de arrumar para **Partial<Course>**.

```
save(record: Partial<Course>) {
```

Quando fazemos isso significa que no retorno iremos aceitar um objeto desde que ele tenha um atributo do `Course`, para restringirmos isso temos de configurar no Back-end.

FormGroup:

Sempre que temos um grupo de campos a gente chama de `FormGroup` (instância).

@Input():

`Decorator` (anotação) que é tudo que vai ser passado para algum componente, colocamos ele atrás da declaração de algum component.

Isso se dá quando temos um relacionamento entre componente filho e componente pai ou mãe.

@Output():

Tudo que sai, ou seja os eventos que estão saindo, usaremos no projeto juntamente com o `input` para deixar mais fácil criar os testes já que temos componentes inteligentes podemos passar para o principal o `Router` que estava no `courses-list` para não usarmos nada vindo de fora do `courses-list`, fazendo o component inteligente (`courses-list`) virar um component de apresentação. Para usar o método de `click` na tabela dos courses a gente gera um `Output` e dá o valor a ele de um emissor de eventos (`EventEmitter`) que dispara alguma ação quando acionado, daremos então no HTML do courses uma chamada no `add` e daremos o valor do `onAdd()` (que está no `courses.component` principal) para ele, que será acionado e disparado o método.

```
<app-courses-list class="column-flex" *ngIf="courses$ | async as courses; else loading" [courses]="courses" (add)="onAdd()"></app-courses-list> <!-- Está verificando se há algo no courses$ se tiver carrega os dados de courses$ para o courses e fazemos um data-bind do courses de courses-list dando o valor de courses do courses normal (daqui). E com esse selector (app-courses-list) que carregamos o component de courses-list aqui nesse outro component (courses). (add) está pegando um emissor de eventos do courses-list e dando o valor de onAdd() para ser feito este evento que está no courses component principal (inteligente). -->
```

Guarda de rotas (Resolver):

Quando geramos um `resolver` temos que ter em mente que iremos tomar cuidado como iremos utiliza-lo, pois até quando nosso resolver conseguir carregar a informação não iremos fazer o redirecionamento, então se for algo que irá demorar muito tempo para carregar essa acaba não sendo a melhor maneira de fazer isso, no caso a melhor maneira seria tratar diretamente no formulário no caso.

Porém como o `course` só tem 3 atributos (`id`, `name`, `category`) é possível fazer desta maneira.

Então vamos gerar um resolver que seria um guarda de rotas que irá pegar o `id` que está no `end-point` para identificar o `course` em questão e carrega-lo para

modifica-lo, nós conseguiríamos pegar essa informação de algum lugar (front-end ou back-end).

```
ng g resolver courses/guards/course
```

O **resolve** no caso vai ficar escutando qual é a nossa rota, ele tem o **Snapshot** (uma cópia da rota) em um determinado período do tempo, com isso conseguimos obter informações da rota como por exemplo qual o parâmetro que no caso é o **id**, e conseguimos fazer algo com isso, no caso vamos buscar a informação que estamos querendo seja ela **cacheada** no nosso front-end ou então no back-end.

```
constructor(private service: CoursesService) { }

resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Course> {
  if (route.params && route.params['id']) { // Confere se temos parâmetros na url em questão e se esse parâmetro é um id como está especificado no
    courses-routing.module e se tiver ele vai ir no service utilizar o método loadById() passando o id da route como parâmetro e retornando a resposta.
    return this.service.loadById(route.params['id'])
  }
  return of({id: '', name: '', category: ''});
}
```

Após terminarmos de fazer o resolve precisamos voltar no **routing-module** e colocar a informação lá na parte de caminhos da nossa rota em questão, tem uma propriedade chamada **resolve** e damos a ela o nome que quisermos e quem vai resolve-lo será o **CourseResolver** e colocamos isso no **edit** e no **new** para não termos problema quando formos fazer o formulário, e foi por isso que no resolve colocamos o retorno de um objeto vazio para se no caso não tiver o **id** como parâmetro e for um novo.

```
const routes: Routes = [
  { path: '', component: CoursesComponent },
  { path: 'new', component: CourseFormComponent, resolve: {course: CourseResolver} },
  { path: 'edit/:id', component: CourseFormComponent, resolve: {course: CourseResolver} } // Para darmos nome para parâmetros no Angular a gente coloca (:)
  e o nome que quisermos chamar depois, que no caso é o 'id', e aí depois conseguimos pegar esse parâmetro na rota e fazer alguma coisa com essa informação como por
  exemplo ir na nossa API e buscar o curso que está sendo editado.
];
```

SetValue & PathValue:

Quando estamos **setando** todos os valores do nosso formulário fazemos um **SET**, **Path** quando estamos **setando** apenas alguns valores do formulário.

FormArray:

Quando a gente trabalha com um formulário no **Angular** a gente utiliza o **FormBuilder** para poder ajudar a gente a construir o formulário e vale lembrar que no **Angular** cada campo que nós temos é um campo de **Form Control**, então cada campo no formulário no **Angular** é **Form Control**, quando temos vários **Form's Control** temos um grupo de **Form Control** e quando temos o mesmo tipo de **Form Control** várias vezes a gente tem o que chamamos de **Form Array**. A forma mais fácil que temos de fazer isso no **Angular** é criando métodos para separar cada pedaço do que a gente precisa construir, nós também podemos fazer isso com Componentes diferentes caso seja um formulário bem complexo, mas no nosso caso iremos fazer tudo no mesmo componente.

Outro ponto importante é que a partir da versão 15 do **Angular** a gente tem os **Typed Forms**, no nosso caso estamos utilizando o **NonNullableFormBuilder** que significa que **nenhum campo pode ser vazio**, todos precisam ter dados, porém o **Form Array** no **Angular** como pode ser um grupo de **Form Control's** o **Angular** não trabalha com formulários tipados em **Form Array's**.

```
form = this.formBuilder.group(controls: {  
  _id: [''],  
  name: ['', [Validators.required,  
    Validators.minLength(minLength: 5),  
    Validators.maxLength(maxLength: 100)]],  
  category: ['', [Validators.required]]  
});
```

No `_id`, `name` e `category` a gente consegue ter o **intellisense** da **IDE** por conta de ser um formulário tipado do **Angular**, porém no **Form Array** não temos mais isso e é por esse motivo que iremos trabalhar com **Untyped Form Control**.

Classe utilitária para Validação de Formulários:

Podemos criar um **serviço** em **shared** que é o diretório que criamos para compartilhamento que servirá de **classe utilitária** para todos os nossos formulários, isso faz com que não precisemos repetir códigos de validação para cada formulário somente se quisermos fazer algo específico para aquele formulário.