UNIÃO EDUCACIONAL DE CASCAVEL - UNIVEL

**FACULDADE DE CIÊNCIAS SOCIAIS APLICADAS DE CASCAVEL**

CURSO TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

NATALIA ALVES DE SOUZA

RELATÓRIO DE HORAS COMPLEMENTARES DA DISCIPLINA DE SISTEMAS DISTRIBUÍDOS

CURSO DE ANGULAR 2

Cascavel

2017

**1 INTRODUÇÃO**

A SER IMPLEMENTADA.

**2 DEFINIÇÃO**

A SER IMPLEMENTADA

**3 VÍDEO #01: INTRODUÇÃO AO ANGULAR 2**

## 3.1 O que será estudado no curso?

## Ao longo do curso, estudaremos os seguintes assuntos: componentes e *Templates*; Data *binding*; Diretivas; Serviços; Formulários; Roteamento; Interação com servidor e CRUD Mestre Detalhe, utilizando todos os conceitos vistos no curso.

## 3.2 O que é necessário saber para realizar o curso?

## É necessário ter conhecimento das linguagens *HTML*, *CSS* e *JavaScript*. Não é necessário ter conhecimento de Angular JS 1.x.

## 3.3 O que é o Angular 2?

## Angular 2[[1]](#footnote-2) é um *framework* nascido da parceira da Google com a Microsoft; escrito em *TypeScript* e possui código *Open Source* disponível no *GitHub*2[[2]](#footnote-3). O Angular 2 não é continuação do Angular 1, pois foi reescrito para fazer melhor uso *HTML*.

## Esse *framework* é totalmente orientado a Componente, ou seja, tudo aplicação é um componente. Nós criaremos um componente raiz (também chamado de *Root*), que será o Pai ou a Mãe da nossa aplicação. Esse componente pode ser uma lista de contatos, de clientes, um cabeçalho; fica a cargo de o desenvolvedor declarar o que será o componente raiz. Um componente também pode ter outros componentes, assim podemos dividir a aplicação em partes menores, o que facilita os testes unitários a serem realizados pelo desenvolvedor.

## O *framework* é dividido nos Blocos Principais, que são: Componentes; Diretivas; Roteamento; Serviços; *Template*; *Metadata*; Data *Binding* e Injeção de dependência.

## 3.4: Visão geral sobre os blocos:

a) Componentes: O objetivo do componente é mostrar dados, então ele pode realizar integração com o *BackEnd*. Ele é responsável por todo o comportamento da *VIEW* (junção dos Componentes, *Controller* e Escopo da aplicação). Nessa junção, ele é também responsável pelo o que o usuário vai ver, pois encapsula o Template, o *Metadata* e o Data *Binding*.

b) *Template*: Layout da tela como botões e formulários.

c) *Metadata*: Processamento dos metadados; esses metadados permitem que o *framework* ler as classes e fazer seu processamentos.

d) Data *Binding*: Associação dos componentes do nosso projeto + os componentes do *template.*

e) Serviço: Como boa prática, regras de negócios não são escritas no Componente; para isso nós utilizamos um serviço que se comunicará com o *backend*, além de poder ser injetado em outras classes; esse processo recebe o nome de Injeção de Dependência.

f) Roteamento: Responsável pela navegação da aplicação, não só em relação a páginas, mas também a telas (como ir de uma tela a outra).

g) Diretiva: Responsável por modificar elementos DOM e/ou seu comportamento.

**4 VÍDEO #02: AMBIENTE DE DESENVOLVIMENTO (NODE.JS, TYPESCRIPT, ANGULAR CLI)**

Para iniciarmos o desenvolvimento da nossa aplicação, é necessário instalarmos a última versão o *Node.JS*, que está disponível no seguinte link: https://nodejs.org/en/. Para a instalação, não é necessária nenhuma configuração específica; basta ir clicando *Next*>*Next*>*Install*, até a instalação ser concluída. O Node.JS já vem com um repositório exclusivo com as bibliotecas do angular/cli. Chamado de NPM, esse repositório possui todas as bibliotecas necessárias para o desenvolvimento da aplicação (todas as dependências do nosso projeto, serão baixadas desse repositório) e pode ser acessado pelo link: https://www.npmjs.com/package/@angular/cli.

Depois de instalar o Node.JS, é necessário instalar o *TypeScript* através do Prompt de Comando. Siga os seguintes passos para realizar a instalação:

- Abra o Prompt de Comando (ou CMD) e digite o seguite comando:

- npm install -g typescript (se seu SO for Windows) ou

- sudo npm install -g typescript (se seu SO for Linux ou MAC)

Através dessa instalação, o *TypeScript* será baixado do diretório NPM.

Ainda com o Prompt de Comando aberto, vamos instalar o angular/cli. Execute o seguinte comando:

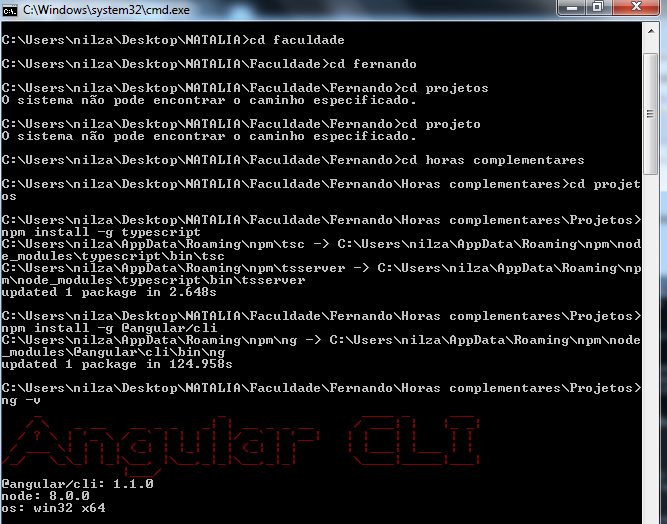
- npm install -g @angular/cli (se seu SO for Windows) ou

- sudo npm install -g @angular/cli (se seu SO for Linux ou MAC)

Para verificarmos se nosso ambiente está tudo OK, depois que o angular/cli for instalado, execute o seguinte comando:

- ng –v

Se o ambiente estiver OK, será apresentada a tela abaixo com a versão do angular/cli e do Node.JS:



Para o desenvolvimento, precisamos também de um editor de texto. Abaixo serão listados quatro editores, que possuem um suporte melhor para o Angular 2. Você pode utilizar o de sua preferência. Todos eles estão disponíveis para Windows, Linux e Mac.

Como o *TypeScript* é mantido pela Microsoft, assim como o Visual *Studio Code*, há uma ótima interação entre eles; para o ATOM, *WebStorm* e *SublimeText*, há *plugins* disponíveis para o uso do TypeScript (apesar do *WebStorm* também suportar o TypeScript, há *plugins* disponíveis para ele):

- Visual *Studio Code*:Ferramenta gratuita que suporta o *TypeScript*. Pode ser baixado do seguinte link: https://code.visualstudio.com/download. Não é necessária nenhuma configuração especifica para sua instalação.

- ATOM: Ferramenta gratuita. Para utilizar esse editor, é necessário baixar *plugin* do *TypeScript*. Esse *plugin* está disponível para *dowloand* no link: https://atom.io/packages/atom-typescript.

- *WebStorm*: Ferramenta paga. O editor também suporta o *TypeScript*, porém há plugin disponível para ele e que pode ser baixado do seguinte link:https://www.jetbrains.com/webstorm/download/#section=windows. Não é necessária nenhuma configuração especifica para sua instalação.

- Sublime *Text*: Ferramenta gratuita, porém caso desejar, você pode adquiri a versão paga. Para utilizar o *TypeScript* nesse editor, é necessário baixar *plugin*, que está disponível no link: https://github.com/Microsoft/TypeScript-Sublime-Plugin.

Para o desenvolvimento desse trabalho, será utilizado o editor Visual *Studio Code*.

**5 VÍDEO #03: HELLO, WORLD! CRIANDO PRIMEIRO PROJETO E O PRIMEIRO COMPONENTE**

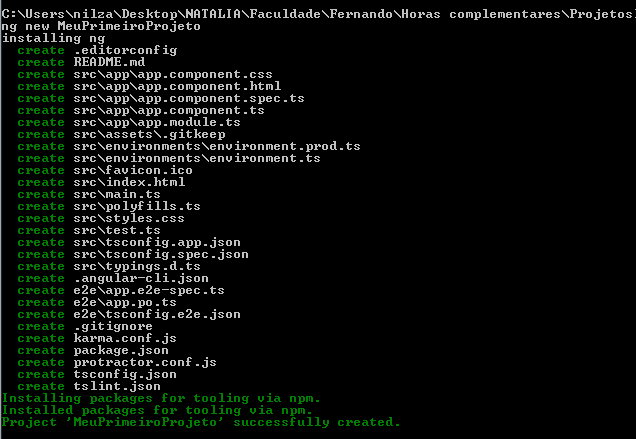
**OBS:** Projeto disponível em: github.com/nasouza2/Angular2B/tree/master/MeuPrimeiroProjeto.

**5.1: Hello World! Criando meu primeiro projeto**

Abra o Prompt de Comando e navegue até o diretório onde seu projeto será criado. Em seguida, digite o seguinte comando:

- ng new NomeDoProjeto

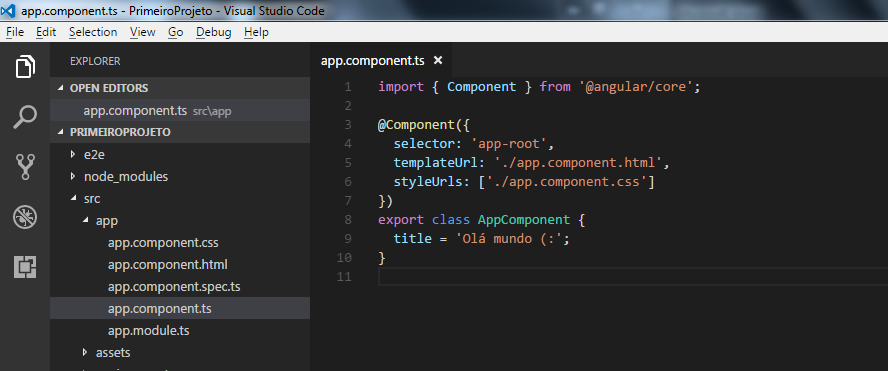
No final, o angular/cli vai criar toda a estrutura padrão do projeto e também vai instalar as dependências do NPM:



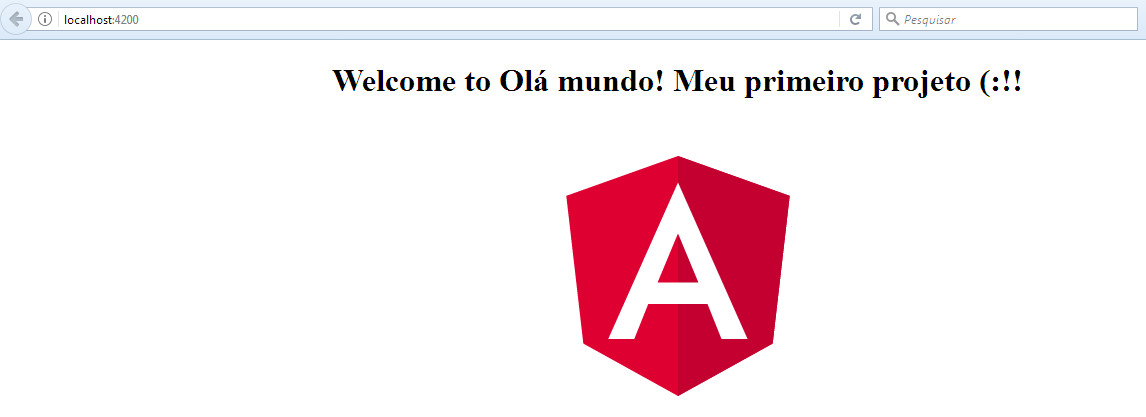
- Ainda com o Prompt de Comando aberto, acesse seu projeto e digite o comando: ng serve. Esse comando indica que nosso projeto vai ser servido ao *browser*.

Abra o projeto com o editor que texto. Dentro do diretório scr – app é onde os componentes serão criados.

Abra o arquivo app.component.js, altere o título do seu projeto e salve. O próprio angular/cli faz o biuld novamente e atualiza a URL:



Com o navegador de sua preferência, acesse http://localhost:4200. Note que seu projeto está funcionando:

****

**5.2 Criando componente manualmente**

Com seu projeto aberto, clique com botão direito no pacote app e clique em *New Folder*. Informe o nome do seu diretório, que por convenção, caso exista mais de uma letra, ele deve ser escrito com hífen (exemplo: meu-primeiro). Dentro desse diretório, clique com o botão direito, selecione a opção New File e informe o nome do seu componente com a extensão .ts (exemplo: meu-primeiro.component.ts).

No projeto, crie uma classe com um nome qualquer; toda classe deve ter a primeira letra de cada palavra em maiúscula. Depois de a classe ser criada, precisamos informar ao Angular que essa classe é um componente; para isso, usamos a anotação @Component. Nesse ponto, precisamos informar ao Angular,onde está o pacote com essa anotação, para que o import seja realizado. Isso deve ser feito através da linha: import {Component} from '@angular/core'.

Depois de o import ser realizado, deve ser informado os metadados do componente; isso é feito através de um selector dentro da anotação @Component. Depois é necessário criar um template. A imagem abaixo mostra como ficou a classe:

@Component({

selector:'meu-primeiro-component',

template:`

<p> Meu primeiro componente com Angular 2.</p>

`

})

classMeuPrimeiroComponent{}

Após o componente ser criado, é necessário utilizá-lo em alguma classe. Copie a tag ‘meu-primeiro-component’ e a cole no arquivo app.component.html.

OBS: Para deixar o código mais limpo, foram removidos os links que haviam nesse arquivo, como pode ser visto na figura x:

<divstyle="text-align:center">

<h1>

Bem vindo ao {{title}}!!

</h1>

</div>

<meu-primeiro-component></meu-primeiro-component>

Da forma que criamos a classe MeuPrimeiroComponent, ela é vista apenas internamente. Para que ela possa ser vista por outras classes e assim evitar erro de compilação, é necessário informar “*export*” antes da classe:

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

@Component({

selector:'meu-primeiro-component',

template:`

<p> Meu primeiro componente com Angular 2.</p>

`

})

exportclassMeuPrimeiroComponent{}

Só isso não basta que nosso projeto rode corretamente; todo componente, serviço, diretiva criada, deve ser informada em um módulo. Abra o arquivo app.modules.ts e declare seu componente, dentro das declarações do modulo e o importe (no import, não é necessário informar a extensão .js):

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

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

import { AppComponent } from'./app.component';

import {MeuPrimeiroComponent} from'./meu-primeiro/meu-primeiro.component';

@NgModule({

declarations: [

AppComponent,

MeuPrimeiroComponent

],

imports: [

BrowserModule

],

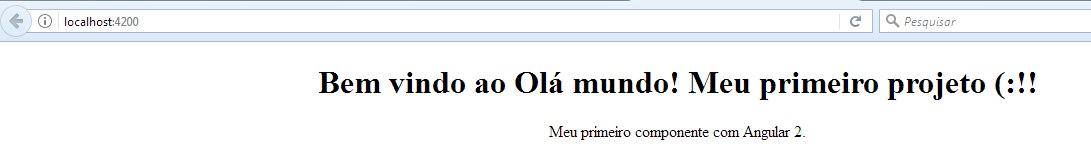
providers: [],

bootstrap: [AppComponent]

})

exportclassAppModule { }

Salve sua aplicação e abra novamente o *browser*; note que seu projeto foi atualizado corretamente:



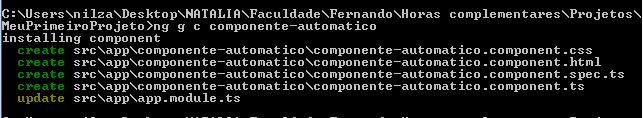
**Dificuldade encontrada:**Atente-se para usar crase e não aspas simples no Template. Se usar aspas simples, ocorrerá erro de compilação.“Atente-se também para extensão do seu componente, que deve ser “.ts” de *TypeScript*. Devido a um erro de digitação, a aplicação não conseguia encontrar o componente; todo o projeto teve que ser revisto, desde a instalação até a declaração do componente no módulo, para então perceber a grafia errada (“js” ao invés de “ts”).

**5.3 Criando componente automaticamente**

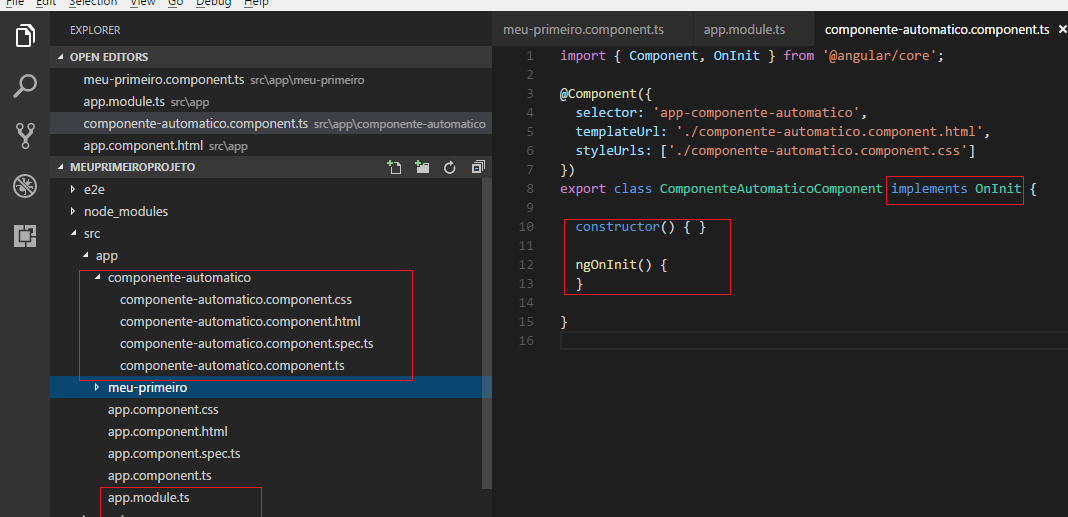
O componente foi criado componente, apenas para entendermos o passo a passo de sua criação e o padrão de nomenclatura. Agora vamos ver com criar os componentes de forma automática.

No Prompt de Comando, execute o seguinte comando: ng g c nome-do-component (g = gerar; c = componente. Você também pode escreve “*component*”).

OBS: Se o ng serve estiver sendo executado, pare sua execução (através de Ctrl + C).



Depois de ser executado, o angular/cli cria os arquivos automaticamente:



No seu component.ts, remova os códigos destacados em vermelho, pois não utilizaremos ele nesse momento. Acesse o arquivo app.module.ts; note que foi criado o import do component:

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

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

import { AppComponent } from'./app.component';

import {MeuPrimeiroComponent} from'./meu-primeiro/meu-primeiro.component';

import { ComponenteAutomaticoComponent } from'./componente-automatico/componente-automatico.component';

@NgModule({

declarations: [

AppComponent,

MeuPrimeiroComponent,

ComponenteAutomaticoComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

exportclassAppModule { }

Abra o arquivo app.component.html e informe a tag do seu component:

<divstyle="text-align:center">

<h1>

Bem vindo {{title}}!!

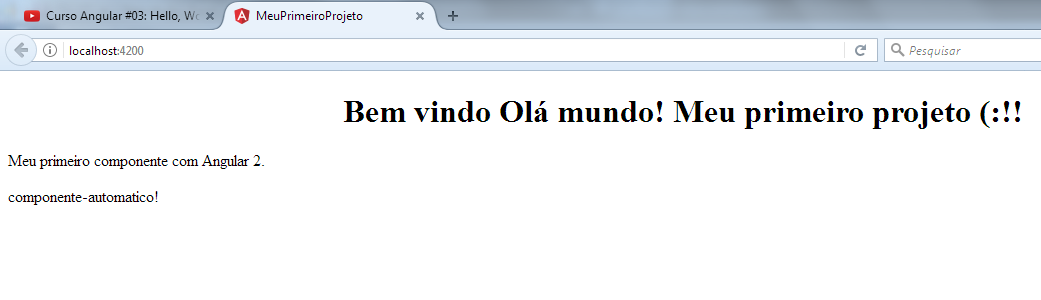
</h1>

</div>

<meu-primeiro-component></meu-primeiro-component>

<app-componente-automatico></app-componente-automatico>

No Prompt de Comando, execute novamente o ng serve e atualize sua URL. Note que o projeto foi executado corretamente:



**Link útil:** Instalar e configurar o git no Windows: http://gabsferreira.com/instalando-o-git-e-configurando-github.

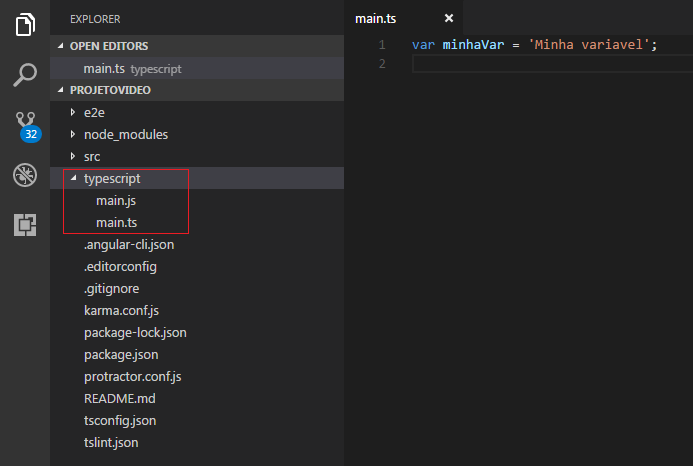
**6 VÍDEO #04: INTRODUÇÃO AO TYPE SCRIPT PARA ANGULAR**

**OBS:** Projeto disponível em: https://github.com/nasouza2/Angular2B/tree/master/ProjetoVideo04.

Abra o projeto criado anteriormente e crie um novo diretório com qualquer nome e crie um arquivo com extensão .ts (exemplo main.ts). Nesse arquivo, declare uma variável qualquer, como exemplo abaixo:

varminhaVar = 'Minha variavel';

Abra o Prompt de Comando e execute o comando: tsc main.ts. Através desse comando, estamos pedindo para o TypeScript compilar nosso arquivo; depois dessa compilação, será criado um arquivo chamado main.js, conforme imagem abaixo:



Esse arquivo main.js é exatamente a mesma coisa que o arquivo main.ts. A diferença é que no arquivo main.ts, utilizamos *JavaScript* puro.

Na compilação, o *TypeScript* remove os espaços em branco; nosso arquivo main.js ficou a mesma coisa do main.ts:

varminhaVar = 'Minha variavel';

functionminhaFuncao(x, y) {

returnx + y;

}

No arquivo main.ts, crie uma função qualquer, conforme exemplo abaixo e compile novamente o projeto:

varminhaVar = 'Minha variavel';

functionminhaFuncao(x, y){

returnx + y;

}

Uma das alterações que teve na *ECMAScript* 2015 (que é a nova versão do *JavaScript*), é utilizar “*let*” ao invés de “*var*”:

varminhaVar = 'Minha variavel';

functionminhaFuncao(x, y){

returnx + y;

}

letnum = 2;

Nem todos os *browser*s estão adaptados para utilizar o *ECMAScprit* 2015 (também chamado de ES6), dessa forma nós precisamos de um *transpiler*, que vai compilar esse código em *JavaScript* puro. Depois que for compilado o código acima, no arquivo main.js ficará da seguinte forma:

varminhaVar = 'Minha variavel';

functionminhaFuncao(x, y) {

returnx + y;

}

varnum = 2;

Mais um exemplo com ES6:

varminhaVar = 'Minha variavel';

functionminhaFuncao(x, y){

returnx + y;

}

//ES6

letnum = 2;

constPI = 3.14;

O arquivo main.js ficou da seguinte forma:

varminhaVar = 'Minha variavel';

functionminhaFuncao(x, y) {

returnx + y;

}

varnum = 2;

varPI = 3.14;

**Dificuldades encontradas:** Ao executar o comando tsc main.ts, ocorria o seguinte erro: "TS6053: arquivo 'main.ts' não encontrado". Para resolvê-lo, foi utilizado o comando “npm install tsd tsd reinstall” na pasta do projeto; depois foi acessada a pasta typescript e então o comando tsc main.ts foi novamente executado; dessa vez não apresentando mais erro de compilação (ajuda retirada do link: https://translate.googleusercontent.com/translate\_c?depth=1&hl=pt-BR&prev=search&rurl=translate.google.com.br&sl=en&sp=nmt4&u=https://github.com/Microsoft/ngconf2015demo/issues/31&usg=ALkJrhjUGfonih-\_ZyKmpv7hwDpki5gMIA).

**Links úteis:**

-Todas as alterações do ECMA2015, podem ser encontradas em: es6-features.org/#Constants.

*- Transpiler* Babel: babeljs.io/repl/#?babili=false&evaluate=true&lineWrap=false&presets=es2015%2Creact%2Cstage-2&targets=&browsers=&builtIns=false&debug=false&code=

**7 VÍDEO #05: MÓDULOS (ngMODULE)**

Módulo é o arquivo que vai ajudar a organizar nossa aplicação, pois centraliza todos os arquivos num mesmo diretório. Abra o primeiro projeto criado e abra o arquivo app.modules.ts; abaixo segue a explicação:

// O BrowserModule prepara a aplicação para ser usada na web

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

// Indicando da onde o NgModule está

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

// Import's da classe de nosso projeto

import { AppComponent } from'./app.component';

import {MeuPrimeiroComponent} from'./meu-primeiro/meu-primeiro.component';

import { ComponenteAutomaticoComponent } from'./componente-automatico/componente-automatico.component';

// Declarando a classe

@NgModule({

// Metadados

// Dentro do declarations, nós listamos todos os componentes, diretivas e pipes que vamos utilizar no módulo

declarations: [

AppComponent,

MeuPrimeiroComponent,

ComponenteAutomaticoComponent

],

// Em imports, colocamos outros módulos que queremos utilizar nesse módulo

imports: [

BrowserModule

],

// Em providers, informados quais são os serviços que ficarão disponíveis para todos os componentes declarados no módulo,

// como por exemplo, autenticação de login, rotas.

providers: [],

// O bootstrap é encontrado apenas no módulo raiz e indica qual componente deve ser iniciado ao executar a aplicação (componente

// que vai servir de container da app ou seja, qual é o componente principal)

bootstrap: [AppComponent]

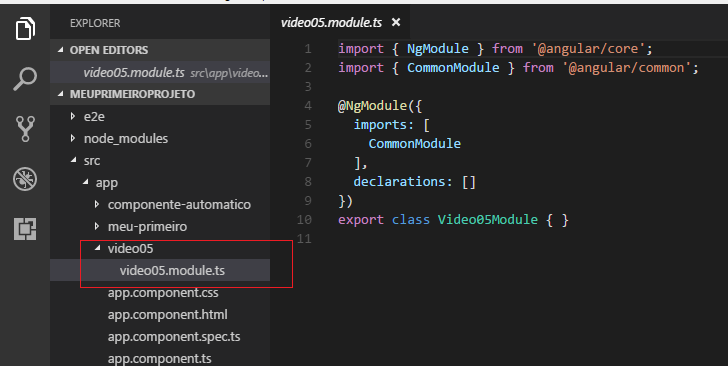
})

exportclassAppModule { } //Declaração da classe

**7.1 Criando módulo**

**OBS:** Projeto disponível em: github.com/nasouza2/Angular2B/tree/master/MeuPrimeiroProjeto/src/app/cursos

Abra o Prompt de Comando até seu projeto e digite o comando: ng g m nomedomodulo (você pode utilizar apenas o “m” ou informar “module”). Depois de executado o comando, note que o módulo foi criado na sua árvore de arquivos:



**8 VÍDEO #06: INTRODUÇÃO AOS TEMPLATES**

Os componentes que nós criamos, sempre têm um código HTML, porque dentro dessa classe é onde vamos colocar nossa lógica de programação. Dentro do código HTML,é onde vamos construir o template que o usuário vai ver.

Abra o arquivo cursos.componente.ts, crie uma variável do tipo String e instancie essa variável.

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

@Component({

selector:'app-cursos',

templateUrl:'./cursos.component.html',

styleUrls: ['./cursos.component.css']

})

exportclassCursosComponentimplementsOnInit {

// Podemos instanciar a variável por aqui, informando o que ela vai receber ou

nomePortal: string;

// podemos iniciar ela através do construtor, usando a palavra "this"

constructor() {

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

}

ngOnInit() {

}

}

Para utilizar à variável, abra o arquivo cursos.component.html e dentro de duas chaves, informe a variável:

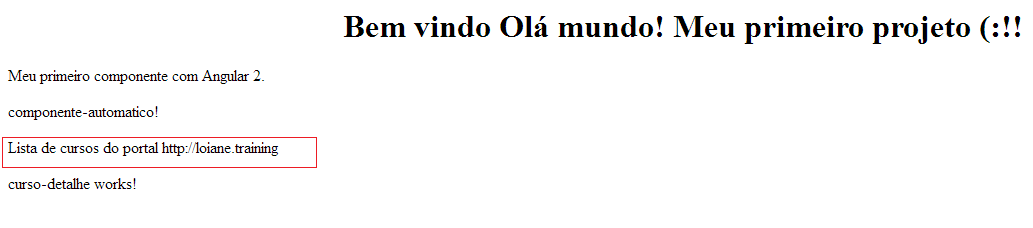
<p>

Lista de cursos do portal {{ nomePortal }} <!-- Isso se chama interpolação-->

</p>

<app-curso-detalhe></app-curso-detalhe>

No browser, o projeto ficou assim:



Para fazer listagem manualmente, utilizamos a tag <ul>:

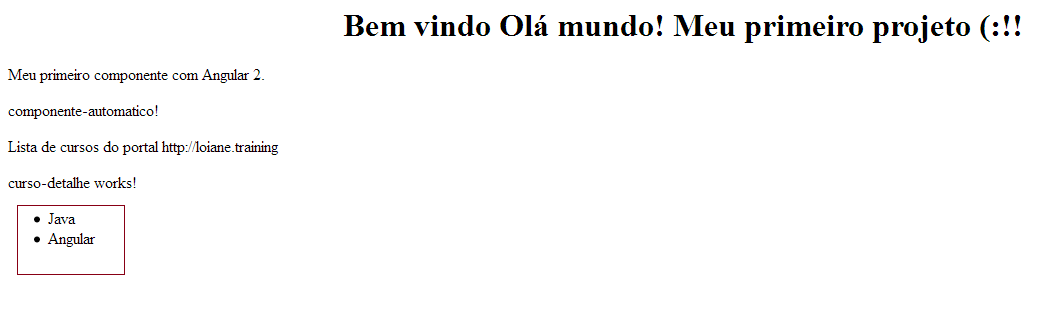
<!-- Para fazer listagem manualmente, geralmente utilizamos a tag <ul>-->

<ul>

<li> Java </li>

<li> Angular </li>

</ul>



Porém os *templates* do Angular nos permitem fazer as listagens dinamicamente. Em cursos.component.ts, crie uma variável do tipo *array*:

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

@Component({

selector:'app-cursos',

templateUrl:'./cursos.component.html',

styleUrls: ['./cursos.component.css']

})

exportclassCursosComponentimplementsOnInit {

// Podemos instanciar a variável por aqui, informando o que ela vai receber ou

nomePortal: string;

cursos: string[] = ['Java', 'Angular', 'Ext JS']

// podemos iniciar ela através do construtor, usando a palavra "this"

constructor() {

this.nomePortal = 'http://loiane.*training'*;

}

ngOnInit() {

}

Para que cada posição ocupe corretamente uma <li>, será utilizada a diretiva *NGFor*, onde é declarada uma variável local utilizando o let e depois o nosso *array*:

<!-- Lista criada pelo template-->

<ul>

<!-- Declaramos uma váriavel curso que vai receber a posição do array cursos -->

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

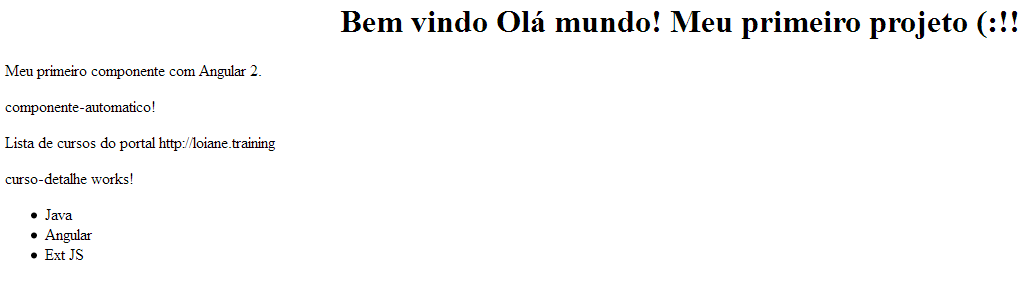
<!-- Depois é feita a interpolação para a saída da variável curso -->

{{ curso }}

</li>

</ul>

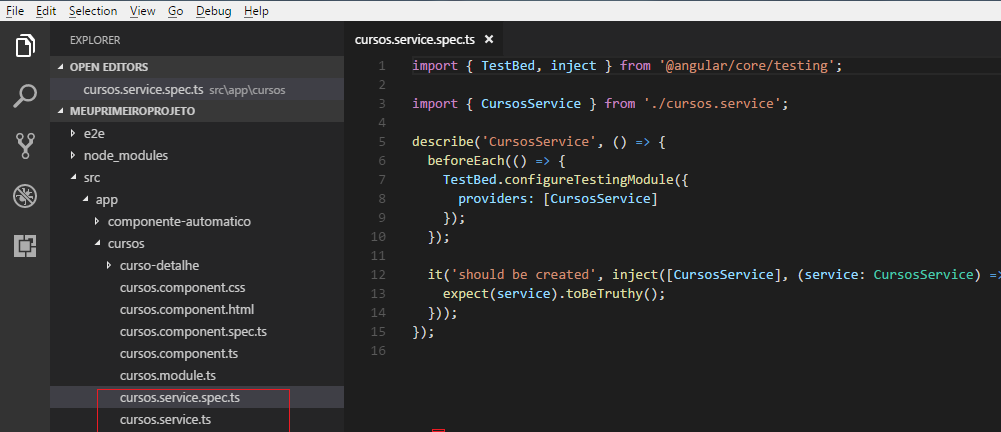
No *browser*, o projeto ficou assim:



**9 VÍDEO #07: INTRODUÇÃO AOS SERVIÇOS E INJEÇÃO DE DEPENDÊNCIA (DI)**

Como uma boa prática, colocamos na classe *Component*, apenas o código responsável por mostrar as informações para o usuário ou interagir com ele e numa classe *Service*, colocamos toda a lógica de comunicação.

Para criar um serviço, use o comando ng g s nomeservico (se preferir pode usar “*service*” ao invés de apenas “s”). Após executar o comando, o Angular criará dois arquivos no nosso projeto:



No arquivo de cursos.service.ts, criamos o método que vai retornar o nosso *array*:

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

@Injectable()

exportclassCursosService {

constructor() { }

getCursos(){

return ['Java ', 'Angular ', 'Ext JS '];

}

}

No arquivo cursos.component.ts, deixamos o componente apenas passando o resultado do nosso método. Precisamos importar, usar a classe cursos.*service* e criar variável que instancia essa classe:

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

// Importação da classe

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

@Component({

selector:'app-cursos',

templateUrl:'./cursos.component.html',

styleUrls: ['./cursos.component.css']

})

exportclassCursosComponentimplementsOnInit {

nomePortal: string;

cursos: string[];

constructor() {

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

// Criando a variável

varservico = newCursosService();

É nesse ponto que entra o conceito de Injeção de Dependência, que é fazer com que o Angular nos forneça uma instância da classe de serviço, assim não precisamos nos preocupar em ter que instanciá-la manualmente. O que indica que um serviço pode ser injetado éa anotação @Injectable() na classe de serviço:

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

@Injectable() // Anotação indicando que o serviço pode ser injetado

exportclassCursosService {

constructor() { }

getCursos(){

return ['Java ', 'Angular ', 'Ext JS '];

}

}

A injeção de dependência é dentro do construtor da classe de seu componente:

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

// Importação da classe

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

@Component({

selector:'app-cursos',

templateUrl:'./cursos.component.html',

styleUrls: ['./cursos.component.css']

})

exportclassCursosComponentimplementsOnInit {

nomePortal: string;

cursos: string[]; // Deixamos o componente passando apenas o resultado

// Declarando o serviço e adicionando o modificador de acesso (que pode ser público ou privado)

constructor(privatecursosService: CursosService) {

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

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

// Criando a variável

//var servico = new CursosService(); como temos a classe serviço, a essa variável por ser apagada.

}

Depois de instanciarmos o ser serviço, no arquivo cursos.module.js é necessário criar o P*roviders* e importar o serviço:

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

// Módulo de funcionalidade não possui bootstrap

// A diferença de um módulo de funcionalidade e um de raiz, é o CommomModule

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

import { CursosComponent } from'./cursos.component';

import { CursoDetalheComponent } from'./curso-detalhe/curso-detalhe.component';

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

@NgModule({

imports: [

CommonModule

],

declarations: [

CursosComponent,

CursoDetalheComponent

],

exports: [

CursosComponent

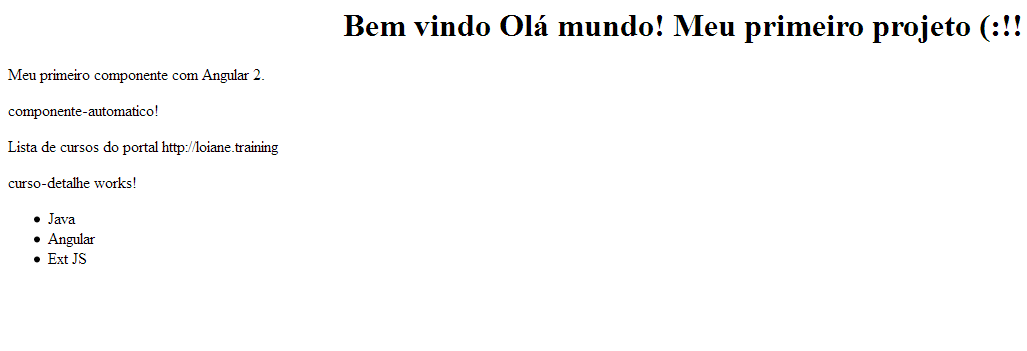
],

providers: [ CursosService ]

})

exportclassCursosModule { }

Depois disso é só executar seu projeto e o resultado no browser será esse:



**10 VÍDEO #08: DICAS PLUGINS ANGULAR PARA ATOM E VC CODE**

**10.1 Dicas de plugins para o Atom**

Para instalar os plugins, clique no menu Atom >*Preferences* (se você estiver usando Windows ou Linux, vá em *File*>*Settings*). Clique em *Install* e pesquise pelo pacote. Plugins sugeridos:

- angular-2-*typeScript-snippets*: Plugin que cria o esqueleto dos métodos.

- atom-*typescript*: plugin que é um compilador do *TypeScript*.

- *linter*: plugin responsável por sublinhar com a cor vermelha, as linhas onde ocorreram erros de compilação, facilitando assim a identificação das falhas.

- *file*-*icons*: Plugin que cria os ícones dos arquivos, facilitando a visualização dos mesmos.

- *open*-*recent*: Plugin que abre os projetos recentes e

- *pigments*: Plugin para ser usado com .css; ele preenche a cor de fundo do texto, com a cor passada no .css.

**10.2 Dicas de plugins para o Visual *Code***

Para instalar os plugins, clique em *Extensions* no menu esquerdo, pesquise por "Angular2" e serão listados todos os plugins disponíveis para o Angular. Plugins sugeridos:

- Angular 2, 4 *andupcominglatestTypeScrit* HTML *Snippets*: Pluginque cria o esqueleto dos métodos

- *AtonOneDark*: Plugin para deixar o tema colorido.

- Auto *Import*: Plugin que realiza automaticamente a importação da classe.

- HTML *Snippets*: Plugin que cria o esqueleto dos métodos

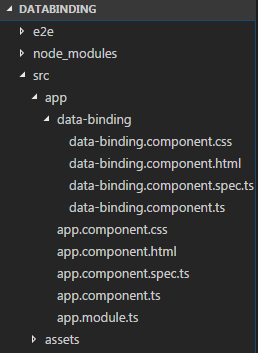
- *Ionic* 2 *CommandswithSnippets* (para quem for trabalhar com *Ionic*).

- *vscode*-*icons*: Plugin que cria os ícones dos arquivos, facilitando a visualização dos mesmos.

**11 VÍDEO #09: *PROPERTYBINDING* E INTERPOLAÇÃO**

**OBS:** Projeto disponível em: https://github.com/nasouza2/Angular2B/tree/master/databinding.

Data *Binging* significa associar informações do componente para o template ou vice-versa. Para esse capítulo, crie um novo projeto chamado *databinding* e crie um componente chamado *databinding*.



Abra o arquivo *data-binding.compenent.ts*, copie o selector e o cole no *app.component*.html (remova todo código criado automaticamente nesse arquivo):

<app-data-binding></app-data-binding>

No arquivo *databinding*.component.html é onde incluímos a interpolação ou a *PropertyBinding*:

<sectionclass="property-binding">

<article>

<h3> Interpolation / Interpolação </h3>

<!-- O Angular consegue fazer interpolação de variável:-->

<p> String renderizada com Interpolação: {{ url }} </p>

<!-- Ele também consegue resolver expressões matemáticas: -->

<p> Resultado de 1 + 1: {{ 1 + 1 }} </p>

<!-- Chamar metódos e: -->

<p> Resultado de 1 + 1 não é: {{ 1 + 1 + getValor() }} </p>

<!-- Resolver expressões boobleanas: -->

<p> Curso Angular e gostou do Curso {{ cursoAngular && getCurtirCurso () }} </p>

</article>

<article>

<h3> Interpolação com imagem e Property Binding </h3>

<!--Exemplo de interpolação com imagem: -->

<imgsrc="{{urlImagem}}">

<!--Exemplo de interpolação com Property Binding: -->

<!-- Nessa situação, usa-se [] em volta do nome da propriedade; o Angular consegue identificar uma Property Binding, não sendo necessário informar as {}. Fica a critério do programador utilizar a interpolação ou a Property Binding-->

<img [src]="urlImagem">

<p></p>

</article>

</section>

Lógica do arquivo *databinding.component.ts*:

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

@Component({

selector:'app-databinding',

templateUrl:'./databinding.component.html',

styleUrls: ['./databinding.component.css']

})

exportclassDatabindingComponentimplementsOnInit {

url: string = 'http://loiane.com'; // Variável

cursoAngular: boolean = true; // Expressão booleana

urlImagem = 'http://lorempixel.com/400/200/nature/'; // Imagem

// Metódo

getValor(){

return1;

}

// Metódo utilizado com a expressão booleana

getCurtirCurso(){

returntrue;

}

constructor() { }

ngOnInit() {

}

}

**21 VÍDEO #10: CLASS E STYLE BINDING**

O *Class* e o *Style* *Binding* também são classificadas como *Property Binding*; a diferença entre eles, é que ao invés de utilizarmos variáveis e expressões no Component (como visto no último capítulo), é que utilizamos .css.

Para esse capítulo, vamos utilizar os componentes de alerta do *Bootstrap*, que é adicionado ao nosso projeto, através do comando: npm install ngx-bootstrap bootstrap –save. Após a instalação, no package.json estarão as dependências do projeto:

"private": true,

"dependencies": {

"@angular/animations": "^4.0.0",

"@angular/common": "^4.0.0",

"@angular/compiler": "^4.0.0",

"@angular/core": "^4.0.0",

"@angular/forms": "^4.0.0",

"@angular/http": "^4.0.0",

"@angular/platform-browser": "^4.0.0",

"@angular/platform-browser-dynamic": "^4.0.0",

"@angular/router": "^4.0.0",

"bootstrap": "^3.3.7",

"core-js": "^2.4.1",

"ngx-bootstrap": "^1.7.1",

"rxjs": "^5.1.0",

"zone.js": "^0.8.4"

Agora vamos configurar o style: Abra o .angular-cli.json e na parte de styles, adicione “../node\_modules/bootstrap/dist/css/bootstrap.min.css”

"index": "index.html",

"main": "main.ts",

"polyfills": "polyfills.ts",

"test": "test.ts",

"tsconfig": "tsconfig.app.json",

"testTsconfig": "tsconfig.spec.json",

"prefix": "app",

"styles": [

"../node\_modules/bootstrap/dist/css/bootstrap.min.css",

"styles.css"

],

Vamos criar um combobox com as opções do bootstap: Abra o databinding.component.html e inclua uma <div> da seguinte forma:

<section class="property-binding">

<article>

<h3> Class and Style Binding </h3>

<div>

Selecione uma classe:

<select>

<option value="alert-sucess">Sucesso</option>

<option value="alert-info">Informação</option>

<option value="alert-warning">Atenção</option>

<option value="alert-danger">Erro</option>

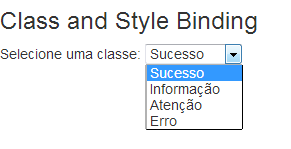
</select>

</div>

</article>

</section>

No navegador, o projeto ficou da seguinte forma:



Com o código dessa forma, não acontece nenhuma ação ao trocar de classe. Vamos então adicionar essa funcionalidade:

<section class="property-binding">

<article>

<h3> Class and Style Binding </h3>

<div>

Selecione uma classe:

<!--- Primeiro é necessário criar uma variável local. Para isso use o

#nomevariavel;

Para ouvir a variável e saber quando houve uma mudança, use o "change";

O "0" indica que nosso componente não executará nenhum método ao trocar

de opção-->

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

<option value="alert-success">Sucesso</option>

<option value="alert-info">Informação</option>

<option value="alert-warning">Atenção</option>

<option value="alert-danger">Erro</option>

</select>

<br><br>

<!--Adicionando as <div> para cada opção que pode ser selecionada;

Para que seja apresentada apenas a div/cor correspondente a opção

selecionada pelo usuário, será utilizado o class binding, que é definido

entre [];

Nesse class binding, estamos comparando se a opção selecionada no combobox,

é o alert-success-->

<div class="alert" role="alert"

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

<div class="alert" role="alert">Informação</div>

<div class="alert" role="alert">Atenção</div>

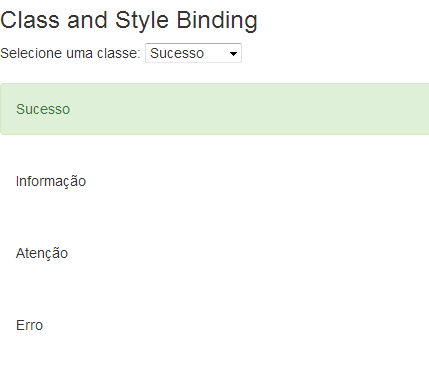
<div class="alert" role="alert">Erro </div>

</div>

</article>

</section>

Se a opção selecionada for “Sucesso”, o fundo ficará na cor verde; as outras opções não possuem ação. Resultado no navegador:



Aplicando o *class binding* para as outras opções:

<div class="alert" role="alert"

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

<div class="alert" role="alert"

[class.alert-info]="classe.value == 'alert-info'">Informação</div>

<div class="alert" role="alert"

[class.alert-warning]="classe.value == 'alert-warning'">Atenção</div>

<div class="alert" role="alert"

[class.alert-danger]="classe.value == 'alert-danger'">Erro </div>

</div>

No navegador, o resultado foi:

- Cor verde para Sucesso;

- Cor azul para Informação;

- Cor amarela para Atenção e

- Cor vermelha para Erro.









Também é possível fazer interpolação com o Class Binding: crie uma nova <div> com o class=”alert”. Como o nome da classe do .css já é o valor do combobox, podemos pegar a classe.value:

</select>

<br><br>

<!-- Interpolação com Class Binding -->

<div class="alert {{classe.value }}" role="alert"> Texto a ser colorido conforme

valor do combobox!</div>

<div class="alert" role="alert"

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

<div class="alert" role="alert"

[class.alert-info]="classe.value == 'alert-info'">Informação</div>

<div class="alert" role="alert"

[class.alert-warning]="classe.value == 'alert-warning'">Atenção</div>

<div class="alert" role="alert"

[class.alert-danger]="classe.value == 'alert-danger'">Erro </div>

No navegador, o resultado é esse:

- Cor verde para Sucesso;

- Cor azul para Informação;

- Cor amarela para Atenção e

- Cor vermelha para Erro.









Exemplo de *Style Binding*:

<div class="alert" role="alert"

[class.alert-danger]="classe.value == 'alert-danger'">Erro </div>

<!-- Exemplo de Style Binding

Aqui estamos comparando se a opção selecionada é do tipo "alert-danger";

se for, então executa o estilo "block"; senão executa o style "none"

block = mostrar nome = esconder-->

<div class="alert alert-danger" role="alert"

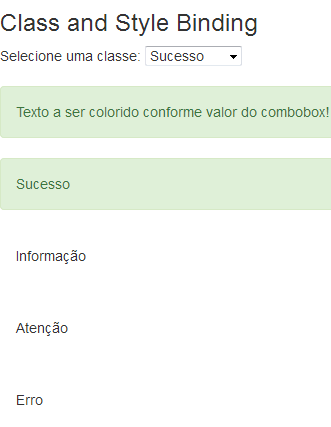
[style.display]="classe.value == 'alert-danger' ? 'block' : 'none'">

Esse texto só vai aparecer em caso de erro! </div>

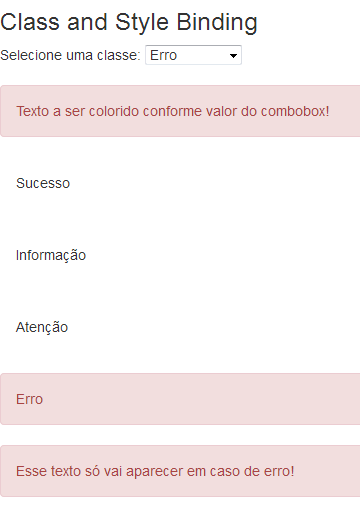
</div>

No navegador, o resultado é esse:

- Qualquer opção diferente de Erro, não apresenta o texto:



- Se a opção for igual a Erro, então apresenta:



**Dificuldades encontradas:** No vídeo 10, o comando para instalar o bootstrap era ”ng2-bootstrap –save” porém a página do *boostrap* foi alterada e agora o comando é “npm install ngx-bootstrap bootstrap --save”

**Links úteis:** Bibliotecas do *bootstrap* para o Angular 2 e Angular 4: http://valor-software.com/ngx-bootstrap/#/

**22 VÍDEO #11: EVENT BINDING**

O *Event Binding* é a forma que escutamos o nosso component e executamos uma ação com ele. No databinding.component.html, crie o seguinte código:

<section class="event-binding">

<article>

<h3> Event Binding </h3>

<div>

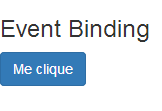
<button class="btn btn-primary"> Me clique</button>

</div>

</article>

</section>

No navegador, o resultado será esse:



Se você clicar no botão, não haverá nenhuma ação, porque não implementamos nenhum método para ele.

Implementando um event com método:

<section class="event-binding">

<article>

<h3> Event Binding </h3>

<div>

<!-- O evento deve ser passado entre ();

Podemos também chamar um método, como é o caso de "BotaoClicado"

-->

<button (click)="BotaoClicado()"

class="btn btn-primary"> Me clique</button>

</div>

</article>

</section>

No databinding.component.ts, criamos o método BotaoClicado, que exibe uma alerta ao ser clicado:

BotaoClicado(){

alert('Botão clicado!');

}

No navegador, o resultado é:



**Link útil**: Todos os eventos que são possíveis de serem feitos: https://developer.mozilla.org/en-US/docs/Web/Events.

**23 VÍDEO #12: TWO-WAY DATA BINDING**

Two-Way data binding deve ser usado quando queremos atualizar o template e o componente ao mesmo tempo.

Como lembrar da sintaxe correta? A sintaxe correta é chamada de Sintaxe de Banana, ou seja, primeiro as chaves simbolizando a caixa e depois os parênteses, simbolizando a banana: [()].

<section>

<article class="two-way data binding">

<h3> Two way data binding </h3>

<div>

<!--Primeiro exemplo:

[value] associação de propriedade

(input) associação de evento-->

<input type="text"

[value]="nome"

(input)="nome = $event.target.value"

/>

<!--Segundo exemplo:

Quando utilizamos o ngModelChange, não é necessário usar o target.value-->

<input type="text"

[ngModel]="nome"

(ngModelChange)="nome = $event" />

<!--Fazemos o two-way informando no evento, o binding de propriedade [] + o binding de evento ()-->

<input type="text" [(ngModel)]="nome">

<p>Você digitou : {{nome}} </p>

</div>

</article>

</section>

Exemplo de two-way com objeto:

Em databinding.component.ts, foi criado o objeto pessoa:

pessoa: any ={

nome: 'def',

idade: 20

}

Em databinding.component.html, utilizamos o objeto + atributo e para apresentar os dados, usamos a interpolação:

<div>

<p>Meu nome é {{pessoa.nome}} e tenho {{pessoa.idade}} anos de idade!</p>

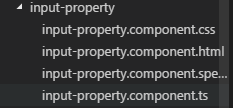
<input type="text" [(ngModel)]="pessoa.nome"> <!—Pegamos o objeto e o atributo -->

<input type="text" [(ngModel)]="pessoa.idade">

</div>

**23 VÍDEO #13: REUSANDO COMPONENTES *INPUT* *PROPERTIES***

O objetivo de usarmos *input* *properties*, é de deixar o código mais limpo e mais organizado. Exemplo: No nosso projeto, foi criado um componente chamado *input-component*:



Em databinding.component.ts, criamos uma variável do tipo string:

nomeDoCurso: string = 'Curso de Angular';

Em databinding.component.html, usamos a variável criada:

<section class="input-output-properties">

<article>

<h3> Input/Output Properties</h3>

<app-curso [nome]="nomeDoCurso"> </app-curso>

</article>

</section>

Em input-property.component.ts, usamos o Input Properties da seguinte forma:

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

@Component({

selector: 'app-curso',

templateUrl: './input-property.component.html',

styleUrls: ['./input-property.component.css']

})

export class InputPropertyComponent implements OnInit {

// Primeira forma de usar o input properties:

// Através do @Input nós conseguimos expor uma propriedade chamada "nome" para o seletor

app-curso

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

// Também podemos passar o properties como parâmetro, usando ela internamente como uma

variável normal (nomeCurso) e externamente com a variável de exposição

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

constructor() { }

ngOnInit() {

}

}

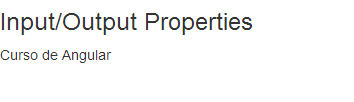
Em input-property.component.html, usamos a interpolação com a variável criada:

<p>

{{nomeCurso }}

</p>

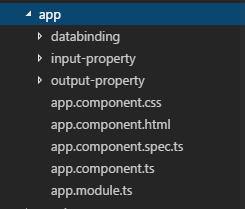
No navegador, o resultado é esse:



**24 VÍDEO #14: EMITINDO EVENTOS COM *OUTPUT* *PROPERTIES***

O objetivo desse capítulo é disparar um evento chamado “mudou”, toda vez que o usuário clicar nos botões “+” ou “-”.

Utilizando o projeto do vídeo anterior, criamos um novo component chamado de output-property:



Em output-property.components.ts, renomeie o seletor para “contador” e em databinding.component.html, passe esse contador:

- output-property.components.ts:

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

@Component({

selector: 'contador',

templateUrl: './output-property.component.html',

styleUrls: ['./output-property.component.css']

})

export class OutputPropertyComponent implements OnInit {

constructor() { }

ngOnInit() {

}

}

- databinding.component.html

<section class="input-output-properties">

<article>

<h3> Input/Output Properties</h3>

<app-curso [nome]="nomeDoCurso"> </app-curso>

<contador> </contador>

</article>

</section>

Vamos trabalhar em cima do output-property.component.html:

- Criamos dois botões e um input, com a variável “valor” que iniciando em “0”:

<div>

<button class="btn-primary">+ </button>

<input type="text" [value]="valor" readonly> <!--Usa-se o readonly para não deixar o

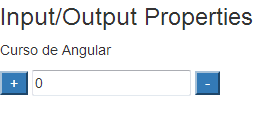
usuário alterar o campo-->

<button class="btn-primary">- </button>

</div>

No navegador, o resultado é esse:

- Ao clicar nos botões não acontecerá nada, porque não existe ação para eles ainda.



Em output-property.components.html, vamos ouvir o clique do mouse e chamar os métodos incrementa e decrementa:

<div>

<button class="btn-primary" (click)="incrementa ()">+ </button>

<input type="text" [value]="valor" readonly>

<button class="btn-primary" (click)="decrementa ()">- </button>

</div>

Metódos criados em output-property.components.ts:

incrementa(){

this.valor++;

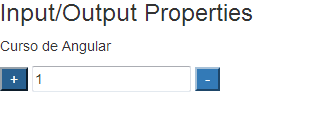
}

decrementa(){

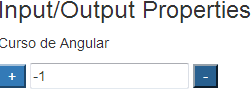
this.valor--;

}

No navegador, o resultado é esse para “+”:



E esse, para “-“:



1. Site oficial do framework: https://angular.io/ [↑](#footnote-ref-2)
2. Diretório do Angular 2 no GitHub: https://github.com/angular [↑](#footnote-ref-3)