



DESENVOLVIMENTO WEB – FRONT END

AULA 3



Prof. Mauricio Antonio Ferste



CONVERSA INICIAL

Quando utilizados em conjunto, o TypeScript, módulos e componentes fornecem uma base sólida para o desenvolvimento de aplicativos escaláveis e robustos no Angular. Eles ajudam a melhorar a legibilidade, a modularidade e a reutilização do código, promovendo uma estrutura organizada e facilitando a colaboração entre os desenvolvedores. Além disso, o TypeScript fornece recursos avançados, como a tipagem estática, que contribuem para a detecção precoce de erros e uma melhor experiência de desenvolvimento.

TEMA 1 – JAVASCRIPT VS TYPESCRIPT – ADIÇÕES DO TYPESCRIPT

JavaScript é uma linguagem de programação amplamente utilizada para desenvolvimento *web*. Ela permite adicionar interatividade e funcionalidades dinâmicas a páginas da *web*. É uma linguagem de *script* interpretada, executada no navegador do cliente.

TypeScript é um *superset* do JavaScript que adiciona recursos de tipagem estática e recursos de programação orientada a objetos à linguagem. Ele oferece uma camada de abstração adicional e permite que os desenvolvedores escrevam código JavaScript mais seguro e escalável.

Em resumo, JavaScript é uma linguagem de programação popular usada para desenvolvimento *web*, enquanto o TypeScript é uma extensão do JavaScript que adiciona recursos de tipagem estática e programação orientada a objetos. Ambas as linguagens são amplamente utilizadas e oferecem recursos poderosos para o desenvolvimento de aplicativos *web* modernos. Existem algumas diferenças importantes entre TypeScript e JavaScript:

- **Tipagem:** TypeScript é um *superset* de JavaScript que adiciona recursos de tipagem estática à linguagem. Isso significa que você pode definir tipos de dados para variáveis, parâmetros de função e retornos de função em TypeScript, o que ajuda a detectar erros de tipo em tempo de compilação e torna o código mais seguro;
- **Orientação a objetos:** TypeScript suporta recursos de programação orientada a objetos, como classes, interfaces, herança e polimorfismo, que não estão presentes no JavaScript puro. Esses recursos permitem uma estruturação mais clara e organizada do código;



- **Compatibilidade:** todo código JavaScript é válido em TypeScript. Isso significa que você pode usar bibliotecas e *frameworks* JavaScript existentes em projetos TypeScript sem a necessidade de modificá-los. No entanto, o TypeScript não possui acesso a recursos específicos da versão mais recente do JavaScript até que sejam adicionados aos padrões da linguagem;
- **Compilação:** o código TypeScript precisa ser compilado para JavaScript antes de ser executado em um navegador ou ambiente de tempo de execução. Isso é feito por meio do compilador TypeScript, que converte o código TypeScript em JavaScript equivalente. O JavaScript, por sua vez, é uma linguagem interpretada e não requer uma etapa de compilação separada;
- **Ferramentas de desenvolvimento:** o TypeScript é altamente integrado com ferramentas de desenvolvimento modernas, como o ambiente de desenvolvimento integrado (IDE) Visual Studio Code. Ele oferece recursos avançados, como conclusão de código, detecção de erros em tempo real e refatoração automatizada, que podem melhorar significativamente a produtividade do desenvolvedor.

Em resumo, o TypeScript adiciona recursos de tipagem estática e programação orientada a objetos ao JavaScript, oferecendo maior segurança, estruturação e produtividade no desenvolvimento de aplicativos. No entanto, o JavaScript continua sendo a linguagem principal da *web* e o TypeScript é uma escolha opcional que traz benefícios adicionais para projetos mais complexos ou que exigem maior segurança e escalabilidade (Typescript, 2023).

1.1 Instalando Typescript

Já vimos que o TypeScript oferece recursos poderosos para o desenvolvimento Angular, como a verificação estática de tipos, autocompletar e refatoração avançada. Aproveite esses recursos para desenvolver aplicativos robustos e de alta qualidade com o Angular.



Saiba mais

Vamos tratar de desenvolvimento, e, em desenvolvimento existem vários comandos que temos de aprender para usar além do código em si. Seguem alguns:

Comando (Windows)	Comando (Mac OS / Linux)	Descrição	Exemplo
exit	Exit	Fecha a janela	exit
cd	Cd	Muda a pasta	cd test
cd	Pwd	Mostra o diretório atual	cd (Windows) ou pwd (Mac OS / Linux)
dir	Ls	Lista as pastas e/ou arquivos	dir
copy	Cp	Copia um arquivo	copy c:\test\test.txt c:\windows\test.txt
move	Mv	Move um arquivo	move c:\test\test.txt c:\windows\test.txt
mkdir	Mkdir	Cria uma pasta	mkdir testdirectory
rmdir (ou del)	Rm	Exclui arquivo	del c:\test\test.txt



<code>rmdir /S</code>	<code>rm -r</code>	Exclui diretório	<code>rm -r testdirectory</code>
<code>[CMD] /?</code>	<code>man [CMD]</code>	obtem ajuda para um comando	<code>cd /?</code> (Windows) or <code>man cd</code> (Mac OS / Linux)

Considere essa tabela como o mínimo para trabalhar, mas é o suficiente.

Dependendo da configuração do sistema, pode ser necessário às vezes instalar o Typescript

```
npm install -g typescript
```

Eventualmente ele pode estar instalado devido a outra configuração anteriormente instalada, inclusive algum passo do próprio `angular@cli`.

Com o comando abaixo podemos ver a versão do Typescript instalado.

```
tsc -v
```

Dependendo do ambiente de tempo de execução JavaScript no qual você pretende executar seu código, pode haver uma configuração básica que você pode usar em github.com/tsconfig/bases que é o repositório Typescript/Node.

1.2 Tipagem

A tipagem é uma característica fundamental do TypeScript que oferece vantagens significativas em relação ao JavaScript. No TypeScript, podemos definir tipos de dados explícitos para variáveis, parâmetros de função, retornos de função, propriedades de objetos e muito mais. Essa abordagem fornece um sistema de tipos estáticos que ajuda a identificar erros em tempo de compilação e a melhorar a qualidade e a robustez do código.

Ao utilizar a tipagem no TypeScript, podemos especificar o tipo de uma variável, como *string*, *number*, *boolean*, entre outros. Isso permite que o compilador verifique se os valores atribuídos às variáveis estão de acordo com seus respectivos tipos. Isso ajuda a evitar erros comuns, como atribuir um valor inválido a uma variável ou realizar operações incompatíveis. Essa verificação em tempo de compilação é extremamente valiosa, pois ajuda a identificar erros antes



mesmo de executar o código. Além de definir tipos para variáveis, a tipagem também é aplicada em funções e objetos. Podemos especificar os tipos dos parâmetros de uma função, bem como o tipo de retorno esperado. Isso facilita a compreensão do contrato da função e ajuda a evitar erros ao chamar a função com argumentos incorretos. Da mesma forma, ao definir objetos, podemos especificar os tipos de suas propriedades, o que torna mais fácil entender a estrutura do objeto e garantir que ele seja usado corretamente.

Outro aspecto importante da tipagem no TypeScript é a capacidade de lidar com tipos opcionais e valores padrão. Podemos indicar que um parâmetro de função ou uma propriedade de objeto é opcional, permitindo que eles sejam omitidos. Além disso, podemos fornecer valores padrão para parâmetros opcionais, simplificando o uso da função sem a necessidade de especificar todos os argumentos. A tipagem no TypeScript traz diversos benefícios. Além de fornecer maior segurança e confiabilidade ao código, ela também melhora a legibilidade e a manutenção. Ao ter informações claras sobre os tipos de dados em uso, os desenvolvedores podem entender melhor o comportamento do código, facilitando a colaboração e a detecção de erros. Além disso, a tipagem fornece recursos avançados em IDEs, como autocompletar, verificação de tipos durante a escrita do código e refatoração automatizada, o que aumenta a produtividade e a eficiência no desenvolvimento.

Em resumo, a tipagem no TypeScript é uma poderosa ferramenta que permite especificar tipos de dados em variáveis, funções e objetos. Essa abordagem estática traz benefícios significativos, como detecção de erros em tempo de compilação, melhor legibilidade e manutenção do código, além de recursos avançados em IDEs. Ao adotar a tipagem, os desenvolvedores podem criar aplicativos mais confiáveis, seguros e escaláveis.

Alguns detalhes sobre tipagem no Typescript:

Tipo explícito

Explícito - escrevendo o tipo:

```
let primeironome: string = "João";
```

Tipo Implícito

Implícito - o TypeScript irá "adivinhar" o tipo, com base no valor atribuído:

```
let Primeironome= "João";
```



1.2.1 Definindo tipos de variáveis

Essas definições permitem especificar o tipo de dado que cada variável pode armazenar, o que ajuda a garantir a consistência e o correto uso das variáveis durante a execução do programa.

```
let nome : string = "João"; // Variável nome do tipo string
let idade: number = 30; // Variáveis idade do tipo number
let ativo: boolean = true; // Variável ativo do tipo boolean
```

nome: É uma variável do tipo string, que armazena texto. No exemplo dado, o valor atribuído à variável nome é "João".

idade: É uma variável do tipo number, que armazena valores numéricos. No exemplo dado, o valor atribuído à variável idade é 30.

ativo: É uma variável do tipo boolean, que armazena valores lógicos (verdadeiro ou falso). No exemplo dado, o valor atribuído à variável ativo é true (verdadeiro).

1.2.2 Tipos de funções

No código abaixo, `let resultado: number = somar (5, 3);`, chama a função "somar" passando os argumentos 5 e 3 e atribui o resultado retornado pela função à variável "resultado". Nesse caso, como a função "somar" retorna um valor do tipo number, a variável "resultado" é declarada como do tipo number.

Portanto, no exemplo dado, temos uma função chamada "somar" que recebe dois parâmetros do tipo number e retorna um valor do tipo number. Essa função é chamada com os argumentos 5 e 3, e o resultado retornado é atribuído à variável "resultado".

```
function somar (a:number, b:number) : number {
  return a + b;
}
let resultado: number = somar (5, 3);
```

1.2.3 Tipos de objetos

Um bom exemplo de implementação de tipo é uma interface que é uma forma de definir a estrutura de um objeto. A interface permite especificar quais



propriedades e métodos um objeto deve ter. No caso da interface "pessoa", poderíamos defini-la da seguinte forma:

```
interface Pessoa {  
  nome: string;  
  idade: number;  
  ativo: boolean;  
}
```

Nesse exemplo, a interface "Pessoa" possui três propriedades:

- nome: do tipo string, representa o nome da pessoa.
- idade: do tipo number, representa a idade da pessoa.
- ativo: do tipo boolean, indica se a pessoa está ativa ou não.

Essa interface serve como um contrato que define a estrutura que objetos do tipo "Pessoa" devem seguir. Por exemplo, um objeto válido que segue essa interface poderia ser:

```
let pessoa1: Pessoa = {  
  nome: "João",  
  idade: 30,  
  ativo: true  
};
```

Nesse caso, pessoa1 é um objeto que possui as propriedades nome, idade e ativo, todas com os tipos especificados pela interface "Pessoa".

1.3 Orientação a objetos

Se já viu classes e orientação a objetos (OO), em Java ou Python vai se sentir em casa com Typescript, a implementação basicamente é a mesma, seguindo rigidamente os conceitos de OO.

Classes:

Você pode definir classes usando a palavra-chave **class**. Uma classe é uma estrutura que encapsula dados (propriedades) e comportamentos (métodos) relacionados. Exemplo:

```
class Animal {  
  private nome: string;
```




```
private idade: number;

constructor (nome: string, idade: number) {
  this.nome = nome;
  this.idade = idade;
}

public emitirSom(): void {
  console.log("O animal emite um som.");
}

public getNome(): string {
  return this.nome;
}

public getIdade(): number {
  return this.idade;
}
}
```

Interessante entender que Typescript, que se assemelha a Java e Javascript não executa diretamente, mas na verdade sempre é compilado para Javascript utilizando o compilador TSC, no caso, ele transformou a classe do Typescript no padrão Prototype que é utilizado pelo JavaScript para substituir classes.

Por isso o Typescript é um grande avanço, traz um padrão de orientação à objetos para o Javascript indiretamente, auxiliando na abstração.

```
// Classe Animal
class Animal {
  constructor(public nome: string) {}

  public emitirSom(): void {
    console.log("O animal emite um som.");
  }
}

// Classe Gato que herda de Animal
```



```
class Gato extends Animal {  
    constructor(nome: string) {  
        super(nome);  
    }  
  
    public emitirSom(): void {  
        console.log("O gato mia.");  
    }  
}  
  
// Criando instâncias das classes  
let animal1: Animal = new Animal("Cachorro");  
let gato1: Gato = new Gato("Bichano");  
  
// Chamando o método emitirSom() em cada instância  
animal1.emitirSom(); // Saída: O animal emite um som.  
gato1.emitirSom(); // Saída: O gato mia.
```

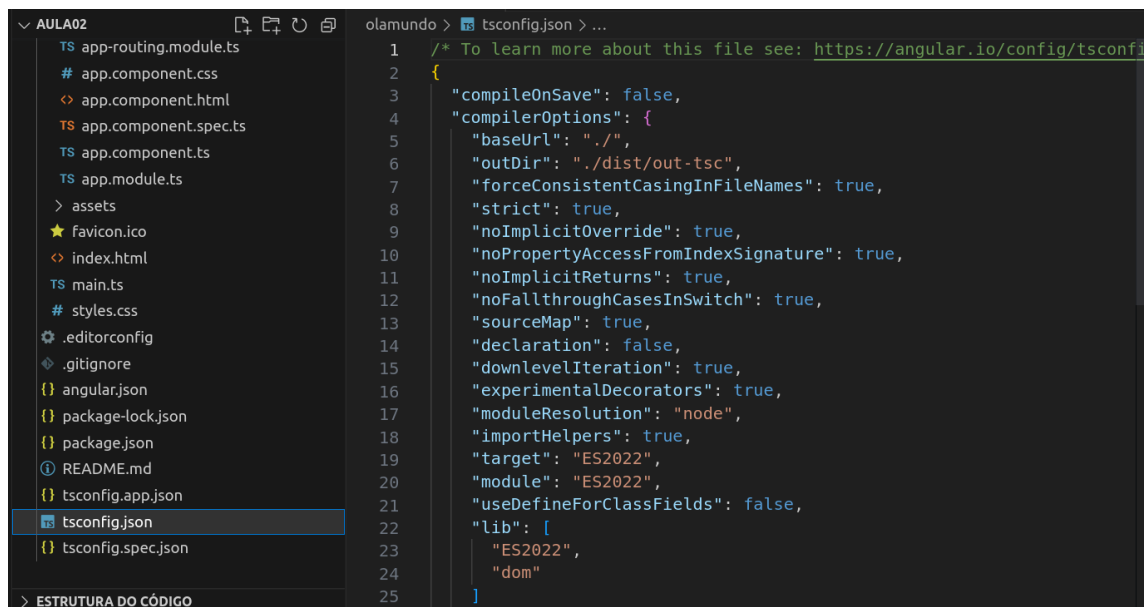
Outra característica fundamental inerente da orientação à objetos é Herança e Polimorfismo, no exemplo abaixo, podemos acompanhar a herança, no extends de gato de animal e o polimorfismo, na redefinição do método Ola. E funciona!

Ou seja, é uma linguagem orientada à objetos com as necessidades para o desenvolvimento de um sistema grande porte e robusto, com segurança.

É importante mencionar que existe um arquivo de configuração usado em projetos TypeScript para definir as configurações do compilador TypeScript.

tsconfig.json:

O arquivo está localizado na raiz do diretório do projeto TypeScript e é usado pelo compilador do TypeScript para determinar como compilar o código TypeScript em JavaScript.



```
1  /* To learn more about this file see: https://angular.io/config/tsconfi
2  {
3    "compileOnSave": false,
4    "compilerOptions": {
5      "baseUrl": "./",
6      "outDir": "./dist/out-tsc",
7      "forceConsistentCasingInFileNames": true,
8      "strict": true,
9      "noImplicitOverride": true,
10     "noPropertyAccessFromIndexSignature": true,
11     "noImplicitReturns": true,
12     "noFallthroughCasesInSwitch": true,
13     "sourceMap": true,
14     "declaration": false,
15     "downlevelIteration": true,
16     "experimentalDecorators": true,
17     "moduleResolution": "node",
18     "importHelpers": true,
19     "target": "ES2022",
20     "module": "ES2022",
21     "useDefineForClassFields": false,
22     "lib": [
23       "ES2022",
24       "dom"
25     ]
26   }
27 }
```

O **tsconfig.json** é uma parte importante de qualquer projeto TypeScript e é usado para configurar as opções do compilador TypeScript. Ele permite que os desenvolvedores personalizem a forma como o TypeScript é compilado em JavaScript e fornece uma maneira conveniente de organizar e gerenciar projetos TypeScript complexos.

Aqui estão algumas das principais configurações definidas no arquivo **tsconfig.json**:

- **compilerOptions**: esta seção define as opções do compilador TypeScript, como o nível de compatibilidade, o caminho de saída, o tipo de módulo, as opções de *target*, entre outras;
- **files**: esta seção define uma lista de arquivos TypeScript para incluir na compilação. Se esta seção estiver presente, o compilador apenas compilará os arquivos listados;
- **include**: esta seção define os padrões de arquivos TypeScript que devem ser incluídos na compilação;
- **exclude**: esta seção define os padrões de arquivos TypeScript que devem ser excluídos da compilação;
- **extends**: esta seção permite que o arquivo **tsconfig.json** estenda outro arquivo **tsconfig.json**;
- **references**: esta seção permite que o arquivo **tsconfig.json** especifique as dependências entre diferentes projetos TypeScript.



Um dos pontos mais cruciais na tecnologia atual web baseados em tecnologia Javascript – Typescript – NodeJS é esta conversão automática, que segue um padrão ECMAScript, que é a versão de desenvolvimento web.

ECMAScript é uma linguagem de programação, baseada em scripts, padronizada pela Ecma International na especificação ECMA-262 e ISO/IEC 16262. Esta linguagem é usada em tecnologias para Internet para a criação de scripts executados no cliente e no servidor (Node.JS), sendo inspirada na linguagem de programação JavaScript, além de permitir outras implementações independentes (ECMA, 2023).

TEMA 2 – TYPESCRIPT NO ANGULAR

O TypeScript é a linguagem de programação principal usada no desenvolvimento de aplicativos Angular. Ele é um superconjunto de JavaScript que adiciona recursos adicionais, como tipagem estática, classes, interfaces, módulos e muito mais.

2.1 Arquivos de configuração

Já trabalhamos com vários arquivos e, de uma maneira geral este processo de construção deve estar já amadurecido para o leitor, mas para contextualizar somente os arquivos relacionados com Typescript no Angular, importantes para o desenvolvimento de aplicativos. Aqui estão alguns dos arquivos mais comuns:

1. **tsconfig.json (já visto)**: este é o arquivo de configuração principal do TypeScript. Ele define as opções de compilação e as configurações do projeto. Nele, você pode especificar o diretório de saída, a versão do ECMAScript a ser usada, habilitar ou desabilitar recursos específicos do TypeScript e muito mais;
2. **tslint.json**: este arquivo define as regras e configurações para o linter TypeScript. O linter ajuda a manter um código consistente e livre de erros, aplicando regras de estilo e boas práticas de programação. Você pode definir suas próprias regras ou usar configurações predefinidas;
3. **angular.json**: este arquivo de configuração é específico do Angular e define as configurações do projeto, como as dependências, os arquivos a serem compilados, a estrutura do diretório, as tarefas de construção, a



configuração do servidor de desenvolvimento e muito mais. É um arquivo essencial para a configuração geral do projeto Angular.

Na Figura 1, temos o exemplo do projeto que já criamos na aula 2:

Figura 1 – Estrutura do projeto criado na aula 2

The screenshot shows a code editor with two panels. The left panel displays the file structure of a project named 'AULA02'. The right panel shows the contents of the 'angular.json' file.

File Structure (Left Panel):

- TS app-routing.module.ts
- # app.component.css
- <> app.component.html
- TS app.component.spec.ts
- TS app.component.ts
- TS app.module.ts
- > assets
- ★ favicon.ico
- <> index.html
- TS main.ts
- # styles.css
- ⚙ .editorconfig
- 📄 .gitignore
- { } angular.json
- { } package-lock.json
- { } package.json
- 📖 README.md
- { } tsconfig.app.json
- 📘 tsconfig.json
- { } tsconfig.spec.json
- > ESTRUTURA DO CÓDIGO

angular.json Content (Right Panel):

```
1 {
2   "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
3   "version": 1,
4   "newProjectRoot": "projects",
5   "projects": {
6     "olamundo": {
7       "projectType": "application",
8       "schematics": {},
9       "root": "",
10      "sourceRoot": "src",
11      "prefix": "app",
12      "architect": {
13        "build": {
14          "builder": "@angular-devkit/build-angular:browser",
15          "options": {
16            "outputPath": "dist/olamundo",
17            "index": "src/index.html",
18            "main": "src/main.ts",
19            "polyfills": [
20              "zone.js"
21            ],
22            "tsConfig": "tsconfig.app.json",
23            "assets": [
24              "src/favicon.ico",
25              "src/assets"
```

Fonte: Ferste, 2023.

3. **karma.conf.js**: se você estiver usando o Karma como test runner para seus testes unitários no Angular, este arquivo contém a configuração para a execução dos testes. Você pode definir os arquivos a serem testados, os frameworks a serem usados, os reporters de resultados e outras opções relacionadas aos testes;
4. **protractor.conf.js**: se você estiver usando o Protractor para executar testes de ponta a ponta (e2e) no Angular, este arquivo contém a configuração para os testes e2e. Você pode definir os caminhos dos arquivos de teste, as configurações do navegador, as URLs a serem testadas e outras opções relevantes para os testes e2e.

Esses são apenas alguns dos arquivos de configuração mais comuns ao trabalhar com o TypeScript no contexto do Angular. Eles fornecem opções e configurações importantes para a compilação, *linting*, teste e execução do aplicativo Angular. Certifique-se de revisar e ajustar esses arquivos de acordo com as necessidades e requisitos do seu projeto.



TEMA 3 – MÓDULOS ANGULAR – COMPONENTES

No Angular, os módulos e componentes são fundamentais para a organização e estruturação de uma aplicação. Os módulos permitem agrupar e organizar componentes, serviços e outros recursos relacionados, enquanto os componentes encapsulam partes específicas da interface do usuário. Um módulo Angular é uma unidade funcional que contém um conjunto de componentes, serviços, diretivas e outros artefatos necessários para uma determinada funcionalidade ou recurso da aplicação. Ele é definido por meio da anotação **@NgModule** e geralmente possui um arquivo separado, como **app.module.ts**. No módulo, é possível importar outros módulos e declarar os componentes que serão utilizados.

Os componentes, por sua vez, são os blocos de construção da interface do usuário em uma aplicação Angular. Cada componente é composto por um template HTML, que define a estrutura do componente, estilos CSS ou SCSS, que definem a aparência visual, e uma classe TypeScript, que define o comportamento e a lógica do componente. Através da anotação **@Component**, o componente é registrado em um módulo, especificando seu seletor, *template*, estilos e outros metadados. Ao criar e registrar componentes em módulos, é possível alcançar uma organização modular e reutilizável da aplicação Angular. Os módulos permitem dividir a funcionalidade em partes distintas, facilitando a manutenção e o desenvolvimento colaborativo. Os componentes, por sua vez, permitem a criação de partes independentes e reutilizáveis da interface do usuário, proporcionando uma abordagem modular para o desenvolvimento de interfaces complexas.

Além disso, a utilização de módulos e componentes no Angular contribui para a escalabilidade da aplicação, permitindo adicionar novas funcionalidades ou componentes de forma isolada, sem afetar o restante da aplicação. Isso promove a separação de responsabilidades e a fácil reutilização de componentes em diferentes partes do aplicativo. Em resumo, os módulos e componentes no Angular são elementos cruciais para a organização, modularização e reutilização de código em uma aplicação. Eles fornecem uma estrutura eficiente e flexível para o desenvolvimento de interfaces de usuário complexas e escaláveis, contribuindo para um código mais legível, modular e de fácil manutenção.



3.1 Criando um módulo

Em Angular, os módulos são uma forma de organizar e agrupar componentes, diretivas, serviços e outros recursos relacionados em um contexto específico. Um módulo é uma classe decorada com o decorator **@NgModule**.

Saiba mais

1. Criamos já um projeto anteriormente, na unidade 2, e lá tivemos de instalar o NodeJS, NPM e VsCode (1.1, 1.2, 1.3)
2. Então criamos o projeto com `ng new web`, aqui faremos assim, dentro do diretório `faculdade`, teremos:
`projetoweb-> raiz`
`web-> NOSSO DIRETÓRIO DE ESTUDO`
`api-> nossa api para consumo de dados`
3. Entrar no diretório, por exemplo, no windows, abrir o terminal e fazer um `cd` para o diretório.
4. Depois chamar o editor, como VsCode com o comando `code`.

Então, gerando o componente com o comando **ng generate component meucomponente**, temos a criação do componente como na imagem da figura 02, onde podemos ver as estruturas que são criadas:

Figura 2 – Acompanhamento da criação de um componente, sempre o angular gera muito conteúdo, note os arquivos que foram gerados

```
? Would you like to share pseudonymous usage data about this project with the Angular Team
at Google under Google's Privacy Policy at https://policies.google.com/privacy. For more
details and how to change this setting, see https://angular.io/analytics. Yes
Pre Thank you for sharing pseudonymous usage data. Should you change your mind, the following
command will disable this feature entirely:
ng analytics disable
Pre Global setting: enabled
Local setting: enabled
Effective status: enabled
CREATE src/app/meucomponente/meucomponente.component.css (0 bytes)
CREATE src/app/meucomponente/meucomponente.component.html (28 bytes)
CREATE src/app/meucomponente/meucomponente.component.spec.ts (608 bytes)
CREATE src/app/meucomponente/meucomponente.component.ts (230 bytes)
UPDATE src/app/app.module.ts (503 bytes)
```

Fonte: Ferste, 2023.

Para criar um novo módulo no Angular, siga estes passos:



Crie um novo arquivo TypeScript para o módulo, por exemplo, **meu-componente.ts**. Por exemplo, inicialmente o arquivo ts do componente tem a estrutura abaixo:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-meucomponente',
  templateUrl: './meucomponente.component.html',
  styleUrls: ['./meucomponente.component.css']
})
export class MeucomponenteComponent {
}
```

O que verificamos acima é a criação básica do componente, com a importação dos módulos, componentes e outros recursos que serão utilizados no módulo.

No bloco de *declarations*, liste os componentes, diretivas e *pipes* que fazem parte do módulo. Em Angular, os *pipes* são recursos poderosos que permitem transformar e formatar dados exibidos no *template* de um componente de forma fácil e legível. Os *pipes* são usados para fazer transformações em *strings*, datas, números e outros tipos de dados antes de serem exibidos na interface do usuário. Os *pipes* são aplicados no *template* do componente usando o caractere de barra vertical (|) e são seguidos pelo nome do *pipe* e, opcionalmente, por um ou mais argumentos separados por dois pontos. A sintaxe geral para usar um *pipe* é a seguinte:

No bloco de **imports**, são importados módulos necessários para o funcionamento do módulo.

Decorator **@Component**: Define o seletor do componente como app-usuario. Define o template HTML do componente como ./usuario.component.html. Define o arquivo de estilo do componente como ./usuario.component.scss.

Exporte o módulo criado na classe **export class MeuModulo**.

Vamos para um exemplo mais concreto, considerando o projeto que estamos criando, veja abaixo o código e depois a análise:



```
import { Component, OnInit } from '@angular/core';
import { UsuarioService } from '../../services/usuario.service'
import { UsuarioModel } from '../../model/usuarioModel'
import { Router, ActivatedRoute } from '@angular/router';
import { MatFormFieldModule } from '@angular/material/form-
field';
import { IUsuario } from '../../interfaces/IUsuario'
import { MatTableDataSource } from '@angular/material/table';
```

```
@Component({
  selector: 'app-usuario',
  templateUrl: './usuario.component.html',
  styleUrls: ['./usuario.component.scss']
})
export class UsuarioComponent implements OnInit {
  model: UsuarioModel = new UsuarioModel();
  constructor(
    private userSrv: UsuarioService,
    private router: Router,
    private active: ActivatedRoute
  ) { }

  ngOnInit() { // lifecyclehook, mostrado mais adiante no item 4,
    ele é chamado ao iniciar o bloco de código.
    this.active.params.subscribe(p => this.getId(p['id']));
  }

  async getId (id: string): Promise<void> {
    // futuramente retornará um id para nosso usuário
  }

  async save(): Promise<void> {
    // futuramente salvará nosso usuário
  }
}
```

Aqui está uma análise do código:

Component e OnInit são importados do @angular/core.



UsuarioService é importado de '../services/usuario.service', que é um serviço personalizado para manipulação de dados de usuários, passaremos o código abaixo, mas veremos serviços com mais calma na unidade futura.

UsuarioModel é importado de '../model/usuarioModel', representando o modelo de dados de um usuário.

Router e ActivatedRoute são importados de '@angular/router', usados para manipular navegação e obter parâmetros da rota.

MatFormFieldModule é importado de '@angular/material/form-field', usado para estilizar campos de formulário.

IUsuario é importado de '../interfaces/IUsuario', representando uma interface para um usuário, ou seja, importa a definição do que é um usuário, este é um dos usos de interface.

```
export interface IUsuario {  
  id: string;  
  firstName: string;  
  lastName: string;  
  age: BigInteger;  
}
```

MatTableDataSource é importado de '@angular/material/table', usado para fornecer dados a uma tabela do Angular Material, que veremos logo abaixo em 3.2.

Classe UsuarioComponent:

Implementa a interface OnInit, o que significa que o método ngOnInit() será executado quando o componente for inicializado.

Declara uma propriedade model do tipo UsuarioModel e a inicializa com uma nova instância de UsuarioModel.

No construtor, são injetados os serviços UsuarioService, Router e ActivatedRoute.

No método ngOnInit(), é usado o ActivatedRoute para obter o parâmetro 'id' da rota e chama o método getId() passando o valor do parâmetro.

O método getId(id: string) verifica se o 'id' é igual a 'new' e, se não for, chama o método GetById(id) do serviço UsuarioService para obter os detalhes do usuário correspondente. O resultado é atribuído à propriedade model. Veremos na unidade 6 o detalhamento.



O método `save()` chama o método `post(model)` do serviço `UsuarioService` para salvar o usuário. Se a operação for bem-sucedida (`result.success`), a rota é redirecionada para `'/usuarios'`. Veremos na unidade 6 o detalhamento.

Depois de criar o módulo, você pode utilizá-lo em outros lugares do seu projeto. Para fazer isso, importe o módulo em um outro módulo ou componente que deseja usar os recursos fornecidos por ele. Por exemplo:

Os módulos são uma parte fundamental do sistema de modularidade do Angular, permitindo uma organização clara e modular do código da aplicação. Eles ajudam a evitar conflitos de nomes, facilitam a reutilização de componentes e fornecem um controle mais granular sobre as dependências.

3.2 Exemplo...Angular Material

Angular Material é uma biblioteca de componentes para o desenvolvimento de interfaces de usuário no Angular. Ela é construída com base no Material Design, uma diretriz de *design* criada pelo Google, que enfatiza uma aparência moderna e intuitiva, além de fornecer uma experiência consistente em diferentes dispositivos. A biblioteca oferece uma ampla variedade de componentes prontos para uso, como botões, menus, listas, caixas de diálogo, tabelas, barras de navegação, entre outros. Esses componentes são facilmente personalizáveis e adaptáveis aos requisitos específicos de cada projeto.

Saiba mais

Além dos componentes visuais, o Angular Material também fornece diretivas e serviços úteis. A diretiva `MatIcon`, por exemplo, permite exibir ícones em sua aplicação com facilidade, enquanto o serviço `MatSnackBar` permite exibir mensagens de notificação em estilo de toast.

A biblioteca também possui recursos avançados, como suporte a temas e estilos flexíveis. Os temas do Angular Material permitem alterar facilmente a aparência da aplicação, escolhendo entre os temas predefinidos ou criando temas personalizados. Já o sistema de layout flexível ajuda a criar *layouts* responsivos e adaptáveis, facilitando a construção de interfaces que se ajustam automaticamente em diferentes tamanhos de tela.



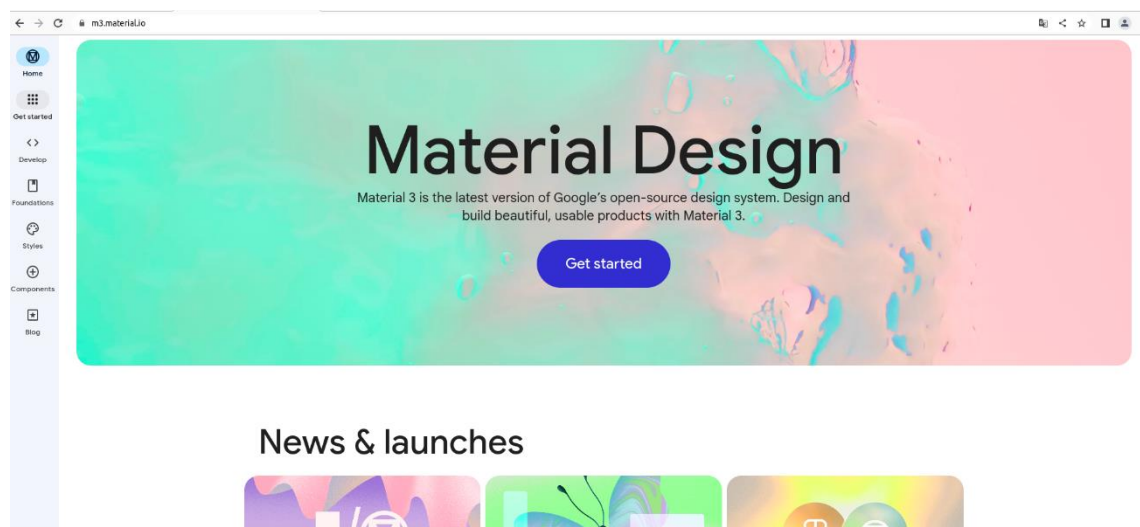
O Material Design é uma linguagem de *design* desenvolvida pelo Google. Ele é usado para criar interfaces consistentes e visualmente atraentes em diferentes plataformas, como Android, iOS e web. O Material Design possui diretrizes específicas para *layout*, cores, tipografia, animações e interações, promovendo uma experiência de usuário intuitiva e coesa. Ele enfatiza o uso de elementos de *design* em camadas, sombras, animações sutis e *feedback* tátil para criar uma aparência moderna e responsiva (Mew,2015).

Uma grande vantagem do Angular Material é a integração perfeita com o Angular. Os componentes são projetados especificamente para funcionar bem com o ciclo de vida do Angular e suportam recursos avançados, como *lazy loading* e suporte a internacionalização. Além disso, a documentação abrangente e os exemplos fornecidos facilitam o aprendizado e a utilização eficaz da biblioteca.

Em resumo, o Angular Material é uma poderosa biblioteca que oferece uma ampla gama de componentes e recursos para facilitar o desenvolvimento de interfaces de usuário bonitas e intuitivas no Angular. Ao utilizar o Angular Material, os desenvolvedores podem acelerar o processo de criação de interfaces, garantindo uma experiência consistente e de alta qualidade para os usuários finais.

ng add @angular/material

Figura 3 – Material Design



Fonte: Material Design, S.d.



É importante entender que diversos componentes externos podem ser integrados ao seu sistema para aperfeiçoá-lo, para trazer melhores resultados, no nosso caso vai ajudar a entender integrações e importações de componentes externos. Temos por exemplo toolbar, ícones e vários outros componentes.

```
import { Component, OnInit } from '@angular/core';
import { UsuarioService } from '../../services/usuario.service'
import { UsuarioModel } from '../../model/usuarioModel'
import { MatFormFieldModule } from '@angular/material/form-field';
import { IUsuario } from '../../interfaces/IUsuario'
import { MatTableDataSource } from '@angular/material/table';
import { Router, ActivatedRoute } from '@angular/router';
```

`MatFormFieldModule` é importado de '@angular/material/form-field', usado para estilizar campos de formulário.

`MatTableDataSource` é importado de '@angular/material/table', usado para fornecer dados a uma tabela do Angular Material.

E temos vários outros componentes...

Precisamos utilizar o AngularMaterial, NÃO! Mas como mencionamos acima ele é muito aderente ao UX e UI e é uma ótima oportunidade de exercitar este tipo de uso.

TEMA 4 – *LIFECYCLY HOOKS* – GANCHOS OU GATILHOS DO CICLO DE VIDA DOS COMPONENTES

Os *Lifecycle Hooks* são métodos especiais fornecidos pelo Angular que permitem que os componentes respondam a eventos específicos durante seu ciclo de vida. Eles são usados para executar ações em momentos chave do ciclo de vida do componente, como criação, atualização e destruição. Esses ganchos permitem que os componentes reajam a diferentes estágios do ciclo de vida e realizem ações apropriadas. Ao usar os ganchos corretamente, é possível criar componentes eficientes, controlar o fluxo de dados e realizar tarefas específicas em momentos específicos do ciclo de vida do componente.



4.1 Lifecycle Hooks

Lifecycle Hooks (ou *ganchos de ciclo de vida*, em tradução livre) são métodos que você pode definir em um componente Angular para serem executados em momentos específicos do ciclo de vida do componente. Esses momentos incluem:

- **ngOnChanges**: este método é chamado quando uma ou mais propriedades vinculadas a um componente mudam. Ele recebe um objeto que contém as mudanças detectadas;
- **ngOnInit**: este método é chamado quando um componente é inicializado. Ele é executado depois que todas as propriedades vinculadas são definidas e depois que o construtor do componente é executado;
- **ngDoCheck**: este método é chamado durante cada detecção de mudança do Angular. Ele permite que você execute ações personalizadas para verificar se o estado do componente mudou;
- **ngAfterContentInit**: este método é chamado depois que o Angular inicializa o conteúdo de um componente que usa a diretiva **ng-content**;
- **ngAfterContentChecked**: este método é chamado depois que o Angular verifica o conteúdo de um componente que usa a diretiva **ng-content**;
- **ngAfterViewInit**: este método é chamado depois que o Angular inicializa a exibição de um componente e suas visualizações filhas;
- **ngAfterViewChecked**: este método é chamado depois que o Angular verifica a exibição de um componente e suas visualizações filhas;
- **ngOnDestroy**: este método é chamado quando um componente é destruído. É usado para limpar recursos, como eventos ou assinaturas de observáveis.

Você pode usar os ganchos de ciclo de vida para executar ações personalizadas em momentos específicos do ciclo de vida do componente, como inicializar dados, atualizar a exibição ou limpar recursos. Para usar os ganchos de ciclo de vida, basta definir um método com o nome do gancho na classe do componente. O Angular irá automaticamente chamar esses métodos quando necessário durante o ciclo de vida do componente (Angular, 2023).

Saiba mais

Normalmente temos de reconfigurar o projeto em Node, motivo: é muito grande, existe um diretório chamado `node_modules` que não deve ser copiado e



transferido pois é muito grande, acostume-se a excluir este diretório quando envia seus fontes, inclusive configure o git (.gitignore) para não considerar este diretório. É normal o projeto que tem 20 Mb aumentar para 200 ou 300 com este diretório que são os fontes node necessários para seu projeto.

Como fazer? Excluir o diretório e recriar ele quando necessário usando:

npm -i

Agora veremos alguns exemplos de lifecycle hooks na unidade 4.2.

4.2 Aprendendo Lifecycle Hooks com exemplos

Os *Lifecycle Hooks* (*ganchos do ciclo de vida*) são métodos especiais fornecidos pelo Angular que são executados em momentos específicos durante o ciclo de vida de um componente. Eles permitem que você realize ações em etapas específicas, como a inicialização do componente, alterações nos valores das propriedades ou a destruição do componente (Angular, 2023).

Saiba mais

São vários exemplos que existem no git:

Aula 3, lifecycle-hooks, adaptados e traduzidos da seguinte página:

COMPONENT Lifecycle. **Angular**, S.d. Disponível em: <<https://angular.io/guide/lifecycle-hooks>>. Acesso em: 21 ago. 2023.

Em nosso código já utilizamos o `onInit`, mostrado acima e é chamado ao iniciar um bloco de código, veja abaixo o `onInit`, para trazer um registro.

```
ngOnInit() {  
  this.active.params.subscribe(p => this.getId(p['id']));  
}
```

Outro uso interessante que faremos é do `ngOnChange`, chamado quando as propriedades de entrada do componente são alteradas.

```
ngOnChanges(changes: SimpleChanges) {  
  // Responder a alterações nas propriedades de entrada  
}
```



TEMA 5 – COMUNICAÇÃO ENTRE COMPONENTES

Comunicação entre componentes: binding e outros; estilos em componentes (CSS e outros); e HTML injection com ng-content

5.1 HTML Injection

HTML injection, também conhecido como Injeção de HTML, é uma técnica em que conteúdo HTML é inserido dinamicamente em um componente ou elemento específico. No Angular, a diretiva **ng-content** é comumente usada para realizar a injeção de conteúdo HTML.

A diretiva **ng-content** permite criar pontos de inserção em um componente, onde o conteúdo HTML externo pode ser incluído. É útil quando você deseja que o conteúdo dentro do componente seja definido fora dele.

Aqui está um exemplo básico de como usar o **ng-content**:

```
@Component({
  selector: 'app-meucomponente',
  template: `
    <div>
    <h2> Título do componente </h2>
    <ng-content> </ng-content>
    </div>`
})
export class MeucomponenteComponent {
```

Use o componente e insira conteúdo HTML dentro dele:

```
<app-meucomponente>
  <p> Conteúdo inserido no componente </p>
</app-meu-componente>
```

No exemplo acima, o conteúdo **<p>Conteúdo inserido no componente</p>** é injetado no ponto de inserção definido por **<ng-content></ng-content>** no *template* do componente **MeuComponenteComponent**. Assim, o resultado renderizado será:

```
<div>
<h2> Título do Componente</h2>
```




```
<p> Conteúdo inserido no componente</p>
</div>
```

5.2 Binding no Angular

O *binding* no Angular é um recurso que permite a comunicação e sincronização de dados entre o componente e o *template*. Ele oferece várias maneiras de conectar e atualizar informações nos dois lados, facilitando a interatividade e a atualização dinâmica da interface do usuário.

Existem quatro tipos principais de *binding* no Angular:

1. Interpolação: Utiliza chaves duplas `{{ }}` para exibir valores do componente no *template*;
2. *Binding* de propriedade: Atribui valores de propriedades do componente a atributos de elementos HTML;
3. *Binding* de evento: Permite tratar eventos gerados por elementos HTML, como cliques ou mudanças de valor, no componente;
4. *Binding* bidirecional: Combina o *binding* de propriedade e de evento para permitir a atualização bidirecional dos dados entre o componente e o *template*.

Essas técnicas de *binding* facilitam a criação de interfaces interativas, permitindo que os componentes e o *template* se comuniquem de forma eficiente. Ao utilizar o *binding*, é possível manter a interface sincronizada com o estado do componente, garantindo uma experiência de usuário dinâmica e responsiva.

Ou seja, o *binding* no Angular é um mecanismo que permite a comunicação e sincronização de dados entre o componente e o *template*. Existem diferentes tipos de *binding* no Angular:

1. Interpolação (*Interpolation*): Utiliza chaves duplas `{{ }}` para exibir valores do componente no *template*.

```
<p> Meu nome é {{ nome }}</p>
```

2. *Property binding* (*binding* de propriedade): Permite atribuir valores de propriedades do componente a atributos de elementos HTML.

```
<button [disable] = "isDesabilitado"> Clique Aqui </button>
```



3. *Event binding (binding de evento)*: Permite que os eventos gerados pelos elementos HTML, como cliques ou mudanças de valor, sejam tratados no componente.

```
<button (click) = "onClick()"> Clique Aqui </button>
```

4. *Two-way binding (binding bidirecional)*: Combina o *binding* de propriedade e de evento para permitir a atualização bidirecional dos dados entre o componente e o *template*.

```
<input [(ngModel)]="nome">
```

5. Essas são algumas formas de *binding* no Angular, e elas ajudam a manter o estado dos componentes e do *template* sincronizados, garantindo uma experiência interativa e dinâmica para o usuário.

5.3 Estilos

No Angular, existem diferentes abordagens e estilos para a criação de componentes. Aqui estão alguns dos estilos mais comuns:

- **Componentes separados**: cada componente Angular é definido em seu próprio arquivo, contendo o código TypeScript, o template HTML e os estilos CSS ou SCSS associados. Essa é a abordagem mais comum e recomendada para organizar e reutilizar componentes;
- **Estilos embutidos**: você pode definir estilos diretamente no arquivo do componente usando o atributo **styles** ou **styleUrls** na anotação **@Component**. Esses estilos serão aplicados apenas ao componente específico;
- **Estilos globais**: você pode adicionar estilos globais para toda a aplicação no arquivo **styles.css** ou em um arquivo separado, que é importado no arquivo principal do aplicativo;
- **Pré-processadores CSS**: o Angular oferece suporte a pré-processadores CSS, como SCSS ou LESS. Eles permitem recursos avançados, como variáveis, mixins e aninhamento, para facilitar a estilização dos componentes;
- **Frameworks de estilos**: é comum utilizar frameworks de estilos, como Bootstrap, Material Design ou Tailwind CSS, em conjunto com o Angular.



Esses *frameworks* fornecem estilos prEdefinidos e componentes reutilizáveis que podem ser facilmente integrados aos componentes Angular.

Lembrando que a escolha do estilo de componente depende das necessidades e preferências do projeto. O importante é manter uma abordagem consistente e organizada para facilitar a manutenção e a escalabilidade do aplicativo Angular.

NA PRÁTICA

Aula prática 3 – Dentro do projeto criado na aula prática 2, mostrar exemplos de comunicação entre componentes pais e filhos (incluindo *bindings* e *ng-content*) e de aplicação de estilos CSS em componentes e globalmente. Todos os exemplos deverão compor partes reais do projeto da disciplina.

FINALIZANDO

Vimos que, no Angular, os componentes são elementos reutilizáveis que encapsulam funcionalidades e permitem criar interfaces de usuário interativas. Os Hooks são uma característica introduzida no Angular que permite compartilhar lógica entre componentes sem a necessidade de hierarquias complexas ou herança de classes. O Material Design é uma diretriz de design desenvolvida pelo Google, que se concentra em criar interfaces de usuário modernas e intuitivas. No Angular, o Material Design pode ser implementado usando a biblioteca Angular Material.

Em conjunto podemos criar vários componentes poderosos para nossos projetos.



REFERÊNCIAS

ANGULAR. Disponível em: <<https://www.angular.io>>. Acesso em: 21 ago. 2023.

TYPESCRIPT. Disponível em: <<https://www.typescriptlang.org/>>. Acesso em: 21 ago. 2023.