

Treinamento Angular First Class –
Material Teórico/Prático

Sumário

O framework Angular	3
A arquitetura do framework Angular	3
Aplicações Angular	5
Visão geral	5
Angular - Configuração do ambiente	6
CLI angular	9
Configuração do Projeto	10
Projeto angular-alpha.....	10
Criando o projeto	11
Pasta App.....	14
Data Binding	19
O que é Data Binding?.....	19
Necessidade do Data Binding.....	20
Interpolação (Interpolation).....	21
Ligação/Vinculação de dados baseada em propriedades (Property-Based Binding)	23
Vinculação/ligação por evento (Event Binding)	24
Vinculação/ligação de dados bidirecional (Two-Way Data Binding)	27
Input property	30
Output Property	33
Directive	38
Structural Directives.....	38
ngIf.....	38
ngIf else	40
ngIf then else.....	40
ngFor	44
Attribute Directives	48
ngStyle.....	51
Directive Component	53
Pipes	56
Pipe personalizado	60
Forms.....	62
Template Driven Form	62
Model Driven Form	65

Validação de formulário	68
Services.....	73
O que é um service.....	73
Para que services são utilizados.....	74
Vantagens ao se usar services.....	74
Como criar um service no Angular	74
Angular Dependency Injection	79
O que é dependência	79
Angular Dependency Injection -definição	79
Estrutura da Angular Dependency Injection	82
Como funciona a injeção de dependência no Angular	83
Como usar a dependency Injection (injeção de dependência)	83
Injetando Service em Service	86
Referências.....	91

O framework Angular

Angular é um *framework* para desenvolver aplicações em diversas plataformas, mantido e desenvolvido pela **Google**.

Ele é uma reescritura completa do antigo *angularjs* e foi escrito em **TypeScript**.

Ele vem com um **conjunto de bibliotecas** poderosas que podemos importar, possibilitando construir aplicações com uma qualidade e produtividade surpreendente.

A arquitetura do framework Angular

A arquitetura do **Angular** permite organizar a aplicação por módulos através dos *NgModules*, que fornecem um contexto para os componentes serem compilados.

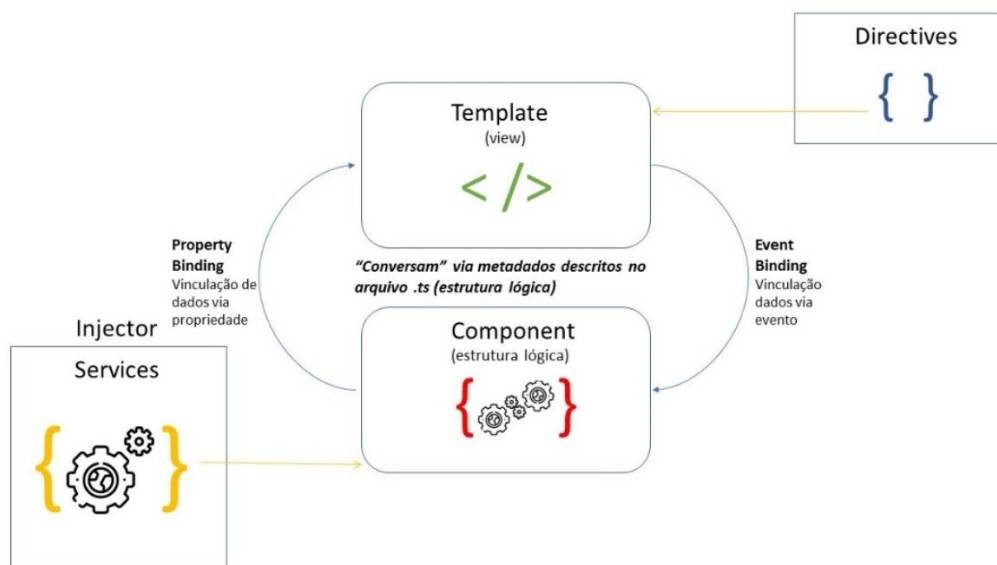
Uma aplicação sempre tem ao menos um **módulo raiz (app module)** que habilita a inicialização e, normalmente, possui outros módulos de bibliotecas.

Os componentes deliberam as visualizações — que são conglomerados de elementos e funcionalidades de tela — que o Angular modifica de acordo com a lógica e os dados da aplicação.

Esses componentes usam serviços que fornecem funcionalidades específicas e que são indiretamente relacionadas a essas visualizações.

Os **service providers** podem ser injetados nos componentes como dependências, tornando seu código modular e reutilizável.

Services e *components* são simples classes com *decorators*, que definem o tipo da estrutura lógica implementada e fornecem metadados para informar o Angular como usá-los. Observe o diagrama abaixo:



A instrução @NgModules

Tem como objetivo declarar e agrupar tudo que criamos no Angular. Existem duas estruturas principais, que são: *declarations* e o *providers*.

Declarations é onde declaramos os itens que iremos utilizar nesse módulo, como por exemplo componentes e diretivas, já nos **Providers** informamos os serviços.

```
@NgModule({
  declarations: [ AppComponent ],
  providers: [ AuthClientService ],
})
```

Assim como módulos **JavaScript**, o *NgModules* também pode importar funcionalidades de outros *NgModules* e permitir que suas próprias funcionalidades também sejam exportadas.

Um exemplo claro disso é que para usar o serviço de roteador no seu *app* basta importar o *RouterModule*.

```
@NgModule({
  declarations: [ AppComponent ],
  imports: [ AppRoutingModule ],
})
```

Componentes

A maior parte do desenvolvimento quando se utiliza o *framework Angular* é feito nos componentes.

Cada componente define uma classe, que contém dados e lógicas do aplicativo e está sempre associada a um *template HTML*, onde são definidas as visualizações deste componente.

O decorator *@Component()* identifica a classe imediatamente como um componente e oferece o modelo e os metadados específicos dele.

Os metadados configuram, por exemplo, como o componente pode ser referenciado no **HTML** e também quais os serviços devem ser utilizados.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [ HeroService ]
})
```

```
})
```

Directives

As diretivas são como marcadores no elemento DOM que comunicam ao **Angular** para incluir um comportamento específico.

Existem três tipos de diretivas no Angular, que são: Diretivas de atributos, Diretivas estruturais e Componentes.

Diretivas de atributos: Alteram a aparência ou o comportamento de um elemento, componente ou outra diretiva, como por exemplo, *NgClass* e *NgStyle*.

Diretivas estruturais: Modificam o *layout* adicionando ou removendo elementos do DOM, como por exemplo, *NgIf* e *NgFor*.

Componentes: São diretivas com um modelo.

Aplicações Angular

Como mencionado anteriormente, o Angular é dos frameworks mais usados na Web. Vou listar alguns deles aqui:

- **Comunidade apoiada pelo Google** - O Google apoia ativamente o Angular e seu desenvolvimento. Angular é usado em vários Google Apps.
- **Desenvolvimento baseado em POJO** - o POJO Angular (Plain Old JavaScript Object) é muito utilizado e ajuda a aprender o Angular de uma maneira mais fácil.
- **Interface do usuário declarativa** - Angular usa HTML como linguagem de exibição e amplia sua funcionalidade. Ajuda no tratamento da diferenciação da interface do usuário versus o código, e a interface do usuário é pouco acoplada ao código.
- **TypeScript** - O TypeScript é um super conjunto de javascript e é fácil de depurar. É altamente seguro e é orientado a objetos.
- **Estrutura modular** - O desenvolvimento angular é altamente modular, é baseado em componentes e é de fácil manutenção.
- **Suporte multiplataforma** - O código angular funciona bem em todas as plataformas sem muita alteração no código

Visão geral

Angular é propriedade do Google e a versão estável foi feita em setembro/outubro de 2010. A última versão lançada é o Angular 11.

Abaixo está a lista de versões angulares lançadas até agora -

Versão	Data de lançamento
JS angular	Outubro de 2010

Angular 2.0	Set 2016
Angular 4.0	Março de 2017
Angular 5.0	Novembro 2017
Angular 6.0	Maio 2018
Angular 7.0	Outubro 2018

As datas de lançamento das próximas duas grandes versões futuras do Angular são fornecidas abaixo:

Versão	Data de lançamento
Angular 8.0	Março / abril de 2019
Angular 9.0	Fevereiro 2020
Angular 10.0	Junho de 2020
Angular 11.0	Novembro 2020
Angular 12.*	Maio de 2021 – versão beta

O Google planeja lançar a principal versão angular a cada 6 meses. A versão lançada até o momento é compatível com versões anteriores e pode ser atualizada para a mais recente com muita facilidade.

Angular - Configuração do ambiente

Neste passo, discutiremos a configuração do ambiente necessária para o Angular. Para instalar o Angular, é necessário ter instalados em seu computador:

- Nodejs
- Npm
- CLI angular
- IDE para escrever seu código

Nodejs

Para verificar se o nodejs está instalado no seu sistema, digite **node -v** no terminal. Isso ajudará você a ver a versão do nodejs atualmente instalada no seu sistema.

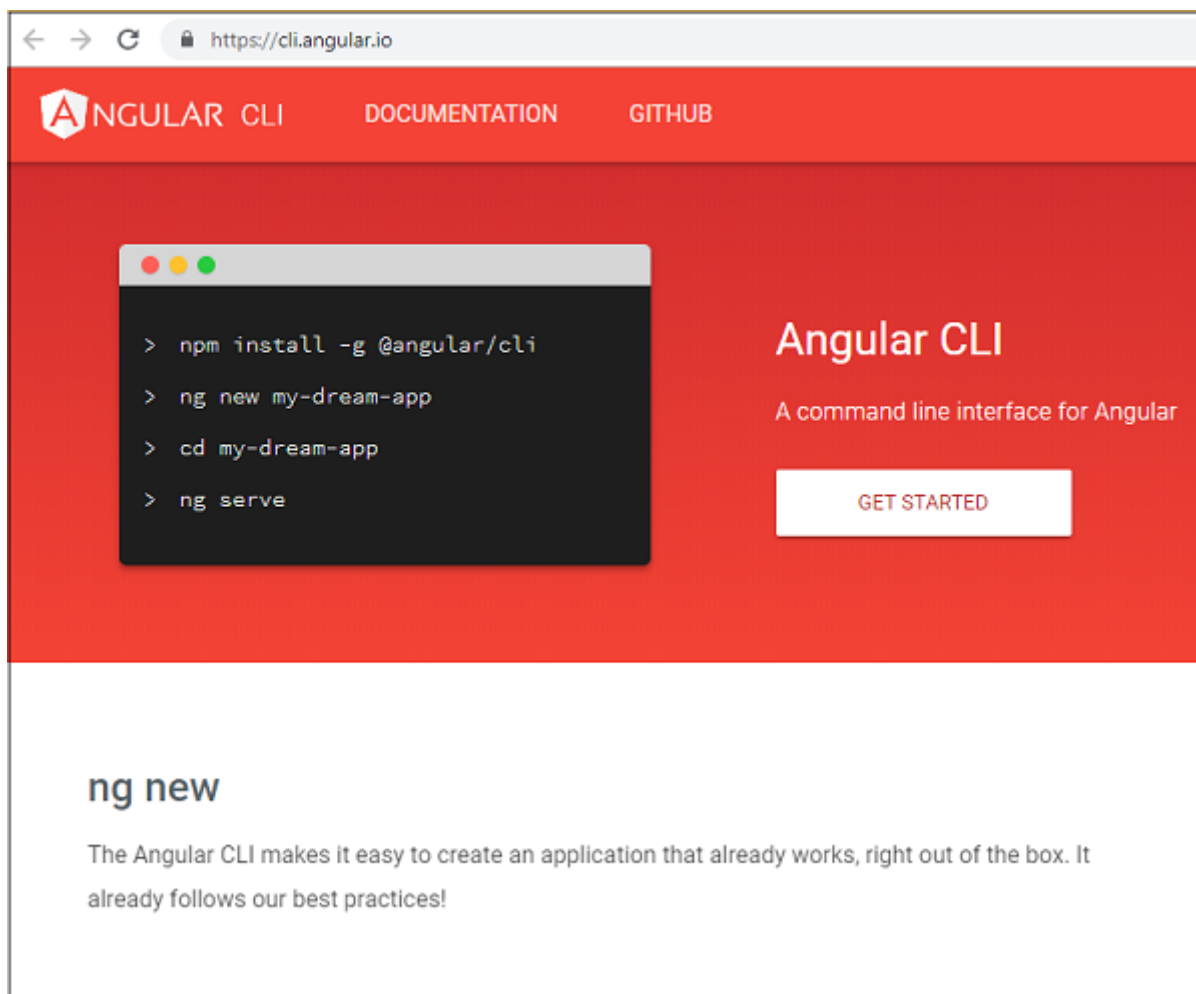
Nodejs deve ser maior que 8.x ou 10.x, e npm deve ser maior que 5.6 ou 6.4.

```
C:\>node -v  
v10.15.1
```

Com base no seu sistema operacional, instale o pacote necessário. Depois que o nodejs estiver instalado, o npm também será instalado junto com ele. Para verificar se o npm está instalado ou não, digite **npm -v** no terminal, conforme indicado abaixo. Ele exibirá a versão do npm.

```
C:\>npm -v  
6.4.1
```

As instalações do Angular são muito simples com a ajuda da CLI angular. Visite a página inicial <https://cli.angular.io/> do angular para obter a referência do comando.



Digite **npm install -g @ angular / cli** no prompt de comando para instalar o angular cli no seu sistema. Vai demorar um pouco para instalar e, uma vez feito, você pode verificar a versão usando o comando abaixo:

```
npm install -g @angular/cli //comando instalar o
Angular
ng new nome_do_projeto // nome do projeto
cd nome-da-pasta
ng serve
```

Os comandos acima ajudam a obter a configuração do projeto no Angular.

Vamos criar uma pasta chamada **Project-Angular** e instalar **angular/cli** como mostrado abaixo

CA: Administrador: Prompt de Comando

```
C:\WINDOWS\system32>npm install -g @angular/cli
```

Depois que a instalação estiver concluída, verifique os detalhes dos pacotes instalados usando o comando `ng version`, como mostrado abaixo:

```

Angular CLI
Angular CLI: 12.0.0
Node: 12.16.1
Package Manager: npm 6.13.4
OS: win32 x64

Angular:
...

Package                                Version
-----
@angular-devkit/architect              0.1200.0 (cli-only)
@angular-devkit/core                   12.0.0 (cli-only)
@angular-devkit/schematics             12.0.0 (cli-only)
@schematics/angular                   12.0.0 (cli-only)

```

Ele fornece a versão para o Angular CLI e as versões dos pacotes disponíveis para o Angular.

Terminamos a instalação do Angular, agora começaremos com a configuração do projeto.

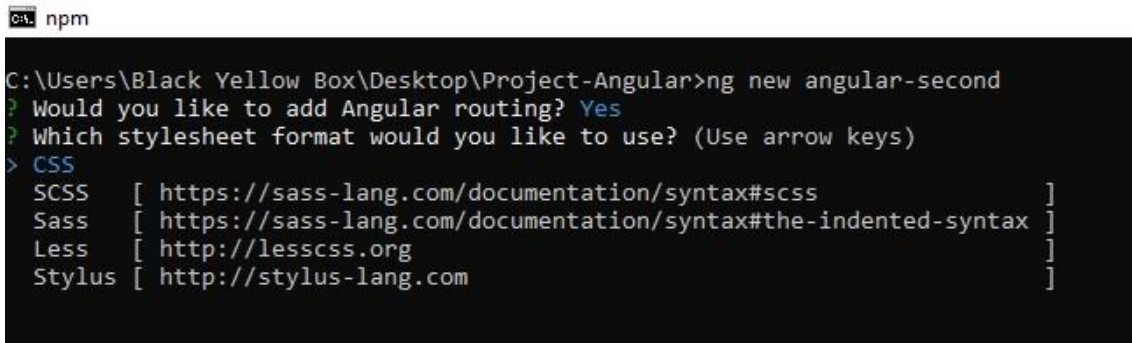
CLI angular

Ao fazer a configuração do projeto usando a CLI angular, ele pergunta sobre os recursos internos disponíveis, como o roteamento e o suporte à folha de estilo, conforme mostrado abaixo:

CA: npm

```
C:\Users\Black Yellow Box\Desktop\Project-Angular>ng new angular-second
? Would you like to add Angular routing? (y/N)
```

Escolhendo o estilo:



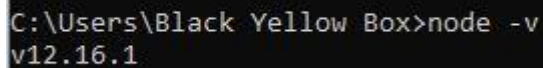
```

C:\Users\Black Yellow Box\Desktop\Project-Angular>ng new angular-second
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
SCSS [ https://sass-lang.com/documentation/syntax#scss ]
Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less [ http://lesscss.org ]
Stylus [ http://stylus-lang.com ]
  
```

Configuração do Projeto

Neste passo, abordaremos a configuração do projeto Angular.

Para começar a configuração do projeto, verifique se você possui o nodejs instalado. **Você pode verificar a versão do node na linha de comandos usando o comando node -v**, como mostrado abaixo:



```

C:\Users\Black Yellow Box>node -v
v12.16.1
  
```

Projeto angular-alpha

Vamos criar uma pasta na área de trabalho. Essa pasta abrigará nossos projetos angular. Para criar uma pasta na área de trabalho – via prompt de comando – basta inserir no prompt as seguintes instruções:



```

C:\WINDOWS\system32>cd C:\
C:\>cd C:\Users\Black Yellow Box\Desktop
C:\Users\Black Yellow Box\Desktop>mkdir Project-Angular
C:\Users\Black Yellow Box\Desktop>
  
```

A imagem acima mostra que: devemos encontrar e inserir no prompt o path (caminho) completo do diretório “Área de trabalho (Desktop)”. Na sequência, inserimos o comando mkdir e o nome da nossa pasta (Project-Angular). Feito isso, basta clicar Enter em seu teclado. Sua pasta para os projetos Angular está criada.

Criando o projeto

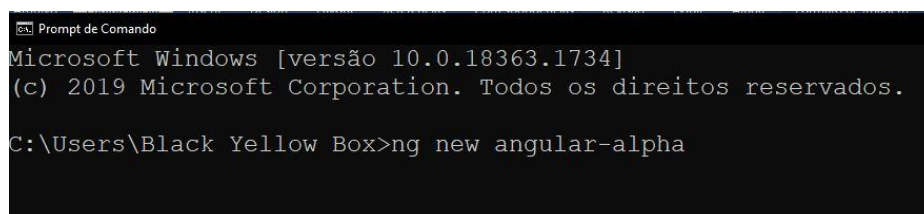
Abra a pasta que você acabou de criar. Basta inserir ainda no Prompt de comando a instrução: *cd Project-Angular*.

Para criar um projeto no Angular, usaremos o seguinte comando - a partir do prompt de comando:

```
ng new nomedoprojeto
```

Você pode usar o *nomedoprojeto* de sua escolha. Vamos agora executar o comando acima através do prompt.

Aqui, usamos o *nome do projeto como angular-alpha*. Depois de executar o comando, ele perguntará sobre o roteamento, como mostrado abaixo:



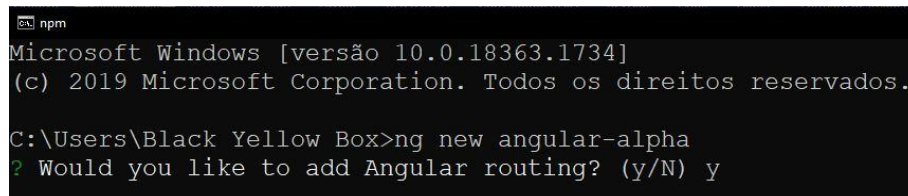
```

Microsoft Windows [versão 10.0.18363.1734]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Black Yellow Box>ng new angular-alpha
  
```

Digite y para adicionar roteamento à configuração do seu projeto.

A próxima pergunta é sobre a folha de estilo:

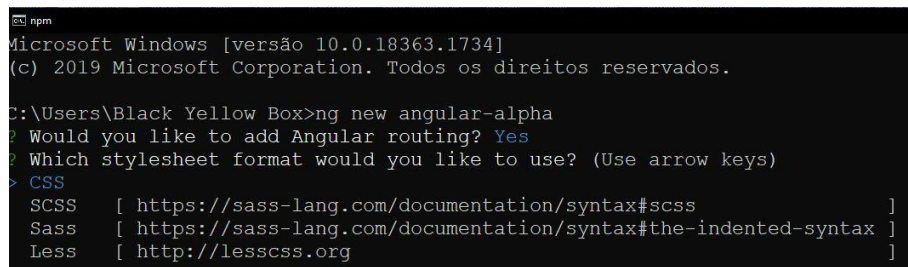


```

Microsoft Windows [versão 10.0.18363.1734]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Black Yellow Box>ng new angular-alpha
? Would you like to add Angular routing? (y/N) y
  
```

As opções disponíveis são CSS, Sass, Less e Stylus. Na captura de tela acima, a seta está no CSS. Para alterar, você pode usar as teclas de seta para selecionar a necessária para a configuração do seu projeto. No momento, discutiremos o CSS para a configuração do projeto.



```

Microsoft Windows [versão 10.0.18363.1734]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Black Yellow Box>ng new angular-alpha
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
  SCSS [ https://sass-lang.com/documentation/syntax#scss ]
  Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less [ http://lesscss.org ]
  
```

O projeto *angular-alpha* foi criado com sucesso. Ele instala todos os pacotes necessários para que nosso projeto seja executado no Angular. Vamos agora entrarna pasta do projeto criado, que está no diretório **angular-alpha**.

Altere o diretório na linha de comando usando a linha de código fornecida:

```
cd angular-alpha
```

Visual Studio Code

Este é um IDE de código aberto do Visual Studio. Está disponível para plataformas Mac OS X, Linux e Windows. O VScode está disponível em - <https://code.visualstudio.com/>

Instalação no Windows - se ainda não estive instalado em seu ambiente, basta seguir as mesmas etapas da instalação anterior (para typescript).

Não iniciamos nenhum projeto nele. O que precisamos fazer, agora, é trazer o projeto que criamos - usando o angular-cli e seu respectivo comando para criação – para dentro da nossa IDE Visual Studio Code. O procedimento para executarmos essa tarefa é o seguinte:

Clique em File > ao clicar em File no menu de tarefas no topo da IDE se abrirá uma janela de contexto: ao abri a janela selecione Open Folder. Na sequencia, encontre o local onde seu projeto Angular foi gerado; clique na pasta que contem seu projeto e, dessa forma ele será importado para dentro da IDE.

Vamos abrir o **angular-alpha** e ver como é a estrutura da pasta.

Agora que temos a estrutura de arquivos do nosso projeto, vamos compilar nosso projeto com o seguinte comando:

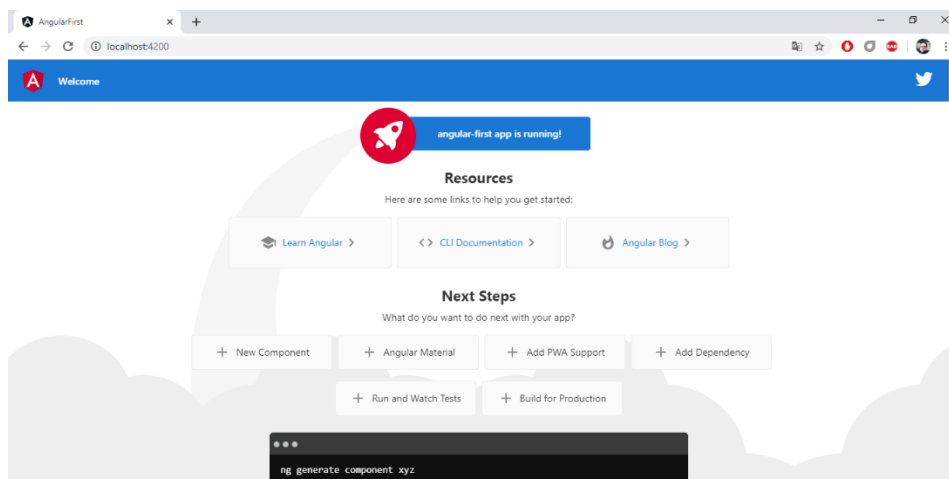
```
ng serve
```

O comando ng serve cria o aplicativo e inicia o servidor web:

Você verá o abaixo quando o comando começar a executar:

O servidor da web inicia na porta 4200. Digite o URL "**http://localhost: 4200/**" no navegador e veja a saída. Depois que o projeto for compilado, você receberá a seguinte saída:

Depois de executar o URL, **http://localhost: 4200/** no navegador, você será direcionado para a seguinte tela:



Vamos concluir a configuração do projeto. Se você vir que usamos a porta 4200, que é a porta padrão usada pelo *angular – cli* durante a compilação. Você pode alterar a porta, se desejar, usando o seguinte comando:

```
ng serve --host 0.0.0.0 -port 4205 por exemplo
```

A pasta *angular-alpha* / tem a seguinte **estrutura de pastas**:

- **node_modules** / - O pacote npm instalado é *node_modules*. Você pode abrir a pasta e ver os pacotes disponíveis.
- **src** / - Esta pasta é onde iremos trabalhar no projeto usando o Angular. Especialmente usando os arquivos localizados dentro da pasta *src / app* que foi criada durante a instalação do projeto e contém todos os arquivos necessários para trabalharmos no *angular-alpha*.

A pasta *angular-alpha* contém alguns arquivos em sua estrutura:

- **editorconfig** - Este é o arquivo de configuração do editor.
- **.gitignore** - Um arquivo *.gitignore* deve ser confirmado no repositório, para compartilhar as regras de ignorar com outros usuários que clonam o repositório.
- **angular.json** - basicamente contém o nome do projeto, a versão do cli etc.
- **package.json** - O arquivo *package.json* informa quais bibliotecas serão instaladas no *node_modules* quando você executar o *npm install*. Caso você precise adicionar mais bibliotecas, adicione-as no arquivo *package.json* e execute o comando *npm install*.
- **tsconfig.json** - basicamente contém as opções do compilador necessárias durante a compilação.
- **tslint.json** - Este é o arquivo de configuração com regras a serem consideradas durante a compilação.

A pasta **src** / é a pasta principal, que possui internamente uma estrutura de arquivos diferente.

Pasta App

Ela contém os arquivos descritos abaixo. Esses arquivos são instalados pelo angular-cli por padrão:

app-routing.module.ts

Este arquivo tratará do roteamento necessário para o seu projeto. Ele está conectado ao módulo principal, ou seja, *app.module.ts*.

Observe a estrutura do arquivo, abaixo:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

app.component.html

O código html que será exibido em tela, geralmente, estará disponível neste arquivo.

A estrutura principal do código é a seguinte:

```
<div class="toolbar" role="banner">
  <img
    width="40"
    alt="Angular Logo"/>
  <span>Bem-vindo</span>
  <div class="spacer"></div>
</div>

<div class="content" role="main">

  <!-- Highlight Card -->
  <div class="card highlight-card card-small">

    <svg id="rocket" xmlns="http://www.w3.org/2000/svg" width="101.678" height="101.678" viewBox="0 0 101.678 101.678">
      <title>Rocket Ship</title>
      <g id="Group_83" data-name="Group 83" transform="translate(-141 -696)">
```

```

        <circle id="Ellipse_8" data-
name="Ellipse 8" cx="50.839" cy="50.839" r="50.839" transfo
rm="translate(141 696)" fill="#dd0031"/>
        <g id="Group_47" data-
name="Group 47" transform="translate(165.185 720.185)">
            <path id="Path_33" data-
name="Path 33" transform="translate(0.371 3.363)" fill="#f
ff"/>
            <path id="Path_34" data-
name="Path 34" fill="#fff"/>
        </g>
    </g>
</svg>

<span>{{ title }} app is running!</span>

<svg id="rocket-
smoke" xmlns="http://www.w3.org/2000/svg" width="516.119" h
eight="1083.632" viewBox="0 0 516.119 1083.632">
    <title>Rocket Ship Smoke</title>
    <path id="Path_40" fill="#f5f5f5"/>
</svg>

</div>

<svg id="clouds" xmlns="http://www.w3.org/2000/svg" width
="2611.084" height="485.677" viewBox="0 0 2611.084 485.677"
>
    <title>Gray Clouds Background</title>
    <path id="Path_39"/>
</svg>

</div>

```

Este é o código html padrão (aqui, com a exclusão de identificação de imagens svg) atualmente disponível com a criação do projeto.

app.component.spec.ts

Esses são arquivos gerados automaticamente que contêm testes de unidade para o componente de origem.

app.component.ts

A classe (estrutura lógica) para as regras de funcionamento do componente é definida neste arquivo. Você pode fazer o processamento da estrutura html no arquivo `.ts`. O processamento incluirá atividades como conectar-se ao banco de dados, interagir com outros componentes, roteamento, services, etc.

Observe a estrutura de código abaixo:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-alpha';
}
```

app.module.ts

Se você abrir o arquivo, verá que o código faz referência a diferentes bibliotecas importadas. O Angular-cli usou essas bibliotecas padrão para a importação: angular / core.

Os próprios nomes explicam o uso das bibliotecas. Eles são importados e salvos em variáveis como declarações, importações, providers e autoinicialização.

Podemos ver também **app-routing.module**. Isso ocorre porque selecionamos o roteamento no início da instalação. O módulo é adicionado por @ angular / cli.

Observe a estrutura de código abaixo:

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

import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModuleModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

O `@NgModule` é importado de `@ angular/core` e possui um objeto com as seguintes propriedades -

Declarations - nas declarações, a referência aos componentes é armazenada. O componente App é o componente padrão criado sempre que um novo projeto é iniciado. Aprenderemos sobre a criação de novos componentes em uma seção diferente.

Imports - Os módulos serão importados conforme mostrado acima. No momento, o BrowserModule faz parte das importações importadas do @angular / platform-browser. Há também o módulo de roteamento adicionado AppRoutingModuleModule.

Providers - Isso terá referência aos services criados. O service será discutido em um capítulo subsequente.

Bootstrap - Isso tem referência ao componente padrão criado, ou seja, AppComponent.

app.component.css - Você pode escrever o css do compoente neste arquivo. No momento, adicionamos a cor de fundo à div, como mostrado abaixo.

Assets

Você pode salvar suas imagens, arquivos js nesta pasta.

Environment

Esta pasta possui detalhes para a produção ou o ambiente de desenvolvimento. A pasta contém dois arquivos.

- environment.prod.ts
- environment.ts

Ambos os arquivos têm detalhes sobre se o arquivo final deve ser compilado no ambiente de produção ou no ambiente de desenvolvimento.

A estrutura adicional do arquivo *angular-alpha* / folder inclui o seguinte:

favicon.ico

Esse é um arquivo geralmente encontrado no diretório raiz de um site.

index.html

Este é o arquivo que é exibido no navegador.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularFirst</title>
  <base href="/">
  <meta name="viewport" content="width=device-
width, initial-scale=1">
```

```

<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>

```

O corpo tem **<app-root></app-root>**. Esse é o seletor usado no arquivo **app.component.ts** e exibirá os detalhes do arquivo **app.component.html**.

main.ts

main.ts é o arquivo de onde começamos o desenvolvimento do projeto. Começa com a importação do módulo básico que precisamos. No momento, se você ver *angular / core*, *angular / plataforma-dinâmica* do navegador, *app.module* e *ambiente*, é importado por padrão durante a instalação do *angular-cli* e a configuração do projeto.

```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

O *platformBrowserDynamic(). BootstrapModule(AppModule)* possui a referência do módulo pai *AppModule*. Portanto, quando é executado no navegador, o arquivo é chamado *index.html*. *Index.html* refere-se internamente a *main.ts* que chama o módulo pai, ou seja, *AppModule* quando o código a seguir é executado:

```

platformBrowserDynamic().bootstrapModule(AppModule).catch(
  err => console.error(err));

```

Quando o *AppModule* é chamado, ele chama *app.module.ts*, que chama mais o *AppComponent* com base na autoinicialização da seguinte maneira:

```
bootstrap: [AppComponent]
```

polyfill.ts

Isso é usado principalmente para compatibilidade com versões anteriores.

styles.css

Este é o arquivo de estilo necessário para o projeto.

test.ts

Aqui, os casos de teste de unidade para testar o projeto serão tratados.

tsconfig.app.json

Isso é usado durante a compilação, possui os detalhes de configuração que precisam ser usados para executar o aplicativo.

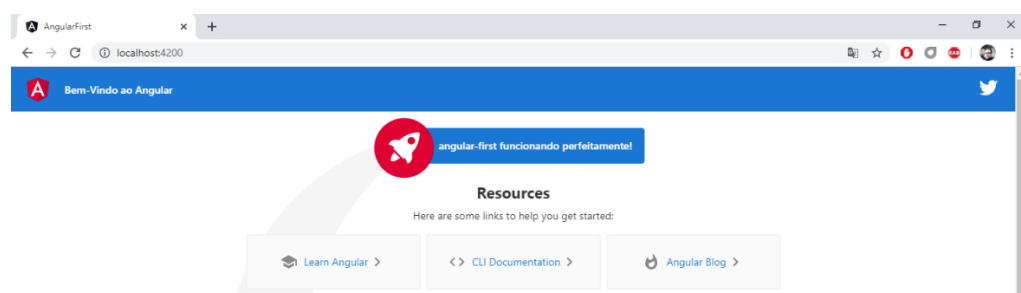
tsconfig.spec.json

Isso ajuda a manter os detalhes para o teste.

A estrutura final do arquivo será a seguinte:

No **app.component.ts**, existe um seletor: **app-root**, usado no arquivo **index.html**. Isso exibirá o conteúdo presente em **app.component.html**.

A imagem abaixo, se correu tudo certo com a criação do projeto, ao executar a instrução `ng serve`, representa a renderização do projeto exibido no browser através do endereço: `http://localhost:4200`



Data Binding

O que é Data Binding?

A ligação de dados (*data binding*) é um dos mais úteis recursos da estrutura Angular. Devido a esse recurso, precisamos escrever menos código

em comparação com qualquer outra biblioteca ou estrutura do *client-side*. Em função do data binding - em um aplicativo Angular - a sincronização automática de dados entre o modelo (model) e os componentes da visualização (view) é automático. No Angular, sempre podemos tratar o modelo (model) como uma única *single-source-of-truth* de dados em nosso aplicativo Web usando *data binding*. Dessa forma, a interface do usuário ou a visualização (view) sempre representa o modelo (model) de dados o tempo todo. Com a ajuda da data binding, podemos estabelecer uma relação entre a interface do usuário do aplicativo e a lógica de negócios. Se estabelecermos a ligação de dados (data binding) de maneira correta e os dados fornecerem as notificações apropriadas para a estrutura, quando o usuário fizer alterações nos dados na visualização, o

Necessidade do Data Binding

A estrutura Angular fornece esse recurso especial e poderoso chamado Data Binding, que traz flexibilidade em qualquer aplicativo Web. Com a ajuda da ligação de dados (data binding), podemos obter melhor controle sobre o processo e as etapas relacionadas ao processo de comunicação entre componentes. Esse processo facilita a vida do desenvolvedor e reduz o tempo de desenvolvimento em relação a outras estruturas. Alguns dos motivos relacionados aos motivos pelos quais a ligação de dados é necessária para qualquer aplicativo baseado na Web estão listados abaixo

1. Com a ajuda da ligação de dados (data binding), as páginas Web baseadas em dados podem ser desenvolvidas de maneira rápida e eficiente.
 2. Sempre obtemos o resultado desejado com poucas linhas de código.
 3. Devido a esse processo, o tempo de execução aumenta. Como resultado, melhora a qualidade do aplicativo.
- Com a ajuda do emissor do evento (event emitter), podemos obter um melhor controle sobre o processo de ligação de dados.

Diferentes Tipos de Ligação de Dados (Data Binding)

No Angular, existem quatro tipos diferentes de processos de ligação de dados disponíveis. São eles:

- Interpolação (Interpolation)
- Ligação de propriedade (Property Binding)
- Ligação bidirecional (Two-Way Binding)
- Ligação através de evento (Event Binding)

Interpolação (Interpolation)

A ligação de dados de interpolação (interpolation) é a maneira mais popular e mais fácil de ligação de dados no Angular. Esse recurso também está disponível nas versões anteriores da estrutura Angular. Na verdade, o **contexto entre chaves** é a expressão do modelo (model) que o Angular avalia primeiro e depois converte em sequências de caracteres (string). A interpolação (interpolation) usa a expressão de chaves, ou seja, `{{}}` para renderizar o valor associado ao componente (component). Pode ser uma sequência estática, valor numérico ou um objeto do seu modelo (model) de dados. No

Angular, usamos assim: `{{firstName}}`. O exemplo abaixo mostra como podemos usar a interpolação no componente para exibir dados no front end.

```
<span>{{ title }} aplicativo angular funcionando!</span>
```

Implementação Interpolation (Interpolação)

Neste passo, vamos implementar cada um dos data binding estudados até este momento. No exemplo abaixo, faremos uso da Interpolação (Interpolation) no Angular para diferentes tipos de dados.

Você pode criar um novo projeto Angular chamado *ExemploInterpolation* – ou utilizar seu projeto *angular-alpha* para a implementação. Edite os arquivos conforme indicado abaixo:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

```
export class AppComponent {
  title = 'angular-alpha';
  public x: number = 10;
  public umArray: Array<number> = [10, 22, 14];
  public dataHoje: Date = new Date();

  public statusBooleano: boolean = true;

  public exibirTexto(): string {
    return 'Texto retornado a partir da função!';
  }
}
```

app.component.html

```
<!-- Interpolation data binding-->
<div>
  <span>O número descrito é: {{ x }}</span>
  <br /><br />
  <span>O array de números é: {{ umArray }}</span>
  <br /><br />
  <span>A data de hoje é:{{ dataHoje }}</span>
  <br /><br />
  <span>O status booleano indicado na variavel é: {{ statusBooleano }}</span>
  <br /><br />
  <span>{{statusBooleano ? "Este é o resultado para o status True" : "Este é para o status False"}}</span>
  <br /><br />
  <span>{{ exibirTexto() }}</span>
</div>
```

Salve o projeto e observe a saída.



angular-first funfanfo lindamente!

O número atribuído a variável x é: 10

O array de números é: 23,56,78

A data de hoje é: Tue Aug 24 2021 09:47:51 GMT-0300 (Horário Padrão de Brasília)

O status booleano indicado é: true

Este é o texto exibido quando o valor é true

Texto retornado a partir da função!

Ligação/Vinculação de dados baseada em propriedades (Property-Based Binding)

No Angular, existe outro mecanismo de ligação, chamado de Vinculação de Propriedade (property binding). Em sua essência, vinculação/ligação de dados baseada em propriedade é exatamente o mesmo que interpolação. É também chamado de ligação unidirecional. Para ligação de propriedade (property binding) é usado [] para enviar os dados do componente para o modelo HTML. A maneira mais comum de usar a associação de propriedades é atribuir qualquer propriedade da tag do elemento HTML ao [] com o valor da propriedade do componente; observe o snippet abaixo:

```
Exibir valor com property-  
binding : <input [value]="umvalor" />
```

Implementação property Binding

Nesta sequência, vamos implementar o data binding - conforme exemplo abaixo – baseado em propriedade (property binding). Você pode criar um novo projeto Angular chamado *ExemploPropertyBinding* – ou utilizar seu projeto *angular-alpha* para a implementação. Edite o arquivo *app.component.html*

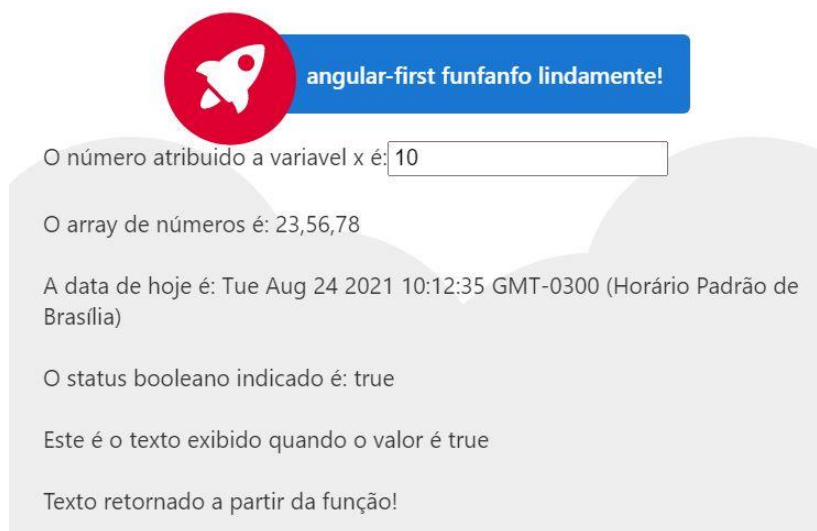
conforme indicado abaixo – o arquivo *app.component.ts* não sofre qualquer alteração, permanece o mesmo:

app.component.html

```
<!-- Property data binding-->
<div>
  <span>O número descrito é - com o property-
  binding: <input [value]="x"></span>

  <span>A data de hoje é: {{ dataHoje }}</span>
  <br /><br />
  <span>O status booleano indicado na variavel é: {{ status
  Booleano }}</span>
  <br /><br />
  <span>{{statusBooleano ? "Este é o resultado para o statu
  s True" : "Este é para o status False"}}</span>
  <br /><br />
  <span>{{ exibirTexto() }}</span>
</div>
```

A saída é a seguinte:



Vinculação/ligação por evento (Event Binding)

A ligação de eventos (event binding) é outra das técnicas de ligação de dados (data binding) disponíveis no Angular. Essa técnica não funciona com o valor dos elementos da interface do usuário - funciona com as **atividades de**

eventos dos elementos da interface do usuário, como **evento de clique**, **evento de desfoque** (blur-event) etc. Nas versões anteriores do Angular, são usadas diferentes tipos de diretivas (directives). como ng-click, ng-blur para vincular qualquer ação de evento específica de um controle (control) HTML. Porém, na versão Angular atual, precisamos usar a mesma propriedade do elemento HTML (como clicar, alterar etc.) e usá-la entre parênteses. No Angular, para propriedades, usamos colchetes e, em eventos, parênteses.

```
<div>

  <h2>Exemplo Event Binding Angular</h2>
  <input type="button" value="Click" class="btn-
block" (click)="showAlert()" />
  <br /><br />
  <input type="button" value="Mouse Enter" class="btn-
block" (mouseenter)="exibirAlerta()" />
</div>
```

Implementação Event Binding (Ligação/Vinculação de dados baseada em evento)

Agora, vamos implementar o data binding - conforme exemplo abaixo – baseado em evento (event binding). Você pode criar um novo projeto Angular chamado *ExemploEventBinding* – ou utilizar seu projeto *angular-alpha* para a implementação. Implemente os códigos abaixo como se seguem:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'angular-alpha';
  // CRIAÇÃO DA FUNÇÃO PARA SER VINCULADA VIA EVENT BINDING
  public exibirAlerta(): void {
    console.log('Evento Disparado...');
    alert('Evento disparado...');
  }
}
```

```
}
```

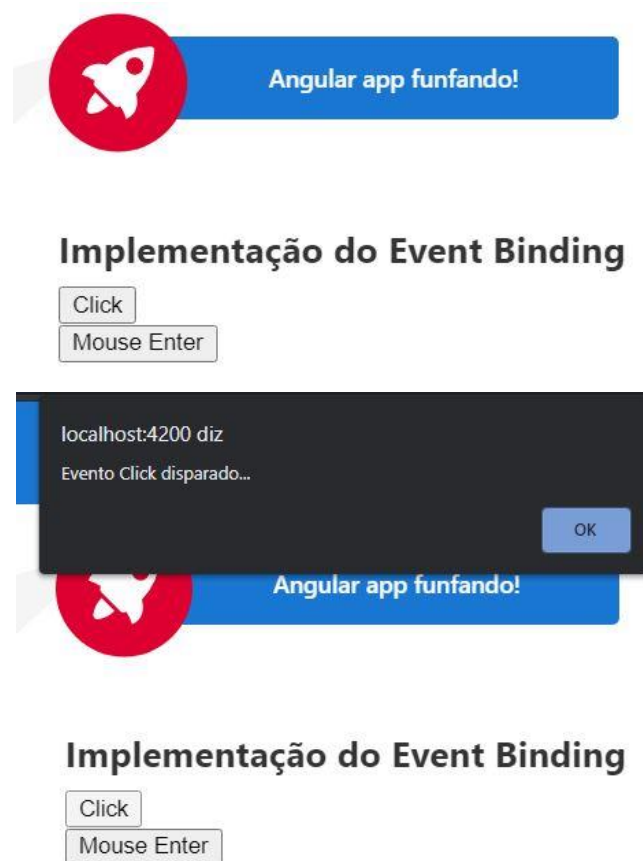
app.component.html

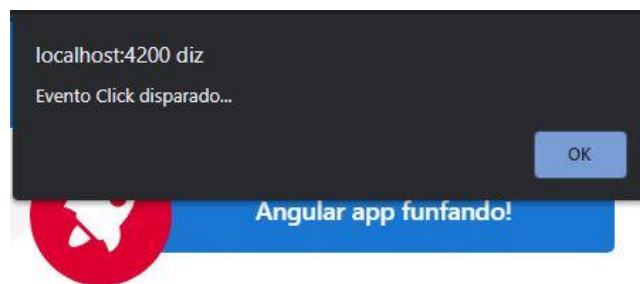
```
<div>
  <h2>Implementação do Event Binding</h2>

  <input type="button" value="Click" (click)="exibirAlerta(
) " />
  <br />
  <input type="button" value="Mouse Enter"
(mouseenter)="exibirAlerta() " />

</div>
```

A saída será a seguinte:





Implementação do Event Binding

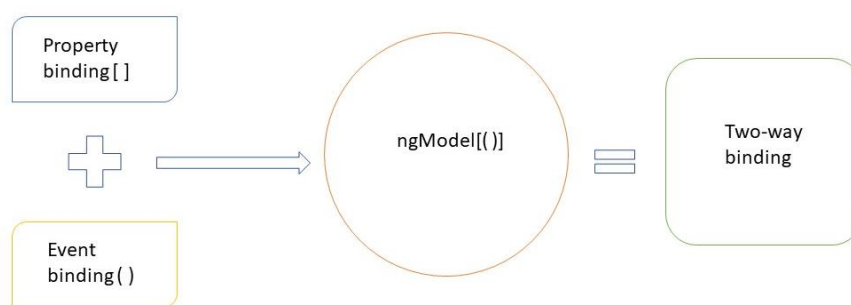


Vinculação/ligação de dados bidirecional (Two-Way Data Binding)

No Angular Framework, as técnicas de ligação de dados (data binding) mais usadas e importantes são conhecidas como Vinculação de Dados Bidirecional (two-way data binding). A ligação bidirecional é usada principalmente no campo do tipo de entrada ou em qualquer elemento de formulário em que o usuário possa fornecer valores de entrada do navegador ou fornecer qualquer valor ou, ainda, alterar qualquer valor de controle por meio do navegador. Por outro lado, o mesmo é atualizado automaticamente nas variáveis do componente e vice-versa. Da mesma forma, no Angular, temos uma diretiva chamada `ngModel`, e ela precisa ser usada como abaixo:

```
<div>
  <div>
    <span>Informe seu nome </span>
    <input [(ngModel)]="x" type="text"/>
  </div>
  <div>
    <span>Seu nome é: </span>
    <span>{{ x }}</span>
  </div>
</div>
```

Usamos `[]`, pois na verdade é uma ligação de propriedade (property binding), e parênteses são usados para o conceito de ligação de evento (event binding), ou seja, a notação de ligação de dados bidirecional é `[]()`.



`ngModel` executa a ligação de propriedades (property binding) e a ligação de eventos (event binding). Na verdade, a ligação de propriedade (property binding) do `ngModel` (ou seja, `[ngModel]`) realiza a atividade para atualizar o elemento de entrada com um valor. Enquanto que (`ngModel`) (evento (`ngModelChange`)) instrui o “mundo externo” quando qualquer alteração ocorreu no elemento DOM.

Implementação Two Way Data Binding (Vinculação/Ligação de dados bidercional)

Agora, nesse exemplo, mostraremos como usar a ligação de dados bidirecional em angular. Para esse fim, você pode criar um novo projeto Angular chamado *ExemploTwoWayBinding* – ou utilizar seu projeto *angular-alpha* para a implementação. Implemente os códigos abaixo como se seguem:

Quando a diretiva *ngModel* ou ligação de dados bidirecional em nossos componentes são implementados, é necessário incluir o módulo de dependencia *FormsModule* do Angular no arquivo *app.module.ts*, como abaixo - o *ngModel* não funcionará sem a inclusão do *FormsModule*:

app.module.ts

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
// imnportando o modulo Forms
import { FormsModule } from '@angular/forms';
  
```

```

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

app.component.ts

```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'Angular';
  // ELEMENTOS PARA IMPLEMENTAR O TWO-WAY BINDING

  public nome: string = '';
}

```

app.component.html

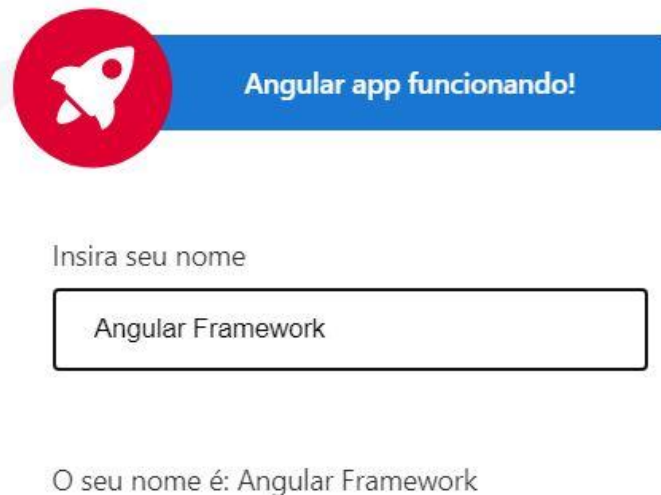
```

<div>
  <div>
    <span>Insira seu nome </span>
    <input [(ngModel)] = "nome" type="text" />
  </div>
  <br /><br />
  <div>
    <span>O seu nome é: </span>
    <span>{{ nome }}</span>
  </div>

```

```
</div>  
</div>
```

A saída é a seguinte:



Input property

No Angular Framework, todos os componentes são usados como componente sem estado (stateless component) ou com estado (statefull component). Normalmente, os componentes são usados e definidos como: sem estado (stateless component). Mas, às vezes, precisamos usar alguns componentes com estado (statefull component). O motivo por trás do uso de um componente com estado (statefull componente) em um aplicativo Web é devido à passagem ou recebimento de dados do componente atual para o componente pai ou um componente filho. Portanto, dessa forma, precisamos indicar ao Angular que tipo de dados ou quais dados podem chegar ao nosso componente atual. Para implementar esse conceito, precisamos usar o decorador `@Input ()` em qualquer variável. Os principais recursos do decorador `@Input ()` são os seguintes:

- `@Input` é um decorador (decorator) para marcar uma propriedade de entrada. Com a ajuda dessa propriedade, podemos definir uma propriedade de parâmetro de entrada, assim como os atributos

normais da tag HTML, para transmitir e vincular esse valor ao componente como uma ligação de propriedade (property binding).

- O decorador (decorator) do `@Input` sempre fornece uma **comunicação de dados unidirecional do componente pai para o componente filho**. Usando esse recurso, podemos fornecer parte do valor em relação a qualquer propriedade de um componente filho dos componentes pai.
- A propriedade *component* deve ser anotada com o decorador `@Input` para atuar como uma propriedade de entrada. Um componente pode receber valor de outro componente usando a ligação de propriedade do componente.

Ele pode ser anotado como qualquer tipo de propriedade, como número, string, array ou classe definida pelo usuário. Para usar um *alias* para o nome da propriedade de ligação (property binding), precisamos atribuir um nome alternativo como `@Input (alias)`. Uso de `@Input` com o tipo de dados string.

```
@Input() public mensagem :string = '';  
  
@Input('alerta') public outraMensagem :string= ''
```

Implementação Input Property

Nesse exemplo, será possível observar a maneira com a qual é possível implementar a propriedade de entrada de um componente. Para esse propósito, é necessário desenvolver o primeiro componente no qual a propriedade de entrada será definida.

Para esse fim, você pode criar um novo projeto Angular chamado *ExemploInputProperty* – ou utilizar seu projeto *angular-alpha* para a implementação. Na sequência, crie um novo componente chamado **message** –

dentro da pasta app do seu projeto – para criar este novo componente - via angular cli – é necessário usar o comando abaixo:

ng generate component message

no terminal do VS Code ou pelo prompt de comando. Edite os arquivos conforme indicado abaixo

message.component.ts

```
import { Component, Input, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-message',
  templateUrl: './message.component.html',
  styleUrls: ['./message.component.css']
})
export class MessageComponent {

  // VAMOS CRIAR OS ELEMENTOS PARA FAZER USO DA PROPRIEDAD
  @INPUT
  @Input() public mensagem: string = '';

  @Input('alerta') public outraMensagem: string = '';

  public exibirAlerta(): void{
    alert(this.outraMensagem);
  }
}
```

message.component.html

```
<div>
  Mensagem: <h3>{{ mensagem }}</h3>
  <input type="button" value="Exibir Alerta" (click)="exibirAlerta()" />
</div>
```

Agora, precisamos consumir esse componente de informações da mensagem em outro componente e passar o valor de entrada usando as propriedades de entrada (input properties).

app.component.ts

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

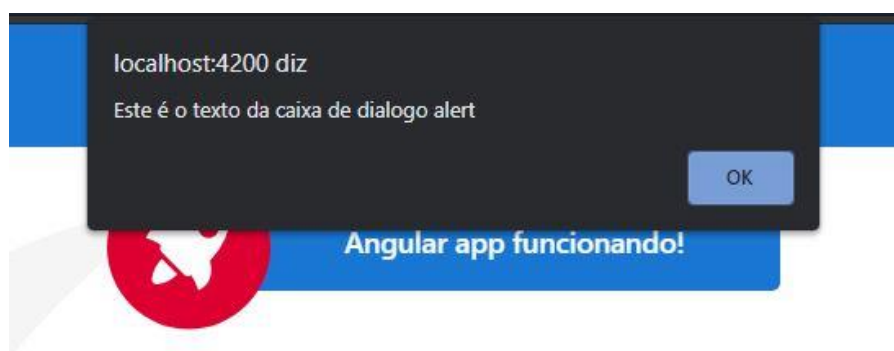
```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular';

  public texto: string = 'Este é o texto da caixa de dialogo
o alert';
}
```

app.component.html

```
<div>
  <app-
message [mensagem]="Desmontração do Input Property de um C
omponent" [alerta]="texto">
  </app-message>
</div>
```

A saída é a seguinte:



Mensagem:

Desmontração do Input Property de um Component

Exibir Alert

Output Property

No Angular, o @Output é um decorador (decorator) usado normalmente como uma propriedade de saída. @Output é usado para definir propriedades de

saída para obter ligação de evento (event binding) personalizada. @Output será usado com a instância do Event Emitter. Os principais recursos do decorador (decorator) @Output () são os seguintes:

- O decorador (decorator) @Output vincula uma propriedade de um componente para enviar dados de um componente (componente filho) para um componente de chamada (componente pai).
- Essa é uma comunicação unidirecional de um filho para um componente pai.
- @Output vincula uma propriedade da classe *EventEmitter*. Se assumirmos o componente como um controle (control), a propriedade de saída atuará como um evento desse controle.
- O decorador (decorator) @Output também pode fornecer opções para personalizar o nome da propriedade usando o nome alternativo como @Output (alias). Nesse caso, esse nome alternativo personalizado atuará como um nome de propriedade de ligação de evento do componente.

Ele pode ser inserido em qualquer tipo de propriedade, como número, string, array ou classe definida pelo usuário. Para usar um *alias* para o nome da propriedade de ligação (property binding), precisamos atribuir um nome alternativo como @Output (alias). Uso de @Output com o tipo de dados string.

Assim como o decorador de entrada, o decorador de saída também precisa primeiro declarar como abaixo

```
@Output() mostrarDados = new EventEmitter<any>();
```

Agora, é necessário que o emissor (emitter) faça a “emissão” – o anúncio do evento - de um ponto no componente para que o evento possa ser gerado e rastreado pelo componente pai.

```
public enviarDados() :void{
    this.mostrarDados.emit(this.data);
}
```

Se queremos passar qualquer valor através desse emissor de evento (event emitter), precisamos passar esse valor como parâmetro através do emissor ().

No componente pai, essa propriedade de saída será definida como abaixo

```
<div>
  <message-
info [message]='Demonstração do Input Property de um Compo
nent' [alert-
pop]='val " (mostrarDados)="exibirDados($event) "></message-
info>
</div>
```

Implementação Output Property

Agora, nesse exemplo, abordaremos como usar a propriedade de saída de qualquer componente. Para esse fim, crie uma copia do exemplo anterior e nomeie-o como *ExemploInputOutputDecorator*. ou utilizar seu projeto *angular-alpha* para a implementação. Será necessário fazer as seguintes alterações no componente de informação da mensagem para definir a propriedade de saída. Observe os conjuntos de código abaixo e implemente-os como se segue:

message.component.ts

```
import { Component, Input, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-message',
  templateUrl: './message.component.html',
  styleUrls: ['./message.component.css']
})
```

```

}))
export class MessageComponent {
  // VAMOS CRIAR OS ELEMENTOS PARA FAZER USO DA PROPRIEDADE
  @OUTPUT

  // aqui, vamos implementar nosso Event Emitter

  @Output() mostrarDados = new EventEmitter<any>();

  // criando a nossa variavel para nosso objeto

  public data:any = {};

  // vamos criar nosso emissor de eventos
  public enviarDados(): void{
    this.mostrarDados.emit(this.data);
  }
}

```

message.component.html

```

<!--
Agora, vamos implementar a estrutura para criarmos um conjunto
de dados com pares chave-
valor para que nosso emitter possa exibir
o conteudo -->
<h2>Output Property</h2>
<br /> <br />

Informe seu nome completo: <input type="text" [(ngModel)]="
data.nome">
<br />
Informe seu email: <input type="email" [(ngModel)]="data.em
ail">
<br />
<input type="button" value="Enviar" (click)="enviarDados()"
/>

```

Portanto, no trecho acima, definimos uma propriedade de saída chamada *enviarDados()* dentro do componente *app-message*.

Para melhorar a visualização dos componentes em tela, implemente este trecho de propriedades dentro do arquivo css do componente *message*:

message.component.css

```
input[type=text], input[type=email]{
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #040670;
  box-sizing: border-box;
}
```

Agora, precisamos consumir esse evento no componente-pai, como mostrado abaixo:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent{
  title = 'Angular';
  // vamos implementar nossa função enviarDados
  public exibirDados(data:any): void{
    let strMessage: string = 'Obrigado por se cadastrar ' +
    data.nome + ".";

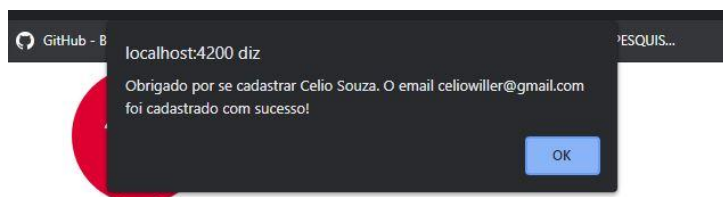
    strMessage += ' O e-
mail ' + data.email + ' foi cadastrado com sucesso.';

    alert(strMessage);
  }
}
```

app.component.html

```
<div>
  <app-message (mostrarDados)="exibirDados($event)">
  </app-message>
</div>
```

A saída é a seguinte:



Output Property

Informe seu nome:

informe seu email:

Directive

As *Directives Angular* proporcionam a manipulação do DOM. Fazendo das directives é possível alterar aparência, comportamento ou a estrutura do layout de um elemento DOM.

Existem três tipos de directives no Angular:

1. Directive Component
2. Structural Directives
3. Attribute Directives

Structural Directives

As directives estruturais (structural directives) podem alterar o layout do DOM adicionando e removendo elementos DOM. Todas as directives estruturais são precedidas pelo símbolo asterisco (*).

ngIf

A diretiva `ngIf` é uma das diretivas Angular estrutural mais usadas; fazendo uso dessa diretiva é possível incluir/excluir elementos DOM considerando alguma condição para tomada de decisão. Seu uso parte da premissa de acoplamento da diretiva a um elemento DOM – pode ser adicionada

a qualquer elemento DOM como *div*, *p*, *h1*, *seletor de componente*, entre outros. Como qualquer outra diretiva estrutural, é prefixado com ***(asterisco)

A descrição como se segue - **ngIf* - é vinculada a uma expressão condicional. Dessa forma, a expressão é, então, avaliada pela diretiva. O retorno da verificação deve ser *“true”* ou *“false”*. Se a expressão for avaliada como *false*, o Angular remove o elemento inteiro do DOM. Se *true*, o elemento será inserido no DOM.

Atributo hidden

Observe o snippet abaixo:

```
<p [hidden]="condição">
    Algum conteúdo a ser exibido
</p>
```

Considerando o uso da diretiva *ngIf* – em relação a propriedade *hidden* observa-se a seguinte definição: *ngIf* não oculta o elemento DOM. Ele remove o elemento inteiro junto com sua sub-árvore; também remove o estado correspondente, liberando os recursos anexados ao elemento; o atributo *hidden* não remove o elemento do DOM, apenas esconde.

Essa é a diferença entre *[hidden]='false'* e **ngIf='false'*; o primeiro método simplesmente oculta o elemento. O segundo método *ngIf* remove o elemento completamente do DOM. Usando o **NOT lógico** – caracter exclamação (!) -, é possível criar uma imitação da condição *else*. Observe o snippet abaixo:

```
<p *ngIf="!condicao">
    conteúdo exibido quando a condição for false
</p>
```

condição - aqui, a condição pode ser qualquer coisa; uma propriedade da classe do componente – por exemplo; também um método na classe do componente. Mas deve ser avaliado como *true/false*. A diretiva *ngIf* tentará forçar o valor para verificação booleana.

Considerando este cenário, a implementação adequada é usar o bloco *else* – como opção.

ngIf else

ngIf possibilita a definição do bloco else - como opção - usando o **ng-template**; a documentação oficial define ng-template como: *“a diretiva ng-template representa um template Angular: isso significa que o conteúdo desta tag irá conter parte de um template (estrutura de view), que pode então ser composto junto com outros templates para formar o template do componente final.”*

```
<div *ngIf="condicao; else blocoElse">
    conteúdo exibido quando a condição for "true"
</div>

<ng-template #blocoElse>
    conteúdo exibido quando a condição for "false"
</ng-template>
```

A expressão algorítmica começa com uma condição seguida por um ponto e vírgula. Na sequência, observa-se a implementação da instrução *else* acoplada a um template denominado *blocoElse*. O *template* pode ser definido em qualquer lugar do bloco de código acima, usando a diretiva *ng-template*. Uma boa prática é indicá-lo logo após a implementação da diretiva *ngIf* facilitando, assim, a leitura do código implementado.

Quando a condição é avaliada como *false*, a diretiva *ng-template* com o nome *#blocoElse* é, então, processada pela diretiva *ngIf*.

ngIf then else

É possível, também, definir o bloco *then else* usando o *ng-template*. Observe o snippet abaixo:

```
<div *ngIf="condicao; then blocoThen else blocoElse">
    Conteúdo não exibido
</div>

<ng-template #blocoThen>
    Conteúdo exibido quando a condição for "true"
</ng-template>
```

```
<ng-template #blocoElse>
  Conteúdo exibido quando a condição for "false"
</ng-template>
```

Considerando o *blueprint* acima, observa-se a instrução *then* seguida por um *template* com o nome *blocoThen*

Quando a condição é *true*, o *template blocoThen* é exibido. Caso a condição seja avaliada como *false*, o *template blocoElse* é exibido.

Implementação

Agora, crie um novo projeto chamado *directive* ou continue usando o projeto angular-alpha para implementar a diretiva *ngIf*.

Abra seu projeto e, no arquivo *app.component.ts*, crie uma variável chamada *exibir* – essa variável será do tipo booleana. Observe o código abaixo e implemente-o como se segue:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-alpha';
  exibir!: boolean;
}
```

No arquivo *app.component.html* implemento seguinte código:

app.component.html

```
<h1>Usando a diretiva ngIf </h1>
<div class="row">
```

```

        Exibir <input type="checkbox" [(ngModel)]="exibir"
/>
</div>
<h1>ngIf </h1>
<p *ngIf="exibir">
    Variavel "exibir" com checkbox ativado
</p>
<p *ngIf="!exibir">
    Variavel "exibir" com checkbox desativado
</p>
<h1>Usando a diretiva ngIf-Else</h1>
<p *ngIf="exibir; else blocoCodigoElse">
    Variavel "exibir" com checkbox ativado usando o "blocoCodigoElse"
</p>
<ng-template #blocoCodigoElse>
    <p>Variavel "exibir" com checkbox desativado usando o "blocoCodigoElse"</p>
</ng-template>
<h1>Usando a diretiva ngIf-then-else</h1>
<p *ngIf="exibir; then blocoCodigoThen else novoBlocoElse">
    Conteudo não exibido!
</p>
<ng-template #blocoCodigoThen>
    <p>Variavel "exibir" com checkbox ativado usando o "blocoCodigoThen"</p>
</ng-template>
<h1>Usando o hidden</h1>
<p [hidden]="exibir">
    Fazendo uso da hidden property binding - este conteúdo será exibido quando a condição for "true"
</p>
<p [hidden]="!exibir">
    Fazendo uso da hidden property binding - este conteúdo será exibido quando a condição for "false"
</p>

```

No arquivo *app.module.ts* é necessário importar o módulo de dependência *FormsModule*. Observe o código abaixo:

app.module.ts

```

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

import { AppRoutingModuleModule } from './app-routing.module';

```

```
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

O resultado no browser deve ser semelhante ao indicado abaixo. Com o checkbox desativado:



Agora, com o checkbox ativado:



ngFor

O `ngFor` é uma *structural directive Angular*, que repete uma parte do template HTML uma vez por cada item de uma lista iterável (coleção); observe o snippet abaixo:

```
<tr *ngFor="let <item> of <items>;">
  <td>{{item.items}}</td>
</tr>
```

`<tr></tr>` - é o elemento sobre o qual a diretiva `ngFor` é aplicada. Este elemento `<tr></tr>` será repetido para cada item encontrado dentro da coleção.

`*ngFor` – é a sintaxe que indica o uso da diretiva. O elemento `*` (*asterisco*) representa sua sintaxe como diretiva estrutural.

`let <item>of<items>` - *item* é a variável auxiliar (de contagem/iteração) do template. Representa o *elemento* atualmente iterado da coleção `<items>`, que deve ser exibida ao usuário. Normalmente a coleção é array de elementos.

O escopo da variável *item* está dentro do `<td></td>`. Você pode acessá-lo em qualquer lugar dentro dele (`<td></td>`), mas não fora dele.

Implementação

Agora, crie um novo projeto chamado *directive* ou continue usando o projeto *angular-alpha* para implementar a diretiva *ngFor*. Abra seu projeto e, no arquivo *app.component.ts*, implemente o seguinte código, como se segue:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = "angular-alpha" ;

  subtítulo: string = "5 filmes sensacionais!";

  filmes: Filme[] =[

    {titulo:'O poderoso Chefão',direcao:'Francis Ford Coppola',elenco:'Marlon Brando, Al Pacino, James Caan',anoLancamento:'10 de setembro, 1972'},
    {titulo:'Um Estranho no Ninho',direcao:'Milos Forman',elenco:'Jack Nicholson, Louise Fletcher, Michael Berryman ',anoLancamento:'26 de maio, 1976'},
    {titulo:'A lista de Schindler',direcao:'Steven Spielberg',elenco:'Liam Neeson, Ralph Fiennes, Ben Kingsley',anoLancamento:'11 de março, 1993'},
    {titulo:'Forrest Gump - O contador de histórias',direcao:'Robert Zemeckis',elenco:'Tom Hanks, Robin Wright, Gary Sinise',anoLancamento:'7 de dezembro, 1994'},
    {titulo:'Laranja Mecânica',direcao:'Stanley Kubrik',elenco:'Malcolm McDowell, Patrick Magee, Michael Bates ',anoLancamento:'04 de setembro, 1971'},
  ]
}

class Filme {
  titulo!: string;
  direcao!: string;
  elenco!: string;
  anoLancamento!: string;
}
```

Para implementar a diretiva ngFor será necessário:

1. Criar um bloco de elementos HTML, que pode exibir um único filme.
2. Usar o ngFor para repetir o bloco para cada filme.

Navegue até o arquivo *app.component.html* e implemente o código abaixo – como se segue:

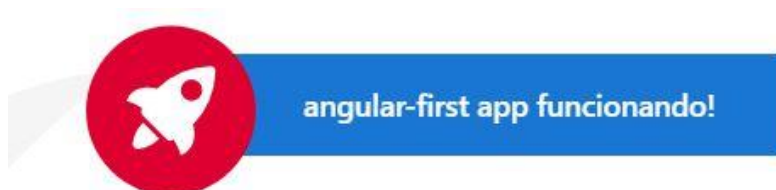
app.component.html

```
<h2>{{ subtítulo }}</h2>

<ul>
  <li *ngFor="let i of filmes">
    {{ i.título }} - {{ i.direcao }}
  </li>
</ul>
```

O elemento `` foi usado para exibir os filmes. O elemento `` exibe um único filme. É necessário, então, repetir o elemento `` para cada filme. Portanto, aqui, se aplica a diretiva `*ngFor` - no elemento ``.

A instrução *let i of filmes* irá iterar sobre a coleção *filmes*, que é uma propriedade da classe do componente. A variável *i* é a variável auxiliar do *template*, que representa o filme iterado da coleção. Foi usada a vinculação de interpolação angular para exibir o título do filme e o nome do diretor. Observe o resultado no browser:



5 filmes sensacionais!

- O poderoso Chefão - Francis Ford Coppola
- Um Estranho no Ninho - Milos Forman
- A lista de Schindler - Steven Spielberg
- Forrest Gump - O contador de histórias - Robert Zemeckis
- Laranja Mecânica - Stanley Kubrik

No console do browser, é possível observar que o projeto Angular gerou um elemento-maracado para cada filme encontrado:

```

▼<ul _ngcontent-ksi-c16>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " O poderoso Chefão - Francis Ford Coppola "
  </li>
  ▼<li _ngcontent-ksi-c16> == $0
    ::marker
    " Um Estranho no Ninho - Milos Forman "
  </li>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " A lista de Schindler - Steven Spielberg "
  </li>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " Forrest Gump - O contador de histórias - Robert Zemeckis "
  </li>
  ▼<li _ngcontent-ksi-c16>
    ::marker
    " Laranja Mecânica - Stanley Kubrik "
  </li>
  <!--bindings={
    "ng-reflect-ng-for-of": "[object Object],[object Object]"
  }-->
</ul>

```

Da mesma forma, é possível usar o elemento *table* para exibir os filmes. Aqui, será necessário repetir o elemento *tr* para cada filme. Portanto, basta aplicar a diretiva **ngFor* sobre o elemento *tr*. Observe o código abaixo e implemente-o como se segue – o arquivo *app.component.ts* não sofre alterações neste passo:

app.component.html

```

<h2>{{ subtítulo }}</h2>

<table class='bordaTabela'>
  <thead>
    <tr>
      <th>Título</th>
      <th>Diretor</th>
      <th>Elenco</th>
      <th>Ano de Lançamento</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let i of filmes;">
      <td>{{ i.título }}</td>
      <td>{{ i.direcao }}</td>

```



```

        <td>{{ i.elenco }}</td>
        <td>{{ i.anoLancamento }}</td>
    </tr>
</tbody>
</table>

```

Para melhorar o visual da tabela, insira o código abaixo dentro do arquivo `app.component.css`:

app.component.css

```

/** estilo da table ngFor*/
.bordaTabela {
  font-family: Arial, Helvetica, sans-serif;
  border-collapse: collapse;
  width: 100%;
}

.bordaTabela td, .bordaTabela th {
  border: 1px solid #ddd;
  padding: 8px;
}

.bordaTabela tr:nth-child(even) {background-color: #f2f2f2;}

.bordaTabela tr:hover {background-color: #ddd;}

.bordaTabela th {
  padding-top: 12px;
  padding-bottom: 12px;
  text-align: left;
  background-color: #581845;
  color: white;
}

```

Attribute Directives

Uma attribute directive ou estilo pode alterar a aparência ou o comportamento de um elemento.

ngClass

A diretiva *ngClass* é usada para adicionar ou remover as classes CSS de um elemento HTML. Usar *ngClass* pode criar estilos dinâmicos em páginas HTML.

Implementação

Agora, crie um novo projeto chamado *directive* ou continue usando o projeto angular-alpha para implementar a diretiva *ngClass*.

Abra seu projeto e, no arquivo *app.component.ts*, crie uma variável chamada *cssalteradoViaVar* – essa variável será inicializada com duas propriedades *'color'* e *'size'*. Além de criar essa variável, fora da classe principal – *AppComponent* – crie uma nova classe chamada *UmaClasseCss* e, na sequencia, crie sua instância dentro da classe principal. Observe o código abaixo e implemente-o como se segue:

app.component.ts

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = "angular-alpha" ;

  cssAlteradoViaVar: string= 'color size';
  umaClasseCss: UmaClasseCss = new UmaClasseCss();
}

// criando uma nova classe fora da classe principal
class UmaClasseCss {
  color: boolean= true;
  size: boolean= true;
}
```

Agora, é necessário implementar as linhas de código abaixo dentro do arquivo *.css* do componente. Observe o código abaixo e implemente-o como se segue:

app.component.css

```
/*estilo para ngClass*/
```

```
.color {
  color: blue;
}

.size {
  font-size: 35px;
}
```

Implemento o código abaixo dentro do arquivo *app.component.html* como se segue:

app.component.html

```
<p>Diretiva ngClass </p>
  <!-- primeira indicação de implementação-->
  <div [ngClass]="['color size']">
    Texto com fonte e cor indicados no arquivo CSS: con
figurado no template como uma propriedade string
  </div>

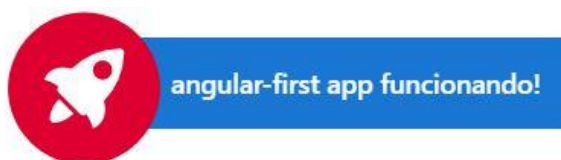
  <div [ngClass]="['color','size']">
    Texto com fonte e cor indicados no arquivo CSS: con
figurado no template como um array
  </div>

  <!-- Obtendo os dados a partir do arquivo .ts. -->
  <!--
- É possível alterar o valor a partir do arquivo .ts. -->
    <div [ngClass]="cssAlteradoViaVar">
      Texto com fonte e cor indicados no arquivo CSS:
configurado no template a partir do arquivo app.component.
ts
    </div>

    <div [ngClass]="umaClasseCss">
      Texto com fonte e cor indicados no arquivo CSS:
configurado no template a partir do arquivo app.component.
ts como object
    </div>
```

Acima, é possível observar as diferentes formas de implementação das propriedades de estilo configuradas no arquivo `.css`. A diretiva `ngClass` é aplicada em todas as modalidades com diferentes implementações.

Salve o projeto. Observe o resultado no browser. Deve ser semelhante ao indicado abaixo:



Diretiva `ngClass`

Texto com fonte e cor indicados no arquivo CSS: configurado no template como uma propriedade string

Texto com fonte e cor indicados no arquivo CSS: configurado no template como uma string

Texto com fonte e cor indicados no arquivo CSS: configurado no template como um array

Texto com fonte e cor indicados no arquivo CSS: configurado no template como um object

Texto com fonte e cor indicados no arquivo CSS: configurado no template a partir do arquivo `app.component.ts`

Texto com fonte e cor indicados no arquivo CSS: configurado no template a partir do arquivo `app.component.ts` como object

ngStyle

`ngStyle` é usado para alterar as várias propriedades de estilo de nossos elementos HTML. Também podemos vincular essas propriedades a valores que podem ser atualizados pelo usuário ou por nossos componentes. Observe o *snippet* abaixo:

```
<div [ngStyle]="{'color': 'blue', 'font-size': '24px',  
'font-weight': 'bold'}">  
    Algum texto  
</div>
```

Implementação

Agora, crie um novo projeto chamado *directive* ou continue usando o projeto angular-alpha para implementar a diretiva `ngStyle`.

Abra seu projeto e, no arquivo `app.component.ts`, crie três variáveis: `tamanho`, `color`, `estiloClasse` – essas variáveis serão inicializadas com o o

valores indicados no código abaixo. Além de criar as variáveis, fora da classe principal – AppComponent – crie uma nova classe chamada EstiloClasse, na sequência, crie sua instância dentro da classe principal. Observe o código abaixo e implemente-o como se segue:

app.component.ts

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = "angular-alpha" ;

  tamanho: number = 12;
  color: string= '#380F6C';
  estiloClasse: EstiloClasse = new EstiloClasse();
}

// criando uma nova classe fora da classe principal
class EstiloClasse {
  color: string= '#FF5733';
  'font-size.%': number= 150;
  'font-weight': string= 'bold';
}
```

Agora, implemento o código abaixo – como se segue – dentro do arquivo *app.component.html*

app.component.html

```
<h1>Diretiva ngStyle</h1>

  <br /><br />
  <input [(ngModel)]="color" />

  <div [ngStyle]="{'color': color}">
```

```

    Mudar a cor do texto
  </div>

  <input [(ngModel)]="tamanho" />

  <div [ngStyle]="{'font-size.px': tamanho}">
    Mudar o tamanho da fonte
  </div>

  <div [ngStyle]="estiloClasse">
    Mudar a cor e o tamanho do texto alterando as p
    roriedades no arquivo app.component.ts
  </div>

```

Acima, o código demonstra as diferentes implementações da diretiva `ngStyle`. Dessa forma é possível alterar o estilo dos componentes do projeto Angular.

Salve o projeto e execute. O resultado no browser deve ser semelhante ao indicado abaixo:



Directive Component

Componentes (Components) são directives especiais em Angular. Eles são as directives com um template (exibição)

Criando directives personalizadas

É possível criar directives personalizadas no Angular. O processo é criar uma classe JavaScript e aplicar o atributo **@Directive** a essa classe. Você pode escrever o comportamento desejado na classe.

Como criar directives personalizadas?

Diretivas personalizadas são criadas por nodes e não são padrão.

Vamos ver como criar a diretiva personalizada. Criaremos a diretiva usando a linha de comando. O comando para criar a diretiva usando a linha de comando é o seguinte:

```
ng g directive nomedadiretiva
→ ng g directive alterar-texto
```

Ele aparece na linha de comando, conforme indicado no código abaixo:

```
Projetos-Angular\angular-alpha>ng g directive alterar-
texto
CREATE src/app/alterar-texto.directive.spec.ts (249
bytes)
CREATE src/app/alterar-texto.directive.ts (153 bytes)
UPDATE src/app/app.module.ts (487 bytes)
```

```
CREATE src/app/alterar-texto.directive.spec.ts (249 bytes)
CREATE src/app/alterar-texto.directive.ts (153 bytes)
UPDATE src/app/app.module.ts (487 bytes)
```

Os arquivos acima, ou seja, *alterar-texto.directive.spec.ts* e *alterar.directive.ts*, são criados e o arquivo *app.module.ts* é atualizado.

app.module.ts

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

import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AlterarTextoDirective } from './alterar-
texto.directive';

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective
```

```

],
imports: [
  BrowserModule,
  AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

A classe *AlterarTextoDirective* está incluída nas declarações do arquivo acima. A classe também é importada do arquivo fornecido abaixo:

alterar-texto.directive.ts

```

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

@Directive({
  selector: '[appAlterarTexto]'
})
export class AlterarTextoDirective {

  constructor() { }

}

```

O arquivo acima tem uma diretiva e também possui uma propriedade seletora. Tudo aquilo que for definido na propriedade *selector* deve corresponder na *view* onde a diretiva customizada for atribuída

Na *view app.component.html*, adicione a diretiva como se segue abaixo:

app.component.html

```
<h1 appAlterarTexto></h1>
```

Abaixo, serão implementadas as alterações *alterar-texto.directive.ts*. Implemente o código como se segue:

alterar-texto.directive.ts

```

import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appAlterarTexto]'
})
export class AlterarTextoDirective {

  constructor(Element: ElementRef) {
    console.log(Element);
    Element.nativeElement.innerText = "Este é o texto inserido a partir da diretiva customizada!";
  }

}

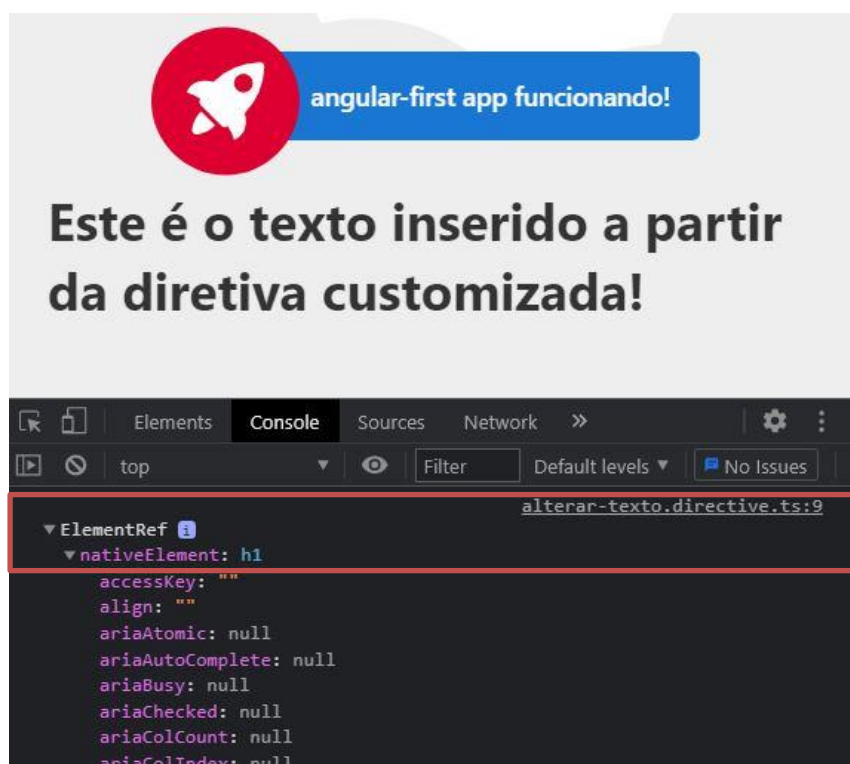
```


}

No arquivo acima, a classe principal se chama *AlterarTextoDirective* e um construtor, que utiliza o elemento do tipo ***ElementRef*** - obrigatório. O elemento possui todos os detalhes aos quais a diretiva ***appAlterarTexto*** é aplicada.

Ao adicionar a instrução *console.log* a saída de *ElementRef* pode ser observada no console do navegador. O texto atribuído a *Element* é inserido.

Agora, o navegador mostrará o seguinte:



Acima, observa-se todas as propriedades do elemento nativo html para o qual o seletor da diretiva – *appAlterarTexto* – foi referenciado. Como a diretiva ***appAlterarTexto*** foi adicionada a uma tag `<h1></h1>`, os detalhes do elemento estão disposto no console do browser.

Pipes

Neste próximo passoserá possível observar o trabalho com *pipes* Angular. Os pipes foram anteriormente chamados de filtros na primeira versão do angular e renomeados para pipes – a partir da segnda versão.

O caractere `|` é usado para aplicar transformações em dados. Observe a sintaxe de um pipe *built-in*:

```
{{ Ola Angular | lowercase }}
```

É possível trabalhar com pipe aplicando qualquer tidpo de dado -por exemplo: números inteiros, seqüências de caracteres, matrizes, entre

outros.usa-se o caractere | (barra vertical) para converter no formato conforme necessário e exibi-lo no navegador.

Aqui, o objetivo é exibir o texto fornecido em maiúsculas. Isso pode ser feito usando pipes da seguinte maneira:

Aqui estão alguns pipes embutidos disponíveis com angular:

- Lowercasepipe
- Uppercasepipe
- Datepipe
- Currencypipe
- Jsonpipe
- Percentpipe
- Decimalpipe
- Slicepipe

Agora, observe a aplicação do pipe. As seguintes linhas de código nos ajudarão a definir as variáveis necessárias no arquivo *app.component.ts*:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-alpha';

  dataHoje = new Date();
  umJson = {nome:'Dexter', idade:'8', endereco:{rua:'Rua da Casinha', numero:'01'}};
  mesesAno = ['Jan', 'Fev', 'Mar', 'Abr', 'Mai', 'Jun', 'Jul', 'Ago', 'Set', 'Out', 'Nov', 'Dez'];
}
```

Os pipes serão implementados no arquivo *app.component.html*, como mostrado abaixo:

app.component.html

```
<div style="width:100%">
  <div style="width:40%; float:left;border:solid 1px black">
```

```

    <h2>Uppercase Pipe</h2>
    <p>{{ title | uppercase }}</p>
    <br />

    <h2>Lowercase Pipe</h2>
    <p>{{ title | lowercase }}</p>
    <br />

    <h2>Currency Pipe - Pipe para conversão de moeda</h2>
    <p>{{ 6589.23 | currency:"USD" }}</p>
    <p>{{ 6589.23 | currency:"USD":true }}</p>
    <br />

    <h2>Date Pipe</h2>
    <p>{{ dataHoje | date:'d/M/y' }}</p>
    <p>{{ dataHoje | date:'shortTime' }}</p>
    <br />

    <h2>Decimal Pipe</h2>
    <p>{{ 454.78787814 | number:'3.4-4' }}</p>
    <br />
  </div>
  <div style="width:40%; float:left;border:solid 1px black">
    <h2>Json Pipe</h2>
    <b>{{ umJson | json }}</b>
    <br />

    <h2>Percent Pipe</h2>
    <b>{{ 00.325 | percent }}</b>
    <br />

    <h2>Slice Pipe</h2>
    <b>{{ mesesAno | slice:2:6 }}</b>
    <br />
  </div>

```

Para que o uso pipe aplicado a conversão de moedas – neste cenário, convertendo o número para a moeda BRL (real brasileiro) – deve sere implementado no arquivo *app.module.ts* algumas dependências Angular. Implemento o código abaixo como indicado:

app.module.ts

```

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

```

```

import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AlterarTextoDirective } from './alterar-
texto.directive';
import { ComponenteFilhoComponent } from './componente-
filho/componente-filho.component';

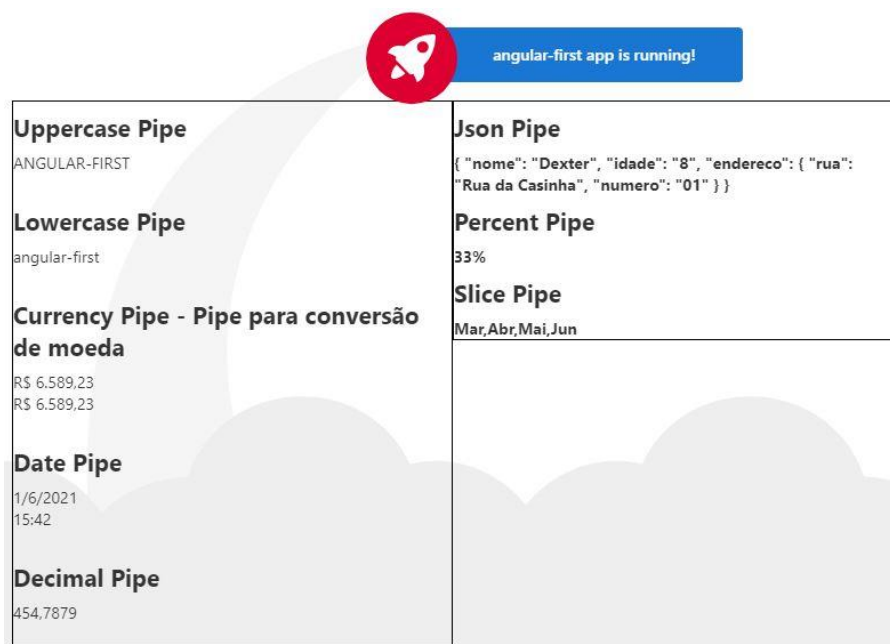
import { LOCALE_ID, DEFAULT_CURRENCY_CODE } from '@angular/co
re';
import localePt from '@angular/common/locales/pt';
import { registerLocaleData } from '@angular/common';

registerLocaleData(localePt, 'pt');

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModuleModule
  ],
  providers: [ {
    provide: LOCALE_ID,
    useValue: 'pt'
  },
  {
    provide: DEFAULT_CURRENCY_CODE,
    useValue: 'BRL'
  }, ],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

A captura de tela a seguir mostram a saída para cada pipe:



Pipe personalizado

Para criar um pipe personalizado, será necessário criar um novo arquivo `.ts`. Com este pipe personalizado será possível calcular a raiz quadrado de algum número fornecido e lido pelo pipe. Para criar o pipe, clique com o botão direito do mouse em cima da pasta `app` e escolha *New File*; nomeie-o como *pipe-raiz-quadrada.ts*. Na sequência, abra o arquivo recém-criado e implemente o código abaixo como indicado:

→ **new file > pipe-raiz-quadrada.ts**

pipe-raiz-quadrada.ts

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe ({
  name : 'raizquadrada'
})
export class RaizQuadrada implements PipeTransform {
  transform(numero : number) : number {
    return Math.sqrt(numero);
  }
}
```

Para criar um pipe personalizado, foi necessário importar a dependência *Pipe* e *PipeTransform* de *@angular/core*. Na diretiva *@Pipe*, foi indicado um nome para o pipe, que será referenciado dentro do arquivo *app.component.html*.

Na sequência foi criada a classe *RaizQuadrada*. Esta classe implementará o *PipeTransform*.

O método de *transform()* definido na classe terá o argumento *número* e retornará o valor depois de obter a raiz quadrada.

Agora que o pipe foi criado e implementado, será necessário adicioná-lo e registrá-lo no arquivo *app.module.ts*. Isso é feito da seguinte maneira:

app.module.ts

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

import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AlterarTextoDirective } from './alterar-texto.directive';
import { ComponenteFilhoComponent } from './componente-filho/componente-filho.component';

import { RaizQuadrada } from './pipe-raiz-quadrada';

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent,
    RaizQuadrada
  ],
  imports: [
    BrowserModule,
    AppRoutingModuleModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Agora, implemente a chamada feita para o *pipe* raiz-quadrada no arquivo *app.component.html*.

app.component.html

```
<h2>Custom Pipe</h2>
<b>Raiz quadrada de 25 é: {{ 25 | raizquadrada }}</b>
<br />
<b>Raiz quadrada de 231 é: {{ 231 | raizquadrada }}</b>
<br />
<b>Raiz quadrada de 2 é: {{ 2 | raizquadrada }}</b>
```

A seguir está a saída:



Forms

Neste passo, veremos como os formulários são usados no Angular. Discutiremos duas maneiras de trabalhar com formulários:

- Template Driven Form
- Model Driven Form

Template Driven Form

Com um Template Driven Form (formulário orientado a modelo), a maior parte do trabalho é feita no *template*. Com o formulário controlado pelo *model*, a maior parte do trabalho é realizada na classe de componentes (camada lógica).

Vamos, agora, trabalhar com formulário orientado *template*. Criaremos um formulário simples e adicionaremos o email, a senha e o botão enviar. Para começar, precisamos importar para *FormsModule* a partir de *@angular/forms*, o que é feito em *app.module.ts* da seguinte maneira:

app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
```

```

],
imports: [
  BrowserModule,
  AppRoutingModule,
  BrowserModule,
  FormsModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Portanto, em *app.module.ts*, importamos o *FormsModule* e seu registro é adicionado ao array *imports*, conforme mostrado no código destacado.

Vamos agora criar o formulário no arquivo *app.component.html*

app.component.html

```

<form #userlogin = "ngForm" (ngSubmit) = "onClickSubmit(u
serlogin.value)">
  <input type = "text" name = "email" placeholder = "Insi
ra seu email" ngModel>
  <br/>
  <input type = "password" name = "senha" placeholder = "
Insira sua senha" ngModel>
  <br/>
  <input type = "submit" value = "Enviar">
</form>

```

Criamos um formulário simples com tags de entrada com identificação de email, senha e o botão enviar.

Nos formulários controlados por *template*, é necessário criar os controles do formulário adicionando a diretiva *ngModel* e o atributo *name*. Portanto, onde quisermos que o Angular acesse os dados a partir de formulários, basta adicionar a diretiva *ngModel* a essa tag, como mostrado acima. Agora, se precisarmos ler os atributos *email* e o *senha*, precisamos adicionar o *ngModel*.

Se observarmos, também adicionamos o *ngForm* ao elemento identificador *#userlogin*. A diretiva *ngForm* precisa ser adicionada ao *template*. Também adicionamos a função *onClickSubmit* e atribuímos *userlogin.value* a ela.

Vamos agora criar a função no *app.component.ts* e buscar os valores inseridos no formulário. Implemente o código abaixo – como indicado:

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-frist';

  enviarDados(data:any) {
    alert("O e-mail inserido foi: " + data.email);
  }
}
```

No arquivo *app.component.ts* acima, definimos a função *enviarDados*. Quando o usuário clicar no botão de envio do formulário, o controle chegará à função acima.

O css do formulário é adicionado em *app.component.css*:

app.component.css

```
input[type = text], input[type = password] {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #B3A9A9;
  box-sizing: border-box;
}
input[type = submit] {
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #B3A9A9;
  box-sizing: border-box;
}
```

O navegador exibirá nossa tela como mostrado abaixo:



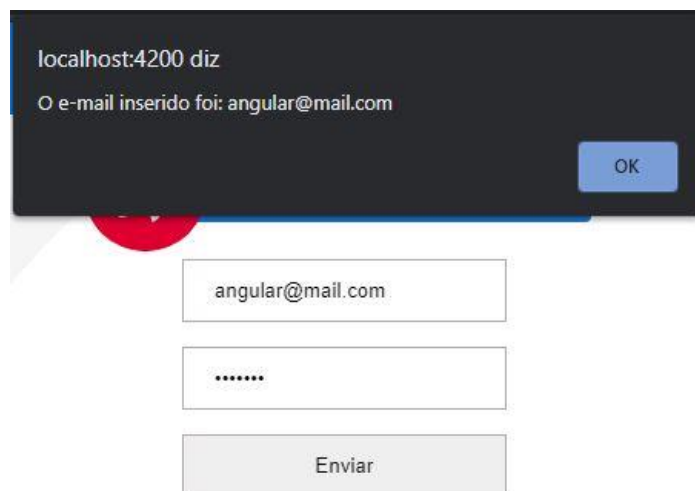
angular-frist app funcionando!

Insira seu email

Insira sua senha

Enviar

Inserindo os dados no formulário, na função de envio, o email é exibido na janela de alerta, como mostrado abaixo:



localhost:4200 diz

O e-mail inserido foi: angular@mail.com

OK

angular@mail.com

.....

Enviar

Model Driven Form

No Model Driven Form (formulário controlado pelo *model*), é necessário importar a dependencia *ReactiveFormsModule* de *@angular/forms* e registrá-lo no array *imports*:

Há uma alteração em ***app.module.ts***. Observe e implemente como se segue:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { ReactiveFormsModule } from '@angular/forms';
import { BrowserModuleAnimationsModule } from '@angular/platform-browser/animations';
```

```
import { FormsModule, ReactiveFormsModule } from '@angular/
forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

No arquivo *app.component.ts*, será preciso importar alguns módulos para o formulário controlado pelo *model*. Por exemplo, *import {FormGroup, FormControl}* de ' @ angular / forms '. Observe o código abaixo e implemente-o como se segue:

app.component.ts

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular
r/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-alpha';
  email:any;
  formdata:any;

  ngOnInit() {
    this.formdata = new FormGroup({
      email: new FormControl("agular@mail.com"),
      senha: new FormControl("#$@%")
    });
  }
  enviarDados(data:any) {this.email = data.email;}
}
```

Os dados do formulário são inicializados no início da classe e, posteriormente, inicializado com *FormGroup*, como mostrado acima. As variáveis *email* e *senha* são inicializadas com valores padrão a serem exibidos no formulário. Você pode mantê-lo em branco, se desejar.

Usamos *formdata* para inicializar os valores do formulário; será necessário usar o mesmo no formato no arquivo ***app.component.html***.

app.component.html

```
<div>
  <form [formGroup] = "formdata" (ngSubmit) = "enviarDados(
formdata.value)" >
    <input type = "text" name = "email" placeholder = "Ins
ira seu email" formControlName = "email">
    <br/>
    <input type = "password" name = "senha" placeholder =
"Insira sua senha" formControlName = "senha">
    <br/>
    <input type = "submit" value = "Enviar">

  </form>
</div>
<p> O email informado foi : {{ email }} </p>
```

É assim que os valores serão vistos no formulário da interface do usuário.



The screenshot displays a web application interface. At the top, there is a red circular logo with a white rocket icon, followed by a blue banner with the text "angular-first app funcionando!". Below this, there is a form with three input fields: the first contains "angular@mail.com", the second is a password field with masked characters "*****", and the third is a button labeled "Enviar". At the bottom of the form, there is a label "O email informado foi :".



angular-first app funcionando!

angular@mail.com

Enviar

O email informado foi : angular@mail.com

No arquivo *app.component.html*, usamos *formGroup* entre colchetes para o formulário; por exemplo, *[formGroup] = "formdata"*. No envio, a função é chamada *enviarDados* para a qual *formdata.value* é passado.

A marcação de entrada *formControlName* é usada. É dado um valor que foi implementado no arquivo *app.component.ts*.

Ao clicar em enviar, o controle passará para a função *enviarDados*, definida no arquivo *app.component.ts*.

Validação de formulário

É possível usar a validação de formulário interna ou também a abordagem de validação personalizada. Usaremos as duas abordagens no formulário. Continuaremos com o mesmo exemplo que criamos em uma de nossas seções anteriores. Com o Angular, precisamos importar *validators* (validadores) de *@angular/forms* como mostrado abaixo:

```
import { FormGroup, FormControl, Validators } from
'@angular/forms'
```

Angular foi construído com validadores tais como: *campo de preenchimento obrigatório* (*require*), *minlength*, *maxlength* entre outros. Devem ser acessados usando o módulo de dependência *Validators*.

É possível apenas adicionar validadores ou um array de validadores necessários para informar ao Angular se um determinado campo é obrigatório. Agora, o objetivo é implementar validadores em uma das caixas de texto de entrada, ou seja, a caixa para inserção de email. Para email, serão adicionados os seguintes parâmetros de validação:

- *require*
- Correspondência de padrões

Observe o código abaixo e implemente-o como indicado:

app.component.ts

```

import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-alpha';
  email:any;
  formdata:any;

  ngOnInit() {
    this.formdata = new FormGroup({
      email: new FormControl("", Validators.compose([
        Validators.required,
        Validators.pattern("^[^ @]*@[^ @]*")
      ])),
      senha: new FormControl("")
    });
  }
  enviarDados(data:any) {this.email = data.email;}
}

```

A partir do elemento *Validators.compose*, é possível adicionar a lista de itens que se deseja validar no campo de entrada. No momento, foram adicionados os parâmetros **required** e de **Validators.pattern** para receber apenas emails válidos.

No arquivo *app.component.html* o botão enviar será desativado se alguma das entradas do formulário não for válida. Isso é feito da seguinte maneira:

app.component.html

```

<div>
  <form [formGroup] = "formdata" (ngSubmit) = "enviarDados(
formdata.value)" >
    <input type = "text" name = "email" placeholder = "Insira seu email" formControlName = "email">
    <br/>
    <input type = "password" name = "senha" placeholder = "Insira sua senha" formControlName = "senha">
    <br/>
    <input type = "submit" [disabled] = "!formdata.valid" value = "Enviar">

  </form>
</div>

```

```
<p> O email informado foi : {{ email }} </p>
```

Para o botão enviar, foi adicionada a instrução [disabled] no colchete, que recebe o valor da expressão lógica de negação conforme indicado abaixo:

```
<input type = "submit" [disabled] = "!formdata.valid" value  
= "Enviar">
```

Portanto, se o *formdata.valid* não for válido, o botão permanecerá desativado e o usuário não poderá enviá-lo.

Vamos ver como isso funciona no navegador:



angular-first app funcionando!

Insira seu email

Insira sua senha

Enviar

O email informado foi :

No caso acima, não foi inserido qualquer email, portanto, o botão de login está desativado. Vamos agora tentar inserir o ID de email válido e ver a diferença.



angular-first app funcionando!

angular@mail.com

.....

Enviar

O email informado foi : angular@mail.com

Agora, email inserido é válido. Assim, é possível ver que o botão de login está ativado e o usuário poderá enviá-lo. Com isso, o email inserido é exibido na parte inferior.

Neste próximo passo, será possível implementar uma validação personalizada com o mesmo formulário. Para validação personalizada, é possível definir nossa própria função e adicionar os detalhes necessários. Observe o código abaixo e impelete como indicado:

app.component.ts

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-alpha';
  email:any;
  formdata:any;

  ngOnInit() {
    this.formdata = new FormGroup({
      email: new FormControl("", Validators.compose([
        Validators.required,
        Validators.pattern("^[^ @]*@[^ @]*")
      ])),
      senha: new FormControl("", this.validacaoSenha)
    });
  }

  validacaoSenha(formcontrol:any) {
    if (formcontrol.value.length < 5) {
      return {"senha" : true};
    }
    return null;
  }

  enviarDados(data:any) {this.email = data.email;}
}
```


No exemplo acima, foi criada a função *validacaoSenha* e usada em *formcontrol*:

```
senha: new FormControl("", this.validacaoSenha)
```

Na função criada, será possível verifica o “comprimento” (quantidade de caracteres) dos caracteres digitados dentro do input. Se os caracteres forem menores que cinco, ele retornará com a senha true, como mostrado acima - retorne {"senha": true};. Se a senha inserida for composta por mais de cinco caracteres, será considerado válido e o login será ativado. Caso contrario o botão *Enviar* permanecerá desativado.

O código do arquivo *app.component.html* permanece sem alterações:

app.component.html

```
<div>
  <form [formGroup] = "formdata" (ngSubmit) = "enviarDados(
formdata.value)" >
    <input type = "text" name = "email" placeholder = "Ins
ira seu email" FormControlName = "email">
    <br/>
    <input type = "password" name = "senha" placeholder =
"Insira sua senha" FormControlName = "senha">
    <br/>
    <input type = "submit" [disabled] = "!formdata.valid"
value = "Enviar">

  </form>
</div>
<p> O email informado foi : {{ email }} </p>
```

Vamos agora ver como isso é exibido no navegador:



Inserimos apenas quatro caracteres são inseridos dentro da caixa referente a inserção da senha e o botão de *Enviar* está desativado. Para habilita-lo, será necessário inserir cinco caracteres ou mais. Vamos agora inserir um tamanho válido de caracteres e verificar o resultado. Observe a imagem abaixo:



The image shows a web interface for a login form. At the top, there is a red circular icon with a white rocket. To its right is a blue rectangular button with the text "angular-first app funcionando!". Below this, there is a light blue input field containing the email "angular@mail.com". Underneath the email field is a white input field for a password, which contains five dots. Below the password field is a grey button with the text "Enviar". At the bottom of the form, there is a line of text that reads "O email informado foi : angular@mail.com".

O botão enviar foi ativado, pois o email e a senha são válidos – se encaixam nos padrões descritos no *arquivo app.component.ts*.

Services

Nossos componentes precisam acessar dados (conteúdo ou funcionalidade). É possível escrever o código de acesso a dados em cada componente, mas isso, por vezes, é ineficiente e pode quebrar a regra da “responsabilidade única”. O componente deve se concentrar em apresentar dados ao usuário. A tarefa de obter dados do servidor *back-end* deve ser delegada para outra classe. Essa classe é chamada de classe de serviço (service class). Porque proíbe ao service fornecer dados para todos os componentes que precisam.

O que é um service

O service é um pedaço de código reutilizável com um objetivo focado. Um código que você o usará em muitos componentes em seu aplicativo

Para que services são utilizados

1. Recursos independentes de componentes como services de log
2. Compartilhe lógica ou dados entre componentes
3. Encapsula interações externas, como acesso a dados

Vantagens ao se usar services

1. Os services são mais fáceis de testar.
2. Os services são mais fáceis de depurar.
3. Você pode reutilizar o service.

Como criar um service no Angular

Um *service* Angular é simplesmente uma função Javascript. Tudo o que precisamos fazer é criar uma classe e adicionar métodos e propriedades. Em seguida, podemos criar uma instância dessa classe em nosso componente e chamar seus métodos.

Um dos melhores usos dos *services* é obter os dados de uma fonte interna ou externa. Neste primeiro, será criado um *service* simples, que obtém os dados de produtos e os transmite ao componente.

Implementação

Usando o projeto *angular-alpha* para implementar o *Service* será necessário criar um novo arquivo que servirá como model para fazer o tratamento dos dados dispostos na aplicação. Para criar este novo arquivo, clique com o botão direito do mouse na pasta *app* do seu projeto – escolha *New File* e dê o nome ao arquivo de *product.ts*.

→ new file > product.ts

Product (produtos)

O arquivo criado na pasta *src/app* e nomeado como *product.ts* deverá conter o código abaixo:

product.ts

```
export class Produto {
```

```

    constructor(idProduto:number,      nomeProduto: string ,
    precoProduto:number) {
        this.idProduto= idProduto;
        this.nomeProduto = nomeProduto;
        this.precoProduto = precoProduto;
    }

    idProduto:number ;
    nomeProduto: string ;
    precoProduto:number;
}

```

A classe *Product* acima é o nosso *domain model* (modelo de domínio).

Product Service (Service do produto)

Em seguida, vamos criar um *service* angular, que retorna a lista a partir do *model products*. Para criar o novo service, abra o prompt de comando – via *vs code* ou pelo próprio computador – e, dentro da pasta de seu projeto (*angular-alpha*), insira a seguinte instrução:

```

ng g service nomedoservice
→ ng g service product

```

```

CREATE src/app/product.service.spec.ts (362 bytes)
CREATE src/app/product.service.ts (136 bytes)

```

```

CREATE src/app/product.service.spec.ts (362 bytes)
CREATE src/app/product.service.ts (136 bytes)

```

Após a criação do service, implemente o código abaixo como se segue:

product.service.ts

```

import { Produto } from './product';

export class ProductService{

    public getProdutos() {

        let produtos:Produto[];

        produtos = [
            new Produto(1,'Quadro Mestre Yoda',199),
            new Produto(2,'Mascara Darth Vader',159),
            new Produto(3,'Lightsaber',89)
        ]
    }
}

```

```

        return produtos;
    }
}

```

Primeiro, foi importado para o arquivo *product.service.ts* a class *Product* (a partir de ser arquivo de origem *product.ts*) - criado para estabelecer o model domain.

A classe *ProductService* foi criada e exportada. Foi criado, também, um método chamado *getProdutos()* que retorna a coleção *produtos[]*. Aqui, o conjunto de dados está disposto dentro da aplicação, no entanto, é possível utilizar procedimento semelhante para obter dados de uma fonte externa a partir do envio de uma solicitação HTTP para uma API de back-end para obter os dados.

O service está pronto. Observe que a classe acima é uma função JavaScript simples. Não há nada angular nisso.

Invocando o ProductService

A próxima etapa é chamar o *ProductService* do componente. Abra o *app.component.ts* e adicione o seguinte código.

app.component.ts

```

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

import { FormsModule } from '@angular/forms';
import { ProductService } from './product.service';
import { Produto } from './product';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent{
  title: string = "angular-alpha";
  produtos!: Produto[];
  productService;

```

```

constructor() {
  this.productService = new ProductService();
}

obterProdutos() {
  this.produtos = this.productService.getProdutos();
}
}

```

Primeiro, foram importadas as classes *Produto* & *ProductService* – a partir de seus arquivos de origem.

No construtor do *AppComponent*, foi criada instância da classe *ProductService*().

O método *obterProduto()* chama o método *getProdutos()* implementado dentro do arquivo *product.service.ts*. A lista retornada de *Produtos* é armazenada na variável local *produtos*

Template

O próximo passo é exibir *Products* para o usuário. Abra o arquivo ***app.component.html*** e adicione o seguinte código abaixo, como se segue:

app.component.html

```

<div>
  <h1><strong>Implementação </strong>Services</h1>

  <button type="button" (click)="obterProdutos()">Obter P
rodutos</button>

  <div>
    <br />
    <table class="bordaTabela">
      <thead>
        <tr>
          <th>Identificação do Produto</th>
          <th>Nome do Produto</th>
          <th>Preço do Produto</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let x of produtos;">
          <td>{{ x.idProduto }}</td>
          <td>{{ x.nomeProduto }}</td>

```

```

<td>{{ x.precoProduto }}</td>
</tr>
</tbody>
</table>
</div>
</div>

```

No código acima foi adicionado um botão chamado "*Obter Produtos*", que é vinculado ao método *obterProdutos()* da classe do componente por meio de *event binding* (associação de eventos).

Os produtos são exibidos/retornados via diretiva ***ngFor***.

Por fim, execute o código e clique no botão "*Obter Produtos*" e observe o resultado no browser. Deve ser semelhante ao indicado abaixo:



Injetando services no componente

No projeto *Services* implementado acima, *productService* foi instanciado no componente diretamente, como mostrado abaixo:

```

constructor() {
  this.productService = new ProductService();
}

```

A instanciação direta do service, como mostrado acima, tem muitas desvantagens

1. O *ProductService* está firmemente acoplado ao componente. Se a definição da classe *ProductService* for alterada, será necessário atualizar todos os códigos em que o service é usado
2. Se, por exemplo, a classe *ProductService* tiver seu nome alterado para *NovoProductService*, será necessário procurar o nome "antigo" *ProductService* e, manualmente, alterá-lo
3. Acoplar a instaciação da classe *ProductService* diretamente no componenete faz com que os testes sejam mais difíceis para configuração e execução.

Esse problema pode ser resolvido por injeção de dependência.

Angular Dependency Injection

A Angular Dependency Injection agora é parte central do Angular e permite que as dependências sejam injetadas no componente ou classe. Neste tutorial, aprenderemos o que é Angular Dependency Injection e como injetar dependência em um componente ou classe usando um exemplo

O que é dependência

O service *ProductService* foi construido utilizando Angular Services. O *AppComponent* é dependente do *ProductService* para fornecer a lista de produtos que será exibido em tela para o usuário.

Em suma, o *AppComponent* tem uma dependência em *ProductService*.

Angular Dependency Injection -definição

Injeção de Dependência (DI) é uma técnica na qual é fornecida uma instância de um objeto para outro objeto, que depende dele. Essa técnica também é conhecida como "Inversão de controle" (IoC)

Observe o service – *ProductService* - criado anteriormente: *ProductService* retorna os produtos dispostos dentro do *array* quando o método *getProdutos* é chamado. Observe, novamente, a estrutura de código abaixo:

```
import { Produto } from './product';  
  
export class ProductService{  
  
    public getProdutos() {
```



```

    let produtos:Produto[];

    produtos = [
        new Produto(1,'Quadro Mestre Yoda',199),
        new Produto(2,'Mascara Darth Vader',159),
        new Produto(3,'Lightsaber',89)
    ]

    return produtos;
}

```

productService foi instaciado diretamente dentro do arquivo *app.component.ts*. Observe, novamente, o código abaixo:

```

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

import { FormsModule } from '@angular/forms';
import { ProductService } from '../product.service';
import { Produto } from '../product';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent{
  title: string = "angular-alpha" ;

  produtos!:Produto[];
  productService;

  constructor() {
    this.productService = new ProductService();
  }

  obterProdutos() {
    this.produtos = this.productService.getProdutos();
  }
}

```

A instância *ProductService* possui o escopo de “instância local” para o componente. *AppComponent*, agora, está fortemente acoplado ao ***ProductService***. Este acoplamento rígido pode trazer problemas – como visto no texto acima.

Idealmente, é necessário criar um service ***ProductServiceSingleton*** para que possa ser usado na aplicação.

Para minimizar os problemas enfrentados com o escopo local do *service* criado basta mover a instanciação de ***ProductService*** para o construtor da classe ***AppComponent***, como mostrado abaixo:

```
constructor(private productService:ProductService) {  
  
}
```

Agora, *AppComponent* não cria a instância do *ProductService*. Ele apenas “pede” em seu Construtor. *AppComponent* passa a ser dissociado do *ProductService*; “não sabe nada” sobre o *ProductService*. Apenas funciona com o *ProductService* passado para ele. Dessa forma, é possível passar qualquer alteração que seja feita em qualquer service, por exemplo: *ProductService*, *BetterProductService*, *MockProductService*.

O padrão acima é conhecido como ***Dependency Injection Pattern (injection pattern de dependência)***.

Importância de se usar injeção de dependência

O componente agora está fracamente acoplado ao *ProductService*. *AppComponent* “não sabe” como criar o *ProductService*.

AppComponent não depende mais de uma implementação particular *ProductService*. Funcionará com qualquer implementação *ProductService* transmitida a ele. Para, por exemplo, criar teste unitários basta, apenas, criar uma classe *mockProductService* e passá-la durante o teste.

A reutilização do componente fica mais fácil. O componente agora funcionará com qualquer **ProductService**, desde que a interface seja respeitada.

O dependency injection pattern tornou nosso **AppComponent** “testável” (passível de testes), e passível de manutenção, etc.

Ainda, nesse momento, o problema todo não foi resolvido -somente minimizado. O problema foi movido do *Component* para o Criador do *Component*.

Se o projeto for salvo e executado com a alteração feita acima, nada será exibido em tela.

Como criamos uma instância **ProductService** e a passamos para o **AppComponent**? É isso que **Angular Dependency Injection** faz.

Estrutura da Angular Dependency Injection

A estrutura de Angular Dependency Injection implementa o Injection pattern de Dependência em Angular. Ele cria e mantém as Dependências e as “injeta” nos Components ou Services que solicitam.

Partes da estrutura de Angular Dependency Injection

Existem cinco partes principais da estrutura de Angular Dependency Injection.

Consumer (Consumidor)

É o componente que precisa da dependência. No exemplo acima, o AppComponent é o Consumidor

Dependency (Dependência)

O serviço que está sendo “injetado”. No exemplo acima, *ProductService* é a Dependency

DI Token

O DI Token identifica exclusivamente uma Dependência. Usamos o DI Token quando registramos dependência

Provider (Fornecedor)

Os providers (fornecedores) mantêm a lista de dependências junto com seus tokens . O DI Token é usado para identificar a Dependência.

Injector

O *Injector* é o mantenedor dos *Providers* e é responsável por resolver as dependências e injetar a instância da *Dependency* (dependência) para o *Consumer* (Consumidor)

Como funciona a injeção de dependência no Angular

Observando o conceito de Inject, Injector e Injectable

As dependências são registradas com o *Provider*. Isso é feito nos *Providers* metadados do *Injector*.

O Angular fornece uma instância de *Injector* & *Provider* para todo *consumer* (consumidor).

O *consumer* (consumidor), quando instanciado, declara as dependências de que precisa em seu construtor.

O *Injector* lê as dependências do construtor do *consumer* (consumidor) e procura a dependência no *provider* (provedor). O *provider* (provedor) fornece a instância e o *injector* e injeta no *consumer* (consumidor). Se a instância da *Dependency* já existir, ela será reutilizada, tornando a dependência única.

Como usar a dependency Injection (injeção de dependência)

Nós criamos um simples ***ProductService*** em nosso último passo-a-passo. Vamos atualizá-lo agora para usar a *dependency injection* (Injeção de Dependência).

Primeiro, precisamos registrar as dependências no *provider*. Isso é feito no array de metadados *providers* do *decorator* `@Component`.

```
providers: [ProductService]
```

Observe o código abaixo. Implemente-o como se segue:

app.module.ts

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

import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';
import { AlterarTextoDirective } from './alterar-texto.directive';
import { ComponenteFilhoComponent } from './componente-filho/componente-filho.component';
import { ProductService } from './product.service';

@NgModule({
  declarations: [
    AppComponent,
    AlterarTextoDirective,
    ComponenteFilhoComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModuleModule,
    FormsModule
  ],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Assim, com o registro do service dentro do array providers, é possível referencia a *dependency injection* no construtor de *AppComponent*. Salve o projeto e execute. Veja o resultado no browser. Deve ser semelhante ao indicado abaixo:



Identificação do Produto	Nome do Produto	Preço do Produto
1	Quadro Mestre Yoda	199
2	Mascara Darth Vader	159
3	Lightsaber	89

Em seguida, precisamos informar angular que nosso componente precisa de “injeção” de dependência. Isso é feito usando o decorator **@Injectable()**.

O decorator **@Injectable()** não é necessário, se a classe já tem outros decorators. O Angular utiliza: **@Component**, **@pipe** ou **@directive**, entre outros. Porque todos estes são um subtipo de **Injectible**.

Como nosso **AppComponent** já está “decorado” com **@Component**, não precisamos “decorar” com o **@Injectable**

Nota: @Injectable também não é necessário se a classe não tiver nenhuma dependência a ser injetada. No entanto, a melhor prática é “decorar” todas as classes de service @Injectable(), mesmo aquelas que não têm dependências .

Obs.: Os services geralmente não são adicionados ao array Providers do componente, mas ao array Providers do @NgModule . Em seguida, eles estarão disponíveis para serem usados em todos os componentes no aplicativo. No entanto, é possível registrar qualquer service dentro do array providers de um componente. Veremos este uso a frente.

Quando **AppComponent** é instanciado, obtém sua própria instância **Injector**. O **Injector** “sabe” que o **AppComponent** faz a requisição de **ProductService** para o construtor. Na sequência, ele “olha” para o **Providers** para encontrar e, então, “prover”, uma instância de **ProductService** para o **AppComponent**

Observe, agora que o arquivo **app.component.ts** deve ser composto com o seguinte código:

app.component.ts

```
import { Component } from '@angular/core';  
  
import { FormsModule } from '@angular/forms';  
import { ProductService } from '../product.service';  
import { Produto } from '../product';
```

```

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title: string = "angular-alpha" ;

  produtos!: Produto[];

  constructor(private productService: ProductService) {

  }

  obterProdutos() {
    this.produtos = this.productService.getProdutos();
  }
}

```

Injetando Service em Service

Até este momento, foi observado como injetar *ProductService* a um componente. Agora, observe como injetar um service em outro service.

Crie um novo *service* e nomeie-o como *loggerService*, que registra todas as operações em uma janela do console e as injeta em *ProductService*. Para criar o novo service, abra o prompt de comando – via *vs code* ou pelo próprio computador – e, dentro da pasta de seu projeto (*angular-alpha*), insira a seguinte instrução:

```

ng g service nomedoservice
→ ng g service logger

```

```

CREATE src/app/logger.service.spec.ts (357 bytes)
CREATE src/app/logger.service.ts (135 bytes)

```

```

angular-alpha>ng g service logger
CREATE src/app/logger.service.spec.ts (357 bytes)
CREATE src/app/logger.service.ts (135 bytes)

```

Agora abra o novo arquivo service criado – *logger.service.ts* – e implemente o código abaixo como se segue:

logger.service.ts

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

@Injectable()
export class LoggerService {
  log(message:any) {
    console.log(message);
  }
}
```

A classe **LoggerService** possui apenas um método nomeado como *log()*, que captura o valor atribuído ao argumento *mensagem* e exibe no console.

Também está sendo usado o decorator, com os metadados, *@Injectable* para “decorar” a classe *LoggerService*. Tecnicamente, isso não é necessário aqui, pois a classe não possui nenhuma dependência externa. É uma boa prática adicionar metadados *@Injectable* pelos seguintes motivos:

- Provas futuras: Não é necessário lembrar *@Injectable()* quando adicionamos uma dependência posteriormente.
- Consistência: todos os services seguem as mesmas regras e não precisamos nos perguntar por que um decorator está ausente.

Inject na classe ProductService

Agora, o próximo passo é injetar a classe **LoggerService** na classe **ProductService**

ProductService necessita que *loggerService* seja injetado. Portanto, a classe requer metadados *@Injectable*. Observe a instrução abaixo e implemente-a como se segue – faça a importação, também, de *LoggerService* e *Injectable*:

```
import { Produto } from './product';
import { LoggerService } from './logger.service';
import { Injectable } from '@angular/core';

@Injectable()

export class ProductService{}
```

Na sequência, no construtor de **ProductService** é feita a solicitação de **loggerService**. Crie o construtor da classe e implemente o código abaixo como se segue:


```
// implementando o construtor
constructor(private loggerService: LoggerService) {
    this.loggerService.log("ProductService foi
    construido");
}
```

Altere o método **GetProdutos** para usar o **loggerService**.

```
public getProdutos() {

    this.loggerService.log("metodo getProdutos foi cham
    ado!");

    let produtos: Produto[];

    produtos = [
        new Produto(1, 'Quadro Mestre Yoda', 199),
        new Produto(2, 'Mascara Darth Vader', 159),
        new Produto(3, 'Lightsaber', 89)
    ]

    this.loggerService.log(produtos);

    return produtos;
}
```

Finalmente, será necessário registrar **LoggerService** com os metadados em **Providers**. Abra o *app.module.ts* para atualizar o providers e incula o registro de **LoggerService**. Observe o código abaixo e implemente-o como indicado:

app.module.ts

```
import { ProductService } from './product.service';
import { LoggerService } from './logger.service';

@NgModule({
    declarations: [
        AppComponent,
        AlterarTextoDirective,
        ComponenteFilhoComponent
    ],
    imports: [
        BrowserModule,
        AppRoutingModule,
        FormsModule
    ],
    providers: [
        ProductService,
        LoggerService
    ],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

```
],
providers: [ProductService, LoggerService],
bootstrap: [AppComponent]
}))
export class AppModule { }
```

Ao clicar no botão “Obter Produtos”, é possível observar pelo console do browser com as respectivas mensagens de log. Salve o projeto e observe o resultado. Deve ser semelhante ao indicado abaixo:



```
ProductService foi construido
Angular is running in development mode. Call enableProdMode() to enable production mode.
[WDS] Live Reloading enabled.
metodo getProdutos foi chamado!
▼ (3) [Produto, Produto, Produto]
  ► 0: Produto {idProduto: 1, nomeProduto: "Quadro Mestre Yoda", precoProduto: 199}
  ► 1: Produto {idProduto: 2, nomeProduto: "Mascara Darth Vader", precoProduto: 159}
  ► 2: Produto {idProduto: 3, nomeProduto: "Lightsaber", precoProduto: 89}
  length: 3
```

O fornecimento do *service* no módulo raiz cria uma instância única e compartilhada desse *service* e injetará em qualquer classe que solicitar.

O código acima funciona porque, o Angular, cria a Árvore dos Injetores com o relacionamento pai-filho semelhante à Árvore dos Componentes.

Obs.: Os serviços injetados no nível do módulo têm escopo no aplicativo, o que significa que eles podem ser acessados de todos os componentes / serviços do aplicativo. Qualquer serviço fornecido no **Módulo filho está disponível em todo o aplicativo.**

Os serviços são fornecidos em um módulo lento, com escopo definido no módulo e disponíveis apenas para o módulo carregado lento.

Os serviços fornecidos no nível do componente estão disponíveis apenas para o componente e para os componentes filho.

Onde você registra sua dependência, define o escopo da dependência. A dependência registrada no Módulo usando o decorator `@NgModule` é anexada ao Provedor Root (Provedor anexado ao `InjectorRoot`). Essa dependência está disponível para todo o aplicativo.

A dependência registrada com o componente está disponível para esse componente e para qualquer componente filho desse componente.

ProductService possui uma relação de dependência do ***LoggerService***. Por isso, é “decorado” com o decorator `@Injectable`. Remova `@Injectable()` de *ProductService* será indicado a seguinte exceção:

```
Uncaught Error: Can't resolve all parameters for ProductService: (?)
```

Isso porque, sem DI Angular não é possível saber como injetar ***LoggerService*** em ***ProductService***.

Remover `@Injectable()` de *LoggerService* não resultará em nenhum erro, pois *LoggerService* não há dependência.

Os *components* e *directives* já estão “decorados” com `@Component` & `@Directive`. Esses decorators também dizem ao Angular para usar o DI, portanto você não precisa adicionar o `@Injectable()`.

Referências

adaptado do original que pode ser obtido pelo link abaixo

Alexandre Beato: <https://medium.com/@alexandrebeato/pt-br-arquitetura-para-projeto-em-angular-com-f%C3%A1cil-desenvolvimento-e-manuten%C3%A7%C3%A3o-817a15e0d10c>
<https://blog.bitsrc.io/data-binding-in-angular-cbc433481cec>

Traduzido e adaptado do original que pode ser obtido através do link abaixo:

<https://www.c-sharpcorner.com/article/learn-angular-8-step-by-step-in-10-days-data-bindingday-3/>

<https://angular.io/>

<https://blog.angular-university.io/angular-ng-template-ng-container-ngtemplateoutlet/>

<https://www.tektutorialshub.com/angular/angular-services/>

<https://www.tektutorialshub.com/angular/angular-ngfor-directive/>

<https://www.tektutorialshub.com/angular/angular-ngswitch-directive/>

<https://www.tektutorialshub.com/angular/angular-ngif-directive/>

<https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/>

<https://www.tektutorialshub.com/angular/angular-providers/>

<https://www.freakyjolly.com/angular-material-modal-popup-example/>

<https://material.angular.io/components/dialog/examples>

<https://www.tektutorialshub.com.com/angular/angular-hierarchical-dependency-injection/>