

Crie aplicações com Angular

O novo framework do Google



ISBN

Impresso e PDF: 978-85-5519-270-8

EPUB: 978-85-5519-271-5

MOBI: 978-85-5519-272-2

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Nada na vida é por acaso. Como dizia Steve Jobs, "ligue os pontos" e acredite que as situações da vida vão se ligar em algum momento para que, no final, tudo faça sentido. Assim, você sempre terá coragem de seguir seu coração, mesmo que ele o leve para um caminho diferente do previsto. No final, tudo fará sentido na vida.

Se alguém falasse que um dia eu teria meu próprio livro, eu daria risada e falaria que esse era um dos meus desejos. Mas não sabia que isso seria verdade um dia. Sempre temos altos e baixos, e são nas dificuldades que aprendemos, nos direcionamos, e ficamos mais fortes.

Sempre gostei de ensinar e passar meu conhecimento para a frente. Acredito que essa é uma maneira de continuar vivo mesmo quando partimos, um livro, um ensinamento, uma dica, tudo o que possa melhorar a vida do próximo. Quem me conhece sabe a paixão que tenho em aprender e dividir conhecimento, afinal, ninguém é tão inteligente que não tenha nada para aprender, e ninguém é tão burro que não tenha a ensinar. E quem sabe esta paixão possa estar em uma sala de aula algum dia sendo um professor. Infelizmente, este momento ainda não chegou.

Hoje, estou realizando um dos meus objetivos na vida, que é ter um livro publicado. Este tema vem como um presente para mim, e poder falar dele é algo sensacional.

Agradeço a toda equipe da editora Casa do Código pela confiança e ajuda com a construção deste livro. Agradeço também por toda as orientações e feedbacks de conteúdo até a publicação.

Agradeço a minha companheira, Andressa Fernandes, por sempre estar ao meu lado, e por nossas conversas, brincadeiras, brigas, risos, passeios, e tudo que passamos juntos. Esteja certa que tudo que faço será pensando em nós dois.

Agradeço a toda a minha família, professores, pessoas e amigos. Bons ou ruins, todos têm uma influência sobre o que eu sou hoje.

Finalizo dizendo que, se você quer chegar onde a maioria não chega, faça o que a maioria não faz. Então, espero que estude muito, aprenda bastante e goste da leitura deste livro. Ele foi feito com esforço e dedicação. Espero que sirva como uma ótima referência para seu aprendizado e desenvolvimento profissional e pessoal.

Bons estudos!

SOBRE O AUTOR



Figura 1: Thiago Guedes

Sou Thiago Guedes, desenvolvedor Web e Mobile. Tenho conhecimento em desenvolvimento Android, Angular 2, Nodejs, Ionic 2, JavaScript, HTML5, CSS3, Java, C#, PHP, MySQL, Webservice RestFul, SOAP, metodologia ágil Scrum, XP e certificação Microsoft.

Minhas especialidades em Mobile são desenvolvimento nativo para Android e Híbridos com Ionic 2. Minhas especialidades em Web são desenvolvimento com Angular 2, HTML5, CSS3, JavaScript. Já em serviços, são com Node.js e Restful.

Atualmente, trabalho como analista desenvolvedor, na capital de São Paulo.

LinkedIn: <https://br.linkedin.com/in/thiago-guedes-7b0688100>

PREFÁCIO

Já pensou em desenvolver aplicações web de forma rápida e fácil usando todas as novidades da web moderna? Já pensou em aprender a usar um dos melhores frameworks front-end hoje do mercado, e ainda poder fazer aplicações web e mobile, tudo no mesmo lugar?

Com o Angular 2, você desenvolverá de forma fácil e rápida qualquer aplicação web que desejar, e ainda poderá expandir para plataforma mobile de forma simples. Neste livro, você vai aprender tudo o que precisa para iniciar seu desenvolvimento usando este framework feito pelo Google.

Veremos todas as novidades da web moderna, desde o novo JavaScript e as novas funções do ES6. Aprenderemos a usar Default Parameters, Arrow functions, destructuring, Map, Reduce, Filter e muito mais. Veremos como usar o TypeScript, desde a criação e declaração de variáveis até classes, funções `any` e `void`, passagens de parâmetros e parâmetros opcionais.

Para configurar todo o ambiente para desenvolvimento, instalaremos o Node.js e o NPM, e usaremos os comandos do NPM, o TypeScript. Vamos desenvolver aplicações Angular 2 usando o facilitador `angular-cli`.

Vamos explicar e mostrar cada parte que compõe uma aplicação Angular 2, explicando para que serve cada uma. Veremos o que são componentes e como interagi-los com HTML5 e com CSS3. Vamos aprender cada marcação do Angular 2 e quando utilizar cada uma delas, como diretivas, serviços, injeção

de dependências, bem como separar a aplicação em módulos.

No final, faremos um projeto usando tudo o que foi ensinado durante o livro e construiremos uma aplicação do zero. Vamos colocá-la em produção, subir um servidor e utilizar o projeto em nível de produção como sendo um usuário real.

Se você deseja criar aplicações para web e mobile, usando tudo o que tem de melhor para desenvolvimento, então você deve ler este livro. No final, você será capaz de realizar aplicações usando o Angular 2.

Os projetos que foram desenvolvidos durante o livro estão no meu GitHub.

Projeto feito durante o livro:
<https://github.com/thiagoguedes99/livro-Angular-2>

Projeto final:
<https://github.com/thiagoguedes99/projeto-final-livro-angular-2>

Conhecimentos necessários

Para desenvolver em Angular 2, de preferência você precisa saber sobre HTML5, CSS3, JavaScript, TypeScript. Como se trata de aplicações client-size ou híbridas, os conceitos front-end serão usados a todo momento.

Entretanto, se você não tem conhecimento sobre isso, fique tranquilo. Com o tempo e fazendo todos os exercícios apresentados neste livro, garanto que no final você estará desenvolvendo em Angular 2 muito bem, e fazendo sistemas muito

mais fácil do que com uma codificação tradicional.

Sumário

1 Introdução	1
1.1 O que é Angular 2	1
1.2 O que é SPA	3
1.3 As novidades do ES6	4
1.4 Default parameters	7
1.5 Arrow functions	7
1.6 Destructuring	8
1.7 Map	9
1.8 Filter	11
1.9 Reduce	12
1.10 TypeScript	13
1.11 Criação de variáveis	17
1.12 Declaração de variáveis	18
1.13 Classes	18
1.14 Declaração de funções e uso de any e void	19
1.15 Visibilidade em TypeScript	20
1.16 Parâmetros opcionais e com dois tipos	21
1.17 Resumo	23

2 Configuração do ambiente	24
2.1 Instalação do Node.js	24
2.2 Instalação do NPM	26
2.3 Instalação do TypeScript	28
2.4 Ambiente de desenvolvimento	29
2.5 Instalando o Angular CLI	31
2.6 Comandos do Angular CLI	32
2.7 Iniciando projeto em Angular 2	35
2.8 Resumo	40
3 Arquitetura do sistema e componentes do Angular 2	41
3.1 Construção do projeto em Angular 2 e seus arquivos	41
3.2 Como funciona o build no Angular 2	53
3.3 Partes do componente em Angular 2	55
3.4 Componente	56
3.5 Template	58
3.6 Metadata	59
3.7 Data Binding	64
3.8 Diretivas	72
3.9 Serviços	77
3.10 Injeção de dependência	81
3.11 NgModule	87
3.12 Melhorando nosso componente	94
3.13 Resumo	97
4 Exibindo dados para o usuário	99
4.1 Interpolation	100
4.2 Property binding	105

4.3 Two-way data binding	109
4.4 ngIf	113
4.5 ngSwitch	121
4.6 ngFor	126
4.7 Reaproveitamento de lista dentro do Angular 2	129
4.8 ngClass	135
4.9 ngStyle	144
4.10 ngContent	151
4.11 Resumo	158
5 Entrada de dados do usuário	160
5.1 Event binding	160
5.2 Variável de referência do template	165
5.3 (click)	166
5.4 (keyup)	167
5.5 (keyup.enter)	168
5.6 (blur)	170
5.7 @Input() property	174
5.8 @Output property	177
5.9 Resumo	182
6 Formulários	184
6.1 ngModel, variável de template e atributo name da tag HTML	
6.2 Validações de formulário	190 ¹⁸⁵
6.3 ngModelGroup	196
6.4 Enviando dados do formulário	199
6.5 Resumo	203

7 Injeção de dependências	205
7.1 O que é injeção de dependência	205
7.2 Instanciando serviços manualmente	206
7.3 Injetando uma dependência	212
7.4 Dependências opcionais	215
7.5 Declaração global no projeto ou dentro do componente	217
7.6 Resumo	223
8 Projeto final	225
8.1 Criação do novo projeto	227
8.2 Instalação do bootstrap	230
8.3 Arquitetura do projeto	231
8.4 Classe de modelo para os contatos	233
8.5 Classe de serviço e data-base	233
8.6 Componente de cadastro de contato	235
8.7 Componente de lista dos contatos	239
8.8 Componente de detalhe do contato	241
8.9 Fazendo build para produção	243
8.10 Resumo	245
9 Angular 4	247
9.1 Por que Angular 4?	249
9.2 Somente Angular	251
9.3 Atualizar projeto para Angular 4	252
9.4 Novas features	256
9.5 Resumo	261
10 Bibliografia	263

INTRODUÇÃO

1.1 O QUE É ANGULAR 2

Angular 2 é um framework para desenvolvimento front-end com HTML, CSS e TypeScript que, no final, é compilado para JavaScript. Com ele, você constrói aplicações juntando trechos de código HTML, que são chamados de templates — com suas tags customizadas e classes que são marcadas com `@component` para gerenciar o conteúdo que será exibido no template. O conjunto desses dois conceitos forma um componente em Angular 2.

Ao contrário do que algumas pessoas acham, o Angular 2 não é continuação do AngularJS, também conhecido como Angular 1. Angular 2 é um framework totalmente novo e remodelado, codificado do zero, com lições aprendidas do AngularJS.

Muitas coisas foram mudadas internamente, inclusive vários conceitos para desenvolvimento. O Angular 2 vem com cara nova, aproveitando o máximo da nova web (com todo o poder do HTML5), usando conceito de SPA (*Single Page Application*) e sendo base para aplicações mobiles. Com ele, podemos fazer desde simples páginas web até grandes sistemas e aplicações, aplicativos mobile com Ionic 2 — que tem como base de desenvolvimento o Angular 2.

O conhecimento desse framework do Google deixará você interagido com tudo o que há de novo no mundo do desenvolvimento web e mobile.

Uma das principais mudanças do AngularJS para o Angular 2 é a performance. Enquanto o AngularJS era muito mais lento em relação aos outros frameworks front-end devido a várias interações feitas no DOM, o Angular 2, com sua nova configuração, deixa a resposta muito mais rápida e a usabilidade muito mais dinâmica.

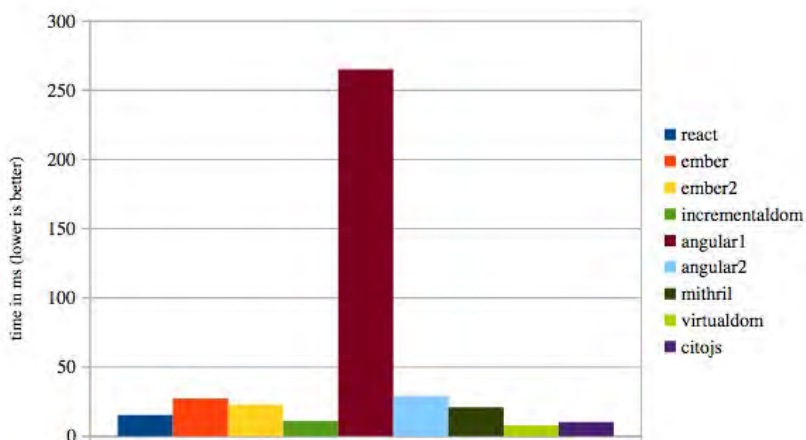


Figura 1.1: Benchmark Framework front-end. Fonte: Peyrott (2016)

O desenvolvimento com Angular 2 é feito por meio de codificação TypeScript, que é um superset para JavaScript. Além de implementar funcionalidades do ES6, traz uma série de poderes ao desenvolvedor com as tipagens nas variáveis, e uma sintaxe mais clara e fácil de entender, parecendo-se bastante com C# ou Java.

Se você quiser fazer desenvolvimento web ou mobile, Angular

2 é o caminho certo para você aprender e desenvolver tudo o que deseja.

1.2 O QUE É SPA

Com a evolução do JavaScript e também muitas das grandes empresas desenvolvendo padrões e técnicas para melhorar a performance dos scripts, chegamos a um padrão de desenvolvimento chamado de **Single Page Applications** (ou **SPA**). As *SPAs* são aplicações desenvolvidas em JavaScript que rodam quase inteiras no lado do cliente (browser). Assim que o usuário acessa o site, a aplicação fica armazenada do lado do cliente em forma de templates (pedaços de HTML).

As aplicações feitas em SPA fazem uma transição entre templates que estão dentro do browser, e só fazem requisições no servidor para buscar dados brutos de conteúdo que são enviados normalmente em JSON. O Google foi o primeiro nesta tecnologia com o *Gmail*, e hoje temos uma infinidade de sites que usam esse tipo de padrão.

Com uma aplicação desenvolvida em SPA, temos muitas vantagens, como melhor experiência do usuário, performance e menor carga de trabalho no servidor. A experiência do usuário é bastante melhorada com uma aplicação em SPA, dando a impressão até de ser uma aplicação desktop. Não temos reloads nem carregamento da página inteira, são apenas trechos da página que são mudados dependendo da ação do usuário, fazendo uma conexão e transferência do servidor para o cliente muito mais leve e rápida.

Quanto à performance, não temos como discordar. Uma aplicação tradicional carrega uma página inteira do servidor em toda interação com o usuário, gerando mais tráfego na rede, mais processos no servidor e mais processos no navegador.

Em uma aplicação em SPA, tudo fica mais rápido e performático. Quando o usuário inicia a aplicação, quase todo o HTML é enviado para o navegador. As transições entre as partes da aplicação não precisam ser montadas e enviadas do servidor. O que o servidor envia para o navegador do usuário são somente os dados do processo em formato JSON.

O que temos na verdade com o padrão SPA é um `hidden e show` de partes do HTML, formando uma página apresentável para o cliente. Isso torna todo o processo muito mais rápido, leve e com performance elevada.

Com o servidor não precisando montar todo o HTML para enviar ao cliente, ele só precisa receber a requisição, processar e montar um JSON, e enviar como resposta. O resultado disso é que o serviço fica com menos processos, o tráfego entre server e cliente fica mais leve, e tudo fica mais rápido.

1.3 AS NOVIDADES DO ES6

ES6, ECMAScript 6 ou ES2015? Uma dúvida bem comum é por que a mudança de nome, mas na verdade não houve mudança. JavaScript como conhecemos é somente um dialeto/apelido para a linguagem ECMAScript, abreviada carinhosamente de ES.

Quando falamos ECMAScript ou JavaScript, ambas tratam das evoluções constantes que a linguagem vem tendo durante os anos.

Após um longo tempo sem atualizações, o comitê responsável conhecido como TC39, definido pela norma ECMA-262, decidiu fazer um release anual. Ou seja, cada edição é um superset da edição anterior, implementando novas funcionalidades. Em 2015, o ECMAScript ou JavaScript (chame como quiser) chegou à sua 6ª versão, o ES6, trazendo muitas melhorias e resolvendo problemas antigos em relação a versões anteriores.

A primeira mudança é uma com que desenvolvedores já sofreram bastante: o uso do `var` para declaração de variável. Quem já programa em JavaScript sabe que a variável tinha escopo global no projeto, e no código com o ES6 temos o uso do `let` para declaração de variáveis. Esta é a nova forma, fazendo a variável como escopo de bloco. Isso ajuda bastante para pegar erros como uma declaração dentro de um bloco `if`, `for`, `while` ou outro qualquer, pois a variável morre quando o bloco termina.

```
var qualquerCoisa = true;

if (qualquerCoisa) {
  let nomeVariavel = 'Angular 2';
  alert(nomeVariavel);
}

console.log(nomeVariavel); // nomeVariavel is not defined
console.log(qualquerCoisa); // printa no console True;
```

Vemos nesse pequeno exemplo o uso do `let` em ação. Enquanto a variável com `var` está sempre ativa, a com `let` só está ativa dentro do bloco onde realmente ela será usada, não fazendo sentido algum continuar com ela fora do bloco `if`. Em resumo, o `let` conserta um antigo problema de declaração de variáveis, fazendo com que elas funcionem da forma esperada

pelos desenvolvedores.

Outra mudança introduzida no ES6 é o uso do `const`. Ele funciona igual como em outras linguagens de programação. Quando a variável é criada, não pode ser alterada, e o valor inicial declarado será sempre o mesmo. Caso se tente atribuir um novo valor, será lançado um erro de **Uncaught SyntaxError: read-only**.

Algumas alterações foram feitas em relação aos parâmetros de funções, que trazem grandes benefícios e agilidade na codificação, tanto no ES5 (versão anterior) quanto no ES6. Mais para a frente, veremos as modificações em TypeScript.

Na versão anterior do ES6, o ES5, já tivemos algumas mudanças na parametrização de funções que deram uma grande ajuda no código do programador. Veja o exemplo a seguir:

```
function somar(x, y) {  
    return x + y;  
}  
  
console.log("minha soma é: " + somar(5)); // saída será NaN
```

Esse código dará um erro de *NaN* (não é número), pois o valor de `y` é *undefined*, sendo o programador que tratará todas essas exceções em seu código, deixando a função cheia de linhas desnecessárias. No ES5, já podemos tratar isso de forma bem simples. Veja:

```
function somar(x, y) {  
    y = y | 1;  
    return x + y;  
}  
  
console.log("minha soma é: " + somar(5)); // saída será 6
```

Rodando esse código, você verá que a soma é 6. O pipe (guarde

bem esse nome, pois vamos falar dele em Angular 2 também!) faz uma função de `if` nesse lugar. Ou seja, caso o valor de `y` seja *undefined*, o `y` será 1. Com essa linha, nossa função não dará mais erro.

Mas, mesmo assim, temos uma linha que não faz parte da lógica da função e só serve como verificação. Então, surge a evolução do ES6: os **default parameters**.

1.4 DEFAULT PARAMETERS

Parâmetros padrão são colocados na declaração da função, deixando todo o bloco livre para colocar somente a lógica do problema.

```
function somar(x, y = 1) {  
    return x + y;  
}  
  
console.log("minha soma é: " + somar(5)); // saída será 6
```

Veja que nossa função volta a ter somente uma linha, e somente a lógica do problema está dentro do bloco. O próprio ES6 já faz toda a verificação de valores da variável na declaração. Ótimo, né? Mas acha que isso é tudo? Então, sente aí que ainda vêm mais.

Nossa função tem 3 linhas ao todo: uma na declaração, uma da lógica envolvida e uma para fechamento. E se fosse possível fazer isso tudo em apenas uma linha? Mágica? Não, é apenas uma evolução da linguagem. Apresentarei os **arrows functions**.

1.5 ARROW FUNCTIONS

É preciso entender bem esse conceito, pois, em Angular 2, muita coisa é escrita dessa forma. Veja como ficará nosso código:

```
const somar = (x, y = 1) => x + y;

console.log("minha some é: " + somar(5)); // saída será 6
```

Primeiro, declaramos uma constante com o nome da função. Com isso, já podemos retirar a palavra `function`. Em seguida, retiramos os `{}` e o `return`, trocamos os dois por esse símbolo `=>`, que significa que o valor a seguir será retornado. Simples, né?

Mas calma, com a evolução do Angular 2 e o TypeScript, isso fica menor ainda. Continue a leitura que você verá.

1.6 DESTRUCTURING

Você que programa em C#, Java, PHP, JavaScript, ou alguma outra linguagem do mercado de softwares, já se deparou com a retirada de valores de array ou objetos em JSON? Em ambos os casos, temos de, ou declarar a posição do array cujo valor queremos obter, ou iterar dentro do JSON com o nome da chave para recuperar o valor. Correto?

Entretanto, no ES6, temos uma forma muito mais simplificada para recuperar valores: usamos o **destructuring** (ou em português, **desestruturação**). Veja só:

```
meuArray = [1, 2, 3];

const [a, b, c] = meuArray;

console.log(a); // saída será 1
console.log(b); // saída será 2
console.log(c); // saída será 3
```

No `meuArray` , temos 3 posições. No ES6, podemos declarar um novo array ou novas variáveis já atribuindo os valores do `meuArray` . A variável `a` recebe o valor de `meuArray[0]` , a variável `b` recebe o valor de `meuArray[1]` , e assim sucessivamente. Simples, certo?

Mas e com objetos? Também é fácil.

```
meuObjeto = {livro:'Angular 2', autor:'thiago guedes'};

const{livro, autor} = meuObjeto;

console.log(livro); // saída será Angular 2
console.log(autor); // saída será thiago guedes
```

No JSON `meuObjeto` , temos dois valores: `livro` e `autor` . Podemos recuperar os valores e atribuí-los em novas variáveis de maneira muito simples. Isso serve para qualquer tipo de objeto e é muito útil quando se recebe um Json em uma requisição de um web service.

Para manipulação e interação de array, o ES6 introduziu novas funções para facilitar muito a vida do programador. Isso será constantemente usado quando formos desenvolver em Angular 2, ou em qualquer outro framework para JavaScript.

Em algum momento, todo programador já teve que fazer interação com array, manipulação nas posições, filtros de acordo com determinado pedido, gerar um novo array com base no primeiro etc. Vamos verificar como podemos fazer isso no ES6, com `map` , `filter` e `reduce` .

1.7 MAP

O método `map` interage com cada elemento de um array, chamando uma função de callback e executando determinada lógica descrita no callback. No final, o método retorna um novo array com as modificações.

Vamos considerar que queremos multiplicar cada elemento de um array por 2. Com o método `map`, isso fica fácil de fazer.

```
arrayNumeros = [2, 3, 4, 5, 6];

novoValor = arrayNumeros.map(function(num){
  return num * 2;
});

console.log(novoValor); // saída será [4, 6, 8, 10, 12]
```

Temos um array chamado `arrayNumeros`. Com o método `map`, passamos uma função que vai fazer a interação e modificação em cada elemento do array original. O método `map` retorna um novo array chamado `novoValor`. No começo, pode parecer complexo, mas não se assuste, logo, com prática, ficará fácil de fazer.

Porém, aqui eu não gostei de uma coisa: essa função de callback está muito grande e ocupa muito espaço. Que tal deixá-la pequena? Como? Com os *arrow functions* que acabamos de aprender.

```
arrayNumeros = [2, 3, 4, 5, 6];

novoValor = arrayNumeros.map((num) => num * 2);

console.log(novoValor); // saída será [4, 6, 8, 10, 12]
```

Em uma linha, fizemos todo o bloco de função; ficou elegante e com o mesmo resultado. Muito melhor, né?

Funções `map` podem fazer muitas coisas. O que usei aqui foi um exemplo simples, mas a ideia é **sempre que precisar fazer manipulação com cada elemento de um array, use o método `map`**.

1.8 FILTER

O método `filter`, na minha opinião, é o mais legal de todos. Imagine que você precisa saber quais elementos satisfazem determinado contexto dentro de um array. Por exemplo, dado um array de nomes, quais nomes têm a letra `'a'`?

Como você faria isso? Com `substring`? Com `indexOf`? Não, com o `filter`.

```
nomes = ['jorge', 'carlos', 'roberto', 'lucas'];

novoNome = nomes.filter(function(nome){
    return nome.includes('a');
});

console.log(novoNome); // saída será "carlos" e "lucas"
```

Repare aqui que usamos um novo método, o `includes`. Ele se parece com o `indexOf`, mas com `includes` não precisamos fazer uma validação. Automaticamente ele já retorna `true` ou `false`.

O `filter` interage com cada elemento do array, executando a função de callback. A cada retorno `true` da função de callback, esse elemento é filtrado para o array `novoNome`.

E temos mais? Sim, podemos fazer com *arrow function*.

```
nomes = ['jorge', 'carlos', 'roberto', 'lucas'];
```



```
novoNome = nomes.filter((nome) => nome.includes('a'));  
  
console.log(novoNome); // saída será "carlos" e "lucas"
```

Bonito, né? ES6 com arrow function reduzem bastante o código. O programador só precisa pensar na lógica do problema, pois o resto a linguagem faz.

1.9 REDUCE

O método `reduce` basicamente junta todos os elementos em um só. Isso mesmo, o `reduce` reduz todo o array para um elemento ou uma string. Vejamos o exemplo.

Primeiro, vamos juntar todos os nomes em uma string, separando por `'-'`.

```
nomes = ['jorge', 'carlos', 'roberto', 'lucas'];  
  
novoNome = nomes.reduce((acumulado, nome) => acumulado + '-' + nome);  
  
console.log(novoNome); // saída será: jorge-carlos-roberto-lucas
```

Usei *arrow function* direto, pois agora você já sabe como funciona. Veja que o método `reduce` está com 2 parâmetros. Podemos passar até 4 parâmetros, mas 2 são obrigatórios. O primeiro é o valor atual, o segundo o valor corrente. Em valor atual (acumulado), está contida toda a interação até aquele momento; e o valor corrente (nome) é a posição do elemento no loop atual.

Complicou? Fácil, o valor corrente é a interação com cada elemento, e faz determinada lógica dentro do callback. O resultado vai para o primeiro parâmetro (valor atual), assim, no final da

interação com todos os elementos, o retornado é o primeiro parâmetro.

Vamos agora com um exemplo de soma de valores:

```
valores = [10, 20, 30, 40, 50];  
  
novoValor = valores.reduce((acumulado, corrente) => acumulado + c  
orrente);  
  
console.log(novoValor); // saída será: 150
```

Nesse exemplo, estamos usando o `reduce` para somar os valores de cada elemento. Veja que o valor acumulado vai sendo somado como cada elemento corrente.

Os métodos `reduce`, `filter` e `map` são apenas algumas melhorias feitas no ES6 durante o tempo. Claro que temos outras coisas novas na linguagem, mas, por enquanto, isto é o suficiente. A seguir, veremos outras novidades introduzidas no ES6: classes, heranças, setters e getters.

1.10 TYPESCRIPT

Quem já programa em JavaScript sabe da dificuldade que era fazer códigos reaproveitáveis usando conceitos de classe. Tínhamos de codificar usando *prototypes*, e isso gerava muitos problemas e margens de erros grandes, fora todo o trabalho e dor de cabeça para fazer isso.

Com a entrada do ES6, isso já ajudou bastante com os conceitos de classe herança e muito mais. Porém, para muitos programadores, uma coisa que sempre deixou o JavaScript um pouco bagunçado era o fato de a linguagem não ter características

nas variáveis funções. Por ser uma linguagem dinâmica, a variável em JavaScript ora pode ser uma `string`, ora pode ser um `number`, dependendo do seu valor contido. Isso gera uma certa confusão quando fazemos manutenção, ou até na construção de softwares.

```
let meuNome = "Thiago guedes";

console.log(meuNome); // saída será: Thiago guedes

meuNome = 15;

console.log(meuNome); // saída será: 15
```

A variável `meuNome` pode ser tanto uma `string` quanto um número, mas é em cima desse problema que o TypeScript foi criado, trazendo um conceito de tipagem igual ao já usado em outras linguagens de programação.

O próprio site do TypeScript fala que *"TypeScript é uma linguagem para desenvolvimento JavaScript em larga escala. Com TypeScript, podemos escrever códigos utilizando estruturas fortemente tipadas e ter o código compilado para JavaScript"*.

Simplificando, TypeScript é um *superset*, um superconjunto JavaScript. Isso significa que você escreve código na forma na qual é acostumado quando codifica com Orientação a Objetos. O TypeScript ainda acrescenta uma variedade de sintaxe útil, e ferramentas sobre a linguagem JavaScript. Ele traz o poder e a produtividade da tipagem estática e técnicas de desenvolvimento orientadas a objetos. No final, o código em TypeScript é compilado para JavaScript e fica totalmente compatível e entendível pelos browsers.

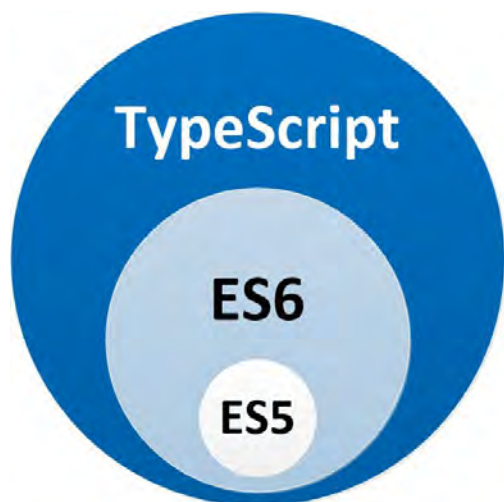


Figura 1.2: TypeScript engloba tudo de novo do ES5 e adiciona as tipagens

Achou isso ruim? Acha que perde o poder dinâmico do JavaScript? Caso queira, é possível programar normalmente em JavaScript tradicional, sem tipagens. Mas olhe, vou lhe mostrar como o TypeScript ajudará muito sua vida.

Segue uma simples função que vai escrever um nome no console do navegador, com TypeScript.

```
let meuNome = "Thiago guedes";

function dizOla(nome: string){
    console.log("Olá " + nome); // saída será: Olá Thiago guedes
}

dizOla(meuNome);
```

Essa função é muito parecida com a anterior. Veja que, na declaração da função, foi colocado `: string`. Isso é parte do TypeScript já tipando a variável `nome` como `string`. Rodando

essa função, a saída será `Thiago guedes` . Mas e se eu colocar um número, será que vai? Vamos verificar.

```
let meuNome = 15;

function dizOla(nome: string){
    console.log("Olá " + nome); // Argument of type 'number' is
    not assignable to parameter of type 'string'.
}

dizOla(meuNome);
```

Olha aí o TypeScript em ação. Na hora da compilação, ele verifica que a função espera uma `string` e o que foi passado é um `number` . Assim ele reclama e dá um erro de **Argument of type 'number' is not assignable to parameter of type 'string'**.

Ótimo! Menos uma coisa para eu me preocupar em validar. Simplesmente coloco o tipo esperado da função e o TypeScript faz a verificação toda.

Mas quando compilar? O que vai acontecer com o código? Será que o tipo do parâmetro vai junto? Vamos ver isso agora. Mostrarei o código depois de compilado, do TypeScript para o JavaScript.

```
let meuNome = "Thiago guedes";

function dizOla(nome){
    console.log("Olá " + nome); // saída será: Olá Thiago guedes
}

dizOla(meuNome);
```

Ué, cadê a tipagem na função? Como disse anteriormente, o TypeScript é um superset, é uma máscara, um escudo para desenvolver JavaScript colocando tipos. Na verdade, quando o

compilador traduz o TypeScript para código JavaScript, ele retira tudo que é TypeScript e transforma para um JavaScript entendível pelo browser.

A função do TypeScript é facilitar no desenvolvimento e fazer uma camada até o JavaScript, de forma a diminuir os erros de codificação. Após a compilação, tudo vira JavaScript e será rodado no navegador.

1.11 CRIAÇÃO DE VARIÁVEIS

No TypeScript, temos tipos de declaração para todas as variáveis, desde tipos simples (que vão de *string*, *números*, *array*), até objetos complexos (*interfaces*). Da mesma forma que declaramos tipos em C# ou Java, declaramos em TypeScript.

Quando vamos declarar uma variável, temos três partes principais:

```
let nome_da_variavel: TIPO = valor;
```

`let` — declaração da variável no arquivo JavaScript;
`nome_da_variavel` — nome de referência da variável;
`: TIPO` — tipo que essa variável armazena;
`valor` — valor inicial da variável,

Repare que, entre `nome_da_variavel` e `TIPO`, temos de colocar os `:` (dois pontos), pois ele que vai declarar o tipo para a variável. O valor inicial é opcional, assim como em qualquer outra linguagem.

1.12 DECLARAÇÃO DE VARIÁVEIS

Podemos declarar variáveis com seus respectivos tipos de forma simples e já conhecida de outras linguagens. Perceba que constantemente estou falando *de forma parecida com outras linguagens*, e é para isso que serve o TypeScript. Ele foi criado para deixar a forma de codificação similar às codificações tradicionais.

Para declaração de tipos primitivos, temos:

```
let nome: string = "Thiago guedes";
let ano: number = 2017;
let desenvolvedor: boolean = true;
```

É simples, basta colocar `: tipo` entre o *nome da variável* e o *valor dela*.

Para declaração de *array*, vamos fazer assim:

```
let lista: number[] = [1, 2, 3, 4];
```

Quer usar *generics*? Ok, o TypeScript tem suporte.

```
let lista: Array<number> = [1, 2, 3, 4];
```

Fácil, simples, rápido e prático. Isso é TypeScript.

1.13 CLASSES

Classes é um conceito que também temos em ES6, mas prefiro falar dele agora, pois, no desenvolvimento de Angular 2, usaremos TypeScript e não codificaremos em JavaScript direto.

As classes seguem o mesmo padrão de facilidade que as variáveis, bem parecido com o tradicional:

```
class Pessoa{
    nome: string;

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

    dizOlá():string{
        return "Olá, " + this.nome;
    }
}
```

Para declarar a classe `Pessoa` no projeto, faremos isso:

```
var thiago = new Pessoa("Thiago");
```

Será que é difícil fazer um *array* de classes do tipo `Pessoa` ? Não, meu caro leitor, isso é fácil; isso é TypeScript.

```
var pessoas: Pessoa[] = new Array();
```

```
pessoas.push(thiago);
```

1.14 DECLARAÇÃO DE FUNÇÕES E USO DE ANY E VOID

Da mesma forma como declaramos nas `functions` o tipo de uma variável no TypeScript, seja ela `string` , `number` , `boolean` ou `array` , também podemos declarar qual será o tipo de variável que ela vai retornar no final do seu processo. Isso facilita bastante para o programador quando for usar esta `function` .

```
dizOlá(): string{
    return "Olá, " + this.nome;
}
```

Essa `function` tem como retorno uma `string` . Veja que é a mesma função que está na classe `Pessoa` . Volte e veja como ela já

estava declarada como retorno `string` .

Da mesma forma como podemos declarar o tipo de variável, ou o tipo de retorno de uma função, também podemos declarar que uma variável aceita qualquer coisa ou uma função não retorna nada. Fazemos isso usando `any` e `void` .

```
var lista:any[] = [1, true, "angular 2"];
```

A lista foi declarada como `: any` , ou seja, ela aceita qualquer coisa. Fica muito parecido com o JavaScript tradicional. Para declarar a função que não retorna nada, declararemos como `: void` .

```
function logConsole(): void {  
    console.log("simples log no console");  
}
```

Podemos também não colocar o tipo da variável ou da função. O TypeScript vai considerar tudo como `any` . Entretanto, isso não é boa prática. Para o código ficar no padrão, vamos sempre colocar os tipos das variáveis e das funções.

1.15 VISIBILIDADE EM TYPESCRIPT

Da mesma forma que linguagens tipadas têm padrões de visibilidade de métodos, variáveis e propriedades, o TypeScript usa sua forma de nível de visibilidade das declarações. Podemos declarar a variável `nome` dentro da classe `Pessoa` , apresentada anteriormente, com uma visibilidade privada. Assim, ela só poderá ser vista dentro desta classe.

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

```

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

    dizOlá():string{
        return "Olá, " + this.nome;
    }
}

```

Veja que é tudo muito parecido, ou até igual, com o que estamos acostumados a fazer em linguagens fortemente tipadas.

1.16 PARÂMETROS OPCIONAIS E COM DOIS TIPOS

Podemos declarar os parâmetros da função de diferentes formas de tipos. Mas quando o parâmetro pode ou não ser passado? Ou quando ele pode ser uma `string` ou um `number`? Como fazer isso? Vamos ver agora.

Essa função é simples, porém mostra como usar os **parâmetros opcionais**. Ela vai receber duas `string`: `primeiroNome` e `segundoNome`, sendo que o `segundoNome` é opcional.

```

function juntarNome(nome: string, sobrenome?: string): string{

    if(sobrenome){
        return nome + " " + sobrenome;
    }
    else{
        return nome;
    }
}

```

Aqui, o sinal de `?` (interrogação) mostra que o segundo parâmetro é opcional. Quando a função for chamada com somente um parâmetro, ela vai funcionar normalmente; se for passado os

dois parâmetros, ela vai concatenar o nome e sobrenome .

Vamos para outro exemplo prático. Se eu quiser imprimir no console uma informação, esta pode ser uma string ou um number ? Podemos declarar o parâmetro com dois tipos.

```
function logConsole(log: (string | number)){  
    console.log(log);  
}
```

O **pipe** (olha o pipe aí novamente) deixará ser passado tanto uma string quanto um number para a função logConsole . Da mesma forma como passamos string e number , podemos declarar qualquer outro tipo:

```
function logConsole(log: (string | number), conteudo: (number | a  
ny[])){  
    console.log(log);  
    console.log(conteudo);  
}
```

Modificamos a função logConsole , e agora temos dois parâmetros, log e conteudo . Podemos tipar os parâmetros com qualquer tipo dentro do TypeScript na compilação, tudo isso será retirado deixando somente o conteúdo JavaScript para rodar no navegador.

Esses conceitos passados, tanto de ES6 e TypeScript, já mostram a evolução do desenvolvimento front-end atual. Mas isso tudo foi só a introdução para o que vem de melhor, que é o desenvolvimento em **Angular 2**.

Chega de tanta teoria, vamos colocar a mão na massa e começar a construir nossa SPA com Angular 2. No próximo capítulo, vamos verificar o que precisamos instalar para

desenvolvimento, e deixar nossa máquina prontinha para as próximas partes de codificação.

1.17 RESUMO

Neste capítulo, fizemos uma apresentação do que é Angular 2, o que é SPA, as novidades do ES6, com todas suas novas funções de *default parameters*, *arrow functions*, *destructuring*, *map*, *filter* e *reduce*. Apresentamos o TypeScript, mostramos para que ele serve e o que vai facilitar no desenvolvimento com Angular 2.

Mostramos como *criar variáveis*, *declarar variáveis*, como usar *classes*, e como e quando usar *any* e *void*. Aprendemos a declarar *visibilidade em TypeScript*, e a usar *parâmetros opcionais e com dois tipos*. Tudo isso será de grande ajuda para o desenvolvimento de aplicações com Angular 2.

CONFIGURAÇÃO DO AMBIENTE

2.1 INSTALAÇÃO DO NODE.JS

Esta é a afirmação que você constantemente vai ver: *"Node.js é uma plataforma para construir aplicações web escaláveis de alta performance usando JavaScript"*. O Node.js utiliza o motor JavaScript V8 do Google, o mesmo usado no navegador *Chrome*. É um ultra-rápido interpretador JavaScript escrito em C++, que compila e executa os arquivos JavaScript, manipula a alocação de memória, e usa um *garbage collector* preciso e eficiente, dando velocidade na execução do arquivo.

Precisamos da instalação do Node.js, pois o Angular 2 roda em cima dessa plataforma. Também vamos usar comandos *NPM* que serão abordados mais à frente. Esse também tem dependências da plataforma Node.js. Para fazer a instalação do Node.js pelo site <https://nodejs.org/>, é bastante simples e rápido.

Entrando no site, a primeira página já mostra uma breve descrição sobre o Node.js e dois botões para download. Do lado esquerdo, temos o download recomendado; este já foi testado e está em produção. Do lado direito, fica o download da próxima

versão do Node.js; esta pode conter *bugs* ou não rodar conforme o esperado.

Vamos baixar a versão que já está confiável para não termos problemas futuros. Usaremos o botão do lado esquerdo e seguiremos os passos de `next`, `next`, `next`, `next`, `finish`. Não tem segredo.

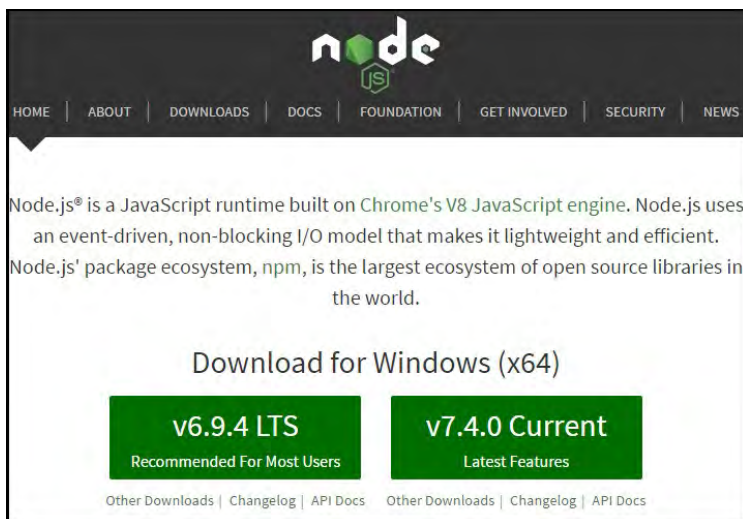


Figura 2.1: Página oficial do site Node.js

Agora abra o prompt de comando, e digite `node -version` ou `node -v`, para ver se a instalação foi concluída e a versão instalada.

```
E:\>node -v
v6.9.5

E:\>
```

Figura 2.2: Versão do Node.js instalada no computador

2.2 INSTALAÇÃO DO NPM

O NPM é distribuído com Node.js. Se você fez a instalação anterior do Node.js com sucesso, o NPM já está na sua máquina. Para confirmar isso, digite no prompt de comando `npm -v` para ver a versão atual.

```
E:\>npm -v
4.2.0

E:\>
```

Figura 2.3: Versão do NPM instalada no computador

A sigla NPM significa *node package manager* (**gerenciador de pacotes do node**). Ele é usado como um repositório para publicação de projetos com código aberto. Interagimos com ele por meio de linha de comando no console do computador para fazer instalação de pacotes e gerenciamento de dependências do projeto.

estava declarada como retorno `string` .

Da mesma forma como podemos declarar o tipo de variável, ou o tipo de retorno de uma função, também podemos declarar que uma variável aceita qualquer coisa ou uma função não retorna nada. Fazemos isso usando `any` e `void` .

```
var lista:any[] = [1, true, "angular 2"];
```

A lista foi declarada como `: any` , ou seja, ela aceita qualquer coisa. Fica muito parecido com o JavaScript tradicional. Para declarar a função que não retorna nada, declararemos como `: void` .

```
function logConsole(): void {  
    console.log("simples log no console");  
}
```

Podemos também não colocar o tipo da variável ou da função. O TypeScript vai considerar tudo como `any` . Entretanto, isso não é boa prática. Para o código ficar no padrão, vamos sempre colocar os tipos das variáveis e das funções.

1.15 VISIBILIDADE EM TYPESCRIPT

Da mesma forma que linguagens tipadas têm padrões de visibilidade de métodos, variáveis e propriedades, o TypeScript usa sua forma de nível de visibilidade das declarações. Podemos declarar a variável `nome` dentro da classe `Pessoa` , apresentada anteriormente, com uma visibilidade privada. Assim, ela só poderá ser vista dentro desta classe.

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


precisa.

Ao longo do livro, vamos criar novos projetos e novas declarações dentro do `package.json`. Este início é uma apresentação das partes e comandos para o desenvolvimento do Angular 2.

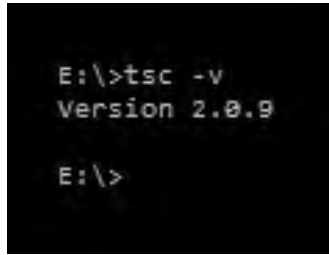
2.3 INSTALAÇÃO DO TYPESCRIPT

Esse já é um velho conhecido nosso. Estudamos bastante o TypeScript e vimos o que ele beneficia na codificação de JavaScript. Para fazer um resumo, poderia dizer que o TypeScript é um superset do JavaScript. Ele inclui todas as funcionalidades do JavaScript e adiciona funcionalidades extras para facilitar o desenvolvimento.

Para fazer a instalação do TypeScript, é ainda mais fácil do que o Node.js. Abra o prompt de comando e digite `npm install -g typescript`, e pronto! O gerenciador de pacotes (NPM) vai ao repositório e baixa a versão em produção do TypeScript.

Mas o que é aquele `-g` ali no meio? Isso diz para o sistema que queremos instalar o TypeScript de forma global dentro do computador. Ou seja, a qualquer momento que você digitar algum comando que faz parte do *TypeScript*, o sistema vai reconhecer e responder o pedido.

Para verificar se a instalação foi feita, ou ver qual versão está instalada, seguimos o mesmo passo de comando feito para o *Node.js* e para o *NPM*. Vamos digitar no prompt de comando `tsc -v`, e logo abaixo aparecerá a versão instalada.

A screenshot of a terminal window with a black background and green text. The text shows the command 'tsc -v' being executed, resulting in the output 'Version 2.0.9'. The prompt 'E:\>' is visible at the top and bottom of the terminal output.

```
E:\>tsc -v
Version 2.0.9

E:\>
```

Figura 2.4: Versão do TypeScript instalada no computador

2.4 AMBIENTE DE DESENVOLVIMENTO

Essa parte é muito contraditória. Não temos uma definição de qual ambiente de desenvolvimento usar, ou qual é o melhor ou tem mais desempenho. Isso é relativo para cada pessoa e fica muito no gosto pessoal.

Para a continuação do livro, vou escolher usar o **Visual Studio Code**. Na minha opinião (e você pode discordar tranquilamente disso), é o que tem melhor suporte para TypeScript, e tem um *autocomplete* mais informativo. Além disso, ele vem integrado com um *prompt de comando*, assim não precisamos ficar trocando de tela toda hora quando quisermos fazer um comando do *NPM*. Também tem uma maior interação com *classes* e *interfaces*, podendo clicar em cima delas e automaticamente ser redirecionado para o documento raiz.

São por esses detalhes que escolho o **Visual Studio Code** para desenvolvimento. Mas fique à vontade para usar Sublime, Brackets, WebStorm, Atom, ou outro qualquer ambiente que achar melhor. Como falei anteriormente, isso vai muito da escolha pessoal.

Eu já utilizei *Brackets* e *Sublime*, e são ótimos também. Mas nesse momento, vou dar continuidade com o *Visual Studio Code*.

Para fazer a instalação, segue a mesma facilidade dos outros. Entre no site <https://code.visualstudio.com/>, e clique no botão de download. No meu caso, está *Download for Windows*, pois estou usando uma máquina Windows. Mas se você estiver com *Mac* ou *Linux*, isso vai aparecer de acordo com seu sistema.

O Visual Studio Code, mesmo sendo da Microsoft, é plataforma livre, então roda em qualquer sistema operacional. O processo de instalação segue o padrão de *next, next, next, finish*.

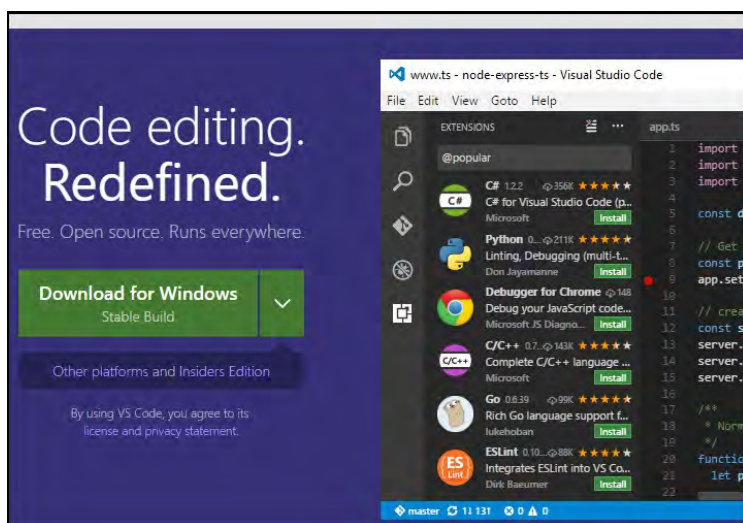


Figura 2.5: Página oficial do site Visual Studio Code

Para verificar a versão do Visual Studio Code instalada no computador, digite no prompt de comando `code -v`.

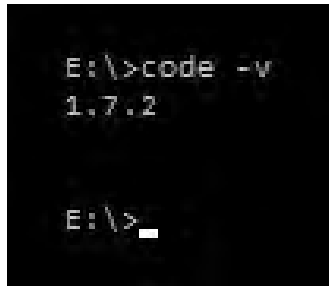
A terminal window with a black background and white text. The first line shows the command 'E:\>code -v' and the second line shows the output '1.7.2'. Below this, the prompt 'E:\>' is visible with a white cursor line.

Figura 2.6: Versão do Visual Studio Code instalada no computador

Finalizado isto, vamos conhecer os comandos para usar no desenvolvimento com Angular 2.

2.5 INSTALANDO O ANGULAR CLI

Existem duas formas de iniciar um projeto em Angular 2. A primeira é criando todos os arquivos na mão, um por um, escrevendo todos os códigos dentro de cada arquivo, declarando as dependências do projeto, e fazendo download dos pacotes até deixá-los pronto para uso. Este é um processo cansativo, repetitivo, complicado e pode ter muitos erros.

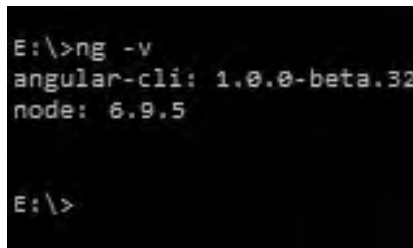
Estamos estudando uma tecnologia totalmente nova. Será que não existe algo para automatizar todo esse processo repetitivo? Sim, temos, e ele é muito prático no dia a dia.

O Angular CLI é a interface de linha de comando para usar no Angular 2. O significado de CLI é **Command Line Interface**, e o nome já diz tudo: com alguns comandos do CLI, podemos criar projetos rápidos, baixar as dependências, criar novos componentes, rodar o projeto e muito mais. Essa ferramenta é fundamental na construção de um projeto em Angular 2, evitando

erros e aumentando a produtividade.

Para baixar o Angular CLI, vamos no prompt de comando e digitamos `npm install -g angular-cli`. Para verificar a instalação, seguiremos o mesmo padrão que antes. Neste caso, somente mudando as iniciais.

Digite no prompt de comando `ng -v` para ver a versão instalada (isso pode demorar alguns segundos). Logo em seguida, aparecerá a versão do CLI e a do Node.js.

A screenshot of a Windows command prompt window with a black background and white text. The prompt is 'E:\>'. The user has entered 'ng -v'. The output shows 'angular-cli: 1.0.0-beta.32' and 'node: 6.9.5' on separate lines. The prompt 'E:\>' is visible again at the bottom.

```
E:\>ng -v
angular-cli: 1.0.0-beta.32
node: 6.9.5
E:\>
```

Figura 2.7: Versão do Angular CLI instalada no computador

2.6 COMANDOS DO ANGULAR CLI

Para ajudar no desenvolvimento das aplicações em Angular 2, o Angular CLI traz uma série de comandos para executar determinadas tarefas, retirando a responsabilidade do programador em codificar tudo no projeto. A seguir, mostrarei os comandos que vamos usar neste livro. Eles serão executados sempre dentro de um prompt de comando — de preferência, faça no prompt do próprio Visual Studio Code.

ng new

Este comando cria uma nova pasta, com um novo projeto

dentro, e gera todos os arquivos necessários para o funcionamento da aplicação. Cria também bases para teste com Karma, Jasmine, e2e com Protractor e, em seguida, faz toda a instalação das dependências listadas no `package.json`.

Para execução, colocamos o `ng new` e o nome do projeto, como `ng new novoProjeto`. Será criado um projeto com o nome `novoProjeto`. Temos de seguir as boas práticas para nomeação, isto é, *não iniciar com números, somente iniciar com letras ou* (*underline*). Após isso, podemos colocar qualquer número.

ng init

Este comando faz praticamente a mesma coisa que o `ng new`. Ele somente não cria uma nova pasta, mas cria um novo projeto com as dependências e deixa tudo pronto para execução.

Ele será usado quando você já tem uma pasta criada dentro do computador e somente quer criar o projeto dentro desta pasta. No prompt de comando, entre na pasta existente, logo após digite `ng init` e um novo projeto será criado dentro dela.

ng server

Após a criação do projeto, executamos o comando `ng serve` para buildar a aplicação, criar um servidor e rodar tudo no browser. O Angular CLI usa o servidor `livereload` como padrão. Digite no prompt de comando `ng serve`.

ng generate ou ng g

Este comando será o mais usado no projeto. Com o `ng`

`generate` , criamos todos os tipos de objetos que precisamos no Angular 2, e sempre seguindo os padrões e boas práticas de desenvolvimento da equipe do Google.

Para criarmos *componentes*, *serviços*, *classes* e tudo o que for utilizado dentro do projeto, usaremos este comando, ou sua abreviação, que é `ng g` . Sempre vamos seguir o padrão de criação: `ng g [objeto] [nome do objeto]` , em que, no lugar do `[objeto]` , será colocado o tipo que queremos criar. Veja conforme a tabela.

Tipo	Comando
Componente	<code>ng g component meu-nome</code>
Serviços	<code>ng g service meu-nome</code>
Classe	<code>ng g class meu-nome</code>
Interface	<code>ng g interface meu-nome</code>

Assim como podemos abreviar a palavra `generate` , também podemos abreviar o nome do tipo de objeto para criação. Então, a mesma tabela ficará assim.

Tipo	Comando
Componente	<code>ng g c meu-nome</code>
Serviços	<code>ng g s meu-nome</code>
Classe	<code>ng g cl meu-nome</code>
Interface	<code>ng g i meu-nome</code>

ng lint

Este comando serve para ajudar na organização do código e

colocar as boas práticas no projeto de acordo com a documentação. Se você é um pouco desorganizado na codificação, esse comando verificará todas as partes do projeto mostrando possíveis erros de espaços, indentação de código ou chamadas não usadas, de acordo com o *style guide* do Angular 2.

Será mostrado no prompt de comando *o arquivo, a linha, e qual o erro* para fazer a correção. Para executar, digite no prompt de comando `ng lint`.

ng test

Rodando este comando, vamos executar os testes unitários dos componentes usando o Karma e Jasmine. Serão mostrados no prompt de comando os possíveis erros, ou o sucesso dos testes, dentro de cada componente.

2.7 INICIANDO PROJETO EM ANGULAR 2

E chegou a hora! Chega de tanta teoria e vamos colocar as mãos na massa. Até o momento, temos conhecimento de ES6, TypeScript, comandos do Angular CLI e os conhecimentos necessários na introdução do Angular 2.

Agora criaremos um novo projeto que será a base de todos os exemplos do livro. Então, faça uma pasta onde quiser no seu computador para guardá-lo. Após você saber onde vai deixar armazenado o projeto, abra o *Visual Studio Code*.

No menu superior, clique em `View` e, em seguida, clique em `Integrated Terminal`. Também podemos usar o atalho `Ctrl + ``, que abrirá um pequeno painel na parte inferior da tela. Este é o

terminal do *VS Code*, o apelido dado para o *Visual Studio Code*

A partir de agora, quando eu falar *VS Code*, entenda que estou me referindo ao ambiente *Visual Studio Code*.

Este terminal é a mesma coisa do que usarmos o prompt de comando do sistema operacional. Usando os comandos no terminal do *VS Code*, economizamos tempo não precisando trocar de tela. Mas sintam-se à vontade para usar o prompt do seu computador caso queira.

Vá até o local que deseja guardar o nosso projeto pelo terminal. Após isso, vamos criar nosso projeto com o `ng new`. Mas antes, verifique que esteja conectado na internet, pois, como foi dito anteriormente, o comando `ng new` faz um novo projeto e já baixa as dependências para dentro da pasta `node_module`. E se você não estiver com internet no momento do processo de instalação, ele nunca vai terminar, isto é, seu projeto nunca estará liberado para começar o desenvolvimento.

Verificado isso e com tudo certinho, vamos criar o projeto. Nosso projeto vai chamar `LivroAngular2`, bem sugestivo para o momento. Portanto, digite `ng new LivroAngular2`.

```
E:>ng new LivroAngular2
installing ng2
  create .editorconfig
  create README.md
  create src\app\app.component.css
  create src\app\app.component.html
  create src\app\app.component.spec.ts
  create src\app\app.component.ts
  create src\app\app.module.ts
  create src\app\index.ts
  create src\assets\.gitkeep
  create src\environments\environment.prod.ts
  create src\environments\environment.ts
  create src\favicon.ico
  create src\index.html
  create src\main.ts
  create src\polyfills.ts
  create src\styles.css
  create src\test.ts
  create src\tsconfig.json
  create src\typings.d.ts
  create angular-cli.json
  create e2e\app.e2e-spec.ts
  create e2e\app.po.ts
  create e2e\tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Installing packages for tooling via npm.
Installed packages for tooling via npm.

E: >|
```

Figura 2.8: Projeto criado com sucesso

Finalizado a criação do projeto e com todas as dependências devidamente configuradas, nosso projeto em Angular 2 já pode ser inicializado. Uma dica que eu dou para você é abrir um prompt de comando do próprio sistema operacional, fora do VS Code, pois

agora vamos iniciar o servidor.

Quando o servidor é inicializado, o terminal fica travado, executando o serviço. Então, se rodar o `ng server` dentro do terminal do VS Code, ele ficará bloqueado e não vamos poder usá-lo.

Vamos abrir um prompt de comando do sistema operacional e ir até a pasta do projeto. Dentro da pasta raiz, iniciaremos o servidor com o comando `ng server`.

```

E:\LivroAngular2>ng server
** Web Live Development Server is running on http://localhost:4200 **
8939ms building modules
45ms sealing
1ms optimizing
0ms basic module optimization
137ms module optimization
1ms advanced module optimization
10ms basic chunk optimization
0ms chunk optimization
0ms advanced chunk optimization
1ms module and chunk tree optimization
81ms module reviving
8ms module order optimization
5ms module id optimization
4ms chunk reviving
1ms chunk order optimization
20ms chunk id optimization
71ms hashing
1ms module assets processing
192ms chunk assets processing
5ms additional chunk assets processing
0ms recording
0ms additional asset processing
1797ms chunk asset optimization
193ms asset optimization
45ms emitting
Hash: 386420ce83ecfc0941af
Version: webpack 2.1.0-beta.25
Time: 11584ms

   Asset      Size  Chunks             Chunk Names
  main.bundle.js  2.72 MB    0, 2  [emitted]  main
  styles.bundle.js  10.3 kB    1, 2  [emitted]  styles
  inline.bundle.js   5.54 kB         2  [emitted]  inline
  main.bundle.map   2.84 MB    0, 2  [emitted]  main
  styles.bundle.map  14.2 kB    1, 2  [emitted]  styles
  inline.bundle.map   5.6 kB         2  [emitted]  inline
  index.html    487 bytes             [emitted]

Child html-webpack-plugin for "index.html":
   Asset      Size  Chunks             Chunk Names
  index.html  2.81 kB         0

webpack: bundle is now VALID.
[default] Checking started in a separate process...
[default] Ok, 3.047 sec.

```

Figura 2.9: Servidor inicializado em localhost:4200

Logo no final do processo, aparecerá que o serviço foi iniciado. Na segunda linha, será mostrado em qual porta ele está respondendo. Abra o browser de sua preferência e digite localhost [sua porta] , e pronto! Seu primeiro projeto em

Angular 2 está rodando. Parabéns!



Figura 2.10: Projeto inicializado em localhost:4200

2.8 RESUMO

Neste capítulo, fizemos todas as instalações necessárias para o desenvolvimento com Angular 2. Começamos instalando o *Node.js* e, com ela, automaticamente já temos disponíveis os comandos do *NPM*.

Seguindo, instalamos o *TypeScript*, que vai servir para codificações no projeto. Logo após, baixamos um ambiente para desenvolvimento que, neste caso, será o *Visual Studio Code*, também conhecido como *VS Code*.

Outra instalação muito importante foi o *angular-cli* para facilitar no desenvolvimento com o Angular 2 e na criação das partes no projeto. Com ele, mostramos os principais comandos para facilitar a codificação.

Com tudo instalado, iniciamos um novo projeto, que será a base de todo o livro. Nele criaremos todos os exemplos durante o aprendizado com Angular 2.

ARQUITETURA DO SISTEMA E COMPONENTES DO ANGULAR 2

3.1 CONSTRUÇÃO DO PROJETO EM ANGULAR 2 E SEUS ARQUIVOS

Vamos iniciar o projeto verificando o que o `ng new` construiu automaticamente e para que serve cada arquivo criado. Abra o *VS Code*. Na parte esquerda, temos um botão azul escrito `open folder`. Clique neste botão e vamos procurar nossa pasta.

Apenas selecione a pasta `LivroAngular2` (sem entrar nela, porque o projeto precisa ser carregado com todos os arquivos). Após confirmar, será aberto o projeto no *VS code* no lado esquerdo, conforme a figura:

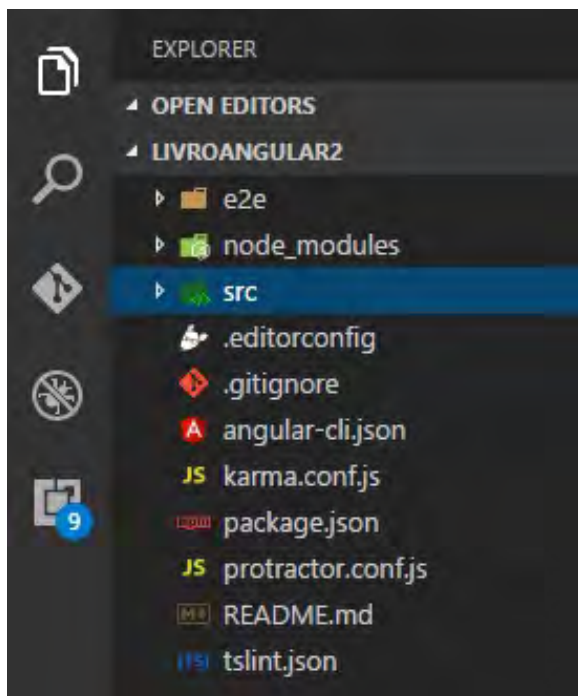


Figura 3.1: Projeto carregado no VS Code

Vamos passar por cada arquivo explicando a sua função. Logo no começo, temos a pasta `e2e`. Nela ficam os arquivos para teste de integração do projeto com *Protractor*. Nos arquivos dentro desta pasta, estão as configurações para os testes de todo o projeto, como instância de componentes e verificação de conteúdo dentro deles.

Diferente dos testes unitários, que no Angular 2 são feitos com o **Karma** e **Jasmine**, os testes de integração são feitos com *Protractor* e têm o objetivo de verificar a integração e comunicação entre os componentes da aplicação.

A próxima pasta já é uma velha conhecida nossa: a `node_modules`, na qual ficam armazenados todos os programas de dependência para nosso projeto funcionar. Tudo que estiver declarado no arquivo `package.json` será baixado e armazenado nela.

A pasta `src` é uma das mais importantes. Nela fica todo o código-fonte da aplicação TypeScript, JavaScript, HTML, entre outros. Vamos dar uma atenção especial para ela em outro momento.

O arquivo `angular-cli.json` contém as informações e configurações do projeto, como nome do projeto, sua versão, caminho dos arquivos internos, configuração de testes, configuração de estilos, pacotes, scripts e objetos.

O `angular-cli.json` é uma representação do projeto, o que ele tem e o que contém. Veja o código desse arquivo:

```
{
  "project": {
    "version": "1.0.0-beta.21",
    "name": "livro-angular2"
  },
  "apps": [
    {
      "root": "src",
      "outDir": "dist",
      "assets": [
        "assets",
        "favicon.ico"
      ],
      "index": "index.html",
      "main": "main.ts",
      "test": "test.ts",
      "tsconfig": "tsconfig.json",
      "prefix": "app",
      "mobile": false,
```



```

    "styles": [
      "styles.css"
    ],
    "scripts": [],
    "environments": {
      "source": "environments/environment.ts",
      "dev": "environments/environment.ts",
      "prod": "environments/environment.prod.ts"
    }
  },
  "addons": [],
  "packages": [],
  "e2e": {
    "protractor": {
      "config": "./protractor.conf.js"
    }
  },
  "test": {
    "karma": {
      "config": "./karma.conf.js"
    }
  },
  "defaults": {
    "styleExt": "css",
    "prefixInterfaces": false,
    "inline": {
      "style": false,
      "template": false
    },
    "spec": {
      "class": false,
      "component": true,
      "directive": true,
      "module": false,
      "pipe": true,
      "service": true
    }
  }
}

```

Dentro, temos as configurações da aplicação com um array em JSON chamado de `apps`. Lá vão estar o caminho do projeto, os

componentes e configurações. Dentro de `apps`, temos o arquivo de estilos com o array `styles`, que pode ser declarado algum arquivo CSS do projeto. Também dentro de `apps`, temos o arquivo de scripts adicionais com o array `scripts`.

Mais abaixo, temos os arrays de teste unitário e teste de integração, chamados de `e2e` e `test`, respectivamente. Caso for mudar algum tipo de configuração no projeto, será adicionado ou retirado neste arquivo.

O arquivo `karma.conf.json` é o arquivo de configuração do Karma, uma biblioteca feita para testes unitários junto com o **Jasmine**. Já o arquivo `package.json` é outro velho conhecido nosso. É nele que vamos colocar todas as dependências do projeto. Tudo o que o projeto precisa para funcionar será declarado nele. Vamos abri-lo para verificar o conteúdo.

```
{
  "name": "livro-angular2",
  "version": "0.0.0",
  "license": "MIT",
  "angular-cli": {},
  "scripts": {
    "start": "ng serve",
    "lint": "tslint \"src/**/*.ts\"",
    "test": "ng test",
    "pree2e": "webdriver-manager update",
    "e2e": "protractor"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "2.2.1",
    "@angular/compiler": "2.2.1",
    "@angular/core": "2.2.1",
    "@angular/forms": "2.2.1",
    "@angular/http": "2.2.1",
    "@angular/platform-browser": "2.2.1",
    "@angular/platform-browser-dynamic": "2.2.1",
```

```

    "@angular/router": "3.2.1",
    "core-js": "^2.4.1",
    "rxjs": "5.0.0-beta.12",
    "ts-helpers": "^1.1.1",
    "zone.js": "^0.6.23"
  },
  "devDependencies": {
    "@angular/compiler-cli": "2.2.1",
    "@types/jasmine": "2.5.38",
    "@types/node": "^6.0.42",
    "angular-cli": "1.0.0-beta.21",
    "codelyzer": "~1.0.0-beta.3",
    "jasmine-core": "2.5.2",
    "jasmine-spec-reporter": "2.5.0",
    "karma": "1.2.0",
    "karma-chrome-launcher": "^2.0.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
    "karma-remap-istanbul": "^0.2.1",
    "protractor": "4.0.9",
    "ts-node": "1.2.1",
    "tslint": "3.13.0",
    "typescript": "~2.0.3",
    "webdriver-manager": "10.2.5"
  }
}

```

O arquivo está separado em três partes. A primeira parte são dados do projeto, como *nome*, *versão*, *scripts* e *teste*.

Na segunda parte, dentro do `{}` com o nome de `dependencies`, estão as dependências do projeto para produção. Tudo o que estiver contido dentro dessa configuração será necessário para o projeto rodar em produção.

Na terceira parte, dentro do `{}` com o nome de `devDependencies`, estão todos os arquivos necessários para o projeto em fase de **desenvolvimento**. Tudo o que é necessário para o desenvolvimento do projeto será colocado nesta configuração.

Veja que a parte de `dependencies` é menor do que a parte de `devDependencies`, pois os arquivos de *testes*, *TypeScript*, *angular-cli*, *Karma*, *Jasmine*, entre outros, não precisam estar em produção. Eles só serão usados em desenvolvimento.

O `protractor.conf.js` é o arquivo de configuração da biblioteca Protractor, feito para testes de integração do projeto. E por último, mas não menos importante, no arquivo `tslint.json` estão contidas as configurações para usar no comando `ng lint`, que, como vimos no capítulo anterior, são as boas práticas de desenvolvimento do Angular 2. Abra e veja o que está lá, mas não altere nada para não mudar as boas práticas de desenvolvimento.

Todos os arquivos e as pastas apresentados até o momento são de configuração do projeto, e dificilmente vamos mexer em algo. Tudo já está bem ajustado e não precisamos nos preocupar com eles. Mas faltou comentarmos sobre uma pasta, a `src`. Vamos vê-la em detalhes.

Como foi dito, a pasta `src` é uma das mais importantes da aplicação. Dentro dela, está contido todo o conteúdo do projeto.

Abrindo a pasta `src`, temos a pasta `app`, dentro da qual ficará todo o código-fonte da aplicação. Ela é a pasta principal e passaremos os detalhes dela posteriormente. Temos também a pasta `environment` com dois arquivos, `environment.prod.ts` e `environment.ts`, que são para configuração na hora de fazer o *build*. Quando formos fazer o *build* para produção, poderemos fazer alguma configuração especial caso necessário.

O ícone `favicon.ico` é a imagem que ficará na barra de navegação do browser. Isso pode ser mudado a qualquer momento

que você deseja.

O arquivo `index.html` é o já conhecido por páginas web, o arquivo inicial de uma aplicação web. Essa `index.html` é a página inicial do projeto em Angular 2, e é nela que vão as tags para criar as páginas web.

Mas espere! Você não vai mexer nisso! No Angular 2, construímos componentes que são renderizados na tela do computador, logo você verá que não mexeremos no `index.html`. Vamos construir nossas páginas de outras formas.

O arquivo `main.ts` é o arquivo *bootstrap* da aplicação. É nele que damos a direção para a inicialização do projeto no browser do computador.

```
import './polyfills.ts';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';
import { environment } from './environments/environment';
import { AppModule } from './app/';

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

Veja que neste arquivo temos somente uma linha, o `platformBrowserDynamic().bootstrapModule(AppModule);`, que diz para inicializar as configurações contidas no `AppModule`. O Angular 2 não sabe onde fica este `AppModule`, então temos de importá-lo como está na linha 6, o `import { AppModule } from './app/';`. Isso diz ao Angular 2 que o `AppModule` está no

caminho raiz/app da aplicação.

O arquivo `polyfills.ts` são as bibliotecas que auxiliam no projeto para codificação de JavaScript.

```
import 'core-js/es6/symbol';
import 'core-js/es6/object';
import 'core-js/es6/function';
import 'core-js/es6/parse-int';
import 'core-js/es6/parse-float';
import 'core-js/es6/number';
import 'core-js/es6/math';
import 'core-js/es6/string';
import 'core-js/es6/date';
import 'core-js/es6/array';
import 'core-js/es6/regexp';
import 'core-js/es6/map';
import 'core-js/es6/set';
import 'core-js/es6/reflect';

import 'core-js/es7/reflect';
import 'zone.js/dist/zone';
```

AQUI VAI UMA TEORIA MUITO IMPORTANTE

Na codificação do Angular 2, vamos usar o TypeScript, mas os browsers atuais ainda não suportam esse padrão de código. Sendo assim, para o projeto rodar no navegador, precisamos compilar o código TypeScript para JavaScript, e é aí que entra outro problema. O ES6 não é suportado em todos os navegadores atuais. Então, não podemos compilar para ES6, temos de compilar todo o projeto para ES5.

Entretanto, no ES6, temos funções e métodos adicionados que não existe no ES5. Como vamos usar coisas do ES6 no TypeScript, sendo que ele será compilado para ES5? É então que entram os *polyfills*.

Eles vão auxiliar na codificação. Quando usarmos alguma coisa do ES6 no TypeScript, e esse TypeScript for compilado para ES5, o arquivo `polyfill` vai mostrar como executar o procedimento que é do ES6 no ES5. A grosso modo, os *polyfills* fazem um **de / para** do ES6 para ES5.

O arquivo `styles.css` é o CSS global da aplicação. Como assim global? Logo você vai entender.

Já o arquivo `test.ts` é outro de configuração de testes, e não vamos mexer nesses arquivos. Em resumo, ele é o responsável por mapear todos os arquivos de teste existentes em toda a aplicação. Cada componente tem um arquivo de teste separado, e esse arquivo `test.ts` é responsável por achar cada um e executá-los

para fazer os procedimentos contidos.

O arquivo `tsconfig.json` é o de configuração do TypeScript. Nele estarão contidas as configurações e a compilação do Typescript para JavaScript, como: qual a versão do JavaScript alvo para compilação, os caminhos dos arquivos dentro do projeto, entre outras coisas.

Para finalizar as pastas e arquivos de configuração, falaremos sobre a pasta `app`. É nela onde vamos passar praticamente todo o tempo de desenvolvimento do projeto. Neste local é armazenado tudo o que fazemos na aplicação, como novos componentes, serviços, templates, configuração do CSS e arquivos de teste de componentes.

A cada novo objeto criado para a aplicação, é na pasta `app` que ele vai ser configurado e utilizado. Tudo o que fizermos de novo ficará lá. Logo no início da pasta, temos 4 arquivos com o mesmo nome: o `app.component`, somente mudando a extensão do arquivo para `.css`, `.html`, `.spec.ts` e `.ts`. Todos esses arquivos falam do mesmo objeto, mas cada um de uma parte dele.

Toda vez que criarmos um novo objeto para o projeto com o comando `ng g [objeto] [nome-objeto]`, será criada uma pasta dentro da pasta `app` com o nome dele. E dentro desta pasta, serão criados quatro arquivos com o mesmo formato descrito anteriormente.

O `app.module.ts` é o arquivo de declaração dos objetos dentro do projeto. Nele será mapeado para que serve e onde está cada objeto criado na aplicação. O `app.module.ts` é o nosso cartão postal, ele diz tudo o que tem dentro do projeto.


```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

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

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

```

Dentro do array de configuração `declarations`, é colocado o nome de cada novo objeto criado dentro do projeto. Essa é a maneira como o Angular 2 usa para saber o que tem e o que não tem para ser usado. Por exemplo, se criarmos um componente e ele não for declarado neste array, o Angular 2 não vai reconhecer este novo componente como sendo parte do projeto.

O array de `imports` é colocado em todas as bibliotecas internas do Angular 2 que serão usadas no projeto. Neste código, temos os módulos `FormsModule` e `HttpModule` que são importados para dentro do projeto, permitindo trabalhar com formulários e também com conexão HTTP para comunicação com servidores.

O array `providers` será colocado em todos os serviços do projeto, e estes são as lógicas de negócio da aplicação, conexão com servidor, entre outros. Já o array `bootstrap` mostra para o

sistema qual o arquivo a ser carregado primeiro na aplicação.

BOOTSTRAP DO ANGULAR 2

Não confunda o *bootstrap* do Angular 2 com o framework Bootstrap para CSS. O bootstrap do Angular 2 tem o sentido de **inicialização** de alguma coisa. Ele serve para dar o *starter*, o início para algo acontecer. Sempre que for falado *bootstrap* fora do ambiente de HTML, considere que o Angular 2 vai iniciar algum arquivo.

Veja que, dentro do `bootstrap`, está o `AppComponent`. Ele tem o mesmo nome dos quatro arquivos descritos da pasta `app`, e esse arquivo, `app.module` (cujo conteúdo estamos vendo), já foi citado quando falamos do arquivo `main.ts`.

Será que tudo isso tem alguma ligação? Sim, tem. E vou lhe explicar agora como tudo funciona.

3.2 COMO FUNCIONA O BUILD NO ANGULAR 2

Quando alguém faz uma requisição no servidor onde está armazenada nossa aplicação, ele vai à procura do nosso arquivo `index.html` para renderizar o conteúdo na tela. Até aqui normal, toda aplicação web faz isso. É aí que entram as configurações do Angular 2.

Quando o `index.html` é chamado, logo em seguida é

iniciado o arquivo `main.ts` . Ele diz ao Angular 2 onde achar as declarações dos objetos que o projeto tem. O `main.ts` inicializa o arquivo `AppModule` para fazer a criação de todos os componentes do projeto.

Chegando ao arquivo `AppModule` , serão instanciados todos os objetos do array `declarations` e `providers` . Todos esses objetos estão declarados no início deste arquivo como `import` , para o Angular 2 saber o caminho dos arquivos de cada objeto para instanciação.

Após reconhecidos todos os objetos que compõem o projeto, o Angular 2 vai ao array `bootstrap` para saber qual deles é o primeiro a ser iniciado. Então, vamos à instância do `AppComponent` que contém os quatro arquivos, que são o HTML, o CSS, o arquivo `Typescript` (`.ts`), e o arquivo de teste (`spec.ts`) do `AppComponent` .

Sabendo desses arquivos, o Angular 2 executa o `app.component.ts` , pois é onde estão o código `TypeScript` e toda a lógica de funcionamento deste objeto. Dentro do `app.component.ts` , ele procura a declaração desse objeto que, neste caso, fica dentro do `@Component` .

Dentro dessa declaração, temos o **metadado** chamado `templateUrl` . É esse caminho que leva aonde o HTML deste objeto está, então o Angular 2 renderiza o HTML no navegador.

Nossa, quanta coisa! É muito complicado isso? Não, é simples e fácil. Basta somente seguir o processo. Mas nesta explicação, falamos duas palavras novas, **declaração e metadata**. Guarde-as e logo você saberá o que significam.

3.3 PARTES DO COMPONENTE EM ANGULAR 2

Muitas vezes, referimo-nos a *objetos do Angular 2* ou *componentes do Angular 2*, mas na verdade o que é isso? Neste momento, você já sabe para que serve o framework Angular 2, correto? Ele é um framework SPA para desenvolvimento front-end com uma característica muito interessante, que é o conceito de modularização.

Modularização é a forma de desenvolvimento em pequenas partes para ficarem mais fáceis a construção, os testes e a reutilização. O carro chefe do Angular 2 é fazer um sistema modularizado, em que a junção de pequenas partes forma uma página completa.

O grande benefício dos módulos é desmembrar a aplicação em partes pequenas, em que cada uma terá sua responsabilidade. Ela poderá ser facilmente testada, além de permitir uma manutenção mais simples.

O Angular 2 usa várias bibliotecas, algumas do próprio núcleo do framework e outras opcionais para cada projeto. Você escreve uma aplicação modular com trechos de HTML, sendo o template, classes para a lógica do componente e decorações do Angular 2 usados para diferenciar um componente do outro. Em Angular 2, tudo é um componente, e eles são a principal forma de construir e especificar elementos e lógica na página.

Neste momento, vamos passar pelas oito principais partes de uma aplicação Angular 2, e explicar cada uma delas. No término de cada tópico, montaremos uma parte de um diagrama e

acompanharemos nossa evolução nos conceitos apresentados. A cada término, vamos incrementando o diagrama para, no final, ter um desenho de fluxo completo de como funciona o Angular 2.

Vamos usar como exemplo um componente que vai adicionar e listar nomes de pessoas. Para isso, criaremos um novo componente através do prompt de comando do VS Code, e usaremos os comandos do *Angular CLI*. Vamos digitar o comando `ng g c` ou `ng generate component`, em que:

Comando	Descrição
<code>ng</code>	Comando do Angular cli
<code>g</code>	Abreviação do <i>generate</i>
<code>c</code>	Abreviação do <i>component</i>

O nome que vamos usar é `lista-pessoa`, então criaremos nosso componente com o comando `ng g c lista-pessoa`, ou `ng generate component lista-pessoa` — os dois têm o mesmo resultado.

3.4 COMPONENTE

Um componente é uma classe responsável por controlar a *view*. Nela definimos a lógica que será aplicada para controle e ações da *view*.

Devemos somente colocar nessa classe a lógica de controle do componente. Nada que se refere à lógica de negócio pode ser colocado nesta classe, pois ela é única e exclusivamente para controle de conteúdo e controle de ações da *view*.

Esta é uma classe do componente no Angular 2:

```
export class ListaPessoaComponent implements OnInit {  
  
  pessoas: string [];  
  
  constructor() { }  
  
  ngOnInit() {  
  }  
  
  listar() {  
  
  }  
  
}
```

Definimos uma classe com o nome de `ListaPessoaComponent` e, dentro, vamos criar um array com o nome `pessoas` do tipo `string`. Logo em seguida, temos o construtor da classe; ele é executado logo no início quando o componente é instanciado.

O método construtor é utilizado quando queremos inicializar classes ou variáveis que precisam ser usadas logo no começo da apresentação do componente no navegador. Tudo o que o componente necessita para começar a ser usado será declarado nesse método construtor. Geralmente, declaramos variáveis e injeção de dependência de classes de serviço dentro do construtor para, quando o componente começar a ser usado, já ter dados para mostrar no navegador.

Fique tranquilo, pois logo falaremos sobre injeção de dependência e classe de serviços. Seguindo pelo código, vamos criar o método `listar()`, que chamará um serviço que vai criar, listar as pessoas e atribuir o retorno no array `pessoas`.

Com a **classe componente** devidamente apresentada, colocamos uma função em nosso diagrama.

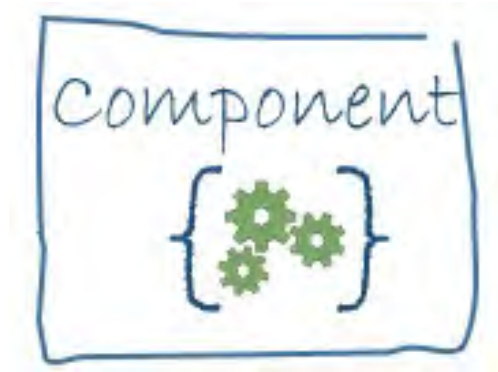


Figura 3.2: Classe do componente

3.5 TEMPLATE

O template define a visão do componente. Ele é a *view* do componente. É feito de HTML e de algumas marcações do próprio Angular 2 que indicam como renderizar o conteúdo no navegador.

O código que faremos para o template para ser exibido no navegador ficará assim:

```
<h2>Lista de Pessoas</h2>

<ul>
  <li>

    </li>
</ul>
```

Esse código em HTML será renderizado no navegador com o título *Lista de Pessoas* na tag `h2` , seguido de uma lista não ordenada com as tags `ul` e `li` .

Com o **template** devidamente apresentado, colocamos mais uma função em nosso diagrama:

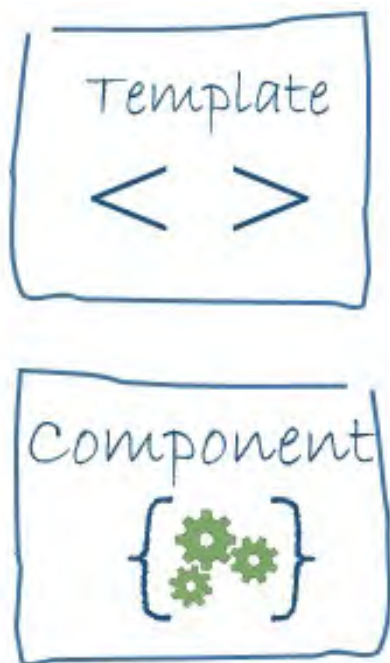


Figura 3.3: Template no Angular 2

3.6 METADATA

O metadata diz ao Angular 2 como processar uma classe. Até agora, fizemos uma classe `ListaPessoaComponent` e um template para, no futuro, poderem exibir as pessoas encontradas do serviço que vamos criar. Mas como o Angular 2 saberá que o template que vai listar as pessoas faz ligação com a classe `ListaPessoaComponent` ? Através das *decorações* (**decorations**) e **metadata**.

O *metadata* é um decorador no Angular 2 que fica responsável por fazer a configuração entre a classe e o template. A `ListaPessoaComponent` é uma classe comum até que você coloque o *decoration*, assim, o Angular reconhece-a como um componente, devido aos seus *metadatas*.

O código da classe `ListaPessoaComponent` com o *decoration* e o *metadata* ficará assim:

```
@Component({
  selector: 'app-lista-pessoa',
  templateUrl: './lista-pessoa.component.html',
  styleUrls: ['./lista-pessoa.component.css']
})
export class ListaPessoaComponent implements OnInit {

  /* . . . resto do código é o mesmo . . . */
}
```

Logo no início, temos o decorador `@Component` que indica que essa classe é um componente. Quando decoramos uma classe com o `@Component`, temos de configurar os *metadatas* necessários para o Angular 2 reconhecer o componente por completo.

O *metadata selector* é o nome dado para este componente dentro de uma página HTML. Este seletor diz para o Angular 2 inserir uma instância dele onde estiver a marcação `< app-lista-pessoa> </ app-lista-pessoa>` na página HTML.

Imagine que, em determinada parte do projeto, precisamos exibir uma lista de pessoas no navegador, e nosso componente que estamos criando fará essa função. Então, nesta parte só precisamos colocar a marcação `<app-lista-pessoa> </ app-lista-pessoa>`, que o Angular 2 vai identificá-la como sendo um

componente personalizado e, no lugar dela, será renderizado o conteúdo do template deste componente.

Vá ao arquivo `app.component.html` e coloque `<app-lista-pessoa> </ app-lista-pessoa>` . Volte ao navegador em que está rodando o projeto e veja nosso template sendo mostrado.

```
<h1>
  {{title}}
</h1>

<app-lista-pessoa></app-lista-pessoa>
```

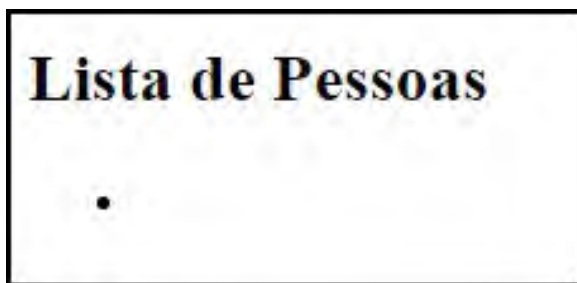


Figura 3.4: Componente Lista Pessoas sendo renderizado na tela

Sempre que precisar usar uma lista de pessoas, só precisamos colocar a marcação `<app-lista-pessoa> </app-lista-pessoa>` , e o Angular 2 já saberá que estamos nos referindo ao componente criado. O metadata `templateUrl` mostra qual é o template em HTML que este componente tem e o caminho para encontrá-lo.

No nosso componente que vai listar os nomes de pessoas, temos um template em HTML que está em alguma parte dentro do projeto. Com este metadata `templateUrl` , o Angular 2 sabe que o caminho para usar o template que está em `./lista-`

`pessoa.component.html` , em que o `./` significa a pasta raiz do componente (no nosso caso, a pasta `lista-pessoa`) e `lista-pessoa.component.html` é o nome do template que ele precisa renderizar no navegador.

Esses dois metadatas são obrigatórios em todo o componente, mas temos outros opcionais para o decorador `@Component` . O metadata `styleUrls` tem quase a mesma função do `templateUrl` . Ele mostra qual é o arquivo CSS deste componente e o caminho para encontrá-lo. Cada componente tem seu próprio arquivo CSS e, dentro do projeto, temos o CSS global. Fique à vontade para codificar em qualquer um deles.

O metadata `providers` descreve dentro de um array todos os serviços que este componente vai precisar. Essa é uma das maneiras de dizer ao Angular 2 que o componente `ListaPessoaComponent` precisa de um serviço ou de alguma informação que está fora do componente.

Se um serviço for declarado dentro do componente, somente este terá acesso ao serviço. Caso queira que um serviço seja usado por vários componentes, teremos de declará-lo no arquivo de configuração global do projeto, o `app.module.ts` .

Nosso exemplo terá dois serviços: um de listar nomes de pessoas, e outro global para mostrar uma mensagem de alerta no navegador. O serviço que lista nomes de pessoas só será usado dentro do nosso componente `lista-pessoas` , logo, vamos declará-lo no metadata `providers` do decoration do componente. Já o serviço que envia um alerta no navegador será declarado no arquivo global do projeto para outros componentes poderem usar.

Existe outros tipos de *decorations* que serão abordados nos próximos capítulos deste livro. Um exemplo será o `@Injectable` para descrição de serviços.

UM RESUMO ATÉ AQUI

Uma aplicação em Angular 2 é feita por meio de componentes. Um componente é a combinação de um *template* em HTML e uma classe que é decorada como `@Component`. Todo componente começa com um decorador `@Component` e com seus objetos *metadata*, que descrevem como o template HTML e a classe componente vão trabalhar em conjunto.

Dentro do decorador `@Component`, temos dois metadados principais, que são `selector` e `template`. A propriedade `selector` descreve como será a tag HTML deste componente dentro do projeto, e o `template` mostra qual é o conteúdo dele exibido no navegador. Esta definição é muito importante, pois tudo gira em torno dos componentes em Angular 2.

Com **decoration** e **metadata** devidamente apresentados, colocamos mais funções em nosso diagrama.

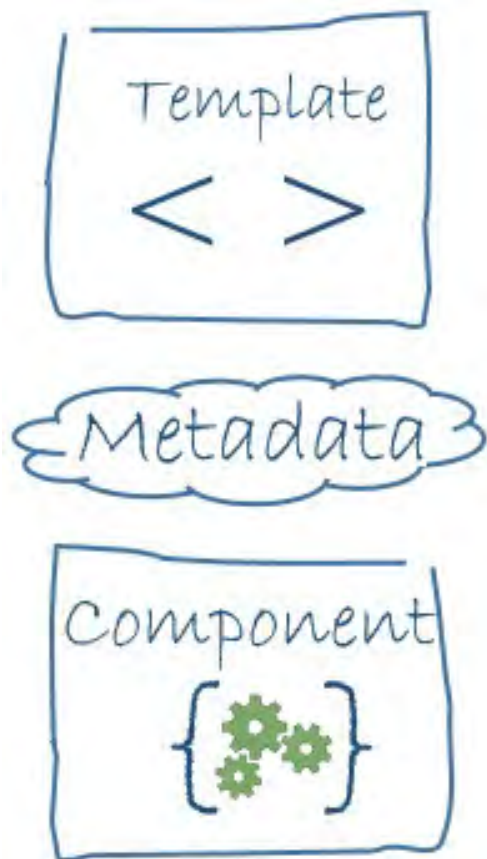


Figura 3.5: Componentes no Angular 2

3.7 DATA BINDING

O **data binding** é usado para enviar ou sincronizar os dados entre classe do componente e o template. Podemos passar informações da sua classe para serem mostradas no template, ou passar informações do template para serem executadas na classe do componente.

Temos quatro tipos de *data binding*: *interpolation* , *property binding* , *event binding* e o *two-way data binding* .

Interpolation

A interpolação (ou **interpolation**) é usada quando queremos passar dados que estão na classe do componente para serem mostrados no template. Ele sempre terá esse sentido de transporte **componente para template**.

Para passar as informações que estão na classe do componente, usamos chaves duplas no template. Dessa forma, `{{variável}}` e, dentro das chaves, colocamos o nome da variável que está na classe do componente. Isso vai dizer ao Angular 2 que queremos mostrar os dados da variável que estão na classe do componente dentro do template.

Agora vamos usar o *interpolation* no nosso projeto, vamos mudar o título da nossa página. Vamos abrir o arquivo `app.component.ts` . Veja que temos uma variável chamada `title` , com o valor de `app works!` . É esse valor que vamos mudar para ser exibido no navegador.

Abra agora o arquivo `app.component.html` , veja que temos uma tag `h1` e, dentro dela, o **interpolation**.

```
<h1>
  {{title}}
</h1>
```

Estamos passando os dados que estão na variável `title` da classe do componente para o template com interpolação `{{title}}` . Todo o conteúdo desta variável será mostrado neste

espaço do template.

Modifique o valor da variável `title` para `Livro Angular 2`, salve e veja no navegador o valor atual sendo mostrado. Minha aplicação está em `localhost:4200`, onde o servidor está rodando.

Meu arquivo `app.component.ts`:

```
export class AppComponent {  
  title = 'Livro Angular 2';  
}
```

Meu arquivo `app.component.html`:

```
<h1>  
  {{title}}  
</h1>
```



Figura 3.6: Navegador com novo texto

Property binding

Este segue praticamente o mesmo sentido do *interpolation*. O **property binding** é usado quando queremos passar informações da classe do componente para alguma propriedade de tag do template.

Um exemplo bem comum é a tag `img`, que tem a propriedade `src`, na qual passamos o caminho de alguma imagem. Para usar,

precisamos colocar a propriedade dentro de `[]` , isto é, no final, ficará assim: `[propriedade]` .

Para exemplificar o uso do property binding, vamos adicionar uma imagem abaixo do nosso título. Vamos colocar abaixo da tag `h1` do arquivo `app.component.html` uma tag `img` , que mostrará a imagem do símbolo do Angular 2. Esta imagem está dentro do nosso projeto com o nome `favicon.ico` .

Agora vamos criar uma variável na classe do componente (`app.component.ts`) com o nome `foto` , e atribuir a ela o caminho da imagem.

```
foto: string = 'favicon.ico';
```

Dentro da tag `img` que está no template, colocaremos uma propriedade `src` , e em volta colocamos `[]` para dizer ao Angular 2 que queremos fazer um *property binding* com a propriedade `src` . Em seguida, passamos o nome da variável que será a referência do property binding, desta forma:

```
<img [src]="foto">
```

Veja no nosso navegador a imagem:

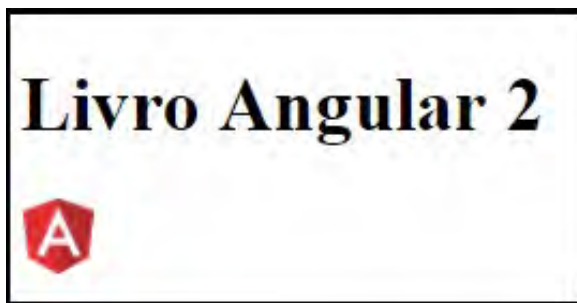


Figura 3.7: Navegador com a imagem

Event binding

O **event binding** segue o caminho inverso dos outros. Ele é utilizado para passar dados do **template para a classe do componente**. É muito usado para eventos de tela, como um *clique* em algum botão ou alguma *entrada de dados* na tela.

O *event binding* é usado na interação com o usuário e, a cada ação dele (seja um clique, entrada de dados ou alteração no template), o evento atualiza ou manipula algo dentro da classe do componente.

Para usar o *event binding*, temos de colocar dentro da tag do template os `()` e, dentro, a ação que queremos executar. Por exemplo, ficaria: `(click)="ação"` .

No nosso projeto, vamos usar o event binding em dois momentos: um no `app.component.html` e outro dentro do `lista-pessoa.component.html` . Ele vai servir para executar os métodos que estarão na classe do componente quando o usuário fizer um click no botão.

Abaixo da tag `img` que está no template do `app.component.html` , vamos colocar uma tag `button` e, dentro dela, colocamos um event binding de `click` dessa forma:

```
<button (click)="msgAlerta()">Enviar Alerta</button>
```

Dentro do arquivo `app.component.ts` , faremos o método que será executado quando o botão for clicado.

```
msgAlerta(): void {  
    alert('Livro Angular 2');  
}
```

Quando atualizar o projeto no navegador e clicar no botão, será exibido o conteúdo `Livro Angular 2` em um `alert` no navegador.

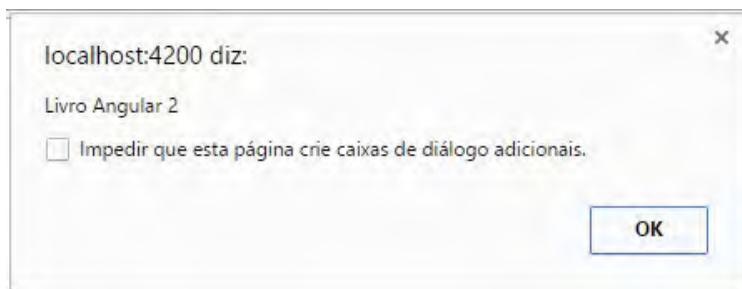


Figura 3.8: Alerta na tela

Two-way data binding

O último *data binding* é o **two-way data binding** que junta tanto o binding de propriedade (*property binding*) quanto o binding de evento (*event binding*). Representamos com `[(ngModel)]="variável"`, que contempla tanto a representação de propriedade quanto de evento `[()]`.

Dessa forma, o *two-way data binding* atualiza o template e a classe do componente a qualquer momento que a variável declarada for mudada. Se for atualizado pelo template, o valor vai para a classe do componente; se for atualizado pela classe do componente, o valor é enviado para o template.

Podemos usar essa forma de data binding no preenchimento de um formulário, em que as informações estão sendo alteradas constantemente, usando o *two-way data binding* dentro de formulários. A cada mudança de informação feita em qualquer

campo, o valor será enviado para a classe `componente` e já poderá ser validado conforme a regra do formulário.

Vamos fazer um exemplo dentro do componente de lista de pessoas para verificar como funciona o *two-way data binding*. Adicionaremos a tag `input`, dentro do arquivo `lista-pessoa.component.html`:

```
<input type="text">
```

Vamos declarar no arquivo `lista-pessoa.component.ts` uma variável chamada `nome`, com um valor inicial.

```
nome: string = "Thiago";
```

A variável `nome` será atualizada sempre que tiver uma alteração no campo de `input` da nossa página. Vamos colocar o *two-way data binding* `[(ngModel)]="nome"` dentro da tag de `input`.

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

Salvando o arquivo, quando voltarmos ao navegador, o campo `input` estará com o valor da variável `nome`, pois o *two-way data binding* já fez a união da variável com a tag. Para ver o valor da variável sendo mudada, adicionaremos uma tag de parágrafo abaixo da tag `input` no arquivo `lista-pessoa.component.html`. Ele vai servir somente para visualização das mudanças do conteúdo da variável `nome`, quando alterarmos o conteúdo que está dentro do `input`.

```
<input type="text" [(ngModel)]="nome">  
<p>{{nome}}</p>
```

Salvando e voltando ao navegador, digite alguma coisa dentro da caixa de `input` e veja no parágrafo o que vai acontecer. Viu?

Ele mudou sozinho! Automaticamente, quando modificamos o valor da variável em qualquer lugar (tanto template quanto componente), os dois lugares recebem a atualização.

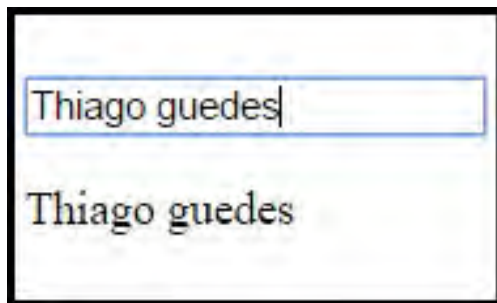


Figura 3.9: Two-way data binding

Com o **data binding** devidamente apresentado, colocamos mais uma função em nosso diagrama:

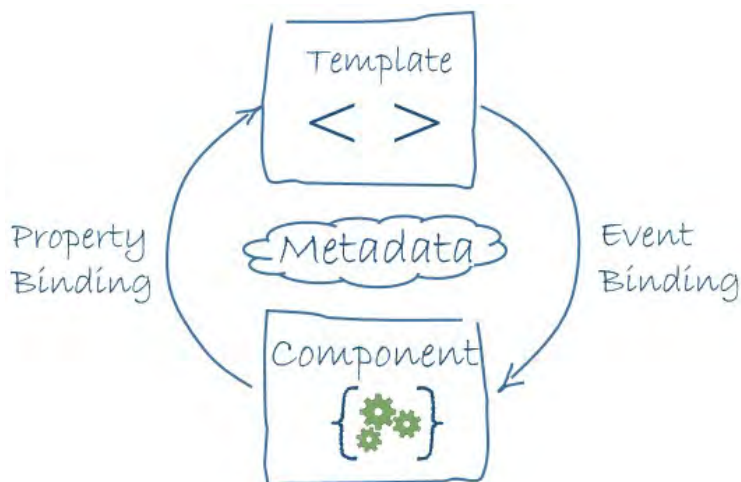


Figura 3.10: Componente com data binding

3.8 DIRETIVAS

As *diretivas* são uma forma de interação com o template. Dependendo de qual for a regra de negócio, elas modificam ou interagem dinamicamente com as tags do HTML.

Tudo que interage ou modifica a estrutura do DOM (*Document Object Model*) pode ser considerado uma diretiva. Ou seja, nossos componentes criados também são um tipo de diretiva, pois, quando instanciados e renderizados no navegador, serão responsáveis por modificar a estrutura do DOM.

Podemos dividir as diretivas em duas partes: **diretivas estruturais** e **diretivas de atributos**.

Diretivas estruturais

As diretivas estruturais são todas que modificam a estrutura da página. Elas interagem com o template, modificando a estrutura mostrada no navegador.

Podemos colocar nesta lista todas as estruturas de decisão ou laços de repetição? Sim! O Angular 2 tem toda uma arquitetura de estrutura de decisão e laços de repetição dentro do template. Podemos usar o `*ngFor` , `*ngIf` , `*ngSwitch` , entre outros. A forma de funcionamento é a mesma que em linguagens tradicionais, como C#, Java, PHP etc.

Declaramos como *diretivas estruturais* pois, dependendo dos dados enviados e das validações, elas modificam as tags do HTML, podendo adicionar ou remover partes da estrutura da página. Podemos fazer um exemplo para este tipo de diretiva. No arquivo

`lista-pessoa.component.html` , vamos usar uma diretiva estrutural chamada de `*ngFor` .

Esta vai receber um array e mostrar no navegador o conteúdo de cada posição dentro deste array. Nosso código HTML ficará assim:

```
<ul>
  <li *ngFor="let pessoa of pessoas">
    {{pessoa}}
  </li>
</ul>
```

Agora vamos ao arquivo `lista-pessoa.component.ts` , criamos um array com o nome `pessoas` e colocamos alguns nomes dentro dele. Ele é o mesmo que está referenciado no `*ngFor` do template. Neste `ngFor` , estamos dizendo para o template que **cada elemento do array `pessoas` coloque na variável `pessoa` e mostre o conteúdo no interpolation `{{pessoa}}` .**

A cada interação com o array de `pessoas`, o `ngFor` pega uma posição e atribui o conteúdo na variável `pessoa` . Essa variável será mostrada no navegador através da interpolação.

Neste caso, podemos ter uma quantidade maior ou menor de `li` , dependendo do tamanho do array `pessoas` . Isso vai modificar toda a estrutura do DOM (estrutura da página).

Nosso código no arquivo `lista-pessoa.component.ts` será esse:

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

@Component({
  selector: 'app-lista-pessoa',
```

```

    templateUrl: './lista-pessoa.component.html',
    styleUrls: ['./lista-pessoa.component.css']
  })
  export class ListaPessoaComponent implements OnInit {

    pessoas: string [] = ['João', 'Maria', 'Angular 2'];
    nome: string = "Thiago";

    constructor() { }

    ngOnInit() {
    }

    listar() {

    }
  }

```

Colocamos os nomes *João*, *Maria*, *Angular 2* dentro do array `pessoas` . Ao salvarmos e voltarmos ao navegador, estes serão mostrados em uma lista.



Figura 3.11: Dois componentes na mesma página

Veja que o navegador está mostrando uma página completa em HTML, mas, dentro do nosso projeto, os objetos estão separados em arquivos: uma parte está no `app.component.html`, e outra no `lista-pessoa.component.html`. Este é um conceito de modularização, em que pequenas partes juntas formam uma página inteira.

Diretivas de atributos

Esse tipo de diretiva altera a aparência, comportamento ou conteúdo do elemento que está na tela. Entra nesta lista os *data bindings* (property binding, event binding, two-way data binding), atributos de classe (o `*ngClass`) e atributos de estilos (o

`*ngStyle`) — que cuidam da adição de classes e estilos, respectivamente. Vamos falar deles mais adiante.

Tudo que não modifica a estrutura da página, mas modifica os elementos que estão nela, é considerado *diretivas de atributos*. Um exemplo é o `[(ngModel)]`, visto no tópico anterior. Ele modifica o conteúdo do elemento que está na tela, porém não retira nem adiciona nada na estrutura do HTML.

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

Em resumo, se a diretiva modificar a estrutura da página, adicionando ou removendo tags do HTML, ela é uma **diretiva estrutural**. Se a diretiva modificar o conteúdo, aparência ou comportamento de alguma tag da página, é uma **diretiva de atributo**.

Com as **diretivas** devidamente apresentadas, colocamos mais uma função em nosso diagrama:

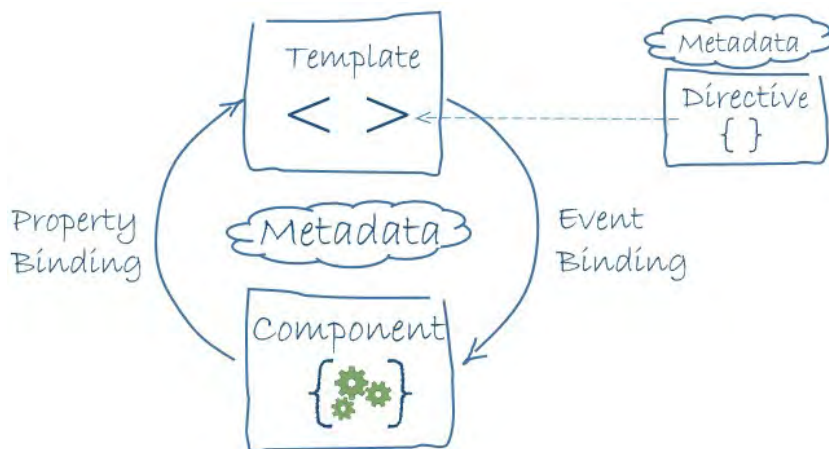


Figura 3.12: Diagrama com as diretivas

3.9 SERVIÇOS

No Angular 2, não temos um conceito definido sobre serviços. Em geral, serviço é uma classe com um *decorator* `@Injectable`, responsável por fazer algumas funções, regras de negócios, métodos e procedimentos que não são parte dos componentes, e essas regras e funções não se limitam a somente buscar e enviar algo no servidor.

Como já foi dito algumas vezes, um componente somente deverá ter código que é de manipulação do componente. Toda regra de negócio, validações, conexão do servidor e tratamento de dados deverão ser feitos em outras classes. Essas classes são os serviços.

Nas boas práticas do Angular 2, recomenda-se que uma classe só pode ter até 400 linhas de código e, cada método, até 75 linhas. Isso melhora a organização do código e do projeto, o debug da aplicação e os testes, tanto individual quanto de integração.

Sabendo dessas duas regras, concluímos que um serviço vai muito além de um simples conceito. Ele pode ser usado para tudo que não seja específico de um componente dentro do projeto.

Usamos serviços para validação de dados, serviços de log, tratamento de erros, tratamento de dados, conexão com banco, isto é, tudo o que está fora do escopo de um componente será visto com uma classe de serviço. Sempre que algum código não fizer parte de um componente, este vai para uma classe serviço.

Com as classes de serviços e seguindo as boas práticas do Angular 2, temos mais uma vantagem, fora as já descritas

anteriormente, a **reutilização de código**. Quando um método for usado em mais de um componente, este deve sair do componente e ficar em uma classe de serviço. Fazendo isso, podemos importar essa nova classe para dentro dos componentes e usá-la normalmente em cada um deles.

Escrevemos o método somente uma vez, e usamos quantas vezes necessário. Assim, diminuímos linhas de código no projeto e, consequentemente, seu tamanho final. Há também uma melhora na manutenção e nos testes, pois, com o método somente em um lugar, teremos de alterar uma vez e todos os componentes estarão atualizados.

Para completar esse tópico, lembra do método `msgAlerta()` que fizemos dentro do `app.component.ts` do nosso projeto? Ele foi colocado como evento de clique do botão, e sua função é colocar um alerta na tela com a frase `Livro Angular 2`.

Imagine que esse método seja usado em mais quatro componentes diferentes. Será que teremos de fazer quatro vezes o mesmo método? Não! Vamos criar uma classe de serviço com ele e, sempre que precisarmos colocar alguma informação na tela, usaremos essa classe serviço, que terá esse método de alerta. Prático e simples, certo?

Para criar uma classe serviço, usaremos os comandos do *Angular CLI*. Vamos utilizar o comando `ng g s` ou `ng generate service`, em que:

	Comando	Descrição	
	<code>ng</code>	Comando do Angular CLI	
	<code>g</code>	Abreviação do <i>generate</i>	

	s	Abreviação do <i>service</i>	
--	---	------------------------------	--

O nome que vamos usar é **alerta**, então criaremos nosso serviço com o comando `ng g s alerta`, ou `ng generate service alerta`. Ambos terão o mesmo resultado.

Após a classe de serviço criada, receberemos duas confirmações de criação de arquivos, `alerta.service.spec.ts` e `alerta.service.ts`, junto de um aviso: **Service is generated but not provided, it must be provided to be used**. Guarde esse erro, pois já falaremos dele!

Vamos seguir nas mudanças e modificar o método `msgAlerta()` para o serviço que fizemos. A nova classe serviço `alerta.service.ts` ficará assim:

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

@Injectable()
export class AlertaService {

  constructor() { }

  msgAlerta(): void {
    alert('Livro Angular 2');
  }
}
```

Agora, pelo prompt de comando do VS code, vamos entrar na pasta do componente `lista-pessoa` e criar uma classe de serviço com o nome de `pessoa-service`.

No prompt, digitaremos `cd src` e daremos `Enter`. Depois, no mesmo prompt, vamos digitar `cd app` e `Enter`, e depois digitar `cd lista-pessoa` e `Enter` novamente. Agora estamos

dentro da pasta do componente de lista de pessoas, onde vamos criar uma classe de serviço da mesma forma que criamos a classe de alerta, ng g s `pessoa-service` .

Nossa pasta do componente `lista-pessoa` ficará assim:

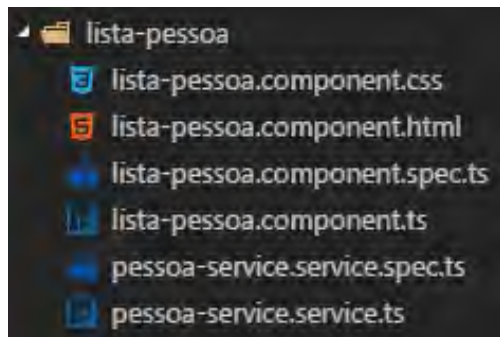


Figura 3.13: Pasta do componente `lista-pessoa`

Para finalizar, vamos falar sobre injeção de dependência mais à frente. Porém, precisamos primeiro saber injetar a classe de serviço dentro de um componente. No término deste tópico, faremos a injeção do serviço nos componentes e você verá que ficará um código muito mais reaproveitável.

Com os **serviços** devidamente apresentados, colocamos mais uma função em nosso diagrama.

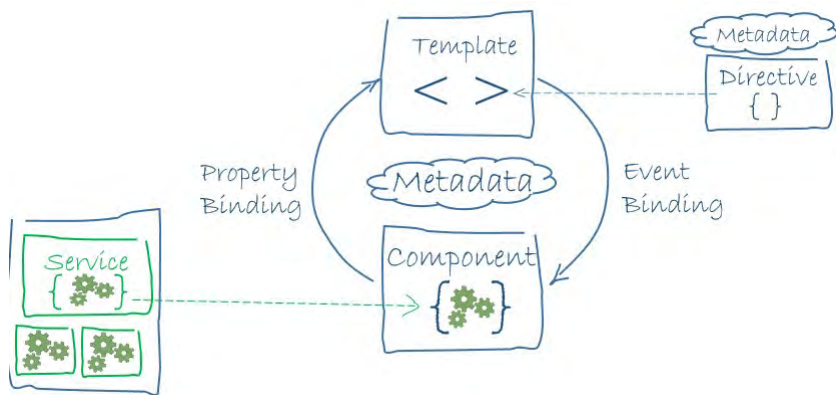


Figura 3.14: Diagrama completo

3.10 INJEÇÃO DE DEPENDÊNCIA

Temos uma situação: a classe `app.component.ts` quer usar recursos que existem na classe `alerta.service.ts`. Para usar a classe de alerta dentro da classe `app.component.ts`, teremos de injetar uma dependência da classe alerta dentro da classe `app.component.ts`. Isso diz ao Angular 2 a seguinte situação: *para o componente funcionar corretamente, ele depende de uma injeção da classe alerta.*

Para utilizar os recursos da classe alerta, precisamos injetá-la dentro da classe do componente. Assim, formamos a **injeção de dependência** que nada mais é do que declarar e injetar dentro da classe do componente uma classe de serviço.

Geralmente, injetamos classes de serviços dentro dos componentes para usar os recursos que cada serviço disponibiliza. Podemos fazer a instância de um serviço manualmente, ou deixar para o Angular 2 cuidar do gerenciamento das instâncias.

Voltando à classe `app.component.ts`, faremos uma instância de `AlertaService`, da seguinte forma:

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

import { AlertaService } from '../alerta.service';

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

  title: string = 'Livro Angular 2';
  foto: string = 'favicon.ico';

  constructor(private service: AlertaService) {

  }
}
```

Veja que, quando criamos um componente, declaramos no seu construtor (`constructor`) que, para ele funcionar corretamente, necessita de uma instância de `AlertaService`. Declarando dessa forma, você diz ao Angular 2 gerenciar todas as instâncias de serviços existentes no projeto. Esta é a forma correta de se fazer, mas vamos ver de outra forma.

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

import { AlertaService } from '../alerta.service';

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

  title: string = 'Livro Angular 2';
```

```
foto: string = 'favicon.ico';

constructor() {
  private service = new AlertaService;
}
}
```

Veja que, dessa forma, estamos instanciando a classe `AlertaService` manualmente, e isso no Angular 2 não é uma boa prática. Também podemos ter problemas com esse tipo de declaração manualmente, por exemplo, a mesma classe sendo instanciada várias vezes, consumindo mais memória do computador.

Para injetar uma dependência nos componentes e nos serviços, vamos instanciar **sempre** na declaração do método construtor. Falei aqui que, *para injetar uma dependência nos componentes e nos serviços*, podemos injetar serviço dentro de serviço? Sim, podemos, e vamos ver isso em outras partes do livro.

Declarando todas as dependências no construtor, além de o código ficar mais organizado pois todas ficam no mesmo lugar, o Angular 2 pode verificar de quais dependências o componente precisa, só de ver os parâmetros do construtor.

Quando o Angular 2 vai criar um componente, ele verifica quais são suas dependências e solicita a um *gerenciador de injeção* todas as instâncias necessárias. O gerenciador de injeção é um *container* que guarda todas as instâncias ativas no projeto.

Se uma instância já existe — ou seja, já foi criada anteriormente por outro componente —, o gerenciador pega essa instância existente e adiciona no segundo componente. Caso a dependência ainda não tenha sido criada, ele criará uma nova

de módulo dentro da aplicação.

O `metadata declarations` é onde vamos colocar todas as classes e componentes que criaremos. A cada novo componente ou nova classe criada para o projeto, será dentro deste array de *declarations* que vamos colocar o nome do objeto para poder usarmos no projeto.

O `metadata imports` é onde colocaremos todos os módulos importados para o nosso projeto usar. Neste local, colocamos os módulos internos do Angular 2 e de terceiros, caso nosso projeto tenha. Tudo o que for usado no projeto que seja do próprio Angular 2 será declarado neste array.

O `metadata providers` será onde vamos declarar todos os serviços disponíveis na aplicação. Sempre que fizermos um novo serviço e quisermos deixá-lo disponível para todo o projeto, será nesse array que declararemos sua classe.

O `metadata bootstrap` diz ao Angular 2 por onde começar a renderizar os componentes. O componente inicial do projeto será declarado neste array. Isso diz ao Angular 2 que, quando a aplicação for inicializada, queremos que ele comece a renderização dos componentes no navegador por esse componente declarado.

Sempre temos de fazer duas coisas em conjunto: importar o objeto informando o caminho dentro do projeto com a declaração nas primeiras linhas do arquivo, e também declarar dentro de algum array no decorador `@NgModule`. Seguindo esses passos, o componente, o serviço, a classe ou o módulo ficará disponível para toda a aplicação.

E agora? Precisamos corrigir o problema do serviço que

```

    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
  })
  export class AppComponent {

    title: string = 'Livro Angular 2';
    foto: string = 'favicon.ico';

    constructor(private service: AlertaService) { }

    enviarMsg(): void {
      this.service.msgAlerta();
    }
  }
}

```

Veja que agora nosso componente não envia um alerta diretamente. Chamamos um método que está dentro de uma classe serviço e o serviço que vai executar a função.

Para o serviço funcionar corretamente, precisamos importar o arquivo para dentro do `app.component`, então vamos colocar na linha 3 o seguinte código:

```
import { AlertaService } from './alerta.service';
```

Vamos ao arquivo `app.component.html` e mudamos a chamada do event binding, que está dentro da tag `button`. Colocaremos o mesmo método que está no `app.component.ts`.

```
<button (click)="enviarMsg()">Enviar Alerta</button>
```

Depois de salvar as alterações, volte ao navegador que está rodando o projeto e veja o que aconteceu. Aconteceu um erro? Não está aparecendo nada, correto? Agora aperte o botão `F12` do teclado. Abrirá uma aba na parte inferior do navegador; nela há um menu, clique em `console`.

Você verá várias linhas em vermelho. Vá até o começo delas e

veja o erro: **EXCEPTION: No provider for AlertaService!.** Viu?

O sistema está lhe alertando que não existe um serviço provido para o `AlertaService`. Mas como não existe se nós o fizemos? Você já percebeu o erro? Vamos analisar os fatos e achá-lo.

Quando criamos o serviço com o comando `ng g s alerta`, você lembra de que, após a confirmação de criação de duas classes, foi enviado um alerta de *Service is generated but not provided, it must be provided to be used?* Isso quer dizer que: **o serviço foi criado, mas deve ser fornecido**, ou seja, **deve ser declarado como provedor**.

Quando rodamos o projeto, vimos que no navegador tivemos um erro parecido: *EXCEPTION: No provider for AlertaService!* Ou seja, o serviço não foi fornecido.

Com os erros, tudo fica bem claro, **temos de fornecer o serviço para o projeto**. Então, vamos declarar o `AlertaService` como provedor, mas onde vamos declarar o serviço? No componente ou global no projeto? Você se lembra da regra?

Se o serviço for utilizado por mais de 1 componente, temos de declarar globalmente no `app.module.ts`, já se for usado por somente 1 componente, vamos declarar dentro dele próprio. Então, colocaremos esse serviço como global, assim outros componentes poderão enviar alerta também.

Vamos aprender a declarar um serviço como global no próximo tópico, quando abordaremos o `app.module` do projeto e, mais precisamente, o `NgModule`.

Com a **injeção de dependência** devidamente apresentada,

colocamos mais uma função em nosso diagrama.

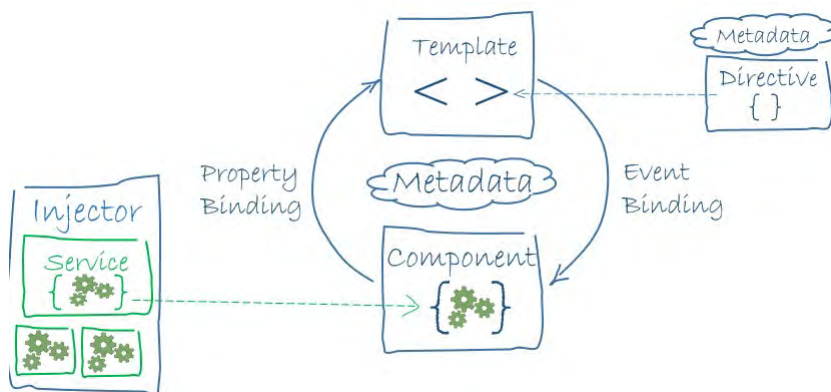


Figura 3.15: Diagrama completo

3.11 NGMODULE

O conceito de módulos é: *juntar alguns componentes, serviços, classes que têm o mesmo propósito e características em blocos*, chamados de módulos. O conjunto desses módulos vão compor uma aplicação.

Um exemplo pode ser o nosso componente de listar pessoas que, dentro da pasta `lista-pessoa`, temos os arquivos HTML, CSS, classe do componente e a classe de serviço, com o mesmo propósito: *listar nomes de pessoas*. Cada um tem sua função dentro do componente e todos juntos formam um módulo de listar nomes de pessoas.

Cada aplicação em Angular 2 tem, no mínimo, um arquivo de configuração de módulo que é chamado de `AppModule`. Ele é responsável pela configuração global do projeto.

Para determinar ao Angular 2 que a classe `AppModule` seja considerada como uma classe de configuração de módulo, temos de colocar o decorador `@NgModule`. Ele facilita a importação das classes e o reuso com uma melhor organização da aplicação.

O próprio Angular 2 é feito em módulos, em que cada um é responsável por uma função dentro do framework. E quando queremos usar algo do Angular 2, precisamos importar para a nossa aplicação o módulo desta função através de `imports` parecidos com os do JavaScript.

```
import { nome da classe } from 'caminho do local do arquivo';
```

Assim informamos ao Angular 2 que a classe precisa de algum módulo que estará no caminho especificado. Abra os arquivos `app.component.ts`, `app.module.ts` e `alerta.service.ts`, e veja logo na parte de cima, nas primeiras linhas, as importações de cada módulo ou arquivo que a classe necessita.

Sempre que for usar um objeto dentro de outro, teremos de fazer esse `import` no início do arquivo. Isso mostrará ao Angular 2 onde ele vai buscar as informações que o objeto importado tem disponível.

Segundo as boas práticas de desenvolvimento do Angular 2, sempre que importarmos algo para dentro de um arquivo, teremos de seguir as regras:

Os módulos do próprio Angular 2 ficarão nas primeiras linhas do arquivo.

Arquivos ou módulos de terceiros ficarão abaixo, **sempre** pulando uma linha para não juntar com os arquivos do próprio Angular 2.

Os arquivos de componentes, serviços ou classes que fizemos dentro do projeto ficarão abaixo, **sempre** pulando uma linha para não juntar com os arquivos de terceiros.

Seguindo essas regras, os arquivos ficarão separados e organizados dentro da aplicação. Dessa forma, fica fácil saber de onde vem cada `import` existente no arquivo.

```
import { classe-angular } from './caminho';
import { classe-angular } from './caminho';
import { classe-angular } from './caminho';

import { arquivo-terceiro } from './caminho';
import { arquivo-terceiro } from './caminho';
import { arquivo-terceiro } from './caminho';

import { meu-componente } from './caminho';
import { meu-service } from './caminho';
import { minha-class } from './caminho';
```

Voltamos ao decorador `@ngModule`, dentro do arquivo `app.module.ts`. Vamos conhecê-lo.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Como todo decorador dentro do Angular 2, temos de colocar os metadados. Vamos verificar quais são os metadados do decorador

de módulo dentro da aplicação.

O `metadata declarations` é onde vamos colocar todas as classes e componentes que criaremos. A cada novo componente ou nova classe criada para o projeto, será dentro deste array de *declarations* que vamos colocar o nome do objeto para poder usarmos no projeto.

O `metadata imports` é onde colocaremos todos os módulos importados para o nosso projeto usar. Neste local, colocamos os módulos internos do Angular 2 e de terceiros, caso nosso projeto tenha. Tudo o que for usado no projeto que seja do próprio Angular 2 será declarado neste array.

O `metadata providers` será onde vamos declarar todos os serviços disponíveis na aplicação. Sempre que fizermos um novo serviço e quisermos deixá-lo disponível para todo o projeto, será nesse array que declararemos sua classe.

O `metadata bootstrap` diz ao Angular 2 por onde começar a renderizar os componentes. O componente inicial do projeto será declarado neste array. Isso diz ao Angular 2 que, quando a aplicação for inicializada, queremos que ele comece a renderização dos componentes no navegador por esse componente declarado.

Sempre temos de fazer duas coisas em conjunto: importar o objeto informando o caminho dentro do projeto com a declaração nas primeiras linhas do arquivo, e também declarar dentro de algum array no decorador `@NgModule`. Seguindo esses passos, o componente, o serviço, a classe ou o módulo ficará disponível para toda a aplicação.

E agora? Precisamos corrigir o problema do serviço que

fizemos no tópico anterior, lembra?

Tínhamos parado sabendo que o problema era no serviço, mas a codificação do nosso serviço está correta. O erro estava dizendo que este serviço que queremos usar não está fornecido, ou seja, o Angular 2 não está reconhecendo.

Temos de informar ao Angular 2 que nossa classe serviço `alerta.service.ts` faz parte do projeto. E como vamos fazer isso?

Veja bem, acabamos de falar sobre os conceitos do `@ngModule`, seu decorador e seus metadatos. Vimos que, dentro desse decorador, existe um metadata que serve para declarar os serviços disponíveis dentro do projeto. Veja agora como está este array? Vazio!

Lembra de quando criamos o `service` dentro do *VS Code* com o comando `ng g s alerta` e, no final, havia dado uma mensagem de alerta: *Service is generated but not provided, it must be provided to be used?* Quando criamos classes de serviços, o Angular 2 não sabe onde colocá-las, se dentro de um módulo específico no projeto, ou de um componente, ou dentro do projeto de forma global.

Como uma classe serviço serve para uma infinidade de coisas, o Angular 2 não sabe ao certo o que vamos fazer com ele. Então, o framework simplesmente cria a classe dentro do projeto e manda a mensagem: *Olha, eu já criei o que você pediu, mas você deve declarar essa classe em algum lugar.*

É isso que faremos para nosso projeto voltar a funcionar. Vamos declarar dentro do array `providers` o serviço que fizemos

e também importar o arquivo informando o caminho dentro do projeto.

Nosso arquivo `app.module.ts` ficará assim:

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

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

import { AlertaService } from './alerta.service';

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

Salvando o arquivo, volte ao navegador e veja o nosso projeto rodando novamente. Clique no botão e veja o alerta sendo enviado ao navegador como estava antes. Entretanto, agora a mensagem está vindo do serviço e não do componente. Com essa mudança, caso um novo componente seja criado, ele também poderá usar essa mensagem sem que precise recriar um novo método.

Uma parte do projeto já está funcionando, agora vamos modificar o componente de lista de pessoas. Inicialmente, mudaremos o conteúdo do array de pessoas para dentro da classe de serviço `pessoa-service.service.ts`.

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

@Injectable()
export class PessoaServiceService {

  constructor() { }

  getPessoas(): string [] {
    return ['João', 'Maria', 'Angular 2', 'Thiago'];
  }
}
```

Vamos injetar no arquivo `lista-pessoa.component.ts` o serviço que enviará os nomes de pessoas. Em seguida, dentro do método construtor, vamos atribuir o retorno do serviço na variável `pessoas`, dentro da nossa classe.

```
@Component({
  selector: 'app-lista-pessoa',
  templateUrl: './lista-pessoa.component.html',
  styleUrls: ['./lista-pessoa.component.css']
})
export class ListaPessoaComponent implements OnInit {

  pessoas: string [];
  nome: string = "Thiago";

  constructor(private service: PessoaServiceService) {
    this.pessoas = service.getPessoas();
  }

  ngOnInit() {
  }

  listar() {
  }
}
```

Como essa classe de serviço, somente o componente `lista-pessoa` poderá usar. Vamos declarar dentro do próprio componente este serviço, assim, ele só será instanciado caso

lista-pessoa seja utilizado.

Dentro do decoration `@Component` , declararemos esta classe de serviço:

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

import { PessoaServiceService } from '../pessoa-service.service';

@Component({
  selector: 'app-lista-pessoa',
  templateUrl: './lista-pessoa.component.html',
  styleUrls: ['./lista-pessoa.component.css'],
  providers: [PessoaServiceService]
})
export class ListaPessoaComponent implements OnInit {

  pessoas: string [];
  nome: string = "Thiago";

  constructor(private service: PessoaServiceService) {
    this.pessoas = service.getPessoas();
  }

  ngOnInit() {
  }

  listar() {
  }
}
```

Salve tudo e volte ao navegador para verificar o resultado. Para ter certeza de que está vindo do serviço, adicionei um novo nome no final do array na classe do serviço.

3.12 MELHORANDO NOSSO COMPONENTE

Nosso projeto voltou a funcionar, e já adicionamos os serviços e toda a informação está vindo de lá. Mas agora vamos melhorar

nosso componente e fazê-lo enviar dados para o serviço.

Vamos colocar um novo botão no HTML do componente `lista-pessoa` e, dentro do botão, um event binding de `click`.

```
<h2>Lista de Pessoas</h2>

<input type="text" [(ngModel)]="nome">
<button (click)="enviarNome()">Enviar Nome</button>
<p>{{nome}}</p>

<ul>
  <li *ngFor="let pessoa of pessoas">
    {{pessoa}}
  </li>
</ul>
```

Dentro da nossa classe de serviço, criaremos um novo método que vai receber um nome e adicioná-lo no novo array, dentro do serviço. Para isso, vamos modificar algumas coisas dentro do `pessoa-service.service.ts`.

Criaremos um array `nomesPessoas`, adicionamos o conteúdo inicial dentro desse array e, no método `getPessoas`, retornaremos o array criado.

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

@Injectable()
export class PessoaServiceService {

  nomesPessoas: string [] = ['João', 'Maria', 'Angular 2', 'Thiago'];

  constructor() { }

  getPessoas(): string [] {
    return this.nomesPessoas;
  }
}
```

Agora vamos criar um novo método que vai adicionar nomes dentro desse novo array, vamos chamar de `setPessoa` .

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

@Injectable()
export class PessoaServiceService {

  nomesPessoas: string [] = ['João', 'Maria', 'Angular 2', 'Thiago'];

  constructor() { }

  getPessoas(): string [] {
    return this.nomesPessoas;
  }

  setPessoa(nome: string): void {
    this.nomesPessoas.push(nome);
  }
}
```

Voltamos para o arquivo `lista-pessoa.component.ts` . Criaremos um método que terá o nome de `enviarNome` , que enviará o nome para a classe serviço.

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

import { PessoaServiceService } from '../pessoa-service.service';

@Component({
  selector: 'app-lista-pessoa',
  templateUrl: './lista-pessoa.component.html',
  styleUrls: ['./lista-pessoa.component.css'],
  providers: [PessoaServiceService]
})
export class ListaPessoaComponent implements OnInit {

  pessoas: string [];
  nome: string = "Thiago";

  constructor(private service: PessoaServiceService) {
    this.pessoas = service.getPessoas();
  }
}
```

```

    }

    ngOnInit() {
    }

    enviarNome() {
        this.service.setPessoa(this.nome);
    }
}

```

Salvando todos os arquivos e voltando ao navegador, se digitarmos alguma coisa dentro do `input` e clicarmos no botão `Enviar nome`, o conteúdo que está dentro do `input` será enviado para a classe serviço `pessoa-service.service.ts` pelo método `setPessoa`. Lá ele será adicionado ao array `nomesPessoas`.

Como na injeção de dependência já estamos vinculando o método `getPessoas` da classe serviço, no array `pessoas` da classe `lista-pessoa.component.ts`, a modificação do array no serviço já vai refletir na lista mostrada no navegador.

3.13 RESUMO

Neste capítulo, fizemos uma descrição de tudo o que temos no projeto Angular 2. Sabemos os conceitos e para que serve cada arquivo, e podemos dividir o projeto em: arquivos de configuração, arquivos de testes e arquivos para desenvolvimento dos componentes que serão usados na aplicação.

Sabemos todo o processo de renderização, desde a requisição do usuário no servidor pela página `index.html` até a renderização total da aplicação. Vimos as partes que constroem o projeto, a função de cada uma delas, e como desenvolver e utilizá-

las.

Neste momento, temos uma visão geral da aplicação, sabendo o que é e para que serve cada parte. Agora podemos aprofundar o conhecimento de cada funcionalidade dentro de um projeto em Angular 2.

Nos próximos capítulos, vamos falar em detalhe cada tópico apresentado até aqui, e criar um componente para cada assunto abordado. Mostraremos passo a passo a utilização e, no final, vamos juntar todos os conhecimentos apresentados e criar um projeto final, que será uma agenda de contatos usando tudo o que Angular 2 tem para oferecer. Então, seguimos nos estudos!

EXIBINDO DADOS PARA O USUÁRIO

Já sabemos que a exibição de dados é feita pelo template, e que ele é construído com HTML e algumas marcações do Angular 2. Praticamente, todas as tags do HTML são válidas e usadas, mas algumas são proibidas ou não faz sentido usá-las dentro dos componentes, como por exemplo, as tags `script`, `html` e `body`.

A tag `script` é proibida dentro dos componentes do Angular 2 por segurança. A proibição de usar essa tag é para evitar injeções de `script` dentro da aplicação, e também por não fazer muito sentido usá-la, já que tudo o que estamos desenvolvendo está no TypeScript. Caso alguma tag `script` seja usada dentro da aplicação, o próprio Angular 2 vai ignorá-la.

As tags `html` e `body` não são utilizadas dentro de componentes porque elas já estão no arquivo `index.html`, e todos os componentes são renderizados dentro da tag `body` do `index.html` da aplicação. Então, colocar `html` e `body` dentro de algum componente não faz sentido algum também. Já o restante das tags conhecidas no HTML poderá ser usado normalmente, com ou sem as marcações do Angular 2.

No capítulo anterior, demos uma introdução sobre cada parte que compõe uma aplicação em Angular 2. Agora nos próximos capítulos, vamos falar em detalhes sobre cada parte e como podemos usá-las.

4.1 INTERPOLATION

Já vimos o conceito de interpolação, agora veremos o que podemos fazer dentro do template com ela. Conhecemos a interpolação usando `{{ variável }}`, e usamos a interpolação para juntar o conteúdo de uma variável que está na classe do componente com a tag HTML do template.

O Angular 2 substitui o local que está com `{{ }}` pelo conteúdo da variável que está na classe do componente. O que o Angular 2 faz antes de exibir o conteúdo é uma **expressão de template**, em que ele avalia o que está dentro dos `{{ }}` e converte para *string*, para depois ser adicionado no HTML.

Vamos criar um novo componente, com o nome de `interpolation-binding`. A cada novo exemplo, criaremos um novo componente para deixar tudo bem separado e organizado, assim praticaremos o uso deste recurso dentro da aplicação.

Quando nomeamos um componente com nome composto, colocamos o `-` (hífen) para separar cada parte do nome. Isso é outra boa prática do Angular 2.

Após a criação do novo componente, com o comando `ng g c interpolation-binding`, será criada uma pasta com os quatro arquivos cujas funções já foram descritas. Vamos abrir o arquivo `interpolation-binding.component.html` e, no lugar de

interpolation-binding works! , colocaremos uma tag `h2` com o conteúdo `Interpolation binding` . Essa será nossa regra a cada componente criado, vamos sempre adicionar um título com o nome do componente.

Ainda no `interpolation-binding.component.html` , podemos fazer um simples exemplo de soma dentro da interpolação. Veja o código.

```
<h2>Interpolation binding</h2>
```

```
<p> a soma é {{1 + 1}} </p>
```

No arquivo `app.component.html` , adicionamos a tag `<app-interpolation-binding></app-interpolation-binding>` que vai dizer ao Angular 2 para renderizar dentro do `app.component.html` uma instância do `interpolation-binding.component` .

No mesmo arquivo `app.component.html` , vamos comentar as tags `img` , `button` e `<app-lista-pessoa></app-lista-pessoa>` , pois agora queremos focar somente no componente de interpolação que acabamos de criar.

Nosso arquivo `app.component.html` ficará assim:

```
<h1>
  {{title}}
</h1>

<!-- <img [src]="foto">

<button (click)="enviarMsg()">Enviar Alerta</button>

<app-lista-pessoa></app-lista-pessoa> -->

<app-interpolation-binding></app-interpolation-binding>
```

Se você voltar ao navegador, verá somente o título `Livro Angular 2` e dois parágrafos, um escrito `Interpolation binding` e um parágrafo escrito `a soma é 2`. Isso mostra que o Angular 2 primeiro avalia a expressão para depois converter o conteúdo em string e, no final, adicionar na tag do HTML.

Neste caso, fizemos uma simples conta, mas também podemos chamar métodos da classe do componente direto nas interpolações. Veja outro exemplo: teremos dois métodos que estão na classe do componente: o primeiro retorna uma `string`, e o segundo retorna um `number` com o conteúdo `6`.

No `interpolation-binding.component.html`, vamos colocar os parágrafos:

```
<p> meu livro é {{getLivro()}} </p>

<p> outra soma é {{1 + getNumero()}} </p>
```

No `interpolation-binding.component.ts`, adicionaremos dois métodos.

```
getLivro(): string {
    return 'Angular 2';
}

getNumero(): number {
    return 6;
}
```

Com a interpolação, podemos tanto colocar somente o valor de uma variável que está na classe do componente como também uma expressão complexa com métodos que retornam valores. No final, tudo será avaliado como uma *expressão de template*.

As expressões de template podem ser usadas de várias formas

dentro do HTML do componente, mas algumas são proibidas pelo próprio framework do Angular 2. São elas:

Atribuições — `+=` , `-=` , `variável = valor`

Instanciação — `new`

Incremento ou decremento — `++` , `--`

Embora possamos fazer várias combinações com as interpolações, devemos seguir algumas regras e boas práticas para não prejudicar a experiência do usuário quando for usar nossa aplicação. Essas regras são: **execução rápida e simplicidade**.

Respeitando uma regra, automaticamente você respeitará a outra. Temos de sempre levá-las em consideração quando fazemos alguma interpolação dentro do HTML.

O Angular 2 constantemente está executando comandos enquanto o usuário usa nossa aplicação. E quando fazemos uma expressão de template complexa dentro de uma interpolação, a aplicação pode demorar muito para executar, especialmente em máquinas mais lentas.

Uma maneira de não comprometer a usabilidade da aplicação é executar interpolações simples e rápidas como: chamadas de métodos da classe do componente, buscar valores de variáveis, ou alguns cálculos mais simples.

Quando tivermos alguma interpolação mais complexa que possa demorar para ser finalizada, podendo comprometer a execução da aplicação, considere deixar os valores armazenados dentro da classe do componente em uma variável. Quando tiver um processo muito demorado ou muito complexo, faça isso dentro de um método na classe do componente, e deixe armazenado em

uma variável. Assim, o template somente buscará o valor da variável e sua execução ficará muito mais leve.

Nosso arquivo `interpolation-binding.component.html` ficou assim:

```
<h2>Interpolation binding</h2>

<p> a soma é {{1 + 1}} </p>

<p> meu livro é {{getLivro()}} </p>

<p> outra some é {{1 + getNumero()}} </p>
```

Nosso arquivo `interpolation-binding.component.ts` ficou assim:

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

@Component({
  selector: 'app-interpolation-binding',
  templateUrl: './interpolation-binding.component.html',
  styleUrls: ['./interpolation-binding.component.css']
})
export class InterpolationBindingComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  getLivro(): string {
    return 'Angular 2';
  }

  getNumero(): number {
    return 6;
  }
}
```

4.2 PROPERTY BINDING

Antes de tudo, já vamos criar um novo componente com o nome de `property-binding` e, após a criação, adicionar a nova tag `<app-property-binding>` `</app-property-binding>` no arquivo `app.component.html`. Vamos seguir os mesmos passos que o tópico anterior:

Comentar a tag do tópico anterior.

Dentro do novo componente, trocar a primeira tag `<p>` `</p>` por `h2` e, dentro, o título do novo componente, que será `Property binding`.

Voltamos ao navegador, e veremos o título do livro e o título do nosso componente. Veja nosso arquivo `app.component.html`:

```
<h1>
  {{title}}
</h1>

<!-- <img [src]="foto">

<button (click)="enviarMsg()">Enviar Alerta</button>

<app-lista-pessoa></app-lista-pessoa> -->

<!-- <app-interpolation-binding></app-interpolation-binding> -->

<app-property-binding></app-property-binding>
```

Como já foi falado, o *property binding* é um tipo de ligação de dados para definir alguma propriedade do elemento HTML com valores que estão na classe do componente. No próximo exemplo, vamos adicionar um parágrafo e um botão. Quando o botão for clicado, o parágrafo será mostrado ou escondido. Veja o código a seguir.

```
<h2>Property binding</h2>
```

```
<button (click)="mostrar()">Mostrar Paragrafo</button>
```

```
<p [hidden]="!verParagrafo">Paragrafo habilitado</p>
```

No arquivo `property-binding.component.ts`, criaremos uma variável `verParagrafo` e um método `mostrar()`.

```
verParagrafo: boolean = true;
```

```
mostrar(): boolean {  
  this.verParagrafo = ! this.verParagrafo;  
}
```

Salvando os arquivos e voltando ao navegador, veremos que a cada clique no botão a variável `verParagrafo` vai atribuir o valor de `true` ou `false` para a propriedade `hidden` da tag `p` e, consequentemente, o parágrafo será visualizado ou escondido dependendo do valor da variável `verParagrafo`.

A propriedade que fica dentro dos `[]` é a que queremos manipular, e o que está entre `" "` é o valor que queremos passar para o HTML. Essa forma de codificação é uma forma simplificada, pois, por baixo dos panos, o Angular 2 transforma o `[]` em uma ligação de dados **bind**, para depois receber o valor desejado.

Se fizermos o mesmo exemplo com a forma inteira de codificação, o *property binding* ficará assim:

```
<h2>Property binding</h2>
```

```
<button (click)="mostrar()">Mostrar Paragrafo</button>
```

```
<p [hidden]="!verParagrafo">Paragrafo habilitado</p>
```

```
<p bind-hidden="!verParagrafo">Paragrafo habilitado com bind</p>
```

Veja que, da mesma forma que `bind`, o `bind` funciona corretamente. O tipo de codificação será de sua escolha, pois os dois fazem o mesmo papel, porém o recomendado é com `bind` entre a propriedade do elemento para ser rapidamente reconhecido como um property binding.

Assim como o interpolation, o property binding também aceita retornos de métodos para atribuir na propriedade do elemento no HTML. O mais importante e fundamental quando usamos o property binding é **não esquecer dos colchetes** (`[]`).

Caso em algum momento você se esqueça de colocar os `[]` em volta da propriedade, o Angular 2 vai considerar a atribuição de uma string para ela, e o componente não será renderizado.

Adicione outra tag `p` com a propriedade `hidden`, sem os `[]`.

```
<h2>Property binding</h2>

<button (click)="mostrar()">Mostrar Paragrafo</button>

<p [hidden]="!verParagrafo">Paragrafo habilitado</p>
<p bind-hidden="!verParagrafo">Paragrafo habilitado com bind</p>
<p hidden="!verParagrafo">Paragrafo habilitado com interpolation
sem []</p>
```

Veja que somente os dois primeiros parágrafos serão mostrados. Como tiramos os `[]` em volta da propriedade `hidden`, o Angular 2 não reconhece o terceiro parágrafo como uma atribuição válida, pois essa propriedade espera receber um booleano, e não uma string.

Aprendemos no capítulo anterior que o interpolation e o property binding têm as mesmas características, que é passar

valores da classe do componente para o template. Podemos usar um ou outro em vários casos e não teremos erros. Porém, para seguir o padrão de desenvolvimento recomendado pela equipe do Angular 2 e também aceito pela sua comunidade, vamos usar a interpolação quando for para mostrar conteúdo no navegador e o property binding quando for para passar valores para propriedades dos elementos HTML do template.

Nosso arquivo `property-binding.component.html` está assim:

```
<h2>Property binding</h2>

<button (click)="mostrar()">Mostrar Paragrafo</button>

<p [hidden]="!verParagrafo">Paragrafo habilitado</p>
<p bind-hidden="!verParagrafo">Paragrafo habilitado com bind</p>
<p hidden="!verParagrafo">Paragrafo habilitado com interpolation
sem []</p>
```

Nosso arquivo `property-binding.component.ts` :

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

@Component({
  selector: 'app-property-binding',
  templateUrl: './property-binding.component.html',
  styleUrls: ['./property-binding.component.css']
})
export class PropertyBindingComponent implements OnInit {

  verParagrafo: boolean = true;

  constructor() { }

  ngOnInit() {
  }

  mostrar() {
    this.verParagrafo = ! this.verParagrafo;
  }
}
```

```
}  
}
```

4.3 TWO-WAY DATA BINDING

O **two-way data binding** ajuda bastante para fazer atualizações de uma variável que está no template e na classe do componente ao mesmo tempo. Já sabemos como podemos usar e sua representação, agora vamos estudar mais a fundo como funciona toda a dinâmica que o envolve.

Novamente, vamos criar um novo componente, desta vez com o nome de `two-way-binding` — lembre-se sempre de colocar o - para nome composto. Logo em seguida, vamos adicionar a tag no arquivo `app.component.html` e comentar a tag anterior.

Não esqueça de adicionar o título do componente no arquivo `two-way-binding.component.html` :

```
<h2>two-way data binding</h2>
```

Vimos no capítulo anterior que a forma de usar o two-way data binding é com `[(ngModel)]="variavel"` , mas existem outros tipos para se fazer a mesma coisa. É claro que a forma recomendada é com o `[(ngModel)]` , mas mesmo assim vamos estudar outras possíveis soluções.

Sabemos que o `ngModel` atualiza a variável tanto no template quanto na classe do componente, e que para isso é usado o *event binding* e o *property binding*. Quando usamos o `ngModel` , estamos abreviando duas chamadas em uma. Por baixo dos panos, o Angular 2 faz o *property binding* `[(ngModel)]="variavel"` e o *event binding* `(ngModelChange)="meuEvento($event)"` . Vamos

testar isso agora e ver a diferença.

Vamos colocar um `input` logo abaixo da tag `h2` e, dentro desse `input`, colocamos os dois tipos de ligação de dados:

```
<h2>two-way data binding</h2>
```

```
<input type="text" [ngModel]="nome" (ngModelChange)="mudar($event)">
```

```
<p>{{nome}} </p>
```

No arquivo `two-way-binding.component.ts`, criaremos uma variável `nome` e um método chamado `mudar()`.

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

@Component({
  selector: 'app-two-way-binding',
  templateUrl: './two-way-binding.component.html',
  styleUrls: ['./two-way-binding.component.css']
})
export class TwoWayBindingComponent implements OnInit {

  nome: string = "Thiago";

  constructor() { }

  ngOnInit() {
  }

  mudar(valor) {
    this.nome = valor;
  }
}
```

Fazendo dessa forma, separamos o *property binding* do *event binding*. Se voltarmos ao navegador, o resultado será o mesmo.

Mas quando vamos usar a forma separada do `ngModel`? Quando queremos comportamentos diferentes para cada evento.

Dentro do método `mudar()` , vamos adicionar um `-` ao lado do `valor` .

```
mudar(valor) {  
  this.nome = valor + '-';  
}
```

Salve e volte ao navegador. Digite alguma coisa dentro da caixa de texto e veja o que aconteceu.

Nós modificamos o *event binding* e colocamos um `-` no final de cada digitação, mas o *property binding* continuou da mesma forma.

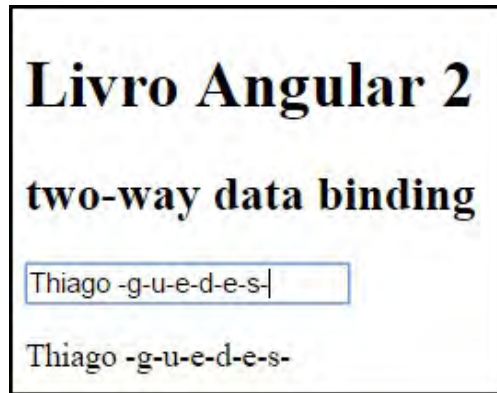


Figura 4.1: Event binding e property binding separados no `ngModel`

Também podemos usar a forma literal `ngModel` , colocando o `bindon-` na frente do `ngModel` . Para exemplificar, vamos colocar outro `input` , e no lugar do `[(ngModel)]` , coloque `bindon-ngModel="nome"` . Salve e volte ao navegador para ver o funcionamento.

```
<h2>two-way data binding</h2>
```

```
<input type="text" [ngModel]="nome" (ngModelChange)="mudar($event
```

```

)">
<p>{{nome}} </p>

<input type="text" bindon-ngModel="nome">
<p>{{nome}} </p>

```

Essas são as formas como podemos trabalhar com `ngModel`. Na maior parte vamos usar a forma simplificada `[(ngModel)]`, mas podemos usar os eventos separados dependendo da regra de negócio, ou para uma validação de conteúdo digitado pelo usuário.

Nosso arquivo `two-way-binding.component.html` ficou assim:

```

<h2>two-way data binding</h2>

<input type="text" [(ngModel)]="nome" (ngModelChange)="mudar($event)">
<p>{{nome}} </p>

<input type="text" bindon-ngModel="nome">
<p>{{nome}} </p>

```

Nosso arquivo `two-way-binding.component.ts` ficou assim:

```

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

@Component({
  selector: 'app-two-way-binding',
  templateUrl: './two-way-binding.component.html',
  styleUrls: ['./two-way-binding.component.css']
})
export class TwoWayBindingComponent implements OnInit {

  nome: string = "Thiago";

  constructor() { }

  ngOnInit() {
  }

  mudar(valor) {

```

```
    this.nome = valor + '-';  
  }  
}
```

4.4 NGIF

Agora falaremos sobre as diretivas estruturais. Vamos começar com uma condicional bem simples, o `*ngIf`. Ele tem o mesmo conceito que o `if/ else` das linguagens de programação tradicionais. Dependendo do resultado da avaliação da expressão condicional, o `*ngIf` vai mostrar ou esconder parte do template.

Os primeiros passos são: criar um novo componente com o nome de `ng-if`, adicionar a tag no arquivo `app.component.html` e comentar a tag anterior. Vamos entrar no arquivo `ng.if.componente.html` e colocar o título para esse componente.

```
<h2>*ngIf</h2>
```

Dentro no novo componente, vamos adicionar duas `div`: uma com frase de `*ngIf verdadeiro` e outra com a frase de `*ngIf falso`.

```
<h2>*ngIf</h2>
```

```
<div>  
  *ngIf verdadeiro  
</div>
```

```
<div>  
  *ngIf falso  
</div>
```

Voltando no navegador, veremos as duas frases sendo mostradas. Agora colocamos a condicional do `*ngIf` em cada

uma das `div` , porém com os resultados diferentes para cada uma delas, e verificamos o que vai acontecer no navegador.

```
<h2>*ngIf</h2>

<div *ngIf="true">
  *ngIf verdadeiro
</div>

<div *ngIf="false">
  *ngIf falso
</div>
```

O navegador só vai mostrar uma frase, exatamente a que está com a condição em `true` . Ou seja, isso mostra que o `*ngIf` do Angular 2 dentro do template segue o mesmo conceito das linguagens de programação tradicionais.

Quando o resultado de uma expressão for verdadeiro, o bloco de código será mostrado, mas quando o resultado da expressão for falsa, o bloco será retirado. Mas como fica a condicional `else` ?

No Angular 2, não temos a condicional `else` , somente vamos usar a condicional `if` . Caso você queira fazer uma expressão com `if/ else` , podemos adicionar dois `ifs` , mas com a expressão condicional invertida igual à do exemplo anterior.

Podemos passar valores de validação para o `*ngIf` de diferentes formas:

- Com métodos;
- Expressões condicionais;
- Valor de variável;
- Valor fixo na tela.

Todos esses tipos são reconhecidos pelo `ngIf` e a escolha de

qual usar vai muito de gosto pessoal, ou em alguns casos dependerá da regra de negócio aplicada. Aqui vamos exemplificar cada um passo a passo, começando com métodos. Para isso, simularemos o cadastro de cursos disponíveis em uma escola de treinamentos.

Primeiro, vamos adicionar no template um botão e um parágrafo:

```
<button (click)="mostrar()">Exibir nome da escola de treinamentos
</button>
<br>
<p *ngIf="getValor()"> Treinamentos com a condicional *ngIf</p>
```

No arquivo `ng-if.component.ts`, adicionaremos uma variável com o nome de `mostraNome` e dois métodos: um com o nome de `getValor()`, que vai retornar um *boolean*; e outro chamado `mostrar()`, que vai mudar o valor da variável. Veja como ficará nosso arquivo.

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

@Component({
  selector: 'app-ng-if',
  templateUrl: './ng-if.component.html',
  styleUrls: ['./ng-if.component.css']
})
export class NgIfComponent implements OnInit {

  mostraNome: boolean;

  constructor() { }

  ngOnInit() {
  }

  mostrar(): void {
    this.mostraNome = !this.mostraNome;
  }
}
```



```

getValor(): boolean {
    return this.mostraNome;
}

```

Voltando ao navegador e clicando no botão, o parágrafo aparece e desaparece conforme retorno do método `getValor()` .

Outra forma bem usada no `*ngif` é com expressão condicionais. No arquivo do `ng-if.component.html` , vamos adicionar dois parágrafos: um com a frase "**Não temos curso disponível**"; e outro com a frase "**Agora temos curso cadastrado**". No final, criamos um botão com o nome `add novo curso` . Veja como ficou o template.

```

<h2>*ngIf</h2>

<div *ngIf="true">
    *ngIf verdadeiro
</div>

<div *ngIf="false">
    *ngIf falso
</div>

<button (click)="mostrar()">Exibir nome da escola de treinamentos
</button>
<br>
<p *ngIf="getValor()"> Treinamentos com a condicional *ngIf</p>

<p *ngIf="cursos.length == 0">Não temos curso disponível</p>
<p *ngIf="cursos.length > 0">Agora temos curso cadastrado</p>
<button (click)="addCurso()">add novo curso</button>

```

No nosso arquivo `ng-if.component.ts` , criaremos um array com o nome de `cursos` e um método chamado `addCurso` , que vai adicionar um curso dentro do array.

```

export class NgIfComponent implements OnInit {

```

```

    mostraNome: boolean;
    cursos: string [] = [];

    /* mesmo código já existente */

    addCurso() {
        this.cursos.push('Angular 2');
    }

```

Salvando os arquivos e voltando ao navegador, somente a frase **"Não temos curso disponível"** será mostrada, pois, neste momento, o array de cursos está vazio e validando como `true` a expressão condicional de `*ngIf="cursos.length == 0"`, que está dentro da tag `p`.

Ao clicar no botão `add novo curso`, vamos adicionar o valor `Angular 2` dentro do array `cursos`, e ele passará a ter conteúdo e não será mais vazio. Voltando ao navegador, a frase *"Agora temos curso cadastrado"* será mostrada, já que, neste momento, o array contém o valor `Angular 2`. Consequentemente, o tamanho do array é maior que zero e a expressão condicional de `*ngIf="cursos.length > 0"`, que está na outra tag `p`, se torna `true`.

Os tipos de *expressão condicional e chamada de método* são os mais usados com o `*ngIf`, mas podemos usá-los também com valor de variável ou valor fixo na tela. Para exemplificar a chamada de variável dentro do `*ngIf`, vamos adicionar uma tag `p` e reutilizar a variável `mostraNome`.

```
<p *ngIf="mostraNome">com uma variável</p>
```

Neste parágrafo, o `ngIf` está sendo validado com a variável `mostraNome` e, dependendo do seu conteúdo sendo `true` ou `false`, o parágrafo será mostrado. Para verificar o seu

funcionamento com uma variável, vamos clicar no botão *Exibir nome da escola de treinamentos*. O click vai executar o método `mostrar()` e ele vai mudar o conteúdo da variável `mostraNome`.

A última forma de utilizar o `ngIf` é com valor fixo na tela, e já fizemos esse tipo de uso logo no início desse tópico, quando foram mostradas as duas `div`.

```
<div *ngIf="true">
  *ngIf verdadeiro
</div>

<div *ngIf="false">
  *ngIf falso
</div>
```

Usando o `ngIf` no Angular 2, não estamos mostrando ou escondendo a mensagem, estamos adicionando ou removendo a nossa tag dentro do DOM. Isso pode ser bom pensando na segurança da aplicação, ou ruim pensando no desempenho se for usado em grande quantidade.

Podemos usar o `hidden` para mostrar ou esconder os parágrafos da tela e, no final, teremos o mesmo resultado visual. Coloque uma tag `p` no nosso arquivo `ng-if.component.html` e, no lugar do `*ngIf`, coloque `hidden`.

```
<p [hidden]="getValor()">parágrafo com hidden</p>
```

Voltando ao navegador, veremos o mesmo resultado. Mas e agora, qual usar? Isso vai depender muito da sua necessidade e do valor a ser mostrado.

As principais diferenças são o desempenho e segurança. O `hidden` não retira as tags da tela; ele simplesmente esconde a visualização. Já o `ngIf` faz o inverso, ele retira a tag da tela.

Um segundo ponto com isso é a segurança, como o `hidden` só esconde a tag, se você inspecionar a tela apertando o botão F12 do teclado e retirar a propriedade `hidden` da tag, ela voltará a ser exibida no navegador. Isso deixa uma segurança muito baixa no sistema, podendo mostrar informações que eram para ser escondidas.

Outro ponto é o desempenho. O `hidden` é mais rápido para processar, pois somente adiciona uma propriedade dentro da tag, enquanto o `ngIf` tem de retirar a tag do DOM.

Dependendo do caso, o `hidden` pode baixar o desempenho de processamento do projeto, pois mesmo não mostrando a tag, o sistema continua a processá-la, adicionando ou retirando informações dela, e assim gerando um processamento desnecessário. E como o `ngIf` retira a tag do DOM, ela não será atualizada, diminuindo o processamento do navegador.

Para lhe ajudar na decisão, veja a seguinte definição:

Se for uma tag ou um conjunto pequeno de tags que não tenham requisitos de segurança, podemos usar o `hidden`.

Se tivermos uma árvore muito grande, por exemplo, uma `div` com várias tags dentro (que constantemente podem ser atualizadas e com informações sigilosas), vamos usar `ngIf`. Assim, caso a árvore não seja usada, ela não necessitará de atualização e também será retirada do navegador.

hidden	ngIf
Só esconde a tag no navegador	Retira a tag do navegador

Mesmo escondida, a tag é processada	Não processa a tag quando retirada
Falta de segurança	Tem mais segurança
Rapidez, pois só adiciona uma propriedade	Um pouco mais lenta, pois tem de retirar a tag do DOM

A escolha fica para cada situação. Sinta-se à vontade para escolher. O importante é desenvolver com qualidade.

Nosso arquivo final do `ng-if.component.html` :

```
<h2>*ngIf</h2>

<div *ngIf="true">
  *ngIf verdadeiro
</div>

<div *ngIf="false">
  *ngIf falso
</div>

<button (click)="mostrar()">Exibir nome da escola de treinamentos
</button>
<br>
<p *ngIf="getValor()"> Treinamentos com a condicional *ngIf</p>

<p *ngIf="cursos.length == 0">Não temos curso disponível</p>
<p *ngIf="cursos.length > 0">Agora temos curso cadastrado</p>
<button (click)="addCurso()">add novo curso</button>

<p *ngIf="mostraNome">com uma variável</p>

<p [hidden]="getValor()">parágrafo com hidden</p>
```

Nosso arquivo final do `ng-if.component.ts` :

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

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

```

}))
export class NgIfComponent implements OnInit {

  mostraNome: boolean;
  cursos: string [] = [];

  constructor() { }

  ngOnInit() {
  }

  mostrar(): void {
    this.mostraNome = !this.mostraNome;
  }

  getValor(): boolean {
    return this.mostraNome;
  }

  addCurso() {
    this.cursos.push('Angular 2');
  }
}

```

4.5 NGSWITCH

Com o uso do `*ngIf`, fazemos validações dentro do template da mesma forma que podemos fazer em linguagens de programação tradicional. Mas quando temos uma validação com várias possibilidades de resultados, vamos ter de colocar vários `*ngIf` dentro do template? Não! Isso porque poluirá toda a estrutura de visualização do template.

Da mesma forma que temos o `ngIf`, também temos o `ngSwitch`, e o seu uso vai diminuir a quantidade de validações e melhorar a estrutura do template. Vamos avaliar quando será melhor um ou outro dentro do projeto. Eu sugiro que:

Quando for uma validação simples, com duas ou três condições, podemos usar `ngIf` .

Quando passar de quatro condições, vamos usar o `ngSwitch` .

O `ngSwitch` do Angular 2 segue o mesmo padrão das linguagens tradicionais, sendo que no começo temos uma *expressão condicional* e, dependendo desse resultado, vamos executar um `ngSwitchCase` correspondente ao seu valor.

A estrutura do `ngSwitch` no Angular 2 é desta forma:

```
<div [ngSwitch]="expressão condicional">
  <p *ngSwitchCase="condição 1">mostra conteúdo 1</p>
  <p *ngSwitchCase="condição 2">mostra conteúdo 2</p>
  <p *ngSwitchCase="condição 3">mostra conteúdo 3</p>
  <p *ngSwitchCase="condição 4">mostra conteúdo 4</p>

  <p *ngSwitchDefault>mostra conteúdo padrão</p>
</div>
```

Temos aqui duas coisas muito importantes para observar:

O `[ngSwitch]` **deve** ser colocado entre `[]` , já que `ngSwitch` é um *property binding*.

O `*ngSwitchCase` **deve** ser iniciado com `*` (asterisco), pois ele é uma condicional.

Sempre devemos seguir essas duas regras para tudo funcionar corretamente. Uma outra informação importante é o uso do `*ngSwitchDefault` : ele é opcional e serve para mostrar um conteúdo padrão caso a condição do `ngSwitch` não esteja validada em nenhum `ngSwitchCase` .

Criaremos um novo componente para exemplo deste

conteúdo. Vamos colocar o nome de `ng-switch-case` e, após a criação do componente, seguiremos no nosso padrão de colocar o título, adicionar a nova tag no arquivo `app.component.html` e comentar a tag anterior. Após feito os procedimentos padrões, vamos ao navegador para verificar se o projeto está funcionando. Vamos praticar!

Faremos um exemplo bem simples que será um contador e, a cada número, será mostrado um parágrafo no navegador. No HTML, teremos as condicionais de `[ngSwitch]` , `*ngSwitchCase` , `*ngSwitchDefault` e um botão para incrementar a numeração.

No arquivo `ng-switch-case.component.html` , vamos adicionar esse conteúdo.

```
<div [ngSwitch]="numero">
  <p *ngSwitchCase="1">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="2">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="3">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="4">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="5">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchDefault>Mensagem padrão</p>
</div>

<button (click)="incrementarNumero()">Incrementar número</button>
```

No arquivo `ng-switch-case.component.ts` , criaremos uma variável com o nome `numero` e um método que vai incrementar o valor da variável `numero` a cada clique no botão.

```
numero: number = 0;
```



```
incrementarNumero() {  
    this.numero ++;  
}
```

Após salvar todos os arquivos, vamos ao navegador. Veja que a frase inicial é a mensagem padrão, pois neste momento o conteúdo da variável `numero` não está executando nenhum `ngSwitchCase` dentro do template.

Quando clicamos no botão *Incrementar número*, o conteúdo no navegador será mudado conforme o conteúdo da variável `numero`. Quando passamos do valor que está dentro das condicionais `ngSwitchCase`, será mostrado na tela a mensagem padrão do `ngSwitchDefault`.

Veja que, junto com as mensagens, adicionamos uma interpolação que vai juntar o conteúdo da variável `numero` com a frase que está no template.

A sequência que será mostrada no navegador será assim:

```
Mensagem padrão;  
A variável está com o valor de 1;  
A variável está com o valor de 2;  
A variável está com o valor de 3;  
A variável está com o valor de 4;  
A variável está com o valor de 5;  
Mensagem padrão.
```

E como será esse mesmo código somente colocando `ngIf`? Esse pode ser um ótimo exercício para você ver a diferença entre `ngIf` e `ngSwitch`.

Veja como construir o template com `ngIf` e `ngSwitch`, e

qual código ficará melhor estruturado. Nosso arquivo `ng-switch-case.component.html` ficou assim:

```
<h2>[ngSwitch] / *ngSwitchCase</h2>

<div [ngSwitch]="numero">
  <p *ngSwitchCase="1">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="2">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="3">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="4">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchCase="5">a variável está com o valor de {{numero}}
</p>
  <p *ngSwitchDefault>Mensagem padrão</p>
</div>

<button (click)="incrementarNumero()">Incrementar número</button>
```

Nosso arquivo `ng-switch-case.component.ts` ficou assim:

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

@Component({
  selector: 'app-ng-switch-case',
  templateUrl: './ng-switch-case.component.html',
  styleUrls: ['./ng-switch-case.component.css']
})
export class NgSwitchCaseComponent implements OnInit {

  numero: number = 0;

  constructor() { }

  ngOnInit() {
  }

  incrementarNumero() {
    this.numero ++;
  }
}
```

4.6 NGFOR

Já demos uma pitadinha de conhecimento no `ngFor` em alguns exemplos passados, agora vamos estudá-lo mais a fundo. O `ngFor` vai servir para interações de lista, e segue o mesmo padrão do laço de repetição `for` das linguagens de programação tradicionais. Entretanto, sua escrita é feita de forma mais simplificada, ficando muito parecido com a instrução `forEach` de algumas linguagens.

A forma mais simples de usar o `ngFor` dentro do template será assim:

```
<li *ngFor="let variável_local of array ">
```

Declaração	Significado
<i>let</i>	Declaração da variável
<i>variável_local</i>	Nome da variável para usar no template
<i>of</i>	Junção do array com a variável
<i>array</i>	Nome do array que está na classe do componente

Caso necessário, podemos usar o `index` de cada elemento colocando mais uma declaração no `ngFor`. Isso pode ser útil para numeração de uma lista, por exemplo.

```
<li *ngFor="let variável_local of array, let i = index">
```

Declaração	Significado
<i>let</i>	Declaração da variável
<i>i</i>	Nome da variável
<i>index</i>	Valor no índice de cada elemento do array

Vamos para os exemplos. Criaremos um novo componente que terá o nome de `ng-for` e, em seguida, faremos nossos procedimentos padrões de adicionar um título dentro do template com o nome de `ngFor`, inserir a tag do novo componente dentro do `app.component.html` e comentar a tag do exemplo anterior.

O título desse componente será `ngFor` e, dentro do arquivo `ng-for.component.ts`, vamos criar um array com nome de `nomes`:

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

@Component({
  selector: 'app-ng-for',
  templateUrl: './ng-for.component.html',
  styleUrls: ['./ng-for.component.css']
})
export class NgForComponent implements OnInit {

  nomes: string [] = ['João', 'Maria', 'Thiago', 'José'];

  constructor() { }

  ngOnInit() {
  }
}
```

Dentro do arquivo `ng-for.component.html`, criaremos uma lista não ordenada com a tag `ul`. Dentro da lista, colocamos a tag `li` para incluir conteúdo nela e, dentro da tag `ul`, vamos incluir a marcação `ngFor` do Angular 2.

```
<ul *ngFor="let nome of nomes">
  <li>nome da pessoa é: {{nome}} </li>
</ul>
```

Salvando e voltando ao navegador, veremos uma lista com os nomes que estão no array `nomes`, que está dentro da classe do

componente.

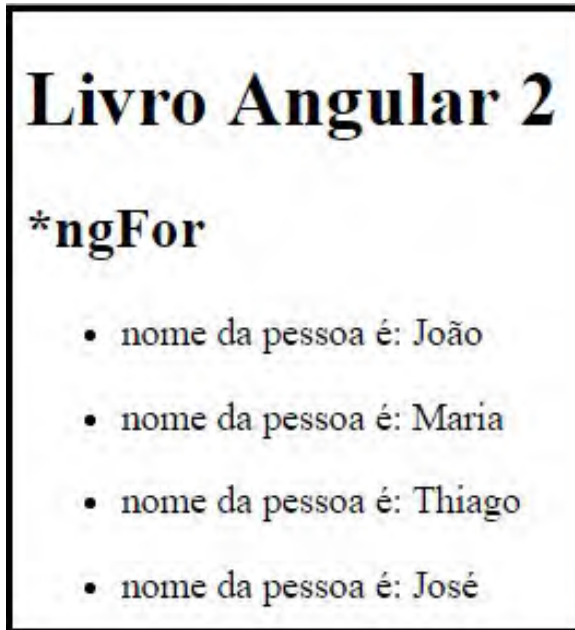


Figura 4.2: Lista com *ngFor

Usar o `*ngFor` é bem simples, como acabamos de ver. Agora, e se precisarmos saber o índice de cada elemento dentro do array? Para isso, vamos usar mais uma declaração dentro do `*ngFor`.

```
<ul *ngFor="let nome of nomes, let i= index">  
  <li>nome da pessoa é: {{nome}}, está na posição {{i + 1}} </li>  
</ul>
```

Colocamos no final da frase uma interpolação com o `i` do `index`, somando mais 1 para que, no navegador, a posição do nome seja mostrada corretamente, pois todo array em programação vai começar em zero.

Com o código `let i= index`, pegamos a posição de cada elemento dentro do array junto com a marcação do `*ngFor`. Veja também que os conceitos do Angular 2 estudados anteriormente são usados juntos com novos exemplos; podemos usar tudo que for possível para ajudar na codificação do projeto.

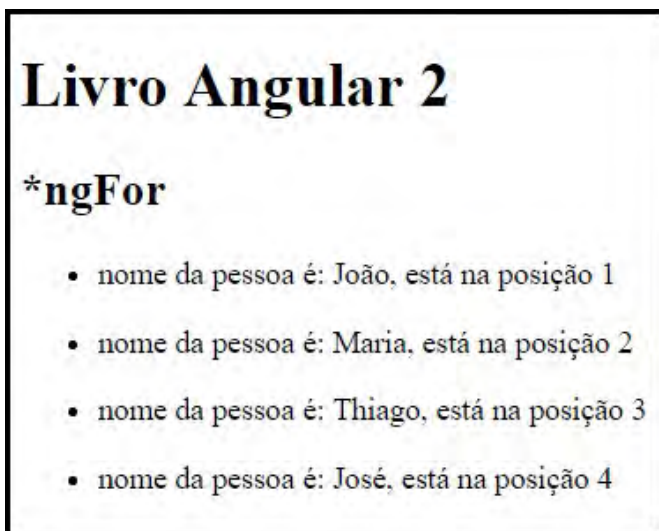


Figura 4.3: Lista `*ngFor` com index

Um detalhe importante para falar é que tanto o `let nome` como o `let i= index` são variáveis que só podem ser usadas dentro do bloco de declaração do `*ngFor`. Se usadas fora do bloco, o Angular 2 vai simplesmente ignorá-las e você não terá o resultado esperado.

4.7 REAPROVEITAMENTO DE LISTA DENTRO DO ANGULAR 2

Quase sempre receberemos as listas de serviços, e muitas vezes as listas serão praticamente iguais as que já estamos mostrando no navegador. Um ponto importante aqui é que, a cada alteração na lista que está sendo mostrada no navegador, seja de inclusão, alteração ou exclusão, o Angular 2 **refaz toda a lista no navegador**.

Isso pode ser um problema se tivermos uma com vários itens. A interação com elementos do DOM será muito grande e comprometerá o desempenho da aplicação. Pensando nisso, temos um conceito de reaproveitamento de lista dentro do Angular 2.

Quando atualizamos uma lista, provavelmente a maioria dos elementos continuarão ali e não precisaríamos refazer os já existentes. Mas, por padrão, o Angular 2 refaz toda a lista. Como podemos reaproveitar os elementos que já existem nela?

Podemos adicionar junto com a marcação `ngFor` uma função que diz: **elementos com os mesmos dados são iguais**. Para isso, teremos de incluir uma informação que não pode se repetir. Então, dentro do array `nomes`, vamos colocar um `id` para cada elemento.

Voltamos ao arquivo `ng-for.component.ts` e modificamos nosso array para que cada elemento tenha um `id`. Teremos de trocar o tipo do array que está como `string` e colocar como `any`, pois agora o array de nomes é um objeto complexo e não somente de `string`.

```
nomes: any [] = [  
  {id:1, nome: 'João'},  
  {id:2, nome: 'Maria'},  
  {id:3, nome: 'Thiago'},  
  {id:4, nome: 'José'}  
]
```

```
];
```

No nosso arquivo `ng-for.component.html`, vamos modificar dentro das interpolações e colocar a propriedade de cada objeto que está dentro do array.

```
<ul *ngFor="let nome of nomes, let i= index">
  <li>nome da pessoa é: {{nome.nome}} com id {{nome.id}} , está
  na posição {{i + 1}} </li>
</ul>
```

Salvando e voltando ao navegador, veremos o `nome`, `id` e `posição` de cada elemento como estava anteriormente.

Voltamos na classe do componente e adicionamos uma função que verifica o ID de cada elemento da nova lista e compara com o ID da lista atual. Se os IDs forem iguais, o elemento que está no navegador não será atualizado.

```
meuSave(index: number, nomes: any) {
  return nomes.id;
}
```

Um detalhe importante para falar é que essa função somente verifica os elementos na mesma posição, ou seja: posição 1 da lista que está no navegador com a posição 1 da lista que veio do serviço, posição 2 da lista que está no navegador com a posição 2 da lista que veio do serviço, e assim por diante. Se a nova lista vir com os elementos nas posições trocadas, o Angular 2 não terá como saber se tem algo novo, e assim refará toda a lista como se fosse uma nova.

Dentro do `*ngFor` que está no template, vamos referenciar nosso método que vai reutilizar nossa lista atual do navegador. Para fazer isso, vamos usar o `trackBy`.


```
<ul *ngFor="let nome of nomes, let i = index; trackBy:meuSave">
```

O `trackBy` diz para o `*ngFor` usar o método `meuSave` para validar o conteúdo da nova lista, e somente renderizar conteúdos que ainda não estão no navegador. Para testar essa função dentro do `*ngFor`, vamos adicionar no template um botão e, no evento de clique, executar o método `atualizar()`, que carregará uma nova lista para a variável `nomes`.

```
<button (click)="atualizar()">Atualizar</button>
<ul *ngFor="let nome of nomes, let i = index; trackBy:meuSave">
  <li>nome da pessoa é: {{nome.nome}} com id {{nome.id}}, está
na posição {{i + 1}} </li>
</ul>
```

Na classe do componente, vamos criar o método `atualizar()`, que vai atribuir uma nova lista para a variável `nomes`.

```
atualizar() {
  this.nomes = [
    {id:1, nome:'João'},
    {id:2, nome:'Maria'},
    {id:3, nome:'Thiago'},
    {id:4, nome:'José'},
    {id:5, nome:'tatat'}
  ];
}
```

Para acompanhar as mudanças, apertaremos o botão `F12` do teclado. Após isso, vamos na aba `elementos`, e lá será mostrada toda a estrutura da página com as tags do HTML. Vamos até a tag `app-ng-for` e, dentro dela, estará nossa lista.

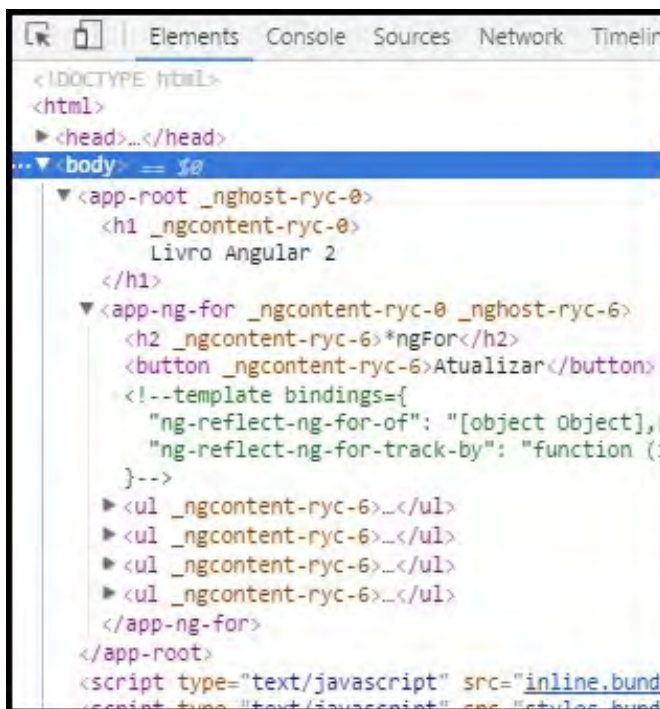


Figura 4.4: Estrutura da página HTML

Clicando no botão `Atualizar` e olhando a estrutura da lista no HTML, veremos que somente uma linha foi alterada. Agora retire o `trackBy` e a chamada da função dentro do `*ngFor`, salve o arquivo, volte ao navegador e clique novamente no botão `Atualizar()`. E agora o que aconteceu? Toda a lista foi refeita.

Esse método junto com o `trackBy`, dentro da marcação do `ngFor`, podemos ganhar mais velocidade na renderização de grandes listas e diminuir a quantidade de processamento para o projeto. Resumindo:

Elementos com o mesmo ID e na mesma posição nos

dois arrays: o Angular 2 mantém o elemento que está no navegador.

Elementos com ID diferente ou com posição diferente dentro do array: o Angular 2 refaz o elemento e coloca os dados do novo array.

Nosso arquivo `ng-for.component.html` ficou assim:

```
<h2>*ngFor</h2>

<button (click)="atualizar()">Atualizar</button>
<ul *ngFor="let nome of nomes, let i = index; trackBy:meuSave">
  <li>nome da pessoa é: {{nome.nome}} com id {{nome.id}}, está
na posição {{i + 1}} </li>
</ul>
```

Nosso arquivo `ng-for.component.ts` ficou assim:

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

@Component({
  selector: 'app-ng-for',
  templateUrl: './ng-for.component.html',
  styleUrls: ['./ng-for.component.css']
})
export class NgForComponent implements OnInit {

  nomes: any [] = [
    {id:1, nome:'João'},
    {id:2, nome:'Maria'},
    {id:3, nome:'Thiago'},
    {id:4, nome:'José'}
  ];

  constructor() { }

  ngOnInit() {
  }

  meuSave(index: number, nomes: any) {
    return nomes.id;
  }
}
```

```

atualizar() {
  this.nomes = [
    {id:1, nome:'João'},
    {id:2, nome:'Maria'},
    {id:3, nome:'Thiago'},
    {id:4, nome:'José'},
    {id:5, nome:'tatat'}
  ];
}
}

```

4.8 NGCLASS

O uso do `ngClass` será para manipulação de classes do CSS dentro do nosso projeto. Essa será nossa primeira diretiva de atributo que vamos estudar. Ela vai servir para adicionar e remover configurações de CSS para cada elemento HTML.

Para começar os exemplos, vamos criar o novo componente com o nome de `ng-class`, e em seguida faremos os procedimentos já conhecidos de adição da nova tag, comentário da anterior e inclusão do título que terá o nome de `[ngClass]`.

Com o `ngClass`, podemos adicionar ou remover desde uma simples classe CSS até um conjunto de classes dentro de um elemento do HTML. Podemos manipular classes CSS no elemento HTML de três formas diferentes, sendo:

Usando o class binding no template.

Declarando a classe do CSS diretamente no `ngClass`.

Declarando um método da classe do componente dentro do `ngClass`.

Aqui veremos os três tipos, suas diferenças e quando será mais

apropriado a utilização de cada um deles. Vamos começar pelo mais simples, que é o **class binding**. Nele podemos configurar uma classe CSS de cada vez dentro do template. Seu uso e configuração são bem parecidos com property binding.

Vamos ao arquivo `ng-class.component.css`, e criaremos uma classe CSS com o nome de `cor-fundo` para usar no exercício. Nossa classe fará um background na cor cinza claro em cada tag do HTML que a usar.

```
.cor-fundo {  
  background-color: lightgray;  
}
```

Dentro do arquivo `ng-class.component.html`, vamos adicionar um tag `p` com o conteúdo de usando class binding, `[class.cor-fundo]="true"` e, dentro da tag, vamos usar a marcação do *class binding*.

```
<h2>[ngClass]</h2>  
  
<p [class.cor-fundo]="true">usando class binding, [class.cor-fund  
o]="true"</p>
```

Três pontos importante para configuração estão nesse *class binding*.

Declaração	Significado
<i>class</i>	Obrigatório na configuração
<i>cor-fundo</i>	Nome da classe no CSS
<i>true</i>	Condição verdadeira ou falsa na configuração

No class binding, usamos a configuração padrão `class` seguindo do nome da classe no CSS e a condição de verdadeiro ou

falsa.

Se verdadeiro, a classe CSS será aplicada no HTML.

Se falso, a classe não será aplicada no HTML.

Mude a condição do `class binding` de `true` para `false`, salve e veja o resultado no navegador. A tag HTML não ficará com o fundo cinza claro.

Assim como podemos colocar valor fixo na tela, também podemos passar variáveis que estão na classe do componente como uma referência. Vamos ao arquivo `ng-class.component.ts` e criamos uma variável com o nome de `valorClassBinding`. Ela será do tipo *boolean* e vai ser responsável por adicionar e remover a classe CSS dentro da tag do HTML. Em seguida, criaremos um método com o nome de `mudarClassBinding()` que vai alterar o valor da variável recém-criada.

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

@Component({
  selector: 'app-ng-class',
  templateUrl: './ng-class.component.html',
  styleUrls: ['./ng-class.component.css']
})
export class NgClassComponent implements OnInit {

  valorClassBinding: boolean = false;

  constructor() { }

  ngOnInit() {
  }

  mudarClassBinding() {
    this.valorClassBinding = ! this.valorClassBinding;
  }
}
```

Vamos adicionar outra tag `p` e, dentro do arquivo `ng-class.component.html`, no lugar de passar `true` para o `ClassBinding`, vamos referenciar a variável `valorClassBinding`. Logo após, adicionamos um botão que terá um evento de clique que vai executar o método `mudarClassBinding` da classe do componente.

```
<h2>[ngClass]</h2>

<p [class.cor-fundo]="true">usando class binding, [class.cor-fundo]="true"</p>

<p [class.cor-fundo]="valorClassBinding">usando class binding, [class.cor-fundo]="valorClassBinding"</p>
<button (click)="mudarClassBinding()">Adicionar CSS</button>
```

Salvando os arquivos e voltando ao navegador, ao clicar no botão *Adicionar CSS*, o fundo da frase `usando class binding, [class.cor-fundo]="valorClassBinding"` ficará mudando de cor.

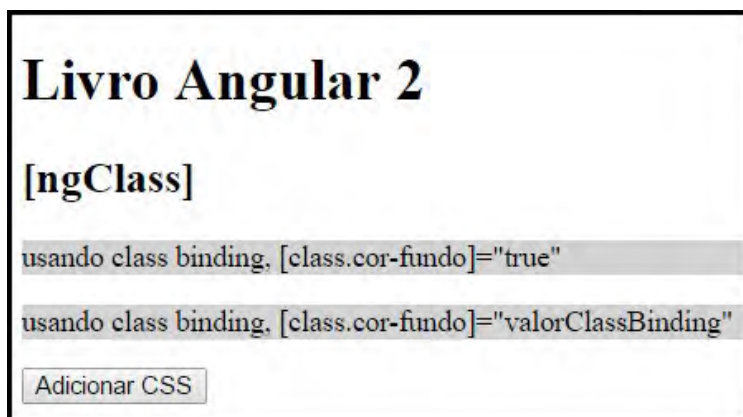


Figura 4.5: Usando class binding para configuração do CSS HTML

Podemos usar o *class binding* para configuração de poucas

classes dentro da tag do HTML, de preferência quando for somente uma classe. Acima disso, nosso arquivo de template ficará muito poluído com essas configurações.

Para configuração de duas ou mais classes em uma tag HTML, vamos usar o `ngClass`, no qual podemos tanto declarar a classe CSS ou referenciar um método da classe do componente.

Usando o `ngClass` dentro da tag HTML, devemos passar sempre duas informações que obrigatoriamente estarão dentro de `{}`. Elas são: **nome da classe** e **valor booleano para exibição**.

Vamos incluir um novo parágrafo com a tag `p` no arquivo do template e, dentro da tag, usaremos a diretiva `ngClass`, atribuindo `true` para a nossa classe do CSS.

```
<p [ngClass]='{'cor-fundo': true}'>usando ngClass declarando a classe CSS no template</p>
```

Salvando e voltando ao navegador, veremos o parágrafo com o fundo cinza.

Declarando a classe CSS com `ngClass` dentro do template, ou fazendo um *class binding* dentro do template, ambos praticamente seguem o mesmo padrão. A diferença será em uma sequência de várias declarações CSS dentro de uma tag HTML porque, com o *class binding*, devemos declarar uma classe CSS por vez, enquanto que, com `ngClass`, podemos criar um array com todas as classes e os valores para exibição.

Para verificar essa diferença, vamos criar mais três classes CSS: uma que vai mudar a cor da letra, uma que mudará o estilo da letra e outra que colocará uma borda no parágrafo.


```
.cor-fundo {
    background-color: lightgray;
}

.cor-letra {
    color: red;
}

.estilo-letra {
    font-style: italic;
}

.borda-paragrafo {
    border: 2px green solid;
}
```

Se precisarmos adicionar nossas quatro classes existentes em uma tag do HTML fazendo o *class binding*, teremos o seguinte código:

```
<p [class.cor-fundo]="valorClassBinding"
    [class.cor-letra]="valorClassBinding"
    [class.estilo-letra]="valorClassBinding"
    [class.borda-paragrafo]="valorClassBinding">usando class bind
ing com várias classes CSS</p>
```

Veja que, para cada classe CSS, foi feito um *class binding*, assim deixamos nosso HTML muito poluído e de difícil entendimento. Se fizermos o mesmo código, mas agora usando o `ngClass`, melhoraremos um pouco o entendimento do nosso HTML. Veja o código a seguir:

```
<p [ngClass]="{ 'cor-fundo': valorClassBinding,
'cor-letra': valorClassBinding,
'estilo-letra': valorClassBinding,
'borda-paragrafo': valorClassBinding}">usando ng Class com vári
as classes CSS</p>
```

Salvando e voltando ao navegador, teremos esse resultado:

Livro Angular 2

[ngClass]

usando class binding, [class.cor-fundo]="true"

usando class binding, [class.cor-fundo]="valorClassBinding"

Adicionar CSS

usando ngClass declarando a classe CSS no template

usando class binding com várias classes CSS

usando ngClass com várias classes CSS

Figura 4.6: Atribuindo diversas classes com ngClass

Veja que, com a declaração do `ngClass` dentro do template, tivemos uma diminuição de código que não faz parte do HTML, mas mesmo assim nosso template está muito poluído. Podemos melhorar com o uso de métodos junto com o `ngClass`.

Vamos criar um novo método no arquivo `ng-class.component.ts` com o nome de `classes()` e, dentro dele, vamos configurar todas as classes CSS para usar na tag do HTML.

```
classes(): any {  
  let valores = {  
    'cor-fundo': this.valorClassBinding,  
    'cor-letra': this.valorClassBinding,  
    'estilo-letra': this.valorClassBinding,  
    'borda-paragrafo': this.valorClassBinding  
  }  
  
  return valores;  
}
```

Agora no nosso template, criaremos uma nova tag `p` com o conteúdo usando `ngClass` com método na classe do componente e, dentro dela, colocamos a diretiva `ngClass` referenciando nosso método `classes()`, que está na classe do componente.

```
<p [ngClass]="classes()">usando ngClass com método na classe do c  
omponente</p>
```

Salvando e voltando ao navegador, veremos que o resultado será o mesmo dos anteriores. Porém, com o `ngClass` referenciando o método que está na classe do componente, o código do nosso template fica muito mais limpo e organizado.

Nosso arquivo `ng-class.component.css` ficou assim:

```
.cor-fundo {  
    background-color: lightgray;  
}  
  
.cor-letra {  
    color: red;  
}  
  
.estilo-letra {  
    font-style: italic;  
}  
  
.borda-paragrafo {  
    border: 2px green solid;  
}
```

Nosso arquivo `ng-class.component.html` ficou assim:

```
<h2>[ngClass]</h2>  
  
<p [class.cor-fundo]="true">usando class binding, [class.cor-fund  
o]="true"</p>  
  
<p [class.cor-fundo]="valorClassBinding">usando class binding, [c
```

```

class.cor-fundo]="valorClassBinding"</p>
<button (click)="mudarClassBinding()">Adicionar CSS</button>

<p [ngClass]="{'cor-fundo': true}">usando ngClass declarando a classe CSS no template</p>

<p [class.cor-fundo]="valorClassBinding" [class.cor-letra]="valorClassBinding" [class.estilo-letra]="valorClassBinding" [class.borda-paragrafo]="valorClassBinding">usando class binding com várias classes CSS</p>

<p [ngClass]="{
  'cor-fundo': valorClassBinding,
  'cor-letra': valorClassBinding,
  'estilo-letra': valorClassBinding,
  'borda-paragrafo': valorClassBinding
}">usando ngClass com várias classes CSS</p>

<p [ngClass]="classes()">usando ngClass com método na classe do componente</p>

```

Nosso arquivo `ng-class.component.ts` ficou assim:

```

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

@Component({
  selector: 'app-ng-class',
  templateUrl: './ng-class.component.html',
  styleUrls: ['./ng-class.component.css']
})
export class NgClassComponent implements OnInit {

  valorClassBinding: boolean = false;

  constructor() { }

  ngOnInit() {
  }

  mudarClassBinding(): void {
    this.valorClassBinding = ! this.valorClassBinding;
  }
}

```

```

classes(): any {
  let valores = {
    'cor-fundo': this.valorClassBinding,
    'cor-letra': this.valorClassBinding,
    'estilo-letra': this.valorClassBinding,
    'borda-paragrafo': this.valorClassBinding
  }

  return valores;
}
}

```

4.9 NGSTYLE

O `ngStyle` é muito parecido com `*ngClass`. A diferença é que, com a diretiva `ngClass`, atribuímos ou removemos as classes CSS de dentro da tag do HTML, enquanto, com a diretiva `ngStyle`, adicionamos ou removemos cada atributo de estilo que uma tag HTML pode ter.

Dentro das classes CSS, podemos configurar vários atributos de uma só vez. Logo, com o uso do `ngClass`, vamos atribuir vários atributos para a mesma tag do HTML.

Usando o `ngStyle`, podemos configurar cada atributo separadamente dentro de uma tag no HTML. Isso pode ser útil para algumas mudanças pontuais no conteúdo do navegador, como por exemplo, trocar uma cor de texto, aumentar o tamanho da fonte, deixar uma frase em negrito, entre outras coisas que podem ter dentro das regras de negócio de um projeto.

Para começar a parte prática usando o `ngStyle`, criaremos nosso novo componente com o nome de `ng-style` e seguiremos nossos procedimentos já conhecidos de adicionar título e tag, e comentar a tag anterior. O título desse componente será

[ngStyle] .

Da mesma forma que temos o *class binding* para configurar uma classe CSS dentro do HTML, na configuração de estilos temos o **style binding** que vai funcionar bem parecido com o anterior.

Vamos adicionar duas tags `p` dentro do nosso template (nosso arquivo `ng-style.component.html`) e, dentro de cada tag, colocaremos o style binding `[style.xxx]` , em que o `xxx` será o nome do atributo seguindo do seu valor. No primeiro parágrafo, vamos atribuir o valor fixo para o tamanho da fonte e, no segundo, vamos referenciar uma variável que estará dentro da classe do componente (nesse caso, o arquivo `ng-style.component.ts`) com o valor da fonte.

```
<h2>[ngStyle]</h2>
```

```
<p [style.font-size]="30 + 'px'">valor fixo na tela</p>
```

```
<p [style.font-size]="valorFonte">método da classe componente sen  
do referenciado no template</p>
```

No nosso arquivo `ng-style.component.ts` , vamos criar uma variável com o nome `valorFonte` e atribuir o valor de 20 pixels.

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

```
@Component({  
  selector: 'app-ng-style',  
  templateUrl: './ng-style.component.html',  
  styleUrls: ['./ng-style.component.css']  
})  
export class NgStyleComponent implements OnInit {  
  
  valorFonte: string = 20 + 'px';  
  
  constructor() { }  
  
  ngOnInit() {
```

```
}  
}
```

Salvando e voltando ao navegador, veremos as duas frases com tamanhos diferentes, de acordo com o valor passado para cada parágrafo.

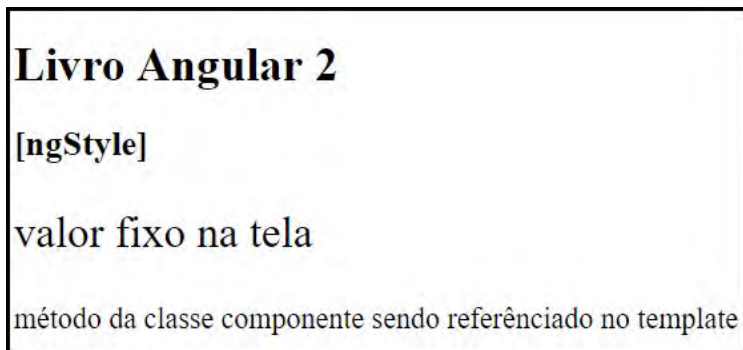


Figura 4.7: Mudando tamanho da fonte com ngStyle

Uma forma interessante de usar o `ngStyle` ou o *style binding* é atribuir de forma dinâmica os valores para cada atributo. Vamos adicionar um botão com a tag `button` no nosso template e, dentro da tag, criar o *event binding* de `click`. Ele vai executar o método `incrementar()` que estará na classe do componente.

```
<button (click)="incrementar()"></button>
```

Na nossa classe do componente, vamos criar uma variável com o nome `tamanho`, que vai concatenar com a variável `valorFonte` para receber a unidade `px`. Estamos fazendo dessa forma para mostrar que o `px` pode ser colocado tanto no template quanto na classe do componente. O importante será não esquecer de usá-lo junto com o valor.

Nosso método `incrementar()` vai aumentando o valor da

variável tamanho a cada clique no botão.

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

@Component({
  selector: 'app-ng-style',
  templateUrl: './ng-style.component.html',
  styleUrls: ['./ng-style.component.css']
})
export class NgStyleComponent implements OnInit {

  tamanho: number = 20;
  valorFonte: string = this.tamanho + 'px';

  constructor() { }

  ngOnInit() {
  }

  incrementar() {
    this.tamanho ++;
    this.valorFonte = this.tamanho + 'px';
  }
}
```

Salvando e voltando ao navegador, a cada clique no botão *Maior*, a variável tamanho vai sendo incrementada e, conseqüentemente, a fonte do parágrafo que está referenciando essa variável também vai aumentando.

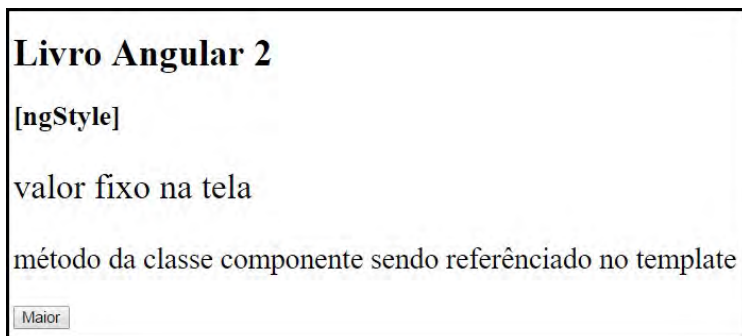


Figura 4.8: Mudando tamanho da fonte a cada clique

Uma coisa muito legal que podemos fazer no *style binding* e `ngStyle` é uma validação ternária, antes de atribuir um valor para algum atributo no HTML. Vamos incluir no nosso template uma tag `p` e dois botões com a tag `button`.

Dentro da tag `p`, teremos dois *style binding*: o primeiro para o tamanho da fonte do parágrafo, e o segundo para a cor do texto que estará dentro do parágrafo. Os botões vão servir para mudar a condição de visualização de cada *style binding* através das atribuições `true` e `false`.

```
<p [style.font-size]="validaFonte ? 'x-large' : 'smaller'"
  [style.color]="validaCor ? 'red' : 'blue'">muda valor com con
dição ternária</p>
<button (click)="mudaFonte()">mudaFonte</button>
<button (click)="mudaCor()">mudaCor</button>
```

No primeiro *style binding*, estamos verificando se a variável `validaFonte` tem o valor de `true` ou `false`. Se for `true`, será atribuído o valor de `x-large` para o `font-size`. E se for `false`, será atribuído o valor de `smaller` a ele.

No segundo *style binding*, estamos verificando se a variável `validaCor` está com o valor de `true` ou `false`. Se for `true`, será atribuído o valor de `red` para a cor do parágrafo. Mas se a variável for `false`, será atribuído o valor de `blue` para ele.

Agora veremos as mudanças na nossa classe do componente:

```
validaFonte: boolean = false;
validaCor: boolean = false;

/* mesmo código anterior*/

mudaFonte() {
  this.validaFonte = ! this.validaFonte;
```

```

}

mudaCor() {
  this.validaCor = ! this.validaCor;
}

```

Primeiro, criamos duas variáveis do tipo *boolean* com os nomes `validaFonte` e `validaCor`, iniciadas em `false`. Após isso, criamos dois métodos com os nomes de `mudaFonte()` e `mudaCor()` que, quando executados, vão mudar os valores das variáveis `validaFonte` e `validaCor`.

Salvando e voltando ao navegador, a cada clique no botão, teremos uma mudança no parágrafo: ou as letras ficarão vermelhas ou azuis, ou o tamanho da fonte ficará grande ou pequena.

Todos os exemplos feitos até agora com o *style binding* podem ser aplicados no `ngStyle`, e a escolha do uso vai seguir a mesma regra que no *class binding* e no `ngClass`. Dependendo da quantidade de atributos a ser mudada, usaremos *style binding* ou `ngstyle`.

Podemos usar o `ngStyle` tanto atribuindo os valores no próprio HTML, ou criando um método na classe do componente com os atributos e respectivos valores e referenciando esse método na nossa tag HTML do template. Não esqueça de sempre usar as `{}`, igualzinho como usamos com `ngClass`.

Para exercício, quero que você crie novas tags `p` e use o `ngStyle` no lugar do *style binding* que usamos nos exemplos. No final, o funcionamento será o mesmo, mas a quantidade de código no template tende a diminuir comparando o *style binding* com o `ngStyle` ao declarar os atributos. Isso deixará o template limpo, mudando o `ngStyle` com declaração para o `ngStyle` com

referência do método na classe do componente.

Para finalizar, vou deixar todo o código, tanto do template quanto da classe do componente, com os exemplos apresentados e com os exercícios resolvidos. Mas antes de verificar o resultado, tente fazer sozinho.

Nosso template final `ng-style.component.html` :

```
<h2>[ngStyle]</h2>

<p [style.font-size]="30 + 'px'">valor fixo na tela</p>

<p [style.font-size]="valorFonte">método da classe componente sen
do referenciado no template</p>

<button (click)="incrementar()">Maior</button>

<p [style.font-size]="validaFonte ? 'x-large' : 'smaller'" [style
.color]="validaCor ? 'red' : 'blue'">muda valor com condição tern
ária</p>
<button (click)="mudaFonte()">mudaFonte</button>
<button (click)="mudaCor()">mudaCor</button>

<h3>Exercício</h3>

<p [ngStyle]="{'font-size': '30px'}">valor fixo na tela</p>

<p [ngStyle]="{'font-size': valorFonte}">método da classe compone
nte sendo referenciado no template</p>

<button (click)="incrementar()">Maior</button>

<p [ngStyle]="{'font-size': validaFonte ? 'x-large' : 'smaller',
'color': validaCor ? 'red' : 'blue'}">muda valor co
m condição ternária</p>
<button (click)="mudaFonte()">mudaFonte</button>
<button (click)="mudaCor()">mudaCor</button>
```

Nossa classe do componente final `ng-style.component.ts` :

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

```

@Component({
  selector: 'app-ng-style',
  templateUrl: './ng-style.component.html',
  styleUrls: ['./ng-style.component.css']
})
export class NgStyleComponent implements OnInit {

  tamanho: number = 20;
  valorFonte: string = this.tamanho + 'px';

  validaFonte: boolean = false;
  validaCor: boolean = false;

  constructor() { }

  ngOnInit() {
  }

  incrementar() {
    this.tamanho ++;
    this.valorFonte = this.tamanho + 'px';
  }

  mudaFonte() {
    this.validaFonte = ! this.validaFonte;
  }

  mudaCor() {
    this.validaCor = ! this.validaCor;
  }
}

```

4.10 NGCONTENT

O último tópico apresentado será o `ngContent`. Ele é um componente genérico que vai servir para colocarmos informações dentro dele.

Com o `ngContent`, podemos criar uma estrutura padrão que será usada em diferentes situações dentro do projeto, como por

exemplo, uma tabela ou uma lista, que são elementos constantemente usados. A diferença de uma tabela para outra, ou uma lista para outra, é somente no conteúdo que será passado. Com isso, podemos padronizar o componente para ser usado em todo projeto.

Vamos criar nosso novo componente com o nome de `ng-content` e seguir todos os passos já falados de adicionar tag, comentar tag antiga e colocar título. Nosso título será `<ng-content>`. Após os procedimentos, salvamos e verificamos no navegador o funcionamento do novo componente.

Para mostrar o funcionamento do `ng-content`, vamos criar uma estrutura de tabela dentro do nosso template que, por enquanto, terá um título fixo no cabeçalho e uma linha no corpo da tabela com a tag `<ng-content>` `</ng-content>`.

```
<h2>
  < ng-content>
</h2>

<table>
  <thead>
    <tr>
      <th>
        Título da tabela
      </th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>
        <ng-content></ng-content>
      </td>
    </tr>
```

```
</tbody>  
</table>
```

Colocando essa tag dentro da estrutura do HTML, dizemos para o Angular 2 que a tag `<ng-content>` `</ng-content>` será substituída por um conteúdo que será enviado pelo componente pai deste componente. Ficou confuso? Vou explicar.

No nosso projeto, temos o componente principal, que é o `app.component`. Dentro do template dele, estamos instanciando todos outros componentes que estamos criando ao longo do livro, por meio da criação de várias tags parecidas com esta: `<app-nome-do-componente>` `</app-nome-do-componente>`.

Quando instanciamos o componente `ng-content.component` através da tag `<ng-content>` `</ng-content>`, dentro de `app.component.html`, o `app.component` se torna o pai do componente `ng-content.component`. E é o pai que fica responsável por enviar o conteúdo para substituir a tag `<ng-content>` `</ng-content>`, que está dentro da estrutura HTML.

Dessa forma, para passarmos informações para dentro do componente `ng-content.component`, teremos de colocar essa informação na tag que está no `app.component.html`, dessa forma:

```
<app-ng-content>meu conteúdo</app-ng-content>
```

Quando o Angular 2 for renderizar, esse componente vai verificar a *string* `meu conteúdo` entre as tags, assim ele já saberá que se trata de uma substituição desse conteúdo por uma tag `<ng-content>` `</ng-content>`.

Salvando e voltando ao navegador, veremos a frase meu conteúdo dentro da estrutura da tabela:



Figura 4.9: Conteúdo sendo enviado por ng-content

Com isso, podemos enviar o conteúdo que quisermos, e tudo será renderizado dentro do corpo da tabela criada em HTML. Veja que não precisamos refazer uma nova tabela, somente mudamos o conteúdo de envio.

Mas como fazemos para enviar o título e várias frases para dentro da nossa tabela padrão? Para isso, vamos usar a propriedade `select` junto com a diretiva `ng-content`, e diferenciar cada `select` com um nome de classe. Veja:

```
<ng-content select=".corpo"></ng-content>
```

Feito isso, estamos dizendo para o Angular 2 somente substituir a tag `<ng-content>` `</ng-content>` por conteúdo que tenha a classe `corpo`.

Agora, para tudo funcionar, vamos à tag pai que está enviando o conteúdo da frase e colocamos uma tag `div` com a classe `corpo` .

```
<app-ng-content>
  <div class="corpo">
    meu conteúdo com select
  </div>
</app-ng-content>
```

Salve, volte ao navegador e veja que teremos o mesmo conteúdo passado, porém agora estamos selecionando onde cada informação deve ficar.



Figura 4.10: Conteúdo sendo enviado por `ng-content` com `select`

Já sabemos como direcionar onde cada conteúdo deve ficar, agora fica fácil enviar o título da tabela junto com conteúdos para adicionar no corpo, certo? É somente especificar os locais com a propriedade `select` .

Vamos usar a classe `titulo` para o título da tabela, e a classe

corpo para o conteúdo da tabela. Podemos usar quantas vezes for necessário para passar os conteúdos. Aqui vamos enviar uma classe `titulo` e duas classes `corpo`.

Vamos enviar esse conteúdo do componente pai:

```
<app-ng-content>
  <div class="titulo">
    meu título passado
  </div>

  <div class="corpo">
    meu conteúdo com select
  </div>

  <div class="corpo">
    segundo conteúdo passado
  </div>
</app-ng-content>
```

No nosso arquivo `ng-content.component.html`, selecionaremos onde cada informação deve ficar com essa estrutura:

```
<h2>
  < ng-content>
</h2>

<table>
  <thead>
    <tr>
      <th>
        <ng-content select=".titulo"></ng-content>
      </th>
    </tr>

  </thead>

  <tbody>
    <tr>
      <td>
        <ng-content select=".corpo"></ng-content>
      </td>
    </tr>
  </tbody>
</table>
```

```

        </td>
    </tr>

</tbody>
</table>

```

Salvando e voltando ao navegador, teremos nossa tabela toda customizada com as informações que passamos.

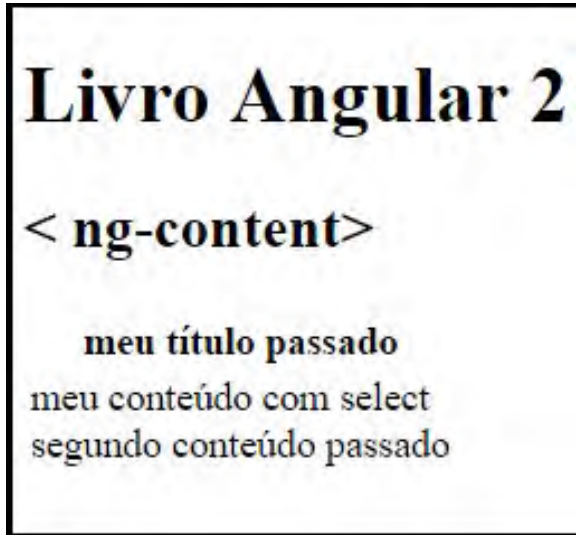


Figura 4.11: Todo conteúdo da tabela sendo enviado por ng-content com select

Com o `ng-content`, fica fácil padronizar uma estrutura para ser usada dentro do projeto e somente mudando os conteúdos. Existem outras formas de passar conteúdos do componente pai para o componente filho, mas isso vamos conhecer no próximo capítulo.

Nosso arquivo `ng-content.component.html` ficou assim:

```

<h2>
  < ng-content>

```

```

</h2>

<table>
  <thead>
    <tr>
      <th>
        <ng-content select=".titulo"></ng-content>
      </th>
    </tr>

  </thead>

  <tbody>
    <tr>
      <td>
        <ng-content select=".corpo"></ng-content>
      </td>
    </tr>

  </tbody>
</table>

```

Nosso arquivo `g-content.component.ts` ficou assim:

```

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

@Component({
  selector: 'app-ng-content',
  templateUrl: './ng-content.component.html',
  styleUrls: ['./ng-content.component.css']
})
export class NgContentComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}

```

4.11 RESUMO

Neste capítulo, aprendemos os tipos de exibição do Angular 2,

começamos mostrando os tipos de *bind* que podemos usar, falamos do *interpolation*, *property binding* e *two-way data binding*. Iniciamos as diretivas estruturais mostrando o `ngIf` , depois `ngSwitch` e `ngFor` . Vimos que é possível fazer o *reaproveitamento de lista* junto com o `ngFor` .

Partimos para as diretivas de atributos com o `ngClass` e `ngStyle` , e fechamos falando do uso do `ngContent` . Espero que esteja gostando da leitura deste livro, e continuemos os estudos!

ENTRADA DE DADOS DO USUÁRIO

Quando o usuário interage com a nossa aplicação, ele está interagindo diretamente com os componentes HTML da página, como por exemplo, um botão, uma caixa de texto, uma lista de opções, um menu, entre outros. Essas interações são chamadas de **eventos**, e o usuário pode fazê-las com um clique em um botão, passar o mouse por cima de algum componente da tela, colocar o foco em um campo de input para digitar um texto, ou até clicar em algum link da tela.

Nós podemos capturar esses eventos no template e tratar diretamente na classe do componente. Para isso, vamos pegar o valor do template HTML e enviar para a classe do componente, e assim efetuar alguma lógica de programação de acordo com a regra de negócio da aplicação.

As interações do usuário com a nossa aplicação sempre serão manipulando dados do template e passando para a nossa classe do componente.

5.1 EVENT BINDING

Neste livro, já falamos sobre *interpolation*, *property binding* (que faz ligação de dados da classe do componente para o template) e *two-way binding* (que tem a via de duas mãos, ou seja, atualiza tanto o template como a classe do componente). Para finalizar os *bindings* do Angular 2, vamos estudar como usar o *event binding*, que faz a ligação de dados do template para a classe do componente.

Vamos usar o *event binding* para disparar eventos dos elementos do DOM, em que o usuário faz a interação. Para vincular um evento do DOM com o *event binding* no Angular 2, teremos de colocar o nome do evento do DOM dentro de parênteses (), e atribuir a ele um método que estará na classe do componente para ser executado. Segue um exemplo de código:

```
<button (click)="meuClick()">Click aqui</button>
```

Do lado esquerdo, colocamos entre parênteses, (), o evento que queremos capturar na interação com o usuário. Neste caso, queremos o evento de clicar no botão. Logo em seguida, após o símbolo de = (igual), temos entre aspas (" ") o método da classe do componente que será executado quando esse botão for clicado, neste caso, o método `meuClick()`.

Essa forma que usamos é a simples de *event binding*, porém na chamada do método da classe do componente, podemos enviar informações para serem manipuladas dentro do método. Para isso, precisamos passar por parâmetro todo o evento do elemento HTML dessa forma:

```
<input (keyup)="digitou($event)">
```

Neste `input`, estamos capturando o evento `keyup` do

`input` . Isto é, a cada tecla pressionada no teclado do computador dentro dessa caixa de `input` , o método `keyup` será executado chamando o método da classe do componente `digitou` , e passando como parâmetro todas as propriedades da tag `input` .

Vamos criar um novo componente com o nome de `event-binding` para exercitarmos nosso conhecimento. Não esqueça que, após a criação, temos nossos procedimentos de adição e comentário de tag, e também criação do título do componente que, nesse caso, será **event binding**.

Logo após os procedimentos, vamos adicionar ao template os dois exemplos apresentados anteriormente.

```
<h2>event binding</h2>

<button (click)="meuClick()">Click aqui</button>

<input (keyup)="digitou($event)">
```

Na nossa classe do componente, vamos criar o método `meuClick()` , que enviará uma mensagem no *console* do navegador, e o método `digitou($event)` , que vai enviar como parâmetro as propriedades do elemento `input` . Dentro do método, enviaremos para o *console* do navegador os valores passados.

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

@Component({
  selector: 'app-event-binding',
  templateUrl: './event-binding.component.html',
  styleUrls: ['./event-binding.component.css']
})
export class EventBindingComponent implements OnInit {

  constructor() { }
```

```

ngOnInit() {
}

meuClick(): void {
  console.log('evento meuClick do botão');
}

digitou($event): void {
  console.log($event);
}

```

Salvando e voltando ao navegador, veremos um botão e uma caixa de texto. Apertando o botão **F12** do teclado, será aberto um painel abaixo no navegador com um menu superior. Nesse menu, clique na opção **console**. Agora clique no botão **Click aqui** do nosso projeto e digite uma letra na caixa de texto.

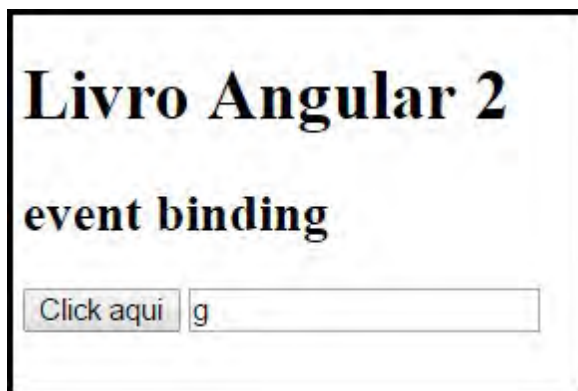


Figura 5.1: Nosso navegador com os dois elementos HTML

Veja no console do navegador que terá a frase **evento meuClick do botão** e, abaixo, um **keyboardEvent**. Clique na seta ao lado para abrir uma série de informações que foram enviadas no *event binding* sobre a tag **input**.


```
evento meuClick do botão  
▶ KeyboardEvent {isTrusted: true, key: "g", code: "KEYG", Location: 0, ctrlKey: false...}
```

Figura 5.2: Conteúdo enviado pelo event binding

Veja que temos uma série de informações sobre a tag `input` e, entre elas, mais setas laterais para abrir mais informações.

Vamos até a propriedade `target` e clicaremos na seta ao lado. Isso abrirá uma outra lista, na qual vamos procurar a propriedade `value`, que terá a letra que você digitou.

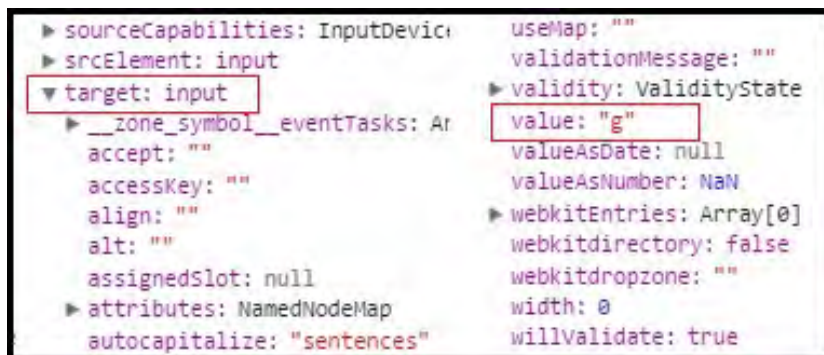


Figura 5.3: Árvore de informação do elemento input

Veja que, nessa forma de passagem de parâmetro, muita informação é enviada sem necessidade, e isso prejudica a performance da aplicação, já que são muitos dados a serem passados do template para a classe do componente.

Pensando nisso que a equipe do Google adicionou no Angular 2 uma outra forma, menos custosa no desempenho do projeto, que é passar informações do template para o componente com a *variável de template*.

5.2 VARIÁVEL DE REFERÊNCIA DO TEMPLATE

Para facilitar o acesso direto ao valor do elemento HTML que está na página, usamos as **variáveis de template**.

Para utilizar uma variável de template dentro de uma tag do HTML, devemos colocar dentro dela o caractere de # , seguido do nome da variável. Com isso, o Angular 2 vai referenciar o valor que está contido no elemento HTML para dentro da variável de template.

Vamos criar um novo `input` que terá o mesmo *event binding* de `keyup` , mas agora ele executará o método `digitouVarTemplate(valorInput)` , que passará como parâmetro a variável de template `valorInput` . Esse método também vai enviar o valor recebido do parâmetro para dentro do console do navegador.

```
<input #valorInput (keyup)="digitouVarTemplate(valorInput.value)":
```

Nosso método na classe do componente ficará assim:

```
digitouVarTemplate(valor): void {  
  console.log(valor);  
}
```

Salvando e voltando ao navegador, quando digitamos alguma letra na nova caixa de texto, somente o conteúdo dessa caixa de texto será enviado para o método da classe do componente. Isso melhorará muito o desempenho da aplicação.

Veja no console do navegador o valor que foi passado usando a *variável de template*.



Figura 5.4: Valor enviado com a variável de template

Essa será a forma mais recomendada para passagem de valor em um evento do template para a classe do componente. Dessa forma, teremos somente o valor do elemento HTML contido na variável, e isso vai facilitar muito, pois não receberemos todo o conteúdo do elemento HTML.

5.3 (CLICK)

Agora estudaremos os eventos mais usados dentro de uma aplicação. Começamos pelo, já bem conhecido, `click`. Esse evento será usado quando queremos capturar o *click do mouse* em algum elemento do HTML na página. Ele é mais usado em botões e links para ser executado algum método dentro da aplicação, ou para o usuário ser redirecionado para alguma outra página.

Não vamos criar nenhum novo elemento HTML para exemplificar o evento de clique, pois, até esse momento do livro, já usamos o evento de clique diversas vezes, então ficaria desnecessário fazer esse exemplo. Mas quero que fique ciente de que o evento de clique pode ser usado em vários elementos dentro da nossa aplicação.

Nosso evento de clique sempre executará um método da classe do componente. Podemos somente executar o método, ou também passar valores para a classe do componente com as variáveis de template que já estudamos.

5.4 (KEYUP)

O evento de `keyup` será sempre usado na captura de alguma tecla pressionada no teclado. Sempre que alguma tecla do teclado for pressionada dentro de uma caixa de texto, esse evento será executado, fazendo alguma validação de método dentro da classe do componente.

Ele é muito usado para fazer validação de conteúdo dentro de uma caixa de texto, na qual, por exemplo, queremos saber se o valor contido nela está vazio ou se tem a quantidade de caracteres necessários de acordo com a regra de negócio.

Vamos criar um exemplo para o uso do evento `keyup`, criando uma caixa de texto para ser digitada uma senha. Esta deverá ter, no mínimo, 5 caracteres. Se a caixa de texto tiver menos que isso, o botão de **gravar senha** será desabilitado.

No nosso template, vamos criar uma tag `input` do tipo `password` com uma variável de template com o nome `#qtdSenha`, e um evento de `keyup` executando o método `validaSenha(qtdSenha.value)`, que passará como parâmetro o valor da variável de template.

Em seguida, criamos uma tag `button` com o conteúdo de **gravar senha**. Dentro da tag, teremos um *property binding* para a propriedade `disabled` do botão, referenciando a variável `!habilitarBotao` que está na classe do componente, e um evento de clique que executa o método `gravarSenha(qtdSenha.value)` que enviará um alerta no navegador.

Nosso template ficará assim:

```

<h3>evento keyup para validação de senha</h3>
<input type="password" #qtdSenha (keyup)="validaSenha(qtdSenha.value)">
<button [disabled]="! habilitarBotao" (click)="gravarSenha(qtdSenha.value)">gravar senha</button>

```

Nossa classe do componente ficará assim:

```

export class EventBindingComponent implements OnInit {

    habilitarBotao: boolean = false;

    /* mesmo código anterior */

    validaSenha(valor: string): void {
        if(valor.length >= 5){
            this.habilitarBotao = true;
        }
        else{
            this.habilitarBotao = false;
        }
    }

    gravarSenha(senha): void {
        alert('senha gravada com sucesso sua senha é: ' + senha);
    }
}

```

Salvando e voltando ao navegador, nosso botão inicia desabilitado, pois a variável `habilitarBotao` é iniciada como `false`. Conforme vamos digitando dentro da caixa de texto, será executado o evento de `keyup`, que verificará a quantidade de caracteres. Se for igual ou maior que 5, o botão será habilitado.

Habilitando o botão, podemos clicá-lo, e será enviado um alerta no navegador com o frase `senha gravada com sucesso sua senha é:`, junto com a senha digitada. Esse foi um dos possíveis usos do evento `keyup` dentro do Angular 2.

5.5 (KEYUP.ENTER)

O evento de `keyup` é executado a cada tecla pressionada do teclado. Algumas vezes, isso pode ser desnecessário, pois só queremos executar alguma coisa quando o usuário pressionar a tecla `Enter`. Para isso, temos o evento de `keyup.enter`, que será executado somente quando essa tecla for pressionada.

Para isso, vamos criar uma tag `input` do tipo `text` com uma variável de template com o nome de `#conteudo` e também um evento `(keyup.enter)="adicionar(conteudo.value); conteudo.value=' '`. Quando a tecla `Enter` for pressionada, ela executará o método `adicionar(conteudo.value)`. Em seguida, vamos limpar a caixa com o comando `conteudo.value=' '`.

Logo abaixo da tag `input`, criaremos uma lista que mostrará tudo o que vamos digitar na caixa de texto. Vamos usar na lista o já conhecido `*ngFor` para listar todo o conteúdo.

Nosso template ficará assim:

```
<h3>evento keyup.enter adicionar na lista</h3>
<input type="text" #conteudo (keyup.enter)="adicionar(conteudo.value); conteudo.value=' '">
<ul>
  <li *ngFor="let item of valores">{{item}}</li>
</ul>
```

Na nossa classe do componente, criaremos um array com o nome de `valores` e um método com o nome de `adicionar`, que vai receber o conteúdo da variável de template `#conteudo` e adicionar no array `valores`.

```
export class EventBindingComponent implements OnInit {

  habilitarBotao: boolean = false;
  valores: string [] = [];

  /* mesmo código anterior*/
```

```
adicionar(conteudo: string): void {
    this.valores.push(conteudo);
}
```

Salvando e voltando ao navegador, cada vez que digitarmos algo na caixa de texto e apertarmos a tecla `Enter`, o conteúdo será enviado para o método `adicionar` e adicionado no array `valores`. Logo, esse novo conteúdo será visto na lista.

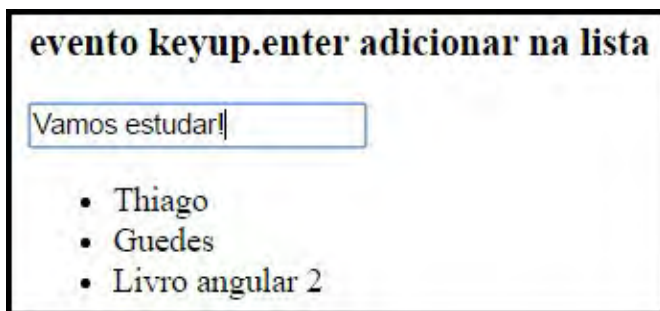


Figura 5.5: Caixa de texto com `keyup.enter` mostrado na lista

5.6 (BLUR)

O último evento que vamos estudar é o `blur`, usado quando o usuário tira o foco de dentro da caixa de texto. Ou seja, quando ele está digitando algo e, após isso, clica em algum lugar fora dessa caixa.

Vamos usar esse evento para executar um método que informará a idade de uma pessoa digitando o ano na caixa de texto. Criaremos uma tag `input` do tipo `text`, com uma variável de template com o nome de `#data` e um evento `(blur)="verIdade(data.value)"`. Este vai executar o método `verIdade` quando o usuário clicar fora da caixa de texto.

Abaixo da tag `input` , vamos adicionar um parágrafo com a tag `p` para verificar o resultado do evento `blur` . Nosso template ficará assim:

```
<h3>blur</h3>
<input type="text" #data (blur)="verIdade(data.value)">
<p>Você tem: {{idade}} anos de idade</p>
```

Na nossa classe do componente, vamos criar um método com o nome `verIdade(valor)` que, quando for executado, vai capturar o ano atual e subtrair com o ano digitado na caixa de texto. O resultado será atribuído na variável `idade` .

Nossa classe do componente ficará assim:

```
export class EventBindingComponent implements OnInit {

  habilitarBotao: boolean = false;
  valores: string [] = [];
  idade: number = 0;

  /* mesmo código anterior*/

  verIdade(valor): void {
    let ano = new Date();
    this.idade = ano.getFullYear() - valor;
  }
}
```

Salvando e voltando ao navegador, ao digitar o ano dentro da caixa de texto e após clicar fora da caixa, o evento `blur` será executado junto com o método `verIdade` , fazendo com que a mensagem no parágrafo seja atualizada.

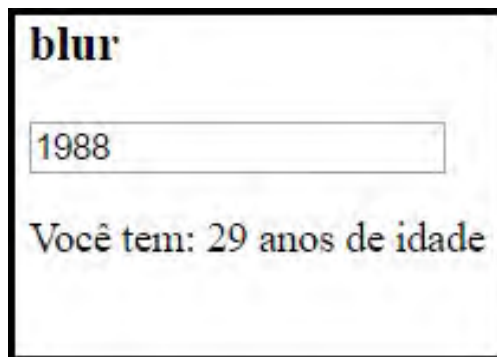


Figura 5.6: Evento blur atualizando o parágrafo de idade

Nosso arquivo `event-binding.component.html` ficou assim:

```
<h2>event binding</h2>

<button (click)="meuClick()">Click aqui</button>

<input (keyup)="digitou($event)">

<input #valorInput (keyup)="digitouVarTemplate(valorInput.value)":

<h3>evento keyup para validação de senha</h3>
<input type="password" #qtdSenha (keyup)="validaSenha(qtdSenha.va
lue)">
<button [disabled]="! habilitarBotao" (click)="gravarSenha(qtdSen
ha.value)">gravar senha</button>

<h3>evento keyup.enter adicionar na lista</h3>
<input type="text" #conteudo (keyup.enter)="adicionar(conteudo.va
lue); conteudo.value=''">
<ul>
  <li *ngFor="let item of valores">{{item}}</li>
</ul>

<h3>blur</h3>
<input type="text" #data (blur)="verIdade(data.value)">
<p>Você tem: {{idade}} anos de idade</p>
```

Nosso arquivo `event-binding.component.ts` ficou assim:

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

@Component({
  selector: 'app-event-binding',
  templateUrl: './event-binding.component.html',
  styleUrls: ['./event-binding.component.css']
})
export class EventBindingComponent implements OnInit {

  habilitarBotao: boolean = false;
  valores: string [] = [];
  idade: number = 0;

  constructor() { }

  ngOnInit() {
  }

  meuClick(): void {
    console.log('evento meuClick do botão');
  }

  digitou($event): void {
    console.log($event);
  }

  digitouVarTemplate(valor: string): void {
    console.log(valor);
  }

  validaSenha(valor: string): void {
    console.log(valor);
    if(valor.length >= 5){
      this.habilitarBotao = true;
    }
    else{
      this.habilitarBotao = false;
    }
  }

  gravarSenha(senha: string): void {
    alert('senha gravada com sucesso sua senha é: ' + senha);
  }
}
```

```

    }

    adicionar(conteudo: string): void {
        this.valores.push(conteudo);
    }

    verIdade(valor): void {
        let ano = new Date();
        this.idade = ano.getFullYear() - valor;
    }
}

```

5.7 @INPUT() PROPERTY

O `Input()` é uma propriedade no Angular 2 que serve para passar o valor do componente pai para o componente filho. Ele é muito parecido com o `ng-content`. Com o `@Input()`, podemos criar componentes genéricos para receber ou enviar informações que usam uma estrutura HTML padrão, como por exemplo, listas e tabelas.

Para usar a propriedade `@Input()`, devemos colocá-la antes de alguma variável dentro de um componente. Assim, ela ficará visível para o componente pai poder adicionar conteúdo nela. Segue o exemplo do *decorator* `@Input()`.

```
@Input() minhaVariavel: string;
```

Neste exemplo, a variável `minhaVariavel` recebe o decorador `@Input()`, assim, o componente pai poderá manipulá-la. Com isso, teremos a comunicação entre componentes: o componente pai manipulando conteúdo do componente filho.

Para usar neste exemplo, vamos criar um componente com o nome `input-output` que terá uma estrutura de lista e receberá um array do tipo `string`.

```
<h2>@Input() / @Output()</h2>
```

```
<ul>
  <li *ngFor="let item of menu">
    {{item}}
  </li>
</ul>
```

Na nossa classe do componente, teremos um array com o nome de menu do tipo string, porém, esse array virá do componente pai. Então, devemos adicionar o decorador @Input(), que faz parte do pacote do @angular/core, e devemos importá-lo antes de usar.

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

@Component({
  selector: 'app-input-output',
  templateUrl: './input-output.component.html',
  styleUrls: ['./input-output.component.css']
})
export class InputOutputComponent implements OnInit {

  @Input() menu: string;

  constructor() { }

  ngOnInit() {
  }
}
```

Neste momento, nosso componente input-output.component já está preparado para receber dados do seu componente pai, que neste caso será o app.component. Vamos ao nosso arquivo app.component.ts para criar um array com o nome de desenvolvimento. Ele será do tipo string.

```
export class AppComponent {

  title: string = 'Livro Angular 2';
```

```
foto: string = 'favicon.ico';
desenvolvimento: string [] = ['Angular 2', 'JavaScript', 'Typescript', 'HTML', 'CSS'];
```

Queremos passar esse array para nosso componente filho, o `input-output.component` . Vamos fazer isso referenciando o array `desenvolvimento` junto com o array `menu` por meio de *property binding*.

No nosso `app.component.html` , temos a tag `<app-input-output></app-input-output>` que vai renderizar o componente `input-output.component` neste local. Este componente tem uma variável que está decorada com `@Input()` e pode ser acessada pelo componente pai por meio de uma ligação de propriedade, ou seja, um *property binding*.

Para passar valores do componente pai para o filho, faremos da seguinte forma:

```
<app-input-output [menu]="desenvolvimento"></app-input-output>
```

Estamos referenciando o array `desenvolvimento` que está no componente pai (`app.component`) com uma propriedade `menu` do componente filho (`input-output.component`).

Salvando e voltando ao navegador, teremos a comunicação entre componentes. Os dados que estão no componente pai estão sendo passados para o filho e sendo renderizados na estrutura de lista dele.



Figura 5.7: Lista renderizada com @Input(), componente pai para o filho

O uso do @Input é muito fácil e simples, e não temos nada de complexo. Ele é muito utilizado para passar informações entre componentes. Constantemente vamos fazer esses tipos de comunicação dentro de aplicações Angular 2.

Por ser um framework dividido em componentes, cada componente construído será responsável por uma parte do problema, e a comunicação e troca de informações entre eles farão a construção de uma aplicação final.

5.8 @OUTPUT PROPERTY

Da mesma forma que o componente pai pode se comunicar com o componente filho enviando dados, também podemos fazer o filho se comunicar com o pai enviando dados por meio de uma

variável decorada com `@output()` , em conjunto com um evento chamado de `EventEmitter()` .

O `@output()` é o inverso do `@Input()` . Ou seja, ele vai servir para o componente filho enviar as informações para o componente pai, e é no pai que os dados passados serão processados.

Porém, para o `@output()` funcionar corretamente, temos de usá-lo em conjunto com o `EventEmitter()` . É ele que vai emitir os dados e avisar ao componente pai que o componente filho tem dados para passar.

Vamos continuar usando o componente `input-output.component` , mas agora queremos capturar o evento de clique em cada elemento da lista. Logo em seguida, vamos capturar o nome que foi clicado e enviar para o componente pai processar a informação.

No nosso arquivo de template `input-output.component.html` , adicionaremos a tag `li` , dentro de uma tag `a` , para a lista atual ter uma aparência de clicável. E dentro da tag `a` , criaremos um evento de clique que executará o método `enviarNome(item)` .

```
<h2>@Input() / @Output()</h2>

<ul>
  <a href="#" (click)="enviarNome(item)" *ngFor="let item of menu">
    <li>
      {{item}}
    </li>
  </a>
</ul>
```

No nosso arquivo `input-output.component.ts`, vamos criar o método `enviarNome(item)` que será o responsável por emitir o nome do elemento clicado. Mas antes vamos criar uma variável com o nome de `nomeClicado` e atribuir na frente da variável o decorador `@Output()`, que também faz parte do `@angular/core` e deve ser importado.

```
import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
/* mesmo código anterior */
@Output() nomeClicado;
```

Como falei anteriormente, o `@Output()` somente funciona em conjunto com o evento `EventEmitter()`, então, para esta variável poder emitir as informações para o componente pai, ela deve ser instanciada como uma emissora de eventos.

```
export class InputOutputComponent implements OnInit {

  @Input() menu: string;
  @Output() nomeClicado = new EventEmitter();
```

Dessa forma, a variável `nomeClicado` já está pronta para emitir o evento quando necessário.

Voltamos para o método `enviarNome(value)`, que será responsável por notificar o componente pai sobre a emissão do evento do `EventEmitter()`. Dentro do método `enviarNome(value)`, vamos emitir o conteúdo que foi passado por parâmetro, da seguinte forma:

```
enviarNome(value){
  this.nomeClicado.emit(value);
}
```

Dessa forma, estamos emitindo pela variável `nomeClicado` o valor que foi passado pelo parâmetro `value`. Para o componente

pai recuperar o evento enviado pelo filho, devemos fazer um *event binding* dentro da tag `<app-input-output [menu]="desenvolvimento"></app-input-output>`, que está no arquivo `app.component.html`.

```
<app-input-output [menu]="desenvolvimento" (nomeClidado)="valorPassado($event)"></app-input-output>
```

Como este é um *event binding*, devemos executar algum método dentro da classe do componente. Então, vamos criar na classe do componente `app.component.ts` um método com o nome de `valorPassado($event)*` que receberá o valor enviado do componente filho como parâmetro.

Vamos criar também uma variável com o nome de `valor`, que vai receber o valor do parâmetro e que, após ele ser atribuído, mostraremos no navegador através de um *interpolation*.

Nosso arquivo `app.component.ts` será esse:

```
export class AppComponent {

  title: string = 'Livro Angular 2';
  foto: string = 'favicon.ico';
  desenvolvimento: string [] = ['Angular 2', 'JavaScript', 'TypeScript', 'HTML', 'CSS'];
  valor: string;

  /* mesmo código anterior */

  valorPassado(valorPassado){
    this.valor = valorPassado;
  }
}
```

Para verificar os valores sendo passados do componente filho para o componente pai, adicionaremos uma tag `p` logo abaixo da tag do componente `input-output.component*`, que está no

arquivo `app.component.html` .

```
<app-input-output [menu]="desenvolvimento" (nomeClicado)="valorPassado($event)"></app-input-output>
<p>você clicou em: {{valor}}</p>
```

Salvando todos os arquivos e voltando ao navegador, a cada clique em um item da lista, será mostrado logo abaixo qual o nome do item clicado. Com isso, mostramos as comunicações pai e filho e o de filho com o pai por meio dos decoradores `@Input()` e `@Output()` , junto com o evento `EventEmitter()` .



Figura 5.8: Comunicação com `@Input` / `@Output`

Nosso arquivo `input-output.component.html` ficou assim:

```
<h2>@Input() / @Output()</h2>
<ul>
  <a href="#" (click)="enviarNome(item)" *ngFor="let item of menu">
```

```

        <li>
            {{item}}
        </li>
    </a>
</ul>

```

Nosso arquivo `input-output.ts` ficou assim:

```

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

@Component({
  selector: 'app-input-output',
  templateUrl: './input-output.component.html',
  styleUrls: ['./input-output.component.css']
})
export class InputOutputComponent implements OnInit {

  @Input() menu: string;
  @Output() nomeClidado = new EventEmitter();

  constructor() { }

  ngOnInit() {
  }

  enviarNome(value){
    this.nomeClidado.emit(value);
  }
}

```

5.9 RESUMO

Neste capítulo, falamos sobre os tipos de tratamentos e execuções na interação do usuário com a nossa aplicação. Começamos falando do *event binding*, explicamos o que é *variável de referência do template*, e usamos os eventos de `click`, `keyup`, `keyup.enter` e `blur`.

Também falamos dos tipos de comunicação entre componente

pai e filho pelos decoradores `@Input()` property e `@Output` property . Com o final deste capítulo, em conjunto com o capítulo anterior, fechamos o ciclo de comunicação, que é exibir informações para o usuário mostrando as diferentes formas para se fazer isso.

Agora mostraremos como podemos recuperar as interações do usuário com a nossa aplicação. É muito importante o entendimento desses capítulos, pois os assuntos abordados até aqui serão constantemente usados. Continuemos os estudos!

FORMULÁRIOS

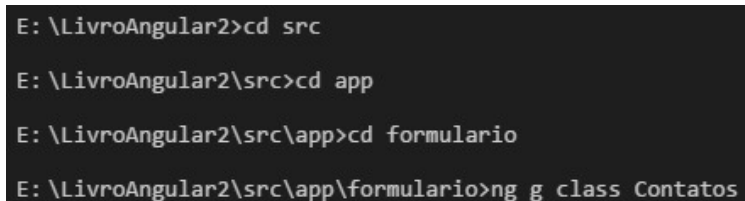
Constantemente interagimos com formulários em várias aplicações, como para efetuar um login em uma aplicação, para fazer um cadastro, para enviar uma solicitação, entre outras coisas. Se você já desenvolveu alguma aplicação, seja web, mobile ou desktop, certamente já se deparou com a construção de um formulário e a quantidade de validações que ele tem, conforme a regra de negócio da aplicação.

Neste capítulo, falaremos sobre formulários no Angular 2, as formas corretas e facilidades na criação de formulários, com validações, tratamentos de erros e envio de conteúdos quando os dados estiverem corretos. Veremos neste capítulo a criação de um formulário e como podemos efetuar tratamentos de erros, envio de mensagens de forma fácil e rápida. E tudo isso com a ajuda dos componentes do Angular 2.

Para iniciar com os exemplos, vamos criar um novo componente com o nome de `formulario` e seguir nossos procedimentos de adição de nova tag e remoção da tag anterior no arquivo `app.component.html`. Vamos colocar como título para este componente o nome de `formulário`.

Logo em seguida, entraremos na pasta deste componente para

criar uma simples *classe* que servirá como modelo de dados para o cadastro no nosso formulário. Vamos digitar no console do VS Code o comando `cd src\app\formulario` e apertar Enter . Agora estamos dentro da pasta do componente, vamos digitar o comando `ng g class Contatos` .



```
E: \LivroAngular2>cd src
E: \LivroAngular2\src>cd app
E: \LivroAngular2\src\app>cd formulario
E: \LivroAngular2\src\app\formulario>ng g class Contatos
```

Figura 6.1: Caminho para o componente formulario

Após a criação da classe dentro da pasta do componente `formulario.component` , vamos criar um método construtor com os argumentos `nome` , `telefone` e `email` — todos do tipo `string` .

```
export class Contato {
    constructor(public nome: string,
                public telefone: string,
                public email: string){
    }
}
```

6.1 NGMODEL, VARIÁVEL DE TEMPLATE E ATRIBUTO NAME DA TAG HTML

As variáveis locais de template em formulários no Angular 2 têm um papel fundamental para o funcionamento do sistema. Até aqui, usamos as variáveis locais somente para recuperar os valores

que estavam contidos dentro dos elementos do HTML. Mas quando estamos usando formulários, essa história muda.

As variáveis locais junto com o atributo `name` da tag `input` vão dar ao formulário uma forma de gerenciar o elemento HTML. Dessa forma, todo elemento HTML dentro do formulário, inclusive a própria tag `form`, obrigatoriamente necessita ter uma *variável local* e o atributo `name` para ser gerenciada e vista pelo gerenciador de formulário do Angular 2.

O atributo `name` dentro da tag HTML vai servir para o Angular 2 ter o controle total do elemento no formulário. E em conjunto com a diretiva `[(ngModel)]`, poderá fazer a consulta e atualização do conteúdo dentro da tag.

A variável local na tag `form` e nos elementos de `input` vão servir para transformar a tag em um objeto de formulário do Angular 2. Também servirá para recuperarmos o estado atual do elemento HTML, e assim efetuar as validações de acordo com a regra de negócio.

Para iniciar, vamos criar a estrutura de formulário que vai conter três campos `input` para digitação de conteúdo: `nome`, `telefone`, `email`. No final, teremos um botão com a tag `button` que vai servir para enviar as informações.

Nossa estrutura HTML ficará assim:

```
<h2>formulário</h2>

<form>
  <div>
    <label for="nome">Nome</label>
    <input type="text" id="nome" name="nome">
  </div>
```

```

<br>

<div>
  <label for="telefone">Telefone</label>
  <input type="text" id="telefone" name="telefone">
</div>

<br>

<div>
  <label for="email">Email</label>
  <input type="text" id="email" name="email">
</div>

<br>

<button type="submit">Enviar</button>
</form>

```

Veja que, logo de começo, já coloquei o atributo `name` para todos os elementos HTML dentro do formulário. Com isso, já conseguimos usar a diretiva `[(ngModel)]`, pois, como falei anteriormente, ela só funciona em conjunto com esse atributo.

Vamos ao arquivo `formulario.component.ts` para criar uma instância da classe `Contatos`. Não podemos esquecer de importar essa classe para dentro da classe do componente.

```

export class FormularioComponent implements OnInit {

  contato = new Contatos('Thiago', '(99)99999-9999', 'email@email.com');
}

```

Com o objeto `contato` instanciado dentro da classe do componente, e a tag do formulário HTML no template com o atributo `nome`, já podemos usar o `[(ngModel)]` para vincular os dados da classe componente no template, dessa forma:

```

<h2>formulário</h2>

```



```

<form>
  <div>
    <label for="nome">Nome</label>
    <input type="text" id="nome" [(ngModel)]="contato.nome" n
ame="nome">
  </div>

  <br>

  <div>
    <label for="telefone">Telefone</label>
    <input type="text" id="telefone" [(ngModel)]="contato.tel
efone" name="telefone">
  </div>

  <br>

  <div>
    <label for="email">Email</label>
    <input type="text" id="email" [(ngModel)]="contato.email"
name="email">
  </div>

  <br>

  <button type="submit">Enviar</button>
</form>

```

Salvando e voltando ao navegador, já teremos o formulário com os dados do contato.



Livro Angular 2

formulário

Nome

Telefone

Email

Figura 6.2: Formulário sendo mostrado no navegador

Já usamos o atributo `name` e a diretiva `[(ngModel)]` para vincular e mostrar os dados da classe do componente para o template. Agora vamos usar as variáveis de referência do template e fazer com que o gerenciador de formulários do Angular 2 reconheça nosso formulário.

Para começar, vamos dizer para o Angular 2 que o que estamos fazendo é um formulário. Para isso, vamos na tag `form` e adicionamos uma variável local de template. Entretanto, essa variável será construída de uma forma diferente:

```
<h2>formulário</h2>
```

```
<form #cadastro="ngForm">
```

```

<div>
  <label for="nome">Nome</label>
  <input type="text" id="nome" [(ngModel)]="contato.nome" n
ame="nome">
</div>

/* restante do código é o mesmo */

```

Veja que estamos atribuindo para a variável de template um valor, o `ngForm`. Mas o que isso significa?

Antes colocávamos somente uma variável local, dessa forma: `#cadastro`. Fazendo isso, podíamos pegar dados do elemento HTML e trabalhar da forma que quiséssemos, porém não conseguíamos fazer nada a mais do que uma simples referência para o elemento HTML.

Atribuindo um valor para a variável de template, nós *transformamos* essa variável em uma diretiva, fazendo com que ela se comporte da mesma forma que uma diretiva tradicional. Então, quando colocamos a variável de template sendo atribuída com o valor de `ngForm`, estamos dizendo para o Angular 2 utilizá-la como fosse uma diretiva de formulário.

Adicionando `#cadastro="ngForm"` na tag `form`, o Angular 2 agora já sabe que essa estrutura se refere a uma estrutura de formulário.

6.2 VALIDAÇÕES DE FORMULÁRIO

Neste momento, nosso formulário está funcionando corretamente. Porém, as caixas de texto `nome`, `telefone` e `email` estão aceitando qualquer valor, inclusive valores nulos.

Em um formulário tradicional, temos várias regras antes de poder enviar ou gravar as informações, como por exemplo, o campo de *nome* não pode estar em branco, já que todo contato deve conter um nome. O mesmo acontece com o campo de *telefone*, como estamos criando um formulário de contato, devemos ter o telefone dele.

Para isso, devemos validar cada elemento dentro do formulário. E se algum elemento que é obrigatório não estiver preenchido, nosso formulário enviará uma mensagem de erro.

Para criar isso no Angular 2, é muito fácil e simples. Devemos colocar uma variável de referência em cada elemento HTML dentro do formulário, e atribuir para essa variável o valor de `ngModel`. Neste momento, você já sabe o que vai acontecer, correto?

Da mesma forma que fizemos `#cadastro="ngForm"`, e a variável de template se transformou em uma diretiva `ngForm`, se fizermos `#nome="ngModel"`, a variável de referência `#nome` vai se transformar em uma diretiva `ngModel`. Ué, mas já tínhamos colocado o `[(ngModel)]` dentro da tag de nome. Por que colocar novamente?

A diretiva `[(ngModel)]="contato.nome"` em conjunto com o atributo `name="nome"` estão trabalhando referenciando a variável `* contato`, que está dentro da classe do componente. Como queremos recuperar o estado atual (se válido ou não) do elemento HTML, devemos criar uma variável que ficará responsável para mostrar esses estados.

E como os estados do elemento HTML podem ficar mudando

constantemente, devemos atribuir para a variável de template o comportamento do `ngModel`. Isso vai fazer com que o elemento HTML se torne um objeto de formulário no Angular 2.

Então, vamos adicionar para cada elemento HTML dentro do formulário a variável de template, sendo atribuído o valor de `ngModel` para cada uma delas. Também devemos colocar nas tags `nome` e `telefone` o atributo `required`, pois esses dois valores serão obrigatórios em nosso formulário.

```
<h2>formulário</h2>

<form #cadastro="ngForm">
  <div>
    <label for="nome">Nome</label>
    <input type="text" id="nome" [(ngModel)]="contato.nome" #
nome="ngModel" name="nome" required>
  </div>

  <br>

  <div>
    <label for="telefone">Telefone</label>
    <input type="text" id="telefone" [(ngModel)]="contato.tel
efone" #telefone="ngModel" name="telefone" required>
  </div>

  <br>

  <div>
    <label for="email">Email</label>
    <input type="text" id="email" [(ngModel)]="contato.email"
#email="ngModel" name="email">
  </div>

  <br>

  <button type="submit">Enviar</button>
</form>
```

Com isso, já podemos recuperar o estado atual de validação de

cada elemento HTML do formulário, e eles se tornaram **objetos de formulário** no Angular 2. Mas como podemos validar cada objeto do formulário?

No Angular 2, temos vários tipos de classes de validações que vão ajudar muito na construção de um formulário. Elas vão nos mostrar se um objeto dele está válido, se foi visitado, se foi alterado, entre outras coisas. Para facilitar, construímos essa tabela:

Estado do objeto	Verdadeiro	Falso
Objeto foi clicado	<i>ng-touched</i>	<i>ng-untouched</i>
Valor do objeto foi mudado	<i>ng-dirty</i>	<i>ng-pristine</i>
Objeto está válido	<i>ng-valid</i>	<i>ng-invalid</i>

Seguindo a tabela, podemos obter diferentes tipos de validações do objeto do formulário. Vamos explicar cada classe.

Se a aplicação foi iniciada e o campo ainda não foi clicado, ele recebe a classe `ng-untouched`.

Se o campo foi clicado, ele recebe a classe `ng-touched`.

Se o conteúdo do campo não foi alterado, ele recebe a classe `ng-pristine`.

Se o conteúdo do campo já foi alterado, recebe a classe `ng-dirty`.

Se o valor do campo está válido para envio, recebe a classe `ng-valid`.

Se o valor do campo está inválido para envio, recebe a

classe `ng-invalid` .

Com essa classe, podemos validar os objetos do formulário e, assim, de acordo com a nossa regra de negócio, receber os dados ou enviar uma mensagem de erro para a aplicação. Vamos adicionar para cada objeto dentro do formulário uma validação, pegando como referência as classes de estado de cada objeto.

Para os campos `nome` e `telefone` , vamos verificar se eles já foram visitados e se estão válidos para envio. Para o campo `email` , vamos fazer duas validações: uma verificando quando o conteúdo foi mudado e outra quando o usuário retirou o foco de digitação desse campo.

Se você lembra no capítulo que falamos sobre a diretiva `ngIf` , comentamos quando usá-lo e usar o `hidden` . Isso vai depender do tipo da árvore de estrutura, ou do nível de segurança da informação.

Aqui vou fazer os exemplos com as duas formas. Não que isso seja correto, mas quero mostrar que, usando `ngIf` ou `hidden` , o processo será o mesmo, o que vai definir sua escolha entre usar um ou o outro são os pontos levantados no capítulo que já apresentamos. Nosso código ficará assim:

```
<h2>formulário</h2>

<form #cadastro="ngForm">
  <div>
    <label for="nome">Nome</label>
    <input type="text" id="nome" #nome="ngModel" [(ngModel)]="
contato.nome" name="nome" required>

    <div [hidden]="nome.valid || nome.pristine">
      <p>O valor nome é obrigatório</p>
    </div>
```

```

</div>

<br>

<div>
  <label for="telefone">Telefone</label>
  <input type="text" id="telefone" [(ngModel)]="contato.tel
efone" #telefone="ngModel" name="telefone" required>

  <div *ngIf="telefone.invalid && telefone.dirty">
    <p>O número do Telefone é obrigatório</p>
  </div>
</div>

<br>

<div>
  <label for="email">Email</label>
  <input type="text" id="email" [(ngModel)]="contato.email"
#email="ngModel" name="email">

  <div [hidden]="email.pristine">
    <p>O conteúdo foi mudado</p>
  </div>
  <div *ngIf="email.touched">
    <p>Clicou fora do campo</p>
  </div>
</div>

<br>

  <button type="submit">Enviar</button>
</form>

```

Salve, volte ao navegador e clique em cada campo modificando os conteúdos. Assim, as mensagens serão mostradas conforme cada validação.

Livro Angular 2

formulário

Nome

O valor nome é obrigatório

Telefone

O número do Telefone é obrigatório

Email

O conteúdo foi mudado

Clicou fora do campo

Figura 6.3: Mostrando todas as validações do formulário

6.3 NGMODELGROUP

Assim como podemos enviar cada campo separado, também podemos agrupar os campos para serem enviados juntos. Usamos o `ngModelGroup` para gerenciar uma quantidade de campos dentro de um formulário. Dessa forma, podemos agrupar alguns

campos que podem ter o significado parecido.

No nosso formulário, temos os campos de telefone e email, e os dois valores servem como meios de contatos. Então, podemos agrupá-los e criar um componente interno no formulário.

Para isso, vamos envolvê-los em uma tag `div`, adicionar uma variável de referência com o nome de `#todosContatos`, e atribuir com o valor de `ngModelGroup*`. Em seguida, adicionamos dentro da tag `div` um `ngModelGroup=""` e atribuímos o nome de `todosContatos`.

No final do formulário, faremos uma interpolação com o operador *pipe* e mostraremos os valores agrupados.

```
<h2>formulário</h2>

<form #cadastro="ngForm">
  <div>
    <label for="nome">Nome</label>
    <input type="text" id="nome" #nome="ngModel" [(ngModel)]="
contato.nome" name="nome" required>

    <div [hidden]="nome.valid || nome.pristine">
      <p>O valor nome é obrigatório</p>
    </div>
  </div>

  <br>
  <div ngModelGroup="todosContatos" #todosContatos="ngModelGrou
p">
    <div>
      <label for="telefone">Telefone</label>
      <input type="text" id="telefone" [(ngModel)]="contato
.telefone" #telefone="ngModel" name="telefone" required>

      <div *ngIf="telefone.invalid && telefone.dirty">
        <p>O número do Telefone é obrigatório</p>
```

```

        </div>
    </div>

    <br>

    <div>
        <label for="email">Email</label>
        <input type="text" id="email" [(ngModel)]="contato.em
        ail" #email="ngModel" name="email">

        <div [hidden]="email.pristine">
            <p>O conteúdo foi mudado</p>
        </div>
        <div *ngIf="email.touched">
            <p>Clicou fora do campo</p>
        </div>
    </div>
</div>

<br>

    <button type="submit">Enviar</button>
</form>

<p>{{todosContatos}}</p>
<p>{{todosContatos.value | json}}</p>

```

Salvando e voltando ao navegador, veremos os dois contatos agrupados em um grupo chamado de `todosContatos`. Veja que agora `todosContatos` é um objeto.

Livro Angular 2

formulário

Nome

Telefone

Email

Clicou fora do campo

[object Object]

```
{ "telefone": "(99)99999-9999", "email": "email@email.com" }
```

Figura 6.4: Mostrando o objeto interno feito no formulário

6.4 ENVIANDO DADOS DO FORMULÁRIO

Nosso formulário já está validando os dados e mandando mensagens de erro. Agora está faltando enviar as informações quando tudo estiver correto.

Enviaremos as informações usando o `(ngSubmit)=" "` do Angular 2. Ele será adicionado dentro da tag `form` e, dentro desse *event binding* de submissão, vamos referenciar um método que criaremos na classe do componente, o qual nomearemos de `enviarDados()`.

```

enviarDados() {
  alert(`seu nome é: ${this.contato.nome}`);
  alert(`seu telefone é: ${this.contato.telefone}`);
  alert(`seu email é: ${this.contato.email}`);
}

```

Agora na tag `form` dentro do evento `ngSubmit`, vamos referenciar esse novo método da classe do componente.

```
<h2>formulário</h2>
```

```

<form (ngSubmit)="enviarDados()" #cadastro="ngForm">
  <div>
    <label for="nome">Nome</label>

```

```
/* o resto do código é o mesmo */
```

Salvando e voltando ao navegador, quando clicamos no botão, enviaremos três alertas: o primeiro para o nome, o segundo com o telefone e o terceiro com o e-mail. Mas veja que, mesmo com as validações sendo mostradas, o botão fica habilitado e conseguimos enviar as informações mesmo estando fora dos padrões estabelecidos. Vamos arrumar isso agora.

Colocamos no formulário uma variável de template que está recebendo o valor de `ngForm`. Com isso, falamos para o Angular 2 que nossa variável de template terá o mesmo comportamento que uma diretiva de `ngForm`.

Agora podemos recuperar o estado atual do formulário e verificar se ele está válido ou inválido. Assim, podemos habilitar ou desabilitar o botão de enviar do formulário. Veja como ficará no botão de enviar.

```

<button type="submit" [disabled]="cadastro.form.invalid">Enviar</button>

```

Salvando e voltando ao navegador, se retirarmos o `nome` ou o

telefone , o formulário passa do *estado válido* para o *estado inválido*, e o botão de enviar os dados ficará desabilitado.

Nosso arquivo final `formulario.component.html` ficou assim:

```
<h2>formulário</h2>

<form (ngSubmit)="enviarDados()" #cadastro="ngForm">
  <div>
    <label for="nome">Nome</label>
    <input type="text" id="nome" #nome="ngModel" [(ngModel)]="
contato.nome" name="nome" required>

    <div [hidden]="nome.valid || nome.pristine">
      <p>O valor nome é obrigatório</p>
    </div>
  </div>

  <br>
  <div ngModelGroup="todosContatos" #todosContatos="ngModelGrou
p">
    <div>
      <label for="telefone">Telefone</label>
      <input type="text" id="telefone" [(ngModel)]="contato
.telefone" #telefone="ngModel" name="telefone" required>

      <div *ngIf="telefone.invalid && telefone.dirty">
        <p>O número do Telefone é obrigatório</p>
      </div>
    </div>

    <br>

    <div>
      <label for="email">Email</label>
      <input type="text" id="email" [(ngModel)]="contato.em
ail" #email="ngModel" name="email">

      <div [hidden]="email.pristine">
        <p>O conteúdo foi mudado</p>
      </div>
      <div *ngIf="email.touched">
```

```

        <p>Clicou fora do campo</p>
      </div>
    </div>
  </div>

  <br>

  <button type="submit" [disabled]="cadastro.form.invalid">Envi
ar</button>
</form>

<p>{{todosContatos}}</p>
<p>{{todosContatos.value | json}}</p>

```

Nosso arquivo final `formulario.component.ts` ficou assim:

```

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

@Component({
  selector: 'app-formulario',
  templateUrl: './formulario.component.html',
  styleUrls: ['./formulario.component.css']
})
export class FormularioComponent implements OnInit {

  contato = new Contatos('Thiago', '(99)99999-9999', 'email@email.com');

  constructor() { }

  ngOnInit() {
  }

  enviarDados() {
    alert(`seu nome é: ${this.contato.nome}`);
    alert(`seu telefone é: ${this.contato.telefone}`);
    alert(`seu email é: ${this.contato.email}`);
  }
}

```

Nosso arquivo final `Contatos.ts` ficou assim:

```

export class Contatos {

```

```

    constructor(public nome: string,
                 public telefone: string,
                 public email: string){
    }
}

```

6.5 RESUMO

Neste capítulo, aprendemos como criar formulários no Angular 2, e as facilidades e procedimentos para que, no final, tudo funcione como o esperado. Vimos todos os procedimentos passo a passo, desde a criação da estrutura HTML, adição das variáveis de template, comunicação do template com a classe do componente, a utilização das classes de estado de cada componente dentro do formulário até o envio dos dados através do método `ngSubmit()`.

Para resumir, temos de:

- Criar a estrutura HTML.

- Adicionar as variáveis de template em cada elemento dentro do formulário, inclusive na tag `form`.

- Adicionar o atributo `name` de cada elemento dentro do formulário.

- Adicionar o data binding `[(ngModel)]` com referência ao objeto modelo que está na classe do componente.

- Efetuar as validações de acordo com a regra de negócio, usando as classes de estado atual do componente do formulário.

Validar o botão de envio de acordo com o formulário, se válido ou não.

Enviar os dados do formulário através do `ngSubmit` .

INJEÇÃO DE DEPENDÊNCIAS

Até agora, ao longo de todos os exemplos, os dados que usamos sempre estavam dentro do componente de forma estática, e sabemos que, em uma aplicação real, não será assim. Sempre teremos os dados que virão de serviços, sejam eles online ou de banco de dados próprio da aplicação. Mas o foco aqui é que os dados nunca estarão dentro do projeto.

Outro ponto que não falamos é sobre as regras de negócio de uma aplicação. Nos nossos exemplos, não nos preocupamos com regras de negócio e como serão formatados os dados, se são válidos, ou até para onde vamos enviá-los.

Todas as respostas para as questões levantadas agora veremos neste capítulo sobre injeção de dependência: o que é, para que serve, como e quando utilizar.

7.1 O QUE É INJEÇÃO DE DEPENDÊNCIA

Injeção de dependência, também conhecida como *DI* (*dependency injection*), é um padrão de codificação no qual uma classe recebe suas dependências de uma fonte externa em vez de

criá-las manualmente. Ou seja, injeção de dependência é passar uma classe de serviço para que uma classe componente possa usar. Assim, para o componente funcionar corretamente, ele necessita dessa classe de serviço.

Podemos criar instâncias de uma classe de serviço dentro dos componentes de duas formas bem conhecidas, que são: **instanciando a classe manualmente**, ou **injetando a dependência no método construtor da classe do componente**.

7.2 INSTANCIANDO SERVIÇOS MANUALMENTE

Para este capítulo, vamos criar um novo componente chamado de `di`. Então, faremos todo nosso velho processo de criação do componente: adicionar um título que receberá o nome de Injeção de dependência, comentar tag anterior e adicionar a nova tag no arquivo `app.component.html`.

No nosso novo componente `di.component`, vamos ao arquivo de template para criarmos uma lista não ordenada com a tag `ul`. Dentro, colocamos a tag `li` para mostrar o conteúdo da lista, e um `*ngFor` dentro da tag `li`, para listar nomes de tecnologias que virão de um serviço.

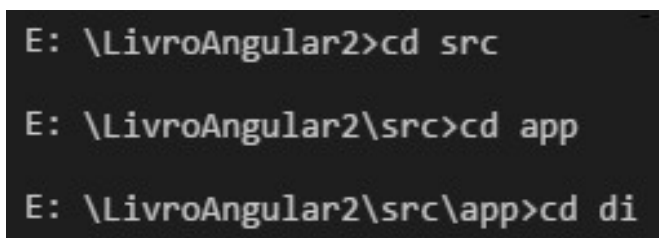
```
<h2>Injeção de dependência</h2>

<ul>
  <li *ngFor="let item of tecnologias">{{item}}</li>
</ul>
```

Agora vamos criar dentro da pasta do componente duas classes de serviço. A primeira terá o nome de `nome-tec.service` e terá

um método com o nome `getNomesTec()` . Esse método vai retornar um array com nomes de tecnologias, e logo em seguida criaremos uma outra classe de serviço com o nome de `meu-log.service` . Ela terá um método `setLog(msg: string)` , que vai receber uma `string` e enviar para o console do navegador.

Antes de criar os dois serviços, devemos entrar na pasta do componente digitando no console do VS Code o comando: `cd src\app\di` , e apertamos `Enter` .



```
E: \LivroAngular2>cd src
E: \LivroAngular2\src>cd app
E: \LivroAngular2\src\app>cd di
```

Figura 7.1: Estrutura do caminho para pasta do componente

Agora que estamos na pasta do componente, criaremos as classes de serviços. Digitamos `ng g s nomes-tec` para criar a classe de serviço que terá o array com os nomes das tecnologias. Após isso, digitamos `ng g s meu-log` para criar outra classe de serviço, que será responsável por enviar uma mensagem de log no console do navegador.

Nossa estrutura da pasta deste componente ficará assim:

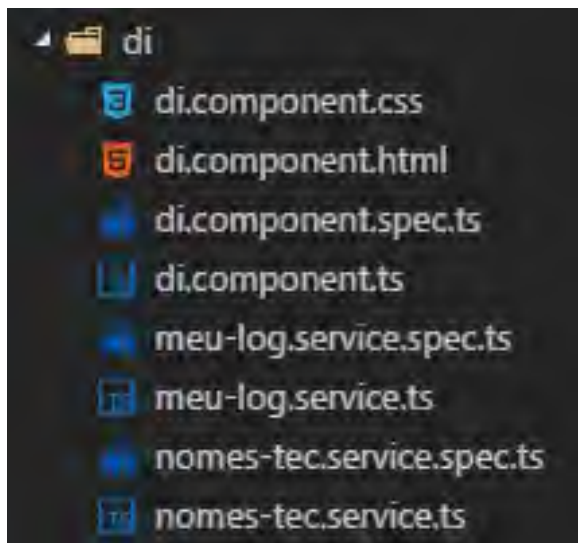


Figura 7.2: Estrutura da pasta do componente

Com a pasta conforme a figura, já podemos exercitar o uso da injeção de dependência.

No arquivo `nomes-tec.service.ts`, criaremos um método que retornará um array do tipo `string` com os nomes de algumas tecnologias.

```
getNomesTec(): string [] {  
    this.meulog.setLog('consultou o array de tecnologias');  
    return ['Angular 2', 'TypeScript', 'JavaScript', 'HTML5', 'CS  
S3', 'Desenvolvendo com Angular 2'];  
}
```

No arquivo `meu-log.service.ts`, criaremos um método que enviará ao console do navegador uma mensagem que será recebida por parâmetro.

```
setLog(msg:string){  
    console.log(msg);
```

```
}
```

Nossa ideia nesse exercício é: chamar a classe de serviço na classe do componente para recuperar os nomes do array. Quando o método da classe de serviço for chamado, ele automaticamente chamará o método da outra classe de serviço (serviço de log), que envia uma mensagem de log no navegador.

Primeiro, vamos fazer as instâncias das classes de forma manual para verificar como seria sem a ajuda da injeção de dependência do Angular 2. No nosso arquivo `nomes-tec.service.ts`, vamos ao método construtor e declaramos que essa classe precisa de uma instância da `MeuLogService` para funcionar corretamente. Também declararemos uma variável com o nome `meuLog` com o tipo `MeuLogService`.

```
export class NomesTecService {  
  
  meuLog: MeuLogService;  
  
  constructor(meulog: MeuLogService) {  
    this.meuLog = meulog;  
  }  
}
```

Veja que, dessa forma, todas as classes que forem usá-la terão de enviar um objeto do tipo `MeuLogService`.

Agora vamos para a nossa classe do componente `di.component.ts`, para criar três variáveis com os nomes: `tecnologias`, que será um array do tipo `string`; outra com o nome de `meuService`, que será do tipo `NomesTecService`; e a última com o nome `meuLog` do tipo `MeuLogService`. Não devemos esquecer de sempre importar as classes para dentro da classe do componente.

No método construtor, instanciamos manualmente as duas classes e, em seguida, usamos o método `getNomesTec` da classe de serviço. Seu retorno será atribuído no array de `tecnologias`. Para finalizar, vamos declarar as duas classes de serviços dentro do decorador do componente com o metadata `providers: []`.

Então, com tudo configurado, nossa classe ficará assim:

```
import { Component, OnInit } from '@angular/core';
import { NomesTecService } from './nomes-tec.service';
import { MeuLogService } from './meu-log.service';

@Component({
  selector: 'app-di',
  templateUrl: './di.component.html',
  styleUrls: ['./di.component.css'],
  providers: [NomesTecService, MeuLogService]
})
export class DiComponent implements OnInit {

  tecnologias: string [] = [];
  meuService: NomesTecService;
  meuLog: MeuLogService;

  constructor() {
    this.meuLog = new MeuLogService;
    this.meuService = new NomesTecService(this.meuLog);
    this.tecnologias = this.meuService.getNomesTec();
  }
}
```

Salvando e voltando ao navegador, veremos a lista de nomes que está vindo da classe serviço. No console do navegador, está sendo mostrada a mensagem de log.

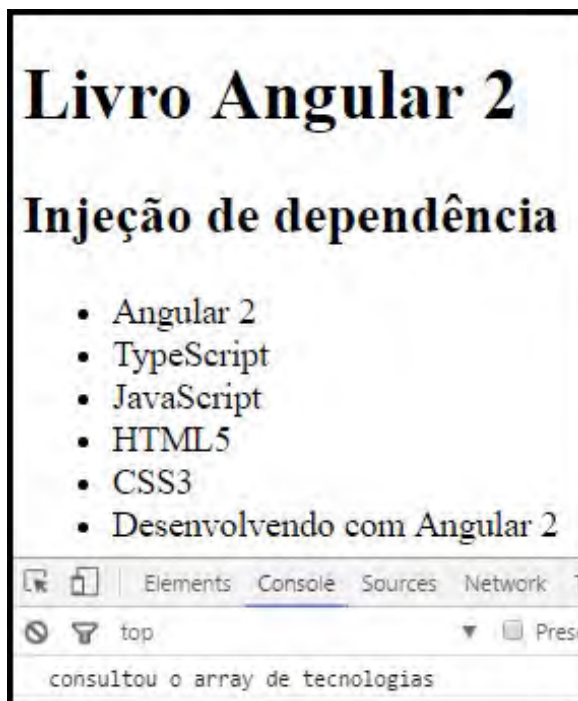


Figura 7.3: Lista sendo mostrada junto com o log no navegador

Embora nosso sistema esteja funcionando, temos um problema muito grande de acoplamento de classe, que nada mais é quando uma classe está explicitamente sendo instanciada dentro de outra. Isso prejudica a manutenção e evolução do software.

Imagine que, no futuro, nossa classe de serviço `nomes-tec.service` precise de uma outra classe (fora a classe de `meu-log.service`) para funcionar, ou se agora queremos criar uma classe que contabilize a quantidade de acessos feitos na classe `nomes-tec.service`. Deveríamos implementar essa nova classe dentro do serviço `nomes-tec.service` e, quando fizemos isso, o nosso código terá vários erros de compilação. A aplicação parará

de funcionar até que todas as instâncias tenham sido implementadas e corrigidas.

Agora, e se nossa aplicação tiver centenas de instâncias dentro de várias classes? Já imaginou a dor de cabeça e o tempo de manutenção que isso vai dar?

Para melhorar as dependências de classes, e facilitar na manutenção ou nas implementações futuras, o Angular 2 recomenda fortemente o uso de injeção de dependências.

7.3 INJETANDO UMA DEPENDÊNCIA

A injeção de dependência serve para facilitar o desenvolvimento e, futuramente, as manutenções, evitar erros e melhorar os testes. Usando a injeção de dependência, podemos desenvolver aplicações muito mais consistentes e com muito menos problemas de desenvolvimento e manutenção.

Para transformar nosso exemplo atual em uma aplicação com injeção de dependências, devemos fazer algumas mudanças nos métodos construtores de cada classe. Então, vamos começar mudando a classe de serviço `nomes-tec.service`.

Primeiro, vamos retirar a variável `meuLog` e mudar o bloco do método construtor, pois atualmente ele espera receber um objeto do tipo `MeuLogService` e, em seguida, ele atribui o objeto na variável `meuLog`. Faremos com que a própria classe `nomes-tec.service` já tenha a classe de log disponível e sem necessidade de receber um objeto no construtor.

Para isso, vamos fazer uma *injeção de dependência* de forma

que a classe de log seja injetada automaticamente sempre que a `nomes-tec.service` for chamada. Devemos passar a dependência no método construtor.

```
export class NomesTecService {  
    constructor(private meulog: MeuLogService) {  
    }  
}
```

Veja que retiramos a variável `meuLog` e a linha dentro do método construtor que fazia a atribuição para ela. O que estamos fazendo agora é pedindo para o Angular 2 gerenciar todas as dependências desta classe de modo automático. Isto é, eu não preciso me preocupar se o objeto será passado ou não, pois quem vai gerenciar isso será o próprio Angular 2.

Dessa forma, para usar a nova variável `meulog`, precisamos somente referenciar dentro do método `getNomesTec` no lugar da antiga variável.

```
getNomesTec(): string [] {  
    this.meulog.setLog('consultou o array de tecnologias');  
    return ['Angular 2', 'TypeScript', 'JavaScript', 'HTML5', 'CS  
S3', 'Desenvolvendo com Angular 2'];  
}
```

Essa simples mudança já faz com que nossa classe de serviço use a injeção de dependência do Angular 2. Vamos agora corrigir a classe do componente `di-component.ts`, que também terá algumas mudanças.

Primeiro, retiraremos as variáveis `meuService` e `meuLog`, pois agora elas serão injetadas na classe pelo Angular 2. Na declaração do método construtor da classe `di.component.ts`, vamos declarar quais classes queremos que o gerenciador de injeção de dependências do Angular 2 adicione.

Então, chegamos em um ponto superimportante! Como estamos usando injeção de dependência, não precisamos declarar tudo que uma classe secundária precisa para funcionar.

No caso anterior, tínhamos instanciado a classe de log somente porque a classe de serviço necessitava de uma instância dela para funcionar. Mas agora, como estamos deixando o Angular 2 gerenciar as dependências de classes, somente precisamos declarar qual classe realmente necessitamos dentro do componente.

Se essa classe precisa de outra para funcionar, não será mais nosso problema, e sim problema do *gerenciador de injeção de dependência* do Angular 2. Dessa forma, nosso método construtor da classe `di.component.ts` somente precisa da injeção da classe de serviço `nomes-tec.service`. Então, nosso novo método construtor ficará assim:

```
constructor(private meuService: NomesTecService) {  
    this.tecnologias = meuService.getNomesTec();  
}
```

Veja que, com a injeção de dependência, nós não precisamos nos preocupar em enviar objetos dentro de novas instâncias. Se no futuro o nosso serviço `nomes-tec.service` precisar de uma nova classe, não vamos mexer em nenhum lugar, pois neste momento nosso componente somente tem dependência dessa classe de serviço para recuperar os nomes das tecnologias. Agora, o gerenciador de injeção de dependência que vai verificar o que é preciso no `nomes-tec.service`.

Assim, fica muito mais fácil para desenvolver uma aplicação, pois o gerenciador de dependência cria um container e adiciona nele todas as dependências que estão sendo usadas. E se alguma

outra classe precisar dessa instância, o gerenciador somente cria uma referência para a nova classe e reutiliza a instância atual.

7.4 DEPENDÊNCIAS OPCIONAIS

Até aqui, vimos primeiramente como usar as classes de forma manual, instanciando cada classe dentro do componente. Após isso, aprendemos como usar a injeção de dependência para facilitar o nosso desenvolvimento.

Mas tanto com instâncias manuais ou com injeção de dependência, sempre devemos ter certeza de que a classe que será adicionada no nosso componente, ou se o componente, pode funcionar sem essa injeção. Seja instanciando a classe de log manualmente ou injetando dentro da classe do componente, mesmo assim essa classe necessariamente deve existir para o projeto funcionar.

Mas e se, por algum motivo, a classe de log não estiver no projeto? Será que o projeto vai parar de funcionar? Para isso, temos as dependências opcionais.

As dependências opcionais vão servir para informar ao Angular 2 que determinada injeção de classe pode ser opcional. Ou seja, caso ela exista, o gerenciador pode injetar a classe dentro do componente; mas caso não exista, ele não adicionará nenhuma classe e o sistema vai continuar funcionando sem nenhum problema.

Para informar ao Angular 2 que uma injeção de dependência de uma classe pode ser opcional, devemos adicionar um decorador `@Optional()` junto com a injeção no construtor do método da

classe que está recebendo a injeção. Com o uso do decorador `@Optional()` dentro do nosso código, dizemos ao Angular 2 que essa injeção é opcional para o funcionamento da classe principal.

Mas além de adicionar o decorador junto com a declaração da classe (que será injetada no método construtor da classe principal), devemos efetuar uma validação para verificar se realmente a injeção foi feita. Se foi, devemos efetuar os procedimentos com a dependência recebida. Entretanto, se na validação dentro do método que usa a injeção der falso, devemos simplesmente ignorar todos os procedimentos que seriam feitos com essa dependência.

Para deixarmos nossa classe de serviço `nomes-tec.component.ts` independente da necessidade da classe de serviço de log, vamos adicionar na frente uma injeção da classe de log, o decorador `@Optional()`, conforme segue no exemplo.

```
constructor(@Optional() private meulog: MeuLogService) {  
}
```

Dessa forma, dizemos ao Angular 2 que, para a classe `nomes-tec.component.ts` funcionar corretamente, temos a opção de receber uma dependência da classe de serviço de log.

Para finalizar, no método `getNomesTec()`, devemos verificar se a injeção da classe de serviço de log foi injetada com sucesso dentro da nossa classe. Para isso, vamos adicionar um `if` e fazer a verificação.

```
getNomesTec(): string [] {  
  
    if(this.meulog) {  
        this.meulog.setLog('consultou o array de tecnologias');  
    }  
  
    return ['Angular 2', 'TypeScript', 'JavaScript', 'HTML5', 'CS
```

```
S3', 'Desenvolvendo com Angular 2'];  
}
```

Dessa forma, estamos verificando se existe a instância da classe de log. Se existir, nós vamos criar o log no navegador; caso não exista, a aplicação funcionará corretamente, sem problemas.

7.5 DECLARAÇÃO GLOBAL NO PROJETO OU DENTRO DO COMPONENTE

Já tivemos uma visão geral de como funciona uma aplicação desenvolvida com o Angular 2. Falamos também que podemos configurar uma classe de serviço na parte de serviços, seja de forma global no projeto, ou particular dentro de um componente. Vimos também quais as diferenças entre global e particular. Agora vamos um pouco mais a fundo neste tema.

Em uma aplicação com Angular 2, temos duas árvores hierárquicas: uma **de componente** e outra **de injetores**. A árvore hierárquica de componente serve para mostrar qual o componente é o pai e quais os componentes são os filhos. Em uma árvore de componentes, sabemos qual é o componente raiz e quais os componentes serão dependentes dele.

Todos os componentes filhos vão herdar os serviços do componente pai. Isto é, se configurarmos uma classe de serviço no componente pai, e esse componente pai tiver outros dois componentes filhos, os três componentes terão acesso ao serviço configurado. Mas isso ainda não torna este serviço como global na aplicação.

Como falamos no capítulo *Arquitetura do sistema e*

componentes do Angular 2, em uma aplicação Angular 2 temos um container onde ficam as instâncias dos serviços que estão sendo usados pela aplicação. A qualquer momento, algum componente pode precisar de uma instância de serviço para continuar com seu processo. Uma questão aqui é: onde configurar as classes de serviços?

Como na aplicação do Angular 2, temos duas árvores independentes, uma de injetores e outra de componentes, então podemos a qualquer momento usar um serviço dentro do componente, desde que este tenha acesso. Se o injetor do serviço foi feito na raiz da aplicação (`app.module.ts`), todos os componentes terão acesso a este serviço a qualquer momento.

Se o injetor do serviço foi feito no componente pai, somente ele e seus filhos terão acesso a este serviço configurado. Mas caso o injetor do serviço tenha sido feito dentro de um único componente, somente este terá acesso a ele.

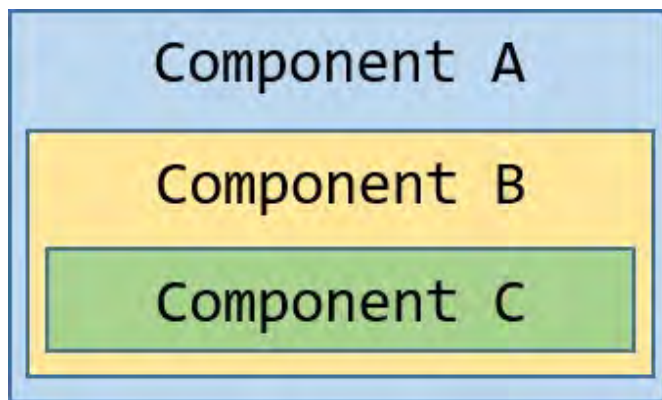


Figura 7.4: Hierarquia de componentes dentro da aplicação Angular 2

Da mesma forma que temos um serviço no componente pai e

este é disponibilizado para os componentes filhos, também podemos fazer uma especialização do serviço entre cada componente. Vamos ver a situação de uma montadora de carros, e verificar como podemos especializar e herdar cada serviço, de acordo com o nível do componente.

Nesta montadora, vamos fabricar três carros, e eles terão uma parte padrão e outra parte específica de cada carro.

Primeiro, teremos o carro A , que terá o serviço de modelo , motor e pneus . Segundo, teremos o carro B , que terá outro modelo e outro motor , mas o mesmo serviço pneus do carro A . Terceiro, teremos o carro C . Este terá outro modelo , mas com o mesmo serviço de motor do carro B e mesmo serviço de pneus do carro A .

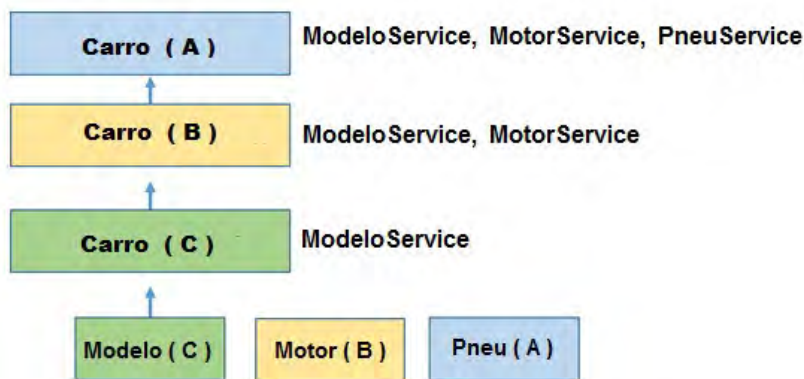


Figura 7.5: Hierarquia de componentes dentro da aplicação Angular 2

Veja que, declarando um serviço dentro dos componentes, cada componente terá sua própria instância desse serviço. Ou caso este componente tenha componentes filhos, todos os filhos terão

acesso ao serviço.

A diferença aqui é que todos os serviços de `modelo` , `motor` e `pneus` só podem ser usados entre os componentes que têm estes serviços configurados. Mas se esses serviços forem configurados dentro da raiz da aplicação, toda a aplicação terá acesso. Quanto disso faz sentido?

Se dentro da nossa aplicação tivermos um componente de fabricação de barcos, ele poderia também usar os serviços de `modelo` e `motor` , pois estariam configurados na raiz da aplicação. Porém, ele também teria acesso ao serviço de `pneus` se também fosse configurado na raiz da aplicação, e isso não faria sentido, pois um barco não tem pneus.

Para uma aplicação desse tipo, poderíamos configurar os serviços de `modelo` e `motor` de forma global e, para cada componente, especificar esses dois serviços de forma que melhor atenda, e configurar o serviço de `pneus` somente para os componentes de fabricação de carro.

No nosso projeto, temos um serviço que está configurado de forma global, o arquivo `alerta.service` . Este pode ser usado por qualquer componente dentro da aplicação, pois, em algum momento, qualquer componente poderá enviar um alerta para o navegador do usuário.

Vamos ao nosso arquivo `di.component.ts` e importaremos o serviço de alerta para dentro deste componente e, logo em seguida, executaremos o método `msgAlerta()` . Veja que tudo vai funcionar corretamente, pois o serviço de alerta está configurado de forma global no projeto.

```

constructor(private meuService: NomesTecService, private meuAlert
a: AlertaService) {
    this.tecnologias = meuService.getNomesTec();
    this.meuAlerta.msgAlerta();
}

```

Salvando e voltando ao navegador, veremos a mesma mensagem.

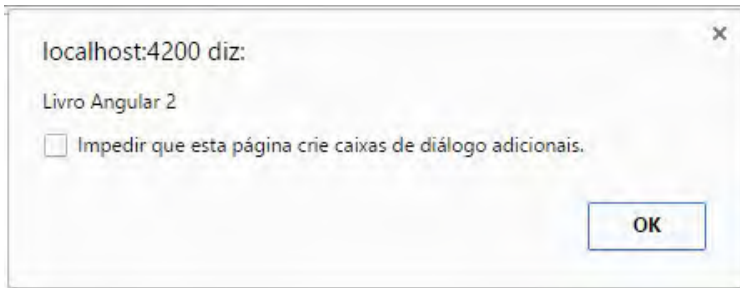


Figura 7.6: Alerta sendo mostrado pelo componente di.component

Porém, e se quisermos usar o serviço `nome-tec.service` em outro componente dentro do projeto? Isso não vai funcionar, porque este serviço semente está configurado dentro do componente `di.component`.

Nosso arquivo final `di.component.html` ficou assim:

```

<h2>Injeção de dependência</h2>

<ul>
  <li *ngFor="let item of tecnologias">{{item}}</li>
</ul>
<p>{{item}}</p>

```

Nosso arquivo final `di.component.ts` ficou assim:

```

import { Component, OnInit } from '@angular/core';
import { NomesTecService } from '../nomes-tec.service';
import { MeuLogService } from '../meu-log.service';

```

```

import { AlertaService } from '../alerta.service';

@Component({
  selector: 'app-di',
  templateUrl: './di.component.html',
  styleUrls: ['./di.component.css'],
  providers: [NomesTecService, MeuLogService]
})
export class DiComponent implements OnInit {

  tecnologias: string [] = [];
  //meuService: NomesTecService;
  //meuLog: MeuLogService;

  /*constructor() {
    this.meuLog = new MeuLogService;
    this.meuService = new NomesTecService(this.meuLog);
    this.tecnologias = this.meuService.getNomesTec();
  }*/

  constructor(private meuService: NomesTecService, private meuAlerta: AlertaService) {
    this.tecnologias = meuService.getNomesTec();
    this.meuAlerta.msgAlerta();
  }

  ngOnInit() {
  }
}

```

Nosso arquivo final nomes-tec.servi.ts ficou assim:

```

import { Injectable, Optional } from '@angular/core';
import { MeuLogService } from '../meu-log.service';

@Injectable()
export class NomesTecService {

  //meuLog: MeuLogService;

  constructor(@Optional() private meulog: MeuLogService) {
    //this.meuLog = meulog;
  }
}

```

```

getNomesTec(): string [] {

    if(this.meulog) {
        this.meulog.setLog('consultou o array de tecnologias');
    }

    return ['Angular 2', 'TypeScript', 'JavaScript', 'HTML5', 'CSS3', 'Desenvolvendo com Angular 2'];
}
}

```

Nosso arquivo final `meu-log.service.ts` ficou assim:

```

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

@Injectable()
export class MeuLogService {

    constructor() { }

    setLog(msg:string){
        console.log(msg);
    }
}

```

7.6 RESUMO

Usamos classes de serviços sempre que queremos executar alguma rotina que não seja específica de algum componente. Para isso, devemos usar uma classe que terá o decorador `@injectable()` e descrever dentro dela os métodos necessários.

Para utilizar a classe de serviço, devemos importar para dentro da classe do componente, usando o `import`. Após importar, é preciso informar ao componente que essa classe é um serviço e, para isso, vamos declarar a classe de serviço dentro do metadata `provider: []`, informando o nome da classe do serviço. Após a declaração, podemos injetá-la dentro do método construtor,

passando as informações da classe de serviço para uma variável interna da classe do componente.

Dentro do construtor, poderemos usar todos os métodos da classe de serviço que foram instanciados, e os métodos retornarão os dados para serem usados dentro dos componentes e mostrados no navegador. Se o serviço foi configurado na raiz da aplicação, todo o projeto terá acesso. Mas se foi configurado dentro de um componente, somente este e seus filhos terão acesso a ele.

PROJETO FINAL

Enfim, estamos quase no fim do livro. Espero que tenha gostado e aprendido bastante sobre Angular 2. Agora vamos pôr em prática tudo o que aprendemos até aqui.

Como foi dito no primeiro capítulo, nosso projeto final será uma lista de contatos. Vamos tentar usar tudo o que aprendemos durante o livro dentro desse projeto. Ele ficará dessa forma:

Projeto de Contatos com Angular 2

Dados do contato

Nome
Angular 2 ✓

Telefone
(99)99999-9999 ✓

Email
email@email.com

Tipo do Contato
Particular
Trabalho
Amigos
Familia

Cadastrar

Contatos Cadastrados

#	Nome	Tipo
---	------	------

Detalhe do Contato

Nome

Telefone

Email

Tipo

Figura 8.1: Foto geral do projeto — 1



Figura 8.2: Foto geral do projeto — 2

Ele terá três componentes: `dados-usuario.component` , `detalhe-usuario.component` e `lista-usuario.component` . Teremos duas pastas para melhor organizar nosso código (organização é muito importante): `modelos` , que terá a classe modelo do contato contendo todos os campos necessários para preenchimento; e uma pasta `servicos` , que será nossa classe para armazenamento dos contatos.

Como não vamos usar banco de dados, vamos gravar nossos contatos em um array dentro de uma classe de serviço. Vamos sempre focar no desenvolvimento com o Angular 2.

Para o projeto ficar com um visual melhorado, teremos também de usar o **Bootstrap 4**. Assim, veremos como instalar e importar no nosso projeto.

Tanto os códigos de todos os exemplos já feitos no livro como

também o código desse projeto final estão lá no meu GitHub:

GitHub: <https://github.com/thiagoguedes99>

Projeto com os exemplos:
<https://github.com/thiagoguedes99/livro-Angular-2>

Projeto final:
<https://github.com/thiagoguedes99/projeto-final-livro-angular-2>

Então, chega de papo e vamos trabalhar!

8.1 CRIAÇÃO DO NOVO PROJETO

Criaremos tudo do zero, o que será bom para relembrar conceitos dos capítulos anteriores. O primeiro passo é criar uma pasta para armazenar o nosso projeto final. Em seguida, no console do computador, vamos dentro desta pasta, onde digitaremos `ng new projeto-final`.

Após criado o projeto, automaticamente serão instaladas todas as dependências para ele funcionar. Basta digitar `ng serve` para rodar o servidor do Angular 2; após isso, abra o navegador e digite `http://localhost:4200`. Pronto! Novo projeto funcionando.



Figura 8.3: Projeto final rodando

Vamos abrir o projeto no **VS Code**. Clique no botão azul do lado esquerdo, e vamos até a pasta do projeto para abri-lo.

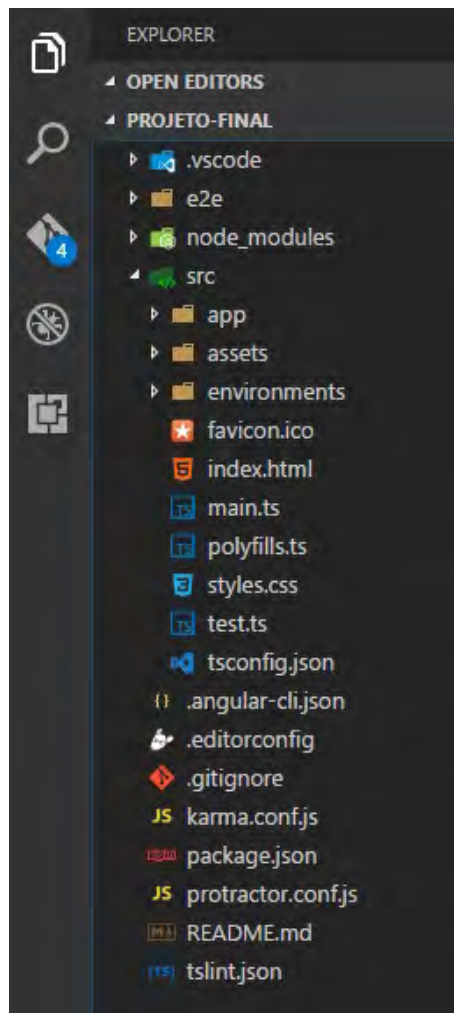


Figura 8.4: IDE com projeto

Como já mencionamos, os nomes dos componentes serão `dados-usuario`, `detalhe-usuario` e `lista-usuario`, e os nomes das pastas serão: `modelos`, que terá um arquivo de serviço com o nome de `contato-model`; e `servicos`, que terá um

arquivo de serviço com o nome `contatos-data-base` .

Nossa estrutura do projeto agora está assim:

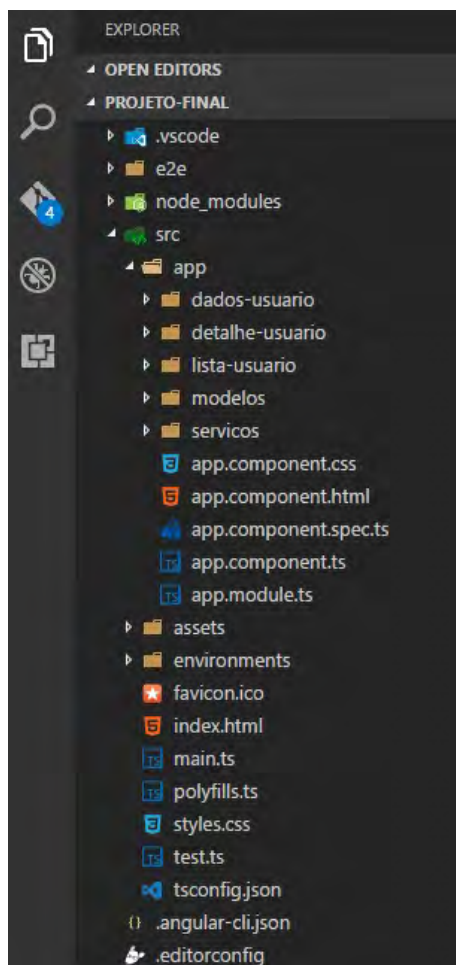


Figura 8.5: Estrutura do projeto

8.2 INSTALAÇÃO DO BOOTSTRAP

Vamos adicionar o *bootstrap* no projeto. Primeiro, digitamos no console `npm install bootstrap@next --save`. Será feito o download do *bootstrap*, e ele será adicionado dentro do nosso projeto.

Agora, vamos importá-lo. Abrimos o arquivo `styles.css`, que é o CSS global da aplicação (lembra disso?). Vamos digitar dentro `dele: @import` `'~bootstrap/dist/css/bootstrap.min.css';`, salvando todos os arquivos do projeto. O *bootstrap* já está pronto para uso. Fácil, né?

Muito fácil. E isso não é só com o *bootstrap*: esse procedimento serve para todas as bibliotecas ou dependências que você queira usar no projeto. Será sempre esse comando de `npm install` e o comando de cada serviço que você queira.

Com tudo funcionando, vamos para a arquitetura.

8.3 ARQUITETURA DO PROJETO

Nosso projeto terá os componentes independentes. O primeiro será um formulário para cadastro do contato, o segundo será para listagem dos contatos gravados, e o terceiro mostrará os detalhes do contato que foi clicado na lista.

Nesse projeto, usaremos diretivas, serviços, formulários, comunicação de componentes pai e filho, injeção de dependência e outros tópicos abordados no livro. O funcionamento do sistema será iniciado com um componente de cadastro, `dados-usuario.component`. Nele faremos um novo contato colocando os dados no formulário.

Quando o formulário estiver válido e clicarmos no botão de cadastrar, os dados serão enviados para um serviço `contatos-data-base.service.ts`. No nosso `data-base`, quando adicionado um novo contato, será emitido um evento com o `EventEmitter()`, notificando que existe um novo contato na lista.

O segundo componente `lista-usuario.component` ficará observando esse evento e, quando ele for emitido pelo `data-base`, ele atualizará a lista de contatos com o novo. Caso queremos ver os detalhes de algum contato da lista, vamos clicar nele. Assim, um evento `(click)=" "` será executado, enviando com o `@Output` e com o `id` do contato clicado.

O componente pai `app.component` receberá o `id` de contato clicado, e fará uma consulta no `data-base` recuperando o contato. Após estar com os dados do contato, o componente pai enviará para outro componente filho o `detalhe-usuario.component`, que ficará responsável por mostrar todos os detalhes dele.

Essa será a arquitetura e o funcionamento do projeto. A figura a seguir vai lhe ajudar a entender melhor todo o processo.

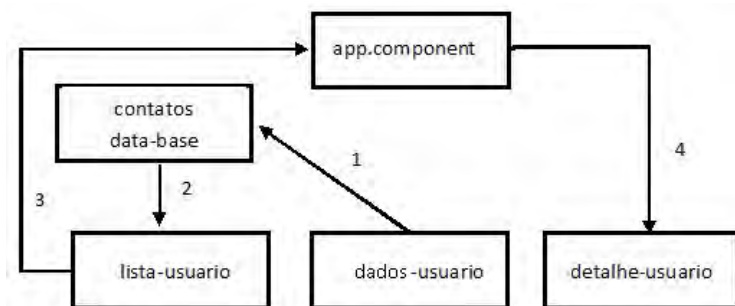


Figura 8.6: Fluxo do processo

8.4 CLASSE DE MODELO PARA OS CONTATOS

Nossa classe modelo será usada em diversos locais para envio das informações do contato. Ela servirá como uma estrutura padrão para todos os contatos e terá os campos de nome , telefone , email e tipo . Mais à frente, explico sobre o tipo de contato.

Nossa classe `contato-model.ts` ficou assim:

```
export class ContatoModel {
  constructor(public nome: string, public telefone: string, public email: string, public tipo: string) {}
}
```

Veja que é uma classe simples e bem parecida com as que já usamos em outros capítulos. Vamos criar esse arquivo dentro da pasta `modelos` , assim já vamos organizando nosso código.

8.5 CLASSE DE SERVIÇO E DATA-BASE

Nossa classe `contatos-data-base.service.ts` será criada dentro da pasta `servicos`, e servirá como um banco de dados da aplicação. Nela vamos armazenar e recuperar as informações do contato que queremos.

Vamos criar um array com o nome `meuContatos` do tipo `ContatoModel`, no qual ficarão todos os contatos. Criaremos uma variável com o nome `enviarContato` do tipo `EventEmitter()`, que emitirá quando um contato foi adicionado na lista.

Teremos dois métodos: `setContato(novoContato: ContatoModel)`, que vai gravar um novo contato no array; e `getContato(id: number)`, que vai receber um número e recuperar os dados do contato pelo `id`.

Nossa classe `contatos-data-base.service.ts` ficará assim:

```
import { EventEmitter, Injectable } from '@angular/core';
import { ContatoModel } from '../modelos/contato-model';

@Injectable()
export class ContatosDataBaseService {

  meuContatos: ContatoModel [] = [];
  enviarContato = new EventEmitter();

  constructor() { }

  setContato(novoContato: ContatoModel): void {
    this.meuContatos.push(novoContato);

    this.enviarContato.emit(this.meuContatos);
  }

  getContato(id: number): ContatoModel {

    let contato: ContatoModel;
```

```

    contato = this.meuContatos[id];

    return contato;
}
}

```

8.6 COMPONENTE DE CADASTRO DE CONTATO

Mostraremos a construção de cada componente e, no final, toda a integração entre eles. Vamos começar com o componente de cadastro. No contato, teremos quatro campos: nome , telefone , email e tipo , sendo somente os campos de nome e telefone obrigatórios. O campo de tipo será preenchido com o tipo do contato, podendo escolher entre particular , trabalho , amigos ou família .

Quando clicarmos no botão cadastrar , os dados serão enviados para o data-base e, em seguida, o formulário será escondido. No seu lugar, aparecerá uma tela de confirmação com todos os dados que foram cadastrados.

Falei o que terá no formulário e nos campos obrigatórios para você já ir imaginando como será criada sua estrutura. Podemos consultar a estrutura do formulário que fizemos no capítulo de formulário, pois este será parecido.

Junto com a estrutura tradicional de HTML e marcações do Angular 2, teremos as classes do *bootstrap*. Isso pode confundir no começo, mas será fácil separar o que é HTML de formulário, marcação do Angular 2 e classe do bootstrap.

Do bootstrap, vamos usar as classes container , form-

group , form-control , btn , card e card-block . Tudo o que estiver dentro de class=" " será dele.

Vamos também customizar a classe card para trocar a cor do *background*. Já passei todas as informações para você construir a estrutura do formulário, agora vou lhe mostrar como ficou nosso arquivo dados-usuario.component.html .

```
<h3 class="container col-6">
  Dados do contato
</h3>
<div class="card container col-6">
  <form class="card-block" (ngSubmit)="enviarDados()" *ngIf="!
enviado" #formulario="ngForm">
    <div class="form-group" [class.has-success]="nome.valid"
class.has-danger]="nome.invalid">
      <label for="nome">Nome</label>
      <input class="form-control" [class.form-control-succe
ss]="nome.valid" type="text" id="nome" [(ngModel)]="_nome" name="
nome" #nome="ngModel" placeholder="digite o nome" required>

      <div [hidden]="nome.valid || nome.pristine">
        <small [ngClass]="{'form-control-danger': 'nome.i
nvalid || nome.drity',
                          'form-control-feedback': 'nome.inv
alid || nome.drity'}">O nome é um campo obrigatório!</small>
      </div>
    </div>

    <div class="form-group" [class.has-success]="telefone.val
id" [class.has-danger]="telefone.invalid">
      <label for="telefone">Telefone</label>
      <input class="form-control" [class.form-control-succe
ss]="nome.valid" type="text" id="telefone" [(ngModel)]="_telefone
name="telefone" #telefone="ngModel" placeholder="digite o telefo
ne" required>

      <div [hidden]="telefone.valid || telefone.pristine">
        <small [ngClass]="{'form-control-danger': 'telefo
ne.invalid || telefone.drity',
                          'form-control-feedback': 'telefone
.invalid || telefone.drity'}">O telefone é uma campo obrigatório!
```

```

</small>
    </div>
</div>

    <div class="form-group">
        <label for="email">Email</label>
        <input class="form-control" type="text" id="email" [(
ngModel)]= "_email" name="email" #email="ngModel">
    </div>

    <div class="form-group">
        <label for="tipo">Tipo do Contato</label>
        <select id="tipo" [(ngModel)]= "_tipo" name="tipo" #ti
po="ngModel">
            <option class="form-control" *ngFor="let item of tipo
s" [value]="item">{{item}}</option>
        </select>
    </div>

    <button class="btn" type="submit" [disabled]="formulario.
invalid">Cadastrar</button>
</form>
</div>

<div class="card card-block container col-6" *ngIf="enviado">
    <div>
        <h2>Dados do contato Enviado!</h2>
    </div>

    <div>
        <p>Nome do Contato: <strong>{{_nome}}</strong></p>
    </div>

    <div>
        <p>Telefone do Contato: <strong>{{_telefone}}</strong></p>
    </div>

    <div>
        <p>Email do Contato: <strong>{{_email}}</strong></p>
    </div>

    <div>
        <p>Tipo do Contato: <strong>{{_tipo}}</strong></p>
    </div>

```

```
<button (click)="voltar()">Voltar</button>
</div>
```

Nossa classe do componente terá: as variáveis para manipulação no template junto com `[(ngModel)]`; um array que mostrará as opções dos tipo de contato; um método `enviarDados()`, que enviará os dados do contato para o data-base; e o método `voltar()` que vai mudar da tela de confirmação para a tela de formulário.

Essa será nosso arquivo `dados-usuarios.component.ts`:

```
import { Component, OnInit } from '@angular/core';
import { ContatoModel } from '../modelos/contato-model';
import { ContatosDataBaseService } from '../servicos/contatos-dat
a-base.service';

@Component({
  selector: 'app-dados-usuario',
  templateUrl: './dados-usuario.component.html',
  styleUrls: ['./dados-usuario.component.css']
})
export class DadosUsuarioComponent implements OnInit {

  enviado: boolean = false;

  _nome: string;
  _telefone: string;
  _email: string;
  _tipo: string;

  tipos: string [] = ['Particular', 'Trabalho', 'Amigos', 'Família'];

  constructor(private dataBaseService: ContatosDataBaseService) {
  }

  ngOnInit() {
  }

  enviarDados() {
    if(this._tipo == undefined){
```

```

        this._tipo = this.tipos[0];
    }

    let novoContato = new ContatoModel(this._nome, this._telefone
, this._email, this._tipo);

    this.dataBaseService.setContato(novoContato);

    this.enviado = ! this.enviado;
}

voltar() {
    this._nome = '';
    this._telefone = '';
    this._email = '';
    this._tipo = '';
    this.enviado = ! this.enviado;
}
}

```

Na customização da classe CSS `card` , mudamos somente o `background`. Veja como ficou.

```

.card {
    background-color: #f0f3f8;
}

```

8.7 COMPONENTE DE LISTA DOS CONTATOS

O componente de lista é bem simples e pequeno. Ele somente vai mostrar uma lista de contatos e notificar o componente pai quando algum for clicado. A estrutura do HTML é uma estrutura de tabela bem simples, com algumas classes do bootstrap e algumas marcações do Angular 2.

As classes do bootstrap serão `row` , `justify-content-start` , `col` e `table` . Das marcações do Angular 2, usaremos `ngFor` , `ngStyle` e `click` . Nosso arquivo de template ficará

assim:

```
<h3>
  Contatos Cadastrados
</h3>

<div class="row justify-content-start">
  <div class="col-9">
    <table class="table">
      <thead>
        <tr>
          <th>#</th>
          <th>Nome</th>
          <th>Tipo</th>
        </tr>
      </thead>
      <tbody>
        <tr [ngStyle]="{'cursor': 'pointer'}" *ngFor="let
          item of listaDeContatos, let i = index" (click)="contatoClicado(
i)">
          <th scope="row">{{i + 1}}</th>
          <td>{{item.nome}}</td>
          <td>{{item.tipo}}</td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

Na classe do componente, teremos um array com o nome de `listaDeContatos`, que será do tipo `ContatoModel` e servirá para listá-los. Teremos também uma variável de `@Output()` com o nome de `idClicado` que será um `EventEmitter()`. Ele vai enviar para o componente pai o `id` do contato clicado.

Para finalizar, teremos o método `contatoClicado(item: number)` que, quando executado, vai emitir o `id` do contato que foi clicado. Nossa classe `lista-usuario.component.ts` ficará

assim:

```
import { Component, EventEmitter, OnInit, Output } from '@angular/core';
import { ContatoModel } from '../modelos/contato-model';
import { ContatosDataBaseService } from '../servicos/contatos-data-base.service';

@Component({
  selector: 'app-lista-usuario',
  templateUrl: './lista-usuario.component.html',
  styleUrls: ['./lista-usuario.component.css']
})
export class ListaUsuarioComponent implements OnInit {

  listaDeContatos: ContatoModel [] = [];
  @Output() idClicado = new EventEmitter();

  constructor(private dataBaseService: ContatosDataBaseService) {
  }

  ngOnInit() {
    this.dataBaseService.enviarContato.subscribe(contatos => this.listaDeContatos = contatos);
  }

  contatoClicado(item: number) {
    this.idClicado.emit(item);
  }
}
```

8.8 COMPONENTE DE DETALHE DO CONTATO

Nosso componente de detalhe somente vai receber as informações e mostrar no navegador. Ele não terá código de desenvolvimento, mas terá uma estrutura simples com classes CSS do bootstrap e *interpolações* do Angular 2, para mostrar os dados no navegador. Vamos usar as classes do bootstrap `row` ,

justify-content-start e form-group.

Nosso arquivo `detalhe-usuario.component.html` ficará assim:

```
<h3>
  Detalhe do Contato
</h3>

<div class="row justify-content-start">
  <div class="col-9">

    <div class="form-group">
      <label for="nome">Nome</label>
      <p id="nome"><strong>{{contato?.nome}}</strong></p>
    </div>

    <div class="form-group">
      <label for="telefone">Telefone</label>
      <p id="telefone"><strong>{{contato?.telefone}}</strong></p>
    </div>

    <div class="form-group">
      <label for="email">Email</label>
      <p id="email"><strong>{{contato?.email}}</strong></p>
    </div>

    <div class="form-group">
      <label for="tipo">Tipo</label>
      <p id="tipo"><strong>{{contato?.tipo}}</strong></p>
    </div>
  </div>
</div>
```

Nossa classe do componente é a mais simples de todos. Ela somente terá a variável de `@Input` para receber os dados do contato.

Nosso arquivo `detalhe-usuario.component.ts` ficará assim:

```
import { Component, Input } from '@angular/core';
import { ContatoModel } from '../modelos/contato-model';

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

  @Input() contato: ContatoModel;
}
```

Construindo todos os códigos apresentados aqui, e salvando e rodando o projeto no navegador, teremos um projeto funcionando e usando vários dos recursos que aprendemos durante o livro. Veja que expliquei e mostrei todos os códigos do nosso projeto. Fazendo todos os procedimentos, seu projeto também funcionará com sucesso.

Caso queira, deixei todo código no meu GitHub. Fique à vontade para clonar e trabalhar em cima dele para comparar com o criado por você.

Agora veremos como vamos fazer o build de produção e como ter um produto final.

8.9 FAZENDO BUILD PARA PRODUÇÃO

Fazer o build para produção é muito fácil (igual a tudo no Angular 2). Somente precisamos digitar um simples comando no console e o build será feito.

No console, vamos digitar somente **um** desses comandos:

```
ng build --target=production --
```



```
environment=prod
ng build --prod --env=prod
ng build --prod
```

E qual a diferença entre eles? Nenhuma! Veja que os comandos são abreviações chegando até o último que é somente `ng build --prod`. Isso mesmo. Para criar o projeto para produção, somente digitamos `ng build --prod` e mais nada.

Será criada uma pasta com o nome de `dist` e, dentro dela, serão criados todos os arquivos `.CSS` e `.js`, já minificados. O build do Angular 2 já minifica e ofusca todo o código, tanto por segurança como para ficar mais leve. Assim, é retirado tudo o que não será utilizado, só deixando o código para funcionamento da aplicação.

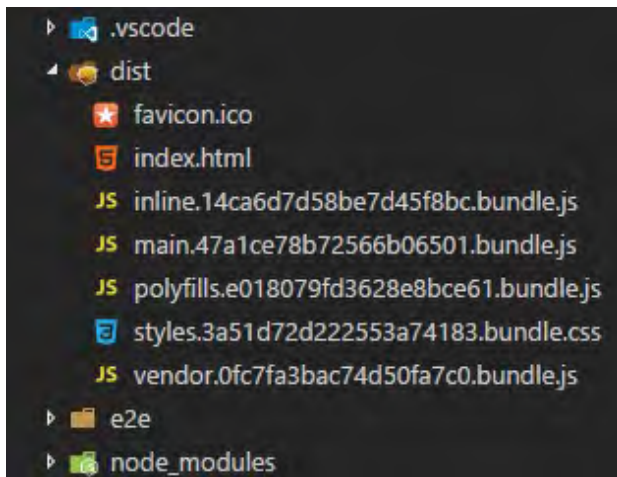
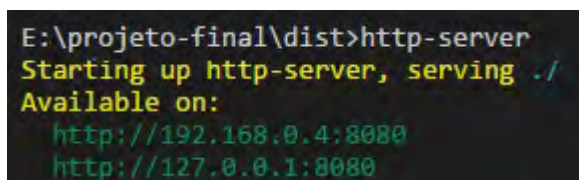


Figura 8.7: Pasta dist criada para produção

Para verificar o funcionamento da nossa aplicação de produção, instalaremos um servidor do Node.js que será o `http`

server . Digitemos no console `npm install http-server -g` , e rapidamente um servidor do Node.js já estará instalado no seu computador.

Para rodar o servidor Node.js, vamos entrar na pasta `dist` e digitar no console `http-server` . Segundos depois, o servidor estará rodando.

A screenshot of a Windows command prompt window. The prompt is 'E:\projeto-final\dist>'. The user has entered 'http-server'. The output is: 'Starting up http-server, serving ./' in yellow, 'Available on:' in yellow, 'http://192.168.0.4:8080' in green, and 'http://127.0.0.1:8080' in green.

```
E:\projeto-final\dist>http-server
Starting up http-server, serving ./
Available on:
http://192.168.0.4:8080
http://127.0.0.1:8080
```

Figura 8.8: Servidor rodando

Abra um navegador e digite a porta que está rodando o servidor — nesse caso, será `127.0.0.1:8080` . Logo em seguida, nossa aplicação estará rodando.

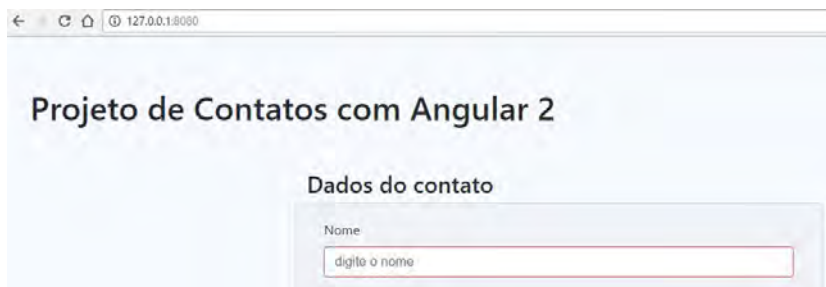


Figura 8.9: Projeto pronto para hospedagem

8.10 RESUMO

Neste capítulo, juntamos todos os conceitos aprendidos durante o livro, e fizemos um novo projeto desde a criação com o

`ng new` até o `build` para produção com o `ng build --prod`.
Conseguimos juntar os conceitos e criar um produto final para produção.

ANGULAR 4

Após dois anos de desenvolvimento, a equipe do Angular lançou a versão 2 no dia 14 de setembro de 2016. Desde seu lançamento, muitos outros produtos foram lançados baseando e integrando-se com o Angular 2. Além disso, seu ecossistema vem crescendo constantemente a cada dia.

O lançamento do Angular 2 não foi de forma completa. Naquele momento, tínhamos um framework estável o suficiente para que a equipe do Google dissesse: *pode confiar em nós e começar a construir em cima disso, e no futuro não vamos quebrar sua aplicação*. E assim, de tempos em tempos, essa equipe de desenvolvimento foi liberando lançamentos menores. Logo, foi lançado o Angular 2.1, em outubro de 2016, que tinha melhorias nos roteamentos.

Já em novembro de 2016, foi lançado o Angular 2.2, que deixou o sistema compatível com o AOT (*Ahead of Time*, sua explicação mais detalhada foge do assunto deste livro). Logo após foi lançado o Angular 2.3, que teve melhorias sobre a linguagem, mensagens de erros e avisos melhorados e, enfim, hoje temos o Angular 2.4.

Para informar quando teremos novas versões, a equipe do

Angular formou um calendário de lançamentos, e informou que, a partir de agora, elas serão feitas no modelo de **SEMVER**. Este modelo é também conhecido como **semantic version**, que traduzindo fica *controle de versão semântica*, em que cada casa numérica no conjunto de 3 números representa uma importância de atualização.

Vamos pegar como exemplo o número **2.3.1**. Esse número de versão será dividido em **Major**, **Minor**, **path**, em que:

- 1 — path — correção de bugs e pequenos ajustes;
- 3 — minor — adição de novas funcionalidades, e terá toda compatibilidade com a versão atual;
- 2 — major — mudança de versão, adição de novas features e possível risco de incompatibilidade.

O *path* é o último dígito e significa que não há recursos novos e sem alterações significativas. Este terá o menor impacto sobre a aplicação. O *Minor* é o número do meio, e diz que teremos novas features (funções), mas que não serão de grande impacto. Já o primeiro número, o *Major*, é onde teremos as alterações significativas e as grandes mudanças, isto é, onde será mexido fortemente no core do framework e possíveis bugs poderão acontecer.

Seguindo o calendário feito pela equipe do Angular, teremos lançamentos semanais, mensais e semestrais. Será lançado um pacote (path) de versão pequena toda a semana, uma versão menor (Minor) todo mês, e uma versão maior (Major) a cada seis meses.

Para evitar possíveis problemas com projetos que já usam o Angular 2, todas as atualizações ou alterações feitas pela equipe de

desenvolvimento serão testadas internamente nos próprios produtos do Google. Esses testes serão feitos em forma de cascata, ou seja, qualquer alteração feita no código todo será recopilada e testada dentro do Google em produtos internos para corrigir possíveis erros e bugs na aplicação.

9.1 POR QUE ANGULAR 4?

Um dos motivos para a nova versão do Angular foi a atualização do TypeScript para 2.2. O Angular 2 tem suporte para o TypeScript 1.8, e quem fizer a atualização para a nova versão dele poderá ter problemas de compatibilidade com a versão 2 do Angular.

Outro ponto muito importante para a mudança de número foi o fato de todos os pacotes do core do Angular estarem na versão 2.4, mas somente o pacote de roteamento estar na versão 3.4, devido a várias mudanças que este já teve. Então chegaram a uma conclusão de que: *como pode um framework chamar Angular 2 se usa pacotes de versão 3.4?*

Além disso, no Angular 4, foram criadas novas features para ajudar no desenvolvimento, e foi mudada a forma de compilação e minificação do código para que a versão final de produção seja menor e mais leve. Referente ao tamanho do código final para a produção ficar menor, isso é um dos pontos mais pedidos pelos desenvolvedores e é onde a equipe de desenvolvimento vem atuando com mais frequência, visto que, em comparação com outros frameworks front-end do mercado, o Angular é o que tem o maior pacote de build final.

Juntando a atualização do TypeScript, as divergências de versões dos pacotes do core e a adição de novas funcionalidades e, seguindo o padrão *SEMVER*, eis que surge o nome de **Angular 4**. Para chegar a ele, foram feitos 8 releases durante os meses, sempre adicionando pequenos pacotes para que nada fosse quebrado nas aplicações.

A cada adição de pacote, o lema sempre era: *entre mais 10% de velocidade e quebrar algum componente, vamos preservar o componente*. Tudo foi muito bem pensado para deixar o sistema sempre compatível.

A versão final do Angular 4 foi lançada em 23 março de 2017, e totalmente compatível com a versão 2 do Angular. Como agora os planos para o Angular é um novo lançamento a cada 6 meses, a evolução incremental será assim:

Versão	Plano
Angular 5	setembro / outubro 2017
Angular 6	março 2018
Angular 7	setembro / outubro 2018

O plano feito para essas mudanças é pelo fato de que o Angular não pode ficar parado no tempo, e tem de evoluir com o resto de todo o ecossistema da web moderna, sempre implementando novas funções e usando tudo o que tem de novo na tecnologia.

E assim como são feitos os testes internos nos produtos do Google, a própria equipe de desenvolvimento do Angular pede para que a comunidade também teste e forneça os feedbacks sobre o que funciona e o que não funciona nas novas versões que são

lançadas conforme a agenda do plano de lançamento, e assim evoluir junto com o framework.

9.2 SOMENTE ANGULAR

Um grande pedido que Igor Minar (líder da equipe do Angular) e toda a equipe de desenvolvimento do Angular fizeram é sobre o nome do framework, pois, a partir de agora, como será lançada uma nova versão a cada 6 meses, não devemos chamar de Angular 2, Angular 3, Angular 4, mas sim somente **Angular**.

Seguindo a própria equipe do Google, para as pessoas que não sabem o novo processo de atualização, sempre acharão de que se trata de um novo framework e de que tudo terá de ser mudado. Com isso, ele dará uma impressão de sistema frágil.

A equipe do Google pede para também padronizar o logo do símbolo do Angular que é usado em vários locais. Esses símbolos estão em: <https://angular.io/presskit.html>.

Usando esses símbolos, sempre teremos os padrões para a plataforma. Eles foram feitos pela equipe de design do Google e deve servir como base para o que você quiser fazer. Não terão direitos autorais, já que eles querem que a comunidade use e faça coisas impressionantes com eles.

Outro ponto importante quanto a codificação, muitos ainda acham ruim ou estranho programar em TypeScript. Porém, o que a equipe do Angular esclarece é que o desenvolvedor pode ficar livre para desenvolver em qualquer padrão, seja ele ES5, ES6, Dart ou TypeScript.

As documentações oficiais serão todas focadas em TypeScript, pois é a linha de código que o Angular usa por ser mais fácil, rápida e limpa no desenvolvimento. Também seria uma loucura para a equipe do Angular desenvolver uma documentação para cada tipo de linguagem.

O Angular continua sendo totalmente suportado nas linguagens ES5 e ES6, mas por questão de padronização, as documentações serão todas focadas em TypeScript. Caso queira usar outro tipo de linguagem, terá de ler a documentação feita em TypeScript e ajustar para a sua necessidade.

9.3 ATUALIZAR PROJETO PARA ANGULAR 4

O processo de atualização do Angular 2 para o Angular é bastante simples. Somente precisamos atualizar os pacotes que estão na pasta `node_modules`.

Vamos na pasta da aplicação pelo prompt de comando e digitamos no **Windows**:

```
npm install @angular/common@latest
@angular/compiler@latest @angular/compiler-cli@latest
@angular/core@latest @angular/forms@latest
@angular/http@latest @angular/platform-browser@latest
@angular/platform-browser-dynamic@latest
@angular/platform-server@latest @angular/router@latest
@angular/animations@latest typescript@latest --save
```

Já no **Mac** ou **Linux** será digitado:

```
npm install @angular/{common,compiler,compiler-
```

```
cli,core,forms,http,platform-browser,platform-browser-  
dynamic,platform-server,router,animations}@latest  
typescript@latest --save
```

Com essa série de atualizações dos pacotes do `node_modules`, o nosso projeto já está na versão 4 do Angular. Muito simples, correto?

Agora vamos atualizar nosso projeto (caso esteja desatualizado) para a versão 4 do Angular. Neste momento, este é nosso `package.json`.

```
{  
  "name": "livro-angular2",  
  "version": "0.0.0",  
  "license": "MIT",  
  "angular-cli": {},  
  "scripts": {  
    "ng": "ng",  
    "start": "ng serve",  
    "test": "ng test",  
    "lint": "ng lint",  
    "e2e": "ng e2e"  
  },  
  "private": true,  
  "dependencies": {  
    "@angular/common": "^2.4.0",  
    "@angular/compiler": "^2.4.0",  
    "@angular/core": "^2.4.0",  
    "@angular/forms": "^2.4.0",  
    "@angular/http": "^2.4.0",  
    "@angular/platform-browser": "^2.4.0",  
    "@angular/platform-browser-dynamic": "^2.4.0",  
    "@angular/router": "^3.4.0",  
    "core-js": "^2.4.1",  
    "rxjs": "^5.1.0",  
    "zone.js": "^0.7.6"  
  },  
  "devDependencies": {  
    "@angular/cli": "1.0.0-beta.32.3",  
    "@angular/compiler-cli": "^2.4.0",  
  }  
}
```

```

    "@types/jasmine": "2.5.38",
    "@types/node": "~6.0.60",
    "codelyzer": "~2.0.0-beta.4",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "karma-coverage-istanbul-reporter": "^0.2.0",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
    "tslint": "~4.4.2",
    "typescript": "~2.0.0"
  }
}

```

Repare que as dependências de produção estão na versão 2.4, menos o pacote `router` (roteamento), que está na versão 3.4. Vamos na pasta do nosso projeto e, em seguida, digitaremos os comandos apresentados anteriormente de acordo com o seu sistema operacional.

Uma dica importante aqui é copiar e colar o código, pois são várias coisas para digitar e, caso você erre qualquer letra, os pacotes não serão instalados. Então vamos copiar para ter certeza de que todas as instalações serão feitas.

No final, o nosso `package.json` ficará assim:

```

{
  "name": "livro-angular2",
  "version": "0.0.0",
  "license": "MIT",
  "angular-cli": {},
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "test": "ng test",
  }
}

```

```

    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "^4.0.1",
    "@angular/common": "^4.0.1",
    "@angular/compiler": "^4.0.1",
    "@angular/compiler-cli": "^4.0.1",
    "@angular/core": "^4.0.1",
    "@angular/forms": "^4.0.1",
    "@angular/http": "^4.0.1",
    "@angular/platform-browser": "^4.0.1",
    "@angular/platform-browser-dynamic": "^4.0.1",
    "@angular/platform-server": "^4.0.1",
    "@angular/router": "^4.0.1",
    "core-js": "^2.4.1",
    "rxjs": "^5.1.0",
    "typescript": "^2.2.2",
    "zone.js": "^0.7.6"
  },
  "devDependencies": {
    "@angular/cli": "1.0.0-beta.32.3",
    "@angular/compiler-cli": "^2.4.0",
    "@types/jasmine": "2.5.38",
    "@types/node": "~6.0.60",
    "codelyzer": "~2.0.0-beta.4",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "karma-coverage-istanbul-reporter": "^0.2.0",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
    "tslint": "~4.4.2",
    "typescript": "~2.0.0"
  }
}

```

Veja que não tivemos problemas com a atualização de versão. Nada parou de funcionar, e tudo está bem compatível.

9.4 NOVAS FEATURES

Muito foi mexido na parte de minificação e velocidade, pois hoje esse é um grande problema do Angular. Com as atualizações, a nova versão do build ficou menor e mais rápida, reduzindo em até 60% o código gerado para os componentes.

Para o desenvolvimento, foram lançadas novas funcionalidades e algumas que englobam assuntos do livro. Para nós, neste momento, podemos citar três mudanças, que são: a adição do `else` dentro dos templates; implementação de parâmetros no laço `for`; e validação automática de e-mail em formulários.

Vamos criar um novo componente com o nome de `Angular4` e seguir os mesmos procedimentos feitos durante todo o livro. A primeira coisa que veremos será no `*ngIf`, já que ele agora tem o `else`.

No template, vamos adicionar um botão com a tag `button` e duas `div`. Uma terá a frase com o `if`, e a outra terá a frase com o `else`. No final, o template ficará assim:

```
<button (click)="mudar()">Angular 4</button>

<div *ngIf="troca">
  <p>com o if</p>
</div>
<div *ngIf="! troca">
  <p>com o else</p>
</div>
```

Na classe do componente, vamos adicionar uma variável booleana com o nome de `troca` e um método com o nome de `mudar()`.

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

```

@Component({
  selector: 'app-angular4',
  templateUrl: './angular4.component.html',
  styleUrls: ['./angular4.component.css']
})
export class Angular4Component implements OnInit {

  troca: boolean = true;

  constructor() { }

  ngOnInit() {
  }

  mudar() {
    this.troca = !this.troca;
  }
}

```

Salvando todos os arquivos e voltando ao navegador, ao clicarmos no botão, a frase será mudada. Até aqui, tudo normal, já fazíamos isso antes. Mas e agora com o Angular 4, como ficou isso? Esta é a nova forma do `if / else`.

```

<div *ngIf="troca; else outro">
  <p>com o if</p>
</div>
<ng-template #outro>
  <p>com o else</p>
</ng-template>

```

Não entendeu? Vou lhe explicar.

Logo no início, temos a grande mudança do `*ngIf`. Agora ele está dessa forma: `*ngIf="troca; else outro"`. O que significa `else outro`?

Já sabemos que o `*ngIf` faz referência à variável `troca`, que está na classe do componente. Até aqui tudo bem. Mas após o

ponto e vírgula, temos o `else` , que faz referência à variável `outro` . Entretanto, onde está essa variável `outro` ?

Simples. Veja logo abaixo, na tag `ng-template` . Lá temos uma variável de template com o nome de `outro` .

Para usar o `if / else` no Angular 4, devemos criar um `<ng-template>` com uma variável de referência. Essa variável será o nosso `else` do nosso `if` . Confuso? Não, com certeza, com o tempo você acostuma.

Mas e se você quiser continuar usando dois `if` , como fizemos antes? Pode usar, tranquilamente. Não teremos erros. A criação do `if / else` é mais uma alternativa para a criação dos templates.

Crie esse template e veja como ficará. Veja se achará mais fácil ou mais difícil do que colocar dois `ifs` .

Neste momento, nosso template está assim:

```
<p>
  Angular 4
</p>

<button (click)="mudar()">Angular 4</button>

<div *ngIf="troca">
  <p>com o if</p>
</div>
<div *ngIf="! troca">
  <p>com o else</p>
</div>

<div *ngIf="troca; else outro">
  <p>com o if</p>
</div>
<ng-template #outro>
  <p>com o else</p>
</ng-template>
```

Outra mudança foi na diretiva `ngFor`, que agora temos um novo parâmetro de `length`. Veja o próximo exemplo.

Na classe do componente, criaremos um array com o nome de `tecnologias`. Vamos atribuir os valores: `['Angular 2', 'Angular 4', 'JavaScript', 'html', 'CSS']`.

No nosso template, mostraremos a posição de cada elemento dentro do array, junto com seu nome e a quantidade total desse array. Este será o nosso código:

```
<ul>
  <li *ngFor="let item of tecnologias; let i = index">
    {{i}} - {{item}} - {{i + 1}} de {{tecnologias.length}}
  </li>
</ul>
```

Veja que, nesse código, tivemos algumas concatenações para mostrar o que queríamos. Mas agora, com o Angular 4, o `ngFor` recebeu um novo parâmetro, o `length`, que faz o cálculo automático do tamanho do array. Veja como ficou:

```
<ul>
  <li *ngFor="let item of tecnologias; count as count; let i =
index">
    {{i}} - {{item}} - {{i + 1}} de {{count}}
  </li>
</ul>
```

Com o Angular 4, tudo fica dentro da diretiva `ngFor`, e no nosso código só ficará o que realmente queremos mostrar.

Ainda tivemos uma pequena mudança quando declaramos o `let i = index`. Podemos deixá-lo um pouco menor. Veja como ficará:

```
<ul>
  <li *ngFor="let item of tecnologias; count as count; index as
```



```

i">
    {{i}} - {{item}} - {{i + 1}} de {{count}}
</li>
</ul>

```

Colocando `index` as `i`, não precisamos declarar uma variável interna com o `let`. Tudo fica na responsabilidade da diretiva `ngFor`.

Salvando e rodando o projeto no navegador, teremos os mesmos resultados:

Livro Angular 2

Angular 4

Angular 4

com o if

com o if

- 0 - Angular 2 - 1 de 5
- 1 - Angular 4 - 2 de 5
- 2 - javaScript - 3 de 5
- 3 - html - 4 de 5
- 4 - CSS - 5 de 5

- 0 - Angular 2 - 1 de 5
- 1 - Angular 4 - 2 de 5
- 2 - javaScript - 3 de 5
- 3 - html - 4 de 5
- 4 - CSS - 5 de 5

- 0 - Angular 2 - 1 de 5
- 1 - Angular 4 - 2 de 5
- 2 - javaScript - 3 de 5
- 3 - html - 4 de 5
- 4 - CSS - 5 de 5

Figura 9.1: Resultados das listas com Angular 2 e Angular 4

A última mudança ficou nos formulários. Nele foi adicionado

um autenticador de e-mail, que servirá para validar automaticamente um campo onde você declará-lo como sendo do tipo `email`.

Vamos criar um formulário parecido com o feito no capítulo de formulário, porém agora teremos somente um campo de `input` e será do tipo `email`. Veja como vai ficar:

```
<form #cadastro="ngForm">
  <div>
    <label for="email">Email</label>
    <input type="email" id="email" #nome="ngModel" [(ngModel)
]= "email" name="email" email required>

    <div [hidden]="nome.valid || nome.pristine">
      <p>Email inválido</p>
    </div>
  </div>
  <button type="submit" [disabled]="cadastro.form.invalid">Envi
ar</button>
</form>
```

Veja que, ao lado da palavra `required`, colocamos a palavra `email`, e isso é tudo para que esse campo tenha essa validação automática. Se quiser fazer o teste, retire a palavra `email` e rode a aplicação. O campo aceitará qualquer coisa que escrever e o botão sempre ficará liberado para enviar.

Colocando a palavra `email` dentro da tag de `input`, e ele sendo do tipo `type="email"`, o próprio Angular vai validar se o conteúdo digitado é um e-mail válido.

9.5 RESUMO

Neste capítulo, mostramos toda a trajetória de construção e atualizações do Angular 2, a nova forma *SEMVER* adotada pela

equipe de desenvolvimento do Angular para futuras versões, e como será usado o tipo de versionamento. Explicamos também o porquê de o nome do Angular 2 pular para o Angular 4, os planos de futuros lançamentos e o pedido de que agora o nosso tão querido framework seja chamado somente de *Angular*.

Para finalizar, mostramos como podemos atualizar um projeto para a nova versão do Angular (e isso é bem tranquilo!) e as novas funcionalidades adicionadas em sua versão 4: `ngIf` / `else` e `ngFor` .

Espero que tenha gostado. Fico feliz por chegar até aqui, e aguardo seu feedback. Nos vemos em breve. Abraço!

BIBLIOGRAFIA

PEYROTT, Sebastián. *More Benchmarks: Virtual DOM vs Angular 1 & 2 vs Others*. Jan. 2016. Disponível em: <https://auth0.com/blog/more-benchmarks-virtual-dom-vs-angular-12-vs-mithril-js-vs-the-rest/>.