IMT - Instituto Mauá de Tecnologia

Engenharia de Computação

ECM252 – Linguagens de Programação II

Professor Rodrigo Bossini

Aula 08 - Angular - Introdução - Componentes - Data Binding

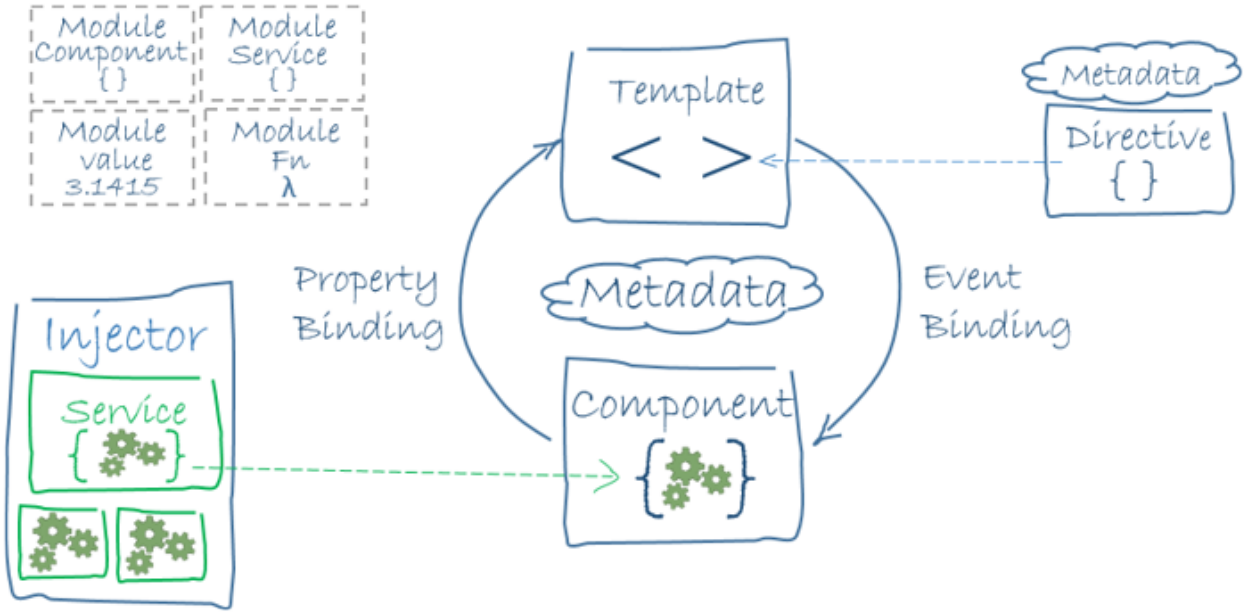
1 Introdução

O uso de SPAs (Single Page Applications) é muito comum nos dias atuais. Uma SPA é, como o nome sugere, uma aplicação de página única. A ideia é que a aplicação forneça uma boa experiência de navegação para o usuário. Que ele possa escolher funcionalidades diferentes sem que o navegador precise “piscar” no momento em que envia uma nova requisição ao servidor.

O Angular, um dos frameworks front-end mais utilizados atualmente, é especialmente projetado para o desenvolvimento de SPAs, embora possa ser aplicado para o desenvolvimento de aplicações em geral.

Trata-se de um framework poderoso com muitos recursos. É interessante ter uma visão geral como mostra a Figura 1.1. Conforme estudamos e utilizamos seus recursos, aquilo que mostra a Figura 1.1 fica mais claro.

Figura 1.1



2 Desenvolvimento

2.1 (Criando uma aplicação e colocando ela em execução) Uma aplicação Angular possui uma estrutura de diretórios e arquivos bem definida. O Angular CLI automatiza a sua criação, nos fornecendo maior produtividade. Neste seção,vamos criar uma primeira aplicação para testar o ambiente.

- Comece criando um diretório que será o seu **workspace**. É uma simples pasta que será utilizada para abrigar os seus projetos Angular. Seu nome pode ser qualquer um. Algo como **angular-workspace** pode ser uma boa ideia.

- O Angular CLI é usado com o comando **ng** (de A**ng**ular). Para criar um projeto, use

ng new **nome-do-projeto**

- Não inclua o módulo de roteamento nesse momento, ele não será necessário.

- Use CSS simples para os estilos. Note que há outras opções como Less ou Sass, que não utilizaremos no momento.

- Quando a criação do projeto terminar, você terá uma pasta chamada **nome-do-projeto** no seu workspace. Use o comando a seguir para navegar até ela.

cd nome-do-projeto

- O Angular possui um servidor que auxilia no desenvolvimento. Para colocá-lo em execução e poder acessar a aplicação no navegador, use

ng serve

Nota: Se for necessário trocar a porta padrão, você pode usar **ng serve --port portaDesejada**.

Nota: Você também pode user **ng serve -o** para abrir automaticamente uma aba do seu navegador.

- Note que há uma pasta chamada **src** na estrutura do projeto. Iremos trabalhar principalmente dentro dela. Por isso, abra o VS Code e use **File >> Open Folder** para abrir essa pasta.

- O conteúdo que a aplicação exibe por padrão está definido no seu componente principal, no arquivo **app.componenent.html**. Apague todo seu conteúdo e escreva o conteúdo da Listagem 2.1.1.

Listagem 2.1.1

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Angular</title>
</head>

<body>
  Meu primeiro app Angular!!!
</body>

</html>
```

- Salve o arquivo e verifique o resultado no navegador.

2.2 (Componentes) Um dos conceitos mais importantes do Angular é a criação de componentes. Quando criamos um componente, estamos de certa forma extendendo a linguagem HTML. Ou seja, adicionando a ela novos elementos que podem ser usados lado a lado com os elementos do HTML como p, div etc. Um componente possui um aspecto visual e possivelmente comportamento dinâmico, que podemos especificar. Veja um exemplo na Figura 2.2.1.

Figura 2.2.1

- Na Figura 2.2.1, cada parte destacada pode ser um componente Angular. Sua definição visual é feita com HTML simples. Associadas à definição visual, podemos ter diversas funcionalidades, o que torna cada componente Angular dinâmico, podendo ter comportamento e aspecto visual diferente em tempo de execução. Além disso, componentes tendem a ser blocos facilmente reutilizáveis, quando bem projetados.

- É possível criar componentes utilizando o Angular CLI ou mesmo manualmente. Vamos criar um primeiro componente manualmente e, para os próximos exemplos, utilizar o Angular CLI, visando o ganho de produtividade.

- Para **criar um componente manualmente**, clique com o direito em **app** e crie uma pasta com o nome do componente. Neste caso, iremos criar um componente que exibe a mensagem “Olá, Angular”. O nome da pasta será, portanto, **ola-angular**.

- A seguir, crie um arquivo chamado **ola-angular.component.ts**. Note que o sufixo **component** é somente uma convenção. Não é obrigatório mas é altamente recomendável, já que é um padrão utilizado por milhões de desenvolvedores.

- Para criar um componente, precisamos criar uma classe, como na Listagem 2.2.1.

Listagem 2.2.1

```
export class OlaAngularComponent {

}
```

- Note que trata-se de uma simples classe. Para que o Angular a considere como um componente, vamos **decorá-la**, como na Listagem 2.2.2. Note que, para usar a anotação Component, precisamos importá-la do pacote core do Angular. Note, também, que a anotação espera receber um objeto JSON, que será usado para especificar as características do componente.

Listagem 2.2.2

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

@Component({

})
export class OlaAngularComponent {

}
```

- Uma das propriedades esperadas se chama **selector**. Ele é utilizado para especificarmos o nome do elemento que será utilizado nas páginas HTML. Na Listagem 2.2.3 escolhemos o seletor **ola-angular**. Isso quer dizer que, nas páginas HTML do projeto, ele pode ser usado assim: **<ola-angular></ola-angular>**.

Listagem 2.2.3

```
@Component({
  selector: "ola-angular"
})
export class OlaAngularComponent {

}
```

- A seguir, podemos especificar o **template** do componente. Trata-se de código HTML padrão, que irá dizer a forma como o componente aparece na tela. É o HTML a ser renderizado pelo navegador. Note o uso de crases para podermos pular linhas. Veja a Listagem 2.2.4.

Listagem 2.2.4

```
@Component({
  selector: "ola-angular",
  template: `
      <p>Olá, Angular</p>
  `
})
export class OlaAngularComponent {

}
```

- A classe pode, evidentemente, ter variáveis. Elas podem ser utilizadas no código do template por meio do operador de **interpolação**. Veja o exemplo da Listagem 2.2.5.

Listagem 2.2.5

```
@Component({
  selector: "ola-angular",
  template: `
    <p>Olá, {{framework}}</p>
  `
})
export class OlaAngularComponent {
  framework = "Angular";
}
```

- Um componente precisa ser declarado por um módulo (veremos sobre isso adiante). O único módulo da aplicação no momento pode ser encontrado no arquvo **app.module.ts**. Abra esse arquivo e ajuste o seu conteúdo como na Listagem 2.2.6.

Listagem 2.2.6

```
import { OlaAngularComponent } from './ola-angular/ola-angular.component';
@NgModule({
  declarations: [
    AppComponent,
    OlaAngularComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- Para testar o componente, abra o arquivo **app.component.html** e ajuste o seu conteúdo como mostra a Listagem 2.2.7.

Listagem 2.2.7

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Angular</title>
</head>

<body>
  <ola-angular></ola-angular>
</body>

</html>
```

- Para se familiarizar com os possíveis erros do Angular, remova a declaraçaõ do componente no módulo e veja o console no Chrome Dev Tools.

- Caso o código HTML seja mais longo, pode ser de interesse criar um arquivo separado para sua definição e referenciá-lo na anotação do componente com o atributo **templateUrl**. Para isso, crie um arquivo chamado **ola-angular.component.html**. Defina seu conteúdo como mostra a Listagem 2.2.8.

Listagem 2.2.8

```
<div style="width: 1280px; margin:auto; text-align: center;">
   <h2>Olá, Angular</h2>
   <hr>
</div>
```

- A Listagem 2.2.9 mostra como referenciá-lo.

Listagem 2.2.9

```
@Component({
   selector: "ola-angular",
   templateUrl: "./ola-angular.component.html"
})
export class OlaAngularComponent {
   framework = "Angular";
}
```

- Também podemos criar um arquivo separado para o código CSS. Crie um novo arquivo chamado **ola-angular.component.css**. Seu conteúdo é dado na Listagem 2.2.10.

Listagem 2.2.10

```
div {
   width: 1280px;
   margin: auto;
   text-align: center;
}
```

- A seguir, remova o CSS inline da div no arquivo **ola-angular.component.html**, como na Listagem 2.2.11.

Listagem 2.2.11

```
<div>
   <h2>Olá, Angular.</h2>
   <hr>
</div>
```

- Para referenciar o arquivo CSS, no arquivo **ola-angular.component.ts**, use o atributo styleUrls, que deverá ser associado a um vetor que contém o nome do arquivo. A Listagem 2.2.12 mostra como.

Listagem 2.2.12

```
@Component({
   selector: "ola-angular",
   templateUrl: "./ola-angular.component.html",
   styleUrls: ['./ola-angular.component.css']
})
export class OlaAngularComponent {
   framework = "Angular";
}
```

- O Angular CLI pode automatizar esse processo. Para **criar um componente como Angular CLI**, use o seguinte comando

ng generate component ola-angular-cli

- Note que ele criou quatro arquivos. Os três conhecidos e um outro, cujo nome tem como fixo **spec.ts**. Trata-se de um arquivo que pode ser utilizado para testes e que não será utilizado no momento.

- Repare que o arquivo **ola-angular-cli.component.ts** possui essencialmente a mesma estrutura que vimos até então. Fora isso, ele define também um construtor padrão e a classe implementa a interface **OnInit**, incluindo um método que ela define. Não utilizaremos nada disso por enquanto. Por isso, ajuste o arquivo como na Listagem 2.2.13.

Listagem 2.2.13

```
@Component({
  selector: 'app-ola-angular-cli',
  templateUrl: './ola-angular-cli.component.html',
  styleUrls: ['./ola-angular-cli.component.css']
})
export class OlaAngularCliComponent {

}
```

- Abra o arquivo **ola-angular-cli.component.html** e ajuste o seu conteúdo como na Listagem 2.2.14.

Listagem 2.2.14

```
<div>
   <h2>Olá, Angular CLI!</h2>
   <hr>
</div>
```

- A seguir, abra o arquivo **app.component.html** e utilize o novo componente, como na Listagem 2.2.15. Note que, por padrão, o Angular CLI usa **app** como prefixo para os componentes que ele cria.

Listagem 2.2.15

```
<body>
  <ola-angular></ola-angular>
  <app-ola-angular-cli></app-ola-angular-cli>
</body>
```

- Repare que o seletor **div** não teve efeito sobre esse componente. Somente os arquivos importados pelo compontente podem alterar seu conteúdo. Abra o arquivo **ola-angular-cli.component.css** e adicione a ele o conteúdo da Listagem 2.2.16.

Listagem 2.2.16

```
div {
   background-color: aliceblue;
   text-align: center;
   width: 1280px;
   margin: auto;
}
```

- Perceba também que não foi necessário adicionar o componente ao módulo da aplicação. Isso ocorre pois o próprio Angular CLI o fez. Abra o arquivo **app.module.ts** e verifique a sua existência na seção **declarations**.

2.3 (Instalação do Bootstrap) O Boostrap pode ser obtido de diferentes formas. Uma delas é por meio do próprio Node Package Manager. Para obter a última versão, por exemplo, use o comando

npm install bootstrap@latest

Certifique-se de executar esse comando no diretório do projeto.

- Os pacotes do Node ficam instalados na pasta **node_modules**. Precisamos encontrar a pasta do Boostrap nesta pasta e referenciá-la. Verifique a existência da seguinte pasta

node_modules/bootstrap/dist/css/bootstrap.css

- Agora encontre o arquivo **angular.json** na raiz do projeto. Encontre a seguinte chave

projects/nome-do-projeto/architect/build/options/styles

Note que ela está associada a um vetor que já inclui o arquivo CSS principal da aplicação. Precisamos incluir o arquivo CSS do Boostrap nesse vetor. A Listagem 2.3.1 mostra como ele deve ficar.

Listagem 2.3.1

```
"styles": [
            "src/styles.css",
            "node_modules/bootstrap/dist/css/bootstrap.css"
          ],
```

Nota: Pode ser necessário reiniciar o servidor depois de fazer ajustes no arquivo **angular.json**.

- Para verificar se o Boostrap funcionou, adicione o código da Listagem 2.3.2 ao arquivo **app.component.html**.

Listagem 2.3.2

```
<table class="table table-hover table-striped">
  <thead>
    <tr>
      <th>Framework</th>
      <th>Aplicação</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Angular</td>
      <td>Front-End</td>
    </tr>
    <tr>
      <td>Spring MVC</td>
      <td>Back-End</td>
    </tr>
  </tbody>
</table>
```

2.4 (Data Binding) Como vimos, a definição de um componente Angular pode envolver diversos arquivos. Independentemente do uso de múltiplos arquivos, conceitualmente temos o componente (código Typescript) e o template (código HTML).

- Por um lado, a classe Typescript pode definir variáveis e/ou métodos que talvez queiramos utilizar no template.

- Por outro lado, o template pode ter elementos visuais (talvez um botão) que sofrem eventos nos quais estamos interessados e que temos de tratar no código Typescript.

Veja a Figura 2.4.1. Ela ilustra quatro possibilidades:

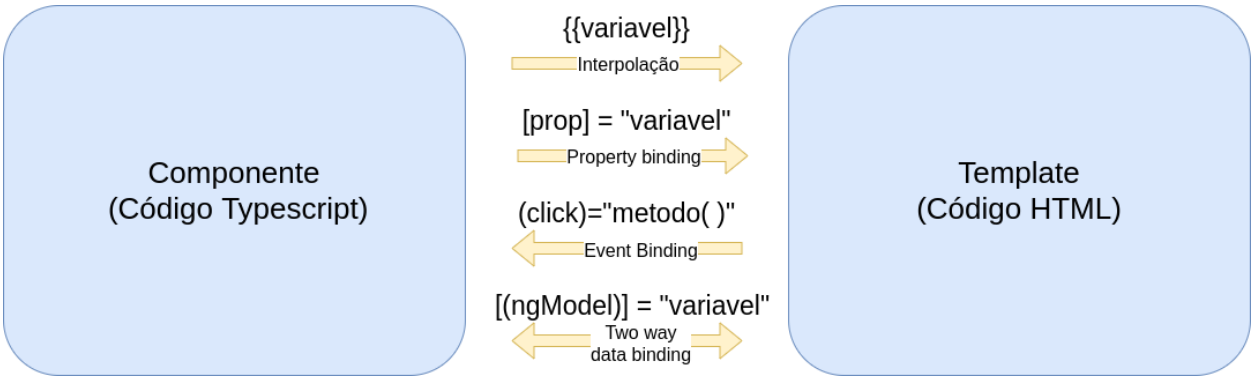
- **Interpolação**: Neste caso, o template usa a notação {{}} para avaliar uma expressão (que pode ser uma variável ou um método do componente, por exemplo) e exibi-la textualmente.

- **Property Binding**: O operador [ ] é usado pelo template para ler uma expressão do componente e associar a uma propriedade de algum elemento que ele defina.

- **Event Binding**; O operador ( ) é usado para a especificação de eventos que podem acontecer no template e que interessam para o componente. Ali especificamos um método a ser chamado quando o evento acontecer.

- **Two Way Data Binding**: É uma mistura do Property Binding com o Event Binding. Quando usamos esse mecanismo, a atualização ocorre em mão dupla (daí o nome): Um elemento visual do template fica associado a uma variável do componente. Caso o elemento seja atualizado, a variável também é. Caso a variável seja atualizada, o componente visual também é. O operador que identifica esse mecanismo é o [( )]. Repare que são os mesmos usados para Property Binding e Event Binding. Faz todo sentido.

Figura 2.4.1



- Para testar o mecanismo de data binding do Angular, crie um novo projeto. Isso pode ser feito com

ng new nome-do-projeto

- Lembre-se de, no terminal, sair do diretório do projeto atual (talvez com cd ..) para não criar um projeto dentro do outro. Seu projeto deve ser criado no seu workspace, que é a pasta criada para abrigar os projetos lado a lado.

- Não adicione o módulo de roteamento e use CSS simples.

- Coloque o projeto em execução com

ng serve -o

- Abra o arquivo **app.component.html** e apague todo o seu conteúdo.

- Escreva nele o conteúdo da Listagem 2.4.1. Esse exemplo mostra que podemos usar qualquer expressão Javascript/Typescript no operador de interpolação.

Listagem 2.4.1

```
2 + 2 = {{2 + 2}}
```

- Agora vamos definir algumas variáveis no componente e renderizá-las textualmente no template. Declare as variáveis da Listagem 2.4.2 no arquivo **app.component.ts**.

Listagem 2.4.2

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nome = "José";
  idade = 20
}
```

- A seguir, edite o conteúdo do arquivo **app.component.html** como na Listagem 2.4.3.

Listagem 2.4.3

```
<p>Oi, meu nome é {{nome}} e tenho {{idade}} anos.</p>
```

- Averigue o resultado no seu navegador.

- Também é possível chamar métodos no operador de interpolação. Adicione o método da Listagem 2.4.4 ao arquivo **app.component.ts** e, depois, utilize-o no arquivo **app.component.html** como na listagem 2.4.5.

Listagem 2.4.4

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nome = "José";
  idade = 20

  lancarDado() {
    return Math.floor(Math.random() * 6) + 1;
  }
}
```

Listagem 2.4.5

```
<p>Oi, meu nome é {{nome}} e tenho {{idade}} anos.</p>
<p>Lançando um dado: {{lancarDado()}}</p>
```

- Para testar o **Event Binding**, vamos começar adicionando o Bootstrap ao projeto. Para isso, no diretório do projeto, execute

npm install bootstrap@latest

- A seguir, não se esqueça de ajustar o arquivo **angular.json**.

- Ajuste o conteúdo do arquivo **app.component.html** como mostra a Listagem 2.4.6. Note que temos um único botão.

Listagem 2.4.6

```
<div class="container">
  <button type="button" class="btn btn-primary">Adicionar</button>
</div>
```

- A seguir, definimos um método que desejamos que seja executado quando o botão for clicado. Ele aparece na Listagem 2.4.7 e deve ser adicionado ao arquivo **app.component.ts**.

Listagem 2.4.7

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

  adicionar() {
    console.log("Adicionando...");
  }
}
```

- Para tratar o evento de clique no botão, aplique o Event Binding, como na Listagem 2.4.8.

Listagem 2.4.8

```
<div class="container">
  <button type="button" class="btn btn-primary" (click)="adicionar()">Adicionar</button>
</div>
```

- Para ver o log, abra o Chrome Dev Tools e clique no botão.

- A seguir, vamos adicionar um form simples para simular a adição de nomes de pessoas. Ajuste o arquivo **app.component.html** como mostra a Listagem 2.4.9.

Listagem 2.4.9

```
<div class="container">
  <form>
  <div class="form-group">
    <label for="nomeInput">Nome</label>
    <input type="nome" class="form-control" id="nomeInput" placeholder="Digite um nome">
  </div>
  <button type="button" class="btn btn-primary btn-block"
(click)="adicionar()">Adicionar</button>
  </form>
</div>
```

- Podemos tratar o evento “entrada de dados” no campo do form. A cada vez que o usuário alterar seu conteúdo, podemos fazer com que um método entre em execução. Declare uma variável e um método que altera seu conteúdo no arquivo **app.component.ts**, como na Listagem 2.4.10.

Listagem 2.4.10

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

  nome;

  alterarNome(nome) {
    console.log(nome);
    this.nome = nome;
  }

  adicionar() {
    console.log("Adicionando...");

  }
}
```

- A seguir, no arquivo **app.component.html**, vincule o método alterNome (com Event Binding) ao componente de entrada de texto. Use o evento **input**. Note que usamos a variável implícita **event.** Veja a Listagem 2.4.11.

Listagem 2.4.11

```
<div class="container">
  <form>
  <div class="form-group">
    <label for="nomeInput">Nome</label>
    <input type="nome" class="form-control" id="nomeInput" placeholder="Digite um nome"
(input)="alterarNome($event)">
  </div>
  <button type="button" class="btn btn-primary btn-block"
(click)="adicionar()">Adicionar</button>
  </form>
</div>
```

- Digite algo no campo textual e veja o resultado no console do Chrome Dev Tools. Repare que o objeto **event** representa o evento como um todo. Ele inclui diversas informações sobre o evento, sendo que uma delas é o texto que foi digitado. Ele pode ser encontrado em **evento.target.value**. Assim, ajuste o método alterarNome como na Listagem 2.4.12 para pegar o nome corretamente.

Listagem 2.4.12

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

  nome;

  alterarNome(nome) {
    console.log(nome.target.value);
    this.nome = nome.target.value;
  }

  adicionar() {
    console.log("Adicionando...");

  }
}
```

- Note que podemos usar o operador de interpolação para exibir a variável conforme ela é editada pelo evento. Adicione o conteúdo da Listagem 2.4.13 ao template (**app.component.html)**.

Listagem 2.4.13

```
<div class="container">
  <form>
  <div class="form-group">
    <label for="nomeInput">Nome</label>
    <input type="nome" class="form-control" id="nomeInput" placeholder="Digite um nome"
(input)="alterarNome($event)">
  </div>
  <button type="button" class="btn btn-primary btn-block"
(click)="adicionar()">Adicionar</button>
  <div class="alert alert-primary mt-3 p-3">
    {{nome}}
  </div>
  </form>
</div>
```

- Digamos que desejamos pegar o que o usuário digita somente no momento em que ele clica no botão. Nesse caso, o primeiro passo é remover o Event Binding do campo de entrada textual do template, como na Listagem 2.4.14.

Listagem 2.4.14

```
<div class="container">
  <form>
    <div class="form-group">
      <label for="nomeInput">Nome</label>
      <input type="nome" class="form-control" id="nomeInput" placeholder="Digite um nome">
    </div>
    <button type="button" class="btn btn-primary btn-block"
(click)="adicionar()">Adicionar</button>
    <div class="alert alert-primary mt-3 p-3">
      {{nome}}
    </div>

  </form>
</div>
```

- A seguir, vamos utilizar uma variável de referência de template. Trata-se de um nome que podemos associar a um elemento do template. A sintaxe usada para isso é **#nomeDesejado**. Uma vez definido um nome, o elemento pode ser referenciado por ele. Ela somente pode ser referenciada no próprio template, o componente não tem acesso a ela.

- Atribua uma variável de referência de template ao campo de entrada textual. A seguir, passe seu nome para o método adicionar, como na Listagem 2.4.15.

Listagem 2.4.15

```
<div class="container">
  <form>
    <div class="form-group">
      <label for="nomeInput">Nome</label>
      <input type="nome" class="form-control" id="nomeInput" placeholder="Digite um nome"
#nomeInput>
    </div>
    <button type="button" class="btn btn-primary btn-block"
(click)="adicionar(nomeInput)">Adicionar</button>
    <div class="alert alert-primary mt-3 p-3">
      {{nome}}
    </div>

  </form>
</div>
```

- Adicione um parâmetro no método adicionar e exiba seu valor no console, como na Listagem 2.4.16. Averigue o resultado no Chrome Dev Tools.

Listagem 2.4.16

```
adicionar(nomeInput) {
    console.log(nomeInput);
    console.log("Adicionando...");

  }
```

- Note que o que recebemos é o próprio objeto input. Ou seja, a variável de referência referencia o próprio nó na árvore DOM. Como sabemos, um elemento input possui a propriedade value. Ela armazena o valor do campo. Ajuste o método adicionar como na Listagem 2.4.17 e veja o resultado no Chrome Dev Tools.

Listagem 2.4.17

```
adicionar(nomeInput) {
   console.log(nomeInput.value);
  }
```

- Para exibir o valor no template, basta fazer com que o método adicionar altere o valor da variável nome. Veja a Listagem 2.4.18.

Listagem 2.4.18

```
 adicionar(nomeInput) {
   console.log(nomeInput.value);
   this.nome = nomeInput.value;
  }
```

- Uma das formas de Data Binding que citamos se chama **Property Binding**. Trata-se de um mecanismo que nos permite vincular expressões (como variáveis ou chamadas a funções) a propriedades de elementos do template. Por exemplo, suponha que desejamos exibir a caixa com o nome adicionado somente depois que o botão for clicado. Para começar, vamos adicionar uma variável ao componente, como na Listagem 2.4.19.

Listagem 2.4.19

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

  nome;
  exibirCaixa = false;

  alterarNome(nome) {
    console.log(nome.target.value);
    this.nome = nome.target.value;
  }

  adicionar(nomeInput) {
    console.log(nomeInput.value);
    this.nome = nomeInput.value;
  }
}
```

- Podemos vincular o valor dessa propriedade ao atributo **hidden** da caixa que exibe o nome no template. Veja a Listagem 2.4.20. Lembre-se que o operador [ ] é usado para fazer Property Binding.

Listagem 2.4.20

```
<div class="alert alert-primary mt-3 p-3" [hidden]=”!exibirCaixa”>
    {{nome}}
  </div>
```

- Quando o botão for clicado, alteramos o valor de exibirCaixa e a caixa aparece. Veja a Listagem 2.4.21.

Listagem 2.4.21

```
adicionar(nomeInput) {
  this.nome = nomeInput.value;
  this.exibirCaixa = true;
}
```

Referências

Angular. 2020. Disponível em <https://angular.io>. Acesso em maio de 2020.