

# Desenvolvimento web com Angular

Jackson Gomes \ [jgomes@ceulp.edu.br](mailto:jgomes@ceulp.edu.br)

# Sumário

<b>Prefácio</b>	<b>v</b>
Fonte de referência e versão do Angular . . . . .	v
Convenções . . . . .	v
Conhecimentos desejáveis . . . . .	vi
<b>1 Introdução</b>	<b>1</b>
1.1 Servidor web . . . . .	1
1.2 Desenvolvimento front-end . . . . .	1
1.2.1 HTML para a marcação . . . . .	4
1.2.2 Manipulação do DOM . . . . .	5
1.2.3 CSS para formatação . . . . .	5
1.2.4 JavaScript para lógica . . . . .	7
1.3 jQuery . . . . .	11
<b>2 Iniciando com o Angular</b>	<b>13</b>
2.1 Elementos da Arquitetura do Angular . . . . .	14
2.1.1 Módulos . . . . .	14
2.1.2 Bibliotecas . . . . .	14
2.1.3 Componentes . . . . .	14
2.1.4 Metadados . . . . .	14
2.1.5 Data binding . . . . .	15
2.1.6 Diretivas . . . . .	15
2.1.7 Serviços . . . . .	15
2.2 Estrutura padrão de um software desenvolvido em Angular . . . . .	16
<b>3 Angular CLI</b>	<b>18</b>
3.1 Instalando . . . . .	18
3.2 Comandos . . . . .	18
3.2.1 help . . . . .	19
3.2.2 new . . . . .	19
3.2.3 generate . . . . .	20
3.2.4 serve . . . . .	20
3.2.5 build . . . . .	21
3.3 Criando e executando um projeto Angular . . . . .	21
3.4 O arquivo package.json . . . . .	22

3.5	Estrutura do projeto . . . . .	23
3.5.1	AppModule . . . . .	24
3.5.2	AppComponent . . . . .	25
3.5.3	index.html . . . . .	27
3.6	O que vem a seguir? . . . . .	28
<b>4</b>	<b>Gerenciador de notícias com Angular</b>	<b>29</b>
4.1	Clonar e executar localmente . . . . .	29
4.2	O projeto . . . . .	30
4.2.1	AppComponent . . . . .	30
4.2.1.1	Controller . . . . .	30
4.2.1.2	Template . . . . .	32
4.3	Depuração de código no browser . . . . .	35
4.4	Conclusão . . . . .	36
<b>5</b>	<b>CRUD completo do noticias-angular</b>	<b>37</b>
5.1	Clonar e executar localmente . . . . .	38
5.2	CRUD . . . . .	38
5.3	Documentação do projeto . . . . .	38
5.4	O projeto . . . . .	39
5.5	Classe <code>Noticia</code> . . . . .	39
5.6	AppComponent . . . . .	41
5.6.1	Interface <code>OnInit</code> . . . . .	41
5.6.2	Exibir telas diferentes . . . . .	41
5.6.3	Listar notícias recentes de forma ordenada . . . . .	44
5.6.4	Apresentar conteúdo da notícia para leitura . . . . .	46
5.6.5	Listar todas as notícias com pesquisa . . . . .	47
5.6.6	Cadastrar e editar notícia . . . . .	48
5.6.7	Excluir notícia . . . . .	51
5.7	Conclusão . . . . .	51
<b>6</b>	<b>Interação entre componentes</b>	<b>52</b>
6.1	Clonar e executar localmente . . . . .	52
6.2	Criar componente com o Angular CLI . . . . .	53
6.3	Configuração do módulo . . . . .	53
6.4	Estrutura inicial do <code>NoticiasRecentesComponent</code> . . . . .	53
6.5	Utilizando o componente <code>NoticiasRecentesComponent</code> . . . . .	54
6.6	Criando eventos . . . . .	56
6.7	Conclusão . . . . .	58
<b>7</b>	<b>Rotas e Serviços</b>	<b>59</b>
7.1	Serviços . . . . .	59
7.2	URI, URL e URN . . . . .	61
7.3	Rotas . . . . .	62
7.4	Rotas no Angular . . . . .	63
7.4.1	Shell component . . . . .	65

7.4.2	Parâmetros de rota . . . . .	65
7.4.3	Navegação . . . . .	66
7.5	Padrão de trabalho (workflow) . . . . .	67
<b>8</b>	<b>Recursos de interface gráfica do Bootstrap</b>	<b>69</b>
<b>9</b>	<b>Comunicação com back-end por meio do HttpClient</b>	<b>70</b>
9.1	Configuração do módulo . . . . .	70
9.2	Recebendo dados de um arquivo JSON . . . . .	71
9.3	Comunicação assíncrona e o <code>Observable</code> . . . . .	74
<b>A</b>	<b>Configuração do ambiente de desenvolvimento</b>	<b>76</b>
A.1	Node.js . . . . .	76
A.2	Angular CLI . . . . .	76
<b>B</b>	<b>TypeScript</b>	<b>78</b>
	<b>Referências</b>	<b>79</b>

# Lista de Tabelas

# Lista de Figuras

1.1	Exemplo de comunicação cliente-servidor . . . . .	2
1.2	Tela de cadastro de notícias no software noticias-js . . . . .	3
1.3	Tela de lista de notícias no software noticias-js . . . . .	3
3.1	Versão inicial do software em execução no Browser . . . . .	22
3.2	Estrutura básica de um projeto Angular . . . . .	23
3.3	Relação entre Model-Template-Controller no AppComponent . . . . .	26
3.4	Execução do Template inicial . . . . .	27
4.1	Estrutura do projeto angular-noticias . . . . .	30
4.2	Janela do browser com ferramentas de desenvolvedor habilitadas . . . . .	35
5.1	Estrutura do projeto angular-noticias, branch crud-telas . . . . .	39
5.2	Diagrama de estados demonstrando navegação das telas do projeto angular-noticias, branch crud-telas . . . . .	43
7.1	Composição da URL . . . . .	61
9.1	Diagrama de sequência da consulta da lista de notícias . . . . .	74

# Lista de Códigos-fontes

1.1	Trecho do arquivo index.html do software noticias-js . . . . .	4
1.2	Trecho do arquivo main.css do software noticias-js . . . . .	6
1.3	Trecho do arquivo main.js do software noticias-js . . . . .	7
3.1	Código-fonte do AppModule no software noticias-angular . . . . .	24

# Prefácio

Este é um livro open-source, tanto no conteúdo quanto no código-fonte associado. O conteúdo é resultado de algumas práticas com o **Framework Angular** e segue uma abordagem prática, com foco no entendimento de conceitos e tecnologias no contexto do desenvolvimento de um software web.

Um *framework* representa um modelo, uma forma de resolver um problema. Em termos de desenvolvimento de software para a web um framework fornece ferramentas (ie. código) para o desenvolvimento de aplicações. Geralmente o propósito de um framework é agilizar as atividades de desenvolvimento de software, inclusive, fornecendo código pronto (componentes, bibliotecas etc.) para resolver problemas comuns, como uma interface de cadastro.

O objetivo deste livro é fornecer uma ferramenta para o desenvolvimento de habilidades de desenvolvimento web com Angular, com a expectativa de que você comece aprendendo o básico (o “hello world”) e conclua com habilidades necessárias para o desenvolvimento de software que consome dados e interage com uma API HTTP REST, por exemplo.

## Fonte de referência e versão do Angular

Parte do conteúdo do livro é baseada na documentação oficial do Angular, disponível em <https://angular.io>.

Como o Angular é um projeto em constante desenvolvimento (pelo menos até agora) serão publicadas atualizações no conteúdo do livro sempre que possível, para refletir novos recursos e funcionalidades. No momento, o conteúdo do livro é baseado na versão **6.0.0**.

## Convenções

Os trechos de código apresentados no livro seguem o seguinte padrão:

- **comandos:** devem ser executados no prompt; começam com o símbolo \$
- **códigos-fontes:** trechos de códigos-fontes de arquivos

A seguir, um exemplo de comando:



```
$ mkdir hello-world
```

O exemplo indica que o comando `mkdir`, com a opção `hello-world`, deve ser executado no prompt para criar uma pasta com o nome `hello-world`.

A seguir, um exemplo de código-fonte:

```
1 class Pessoa:
2     pass
```

O exemplo apresenta o código-fonte da classe `Pessoa`. Em algumas situações, trechos de código podem ser omitidos ou serem apresentados de forma incompleta, usando os símbolos `...` e `#`, como no exemplo a seguir:

```
1 class Pessoa:
2     def __init__(self, nome):
3         self.nome = nome
4
5     def salvar(self):
6         # executa validação dos dados
7         ...
8         # salva
9         return ModelManager.save(self)
```

## Conhecimentos desejáveis

Este livro aborda o desenvolvimento de software front-end para a web do ponto-de-vista do Angular. Isso quer dizer que não trata de conceitos iniciais de HTML, CSS, JavaScript, TypeScript e Bootstrap. Entretanto, os conceitos fundamentais dessas tecnologias vão sendo apresentados no decorrer dos capítulos, conforme surge a necessidade deles.

Para aprender mais sobre essas tecnologias recomendo essas fontes:

- **TypeScript:** [Documentação oficial do TypeScript - Microsoft](#), [TypeScript Deep Dive](#)
- **HTML, CSS e JavaScript:** [W3Schools](#)
- **Bootstrap:** [Documentação oficial do Bootstrap](#)

Este livro não leva em consideração o Sistema Operacional do seu ambiente de desenvolvimento, mas é importante que você se acostume a certos detalhes e a certas ferramentas, como o **prompt** ou **prompt de comando**.

Além destas ferramentas também são utilizadas:

- **Node.js:** disponível em <https://nodejs.org> representa um ambiente de execução do JavaScript fora do browser e também inclui o **npm**, um gerenciador de pacotes

- **Editor de textos ou IDE:** atualmente há muitas opções, mas destaco o **VisualStudio-Code**, disponível em <https://code.visualstudio.com/>
- **Git**
- **Heroku**

O **Git** é um gerenciador de repositórios com recursos de versionamento de código. É uma ferramenta essencial para o gerenciamento de código fonte de qualquer software.

O **Heroku** é um serviço de **PaaS** (de *Platform-as-a-Service*). PaaS é um modelo de negócio fornece um ambiente de execução conforme uma plataforma de programação, como o Python, um tecnologia de banco de dados, como MySQL e PostgreSQL e ainda outros recursos, como cache usando Redis.

---

### Calma! Não pira!

(*In*)Felizmente você não vai usar todas as tecnologias lendo o conteúdo desse livro. Fica para outra oportunidade.

---

Para utilizar o Heroku você precisa criar uma conta de usuário. Acesse <https://www.heroku.com/> e crie uma conta de usuário.

Depois que tiver criado e validado sua conta de usuário instale o **Heroku CLI**, uma ferramenta de linha de comando (prompt) que fornece uma interface de texto para criar e gerenciar aplicativos Heroku. Detalhes da instalação dessa ferramenta não são tratados aqui, mas comece acessando <https://devcenter.heroku.com/articles/heroku-cli>.

# Capítulo 1

## Introdução

### 1.1 Servidor web

Um **servidor web** é um programa que fornece um serviço de rede que funciona recebendo e atendendo requisições de clientes. Um **cliente**, por exemplo, é o browser.

Um **cliente** solicita um arquivo ao **servidor web**, que recebe a solicitação, atende a solicitação e retorna uma resposta para o cliente.

Esse modelo é chamado **cliente-servidor** e, na web, utiliza o protocolo **HTTP** (de *Hypertext Transfer Protocol*). O protocolo HTTP determina as regras da comunicação no modelo cliente-servidor:

- como o cliente deve enviar uma solicitação para o servidor
- como o servidor deve interpretar a solicitação
- como o servidor deve enviar uma resposta para o cliente
- como o cliente deve interpretar a resposta do servidor

Para ilustrar esse processo a Figura 1.1 demonstra a comunicação entre cliente e servidor.

Como a Figura 1.1 apresenta, quem inicia a comunicação é o cliente. O servidor recebe a solicitação e retorna uma resposta. A resposta pode ser interpretada como sucesso ou erro. No caso da figura, se o servidor encontrar o arquivo, ele retorna um código de resposta do HTTP com o número 200 e o conteúdo HTML do arquivo `index.html`, caso contrário ele retorna um código de resposta HTTP com o número 404, indicando que o arquivo não foi encontrado.

### 1.2 Desenvolvimento front-end

O termo **front-end** no contexto do desenvolvimento do software tem relação com a utilização de tecnologias e ferramentas para o desenvolvimento de software que, geralmente, executa em um cliente. Considerando o cenário anterior, da comunicação **cliente-servidor**, estamos falando

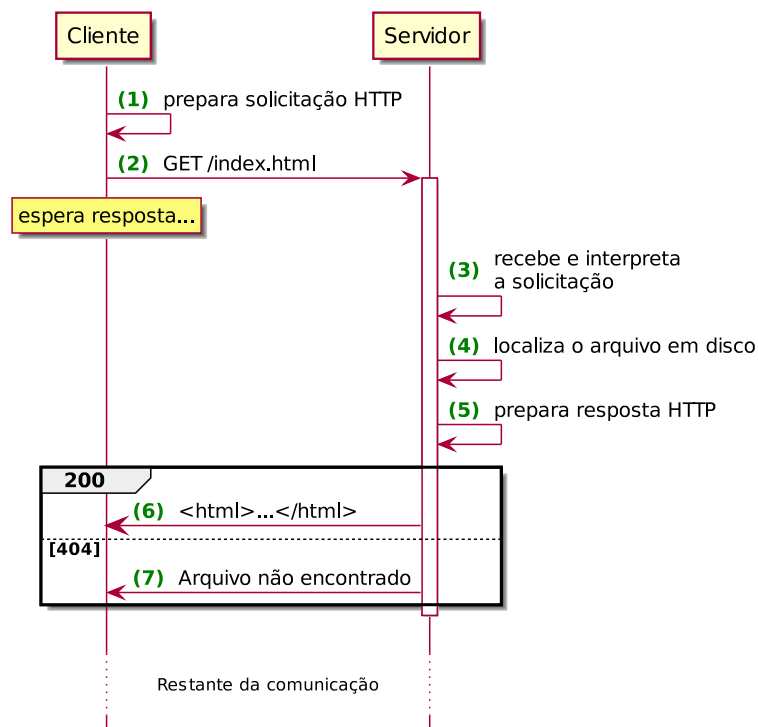


Figura 1.1: Exemplo de comunicação cliente-servidor

justamente do **browser**. O **browser** se torna uma peça fundamental nesse tipo de desenvolvimento de software.

Grande parte do desenvolvimento front-end se direciona para a tríade composta por HTML, CSS e JavaScript:

- **HTML** sendo utilizada como linguagem de marcação
- **CSS** sendo utilizada como linguagem de formatação
- **JavaScript** sendo utilizada como linguagem para adicionar interação (lógica de interface e lógica de negócio)

Para exemplificar, veja o projeto **noticias-js**. **noticias-js** é um software de gerenciamento de notícias com repositório em <https://github.com/jacksongomesbr/webdevbook-noticias-js>, desenvolvido em HTML, CSS e JavaScript e possui as seguintes funcionalidades:

- cadastrar notícia (título e conteúdo)
- ver a lista de notícias (título)
- ver o conteúdo de uma notícia (clicando no título)

Figuras 1.2, 1.3 ilustram o software e essas funcionalidades.

Esse comportamento já não é novidade em software web: a interface com o usuário permite a entrada de dados e a interação por meio de cliques. Os detalhes para fazer esse comportamento estão na utilização de JavaScript. Primeiro, a estrutura do software é baseada em três partes:

- **index.html**: contém o HTML para a marcação
- **main.css**: contém o CSS para a formatação

**Gerenciador de notícias**

**Notícias recentes**

Clique no título da notícia para expandir

**Cadastrar notícia**

Título

Conteúdo

Salvar Limpar

Informar o título da notícia

Informar o conteúdo da notícia

Clicar em "Salvar" para cadastrar a notícia

Clicar em "Limpar" para não cadastrar

Figura 1.2: Tela de cadastro de notícias no software noticias-js

**Gerenciador de notícias**

**Notícias recentes**

Clique no título da notícia para expandir

- **Salvador contra o Galo, Bruno Henrique dedica vitória do Palmeiras a Roger**

Bruno Henrique tirou o Palmeiras do marasmo que havia provocado três empates nas últimas três rodadas e decidiu a vitória por 3 a 2 sobre o Atlético-MG neste domingo. O jogo no Allianz Parque, válido pela 14ª rodada do Campeonato ... - Veja mais em <https://esporte.uol.com.br/futebol/campeonatos/brasileiro/serie-a/ultimas-noticias/2018/07/22/heroi-do-palmeiras-bruno-henrique-vibra-com-vitoria-sobre-rival-direto.htm?cmpid=copiaecola>

Fechar

Clicar no título da notícia para mostrar o conteúdo da notícia

Clicar em "Fechar" para ocultar o conteúdo da notícia

**Cadastrar notícia**

Título

Conteúdo

Salvar Limpar

Figura 1.3: Tela de lista de notícias no software noticias-js

- `main.js`: contém o JavaScript para implementação da lógica da interface

A seguir, as seções demonstram detalhes dessa estrutura.

### 1.2.1 HTML para a marcação

Código-fonte 1.1 apresenta um trecho do arquivo `index.html`.

Código-fonte 1.1: Trecho do arquivo `index.html` do software `noticias-js`

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      ...
5      <link rel="stylesheet" type="text/css" media="screen" href="main.css"
        />
6      <script src="main.js"></script>
7  </head>
8  <body>
9      <h1>Gerenciador de notícias</h1>
10     <h2>Notícias recentes</h2>
11     <p>Clique no título da notícia para expandir</p>
12     <div id="noticias-recentes">
13         <ul id="noticias-recentes-list"></ul>
14     </div>
15
16     <h2>Cadastrar notícia</h2>
17     <form onsubmit="salvar(this); return false;">
18         <div>
19             <label for="frm-titulo">Título</label>
20             <input type="text" id="frm-titulo" name="titulo" required>
21         </div>
22         <div>
23             <label for="frm-conteudo">Conteúdo</label>
24             <textarea id="frm-conteudo" name="conteudo" cols="80" rows="5"
                required></textarea>
25         </div>
26         <div>
27             <button type="submit">Salvar</button>
28             <button type="reset" formnovalidate>Limpar</button>
29         </div>
30     </form>
31 </body>
32 </html>
```

A primeira parte importante é o elemento `ul` com identificador (atributo `id`) `noticias-recentes-list`. A importância se dá para o fato de que esse identificador será utilizado no código JavaScript

para adicionar elementos `li`, um recurso chamado de **manipulação do HTML DOM**. Outra parte importante é em relação ao formulário de cadastro. Primeiro, o elemento `form` possui o atributo `onsubmit` com um valor que é um código JavaScript. Depois, cada campo do formulário está declarado para receber entrada do usuário:

- `input` com identificador `frm-titulo` é usado para o título da notícia
- `textarea` com identificador `frm-conteudo` é usado para o conteúdo da notícia

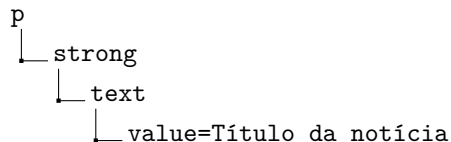
Todos os campos são de preenchimento obrigatório, então está sendo utilizado um recurso de validação diretamente no HTML por meio do atributo `required`. Por fim, o formulário tem dois botões (elemento `button`): “Salvar”, que tem o atributo `type` com valor `submit` e “Limpar”, que além de ter `type` com valor `reset` tem o atributo `formnovalidate`, que é utilizado para desabilitar a validação do formulário no clique do botão (é o comportamento padrão).

### 1.2.2 Manipulação do DOM

O DOM (de *Document Object Model*) é uma representação em memória de um documento HTML, na forma de uma **árvore** composta por nós que correspondem aos elementos do documento HTML. Por exemplo, para um trecho HTML como o seguinte:

```
<p><strong>Título da notícia</strong></p>
```

a árvore teria a seguinte estrutura:



Por causa da estrutura em árvore é possível identificar relações entre os elementos, por exemplo:

- a raiz da árvore é o nó `p`
- o nó `p` tem um filho, o nó `strong`
- o nó `strong` tem um pai, o nó `p`
- o nó `strong` tem um filho, o nó `text`
- o nó `text` tem um filho, o valor `Título da notícia`

Dessa forma grande parte da responsabilidade de **manipulação do DOM** recai sobre tarefas como encontrar um nó, percorrer filhos, adicionar filho em um nó e remover um nó. Para isso o DOM fornece objetos (principalmente o `document`), métodos e propriedades, que serão apresentados a seguir.

### 1.2.3 CSS para formatação

Usar CSS para formatação corresponde a criar **regras CSS** e definir como elas serão aplicadas a determinados elementos do documento HTML por meio dos **seletores**. Uma regra CSS é um

conjunto composto por pares **propriedade:valor**. O seletor informa para o browser como encontrar elementos para aplicar as propriedades. Há seletores: de elemento, de id e de classe. O seletor de elemento contém o nome do elemento. O seletor de id contém o símbolo # seguido de um identificador de elemento (valor do atributo `id`). O seletor de classe contém o símbolo . seguido de uma classe de elemento (um dos valores do atributo `class`).

A tela do software usa formatação em CSS, conforme mostra Código-fonte 1.2.

Código-fonte 1.2: Trecho do arquivo main.css do software noticias-js

```
1  label {  
2      display: block;  
3      font-weight: bold;  
4  }  
5  
6  form div {  
7      margin-bottom: 10px;  
8  }  
9  
10 .noticia .titulo {  
11     font-weight: bold;  
12 }  
13  
14 .noticia .conteudo {  
15     display: none;  
16 }
```

Há quatro grupos de regras, com seletores diferentes:

- `label`: aplica propriedades `display` e `font-weight` para todos os elementos `label`
- `form div`: aplica propriedade `margin-bottom` para todos os elementos `div` dentro de elementos `form`
- `.noticia .titulo`: aplica propriedade `font-weight` para todos os elementos que tenham atributo `class` contendo `titulo` dentro de elementos que tenham atributo `class` contendo `noticia`
- `.noticia .conteudo`: aplica propriedade `display` para todos os elementos que tenham atributo `class` contendo `noticia` dentro de elementos que tenham atributo `class` contendo `noticia`

A propriedade `display` com valor `none` é importante porque é utilizada para ocultar o conteúdo da notícia.

As regras CSS são aplicadas **em cascata** o que significa que há uma ordem de prioridade que o browser considera para resolver conflitos de estilos:

1. estilo **in-line** (definido no atributo `style` do elemento em questão)
2. estilo definido no elemento `style`
3. estilo definido em um arquivo `.css` externo (obtido por meio do elemento `link`)

Aprender a utilizar os seletores é uma parte importante do trabalho com CSS.



### 1.2.4 JavaScript para lógica

O código JavaScript é parcialmente ilustrado por Código-fonte 1.3.

Código-fonte 1.3: Trecho do arquivo main.js do software noticias-js

```
1  var noticias = [];  
2  
3  function atualizarLista(noticia) {  
4  }  
5  
6  function salvar(form) {  
7  }  
8  
9  function mostrarNoticia(id) {  
10 }  
11  
12 function ocultarNoticia(id) {  
13 }
```

O código foi apresentado parcialmente para um entendimento inicial da sua estrutura. A variável `noticias` é um `Array`, utilizado para armazenar objetos que representam as notícias cadastradas. Dessa forma o conteúdo está **apenas em memória** ou **em tempo de execução**. Quando a página é recarregada, o conteúdo é perdido. Na sequência são declaradas quatro funções: `atualizarLista()`, `salvar()`, `mostrarNoticia()` e `ocultarNoticia()`.

A função `salvar()` é chamada por meio de um tratador de evento. No código HTML ([lst:noticias-js-html]), no elemento `form` o atributo `onsubmit` representa um tratador de evento, que é ativado quando algum botão dentro do formulário é clicado (nesse caso, queremos que o botão “Salvar” ative esse tratador de evento). O conteúdo de um tratador de evento é um código JavaScript e, nesse caso, há duas instruções:

- chamar a função `salvar()` passando como argumento `this` (que é uma referência ao objeto DOM que representa o formulário HTML)
- cancelar o evento ao chamar `return false`

A seguir, o código completo da função `salvar()`:

```
1  function salvar(form) {  
2      var titulo = document.getElementById('frm-titulo').value;  
3      var conteudo = document.getElementById('frm-conteudo').value;  
4      var noticia = {  
5          id: noticias.length,  
6          titulo: titulo,  
7          conteudo: conteudo  
8      };  
9      noticias.push(noticia);
```

```
10     atualizarLista(noticia);  
11     form.reset();  
12 }
```

O função `salvar()` tem o parâmetro `form`, que recebe o argumento usado na chamada da função, no tratador de evento `onsubmit`. O interior do código tem duas linhas importantes, que interagem com o HTML DOM para obter valores dos campos do formulário. Isso é feito por meio do método `getElementById()` do objeto `document`, que procura um elemento no documento HTML cujo identificador seja igual ao argumento (`frm-titulo`, por exemplo) e retorna um objeto do DOM que representa o elemento. Por ser um campo de formulário, a propriedade `value` retorna o valor digitado pelo usuário.

Na sequência o código cria um objeto `noticia` com três atributos:

- `id`: que representa um identificador numérico da notícia (começando em zero)
- `titulo`: representa o título da notícia
- `conteudo`: representa o conteúdo da notícia

Depois, a sequência continua:

- o objeto `noticia` é adicionado no `Array noticias` por meio de uma chamada ao método `push()`
- chama a função `atualizarNoticia()` (descrita a seguir), informando como argumento o objeto `noticia` para que a notícia que acaba de ser cadastrada seja apresentada na lista
- chama o método `reset()` do objeto `form`, que é utilizado para redefinir os valores dos campos do formulário

A seguir, o código completo da função `atualizarList()`:

```
1  function atualizarLista(noticia) {  
2      var lista = document.getElementById('noticias-recentes-list');  
3      var li = document.createElement('li');  
4      li.setAttribute('id', 'noticia-' + noticia.id);  
5      li.setAttribute('class', 'noticia');  
6      li.innerHTML = '<p class="titulo" onclick="mostrarNoticia(' + noticia.  
          id + ')">  
7          + noticia.titulo  
8          + '</p>  
9          + '<p class="conteudo">  
10             + noticia.conteudo  
11             + '<br>  
12             + '<span>-----</span>  
13             + '<br>  
14             + '<button onclick="ocultarNoticia(' + noticia.id + ')">Fechar</  
                button>';  
15             + '</p>';  
16      lista.appendChild(li);  
17  }
```

O código utiliza o método `getElementById()` para obter uma referência para o objeto com iden-

tificador `noticias-recentes-list`, que representa o elemento `ul` que contém elementos `li` para apresentar a lista de notícias. A partir de então o objetivo do código é criar um elemento `li` e adicioná-lo ao elemento `ul`. Para isso, começa criando um elemento no DOM por meio do método `createElement()`, cujo argumento `"li"` representa o nome do elemento criado. Essa referência é mantida na variável `li` para o código da sequência:

- utiliza o método `setAttribute()` para definir o valor do atributo `id` (baseado no identificador da notícia)
- utiliza o método `setAttribute()` para definir o valor do atributo `class`
- utiliza a propriedade `innerHTML` para definir o restante do conteúdo HTML

Essa última parte, do valor de `innerHTML` merece destaque. A manipulação do DOM do HTML pode ser feita utilizando métodos (como `getElementById()` e `createElement()`) e também fazendo um **parser** de um conteúdo HTML. Nesse caso, por se tratar de um conteúdo mais longo, o código utiliza a segunda opção. Perceba que o conteúdo da propriedade, uma `string`, é conteúdo HTML, que é interpretado pelo **browser** para modificar o DOM do HTML.

Outra parte importante desse trecho de HTML representado na `string` é sua estrutura:

- elemento `p` com atributo `class` contendo `titulo` e atributo `onclick` (tratador de evento para clique)
- título da notícia
- elemento `p` com atributo `class` contendo `conteudo`
- conteúdo da notícia
- elemento `br`
- elemento `span` contendo traços
- elemento `br`
- elemento `button` com rótulo “Fechar” e atributo `onclick`

O atributo `onclick` representa o tratador de evento para clique. Nesse caso, o elemento `p` que contém o título da notícia tem um tratador de evento que chama a função `mostrarNoticia()`. O botão “Fechar” tem o tratador de evento que chama a função `ocultarNoticia()`. Por fim, o elemento `li` é adicionado na lista de filhos do objeto `lista` por meio do método `appendChild()`.

A seguir, o código da função `mostrarNoticia()`:

```
1 function mostrarNoticia(id) {
2     var li = document.getElementById('noticia-' + id);
3     for (var i = 0; i < li.childNodes.length; i++) {
4         var node = li.childNodes[i];
5         if (node.getAttribute('class') == 'conteudo') {
6             node.setAttribute('style', 'display:inline');
7         }
8     }
9 }
```

A função `mostrarNoticia()` recebe o parâmetro `id`, que representa o identificador da notícia que cujo conteúdo deve ser apresentado. O código opera da seguinte forma:

- encontra o elemento `li` cujo identificador corresponde ao parâmetro `id`
- para cada nó filho do elemento `li` (usa a propriedade `childNodes`):
  - se o nó filho (objeto `node`) tiver atributo `class` com o valor `'conteudo'` (usa o método `getAttribute()`) então
    - \* define o valor do atributo `style` com `'display:inline'`, o que faz com que ele se torne visível (contrário de `display:none`)

De forma semelhante, a função `ocultarNoticia()` recebe o parâmetro `id`, que representa o identificador da notícia cujo conteúdo deve ser ocultado:

```
1 function ocultarNoticia(id) {
2     var li = document.getElementById('noticia-' + id);
3     for (var i = 0; i < li.childNodes.length; i++) {
4         var node = li.childNodes[i];
5         if (node.getAttribute('class') == 'conteudo') {
6             node.setAttribute('style', 'display:none');
7         }
8     }
9 }
```

A principal diferença para a função `mostrarNoticia()` é que a a função `ocultarNoticia()` modifica o atributo `style` para o valor `display:none`, o que torna o conteúdo invisível novamente, completando, assim, a interação com o usuário.

Certamente esse não é um software simples para quem tem a primeira experiência com esse tipo de programação, mas é importante destacar esses aspectos:

- a estrutura do HTML é criada tendo em vista possibilitar a **manipulação do DOM** com o JavaScript (por isso o uso de valores controlados para os atributos `id` e `class`)
- o atributo `onclick` é um tratador de evento para clique
- o atributo `onsubmit` é um tratador de evento para o envio do formulário
- o atributo `formnovalidate` impede a validação do formulário
- o objeto `document` dá acesso ao DOM do HTML e permite usar as funções para manipulação do DOM
- o método `getElementById()` encontra um nó do DOM com base em um identificador (atributo `id`)
- o método `setAttribute()` cria ou altera o valor de um atributo de um nó
- o método `getAttribute()` retorna o valor de um atributo de um nó
- a propriedade `innerHTML` permite fazer parser de um trecho de HTML e inserir o resultado na árvore DOM
- o método `appendChild()` adiciona um nó na lista de nós filhos do nó pai
- a propriedade `childNodes` contém a lista de nós filhos do nó pai (é um `Array`)

## 1.3 jQuery

O **jQuery** é uma das primeiras **bibliotecas JavaScript** e foi criada para evitar uma quantidade enorme de retrabalho e verificações de suporte de diferentes versões e tipos de browser e também inclui funções para manipulação do DOM (THE JQUERY FOUNDATION, [s.d.]).

O repositório no **noticias-js** tem um branch **jquery**, que contém a implementação utilizando a biblioteca jQuery. Uma lista completa das diferenças entre o branch **master** e o **jquery** pode ser obtida em <https://github.com/jacksongomesbr/webdevbook-noticias-js/compare/jquery>. Na prática, as principais modificações estão no arquivo **main.js**, com detalhes para as implementações das funções. Começando pela função **salvar()** temos o seguinte:

```
1 function salvar(form) {
2     var titulo = $('#frm-titulo').val();
3     var conteudo = $('#frm-conteudo').val();
4     ...
5 }
```

O código em ... não muda em relação ao branch **master**. As variáveis **titulo** e **conteudo** continuam recebendo os valores informados pelo usuário no formulário, mas agora utilizam a função **\$()**, que é a principal função do jQuery e, nesse caso, acessa a árvore DOM em busca de elementos com is identificadores indicados por seletores CSS de id: **#frm-titulo** e **#frm-conteudo** encontram, respectivamente, os elementos com identificador **frm-titulo** e **frm-conteudo**. O valor dos campos é obtido pela função **val()**.

Já a função **atualizarLista()** muda bastante:

```
1 function atualizarLista(noticia) {
2     var lista = $('#noticias-recentes-list');
3     var li = $('<li id="noticia-" + noticia.id + ">');
4     li.addClass('noticia');
5     var p_titulo = $('<p>');
6     p_titulo.addClass('titulo');
7     p_titulo.attr('onclick', 'mostrarNoticia(' + noticia.id + ')');
8     p_titulo.html(noticia.titulo);
9     var p_conteudo = $('<p>');
10    p_conteudo.addClass('conteudo');
11    p_conteudo.html(noticia.conteudo
12        + '<br>'
13        + '<span>-----</span>'
14        + '<br>'
15        + '<button onclick="ocultarNoticia(' + noticia.id + ')>Fechar</button>');
16    li.append(p_titulo, p_conteudo);
17    p_conteudo.hide();
18    lista.append(li);
19 }
```

A variável `lista` representa o elemento do DOM com identificador `noticias-recentes-list`. A variável `li` recebe a chamada da função `$()` com uma string HTML como parâmetro (linha 3). Nesse caso, o jQuery cria uma árvore parcial do DOM fazendo **parser** do argumento (como acontece com a propriedade `innerHTML`). Uma classe CSS é adiciona no nó por meio do método `addClass()` (linha 4). Um atributo é adicionado ou alterado por meio do método `attr()` (linha 7). O conteúdo de um nó pode ser definido usando o método `html()` (como com a propriedade `innerHTML`), na linha 8. O método `append()` é utilizado para adicionar um nó na lista de filhos de um pai (linha 15). Por fim, o jQuery tem um modo próprio de esconder e mostrar elementos usando, respectivamente, os métodos `hide()` e `show()`. Esses métodos também são usados nas implementações das funções `ocultarNoticia()` e `mostrarNoticia()`, que se tornam:

```
1 function mostrarNoticia(id) {  
2     $('.conteudo', '#noticia-' + id).show();  
3 }  
4  
5 function ocultarNoticia(id) {  
6     $('.conteudo', '#noticia-' + id).hide();  
7 }
```

A parte importante fica por conta da chamada da função `$()`. Nesse caso há dois argumentos:

1. o seletor de classe `.conteudo`
2. o contexto, que usa um seletor de id (`#noticia-` seguido do identificador da notícia)

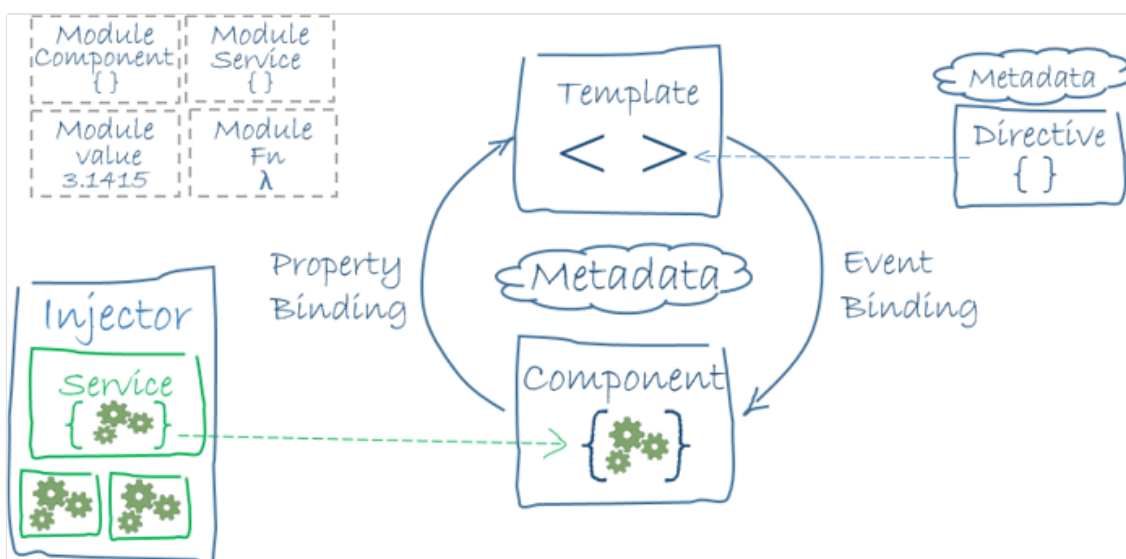
Na prática, o jQuery fornece novas possibilidades de manipulação do DOM e, nesse caso, é utilizado para encontrar um elemento que tenha a classe CSS `conteudo` e esteja dentro de um elemento cujo identificador combina com o da notícia em questão (para ter o conteúdo apresentado ou ocultado).

## Capítulo 2

# Iniciando com o Angular

O **Angular** é um framework para o desenvolvimento de software front-end. Isso quer dizer que utiliza tecnologias padrão do contexto web como HTML, CSS e uma linguagem de programação como JavaScript ou TypeScript (GOOGLE, [s.d.]).

Um software desenvolvido em Angular é composto por diversos elementos como: módulos, componentes, templates e serviços. Esses elementos fazem parte da arquitetura do Angular, que é ilustrada pela figura a seguir.



Essa arquitetura de software orientada a componentes implementa conceitos de dois padrões de arquitetura de software:

- **MVC** (de *Model*, *View*, *Controller*) é um padrão de software que separa a representação da informação (Model) da interação do usuário com ele (View-Controller). Geralmente, Model e Controller são representados por código em linguagem de programação (classes e/ou funções) e View é representado por HTML e CSS (WIKIPEDIA CONTRIBUTORS, 2018a).
- **MVVM** (de *Model*, *View*, *View-Model*) é um padrão de software semelhante ao MVC, com a diferença de que o View-Model utiliza recurso de **data binding** (mais sobre isso depois)

para fazer com que a View seja atualizada automaticamente quando ocorrer uma modificação no Model (WIKIPEDIA CONTRIBUTORS, 2018b).

No contexto do Angular esses elementos são descritos conforme as seções a seguir.

## 2.1 Elementos da Arquitetura do Angular

### 2.1.1 Módulos

Módulos representam a forma principal de modularização de código. Isso significa que um módulo é um elemento de mais alto nível da arquitetura do Angular e é composto por outros elementos, como componentes e serviços.

Um software desenvolvido em Angular possui pelo menos um módulo, chamado **root module** (módulo raiz). Os demais módulos são chamados **feature modules** (módulos de funcionalidades).

### 2.1.2 Bibliotecas

Bibliotecas funcionam como um agrupador de elementos de software desenvolvido em Angular. Bibliotecas oficiais têm o prefixo `@angular`. Geralmente é possível instalar bibliotecas utilizando o **npm** (gerenciador de pacotes do **Node.js**).

Uma biblioteca pode conter módulos, componentes, diretivas e serviços.

### 2.1.3 Componentes

Um componente está, geralmente, relacionado a algo visual, ou seja, uma tela ou parte dela. Nesse sentido, um componente possui código (**Controller**) que determina ou controla o comportamento da interação com o usuário (**View** ou **Template**).

O **Template** determina a parte visual do componente e é definido por código HTML e CSS, além de recursos específicos do Angular, como outros componentes e diretivas.

### 2.1.4 Metadados

Metadados são um recurso do Angular para adicionar detalhes a classes. Isso é utilizado para que o Angular interprete uma classe como um Módulo ou como um Componente, por exemplo.

Os metadados utilizam o conceito de **decorators** do TypeScript, que permitem formas de adicionar informações (metadados) a classes e membros de classe (atributos e métodos) (MICROSOFT, [s.d.]).



### 2.1.5 Data binding

Data binding (que seria algo como “vinculação de dados” em português) é um recurso do Angular que representa um componente importante da sua arquitetura. Considere os seguintes elementos:

- um Model define dados que serão apresentados no Template
- um Template apresenta os dados do Model
- um Controller ou um View-Model determina o comportamento do Template

Se o Controller atualiza o Model, então o Template tem que ser atualizado automaticamente. Se o usuário atualiza o Model por meio do Template (usando um formulário, por exemplo) o Controller também precisa ter acesso ao Model atualizado. O Data Binding atua garantindo que esse processo ocorra dessa forma.

A **sintaxe de data binding** é o mecanismo que controla a ordem entre fonte e destino dos dados:

- **one-way de fonte para view:** pode ser de cinco tipos: **Interpolation**, **Property**, **Attribute**, **Class** e **Style**. A sintaxe de interpolação é `{{expressao}}` para apresentar o valor de `expressao` no Template ou `[target]="expressao"` para que `target` (por exemplo, uma propriedade) receba o valor de `expressao`
- **one-way de view para destino:** pode ser do tipo **Event**. A sintaxe de evento é `(evento)="instrucao"`, indica para o Angular que deve executar `instrucao` no tratador do `evento`
- **two-way:** a sintaxe é `[(alvo)]="expressao"`

### 2.1.6 Diretivas

Diretivas representam um conceito do Angular que é um pouco confuso. Na prática, um Componente é uma Diretiva com um Template. Assim, um Componente é um tipo de Diretiva, que nem sempre está relacionada a algo visual. Angular define dois tipos de diretivas:

- **Diretivas Estruturais:** modificam o Template dinamicamente por meio de manipulação do DOM, adicionando ou removendo elementos HTML
- **Diretivas de Atributos:** também modificam o Template, mas operam sobre elementos HTML já existentes

### 2.1.7 Serviços

Um Serviço é uma abstração do Angular utilizado para isolar a lógica de negócio de Componentes. Na prática, um Serviço é representado por uma classe com métodos que podem ser utilizados em Componentes. Para isso, para que um Componente utilize um serviço, o Angular utiliza o conceito de **Injeção de Dependência** (DI, do inglês **Dependency Injection**). DI é um padrão de software que faz com que dependências sejam fornecidas para quem precisar. Na prática, o Angular identifica as dependências de um Componente e cria automaticamente instâncias delas, para que sejam utilizadas posteriormente no Componente (WIKIPEDIA CONTRIBUTORS, 2018c).

Enquanto esses elementos da Arquitetura do Angular representam conceitos, é importante visualizar a relação deles com elementos práticos do software, ou seja, código. Para isso, a seção a seguir

apresenta a estrutura padrão de um software desenvolvido em Angular.

## 2.2 Estrutura padrão de um software desenvolvido em Angular

Um software desenvolvido em Angular é representado por vários arquivos HTML, CSS, TypeScript e de configuração (geralmente arquivos em formato **JSON**).

```
e2e
├── node_modules
├── src
├── .editorconfig
├── .gitignore
├── angular.json
├── package.json
├── package-lock.json
├── README.md
├── tsconfig.json
└── tslint.json
```

No diretório raiz do software:

- **e2e**: contém especificações de testes **end-to-end**
- **node\_modules**: contém os pacotes (dependências)
- **src**: contém o código-fonte (módulos, componentes etc.)
- **angular.json**: contém configurações do projeto Angular (nome, scripts etc.)
- **package.json**: contém as especificações dos pacotes utilizados no projeto
- **tsconfig.json** e **tslint.json**: contém configurações do processo de tradução de código TypeScript para JavaScript

No diretório **src** também há uma estrutura importante:

```
src
├── app
├── assets
├── environments
├── browserlist
├── favicon.ico
├── index.html
├── karma.conf.js
├── main.ts
├── polyfills.ts
├── styles.css
├── test.ts
├── tsconfig.app.json
├── tsconfig.spec.json
└── tslint.json
```

- `app`: contém o **root module** e os demais **feature modules** do projeto
- `assets`: contém arquivos CSS, JSON e scripts, por exemplo
- `environments`: contém arquivos de configuração do ambiente de desenvolvimento (`environment.ts`) e de produção (`environment.prod.ts`)
- `index.html`: contém o código HTML inicial para o projeto e inclui o componente principal do **root module**
- `main.ts`: contém o código TypeScript necessário para iniciar o software (processo chamado de **Bootstrap**)
- `polyfills.ts`: contém código TypeScript que indica scripts adicionais a serem carregados pelo Browser para funcionamento do software como um todo e para questões de compatibilidade com versões antigas de Browsers
- `style.css`: contém o código CSS para definir estilos globais para o software
- `tsconfig.app.json`: complementa configurações do arquivo `tsconfig.json` específicas para o software em questão

No diretório `app` há os arquivos:

- `app.component.css`, `app.component.html` e `app.component.ts`: definem o component `AppComponent`, respectivamente: apresentação por meio de CSS, **Template** e **Controller**. Basicamente, estes três arquivos formam a base de todo componente
- `app.module.ts`: define o **root module** (`AppModule`)

Esse capítulo apresentou conceitos importantes do Angular. Sempre que necessário, volte a esse capítulo para revisar conceitos do Angular. Muito provavelmente, mesmo desenvolvedores experientes precisem, de tempos em tempos, rever essas definições da arquitetura do Angular.

## Capítulo 3

# Angular CLI

O **Angular CLI** é uma ferramenta para inicializar, desenvolver, criar conteúdo e manter software desenvolvido em Angular (GOOGLE, [s.d.]). A principal utilização dessa ferramenta começa na criação de um projeto. Você pode fazer isso manualmente, claro, mas lidar com todas as configurações necessárias para o seu software Angular pode não ser algo fácil, mesmo se você for um desenvolvedor com nível de conhecimento médio a avançado.

---

### cê-éle-i?

Como parte do vocabulário do Angular, geralmente é interessante pronunciar angular *cê-ele-i*, isso mesmo, CLI é uma sigla para *Command Line Interface* (no português Interface de Linha de Comando). Assim, usar “cli” não é exclusividade do Angular – provavelmente você verá isso em outros projetos.

---

As seções a seguir vão apresentar como instalar e utilizar o Angular CLI para desenvolver software Angular.

## 3.1 Instalando

O Angular CLI é distribuído como um pacote do **Node.JS**, então você não encontrará um instalador, como acontece com software tradicional. Ao invés disso, você precisa de um ambiente de desenvolvimento com o NodeJS instalado (veja Seção [A](#)).

## 3.2 Comandos

O Angular CLI disponibiliza uma série de comandos, que são fornecidos como parâmetros para o programa ng. Além disso, cada comando possui diversas opções (ou **flags**). Faça o exercício de se acostumar a essas opções para aumentar a sua produtividade.

Os principais comandos serão apresentados nas seções seguintes.

### 3.2.1 help

O comando `help` é muito importante porque apresenta uma documentação completa do Angular CLI, ou seja, todos os comandos e todas as suas opções.

#### Exemplo: documentação completa

```
$ ng help
```

Essa linha de comando apresenta todos os comandos e todas as suas opções.

#### Exemplo: documentação do comando `new`:

```
$ ng help new
```

Essa linha de comando apresenta apenas as opções do comando `new`.

As opções de cada comando são sempre precedidas de `--` (dois traços). Algumas opções aceitam um valor e geralmente possuem um valor padrão.

#### Exemplo: comando `new` com duas opções

```
$ ng new escola-app --skip-install --routing true --minimal true
```

Essa linha de comando fornece três opções para o comando `new`: `--skip-install`, `--routing` (que recebe o valor `true`) e `--minimal` (que recebe o valor `true`).

### 3.2.2 new

O comando `new` é utilizado para criar um projeto (um software Angular). Exemplo:

```
$ ng new escola-app
```

Quando a linha de comando do exemplo for executada será criado um software Angular chamado **escola-app** e estará em um diretório chamado `escola-app`, localizado a partir de onde a linha de comando estiver sendo executada.

A parte mais interessante de criar um projeto Angular com esse comando é que ele cria todos os arquivos necessários para um software bastante simples, mas funcional.

Para cada comando do Angular CLI é possível informar opções. As opções mais usadas do comando `new` são:

- `--skip-install`: faz com que as dependências não sejam instaladas

- `--routing`: se for seguida de `true`, o comando cria um módulo de rotas (isso será visto em outro capítulo)

### 3.2.3 generate

O comando `generate` é utilizado para criar elementos em um projeto existente. Para simplificar, é bom que o Angular CLI seja executado a partir do diretório do projeto a partir de agora. Por meio desse comando é possível criar:

- class
- component
- directive
- enum
- guard
- interface
- module
- pipe
- service

Como acontece com outros comandos, o `generate` também pode ser utilizado por meio do atalho `g`.

#### Exemplo: criar component ListaDeDisciplinas

```
$ ng generate component ListaDeDisciplinas
```

Essa linha de comando criará o diretório `./src/app/lista-de-disciplinas` e outros quatro arquivos nesse mesmo local:

- `lista-de-disciplinas.component.css`
- `lista-de-disciplinas.component.html`
- `lista-de-disciplinas.component.spec.ts`
- `lista-de-disciplinas.component.ts`

É importante notar que esses arquivos não estão vazios, mas já contêm código que mantém o software funcional e utilizável. Por isso o Angular CLI considera que as opções (class, component etc.) são como **blueprints** (ou plantas, em português). Outra coisa importante é que o Angular CLI também modificará o arquivo `./src/app/app.module.ts` se necessário (ou outro arquivo que represente um módulo que não seja o **root module** – mais sobre isso depois).

### 3.2.4 serve

O comando `serve` compila o projeto e inicia um servidor web local. Por padrão o servidor web executa no modo `--watch`, que reinicia o comando novamente, de forma incremental, sempre que houver uma alteração em algum arquivo do projeto, e `--live-reload`, que recarrega o projeto no browser (se houver uma janela aberta) sempre que houver uma mudança.

### 3.2.5 build

O comando `build` compila o projeto, mas não inicia um servidor web local. Ao invés disso, gera os arquivos resultantes da compilação em um diretório indicado.

## 3.3 Criando e executando um projeto Angular

Para iniciar, execute o comando `new` do **Angular CLI** para criar um projeto. Exemplo:

```
$ ng new angular-hello-world
```

Aguarde alguns instantes para a conclusão das instalações dos pacotes e configuração inicial do projeto. O comando vai criar a pasta `angular-hello-world`. Execute os comando seguintes dentro da pasta do projeto.

O software **noticias-js** pode ser executado no browser diretamente ao abrir o arquivo `index.html`, disso você já sabe. O que muda em relação a como executar um software angular?

Por causa da estrutura e da arquitetura do Angular a execução não pode ser feita abrindo o arquivo `src/index.html` no browser, pois será necessário um **servidor web**. O **Angular CLI** já inclui um **servidor web local** ou **servidor web de desenvolvimento**, que pode ser iniciado com o comando:

```
$ npm start
```

Esse não é bem um comando do **Angular CLI**, mas um **script** declarado no arquivo `package.json`:

```
1 {
2   "name": "angular-hello-world",
3   "version": "0.0.0",
4   "scripts": {
5     "ng": "ng",
6     "start": "ng serve",
7     "build": "ng build",
8     "test": "ng test",
9     "lint": "ng lint",
10    "e2e": "ng e2e"
11  },
12  ...
13 }
```

O script `start` executa o comando `serve` do **Angular CLI**. Então, na prática, o script é um *atalho* e, por isso, há mais de uma maneira de iniciar o software.

Ao executar o script será apresentada uma saída que indica que você deve acessar a URL `http://localhost:4200` no browser. Faça isso para ver uma imagem semelhante à seguinte:

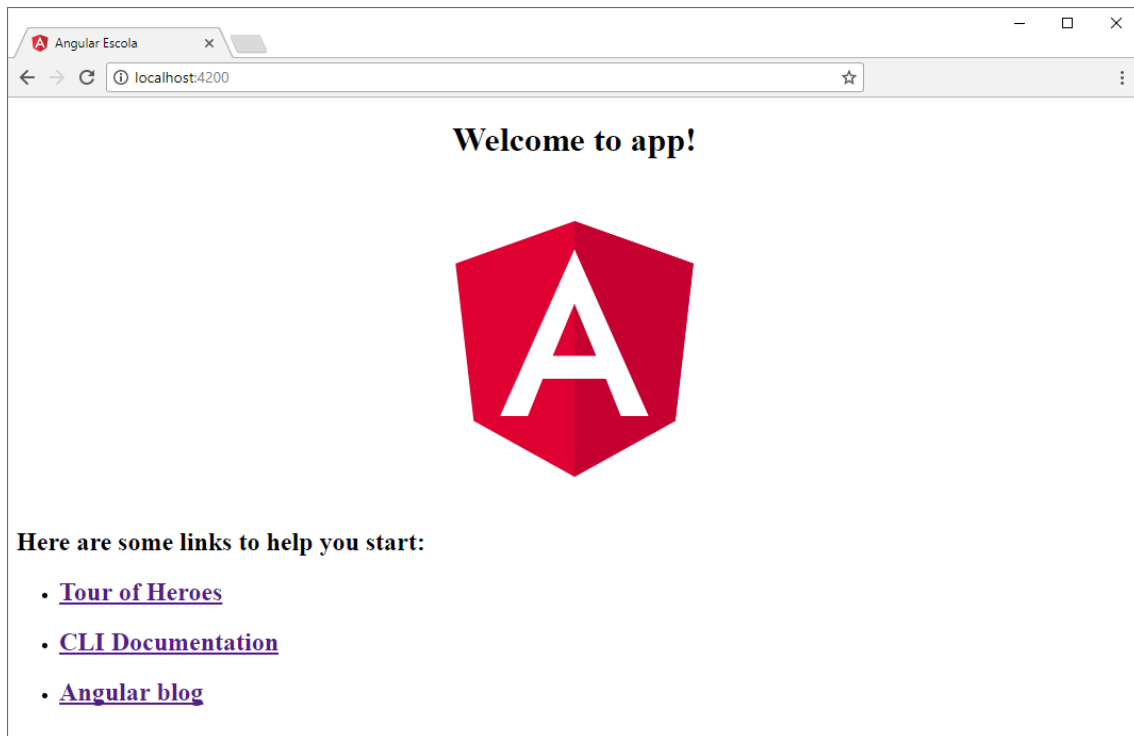


Figura 3.1: Versão inicial do software em execução no Browser

A URL `http://localhost:4200` identifica três elementos importantes:

- uso do protocolo HTTP
- o **host** é identificado por `localhost` (ou seja, a máquina local e, por isso o termo “servidor web local”)
- a porta é 4200

Veremos mais sobre esses e outros comando no decorrer do livro.

## 3.4 O arquivo `package.json`

O arquivo `package.json` contém informações do projeto, scripts e dependências. O arquivo criado pelo **Angular CLI** tem mais ou menos o seguinte conteúdo:

```
1 {  
2   "name": "angular-hello-world",  
3   ...  
4   "scripts": {  
5     ...  
6   },  
7   "private": true,  
8   "dependencies": {  
9     ...  
10  },
```



```
11   "devDependencies": {  
12     ...  
13   }  
14 }
```

Por ser um conteúdo em formato JSON, o interpretamos da seguinte forma:

- **name**: define o nome do projeto para o **Node.JS** (nenhuma relação com o Angular)
- **scripts**: contém scripts que podem ser executados no prompt (exemplos: `npm start` e `npm run build`)
- **dependencies**: define as dependências do projeto (pacotes, e suas versões, que são incluídos na compilação)
- **devDependencies**: define as dependências de desenvolvimento (pacotes que não têm código-fonte incluído na compilação)

---

### Espera! Você disse compilação?

Como já disse, você pode utilizar JavaScript ou TypeScript ao desenvolver projetos Angular. Isso não é bem assim. Na prática, geralmente você encontrará a recomendação (senão uma ordem) de utilizar TypeScript. O problema é que seu Browser não entende TypeScript, por isso é necessário um processo de tradução do código TypeScript para JavaScript. E não é só isso. Há vários recursos utilizados no projeto Angular criado com TypeScript que precisam de ajustes para que funcionem no seu Browser.

Por isso os comandos `serve` e `build` são tão importantes e – por que não dizer? – **browser-friendly** ⇒

---

## 3.5 Estrutura do projeto

Seção 2 apresentou a estrutura de um projeto Angular. De forma visual, a estrutura do projeto criado até o momento é ilustrada pela Figura 3.2.

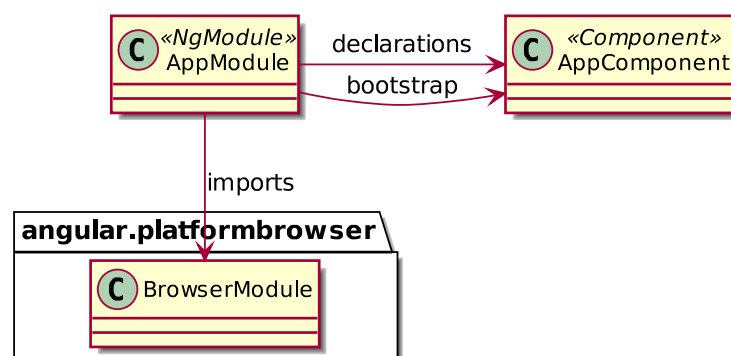


Figura 3.2: Estrutura básica de um projeto Angular

As relações entre as classes merecem destaque, começando por `AppModule`.

### 3.5.1 AppModule

A classe `AppModule` representa o **root module** do projeto e isso é especificado no arquivo `src/main.ts`:

```
1  ...
2  import { AppModule } from './app/app.module';
3  ...
4  platformBrowserDynamic().bootstrapModule(AppModule)
5    .catch(err => console.log(err));
```

A linha 4 contém uma chamada para o método `bootstrapModule()` com argumento `AppModule`, o que indica para o Angular que ele deve usar esse módulo (`AppModule`) na execução inicial (**Bootstrap**). A partir de então os recursos do módulo poderão ser utilizados.

Código-fonte 3.1 apresenta a classe `AppModule`.

Código-fonte 3.1: Código-fonte do AppModule no software noticias-angular

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppComponent } from './app.component';
5
6  @NgModule({
7    declarations: [
8      AppComponent
9    ],
10   imports: [
11     BrowserModule
12   ],
13   providers: [],
14   bootstrap: [AppComponent]
15 })
16 export class AppModule { }
```

A classe `AppModule` é declarada usando o decorator `@NgModule` e entre as linhas 6 e 16 há a utilização desse recurso para informar os metadados:

- `declarations`: a lista dos componentes declarados (ou pertencentes) a o módulo
- `imports`: a lista dos módulos importados
- `providers`: a lista de serviços declarados (vazia, por enquanto)
- `bootstrap`: define o **root component**

Em resumo, o `AppModule` é um módulo do Angular que declara o componente `AppComponent`, importa o módulo `BrowserModule` (do pacote `@angular/platform-browser`) e usa `AppComponent` como **root component**.

O **Angular** adota uma nomenclatura para o arquivo de módulo: `<módulo>.module.ts`. Assim, podemos usar o termo “módulo App” e saber do que se trata.

### 3.5.2 AppComponent

A classe `AppComponent` representa o **root component** do projeto e está declarada no arquivo `src/app/app.component.ts` cujo conteúdo é o seguinte:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'Angular';
10 }
```

A classe `AppComponent` declara o atributo `title` e o código também utiliza um **decorator**, dessa vez o `@Component` que também adiciona metadados à classe:

- `selector`: representa o seletor CSS para que o Angular identifique onde (em um Template) o componente deve ser apresentado
- `templateUrl`: o caminho do arquivo usado como Template (HTML)
- `styleUrls`: a lista de caminhos de arquivos usados como estilo (CSS)

Os metadados declaram, também, a estrutura básica de um Componente:

- **Controller**: a classe, em si
- **Template**: um arquivo HTML que contém marcação e representa a **View**

O **Angular** adota uma nomenclatura para facilitar a localização desses arquivos:

- `<componente>.component.ts` representa o Controller
- `<componente>.component.html` representa o Template
- `<componente>.component.css` representa o arquivo CSS (complemento para o Template)

Assim podemos usar o termo “componente App” e saber que se trata do conjunto de arquivos que declaram esse componente.

O Template desse componente (o arquivo `src/app.component.html`) tem conteúdo que parece HTML. Veja o trecho:

```
1 <h1>
```

```
2 Welcome to {{ title }}!  
3 </h1>
```

Por que eu disse “parece HTML”? Porque isso realmente não é HTML, é a **linguagem de Template do Angular**, que herda características do HTML. Nesse trecho, inclusive, você pode perceber que é realmente semelhante, só diferenciando na linha 2, que traz um conteúdo entre {{ e }}. Essa sintaxe entre chaves duplas é chamada **interpolação** e é utilizada para apresentar valores declarados no Controller por meio do recurso de **data binding**.

Você já viu que a classe `AppComponent` declara um atributo `title`, então o conteúdo do Template instrui o Angular a substituir o trecho `Welcome to {{ title }}` por `Welcome to Angular!`. Nesse ponto vemos, na prática, mais um elemento da arquitetura MCV, o **Model**.

O Model não é um arquivo, é mais um conceito que permite o **data binding**, ou seja, passar dados entre Template e Controller. A Figura 3.3 ilustra esse conceito.

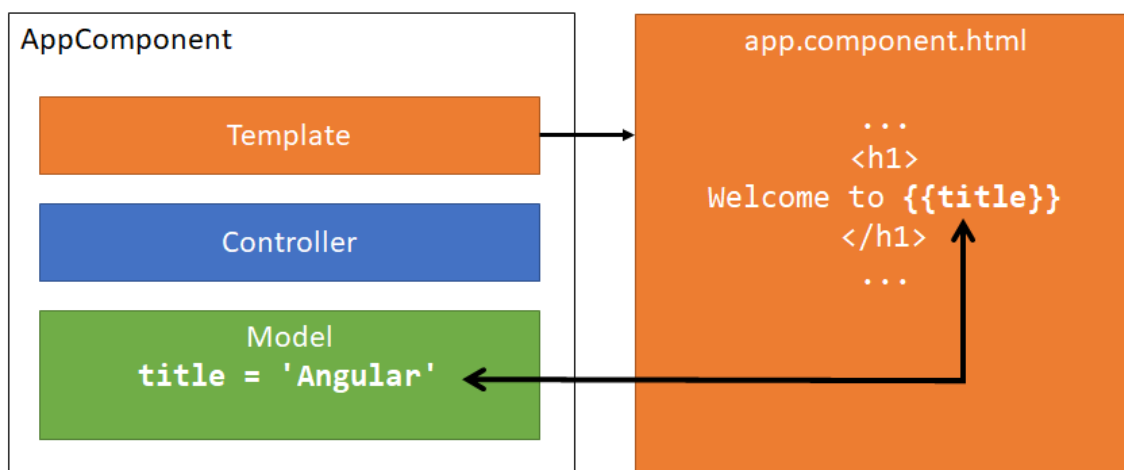


Figura 3.3: Relação entre Model-Template-Controller no AppComponent

Aqui aprendemos algo um conceito importante: o **Angular interpreta o Template**. Isso significa que todo o seu conteúdo, HTML e CSS, é interpretado pelo Angular antes de ser entregue para o browser. É isso que faz com que o recurso de interpolação funcione. É por isso, também, que Data binding é tão importante no Angular. Ele é responsável por fazer com que o atributo `title`, que compõe o Model, esteja disponível para ser usado no Template. Além disso, qualquer alteração no valor desse atributo representará uma alteração na visualização do componente no browser.

---

### MVC ou MVVM?

Se você se lembrar da discussão sobre Angular ser MVC ou MVVM é aqui que o entendimento pesa a favor do segundo. O Controller não está desassociado do Model e, por causa do Data binding, sua função inclui informar o Template de que ocorreu uma alteração no Model, de forma que ele seja atualizado.

---

É isso, o Model é um conceito abstrato e está, geralmente, no Controller, representado por atributos

e métodos – sim, também é possível mostrar o valor retornado por um método no Template (mais sobre isso depois).

### 3.5.3 index.html

Para finalizar essa seção falta entender como o `AppComponent` é, enfim, apresentado. Para isso, veja o arquivo `src/index.html`:

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4 ...
5 </head>
6 <body>
7   <app-root></app-root>
8 </body>
9 </html>
```

Esse arquivo também é um Template, mas não está associado a um componente. O arquivo `angular.json` (por volta da linha 17) é que define que esse arquivo é chamado inicialmente durante a execução do software no servidor web local.

A parte HTML é fácil de entender, bem como saber que o elemento `app-root` não faz parte do HTML. Então, de onde vem esse elemento? Voltando um pouco na declaração do `AppComponent` lembramos que há um metadado chamado `selector`, que tem justamente o valor `'app-root'`. Por ser um seletor de elemento a instrução para o Angular, ao processar o `src/index.html` é: onde encontrar o elemento `app-root` apresente o componente `AppComponent`. Assim, as coisas ficam mais ou menos como ilustra a Figura 3.4.

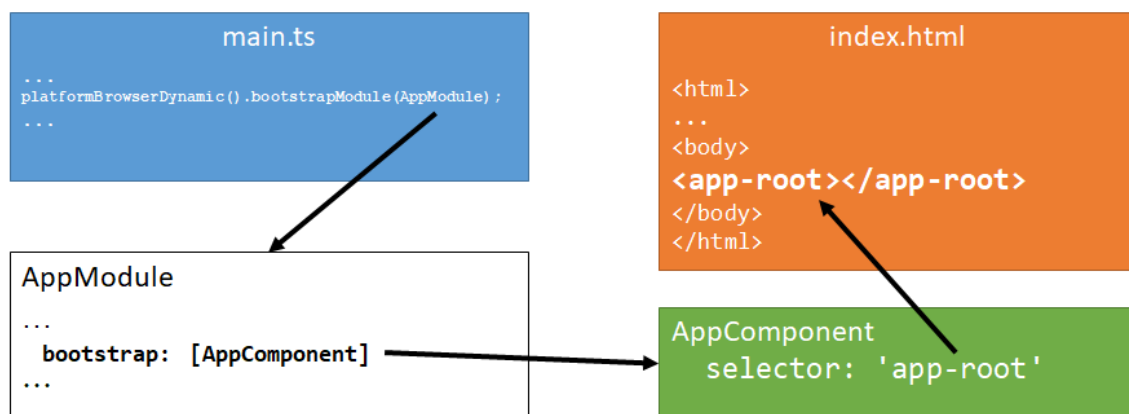


Figura 3.4: Execução do Template inicial

Viu? Não é mágica. É software!

## 3.6 O que vem a seguir?

Como agora você já deve estar entendendo melhor o funcionamento do Angular e o código-fonte do projeto, é hora de sujar as mãos =) No capítulo seguinte você vai implementar o software **noticias-angular**.

## Capítulo 4

# Gerenciador de notícias com Angular

O **Capítulo 1** apresentou conceitos básicos de desenvolvimento web e o software **noticias-js**, um projeto de gerenciador de notícias que é implementado utilizando tecnologias front-end (HTML, CSS e JavaScript). Este capítulo apresenta o **noticias-angular**, basicamente uma outra versão do **noticias-js**, mas que é implementado utilizando **Angular**, com o objetivo de apresentar os recursos desta plataforma de desenvolvimento.

Este capítulo apresenta:

- estrutura de um componente (Controller e Template)
- estrutura completa de um projeto integrando módulo e componente
- linguagem de template, variável de template e data binding
- diretivas estruturais: **NgIf** e **NgForOf**
- diretivas de atributos para lidar com propriedades (data binding)
- diretivas de atributos para lidar com eventos (data binding)
- diretivas de atributos para lidar com formulários (two-way data binding)

Os próximos capítulos apresentarão outras versões deste software ou outros softwares, quando necessário, para que seja possível conhecer os recursos do Angular.

### 4.1 Clonar e executar localmente

Este capítulo está diretamente relacionado com o software **noticias-angular**, branch “início”<sup>1</sup>. Antes de prosseguir, clone e execute o projeto localmente. A interface do software é idêntica à do **noticias-js**. Verifique.

---

<sup>1</sup>O repositório do branch “iniciando” está disponível em <https://github.com/jacksongomesbr/webdevbook-noticias-angular/tree/inicio> e você pode vê-lo em execução acessando <https://stackblitz.com/github/jacksongomesbr/webdevbook-noticias-angular/tree/inicio>

## 4.2 O projeto

O **noticias-angular** foi criado com o comando `new` do **Angular CLI**:

```
$ ng new webdevbook-noticias-angular
```

A estrutura do projeto muda em relação ao projeto básico criado pelo **Angular CLI** (Seção 3.3) em alguns aspectos. Assim, o software será apresentado a partir da Figura 4.1, que apresenta a sua estrutura.

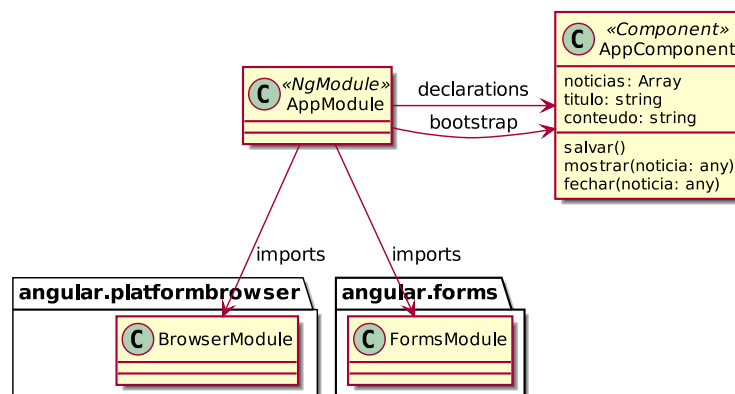


Figura 4.1: Estrutura do projeto angular-noticias

As relações entre os componentes são as mesmas apresentadas na Seção 3.5, com o acréscimo da importação do módulo `FormsModule`, que é necessário para que o projeto possa utilizar os recursos do Angular que permitem o usuário entrar dados por meio de formulários.

### 4.2.1 AppComponent

As maiores modificações estão no `AppComponent` e detalhes são apresentados nas seções seguintes.

#### 4.2.1.1 Controller

Um trecho do Controller do `AppComponent` é apresentado a seguir:

```

1  ...
2  @Component({
3      ...
4  })
5  export class AppComponent {
6      noticias = [];
7      titulo = null;
8      conteudo = null;
9
10     salvar() {

```



```
11     ...
12   }
13
14   mostrar(noticia) {
15     ...
16   }
17
18   fechar(noticia) {
19     ...
20   }
21 }
```

A classe declara três atributos:

- `noticias`: um `Array` que contém a lista de notícias
- `titulo`: representa o título da notícia cadastrada (vinculada ao formulário)
- `conteudo`: representa o conteúdo da notícia cadastrada (vinculada ao formulário)

Há também três métodos:

- `salvar()`: obtém os dados do formulário (título e conteúdo da notícia) e insere na lista de notícias
- `mostrar()`: apresenta o conteúdo de uma notícia (informada como parâmetro `noticia`)
- `fechar()`: oculta o conteúdo de uma notícia (informada como parâmetro `noticia`)

Em detalhes, o método `salvar()`:

```
1  salvar() {
2    const noticia = {
3      id: this.noticias.length,
4      titulo: this.titulo,
5      conteudo: this.conteudo,
6      visivel: false
7    };
8    this.noticias.push(noticia);
9    this.titulo = null;
10   this.conteudo = null;
11 }
```

O método `salvar()` contém código que realiza o seguinte:

- cria um objeto (`noticia`) com quatro atributos: `id`, `titulo`, `conteudo` e `visivel`. Os valores dos atributos `titulo` e `conteudo` são obtidos dos respectivos atributos (note o uso de `this`) e o valor do atributo `visivel` é `false`
- adiciona o objeto `noticia` na lista `noticias` por meio do método `push()`
- redefine valores dos atributos `titulo` e `conteudo` (atribuindo o valor `null`)

Os métodos `mostrar()` e `fechar()` são bem semelhantes:

```

1  mostrar(noticia) {
2      noticia.visivel = true;
3  }
4
5  fechar(noticia) {
6      noticia.visivel = false;
7  }

```

Os métodos recebem o parâmetro `noticia`, um objeto que representa uma notícia, e alteram o valor do atributo `visivel`:

- `true`: mostrar o conteúdo
- `false`: ocultar o conteúdo

#### 4.2.1.2 Template

O Template do `AppComponent` contém três partes:

- cabeçalho
- área da lista de notícias
- área do cadastro de notícias

O seguinte trecho do Template mostra onde estão essas partes:

```

1  <h1>Gerenciador de notícias</h1>
2  <h2>Notícias recentes</h2>
3  ...
4  <h2>Cadastrar notícia</h2>
5  ...

```

Primeiro, a área da lista de notícias:

```

1  <h2>Notícias recentes</h2>
2  <p>Clique no título da notícia para expandir</p>
3  <div>
4      <ul>
5          <li *ngFor="let noticia of noticias">
6              <p (click)="mostrar(noticia)">{{ noticia.titulo }}</p>
7              <p *ngIf="noticia.visivel">
8                  {{ noticia.conteudo }}
9                  <br> -----
10                 <br>
11                 <button (click)="fechar(noticia)">Fechar</button>
12             </p>
13         </li>
14     </ul>
15 </div>

```

O elemento `li` da linha 5 tem o atributo `*ngFor`, que representa a **diretiva estrutural** `NgForOf`, uma **diretiva de repetição** que insere elementos no DOM conforme uma expressão de iteração com a sintaxe: `let variavel of lista`. Nesse caso `variavel` é chamada de **variável de template** porque só existe no contexto do Template. A diretiva `NgForOf` instrui o Angular a repetir uma parte do template para da item de uma iteração. O valor `let noticia of noticias` é interpretado assim:

Para cada notícia na lista `noticias`, crie uma variável de template chamada `noticia` e torne-a disponível para cada iteração (repetição) do template contido em `li`.

Assim, o conteúdo do `li` é repetido para cada item da lista `noticias` (atributo do Controller). O `li` contém dois elementos `p` que são usados para propósitos diferentes. O primeiro `p`, com atributo (`click`), apresenta o título da notícia usando **interpolação**. Perceba que o Angular interpreta a variável de template `noticia` seguindo o determinado no Controller (notícia tem o atributo `título`).

Outra parte importante tem relação com o atributo (`click`), que usa a **sintaxe de evento** (Seção 2.1.5). Nesse caso, o evento é `click` e o conteúdo do atributo, `mostrar(noticia)`, instrui o Angular a chamar o método `mostrar()` (definido no Controller) passando como argumento a variável de template `noticia`, para que seu conteúdo seja apresentado. Perceba a interação com os métodos declarados no Controller e a utilização do **paradigma da Orientação a Objetos**.

O outro elemento `p` tem o atributo `*ngIf`, que representa a diretiva estrutural `NgIf`, uma **diretiva condicional**, que insere e remove elementos no DOM conforme uma expressão booleana. O valor `noticia.visible` é interpretado assim:

Se o valor do atributo `visivel` da variável de template `noticia` for `true`, então insere o elemento `p` no DOM, caso contrário, remove o elemento `p`.

Assim, se o atributo `visivel` da variável de template `noticia` for `true`, seu conteúdo será visível.

Para finalizar, há um elemento `button` que também tem um atributo (`click`) com valor `fechar(noticia)`, o que instrui o Angular a chamar o método `fechar()` passando como argumento a variável de template `noticia`, para que seu conteúdo seja ocultado.

---

### Diretivas manipulam DOM?

O projeto `noticias-js` usa DOM (diretamente ou por meio do jQuery) para manipulação do DOM. De forma simplificada o Angular faz a mesma coisa usando diretivas, como `NgForOf` e `NgIf`.

---

A outra parte importante do Template tem a ver com o formulário de cadastro:

```
1 <h2>Cadastrar notícia</h2>
2 <form #cadastro="ngForm" (submit)="salvar()">
3   <div>
4     <label for="titulo">Título</label>
5     <input type="text" id="titulo" name="titulo" [(ngModel)]="titulo"
      required>
```

```
6     </div>
7     <div>
8         <label for="conteudo">Conteúdo</label>
9         <textarea id="conteudo" name="conteudo" cols="80" rows="5" [(ngModel)]="conteudo" required></textarea>
10    </div>
11    <div>
12        <button type="submit" [disabled]="!cadastro.form.valid">Salvar</button>
13        <button type="reset" formnovalidate>Limpar</button>
14    </div>
15 </form>
```

Primeiro, ao utilizar o elemento `form` o Angular cria automaticamente uma diretiva `NgForm`, que determina o funcionamento do formulário a partir de então (campos, validação etc.). Se você precisar acessar informações do formulário poderá usar uma variável de template. Por isso o `form` tem o atributo `#cadastro` com valor `ngForm`.

Além disso o `form` também usa a sintaxe de evento para chamar o método `salvar()` no `submit` (também pode ser usado o nome `ngSubmit`). O comportamento tradicional de um formulário HTML é que seja submetido quando um `button` for clicado. No Angular, por causa da diretiva `NgForm`, o processo é modificado para permitir a interação com o Controller.

No final do `form` há o `button` “Salvar”, que tem o atributo `[disabled]`, uma **diretiva de atributo** usada para desabilitá-lo com base em uma expressão booleana. Assim, a expressão `!cadastro.form.valid` acessa a informação de validação global do formulário e, se for `true` (formulário não está válido), desabilita o botão (não permite que ele seja clicado). A validação continua utilizando os recursos do HTML (atributo `required`, por exemplo, em cada campo de preenchimento obrigatório).

Os campos do formulário continuam utilizando elementos do HTML (`input` e `textarea`), mas agora também utilizam **two-way data binding**:

```
1 ...
2 <input type="text" id="titulo" name="titulo" [(ngModel)]="titulo" required
3 ...
4 <textarea id="conteudo" name="conteudo" cols="80" rows="5" [(ngModel)]="
5 ...
```

A sintaxe de two-way data binding é utilizada para vincular um campo de formulário a uma expressão (geralmente um atributo declarado no Controller). Isso acontece no `input` por meio de `[(ngModel)]="titulo"`, vinculando o seu valor ao atributo `titulo`, e no `textarea`, por meio de `[(ngModel)]="conteudo"`, vinculando seu valor ao atributo `conteudo`. Isso significa que qualquer alteração em um dos campos gera uma alteração no atributo vinculado, e vice-versa.

### 4.3 Depuração de código no browser

Os browsers fornecem ferramentas para depuração de código HTML, CSS e JavaScript por meio das “Ferramentas do desenvolvedor”<sup>2</sup>. A Figura 4.2 ilustra uma tela do browser executando o **angular-noticias** e com as ferramentas do desenvolvedor ativas.

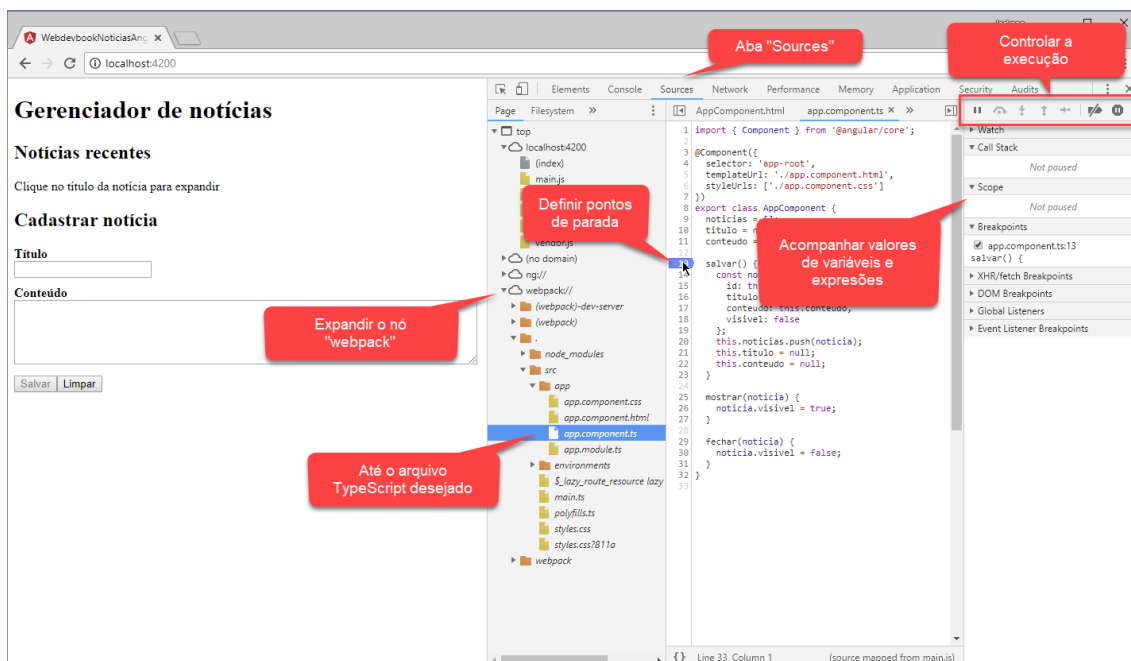


Figura 4.2: Janela do browser com ferramentas de desenvolvedor habilitadas

A Figura 4.2 ilustra que as ferramentas do desenvolvedor são organizadas em abas, sendo a aba “Sources” a de maior interesse para a depuração de código. Do lado esquerdo da conteúdo da aba “Sources” há um navegador de arquivos, sendo o nó “webpack://” o mais importante. Dentro dele estão os arquivos utilizados no servidor web local de desenvolvimento. Para depurar o `AppComponent`, por exemplo, deve-se navegar até o nó “app.component.ts” (a hierarquia completa seria “webpack://./src/app/app.component.ts”). Perceba a relação com o sistema de arquivos local.

Mais ao centro a Figura 4.2 mostra o código-fonte do componente. Podem ser adicionados e removidos os pontos-de-parada (*breakpoints*) que servem para controlar as paradas de execução passo-a-passo do código. A figura ilustra um breakpoint na linha 13, o que significa que a execução do código será interrompida (pausada) quando método `salvar()` for executado.

Na parte superior, mais à direita, há uma barra de ferramentas com botões, nesta ordem:

- pausar/continuar a execução do código
- executar a próxima linha e não entrar no código da função/método
- executar a próxima linha e entrar no código da função/método
- executar a próxima linha e sair do código da função/método

<sup>2</sup>O Chrome chama de “Ferramentas do desenvolvedor”, enquanto o Firefox chama de “Web Developer” e o Edge chama de “Ferramentas de desenvolvedor”. Os nomes são semelhantes e as funcionalidades são parecidas. Reserve um tempo para aprender a lidar com cada browser da sua escolha.

- desativar os breakpoints
- pausar em exceções

Por fim, o lado direito apresenta painéis que permitem inspecionar valores de variáveis e expressões em tempo real.

Essas ferramentas permitem um acompanhamento mais próximo da execução do código sem a necessidade de utilizar, por exemplo, o Console do browser.

## 4.4 Conclusão

Este capítulo apresentou o software **noticias-angular** e sua estrutura. Os destaques são:

- importar o módulo `FormsModule` no **root module**
- uso das diretivas estruturais `NgForOf` e `NgIf`
- uso de data-binding (interpolação, evento, propriedade e vinculação de campo do formulário com atributo do Controller)
- utilizar o browser como ferramenta de depuração de código

## Capítulo 5

# CRUD completo do noticias-angular

O Capítulo 4 apresenta uma implementação do **noticias-angular**, um software de gerenciamento de notícias desenvolvido como um projeto Angular. Este capítulo tem o objetivo de apresentar uma nova versão deste projeto que traz novas funcionalidades:

- listar notícias recentes de forma ordenada
- apresentar notícia para leitura
- listar todas as notícias (com pesquisa)
- cadastrar notícia
- editar notícia
- excluir notícia

Este capítulo apresenta:

- como criar uma interface CRUD baseada em lógica de troca de visibilidade de telas
- como não utilizar apenas o Controller para lógica de negócio
- como representar a navegação de telas usando um Diagrama de Estados da UML (como um diagrama de navegação)
- o software **compodoc**, utilizado para gerar uma documentação do código
- como utilizar métodos importantes da classe `Array`
- como funciona a classe `Date`, alguns dos seus métodos importantes e como o browser representa uma data no campo de formulário
- recurso de **Pipes** do Angular

## 5.1 Clonar e executar localmente

Este capítulo está relacionado com o software **noticias-angular**, branch “crud-telas”<sup>1</sup>. Antes de prosseguir, clone e execute o projeto localmente.

## 5.2 CRUD

**CRUD** é um acrônimo bastante conhecido em desenvolvimento de software e significa:

- **C - CREATE**: funcionalidade de cadastrar
- **R - RETRIEVE**: recuperar (consultar)
- **U - UPDATE**: atualizar
- **D - DELETE**: deletar (excluir)

Um software que fornece funcionalidades CRUD, geralmente, proporciona uma interface padronizada, que contém:

- uma tela que apresenta a lista de todos os registros de dados (elementos ou objetos) contendo, geralmente, funcionalidade de pesquisa/busca
- uma tela que permite cadastrar
- uma tela que permite editar (geralmente semelhante à tela de cadastro ou usando a mesma tela para um propósito diferente)
- e, opcionalmente, uma tela que permite acessar os detalhes do registro de dados de forma apenas para leitura

Esses recursos são implementados no branch “crud-telas” do **noticias-angular** e são apresentados neste capítulo.

## 5.3 Documentação do projeto

O capítulo atual utiliza o software **compodoc**, que interpreta o código-fonte do projeto e gera uma documentação que pode ser disponibilizada on-line (OGLOBLINSKY, 2018). Há, inclusive, uma versão on-line da documentação do projeto **noticias-angular**<sup>2</sup>.

A documentação de um projeto é um passo importante na maturidade do código e do programador (senão de todos os envolvidos no projeto). Documentar código é uma boa prática de desenvolvimento de software porque o desenvolvedor precisa estabelecer uma descrição clara e concisa de cada pequeno elemento do software, desde atributos a métodos. O **compodoc** adiciona recursos nesse sentido porque ele é voltado para projetos Angular e, por isso, gera uma documentação com base nessa estrutura, por exemplo, interpretando módulos e componentes.

<sup>1</sup>O repositório do branch “crud-telas” está disponível em <https://github.com/jacksongomesbr/webdevbook-noticias-angular/tree/crud-telas> e você pode vê-lo em execução acessando <https://stackblitz.com/github/jacksongomesbr/webdevbook-noticias-angular/tree/crud-telas>

<sup>2</sup>A versão on-line da documentação do **noticias-angular** pode ser acessada em <https://jacksongomesbr.github.io/webdevbook-noticias-angular/>. Entretanto, importante notar, a versão on-line é sincronizada com o branch “master”, o que pode torná-la diferente do exposto neste capítulo



Procure manter essa prática de documentar seus códigos-fontes.

## 5.4 O projeto

O branch “crud-telas” do **noticias-angular** não é muito diferente do branch “inicio” em termos de estrutura, como ilustra a Figura 5.1, mas muda bastante em termos de funcionalidades.

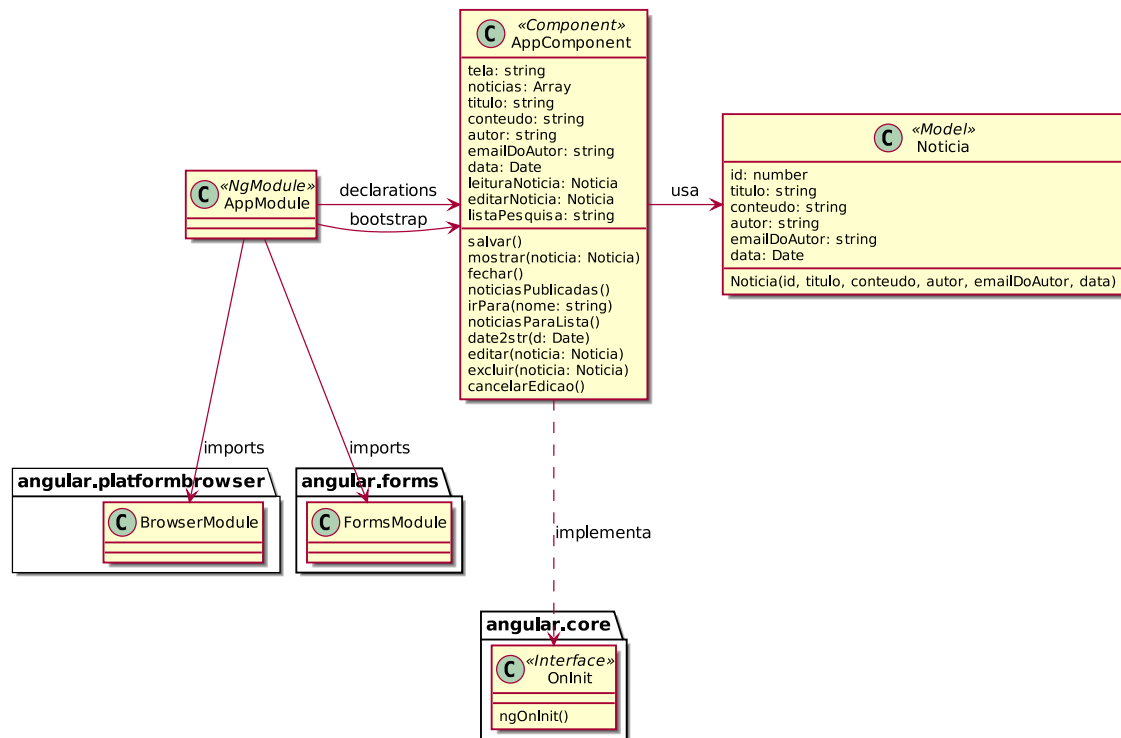


Figura 5.1: Estrutura do projeto angular-noticias, branch crud-telas

A estrutura inclui a classe **Noticia**, que é utilizada no **AppComponent**. As seções seguintes continuam apresentando os elementos dessa estrutura.

## 5.5 Classe Noticia

Uma boa prática de programação TypeScript é utilizar organizar o código, declarar tipos de dados e, para isso, o branch “crud-telas” implementa a classe **Noticia**, no arquivo **src/app/noticia.model.ts**. Não há algo de especial em o nome do arquivo conter “model” e também não é uma recomendação do Angular. Entretanto, escolhi manter o nome dessa forma para seguir a mesma lógica de organização do projeto que já vem sendo utilizada.

Assim, a classe **Noticia** implementa a classe que representa o Model de Notícia e é utilizada no **AppComponent**. A implementação demonstra algumas práticas interessantes, como tratar os dados no construtor e fornecer um método que verifica se a notícia está ou não publicada (ao invés de fazer isso no **AppComponent**). Essa é uma prática interessante para organizar o código-fonte do projeto.

Duas partes importantes do código estão no construtor e no método `estahPublicada()`. Primeiro, o construtor:

```
1  constructor(id: number, titulo: string, conteudo: string, autor: string,
    emailDoAutor: string, data: Date) {
2      this.id = id;
3      this.titulo = titulo;
4      this.conteudo = conteudo;
5      this.autor = autor;
6      this.emailDoAutor = emailDoAutor;
7      if (data) {
8          if (typeof data === 'string') {
9              this.data = new Date(data);
10         } else {
11             this.data = data;
12         }
13     } else {
14         this.data = null;
15     }
16 }
```

O código do construtor realiza um tratamento dos valores dos parâmetros a partir da linha 7, mais especificamente, verificando se o tipo de dados é `string` (quando chama o construtor da classe `Date` para fazer um parser da entrada e criar um objeto) e se o valor é `null`. Voltaremos a falar desse tratamento na seção que apresenta a funcionalidade de cadastrar notícia.

Outra parte importante da classe `Noticia` é o método `estahPublicada()`:

```
1  estahPublicada(): boolean {
2      if (!this.data) {
3          return false;
4      }
5      const dataAtual = new Date();
6      if (this.data > dataAtual) {
7          return false;
8      }
9      return true;
10 }
```

O código do método `estahPublicada()` considera que:

- se a data da notícia não está definida, então não está publicada (linhas 2 e 3)
- se a data da notícia é maior que a data atual, então a notícia não está publicada (linhas 5-8)
- caso contrário, a notícia está publicada

É importante notar a utilização da classe `Date` nesses códigos. `Date` faz parte da referência de objetos do JavaScript (W3SCHOOLS, [s.d.]) e também está disponível no TypeScript.

## 5.6 AppComponent

O Controller do `AppComponent` possui atributos e métodos para implementar as funcionalidades necessárias, que são apresentadas detalhadamente a seguir.

### 5.6.1 Interface OnInit

O pacote `@angular/core` fornece a interface `OnInit`, que disponibiliza o método `ngOnInit()`. Esse método funciona como um evento que é executado quando o componente (que implementa a interface) é criado pelo Angular. Assim, o Controller fornece uma implementação para o método `ngOnInit()`:

```
1  ngOnInit() {  
2      this.noticias.push(new Noticia(  
3          this.noticias.length,  
4          'Teste',  
5          'Conteúdo da notícia',  
6          'João Silva',  
7          'joaosilva@gmail.com',  
8          new Date()  
9      ));  
10 }
```

O código do método `ngOnInit()` não é muito prático, está apenas para demonstração de como adicionar um elemento em `noticias` na instanciação do `AppComponent` e ter alguns dados iniciais. De qualquer forma, o `ngOnInit()` é o local correto para execução de código inicial do Componente.

### 5.6.2 Exibir telas diferentes

Por ter muitas funcionalidades, é necessário adotar alguma organização visual, ou seja, trabalhar a interface gráfica com o usuário. Para esse projeto, achei interessante utilizar o conceito de “telas” e trocar sua visibilidade conforme determinadas situações.

As telas são:

- **home**: apresenta as notícias recentes de forma ordenada
- **leitura**: apresenta o conteúdo de uma notícia
- **lista**: apresenta a lista de todas as notícias, com pesquisa
- **cadastro**: apresenta o formulário de cadastro
- **edicao**: apresenta o formulário de cadastro no modo de edição

A implementação desse recurso começa pelo atributo `tela` e do método `irPara()` do Controller:

```
1  irPara(nome) {  
2      this.tela = nome;  
3      if (nome === 'cadastro') {
```

```
4         this.editarNoticia = null;
5         this.titulo = null;
6         this.conteudo = null;
7         this.autor = null;
8         this.emailDoAutor = null;
9         this.data = null;
10    }
11 }
```

O código do método `irPara()` usa o parâmetro `tela` para trocar o valor do atributo `tela` e, se for a tela “cadastro”, realiza um código adicional, redefinindo para `null` os valores dos atributos `editarNoticia`, `titulo`, `conteudo`, `autor`, `emailDoAutor` e `data`.

No Template há uma estrutura composta por cabeçalho (`nav`) e de vários elementos `div`. No cabeçalho há botões que, ao serem clicados, chamam o método `irPara()` conforme cada necessidade de mudança de tela. Aos `div` foram aplicadas diretivas `NgIf` com expressões baseadas no atributo `tela`:

```
1 <h1>Notícias::Angular</h1>
2 <hr>
3 <nav>
4   <button (click)="irPara('home')">Home</button> |
5   <button (click)="irPara('lista')">Todas as notícias</button> |
6   <button (click)="irPara('cadastro')">Cadastrar</button>
7 </nav>
8 <hr>
9
10 <div *ngIf="tela == 'home'">
11   <h2>Notícias recentes</h2>
12   ...
13 </div>
14
15 <div *ngIf="tela == 'leitura' && leituraNoticia">
16   ...
17 </div>
18
19 <div *ngIf="tela == 'lista'">
20   <h2>Todas as notícias</h2>
21   ...
22 </div>
23
24 <div *ngIf="tela == 'cadastro' || tela == 'edicao'">
25   <h2>Cadastrar notícia</h2>
26   ...
27 </div>
```

Assim, o Template tem uma estrutura modular, baseada em elementos `div` que são inseridos ou removidos do DOM utilizando a diretiva `NgIf`. A Figura 5.2 ilustra a navegação entre as telas.

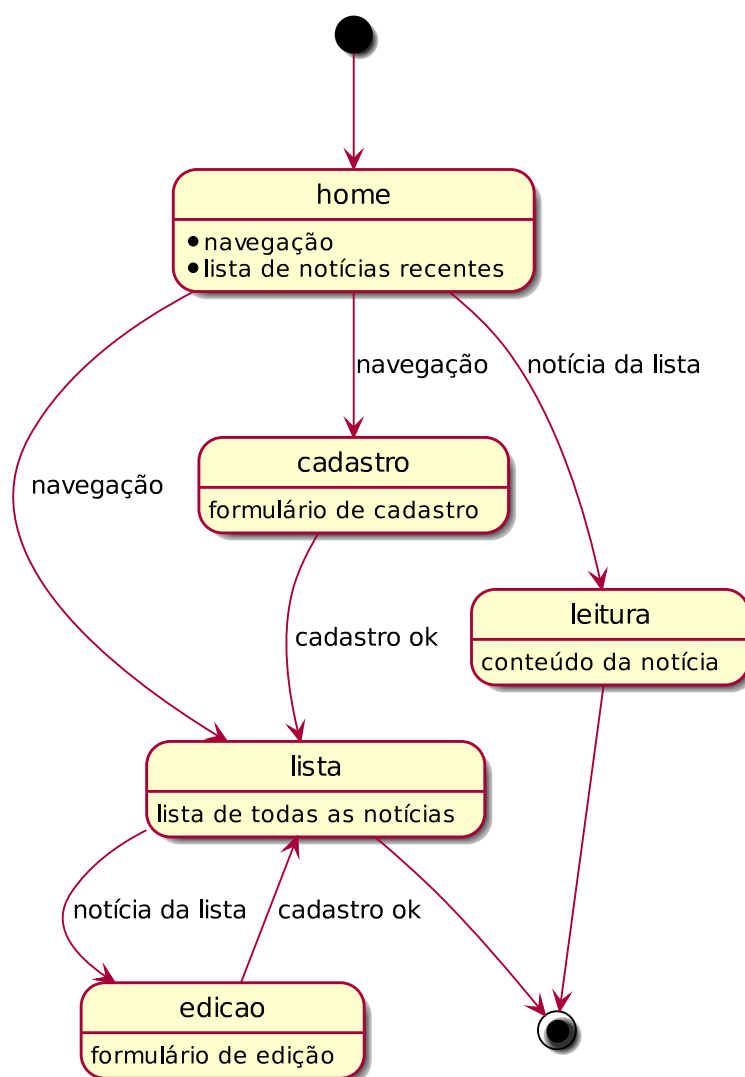


Figura 5.2: Diagrama de estados demonstrando navegação das telas do projeto angular-noticias, branch crud-telas

A Figura 5.2 apresenta um **Diagrama de Estados da UML** como forma de demonstrar os caminhos de navegação entre as telas do software. A idéia principal é demonstrar que o caminho passa pela tela **home** e pode prosseguir para **lista**, **cadastro** e **leitura**. Da **lista** pode-se ir para a **edicao** ou terminar (final da navegação). Do **cadastro** volta para **lista** quando um cadastro for concluído com sucesso e mesmo acontece a partir de **edicao**. Por fim, da **leitura** pode terminar. (LIEBERMAN, 2004) demonstra como utilizar Diagrama de Atividade da UML para demonstrar a mesma ideia. Dá uma olhada.

---

### Diagrama de navegação do site ou Sitegrama

A necessidade de entender os caminhos de navegação no site aumenta na mesma proporção da sua quantidade de páginas (links) ou seções. Para auxiliar o entendimento de um projeto de site é uma prática utilizar um diagrama de navegação (sitegrama), que demonstra, a partir da página inicial, quais as seções do site e a quais páginas elas dão acesso.

Recomendo leitura adicional de (AMSTEL, 2004; CORAIS.ORG, [s.d.]).

---

#### 5.6.3 Listar notícias recentes de forma ordenada

O Controller declara o método `noticiasPublicadas()`:

```
1  noticiasPublicadas() {
2      return this.noticias
3          .filter(n => n.estaPublicada())
4          .sort((a: Noticia, b: Noticia) => {
5              if (b.data < a.data) {
6                  return -1;
7              } else if (b.data > a.data) {
8                  return 1;
9              } else {
10                 return 0;
11             }
12         });
13 }
```

O código do método `noticiasPublicadas()` utiliza dois métodos importantes da classe `Array`: `filter()` e `sort()`. No TypeScript esses métodos aceitam como parâmetro uma função que usa uma sintaxe diferente, a **arrow function** (ou **lâmbda**). A sintaxe:

```
parâmetros => corpo
```

é interpretada assim:

- `parâmetros` contém uma lista de parâmetros, com tipos ou não

- `corpo` contém o corpo da função, que geralmente retorna um valor

Quando o corpo pode ser reduzido a uma linha não é necessário utilizar chaves duplas e nem a palavra `return`.

O método `filter()` aplica um filtro (**predicado**) aos elementos do `Array`, retornando apenas aqueles para os quais o filtro é verdadeiro. No código da função `noticiasPublicadas()` o predicado é a função: `n => n.estaPublicada()`, ou seja, usa o método `estaPublicada()` de cada `n` notícia da lista para saber se ela deve passar pelo filtro. Por isso é uma boa prática implementar essa lógica na classe `Noticia`.

O outro método é `sort()`, que aplica uma função de ordenação ao `Array`. A função de ordenação utilizada no método `noticiasPublicadas()` tem dois parâmetros: `a` e `b`, ambos do tipo `Noticia`. A função é aplicada a cada par de notícias da lista e retorna:

- `-1`: se a data da notícia `b` for menor que a da notícia `a`
- `1`: se a data da notícia `b` for maior que a da notícia `a`
- `0`: caso contrário

O resultado disso é que o Template utiliza o método `noticiasPublicadas()` para apresentar apenas as notícias publicadas e ordenada de forma decrescente pela data da notícia.

O trecho do Template é apresentado a seguir:

```

1 <div *ngIf="tela == 'home'">
2   <h2>Notícias recentes</h2>
3   <p>Clique no título da notícia para ler o conteúdo</p>
4   <div id="lista-noticias">
5     <div class="lista-noticias-item" *ngFor="let noticia of
6       noticiasPublicadas()" (click)="mostrar(noticia)">
7       <div class="lista-noticias-titulo">{{ noticia.titulo }}</div>
7       <div class="lista-noticias-info">Por {{ noticia.autor }} em {{
8         noticia.data | date }}</div>
8     </div>
9   </div>
10 </div>

```

A diretiva `NgForOf` aplicada ao `div` da linha 5 é responsável por criar os elementos da lista das notícias recentes. Perceba o uso da diretiva `(click)` para chamar o método `mostrar()` assim que o usuário clicar em cada item da lista, conduzindo a uma navegação para a tela **leitura** e apresentando o conteúdo da notícia em questão.

Há um recurso novo em ação: o **Pipe**. Na linha 7 é apresentada a data notícia (atributo `data`). Se nenhum tratamento for realizado, por ser um objeto `Date`, será apresentada uma data no formato padrão do JavaScript. Para lidar com isso de forma diferente é usado `DatePipe`:

```

{{ noticia.data | date }}

```

Um **pipe** é aplicado ao valor da esquerda (usando essa sintaxe que adota a barra vertical como

separador). Assim, `DatePipe` converte `noticia.data` para um formato mais amigável. O Angular disponibiliza o `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe` e `PercentPipe`.

#### 5.6.4 Apresentar conteúdo da notícia para leitura

A tela **leitura** é responsável por apresentar o conteúdo da notícia para leitura. No Controller, o método `mostrar()` tem o conteúdo:

```
1  mostrar(noticia) {  
2      this.leituraNoticia = noticia;  
3      this.irPara('leitura');  
4  }
```

O método `mostrar()` recebe o parâmetro `noticia` (a notícia que terá seu conteúdo apresentado) e o atribui para `leituraNoticia`, um atributo que é usado no Template para indicar a notícia que é apresentada. Por fim o método chama `irPara()`, para mudar para a tela **leitura**.

No Template há o código a seguir:

```
1  <div *ngIf="tela == 'leitura' && leituraNoticia">  
2      <h2>{{ leituraNoticia.titulo }}</h2>  
3      <div class="leitura-noticia-info">  
4          Por {{ leituraNoticia.autor }}  
5          <span *ngIf="leituraNoticia.emailDoAutor">({{ leituraNoticia.  
6              emailDoAutor }})</span>  
7          em {{ leituraNoticia.data | date }}  
8      </div>  
9      <hr>  
10     <div class="leitura-noticia-conteudo">  
11         {{ leituraNoticia.conteudo }}  
12     </div>  
13     <div class="leitura-noticia-rodape">  
14         <button (click)="fechar()">Fechar</button>  
15     </div>
```

O código do Template para tela **leitura** usa o atributo `leituraNoticia` para apresentar as informações da notícia (do título à data). Por fim, um `button` permite chamar o método `fechar()`, que oculta a tela **leitura** e apresenta novamente a tela **home**:

```
1  fechar() {  
2      this.leituraNoticia = null;  
3      this.irPara('home');  
4  }
```



### 5.6.5 Listar todas as notícias com pesquisa

A tela **lista** apresenta a lista de todas as notícias e fornece uma funcionalidade de pesquisa. Para implementar essas funcionalidades o Controller declara o método `noticiasParaLista()`:

```
1  noticiasParaLista() {
2      if (this.listaPesquisa) {
3          return this.noticias.filter(n =>
4              n.titulo.indexOf(this.listaPesquisa) !== -1
5              || n.conteudo.indexOf(this.listaPesquisa) !== -1
6              || n.autor.indexOf(this.listaPesquisa) !== -1
7          );
8      } else {
9          return this.noticias;
10     }
11 }
```

O método `noticiasParaLista()` verifica o valor do atributo `listaPesquisa`, que está vinculado a um `input` no Template para aplicar um filtro no `Array` `noticias`, utilizando o método `filter()`. A função predicado informada como parâmetro para o método `filter()` retorna `true` quando um dos atributos `titulo`, `conteudo` ou `autor` da notícia `n` tiver parte do conteúdo de `listaPesquisa`, utilizando o método `indexOf()` de `String`.

No Template o código apresenta o `input` para fornecer o campo de pesquisa e um `table` para apresentar a lista de notícias com base no método `noticiasParaLista()`:

```
1  <div *ngIf="tela == 'lista'">
2      <h2>Todas as notícias</h2>
3      <div id="lista-pesquisar">
4          <input type="search" placeholder="Digite para pesquisar..." [(ngModel)]="listaPesquisa">
5      </div>
6      <table>
7          <thead>
8              <tr>
9                  <th scope="col">#</th>
10                 <th scope="col">Título</th>
11                 <th scope="col">Autor</th>
12                 <th scope="col">Publicada</th>
13                 <th scope="col">Ações</th>
14             </tr>
15         </thead>
16         <tbody>
17             <tr *ngFor="let noticia of noticiasParaLista()">
18                 <td>{{ noticia.id }}</td>
19                 <td>{{ noticia.titulo }}</td>
20                 <td>{{ noticia.autor }}</td>
21                 <td>{{ noticia.data }}</td>
```

```
22         <td>
23             <button (click)="editar(noticia)">Editar</button> |
24             <button (click)="excluir(noticia)">Excluir</button>
25         </td>
26     </tr>
27 </tbody>
28 </table>
29 </div>
```

No `input` usado para a pesquisa está sendo utilizado o recurso de **two-way data binding**, para vinculá-lo ao atributo `listaPesquisa`. As linhas da tabela dentro do elemento `tbody` são criadas usando a diretiva `NgForOf` e a expressão de iteração chama o método `noticiasParaLista()`. A última coluna apresenta dois `button` que dão acesso, respectivamente, à funcionalidade de editar e excluir, chamando os métodos `editar()` e `excluir()` do Controller.

### 5.6.6 Cadastrar e editar notícia

Em termos de **two-way data binding** não há novidade quanto ao branch “início” além de haver um número maior de campos e atributos vinculados (`titulo`, `conteudo`, `autor`, `emailDoAutor` e `data`). Além disso, a lógica de salvar os dados do cadastro está adaptada para executar duas funções: cadastrar e editar. A seguir, o código do método `salvar()`:

```
1  salvar(form) {
2      if (!this.editarNoticia) {
3          const noticia = new Noticia(
4              this.noticias.length,
5              this.titulo,
6              this.conteudo,
7              this.autor,
8              this.emailDoAutor,
9              this.data
10         );
11         this.noticias.push(noticia);
12     } else {
13         const noticia = this.noticias.find(n => n.id === this.
14             editarNoticia.id);
15         noticia.titulo = this.titulo;
16         noticia.conteudo = this.conteudo;
17         noticia.autor = this.autor;
18         noticia.emailDoAutor = this.emailDoAutor;
19         if (this.data) {
20             noticia.data = new Date(this.data);
21         } else {
22             this.data = null;
23         }
24         this.editarNoticia = null;
25     }
26 }
```

```
25     form.reset();
26     this.irPara('lista');
27 }
```

O método `salvar()` recebe o parâmetro `form`, que deve ser informado no Template e é utilizado para redefinir os valores dos campos do formulário (linha 25). O método identifica qual operação realizar (cadastro ou edição) com base no atributo `editarNoticia`: se não estiver definido (não está editando) então opera no modo de cadastro, caso contrário, edição.

A primeira parte do método (linhas 2-10) trata do cadastro: cria uma instância de `Noticia` e a insere no `Array noticias`. A segunda parte (linhas 11-24) tem mais passos:

1. encontra o elemento de `noticias` com base em `editarNoticia.id` (usa o método `find()`)
2. atualiza os dados dessa notícia com base nos atributos vinculados ao formulário de edição

Uma parte importante do formulário (cadastro e edição) tem relação com o campo “data”. O software utiliza o campo “datetime-local” do HTML5 para que o usuário possa informar uma data e hora. Embora a interface apresente a data no formato do usuário (ex: **DD/MM/AAAA hh:mm**) o código recebe esse valor em outro formato **YYYY-MM-DDThh:mm** onde:

- **YYYY**: ano com quatro dígitos
- **MM**: número do mês do ano com dois dígitos
- **DD**: número do dia do mês com dois dígitos
- **hh**: horas com dois dígitos
- **mm**: minutos com dois dígitos

Assim a data informada pelo usuário “30/07/2018 18:00” se torna “2018-07-30T18:00”. O construtor da classe `Date` faz o parser dessa string no formato correto e cria um objeto da forma adequada.

A tela **editar** é apresentada quando o usuário clica no botão “Editar” na linha da tabela de notícias correspondente e chama o método `editar()`:

```
1  editar(noticia) {
2      this.editarNoticia = noticia;
3      this.titulo = noticia.titulo;
4      this.conteudo = noticia.conteudo;
5      this.autor = noticia.autor;
6      this.emailDoAutor = noticia.emailDoAutor;
7      this.data = this.date2str(noticia.data);
8      this.irPara('edicao');
9  }
```

O método `editar()` recebe o parâmetro `noticia` e atualiza os atributos vinculados ao formulário para que este, no modo de edição, apresente os dados da notícia. Uma ressalva trata do atributo `data` de `noticia`: por ser do tipo `Date` não pode ser atribuído diretamente ao campo “data”, por questões do formato. Portanto, a linha 7 chama o método `date2str()` que recebe um objeto `Date` e retorna uma string no formato **YYYY-MM-DDThh:mm**.

O Controller contém também o método `cancelarEdicao()`, que sai do modo de edição de notícia

sem editar os dados e retorna para a tela **lista**.

O Template para as telas **cadastro** e **edicao** é apresentado a seguir:

```

1 <div *ngIf="tela == 'cadastro' || tela == 'edicao'">
2   <h2>Cadastrar noticia</h2>
3   <form #cadastro="ngForm" (submit)="salvar(cadastro)">
4     <div>
5       <label for="titulo">Título</label>
6       <input type="text" id="titulo" name="titulo" [(ngModel)]="titulo"
          required>
7     </div>
8     <div>
9       <label for="conteudo">Conteúdo</label>
10      <textarea id="conteudo" name="conteudo" cols="80" rows="5" [(ngModel)
          ]="conteudo" required></textarea>
11    </div>
12    <div>
13      <label for="autor">Autor</label>
14      <input type="text" name="autor" id="autor" [(ngModel)]="autor"
          required>
15    </div>
16    <div>
17      <label for="emailDoAutor">E-mail do autor</label>
18      <input type="email" name="emailDoAutor" id="emailDoAutor" [(ngModel)
          ]="emailDoAutor">
19    </div>
20    <div>
21      <label for="data">Data</label>
22      <input type="datetime-local" name="data" id="data" [(ngModel)]="data"
          ">
23    </div>
24    <div>
25      <button type="submit" [disabled]="!cadastro.form.valid">Salvar</
          button>
26      <button type="reset" *ngIf="tela == 'cadastro'" formnovalidate>
          Limpar</button>
27      <button *ngIf="tela == 'edicao'" (click)="cancelarEdicao()"
          formnovalidate>Cancelar</button>
28    </div>
29  </form>
30 </div>

```

A diretiva `(submit)` chama o método `salvar()` passando como argumento a variável local de template `cadastro` (que representa o formulário). A sequência do código realiza **two-way data binding** e, por fim, apresenta três `button` que aparecem em situações diferentes:

- “Salvar” e “Limpar” aparecem quando a tela é **cadastro**
- “Salvar” e “Cancelar” aparecem quando a tela é **edicao**

### 5.6.7 Excluir notícia

A funcionalidade de excluir notícia não tem uma tela, mas faz parte da tela **lista**. Um **button** permite chamar o método `excluir()`:

```
1  excluir(noticia) {  
2      if (confirm(`Tem certeza que deseja excluir a notícia: ${noticia.  
          titulo} ?`)) {  
3          this.noticias.splice(this.noticias.findIndex(n => n.id === noticia  
              .id), 1);  
4      }  
5  }
```

O método `excluir()` recebe o parâmetro `noticia`, que representa a notícia para ser excluída. Antes de, efetivamente, excluir, o código chama a função `confirm()` para solicitar uma confirmação ao usuário. Se o usuário confirmar a exclusão, o código chama o método `splice()` do Array `noticias`, que recebe dois argumentos:

- a posição (índice) a partir de onde deve excluir elementos de `noticias`
- a quantidade de elementos para excluir

O índice da notícia a ser excluída é obtido usando o método `findIndex()` de `Array`, que recebe como argumento uma função predicado que encontra um elemento `n` da lista se seu atributo `id` for igual ao mesmo atributo do objeto `noticia` (parâmetro). Assim, se o usuário confirmar a exclusão, um elemento é removido de `noticias` e a lista de notícias é atualizada (por causa do data binding).

## 5.7 Conclusão

Este capítulo apresentou o branch “crud-telas” do software **noticias-angular**. Esse branch destaca a implementação de funcionalidades adicionais que o tornam mais próximo de um software real para o propósito de visualizar e gerenciar notícias, principalmente implementando uma interface CRUD.

A partir daqui serão utilizados mais recursos do Angular, como serviços. Sugiro fortemente que não avance a leitura e seus estudos se não tiver entendido claramente como funciona data binding e o uso de diretivas para as situações apresentadas até aqui.

## Capítulo 6

# Interação entre componentes

O Capítulo 5 apresenta uma implementação do **noticias-angular** com funcionalidades de listar notícias recentes, ler notícia, listar todas as notícias, cadastrar notícia, editar notícia e excluir notícia. Todas essas funcionalidades foram implementadas utilizando um só componente. Embora o software funcione perfeitamente foi necessário adotar uma lógica baseada em condicionais para troca da visibilidade de telas. A lógica fica complexa na medida da quantidade de funcionalidades, o que pode se tornar uma dor de cabeça para manutenções e entendimento do software.

Este capítulo apresenta uma solução alternativa: a utilização de componentes individuais para cada funcionalidade. Por ser um framework orientado a princípios e boas práticas de desenvolvimento de software, o Angular também proporciona recursos para aplicar essas boas práticas. Um deles é apresentado neste capítulo.

Este capítulo apresenta:

- comando do **Angular CLI** para criar componente
- configuração do módulo
- metadados **Input** e **Output**
- interação entre componentes
- componentes **host** e **guest**

### 6.1 Clonar e executar localmente

Este capítulo está relacionado com o software **noticias-angular**, branch “componentes”<sup>1</sup>. Antes de prosseguir, clone e execute o projeto localmente.

---

<sup>1</sup>O repositório do branch “componentes” está disponível em <https://github.com/jacksongomesbr/webdevbook-noticias-angular/tree/componentes> e você pode vê-lo em execução acessando <https://stackblitz.com/github/jacksongomesbr/webdevbook-noticias-angular/tree/componentes>

## 6.2 Criar componente com o Angular CLI

Para criar um componente com o **Angular CLI** usamos o comando `generate component`:

```
ng g c NoticiasRecentes --spec false
```

O comando cria o componente `NoticiasRecentesComponent` na pasta `src/app/noticias-recentes`. A opção `--spec false` é utilizada para não criar arquivos de teste.

## 6.3 Configuração do módulo

Cada componente precisa ser *declarado* por um módulo para poder ser utilizado no software. Isso significa que um módulo precisa conter o componente no array `declarations` dos seus metadados.

O **Angular CLI** faz essa configuração automaticamente, tornando o código do módulo assim:

```
1  ...
2  import { AppComponent } from './app.component';
3  import { NoticiasRecentesComponent } from './noticias-recentes/noticias-
    recentes.component';
4
5  @NgModule({
6    declarations: [
7      AppComponent,
8      NoticiasRecentesComponent
9    ],
10   ...
11 })
12 export class AppModule { }
```

Uma vez que o componente está no array `declarations` de um módulo (nesse caso, o **root module**) ele pode ser utilizado em outros componentes. Mas o que significa isso? É o que mostra a próxima seção.

## 6.4 Estrutura inicial do NoticiasRecentesComponent

O componente `NoticiasRecentesComponent` tem a responsabilidade de implementar a funcionalidade de apresentar a lista de notícias recentes. Veja bem: eu disse apresentar. Isso quer dizer que ele não tem a responsabilidade de obter ou conter a lista das notícias recentes, apenas de apresentá-las.

Além disso, não é responsabilidade desse componente apresentar a tela de leitura da notícia. Entretanto, como cada item da lista de notícias recentes atende ao evento de clique, é necessário utilizar um recurso específico para este propósito.

Primeiro, o componente deve poder receber como entrada a lista de notícias. Isso é representado por meio de metadados:

```
1 import { Component, OnInit, Input } from '@angular/core';
2
3 @Component({
4   ...
5 })
6 export class NoticiasRecentesComponent implements OnInit {
7
8   @Input()
9   noticias;
10  ...
11 }
```

Perceba a importação da função de anotação (metadado) `Input` e do seu uso junto do atributo `noticias`, como metadados. Os metadados podem ser usados também numa sintaxe alternativa:

```
1 import { Component, OnInit, Input } from '@angular/core';
2
3 @Component({
4   ...
5 })
6 export class NoticiasRecentesComponent implements OnInit {
7
8   @Input() noticias;
9   ...
10 }
```

O comportamento é o mesmo: indicar que o atributo `noticias` torna-se uma propriedade de entrada do componente.

## 6.5 Utilizando o componente `NoticiasRecentesComponent`

Uma parte importante de cada componente é que seus metadados incluem o atributo `selector`, como mostra o trecho a seguir:

```
1 ...
2 @Component({
3   selector: 'app-noticias-recentes',
4   templateUrl: './noticias-recentes.component.html',
5   styleUrls: ['./noticias-recentes.component.css']
6 })
7 export class NoticiasRecentesComponent implements OnInit {
8   ...
```



```
9 }
```

O **selector** é uma string que representa um **seletor CSS**. Assim, qualquer seletor CSS é válido e aceito (seletor de elemento, classe, identificador, pseudo-seletor etc.).

Os metadados do `NoticiasRecientesComponent` indicam que seu **selector** é `app-noticias-recientes`, ou seja, é um seletor de elemento. Isso quer dizer que esse componente deve ser utilizado no template de outro componente utilizando o seletor de elemento, ou seja, algo parecido com:

```
1 <app-noticias-recientes></app-noticias-recientes>
```

Vamos chamar o componente que usa outro de **host** (hospedeiro) e o componente que é utilizado de **guest** (convidado).

Assim, o código a seguir mostra um trecho do template do `AppComponent` (**host**):

```
1 <h1>Notícias::Angular</h1>
2 <hr>
3 <nav>
4 ...
5 </nav>
6 <hr>
7
8 <app-noticias-recientes
9     [noticias]="noticiasPublicadas()"
10    *ngIf="tela == 'home'">
11 </app-noticias-recientes>
12
13 <div *ngIf="tela == 'leitura' && leituraNoticia">
14 ...
```

O template indica que está sendo utilizado o componente cujo seletor é `app-noticias-recientes`, ou seja, `NoticiasRecientesComponent` (**guest**). Ainda, a propriedade `noticias` do `NoticiasRecientesComponent` está vinculada, por meio de data-binding, ao método `noticiasPublicadas()` do `AppComponent`.

Isso demonstra uma interação bastante interessante entre os componentes:

- `AppComponent` (**host**) contém a lista de notícias e o método `noticiasPublicadas()`
- `AppComponent` utiliza o `NoticiasRecientesComponent` (**guest**) e, por meio de data-binding, vincula a propriedade de entrada `noticias` ao método `noticiasPublicadas()` do `AppComponent`

Outra coisa: a diretiva `NgIf` é aplicada diretamente ao elemento `app-noticias-recientes`, para demonstrar que as diretivas estruturais podem ser realmente aplicadas a elementos quaisquer<sup>2</sup>.

---

<sup>2</sup>Certamente o mesmo comportamento poderia ser obtido encapsulando o elemento `app-noticias-recientes` em um elemento `div`, aplicando neste último a diretiva `NgIf`

### Uso de componentes nos templates e instaciação

Cada vez que um componente é utilizado no template o Angular cria uma instância do componente. Isso quer dizer que, a princípio, é possível criar mais de uma instância de um componente inserindo mais de um seletor correspondente. Experimente.

---

## 6.6 Criando eventos

Como você já sabe, cada item da lista de notícias pode ser clicado. Quando o clique acontece, o software mostra a tela de leitura da notícia.

O desenvolvimento orientado a componentes envolve a arte de estruturar e pensar a arquitetura do software de tal maneira a elaborar as interações e relações entre os diversos componentes.

Nesse sentido, a questão é que o componente `NoticiasRecentesComponent` pode, claramente, continuar utilizando os recursos de eventos do Angular. O problema é que, como a funcionalidade de apresentar a tela de leitura da notícia está fora dele, é necessário um meio que permita interagir com o componente **host**. A maneira de fazer isso é criando uma propriedade de saída, ou seja, um evento:

```
1 import { Component, OnInit, Input, Output, EventEmitter } from '@angular/
  core';
2 import { Noticia } from '../noticia.model';
3
4 @Component({
5   ..
6 })
7 export class NoticiasRecentesComponent implements OnInit {
8
9   @Input()
10   noticias;
11
12   @Output()
13   mostrou = new EventEmitter<Noticia>();
14
15   ...
16
17   mostrar(noticia) {
18     this.mostrou.emit(noticia);
19   }
20 }
```

O trecho de código mostra que, além de importar a anotação `Input`, também é necessário importar a anotação `Output` e a classe `EventEmitter`. A anotação `Output` é utilizada como metadado no atributo `mostrou`, que recebe uma instância de `EventEmitter`. Novamente, vale a sintaxe alternativa:

```
1  ...
2  export class NoticiasRecentesComponent implements OnInit {
3
4      @Input() noticias;
5
6      @Output() mostrou = new EventEmitter<Noticia>();
7
8      ...
9  }
```

A sintaxe `construtor<tipo>()` pertence ao recurso chamado de **generics** e serve para indicar que o tipo de dados utilizado no evento é `Noticia`<sup>3</sup>. Cada evento contém um objeto com informações adicionais e isso será muito útil a seguir.

Ainda do lado do controller há o método `mostrar()`, que recebe o parâmetro `noticia` e chama o método `emit()` do atributo `mostrou`, passando como argumento o parâmetro `noticia`. Isso indica que o **componente host** terá disponível no tratador de evento o objeto `noticia` que corresponde ao que foi clicado pelo usuário.

No **componente host** o elemento completo, no template, fica como:

```
1  <app-noticias-recentes
2      [noticias]="noticiasPublicadas()"
3      (mostrou)="onMostrou($event)"
4      *ngIf="tela == 'home'">
5  </app-noticias-recentes>
```

O elemento `app-noticias-recentes` tem, além da propriedade `noticias` e da diretiva `NgIf`, o evento `mostrou`. O evento `mostrou` está vinculado ao método `onMostrou()`, definido no `AppComponent`. É importante notar que o template chama o método `onMostrou()` e informa, como argumento, o valor especial `$event`. `$event` representa o dado associado ao evento conforme definido pelo **componente guest** ao chamar o método `emit()`. A seguir o código do método `onMostrou()`:

```
1  onMostrou(noticia) {
2      this.leituraNoticia = noticia;
3      this.irPara('leitura');
4  }
```

O método `onMostrou()` recebe o parâmetro `noticia`. Seu corpo é o mesmo do método `mostrar()` do branch `crud-telas`:

- atribui o parâmetro `noticia` ao atributo `leituraNoticia`
- chama o método `irPara()` com o argumento `leitura`, indicando que deve mostrar a tela `leitura` da notícia.

---

<sup>3</sup>Leia mais sobre **generics** na documentação do TypeScript: <http://www.typescriptlang.org/docs/handbook/generics.html>.

## 6.7 Conclusão

Este capítulo apresentou:

- comando do **Angular CLI** para criar componente
- configuração do módulo para declarar um componente
- metadados `Input` e `Output`
- interação entre componentes
- componentes **host** e **guest**

## Capítulo 7

# Rotas e Serviços

O capítulo 5 apresentou como utilizar componentes para implementar um software de maneira a organizar o código do software em partes e utilizar recurso de interface semelhante a telas. Este capítulo apresenta uma solução mais robusta: utilizar serviços e rotas, bem como o recurso de injeção de dependência.

Este capítulo apresenta:

- serviços
- injeção de dependência
- rotas, URL, URI
- rotas no Angular
- parâmetros de rota
- shell component
- navegação em rotas

### 7.1 Serviços

Conforme a Arquitetura do Angular a utilização de **Serviços** tem o propósito de organizar o projeto de software Angular, isolando lógica de negócio e separando-a dos Controllers. Não é possível afirmar que seja obrigatório utilizar serviços, mas é muito desejável.

Na prática não há diferença para o usuário porque, provavelmente, utilizar serviços não afetará diretamente o comportamento da interface. Assim, os benefícios ficam por conta da melhor organização do projeto e da Engenharia de Software para o projeto que está sendo desenvolvido.

Um serviço é uma classe que pode ser utilizada por outros componentes do projeto. Dois detalhes importantes são: o que diferencia um serviço de outro tipo de componente e como um componente utiliza um serviço. Veremos isso a seguir.

Até o momento a lógica do CRUD de notícias está nos Controllers dos componentes. Isso não é uma boa prática, porque é indicado que a lógica dos Controllers seja voltada para gerenciar o comportamento da interface, ou seja, para controlar a interação entre o usuário e o software.

Vamos começar criando o serviço `NoticiasService` usando Angular CLI e o comando **generate** e a opção **service**:

```
ng g s Noticias
```

A linha de comando cria o arquivo `src/app/noticias.service.ts`, cujo trecho de conteúdo é apresentado a seguir:

```
1 import { Injectable } from '@angular/core';
2 import { Noticia } from './noticia.model';
3
4 @Injectable({
5   providedIn: 'root'
6 })
7 export class NoticiasService {
8   lista: Array<Noticia> = [];
9
10  constructor() {
11    this.lista.push(new Noticia(1, 'Teste de notícia',
12      'Conteúdo da notícia de teste',
13      'José da Silva', 'jose@gmail.com',
14      new Date()));
15  }
16
17  public publicadas() {
18    ...
19  }
20  ...
21 }
```

O código utiliza a função de anotação `Injectable()` (fornecida pelo pacote `@angular/core`) para adicionar metadados à classe). Na prática, essa função de anotação é que determina que a classe é realmente um serviço.

O parâmetro da função de anotação tem um atributo `providedIn` com valor `root`, o que indica que o serviço está disponível para ser utilizado em qualquer componente do software. Há outras formas de modificar esse comportamento, mas isso não será visto agora.

Para que uma classe utilize um serviço o Angular implementa o padrão **Dependency Injection** (DI), um padrão de projeto que faz com que o próprio framework crie instâncias do serviço para serem utilizadas no momento adequado. Para exemplificar, considere o trecho de código a seguir:

```
1 import { Component, OnInit } from '@angular/core';
2 import { NoticiasService } from '../noticias.service';
3 import { ActivatedRoute, Router } from '@angular/router';
4
5 ...
```

```
6 export class LeitorDeNoticiaComponent implements OnInit {
7   noticia = null;
8
9   constructor(private noticias: NoticiasService,
10    private route: ActivatedRoute,
11    private router: Router) { }
12
13   ngOnInit() {
14   }
15
16   ...
17 }
```

A classe `LeitorDeNoticiaComponent` utiliza o recurso de DI para “injetar” o serviço `NoticiasService` – isso é feito no construtor da classe. A sintaxe para utilizar esse recurso é `private variavel : Tipo`. Por exemplo: `private noticias: NoticiasService` declara o atributo `noticias`, do tipo `NoticiasService` e faz com que o Angular use o recurso de DI para tornar disponível uma instância de `NoticiasService` (no atributo `noticias`) para ser utilizada no componente.

As seções a seguir têm conteúdo introdutório, antes de apresentar conceitos de rotas.

## 7.2 URI, URL e URN

**URI** (*Uniform Resource Identifier*) é um identificador de um recurso. A sintaxe desse identificador é:

```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

Onde os elementos entre `[]` são opcionais e:

- **scheme**: representa o protocolo ou o contexto;
- **user** e **password**: representam as credenciais do usuário;
- **host**: representa o identificador do dispositivo que contém o recurso;
- **port**: representa o número da porta do dispositivo;
- **path**: representa o caminho do recurso;
- **query**: uma cadeia de caracteres que representa parâmetros de URL, um conjunto de pares `chave=valor` separados por `&`; e
- **fragment**: uma cadeia de caracteres com formato dependente do contexto.

A figura a seguir ilustra um URI e a sua composição:

**https://jacksongomesbr@github.com/jacksongomesbr/angular-escola.git**

---

scheme      credenciais do usuário      host      caminho

Figura 7.1: Composição da URL

O URI identifica um recurso disponível na internet, mais especificamente um repositório do Github. Provavelmente você já conhece isso e pode ser que use os termos **endereço** e **URL**. Você não está errado. URL é a forma mais comum de URI na internet. **URL** (*Uniform Resource Locator*) é um endereço de um recurso na internet. Além disso, URL é a forma mais comum de criar *links* entre páginas web e incorporar uma imagem em uma página web, por exemplo.

**URN** (*Uniform Resource Name*) é o nome de um recurso em um contexto específico. Por exemplo: o ISBN (*International Standard Book Number*) de uma edição de “Romeu e Julieta”, de William Shakespeare, é 0-486-27557-4; seu URN poderia ser **urn:isbn:0-486-2557-4**.

Voltando para URL e o contexto da internet, os browsers geralmente fornecem uma **barra de endereço**, por meio da qual o usuário indica qual URL deseja acessar, por exemplo a URL de uma página web. A partir do momento que o browser acessa uma página web ele passa a armazenar o primeiro endereço acessado e os demais endereços que forem acessados por meio de cliques em *links*.

Esse é, provavelmente, o formato mais intuitivo e o mais utilizado para acessar páginas web. Justamente por isso é necessário repensar a forma como o aplicativo desenvolvido em Angular não apenas entrega conteúdo para o usuário mas também como permite que o usuário o acesse.

## 7.3 Rotas

Uma rota está diretamente relacionada a URL, ou seja, também funciona como um localizador de um recurso. A diferença é que acrescenta a possibilidade de utilizar **parâmetros de rota**. Por exemplo: considere um site de notícias **noticias.to** que permite acessar a notícia “Governo paga salarios de servidores”, cujo identificador é 7899, está na categoria “política” e foi publicada em 20/12/2017, por meio do URL:

```
https://noticias.to/noticias/politica/2017/12/20/governo-paga-salarios-de-servidores/7899
```

Há informações no URL que pertencem à notícia e mudam de uma notícia para outra:

- **categoria:** politica
- **ano:** 2017
- **mês:** 12
- **dia:** 20
- **titulo:** governo-paga-salarios-de-servidores
- **identificador:** 7899

Analisando URLs de outras notícias alguém poderia chegar à conclusão de que há um padrão:

```
/noticias/categoria/ano/mes/dia/titulo/identificador
```

Independentemente de possuir parâmetros de rota, uma rota é um **padrão**. Cada uma dessas informações (categoria, ano, mes, dia, titulo, identificador), que muda de uma notícia para outra,



pode ser representada como um **parâmetro de rota**.

A implementação desse conceito pode variar entre frameworks, mas provavelmente as mesmas funcionalidades estão disponíveis:

- definir uma rota (e, opcionalmente, usar parâmetros de rota)
- identificar valores dos parâmetros de rota

Além disso, como URLs são localizadores de recursos, rotas também servem para esse propósito, ou seja, uma rota está associada algum recurso e é uma forma de acessá-lo.

## 7.4 Rotas no Angular

Como as seções anteriores apresentam, rotas podem ser entendidas como “caminhos”, não URLs verdadeiras, pois são interpretados conforme algum contexto. No Angular isso também acontece. Uma das formas de definir rotas é modificar o `AppModule`:

```
1  ...
2  import { RouterModule, Routes } from '@angular/router';
3
4  import { AppComponent } from './app.component';
5  import { NoticiasRecentesComponent } from './noticias-recentes/noticias-
    recentes.component';
6  import { PaginaNaoEncontradaComponent } from './pagina-nao-encontrada/
    pagina-nao-encontrada.component';
7  import { LeitorDeNoticiaComponent } from './leitor-de-noticia/leitor-de-
    noticia.component';
8  import { ListaDeNoticiasComponent } from './lista-de-noticias/lista-de-
    noticias.component';
9
10 const rotas: Routes = [
11   {path: 'admin/noticias', component: ListaDeNoticiasComponent, },
12   {path: 'noticias/:id', component: LeitorDeNoticiaComponent, },
13   {path: '', component: NoticiasRecentesComponent, },
14   {path: '**', component: PaginaNaoEncontradaComponent}
15 ];
16
17 @NgModule({
18   declarations: [
19     AppComponent,
20     NoticiasRecentesComponent,
21     PaginaNaoEncontradaComponent,
22     LeitorDeNoticiaComponent,
23     ListaDeNoticiasComponent
24   ],
25   imports: [
26     BrowserModule,
```

```
27     FormsModule ,
28     RouterModule.forRoot(rotas)
29   ],
30   providers: [],
31   bootstrap: [AppComponent]
32 })
33 export class AppModule { }
```

Primeira parte de destaque é que o código importa `RouterModule` e `Routes` do pacote `@angular/router`.

Na sequência, depois de importações de vários componentes, há a definição de `rotas`, do tipo `Routes`, que recebe um `array`. Cada elemento do array é um objeto com dois atributos:

- `path`: representa a rota para o componente; e
- `component`: representa o componente associado à rota.

A primeira rota, `admin/noticias` está associada ao componente `ListaDeNoticiasComponent`.

A segunda rota, `noticias/:id`, está associada ao componente `LeitorDeNoticiasComponent`. Esta rota possui um **parâmetro de rota**. A sintaxe do Angular para parâmetro de rotas é usar o sinal de dois pontos seguido do nome do parâmetro. Nesse caso o parâmetro chama-se `id`.

A terceira e a quarta rotas têm um comportamento especial. A terceira rota é uma string de comprimento zero (“vazia”) e está associada ao componente `NoticiasRecentesComponent`. Isso significa que essa é a **rota padrão**. A quarta rota é `**`, associada ao componente `PaginaNaoEncontradaComponent`. Isso significa um atalho para o seguinte comportamento: se o usuário informar uma URL que não combina com nenhuma das rotas, então navegue até o componente `PaginaNaoEncontradaComponent`.

Por fim, as rotas são inseridas no módulo por meio de um elemento do array `imports` resultante de uma chamada para o método `RouterModule.forRoot()`, passando como parâmetro o array de rotas, o a variável `rotas`.

Quando o software é executado o Angular busca uma combinação entre a URL fornecida no browser e as rotas definidas no módulo. Isso é feito de cima para baixo, procurando no array `rotas`. Ao encontrar uma rota correspondente, o Angular cria uma instância do componente e a apresenta. O modo de comparação entre a URL no browser e o caminho da rota é muito interessante: a forma de comparação considera se o caminho da rota combina com o final da URL do browser.

Para exemplificar, considere que o usuário informa a URL `http://localhost:4200` no browser. Ao iniciar seu ciclo de execução o Angular começa a procurar a rota que combina com essa URL e descobre que é a **rota padrão**.

Note que a *ordem das rotas* no array de rotas (no exemplo, `rotas`) é realmente importante. Como disse, o Angular procura pela rota que combina com o final da URL, assim, se a rota padrão estiver no início da lista das rotas, então o Angular sempre a encontrará primeiro. Da mesma forma, se não encontrar uma rota correspondente o Angular vai até a rota `**` (por isso tem o comportamento de “página não encontrada”).

### 7.4.1 Shell component

Ao utilizar rotas o Angular atribui um comportamento especial a um dos componentes do projeto, chamando-o de **shell**. O componente **shell** será responsável por servir como uma “casca”, contendo outros outros componentes. Nesse contexto, **shell** geralmente representa algo visual, até em termos de HTML. Por exemplo, considere um site com a seguinte estrutura:

- cabeçalho
- navegação
- conteúdo
- rodapé

Todas as páginas do site compartilham dessa mesma estrutura, diferenciando apenas no conteúdo. Assim, o **shell** contém as partes que não mudam: cabeçalho, navegação e rodapé.

No Angular, o **shell** determina onde cada componente será apresentado por meio do elemento `router-outlet`. Por exemplo, o código a seguir apresenta um trecho do template do `AppComponent`:

```
1 <h1>Notícias::Angular</h1>
2 <hr>
3 <nav>
4 ...
5 </nav>
6 <hr>
7
8 <router-outlet></router-outlet>
```

Quando o Angular entrega o conteúdo para o browser ele executa o processo composto, resumidamente, por:

1. identificar o componente shell e obter seu template
2. identificar a rota e o componente que deve ser apresentado
3. obter o conteúdo do componente e inseri-lo no shell

Para o usuário, a experiência é a de ver uma “página” ou “tela” completa.

### 7.4.2 Parâmetros de rota

A rota `noticias/:id`, como já visto, possui um **parâmetro de rota** chamado `id`. Obter o valor desse identificador da notícia na URL é uma tarefa importante desse processo de lidar com rotas no Angular. Para fazer isso, o componente `LeitorDeNoticiaComponent` possui o atributo `route`, uma instância de `ActivatedRoute`, como mostra o trecho de código a seguir:

```
1 import { Component, OnInit } from '@angular/core';
2 import { NoticiasService } from '../noticias.service';
3 import { ActivatedRoute, Router } from '@angular/router';
4
```

```
5 ...
6 export class LeitorDeNoticiaComponent implements OnInit {
7   noticia = null;
8
9   constructor(private noticias: NoticiasService,
10    private route: ActivatedRoute,
11    private router: Router) { }
12
13   ngOnInit() {
14     const id = this.route.snapshot.paramMap.get('id');
15     this.noticia = this.noticias.encontrar(Number.parseInt(id));
16   }
17
18   ...
19 }
```

Como o componente implementa a interface `OnInit`, o código do método `ngOnInit()` é responsável por:

- **identificar o valor do parâmetro de rota:** isso é feito por meio de uma chamada para o método `route.snapshot.paramMap.get()`. O parâmetro para o método `get()` é o nome do parâmetro de rota desejado (nesse caso, `id`)
- **tratar o valor do parâmetro de rota:** o valor retornado por `get()` é do tipo `string`. O método `parseInt()` é utilizado para converter o valor da variável `id` para `number` antes de chamar o método `encontrar()` de `NoticiasService` e atribuir seu retorno para o atributo `noticia`.

### 7.4.3 Navegação

A navegação é o recurso que permite mudar de uma rota para outra. Isso pode ser feito por meio de uma ação do usuário (por exemplo, quando ele clica em um link) ou via código.

A navegação por meio de link utiliza o elemento `a` e o atributo `routerLink`, como mostra o trecho do template do `AppComponent`:

```
1 <h1>Notícias::Angular</h1>
2 <hr>
3 <nav>
4   <a routerLink="">Home</a> |
5   Admin:
6   <a routerLink="admin/noticias">Gerenciar notícias</a>
7 </nav>
8 <hr>
```

Perceba que o valor de `routerLink` está relacionado com o caminho da rota (como definido na variável `rotas`, no módulo `AppModule`).

A outra forma de gerar navegação é via código. Para isso, considere o trecho do controller de `NoticiasRecentesComponent`:

```
1 import { Component, OnInit } from '@angular/core';
2 import { Noticia } from '../noticia.model';
3 import { NoticiasService } from '../noticias.service';
4 import { Router } from '@angular/router';
5
6 ...
7 export class NoticiasRecentesComponent implements OnInit {
8
9     constructor(private noticias: NoticiasService,
10                 private router: Router) { }
11
12     ngOnInit() {
13     }
14
15     mostrar(noticia: Noticia) {
16         this.router.navigate(['noticias', noticia.id]);
17     }
18
19 }
```

O método `mostrar()`, que é chamado no template no clique de cada item da lista de notícias para acessar o leitor da notícia, utiliza o atributo `router`, uma instância de `Router`. O método `navigate()` recebe como parâmetro um array cujo cada elemento é utilizado para compor a rota de navegação. Nesse caso, como o objeto é navegar até a rota `noticias/:id`, o array possui dois elementos: `'noticias'` e `noticia.id`.

## 7.5 Padrão de trabalho (workflow)

Lidar com rotas no Angular é um processo simples, mas que aumenta de complexidade na proporção da quantidade de componentes, módulos ou na complexidade da arquitetura do software. Entretanto, lidar com esse processo contém alguns passos padrão (supondo que o projeto já tenha iniciado e adote a mesma estrutura do Angular CLI):

1. **criar componente:** a primeira coisa a fazer é criar o componente desejado. Como é importante conduzir esse processo de forma iterativa, não se preocupe em construir o componente por inteiro, deixe-o como criado pelo Angular CLI;
2. **definir rotas:** depois, defina as rotas e seus componentes (mantenha o componente com conteúdo padrão até aqui, para simplificar);
3. **implementar a lógica de negócio em um serviço:** ao utilizar serviços, implemente a lógica de negócio do serviço; e
4. **implementar lógica do componente e usar o serviço:** por fim, melhore a lógica do componente, fornecendo código para a lógica da interface e da lógica de negócio, e utilize o

serviço.

Adotar esse padrão de trabalho pode tornar o desenvolvimento mais simples e rápido, bem como reduzir a quantidade de erros e permitir a identificação dos erros de forma mais ágil.

## Capítulo 8

# Recursos de interface gráfica do Bootstrap

(capítulo em construção)

## Capítulo 9

# Comunicação com back-end por meio do HttpClient

Browsers atuais permitem a comunicação entre um software front-end e um back-end (ex: uma API HTTP) por meio da interface `XmlHttpRequest` (XHR).

O módulo `@angular/common/http` fornece o `HttpClient`, um serviço para comunicação com back-ends por meio do `XmlHttpRequest`.

### 9.1 Configuração do módulo

Para utilizar o `XmlHttpRequest` é necessário configurar o módulo do software Angular, importando `HttpClientModule` (fornecido pelo pacote `@angular/common/http`), como mostra o trecho de código a seguir:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { FormsModule } from '@angular/forms';
4 import { RouterModule, Routes } from '@angular/router';
5 import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
6 import { HttpClientModule } from '@angular/common/http';
7
8 import { AppComponent } from './app.component';
9 ...
10
11 const rotas: Routes = [
12 ...
13 ];
14
15 @NgModule({
16   declarations: [
```



```
17     AppComponent,  
18     ...  
19 ],  
20 imports: [  
21     NgbModule,  
22     BrowserModule,  
23     FormsModule,  
24     RouterModule.forRoot(rotas),  
25     HttpClientModule  
26 ],  
27 providers: [],  
28 bootstrap: [AppComponent]  
29 })  
30 export class AppModule { }
```

Uma vez que o `HttpClientModule` é importado no módulo, qualquer elemento do software (ex: componente ou serviço) pode injetar o `HttpClient`. A seguir, um trecho de código do `NoticiasService`:

```
1 import { Injectable } from '@angular/core';  
2 import { HttpClient } from '@angular/common/http';  
3 ...  
4  
5 @Injectable({  
6     providedIn: 'root'  
7 })  
8 export class NoticiasService {  
9     API_URL = 'http://localhost:8000/api/noticias/';  
10  
11     constructor(private http: HttpClient) {  
12     }  
13     ...  
14 }
```

O código importa a classe `HttpClient`, fornecida por `@angular/common/http` e a injeta no serviço `NoticiasService` como o atributo `http`.

Depois dessa configuração inicial o software pode utilizar o `HttpClient` para requisições HTTP, como demonstram as seções a seguir.

## 9.2 Recebendo dados de um arquivo JSON

Uma forma interessante de distribuir dados no software é codificá-los no formato JSON em um arquivo. Isso também é interessante porque qualquer servidor web (incluindo o servidor de desenvolvimento local) pode entregar esse arquivo. Por exemplo, considere que os dados de uma notícia estejam no arquivo `assets/db.json` e o serviço `NoticiasService` deseja consultar seu conteúdo.

Primeiro, o conteúdo do arquivo JSON:

```
1  [
2    {
3      "id": 1,
4      "titulo": "Os seis atos do jogo de equipe da Mercedes que tirou
               vitória de Bottas e beneficiou Hamilton"
5    },
6    {
7      "id": 2,
8      "titulo": "Passa de 800 o número de mortos na Indonésia devastada
               por terremoto e tsunami"
9    }
10 ]
```

O arquivo representa uma lista com dois objetos que possuem a mesma estrutura: dois atributos:

- id (um número)
- titulo (uma string)

Segundo, um trecho de código de `NoticiasService`:

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient } from '@angular/common/http';
3
4  @Injectable({
5    providedIn: 'root'
6  })
7  export class NoticiasService {
8    DB_URL = 'assets/db.json';
9
10   constructor(private http: HttpClient) {
11   }
12
13   public todas() {
14     return this.http.get(this.DB_URL);
15   }
16 }
```

O método `todas()` utiliza o objeto `http` e seu método `get()` para realizar uma requisição HTTP para a URL que direciona para o arquivo JSON.

Para consumir `NoticiasService` o componente `NoticiasRecentesComponent` utiliza injeção de dependência e o método `subscribe()`:

```
1  import { Component, OnInit } from '@angular/core';
2  import { NoticiasService } from '../noticias.service';
3
4  @Component({
5    selector: 'app-noticias-recentes',
```

```
6   templateUrl: './noticias-recientes.component.html',
7   styleUrls: ['./noticias-recientes.component.css']
8 })
9 export class NoticiasRecientesComponent implements OnInit {
10   noticias = null;
11
12   constructor(private noticias: NoticiasService) { }
13
14   ngOnInit() {
15     this.noticias.todas()
16       .subscribe(noticias => this.noticias = noticias);
17   }
18 }
```

O método `todas()` do `NoticiasService` retorna um `Observable`, que ainda não representa os dados de notícias, em si. Para acessar os dados de notícias é utilizado o método `subscribe()`, que recebe uma *callback* com um parâmetro (`noticias`) e copia seu valor para o atributo `noticias` do componente.

Também é possível passar uma *callback* para o método `subscribe()` para tratar erros (como erro de conexão ou erro de execução no servidor, por exemplo). Para isso, basta usar o segundo parâmetro:

```
1  ngOnInit() {
2    this.noticias.todas()
3      .subscribe(
4        noticias => this.noticias = noticias,
5        erro => console.log(erro)
6      );
7  }
```

A segunda *callback* recebe o parâmetro `erro`, um objeto que contém detalhes do erro.

Posteriormente, as notícias são apresentadas no template:

```
1  <h2 class="mb-4"><i class="fa fa-newspaper-o"></i> Notícias recentes</h2>
2  <div class="card-deck">
3    <div class="card" *ngFor="let noticia of noticias">
4      <div class="card-body">
5        <h5 class="card-title">{{noticia.titulo}}</h5>
6      </div>
7    </div>
8  </div>
```

A diretiva `*ngFor` é utilizada no elemento `div.card` para apresentar a lista das notícias.

### 9.3 Comunicação assíncrona e o Observable

O `HttpClient` implementa um **modelo de programação assíncrona** que é utilizado para consumir dados de uma API HTTP e também para suportar passagem de mensagens entre **editores** (ou **publicadores**) e **assinantes**.

Esse modelo de programação permite *atrasar* a execução de um código, executando-o apenas quando for recebida uma resposta. Isso combina muito bem com o modelo de comunicação assíncrona do `XmlHttpRequest` porque os elementos do modelo de programação assíncrona podem ser traduzidos diretamente, assim:

- **publicador**: código que cria uma requisição HTTP (ex: método `get()` do `HttpClient`)
- **assinante**: método `subscribe()` do `Observable`

A Figura 9.1 ilustra ainda mais esse processo por meio de um diagrama de sequência.

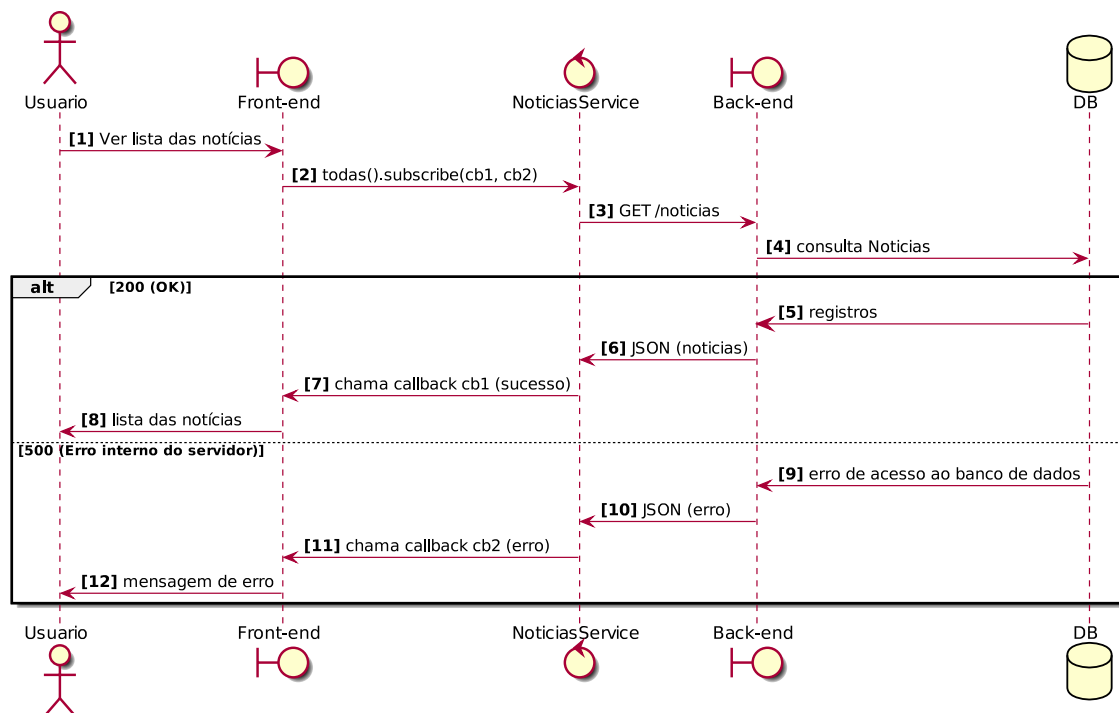


Figura 9.1: Diagrama de sequência da consulta da lista de notícias

Os passos representados no diagrama são os seguintes:

1. O **Usuário** utiliza o **Front-end** (componente) para ver a lista das notícias
2. O **Front-end** chama o método `todas()` do **NoticiasService** (o publicador) e, na sequência, chama o método `subscribe()` para **assinar** o conteúdo, executando as callbacks `cb1` ou `cb2`
3. O **NoticiasService** cria uma requisição HTTP GET para o **Back-end** e fica aguardando uma resposta
4. O **Back-end** atende a requisição e realiza uma consulta no **DB** para obter os dados
5. Se a execução no **Back-end** ocorrer com sucesso, ele vai retornar um código `200 (OK)` para o **NoticiasService**. Antes, o **DB** retorna os registros para o **Back-end**
6. O **Back-end** retorna os registros para o **NoticiasService** em formato JSON

7. O **NoticiasService** retorna a lista de notícias para a callback *cb1* na forma de lista (array de objetos, convertido do JSON)
8. O **Front-end** apresenta a lista das notícias para o **Usuário**
9. Se a execução no **Back-end** ocorrer com erro, ele vai retornar um código *500 (Erro interno do servidor)* para o **NoticiasService**. Antes, o **DB** retorna um erro para o **Back-end**
10. O **Back-end** retorna um JSON para o **NoticiasService** que representa o erro que ocorreu no servidor
11. O **NoticiasService** retorna o erro para a callback *cb2* na forma de um objeto (convertido do JSON)
12. O **Front-end** apresenta uma mensagem de erro para o **Usuário**

Importante notar que o Front-end e o NoticiasService fazem parte de um software (o próprio Front-end) e o Back-end e o DB fazem parte de outro software (o próprio Back-end). Assim, esse modelo de programação assíncrona permite a comunicação entre dois softwares independentemente das suas plataformas de execução. Por exemplo, o Front-end pode ser desenvolvido em Angular, enquanto o Back-end, em Python.

## Apêndice A

# Configuração do ambiente de desenvolvimento

### A.1 Node.js

O **Node.js** é um ambiente de execução do JavaScript independente do browser e multiplataforma (NODE.JS FOUNDATION, [s.d.]). Nos projetos desse livro é necessário utilizar **Node.js** e também a ferramenta **npm**, um gerenciador de pacotes JavaScript para o **Node.js** (NPM, INC., [s.d.]).

A instalação do **Node.js** é simples, bastando acessar <https://nodejs.org/en/download/> para obter os binários de instalação conforme a plataforma desejada. O **npm** também é fornecido junto com a instalação do **Node.js**.

Para verificar se seu ambiente de execução do **Node.js** está operando normalmente, execute os comandos a seguir em um prompt:

```
$ node -v  
$ npm -v
```

A saída dos programas apresenta, respectivamente, as versões do **Node.js** e do **npm** instaladas, como:

```
v10.5.0  
6.2.0
```

### A.2 Angular CLI

O **Angular CLI** é fornecido como um pacote **npm**, então deve ser instalado da seguinte forma:

```
$ npm install -g @angular/cli
```

O comando `install` seguido da opção `-g` faz uma *instalação global* do **Angular CLI**, o que significa que ele estará disponível para qualquer usuário.

## Apêndice B

# TypeScript



# Referências

AMSTEL, F. VAN. **Arquitetura da Informação e de Interação**, 2004. Disponível em: <[http://www.usabilidoido.com.br/arquitetura\\_da\\_informacao\\_e\\_de\\_interacao.html](http://www.usabilidoido.com.br/arquitetura_da_informacao_e_de_interacao.html)>. Acesso em: 29 jul. 2018

CORAIS.ORG. **Sitegrama**, [s.d.]. Disponível em: <<http://www.corais.org/node/184>>. Acesso em: 29 jul. 2018

GIT COMMUNITY. **Git**, [s.d.]. Disponível em: <<https://git-scm.com/>>. Acesso em: 22 jul. 2018

GOOGLE. **Angular**, [s.d.]. Disponível em: <<https://angular.io/>>. Acesso em: 22 jul. 2018a

GOOGLE. **Angular CLI**, [s.d.]. Disponível em: <<https://cli.angular.io/>>. Acesso em: 22 jul. 2018b

LIEBERMAN, B. **UML Activity Diagrams: Detailing User Interface Navigation**, abr. 2004. Disponível em: <<https://www.ibm.com/developerworks/rational/library/4697.html>>. Acesso em: 29 jul. 2018

MICROSOFT. **TypeScript - JavaScript that scales**, [s.d.]. Disponível em: <<https://www.typescriptlang.org/index.html>>. Acesso em: 24 jul. 2018a

MICROSOFT. **Visual Studio Code - Code Editing. Redefined**, [s.d.]. Disponível em: <<https://code.visualstudio.com/>>. Acesso em: 22 jul. 2018b

NODE.JS FOUNDATION. **Node.js**, [s.d.]. Disponível em: <<https://nodejs.org>>. Acesso em: 23 jul. 2018

NPM, INC. **npm**, [s.d.]. Disponível em: <<https://www.npmjs.com/>>. Acesso em: 23 jul. 2018

OGLOBLINSKY, V. **Compodoc – The missing documentation tool for your Angular application**, 2018. Disponível em: <<https://compodoc.app/>>. Acesso em: 29 jul. 2018

THE JQUERY FOUNDATION. **jQuery**, [s.d.]. Disponível em: <<http://jquery.com/>>. Acesso em: 22 jul. 2018

W3SCHOOLS. **JavaScript and HTML DOM Reference**, [s.d.]. Disponível em: <<https://www.w3schools.com/jsref/default.asp>>. Acesso em: 22 jul. 2018a

W3SCHOOLS. **JavaScript Tutorial**, [s.d.]. Disponível em: <<https://www.w3schools.com/js/default.asp>>. Acesso em: 22 jul. 2018b

W3SCHOOLS. **HTML5 Tutorial**, [s.d.]. Disponível em: <<https://www.w3schools.com/html/default.asp>>. Acesso em: 22 jul. 2018c

W3SCHOOLS. **CSS Tutorial**, [s.d.]. Disponível em: <<https://www.w3schools.com/css/default.asp>>. Acesso em: 22 jul. 2018d

W3SCHOOLS. **CSS Reference**, [s.d.]. Disponível em: <<https://www.w3schools.com/cssref/default.asp>>. Acesso em: 22 jul. 2018e

W3SCHOOLS. **HTML Element Reference**, [s.d.]. Disponível em: <<https://www.w3schools.com/tags/default.asp>>. Acesso em: 22 jul. 2018f

W3SCHOOLS. **jQuery Tutorial**, [s.d.]. Disponível em: <<https://www.w3schools.com/jquery/default.asp>>. Acesso em: 22 jul. 2018g

W3SCHOOLS. **jQuery Reference**, [s.d.]. Disponível em: <[https://www.w3schools.com/jquery/jquery\\_ref\\_overview.asp](https://www.w3schools.com/jquery/jquery_ref_overview.asp)>. Acesso em: 22 jul. 2018h

WIKIPEDIA CONTRIBUTORS. **Model–view–controller** — **Wikipedia, The Free Encyclopedia**, 2018a. Disponível em: <<https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=849850595>>. Acesso em: 23 jul. 2018

WIKIPEDIA CONTRIBUTORS. **Model–view–viewmodel** — **Wikipedia, The Free Encyclopedia**, 2018b. Disponível em: <<https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93viewmodel&oldid=851186618>>. Acesso em: 23 jul. 2018

WIKIPEDIA CONTRIBUTORS. **Dependency injection** — **Wikipedia, The Free Encyclopedia**, 2018c. Disponível em: <[https://en.wikipedia.org/w/index.php?title=Dependency\\_injection&oldid=845653474](https://en.wikipedia.org/w/index.php?title=Dependency_injection&oldid=845653474)>