Módulo 2 - Angular

Bootcamp de Desenvolvimento Front End

Danilo Ferreira e Silva

2020

Módulo 2 - Angular

Bootcamp de Desenvolvimento Front End

Danilo Ferreira e Silva

Sumário

Capítulo 1. Introdução e preparação do ambiente

Este capítulo introduz o *framework* Angular e descreve como instalar as ferramentas de desenvolvimento necessárias para acompanhar o curso. Além disso, introduzimos a linguagem TypeScript, que também será utilizada ao longo do curso.

Introdução ao Angular

Angular é um *framework* e plataforma de desenvolvido de aplicações web, utilizado para a construção da interface de usuário, ou seja, o *front end* da aplicação. Tal *framework* é um projeto de código-fonte aberto mantido por um time de desenvolvimento do Google e membros da comunidade *open-source*. Hoje, trata-se de uma das tecnologias mais utilizadas na indústria de software. Por exemplo, Angular foi apontado como o terceiro framework web mais utilizado em estudo recente do site Stack Overflow (2020).

Entre suas principais características, podemos citar:

- É baseado em componentes.
- Codificação na linguagem TypeScript.
- Ferramental completo e poderoso.
- Incentiva boas práticas de projeto e modularização do código.
- Escalabilidade e performance.

Neste curso aprenderemos em detalhes os fundamentos de Angular, com grande enfoque na prática, para que os alunos sejam capazes de criar aplicações totalmente funcionais. Antes de começar, precisaremos preparar o ambiente de desenvolvimento, instalando as ferramentas necessárias, conforme descrito nas próximas seções.

Ferramentas de desenvolvimento

Neste módulo, assumimos como pré-requisitos as seguintes ferramentas introduzidas no módulo anterior:

- **Visual Studio Code**
- **Node.js**

Portanto, se ainda não possui tais ferramentas instaladas, providencie sua instalação, conforme o material do Módulo 1. Além disso, ao longo deste módulo utilizaremos o navegador **Google Chrome** para testar as aplicações desenvolvidas.

Angular CLI

A principal ferramenta para desenvolvimento Angular é sua interface de linha de comando, ou **Angular CLI**. Para instalá-la, execute o seguinte comando em um terminal:

```
npm install -g @angular/cli@10.0.8
```

O comando acima especifica uma versão (10.0.8), para que não haja uma disparidade entre as versões do Angular utilizadas no material e pelos alunos. Note que este processo pode ser demorado, visto que diversos arquivos serão baixados da Internet. Após concluir a instalação, execute o comando `ng version` para garantir que o Angular CLI foi instalado com sucesso.

O Angular CLI é utilizado para criar novas aplicações Angular e realizar diversas tarefas durante o desenvolvimento. Por exemplo, crie uma primeira aplicação com o comando:

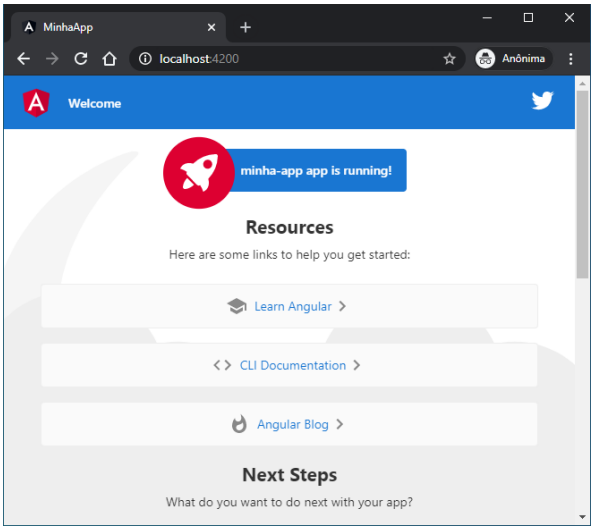
```
ng new minha-app
```

Aceite o valor padrão para todas a perguntas apertando Enter. Ao final do processo, um diretório minha-app contendo a aplicação será criado no diretório atual. Entre neste diretório e execute o comando:

```
cd minha-app
ng serve –open
```

Este comando executará sua aplicação em um servidor de desenvolvimento e a abrirá no navegador. Você verá uma tela como a abaixo:

Figura 1 – Aplicação Angular em execução



Angular Language Service Extension para o VSCode

Além do Angular CLI, é também interessante instalar a extensão Angular para Visual Studio Code. Com ela, teremos uma melhor experiência de desenvolvimento. Por exemplo, ela é capaz de apontar erros de código específicos do Angular, que não seriam detectados apenas com o Visual Studio Code puro.

Para fazer a instalação, entre na seguinte URL e clique em Install:

https://marketplace.visualstudio.com/items?itemName=Angular.ng-template

O sistema pedirá autorização para abrir o Visual Studio Code. Confirme para prosseguir com a instalação. Confirme a instalação dentro do Visual Studio Code e aguarde a conclusão.

Capítulo 2. Introdução à TypeScript

TypeScript é uma extensão da linguagem JavaScript que adiciona um sistema de verificação de tipos em tempo de compilação. Ou seja, em JavaScript erros de tipo ocorrem durante a execução caso não sejam devidamente tratados, enquanto em TypeScript erros de tipo são encontrados pelo compilador TypeScript. Com isso, o desenvolvimento se torna menos propenso a erros, trazendo maior facilidade de manutenção.

Uma das características do Angular é que ele adota a linguagem TypeScript para a codificação da aplicação. Portanto, precisamos de noções básicas desta linguagem. Esta seção não pretende ser um guia extensivo de TypeScript, mas apenas uma breve introdução. Mais detalhes serão apresentados ao longo do curso, conforme necessário.

Verificação de tipos

Todo código JavaScript é também um código TypeScript. Ou seja, TypeScript apenas adiciona novos recursos à linguagem. O mais importante destes recursos é a capacidade de declarar tipos em variáveis, parâmetros de funções, retornos de funções e em várias outras situações. Por exemplo, considere a função abaixo:

```
function soma(a: number, b: number): number {
    return a + b;
}
```

Observe que os parâmetros da função são anotados com o tipo number, assim como o tipo de retorno da função é number. Dessa forma, se tentarmos chamar tal função passando parâmetros do tipo incorreto, por exemplo:

```
soma(3, 'olá'); // erro!
```

O compilador acusaria um erro, pois é esperado que o segundo parâmetro seja number, não string. De forma semelhante, não podemos atribuir um valor do tipo

incorreto para uma variável quando declaramos seu tipo. O Visual Studio Code está preparado para lidar com código TypeScript, e exibe erros de tipo dentro do próprio editor.

Interfaces

Podemos declarar uma interface para descrever a estrutura de um objeto, por exemplo:

```
interface Aluno {
    matricula: number,
    nome: string,
    dataNascimento?: string
}
```

Cabe ressaltar que interfaces existem apenas em TypeScript, e elas desaparecem quanto o código é transformado em JavaScript. Seu propósito, portanto, é apenas a verificação de tipos.

Classes

TypeScript, assim como JavaScript moderno, aceita a sintaxe de declaração de classes, facilitando a programação orientada à objetos. Adicionalmente, podemos definir tipos para os membros da classe:

```
class Retangulo {
    altura: number;
    largura: number;
    calculaArea(): number {
        // ...
    }
}
```

Decorators

Decorators são funções que modificam declarações da linguagem, como classes, propriedades de uma classe, parâmetros de funções, entre outras. O Angular utiliza decorators para definir uma série de configurações. Neste curso não vamos definir decorators, precisaremos apenas utilizar aqueles já fornecidos pelo Angular. A sintaxe para utilizá-los é parecida com uma chamada de função, mas precedida pelo caractere @, por exemplo:

```
@Component({
    selector: 'meu-componente'
})
class MeuComponente {}
```

Capítulo 3. Arquitetura e estrutura da aplicação

Aplicações angular seguem a arquitetura *Single Page Application* (SPA), o que significa que elas são constituídas de um único documento HTML contendo o JavaScript de toda a aplicação. Comunicações com o Back End ocorrem apenas para buscar dados. Portanto, o Angular CLI compila todos os arquivos do projeto em um único HTML e um conjunto de scripts e estilos.

Estrutura física do projeto

Após criar o projeto com o comando **ng new**, os seguintes diretórios serão criados por padrão:

- **e2e**: Diretório para armazenar testes *end-to-end*. Este assunto não será abordado no curso.
- **node_modules**: Diretório com os pacotes Node.js necessários para o desenvolvimento. Este diretório não deve ser alterado manualmente.
- **src**: Diretório com o código fonte. Além disso, este diretório contém o index.html, ponto de entrada de aplicação.

Além disso, na raiz do projeto existe uma série de arquivos de configuração. Porém, a configuração presente já é suficiente para o nosso curso.

Estrutura lógica do projeto

Módulos

Uma aplicação Angular é organizada em módulos, que são definidos pelo decorator **@NgModule**. Toda aplicação deve ter ao menos um módulo principal, que normalmente fica em **app/app.module.ts**. Um módulo deve informar as declarações que o compõem, e também pode importar outros módulos, como no exemplo abaixo:

```
@NgModule({
  declarations: [AppComponent, PanelComponent],
  imports: [BrowserModule],
  bootstrap: [AppComponent]
```

```
})
export class AppModule { }
```

Note que toda aplicação que executa no navegador deve importar BrowserModule.

Componentes

Componentes são o mais importante conceito do Angular. A aplicação é construída por meio da declaração e uso de componentes, partindo de um componente principal que pode usar outros componentes, formando uma hierarquia. Os componentes devem ser incluídos em um módulo para serem utilizados. Toda aplicação deve ter ao menos o componente principal (que normalmente fica em **app/app.component.ts**).

Capítulo 4. Componentes e templates

Declaração de componentes

Componentes são declarados por meio de uma classe anotada com o decorator @Component. Por exemplo:

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

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

Todo componente deve ter um **selector**, que é o nome da tag que corresponderá ao elemento na página. Além disso, o componente deve ter um *template*, que é responsável por sua exibição. O *template* pode ser especificado pela configuração **templateUrl**, informando o caminho de um arquivo, ou pela configuração **template**, informando o conteúdo do *template* como uma string.

A maneira mais prática de adicionar um componente à uma aplicação é com o Angular CLI, por meio do comando:

```
ng generate component nomeComponente
```

Templates

Templates possuem sintaxe semelhante a HTML, mas com algumas extensões do Angular, conforme exemplo abaixo:

```
<h1>Este é um template</h1>
<p>Olá {{nome}}</p>
<img [src]="urlFoto" />
<button (click)="umaFuncao()">Clique aqui</button>
```

Note que no *template* de exemplo temos, além da sintaxe HTML, outras três construções específicas do angular, descritas a seguir. Estes recursos são denominados no Angular pelo termo mais geral *data/event binding*.

Interpolação

Usa a sintaxe `{{expressao}}`. Uma expressão Angular pode acessar propriedades ou funções do componente, usar operadores lógicos e aritméticos, entre outros recursos. No entanto, essa expressão não pode executar um código JavaScript arbitrário.

Property binding

Usa a sintaxe `[property]="expressao"`. Com isso é possível definir propriedades de tags HTML dinamicamente. É importante ressaltar que deve-se usar o nome da propriedade HTML, não de um atributo HTML (e nem todo atributo HTML tem uma propriedade com o mesmo nome).

Também podemos usar a sintaxe alternativa `bind-property="expressao"` ou mesmo a interpolação `property="{{expressao}}"`. Porém, o uso de interpolação só é válido se a propriedade esperar uma string.

Event binding

Usa a sintaxe `(event)="comando"`. Com isso, é possível tratar eventos chamando funções (métodos) do componente. Também é possível fazer atribuições em propriedades do componente. Nestes comandos, existirá sempre a variável implícita **$event** para capturar o evento disparado.

Capítulo 5. Diretivas estruturais e de atributos

Diretivas alteram ou adicionam comportamento a elementos HTML ou componentes Angular. Os próprios componentes tecnicamente são tipos de diretivas, mas por sua importância são tratados com outros termo. Existem dois tipos principais de diretivas:

- **Diretivas de atributos**: Alteram ou adicionam comportamento. Sua sintaxe é semelhante a um atributo da tag.
- **Diretivas estruturais**: Alteram a estrutura, removendo ou repetindo elementos na página, por exemplo. Sua sintaxe sempre começa com o caractere *. Diretivas estruturais são similares a estruturas de controle em linguagens de programação: IF, FOR, SWITCH, etc.

*ngIf

Esta diretiva permite a inclusão condicional de um componente ou tag HTML com base numa expressão booleana, por exemplo:

```
<div *ngIf="situacao == 'enviado'">
    Seu produto foi enviado para entrega.
</div>
```

*ngFor

Esta diretiva permite repetir um componente ou tag HTML iterando em um array. A diretiva permite capturar o item atual e o índice da iteração, definindo variáveis acessíveis no escopo do elementos englobados pela diretiva, por exemplo:

```
<table>
    <tr *ngFor="let task of tasks; let i = index">
        <td>{{i + 1}}</td>
        <td>{{task.description}}</td>
    </tr>
</table>
```

ngSwitch, *ngSwitchCase e *ngSwitchDefault

Com estas diretivas é possível criar uma estrutura de decisão com múltiplas alternativas, semelhante ao comando switch da linguagem JavaScript, por exemplo:

```
<div [ngSwitch]="tasks.length">
    <div *ngSwitchCase="0">Nenhuma tarefa</div>
    <div *ngSwitchCase="1">Uma tarefa</div>
    <div *ngSwitchDefault>{{tasks.length}} tarefas</div>
</div>
```

ng-container

Não é possível utilizar mais de uma diretiva estrutural na mesma tag ou elemento. Para esse tipo de caso, podemos usar a tag ng-container, que cria um elemento “virtual” que pode ter diretivas estruturais, mas não gera um elemento real na página:

```
<table>
    <tr *ngFor="let task of tasks; let i = index">
        <ng-container *ngIf="!task.done">
            <td>{{task.description}}</td>
        </ng-container>
    </tr>
</table>
```

Capítulo 6. Comunicação entre componentes

Input properties

Por meio do decorator **@Input** é possível indicar uma propriedade do componente como input. Dessa forma, um componente pai pode passar parâmetros para o componente por meio de property binding. Por exemplo, supondo que nosso componente foi declarado como:

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-rating',
  templateUrl: './rating.component.html'
})
export class RatingComponent {
  @Input() rating: number;
}
```

É possível utilizá-lo passando rating como parâmetro:

```
<app-rating [rating]="movie.rating"></app-rating>
```

Output properties

Output properties são eventos emitidos por um componente, permitindo que seu componente pai os capture com event binding. Por exemplo, supondo o seguinte componente:

```
import { Component, EventEmitter, Output } from '@angular/core';
@Component({
  selector: 'app-confirm',
  templateUrl: './rating.component.html',
})
export class ConfirmComponent {
  @Output() confirm = new EventEmitter<boolean>();
  onClickOk() { this.confirm.emit(true); }
  onClickCancel() { this.confirm.emit(false); }
}
```

É possível utilizá-lo capturando o evento confirm, conforme o exemplo abaixo. A variável $event conterá o valor passado na chamada a emit de EventEmitter.

```
<app-confirm (confirm)="handleConfirmation($event)"></app-confirm>
```

Two-way binding

Para casos onde um componente recebe uma input property e emite alterações nesta propriedade por meio de output property, podemos usar o sintaxe especial denominada two-way binding:

```
<app-rating [(rating)]="movie.rating"></app-rating></td>
```

Para tanto, é necessário que o componente em questão use o sufixo **Change** no nome da output property. Por exemplo, se a input property chama rating, a output property deve se chamar ratingChange.

ngModel

Os elementos HTML para entrada de dados em formulário normalmente possuem a propriedade value, mas não possuem um evento valueChange, necessário para uso da sintaxe two-way binding. No entanto, para facilitar o tratamento de formulários, o Angular provê as diretivas ngModel e ngModelChange, permitindo o uso da sintaxe:

```
<input type="number" [(ngModel)]="age" />
```

Para usar este recurso, é necessário primeiro importar FormsModule:

```
// outros imports
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [
    BrowserModule,
    FormsModule // import necessário para usar ngModel
  ]
```

```
})
```

ng-content

Em certos casos, precisamos passar para um componente outros componentes ou elementos HTML. Para isso podemos usar a tag ng-content. Por exemplo, dado que temos o seguinte conteúdo:

```
<panel title="Painel 2">
  <p>Aqui temos um conteúdo qualquer</p>
  <img src="foto.jpg" />
</panel>
```

O componente panel pode exibir o conteúdo passando dentro de sua tag usando ng-content, conforme o template abaixo:

```
<h2>{{title}}</h2>
<ng-content></ng-content> <!-- aqui aparecerá o conteúdo da tag panel -->
```

Capítulo 7. Estilização de componentes

Especificando folhas de estilo

Existem três maneiras principais de definir folhas de estilo CSS:

- **CSS global**: definido no arquivo arc/styles.css. O CSS deste arquivo não possui nenhum tratamento especial dado pelo Angular.

- **CSS de componente**: definido pela propriedade **styleUrls** do decorator **@Component,** especificando o caminho dos arquivos. Regras de estilo definidas nestes arquivos só se aplicam aos elementos no template no componente, mesmo que o seletor CSS seja amplo.

- **CSS de componente *inline***: definido pela propriedade **styles** do decorator **@Component**, especificando as próprias regras CSS como string. Também possuem escopo do componente.

Class binding

O Angular provê 4 diferentes formas de definir a propriedade class de elementos HTML dinamicamente, via *data binding*:

`[class]="exprString"`	Expressão que retorna uma string que contém as classes CSS separadas por espaço, por exemplo: `'classe1 classe2'`.
`[class]="exprArray"`	Expressão que retorna um array de string com cada classe CSS, por exemplo: `['classe1', 'classe2']`.
`[class]="exprObjeto"`	Expressão que retorna um objeto, onde cada chave é uma classe CSS e o valor é um boolean indicando se ela será aplicada ou

	não, por exemplo: `{classe1: true, classe2: false}`.
`[class.classe1]="exprBoolean"`	Expressão retorna boolean, indicando se a classe CSS em questão será aplicada.

Style binding

O Angular provê 4 diferentes formas de definir a propriedade style de elementos HTML dinamicamente, via *data binding*:

`[style]="exprString"`	Expressão que retorna uma string que contém as regras de estilo separadas por ponto e vírgula (como o padrão de HTML), por exemplo: `'width: 100px; height: 50px'`.
`[style]="exprObjeto"`	Expressão que retorna um objeto onde cada chave é propriedade CSS e o valor é uma string, por exemplo: `{width: '100px', height: '50px'}`.
`[style.width]="exprString"`	Expressão string com o valor da propriedade em questão, por exemplo: `'100px'`.
`[style.width.px]="exprNumber"`	Expressão number com o valor da propriedade em questão, sem precisar especificar a unidade de medida (que já está definida dentro dos colchetes).

Capítulo 8. Ciclo de vida de componentes

O Angular é responsável por controlar o ciclo de vida dos componentes, incluindo sua construção e destruição. É possível interceptar momentos do ciclo de vida do componente através de *hooks* fornecidos pelo Angular. Entre eles, os principais serão descritos a seguir. A lista completa pode ser consultada em: https://angular.io/guide/lifecycle-hooks.

ngOnInit

É chamado uma única vez na inicialização do componente. Este hook deve ser usado para inicializar propriedades do componente, fazer requisições HTTP, registrar *listener* de eventos na página, ou qualquer outra tarefa similar. Implemente a interface **OnInit** para interceptar este hook. Note que não podemos usar o construtor para tais tarefas, pois o componente ainda não terá suas input properties definidas.

ngOnDestroy

É chamado uma única vez antes de destruir o componente. Este hook deve ser usado para liberar recursos alocados pelo componente, como *listener* de eventos, timers, entre outros. Implemente a interface **OnDestroy** para interceptar este hook. Note que deixar de liberar recursos ao destruir o componente pode causar vazamentos de memória ou problemas de desempenho graves na aplicação, dependendo da situação.

ngOnChanges

É chamado toda vez que o valor de algum input property do componente for alterado. Este hook pode ser usado para reagir a estas mudanças, recarregando dados do Back End, recomputando valores, ou executando qualquer outra tarefa necessária. Implemente a interface **OnChanges** para interceptar este hook. É possível inspecionar o objeto passado como parâmetro para o método **ngOnChanges** para obter as propriedades modificadas e seus valores anteriores.

Capítulo 9. Pipes

Pipes são como funções utilitárias disponíveis em templates para formatar ou realizar qualquer outro tipo de transformação de valores.

Declarando um pipe

Pipes são como funções utilitárias disponíveis em templates para formatar ou realizar qualquer outro tipo de transformação de valores. Um pipe deve ser declarado com o decorator **@Pipe** e implementar a interface **PipeTransform**. No exemplo abaixo temos um pipe que formata o CEP, adicionando o caractere hífen antes dos três últimos dígitos:

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'cep'
})
export class CepPipe implements PipeTransform {
  transform(cep: string) {
    return cep.length == 8 ? cep.substring(0, 5) + '-
' + cep.substring(5) : cep;
  }
}
```

Note que, assim como componentes, pipes precisam ser adicionados a um módulo antes de serem utilizados. A maneira mais prática de criar um pipe é por meio do comando **`ng generate pipe nomeDoPipe`**.

A sintaxe para utilizar um pipe usa o caractere |, como o exemplo abaixo:

```
<p>CEP: {{meuCep | cep}}</p>
```

Um pipe pode receber parâmetros adicionais (além do valor à esquerda do operador pipe), para isso basta passá-los separados pelo caractere :, conforme exemplo a seguir:

```
<p>Exemplo: {{valor | pipeComParam:param1:param2}}</p>
```

Pipes pré-definidos

O Angular provê pipes para casos comum de formatação de valores, como data e números. Abaixo temos alguns exemplos de pipes pré-definidos:

{{text | lowercase}}	Transforma uma string em caixa baixa.
{{text | uppercase}}	Transforma uma string em caixa alta.
{{n | number:'8.0-2'}}	Formata um number, especificando no mínimo 8 dígitos antes da vírgula, e de 0 a 2 dígitos depois da vírgula.
{{n | number:'1.1-1'}}	Formata um number, especificando no mínimo 1 dígito antes da vírgula, e exatamente 1 depois da vírgula.
{{hoje | date:'short'}}	Formata data e hora no padrão curto, exemplo 01/10/20, 14:00.
{{hoje | date:'H:mm'}}	Formata data e hora, exibindo apenas a hora e minuto, conforme especificado.

Configuração de localização

Pipes como number ou date precisam de informações de localização para se adequarem ao idioma desejado. Por padrão, o Angular utiliza o inglês americano (en-US). Para trocar para português, faça as seguintes configurações no **app.module.ts**:

```
import { LOCALE_ID } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localePt from '@angular/common/locales/pt';
registerLocaleData(localePt); // <- registra os dados de localização para pt

@NgModule({
  providers: [
    {provide: LOCALE_ID, useValue: 'pt' } // <- configura o idioma padrão
  ],
  // outras configurações
})
export class AppModule { }
```

Capítulo 10. Serviços e injeção de dependências

O Angular provê a possibilidade de definir serviços, que são classes com dados ou lógica, cujo ciclo de vida é controlado pelo Angular. Serviços podem ser utilizados (injetados) em componentes, pipes, ou em outros serviços.

Declarando um serviço

Defina uma classe anotada com o decorator **@Injectable**, por exemplo:

```
import { Injectable } from '@angular/core';
@Injectable({
    providedIn: 'root'
})
export class TodoListService {
    items: string[] = ['item 1'];
    add(newItem: string) {
        this.items.push(newItem);
    }
}
```

Tal serviço pode ser utilizado (injetado) em um componente declarando um parâmetro em seu construtor cujo tipo é a classe do serviço, por exemplo:

```
export class AppComponent {
  constructor(private todoListService: TodoListService) {
    //
  }
}
```

A maneira mais prática de criar um novo serviço é via Angular CLI, usando o comando **`ng generate service nomeServico`**.

Injectors e providers

Injectors são objetos que gerenciam as instâncias de serviços ou valores arbitrários registrados neles. Toda aplicação possui o injector principal (denominado

de root), além de injectors específicos de cada módulo e de cada componente. Quando um serviço é registrado no injector root, uma única instância dele é compartilhada por toda a aplicação. Por outro lado, se o serviço é registrado em um componente, a instância do serviço é específica deste componente.

Injectors criam instâncias dos valores gerenciados (serviços ou outros) por meio de um provider. Um provider é um objeto que define uma chave identificadora do valor e uma configuração de como obter o valor. Classes de serviços podem ser utilizadas diretamente como providers, neste caso a própria classe é usada como chave e o valor obtido instanciando a classe.

Módulos e componentes podem receber a configuração providers (nos decorators @NgModule e @Component, respectivamente). Com isso, os valores serão registrados em seus respectivos injectors. Quando um componente injeta um serviço ou valor, o Angular o procura em toda a hierarquia de injectors, até chegar na raiz da aplicação.

Capítulo 11. Formulários

Além da manipulação de formulários via ngModel, o Angular provê o módulo **ReactiveFormsModule**, que oferece recursos mais avançados de testes e validação de formulários. Este módulo deve ser importado conforme exemplo abaixo:

```
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule // <- import necessário
  ],
  // outras configuracoes
})
export class AppModule { }
```

Utilização de FormGroup e FormControl

As classes FormGroup e FormControl devem ser usadas para declarar a estrutura do formulário, especificando os nomes dos campos e valores iniciais, conforme exemplo abaixo:

```
export class AppComponent {
  myForm = new FormGroup({
    name: new FormControl(''),
    address: new FormControl('')
  });
  onSubmit() {
    console.log(this.myForm.value);
  }
}
```

Feito isso, faça sua ligação com o template usando a diretiva formGroup na tag form e formControlName nas tags input:

```
<form #name [formGroup]="myForm" (ngSubmit)="onSubmit()">
  <label>Nome: <input formControlName="name" /></label>
  <label>Endereço: <input formControlName="address" /></label>
  <input [disabled]="myForm.invalid" type="Submit" value="Salvar" />
</form>
```

Podemos obter o valor de um FormGroup ou FormControl pela propriedade value. Além disso, podemos obter o status do formulário e de cada controle.

Validadores

Podemos adicionar validadores pré-definidos em um FormControl, ou mesmo criar nossos próprios validadores implementando uma função que retorna **ValidatorFn**, conforme exemplo:

```
export class AppComponent {
  myForm = new FormGroup({
    name: new FormControl('', [Validators.required, forbiddenNameValidator('Da
nilo')]),
    address: new FormControl('')
  });
}

function forbiddenNameValidator(invalidName: string): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} | null => {
    if (control.value === invalidName) {
      return {forbiddenName: {value: control.value}};
    } else {
      return null;
    }
  };
}
```

Capítulo 12. Roteamento

Aplicações na arquitetura Single Page Application (SPA) devem prover roteamento no Front End para permitir o funcionamento adequado dos controles de navegação (voltar) e histórico. O Angular inclui suporte completo à roteamento, que pode ser ativado ao criar um aplicação via `ng new`. Quando o roteamento é habilitado, é criado o módulo **app-routing.module.ts** por padrão, no qual podem ser definidas rotas. O conteúdo de cada roda será exibido dentra da tag <router-outlet>, que deve ser adicionada no componente principal da aplicação no local desejado.

Definindo rotas

Rotas são definidas especificando um caminho (**path**) e um componente. Além disso, é possível definir regras de redirecionamento, conforme exemplo abaixo:

```
const routes: Routes = [
  {path: 'page1', component: Page1Component},
  {path: '', redirectTo: 'page1', pathMatch: 'full'},
  {path: 'page2/:id', component: Page2Component}
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Links para rotas

Para criar um link para uma rota, deve-se utilizar a diretiva **routerLink**:

```
<a routerLink="/page1" routerLinkActive="active">Page 1</a>
```

Caso a rota possua parâmetros, pode ser especificada como um array informando cada parte da rota separadamente (não é preciso separar com barras):

```
<a [routerLink]="['/page2', 2]" routerLinkActive="active">Page 2 (id=2)</a>
```

Também é possível navegar para outra tela programaticamente, injetando o serviço **Router** no componente:

```
this.router.navigate(['page1']);
```

Acessando parâmetros da rota

Quando definimos um rota com parâmetro, por exemplo `'page2/:id'`, precisamos injetar o serviço **ActivatedRoute** para obter os valores dos parâmetros. Tal serviço expõe uma mapa de parâmetros pela propriedade **paramMap**. Porém, como os parâmetros podem mudar, esta propriedade tem o tipo **Observable** (este tipo é definido pela biblioteca RxJs, usada internamente pelo Angular). Para obter seu valor, devemos chamar **subscribe**, passando um *callback* que será executado toda vez que algum parâmetro mudar:

```
export class Page2Component implements OnInit {
  constructor(private route: ActivatedRoute) { }
  id: string;
  ngOnInit(): void {
    this.route.paramMap.subscribe(paramMap => {
      this.id = paramMap.get('id');
    });
  }
}
```

Note que não é necessário desfazer a chamada a subscribe no ngOnDestroy, pois a rota é destruída quando se torna desativada, removendo todos os listener de eventos registrados pelo componente automaticamente.

Observe também que qualquer parâmetro obtido terá o tipo string. Portanto, devem ser feitas as conversões necessárias se for o caso.

Capítulo 13. Comunicação com o Back End

O Angular provê o serviço **HttpClient** para fazer comunicação com o Back End. Antes de usar o serviço, é necessário importar **HttpClientModule**:

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    HttpClientModule // <- import necessário
  ],
  // outras configurações
})
export class AppModule { }
```

O serviço HttpClient pode ser injetado em componentes e serviços como usual, e provê métodos para fazer requisições usando cada verbo HTTP (get, post, delete, put, etc.). No entanto, todas essas chamadas devolvem um **Observable**, pois como a requisição ocorre de forma assíncrona, o resultado deve ser obtido por meio de um call-back passado para o método **subscribe**, conforme exemplo a seguir:

```
export class AppComponent implements OnInit {
  produtos: Produto[] = [];
  constructor(private httpClient: HttpClient) { }

  ngOnInit(): void {
    this.httpClient.get<Produto[]>(`${urlBase}/produtos`)
      .subscribe(dados => {
        this.produtos = dados;
      });
  }
  adicionaProduto(produto: Produto) {
    this.httpClient.post<Produto>(`${urlBase}/produtos`, produto)
      .subscribe(novoProduto => {
        this.produtos = [...this.produtos, novoProduto];
      });
  }
}
```

Note que este padrão é semelhante ao da API fetch disponível no navegador, que retorna uma **Promise**. É importante ressaltar que a requisição HTTP só é feita após chamar subscribe.

Embora o HttpClient por padrão trabalhe com requisições que enviam e recebem dados JSON, é possível passar opções adicionais para ter total controle de como desejamos enviar a requisição e tratar a resposta. Por exemplo, podemos definir headers, pegar o *status code* da resposta, obter o progresso da requisição, entre outras tarefas mais avançadas.

Capítulo 14. Biblioteca de componentes Angular Material

Angular Material é uma biblioteca de componentes de alta qualidade baseada no padrão Material Design da Google. Usando esta biblioteca, temos à disposição um catálogo de componentes visuais que facilitam o desenvolvimento, evitando que "reinventemos a roda" em toda aplicação. Para adicionar dependência para uma biblioteca, use o comando ng add nomeBiblioteca do Angular CLI. Especificamente, o Angular Material é adicionado no projeto com o comando:

```
ng add @angular/material
```

Com isso, as dependências necessárias são adicionadas ao node_modules e alguns ajustes necessários são feitos no projeto. Em particular, os arquivo index.html e styles.css são modificados para adicionar estilos necessários, como a fonte Roboto e a biblioteca de ícones Material Icons.

Para usar um componente do Angular Material, primeiro devemos importar os módulos necessários. Por exemplo, estes são os imports para usar os componentes Button e Toolbar:

```
import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';

@NgModule({
  imports: [
    MatButtonModule,
    MatToolbarModule, // <- O módulo de cada componente deve ser importado
    // outros imports
  ],
})
export class AppModule { }
```

A maneira de utilizar os componentes é diferente para cada caso. Por exemplo, o componente Toolbar é usado com a tag `<mat-toolbar>`:

```
<mat-toolbar color="primary">
  <span>Cardápio</span>
</mat-toolbar>
```

Por outro lado, outros componentes na verdade são apenas diretivas aplicadas a elementos HTML, como é o caso do Button:

```
<button mat-icon-button (click)="excluir(produto)" aria-label="Excluir">
  <mat-icon>delete</mat-icon>
</button>
```

Os componentes normalmente podem ser customizados de diferentes formas, atendendo a vários casos de uso. Consulte a documentação para ver exemplos de código e explorar as configurações de cada componente:

https://material.angular.io/components/categories

Referências

GOOGLE. *Angular Docs, 2020a.* Documentação oficial do Angular. Disponível em https://angular.io/docs. Acesso em: 8 set. 2020.

GOOGLE. *Angular Material, 2020b*. Biblioteca de componentes de UI. Disponível em https://material.angular.io/. Acesso em 8 set. 2020.

MICROSOFT. *TypeScript Documentation, 2020*. Documentação oficial da linguagem TypeScript. Disponível em https://www.typescriptlang.org/docs. Acesso em 8 set. 2020.

STACK OVERFLOW. *Stack Overflow Developer Survey, 2020*. Disponível em https://insights.stackoverflow.com/survey/2020. Acesso em 8 set. 2020.