1 Introdução

O desenvolvimento moderno de aplicações Web é comumente realizado utilizando-se diferentes tecnologias. Muitas vezes, sua combinação dá origem a algo que tem levado o nome de desenvolvimento “Full Stack”. Uma solução desenvolvida segundo esse paradigma possui, em geral, duas aplicações independentes que se comunicam por meio de uma interface bem definida. Uma delas oferece interfaces gráfica para o usuário e geralmente é chamada de aplicação Front End. A outra é responsável por disponibilizar as funcionalidades do sistema e leva o nome de aplicação Back End. Em geral, ela faz uso de um sistema independente que possui implementações eficientes para operações de acesso à memória secundária.

Além disso, seja no Front End ou no Back End, é comum o uso de diferentes *frameworks* que supostamente entregam um nível maior de abstração e promovem a produtividade dos desenvolvedores.

Nos dias atuais, uma das combinações mais utilizadas para esse fim tem a sigla “**MEAN**” associada, a qual deriva de “**MongoDB**”, “**Express**”, “**Angular**” e “**NodeJS**”.

Neste material desenvolveremos uma aplicação que faz uso da “pilha” MEAN.

2 Desenvolvimento

2.1 (Adicionando forms) Neste momento iremos substituir o two way data binding pelo uso de forms. Isso pode ser de interesse pois tende a facilitar atividades com a validação de dados, por exemplo.

- Abra o arquivo **cliente-inserir.component.html**. Os elementos de entrada de dados serão englobados por uma tag form. O uso da diretiva ngModel será alterado, descaracterizando o uso do two way data binding. Adicionaremos o atributo **name** para que o Angular possa adicionar os campos ao objeto JSON que ele irá gerar ao interceptar a submissão do form. Também alteraremos o botão para que ele faça a submissão. Usaremos **event binding** para lidar com esse evento. Veja a Listagem 2.1.1.

Listagem 2.1.1

```
<mat-card>
  <form (submit)="onAdicionarCliente()">
    <mat-form-field>
      <input type="text" matInput placeholder="nome" name="nome" ngModel />
    </mat-form-field>
    <mat-form-field>
      <input type="text" matInput placeholder="fone" name="fone" ngModel />
    </mat-form-field>
    <mat-form-field>
      <input type="text" matInput placeholder="email" name="email" ngModel />
    </mat-form-field>
    <button color="primary" mat-raised-button>
      Inserir Cliente
    </button>
  </form>
</mat-card>
```

- Para acessar os dados do objeto JSON gerado pelo Angular, vamos associar uma variável de referência ao form e atribuir a ela a diretiva **ngForm**. A seguir, entregamos ela ao método. Veja a Listagem 2.1.2.

Listagem 2.1.2

```
<form (submit)="onAdicionarCliente(clienteForm)" #clienteForm="ngForm">
```

- A seguir, o método onAdicionarCliente precisa ser alterado no arquivo **cliente-inserir.component.ts** a fim de receber o novo parâmetro adequadamente. Ele usa a propriedade value do parâmetro recebido para obter os valores a serem utilizado na construção do objeto cliente. Veja a Listagem 2.1.3.

Listagem 2.1.3

```
onAdicionarCliente(form: NgForm) {
   const cliente: Cliente = {
     nome: form.value.nome,
     fone: form.value.fone,
     email: form.value.email,
   };
   this.clienteAdicionado.emit(cliente);
 }
```

- Podemos aplicar algumas validações diretamente no form. Veja a Listagem 2.1.4.

Listagem 2.1.4

```
<mat-card>
  <form (submit)="onAdicionarCliente(clienteForm)" #clienteForm="ngForm">
    <mat-form-field>
      <input
        required
        minlength="4"
        type="text"
        matInput
        placeholder="nome"
        name="nome"
        ngModel
      />
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="fone"
        name="fone"
        ngModel
      />
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="email"
        name="email"
        ngModel
      />
    </mat-form-field>
    <button color="primary" mat-raised-button>
      Inserir Cliente
    </button>
  </form>
</mat-card>
```

- Clique no botão **Inserir Cliente** para ver as validações ocorrendo. Note que a inserção ocorre mesmo que alguma validação falhe. Ocorre que todo form possui uma propriedade chamada **invalid** que somente é verdadeira caso todos os seus controles estejam de acordo com todas as validações aplicadas a eles. Podemos testar essa propriedade antes de fazer a inserção no método **onAdicionarCliente**. Veja a Listagem 2.1.5.

Listagem 2.1.5

```
onAdicionarCliente(form: NgForm) {
   if (form.invalid) {
    return;
   }
   const cliente: Cliente = {
    nome: form.value.nome,
    fone: form.value.fone,
    email: form.value.email,
   };
   this.clienteAdicionado.emit(cliente);
 }
```

- A biblioteca Angular Material possui componentes apropriados para a exibição de mensagens de erro. Ela se chama **mat-error**. Decidimos sobre a sua aparição usando uma diretiva **ngIf**. Para acessar um elemento do form, aplicamos a ele uma variável de referência, atruibuindo a diretiva **ngModel** a ela. Veja a Listagem 2.1.6.

Listagem 2.1.6

```
<mat-card>
  <form (submit)="onAdicionarCliente(clienteForm)" #clienteForm="ngForm">
    <mat-form-field>
      <input
        required
        minlength="4"
        type="text"
        matInput
        placeholder="nome"
        name="nome"
        ngModel
        #nome="ngModel"
      />
      <mat-error *ngIf="nome.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="fone"
        name="fone"
        ngModel
        #fone="ngModel"
      />
      <mat-error *ngIf="fone.invalid">Digite um telefone válido</mat-error>
    </mat-form-field>
    <mat-form-field>
      <input
        required
        type="text"
        matInput
        placeholder="email"
        name="email"
        ngModel
        #email="ngModel"
      />
      <mat-error *ngIf="email.invalid">Digite um nome válido</mat-error>
    </mat-form-field>
    <button color="primary" mat-raised-button>
      Inserir Cliente
    </button>
  </form>
</mat-card>
```

2.2 (Adicionando um serviço à aplicação) O acesso à lista de clientes pode ser centralizado em um serviço, o que pode facilitar o acesso à ela por diferentes componentes. Assim, por exemplo, não será necessário fazer com que o componente que o componente principal entregue a lista de clientes ao componente que a exibe. Cada um deles terá uma referência a uma instância do serviço.

- Na pasta **clientes**, crie um novo arquivo chamado **cliente.service.ts**. Um serviço pode ser implementado como uma classe Typescript. Ela terá uma lista e um método que dá acesso a uma cópia dela. Veja a Listagem 2.2.1.

Listagem 2.2.1

```
import { Cliente } from './cliente.model';

export class ClienteService {
  private clientes: Cliente[] = [];

  getClientes(): Cliente[] {
    return [...this.clientes];
  }
}
```

- Também cabe ao serviço fazer a adição de clientes. O método da Listagem 2.2.2 se encarrega dessa tarefa.

Listagem 2.2.2

```
adicionarCliente(nome: string, fone: string, email: string) {
    const cliente: Cliente = {
      nome: nome,
      fone: fone,
      email: email,
    };
    this.clientes.push(cliente);
  }
```

- A seguir, precisamos injetar uma instância do serviço no componente responsável por exibir os clientes, definido no arquivo **cliente-lista.component.ts**. A injeção de dependências sempre ocorre no construtor do componente. Veja a Listagem 2.2.3.

Listagem 2.2.3

```
import { Component, OnInit, Input } from '@angular/core';
import { Cliente } from '../cliente.model';
import { ClienteService } from '../clientes.service';

@Component({
  selector: 'app-cliente-lista',
  templateUrl: './cliente-lista.component.html',
  styleUrls: ['./cliente-lista.component.css'],
})
export class ClienteListaComponent implements OnInit {
  @Input() clientes: Cliente[] = [];

  constructor(public clienteService: ClienteService) {}

  ngOnInit(): void {}
}
```

- Teste a aplicação agora e veja que ela não funciona. Abra o Chrome Dev Tools para ver a mensagem de erro. Ela diz que não há provider para o serviço. Para resolver esse problema, abra o arquivo **app.module.ts** e adicione a classe **ClienteService** ao vetor **providers**, como na Listagem 2.2.4.

Listagem 2.2.4

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

import { MatInputModule } from '@angular/material/input';
import { MatCardModule } from '@angular/material/card';
import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatExpansionModule } from '@angular/material/expansion';

import { AppComponent } from './app.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
import { CabecalhoComponent } from './cabecalho/cabecalho.component';
import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';

import { ClienteService } from './clientes/clientes.service';

@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,
    MatToolbarModule,
    MatExpansionModule,
  ],
  providers: [ClienteService],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Nota: Também é possível deixar de colocar a classe ClienteService no vetor providers e anotá-la com **Injectable**, como mostra a Listagem 2.2.5. Há uma hierarquia de injetores de dependência e podemos escolher a partir de qual componente a injeção ocorrerá, o que tem a ver com a propriedade **providedIn**. Estudaremos ela mais à frente.

Listagem 2.2.5

```
@Injectable({ providedIn: 'root' })
export class ClienteService {
```

Vamos manter o uso da anotação **Injectable**.

- Feita a injeção de dependência, o componente já pode utilizar a lista que o serviço oferece. Para isso, usaremos o método **ngOnInit** da interface **OnInit** que precisa ser implementada pela classe. Ele executa automaticamente assim que o componente é construído. É recomendável utilizá-lo para esse fim e reservar o construtor para as injeções de dependência. A Listagem 2.2.6 mostra seu uso.

Listagem 2.2.6

```
import { Component, OnInit, Input } from '@angular/core';
import { Cliente } from '../cliente.model';
import { ClienteService } from '../clientes.service';

@Component({
  selector: 'app-cliente-lista',
  templateUrl: './cliente-lista.component.html',
  styleUrls: ['./cliente-lista.component.css'],
})
export class ClienteListaComponent implements OnInit {
  @Input() clientes: Cliente[] = [];

  constructor(public clienteService: ClienteService) {}

  ngOnInit(): void {
    this.clientes = this.clienteService.getClientes();
  }
}
```

- A seguir, precisamos ajustar o componente de inserção de clientes no arquivo **cliente-inserir.component.ts**. Ele precisa fazer a inserção de clientes na lista mantida pelo serviço. Por isso, vamos injetar uma instância do serviço em seu construtor. Note que ele já não precisa emitir o evento que indica que um cliente foi inserido, por isso, vamos remover esse código, como na Listagem 2.2.7.

Listagem 2.2.7

```
import { Component, EventEmitter, Output } from '@angular/core';
//remover
import { Cliente } from '../cliente.model';
import { NgForm } from '@angular/forms';
import { ClienteService } from '../clientes.service';
@Component({
  selector: 'app-cliente-inserir',
  templateUrl: './cliente-inserir.component.html',
  styleUrls: ['./cliente-inserir.component.css'],
})
export class ClienteInserirComponent {
  constructor(public clienteService: ClienteService) {}
  //remover
  @Output() clienteAdicionado = new EventEmitter<Cliente>();
  //remover
  nome: string;
  fone: string;
  email: string;
  onAdicionarCliente(form: NgForm) {
    if (form.invalid) {
      return;
    }
    //remover
    const cliente: Cliente = {
      nome: form.value.nome,
      fone: form.value.fone,
      email: form.value.email,
    };
    this.clienteService.adicionarCliente(
      form.value.nome,
      form.value.fone,
      form.value.email
    );
    //remover
    this.clienteAdicionado.emit(cliente);
  }}
```

- Podemos ajustar o arquivo **app.component.html** removendo o **event binding** e o **property binding** que ele faz com os componentes que utiliza, já que não estamos mais passando clientes entre componentes. Veja a Listagem 2.2.8.

Listagem 2.2.8

```
<app-cabecalho></app-cabecalho>
<main>
  <app-cliente-inserir></app-cliente-inserir>
  <app-cliente-lista></app-cliente-lista>
</main>
```

- Também podemos remover a anotação @Input no componente ClienteListaComponent, no arquivo **cliente-lista.component.ts**. Veja a Listagem 2.2.9.

Listagem 2.2.9

```
import { Component, OnInit } from '@angular/core';
import { Cliente } from '../cliente.model';
import { ClienteService } from '../clientes.service';

@Component({
  selector: 'app-cliente-lista',
  templateUrl: './cliente-lista.component.html',
  styleUrls: ['./cliente-lista.component.css'],
})
export class ClienteListaComponent implements OnInit {
  clientes: Cliente[] = [];

  constructor(public clienteService: ClienteService) {}

  ngOnInit(): void {
    this.clientes = this.clienteService.getClientes();
  }
}
```

- No futuro, a lista de clientes será obtida a partir de um servidor remoto. Idealmente, as operações que a envolvem devem ser realizadas de maneira assíncrona. Para isso, vamos aplicar a API de **Observables** do pacote rxjs. No arquivo **clientes.service.ts**, começamos instanciando um objeto “observável”, capaz de gerar eventos, que no Angular recebe o nome **Subject**. Veja a Listagem 2.2.10.

Listagem 2.2.10

```
import { Cliente } from './cliente.model';
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class ClienteService {
  private clientes: Cliente[] = [];
  private listaClientesAtualizada = new Subject<Cliente[]>();

  getClientes(): Cliente[] {
    return [...this.clientes];
  }

  adicionarCliente(nome: string, fone: string, email: string) {
    const cliente: Cliente = {
      nome: nome,
      fone: fone,
      email: email,
    };
    this.clientes.push(cliente);
  }
}
```

- A seguir, no método **adicionarCliente** do **serviço**, utilizamos seu método **next** cujo funcionamento é análogo ao **emit** de **EventEmitter**. Ele simboliza que um evento aconteceu. Assim, objetos observadores (Observable do pacote rxjs) podem reagir quando esse evento acontecer. Veja a Listagem 2.2.11.

Listagem 2.2.11

```
adicionarCliente(nome: string, fone: string, email: string) {
   const cliente: Cliente = {
     nome: nome,
     fone: fone,
     email: email,
   };
   this.clientes.push(cliente);
   this.listaClientesAtualizada.next([...this.clientes]);
 }
```

- Para permitir que componentes registrem observadores vinculados à lista atualizada do serviço, vamos definir um novo método que devolve um Observable, ainda no serviço. Veja a Listagem 2.2.12.

Listagem 2.2.12

```
getListaDeClientesAtualizadaObservable() {
   return this.listaClientesAtualizada.asObservable();
 }
```

- Os componentes interessados em obter uma cópia da lista toda vez que ela for atualizada utilizarão este método para obter o objeto Observable e, a seguir, chamar o método **subscribe** sobre ele. Ele permite que especifiquemos duas funções. A primeira executa quando a notificação ocorrer com sucesso. A segunda somente executa caso ocorra algum erro. No momento utilizaremos somente a primeira. Vamos atualizar o componente no arquivo **cliente-lista.component.ts** para que ele faça uso dessa estrutura. Veja a Listagem 2.2.13.

Listagem 2.2.13

```
ngOnInit(): void {
   this.clientes = this.clienteService.getClientes();
   this.clienteService
     .getListaDeClientesAtualizadaObservable()
     .subscribe((clientes: Cliente[]) => {
       this.clientes = clientes;
     });
  }
```

Note que, neste momento, seu aplicativo deve estar funcionando corretamente.

- Quando o componente for destruído desejamos remover o registro do Observable, caso contrário ele permanecerá existindo na memória, caracterizando um vazamento de memória. Para isso, começamos armazenando o retorno do método subscribe em uma variável do tipo **Subscription**, como na Listagem 2.2.14.

Listagem 2.2.14

```
import { Component, OnInit } from '@angular/core';
import { Cliente } from '../cliente.model';
import { ClienteService } from '../clientes.service';
import { Subscription, Observable } from 'rxjs';

@Component({
  selector: 'app-cliente-lista',
  templateUrl: './cliente-lista.component.html',
  styleUrls: ['./cliente-lista.component.css'],
})
export class ClienteListaComponent implements OnInit {
  clientes: Cliente[] = [];
  private clientesSubscription: Subscription;

  constructor(public clienteService: ClienteService) {}

  ngOnInit(): void {
    this.clientes = this.clienteService.getClientes();
    this.clientesSubscription = this.clienteService
      .getListaDeClientesAtualizadaObservable()
      .subscribe((clientes: Cliente[]) => {
        this.clientes = clientes;
      });
  }
}
```

- Com o objeto Subscription em mãos, podemos remover o registro quando o componente for destruído. Para isso, vamos fazer com que a classe ClienteListaComponent implemente a interface **OnDestroy**. Ela define um método chamado **ngOnDestroy** que entra em execução automaticamente quando o componente é destruído. Veja a Listagem 2.2.15.

Listagem 2.2.15

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Cliente } from '../cliente.model';
import { ClienteService } from '../clientes.service';
import { Subscription, Observable } from 'rxjs';

@Component({
  selector: 'app-cliente-lista',
  templateUrl: './cliente-lista.component.html',
  styleUrls: ['./cliente-lista.component.css'],
})
export class ClienteListaComponent implements OnInit, OnDestroy {
  clientes: Cliente[] = [];
  private clientesSubscription: Subscription;

  constructor(public clienteService: ClienteService) {}

  ngOnInit(): void {
    this.clientes = this.clienteService.getClientes();
    this.clientesSubscription = this.clienteService
      .getListaDeClientesAtualizadaObservable()
      .subscribe((clientes: Cliente[]) => {
        this.clientes = clientes;
      });
  }

  ngOnDestroy(): void {
    this.clientesSubscription.unsubscribe();
  }
}
```

2.3 (Ajustes no form) Vamos fazer alguns pequenos ajustes no form da aplicação.

- Note que, após a inserção de um cliente, os campos permanecem com os valores digitados pelo usuário. Para limpá-los, podemos chamar o método **resetForm** de **ngForm** no método **onAdicionarCliente** do componente definido no arquivo **cliente-inserir.component.ts**. Veja a Listagem 2.3.1.

Listagem 2.3.1

```
onAdicionarCliente(form: NgForm) {
   if (form.invalid) {
     return;
   }

   this.clienteService.adicionarCliente(
     form.value.nome,
     form.value.fone,
     form.value.email
   );
   form.resetForm();
 }
```

- Vamos adicionar botões que irão permitir a realização de outras operações CRUD sobre cada cliente da lista. O Angular Material possui um componente chamado **mat-action-row** apropriado para isso. Ele será utilizado no arquivo **cliente-lista.component.html**, como mostra a Listagem 2.3.2.

Listagem 2.3.2

```
<mat-accordion *ngIf="clientes.length > 0">
  <mat-expansion-panel *ngFor="let cliente of clientes">
    <mat-expansion-panel-header>Nome: {{ cliente.nome }}</mat-expansion-panel-header>
    <p>Fone: {{ cliente.fone }}</p>
    <hr />
    <p>Email: {{ cliente.email }}</p>
    <mat-action-row>
      <button mat-button color="primary">EDITAR</button>
      <button mat-button color="warn">REMOVER</button>
    </mat-action-row>
  </mat-expansion-panel>
</mat-accordion>
<p class="mat-body-1" style="text-align: center;" *ngIf="clientes.length <= 0">
  Nenhum cliente cadastrado
</p>
```

Referências

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

Angular Material UI component library. 2020. Disponível em <https://material.angular.io>. Acesso em agosto de 2020

Express - Node.js web application framework. 2020. Disponível em <https://expressjs.com>. Acesso em agosto de 2020.

Node.js. 2020. Disponível em <https://nodejs.org/en/>. Acesso em agosto de 2020.

The most popular database for modern apps | MongoDB. 2020. Disponível em <https://www.mongodb.com>. Acesso em agosto de 2020.