

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 gráficas 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 (Executando os servidores) Lembre-se de colocar os dois servidores em execução com

ng serve --open (para acesso à aplicação Angular)

npm run start:server (Back End em NodeJS)

- Execute cada um deles em um terminal separado e mantenha ambos em execução.

2.2 (Lidando com o identificador `_id`) Como vimos, o método `find` permite realizar buscas na base de dados. É interessante olhar para a estrutura dos objetos inseridos na base e descobrir a existência de um campo chamado `_id`, gerado automaticamente pelo MongoDB. Para isso, ajuste seu endpoint **get** (arquivo **app.js**) para exibir a coleção de objetos existentes. Veja a Listagem 2.2.1.

Listagem 2.2.1

```
app.get('/api/clientes', (req, res, next) => {  
  Cliente.find().then(documents => {  
    console.log (documents)  
    res.status(200).json({  
      mensagem: "Tudo OK",  
      clientes: documents  
    });  
  })  
});
```

- Atualize a página e verifique o console do terminal em que o Back End está em execução. Você deverá ver algo parecido com o que exibe a Figura 2.2.1.

Figura 2.2.1

```
{  
  fone: '12223344',  
  _id: 5f5beab198c666a673f1d3de,  
  nome: 'Jose',  
  email: 'jose@email.com',  
  __v: 0  
},
```

Nota: Para saber mais sobre o campo `__v` também existente no documento, visite a documentação oficial, disponível no Link 2.2.1.

Link 2.2.1

<https://mongoosejs.com/docs/guide.html#versionKey>

- Ocorre que as demais operações do CRUD (remoção e atualização) dependem do identificador do documento sobre o qual irão executar. Por essa razão, iremos adicionar o campo **id** à interface que define o que é um cliente, no arquivo **cliente.model.ts**. Veja a Listagem 2.2.2.

Listagem 2.2.2

```
export interface Cliente {  
  id: string;  
  nome: string;  
  fone: string;  
  email: string;  
}
```

- Note, contudo, que optamos por utilizar o nome **id** ao invés de **_id**, já que o primeiro é mais natural. Por essa razão, precisaremos realizar um mapeamento no momento em que os dados são obtidos pela aplicação Angular, explicando que o campo **_id** de cada documento deve ser mapeado ao seu campo **id**.

- Isso pode ser feito utilizando os mecanismos da API de Observables. Uma vez que tenhamos um resultado obtido por meio de um Observable, ele pode passar por uma lista de chamadas de funções (chamadas operadores) que podem realizar transformações arbitrárias. O método que permite esse encadeamento se chama **pipe**. Veja como ele pode ser aplicado na Listagem 2.2.3. Estamos no arquivo **clientes.service.ts**, já que é o serviço o responsável pela recuperação de dados de clientes.

Listagem 2.2.3

```
getClientes(): void {  
  this.httpClient.get <{mensagem: string, clientes:  
    Cliente[]}>('http://localhost:3000/api/clientes')  
    .pipe()  
    .subscribe(  
      (dados) => {  
        this.clientes = dados.clientes;  
        this.listaClientesAtualizada.next([...this.clientes]);  
      }  
    )  
  }  
}
```

- Neste momento, a chamada ao método pipe ainda não está fazendo nada. Precisamos passar para ele uma função (que leva o nome de operador) que será executada uma vez que os dados tenham sido recebidos do servidor. Essa função se chama **map**. Ela recebe uma função que será responsável por fazer o mapeamento desejado. Veja a Listagem 2.2.4.

Listagem 2.2.4

```
import { map } from 'rxjs/operators';  
getClientes(): void {  
  this.httpClient.get <{mensagem: string, clientes:  
    Cliente[]}>('http://localhost:3000/api/clientes')  
    .pipe(map((dados) => {  
      })))  
    .subscribe(  
      (dados) => {  
        this.clientes = dados.clientes;  
        this.listaClientesAtualizada.next([...this.clientes]);  
      }  
    )  
  }  
}
```

- O tipo devolvido pelo servidor (que possui um campo `_id` e que não possui um campo `id`) é incondizente com o tipo que a aplicação Angular define. Por isso, vamos alterar o tipo especificado no método `get`, como na Listagem 2.2.5.

Listagem 2.2.5

```
getClientes(): void {  
  this.httpClient.get <{mensagem: string, clientes: any}>('http://localhost:3000/api/clientes')  
    .pipe(map((dados) => {  
  
    }))  
  .subscribe(  
    (dados) => {  
      this.clientes = dados.clientes;  
      this.listaClientesAtualizada.next([...this.clientes]);  
    }  
  )  
}
```

- Os dados recebidos pela função passada como parâmetro para o operador `map` possuem uma coleção chamada `clientes`. Desejamos executar os itens desta coleção um a um, explicando que cada um deles deve ter seu campo `_id` mapeado para um novo campo, chamado `id`. Isso pode ser feito com a função `map`, própria de listas Javascript. Veja a Listagem 2.2.6.

Listagem 2.2.6

```
getClientes(): void {  
  this.httpClient.get <{mensagem: string, clientes: any}>('http://localhost:3000/api/clientes')  
    .pipe(map((dados) => {  
      return dados.clientes.map(cliente => {  
        return {  
          id: cliente._id,  
          nome: cliente.nome,  
          fone: cliente.fone,  
          email: cliente.email  
        }  
      })  
    }))  
  .subscribe(  
    (dados) => {  
      this.clientes = dados.clientes;  
      this.listaClientesAtualizada.next([...this.clientes]);  
    }  
  )  
}
```

- Aquilo que o método pipe devolve é entregue como parâmetro para a função subscribe. Depois da transformação realizada, o objeto devolvido é a lista de clientes e não mais um objeto com a mensagem de status e a coleção, como antigamente. Por isso, ajuste a função de subscribe para usar a coleção diretamente, como na Listagem 2.2.7.

Listagem 2.2.7

```
getClientes(): void {
  this.httpClient.get <{mensagem: string, clientes: any}>('http://localhost:3000/api/clientes')
    .pipe(map((dados) => {
      return dados.clientes.map(cliente => {
        return {
          id: cliente._id,
          nome: cliente.nome,
          fone: cliente.fone,
          email: cliente.email
        }
      })
    }))
    .subscribe(
      (clientes) => {
        this.clientes = clientes;
        this.listaClientesAtualizada.next([...this.clientes]);
      }
    )
}
```

2.3 (Removendo clientes) Agora que os clientes possuem id, podemos implementar as demais operações de um CRUD básico, como a remoção.

- O primeiro passo é estabelecer um novo endpoint no Back End. Ele deve receber requisições HTTP do tipo DELETE. Em sua URL, iremos incluir o id do cliente a ser removido. Note que o id não é uma parte fixa da URL, ele será adicionado em tempo de requisição. Por isso a notação para escrevê-lo é diferente: iremos preceder um nome escolhido arbitrariamente com o símbolo : (dois pontos). Abra o arquivo **app.js** e faça a sua definição como mostra a Listagem 2.3.1.

Listagem 2.3.1

```
app.delete ('/api/clientes/:id', (req, res, next) => {  
  console.log (req.params);  
  res.status(200).end();  
});
```

- A Listagem 2.3.1 mostra como podemos acessar os parâmetros incluídos na URL de uma requisição usando o objeto **params**. Abra o Postman e faça uma requisição HTTP DELETE no endpoint localhost:3000/api/clientes/1 e veja o resultado no terminal em que o Back End está em execução. A Figura 2.3.1 mostra a requisição sendo feita no Postman e a Figura 2.3.2 mostra a saída esperada no terminal. Veja que o objeto **params** é simplesmente um objeto JSON que nos dá acesso a tudo o que foi passado como parâmetro (daí seu nome) na URL do endpoint.

Figura 2.3.1

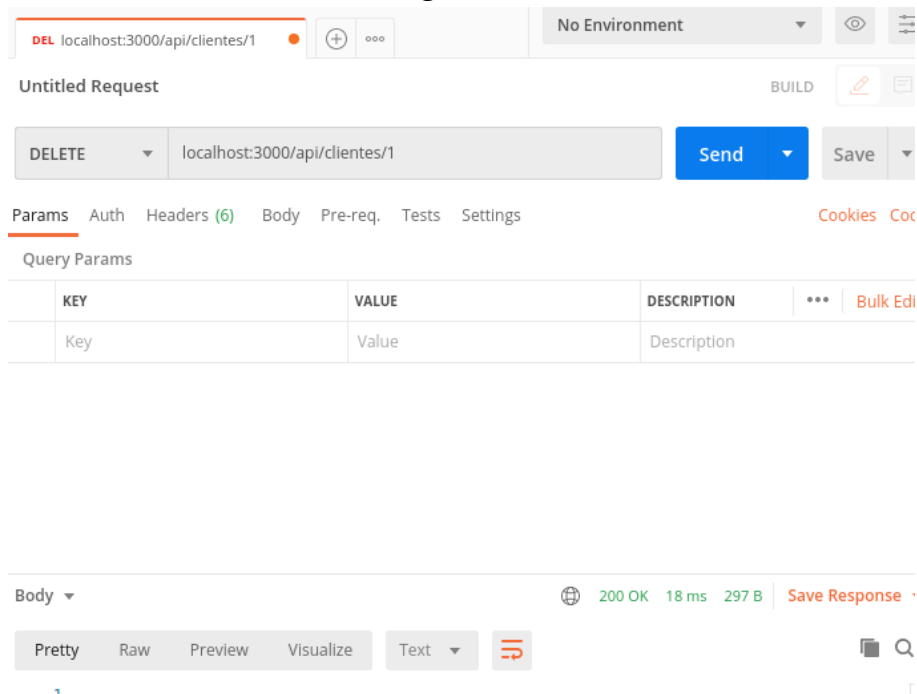


Figura 2.3.2

```
{ id: '1' }
```

- Para fazer a remoção na aplicação Angular, começamos editando o arquivo **cliente-lista.component.html** já que o botão para remoção é definido ali. Basta fazer um **event binding** especificando uma função (que será definida) que recebe o id do cliente a ser removido. Veja 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" (click)="onDelete(cliente.id)">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>
```

- A função **onDelete** será definida no arquivo **cliente-lista.component.ts**. Ela recebe o id e interage com o serviço de manipulação de clientes, solicitando que ele faça a requisição HTTP. A Listagem 2.3.3 mostra a função **onDelete** de **cliente-lista.component.ts**. A Listagem 2.3.4 mostra a função definida em **clientes.service.ts** que é responsável pelo envio da requisição.

Listagem 2.3.3

```
onDelete (id: string): void{
  this.clienteService.removerCliente(id);
}
```

Listagem 2.3.4

```
removerCliente (id: string): void{
  this.httpClient.delete(`http://localhost:3000/api/clientes/${id}`).subscribe(() => {
    console.log (`Cliente de id: ${id} removido`);
  });
}
```

- Faça um teste enviando uma requisição DELETE clicando no botão remover de qualquer cliente na aplicação Angular. O Console do navegador (CTRL+SHIFT+I no Chrome) deve exibir o log com o id do cliente supostamente removido. O console do Back End, por sua vez, exibe o objeto **params** da requisição, que também inclui o id. No entanto, a remoção não foi efetivada já que o MongoDB não foi informado sobre isso.

- Para, de fato, remover o cliente cujo botão remover associado foi clicado, vamos usar o método **deleteOne** de nosso modelo (Cliente, no Back End). Ele espera o id do documento a ser removido. Embora a aplicação Angular tenha se preocupado em mapear o nome `_id` para `id`, ainda há documentos no MongoDB utilizando o nome `_id` e, por essa razão, ainda o utilizamos.

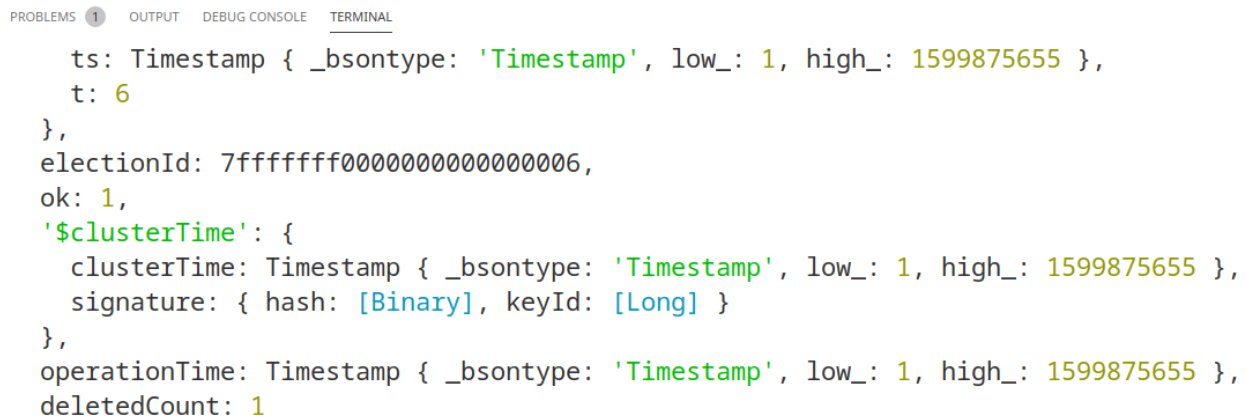
Veja a Listagem 2.3.5. Estamos no arquivo **app.js** neste momento.

Listagem 2.3.5

```
app.delete('/api/clientes/:id', (req, res, next) => {
  Cliente.deleteOne ({_id: req.params.id}).then((resultado) => {
    console.log (resultado);
    res.status(200).json({mensagem: "Cliente removido"})
  });
});
```

- Na aplicação Angular, clique no botão Remover de algum cliente e inspecione a saída no terminal em que o Back End está em execução. Você deve ver algo parecido com o que a Figura 2.3.3 exibe.

Figura 2.3.3



The image shows a terminal window with a dark background. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active. The terminal displays a JSON response from a MongoDB deletion operation. The response includes a timestamp, a cluster time, an election ID, an 'ok' status of 1, and a deleted count of 1. The JSON is color-coded: 'Timestamp' is green, '1' is yellow, and '1599875655' is green. The 'Binary' and 'Long' types are shown in blue brackets.

```
ts: Timestamp { _bsontype: 'Timestamp', low_: 1, high_: 1599875655 },
t: 6
},
electionId: 7fffffff000000000000000006,
ok: 1,
'$clusterTime': {
  clusterTime: Timestamp { _bsontype: 'Timestamp', low_: 1, high_: 1599875655 },
  signature: { hash: [Binary], keyId: [Long] }
},
operationTime: Timestamp { _bsontype: 'Timestamp', low_: 1, high_: 1599875655 },
deletedCount: 1
```

- A coleção exibida pela aplicação Angular não é atualizada após uma remoção. Para que a nova coleção possa ser vista, precisamos clicar em atualizar. Para que essa atualização ocorra automaticamente, iremos atualizar a coleção do serviço de manipulação de clientes (arquivo **clientes.service.ts**) removendo dela o cliente que já foi removido da base. Depois disso, enviamos uma notificação aos componentes interessados em alterações feitas na lista. Veja a Listagem 2.3.4.

Listagem 2.3.4

```
removerCliente (id: string): void {  
  this.httpClient.delete(`http://localhost:3000/api/clientes/${id}`).subscribe() => {  
    this.clientes = this.clientes.filter((cli) => {  
      return cli.id !== id  
    });  
    this.listaClientesAtualizada.next([...this.clientes]);  
  });  
}
```

- Faça o teste inserindo um novo cliente na aplicação Angular e, logo em seguida, fazendo a sua remoção. Note que ainda não estamos inserindo um cliente com id. Isso quer dizer que, após fazer a inserção, precisamos atualizar a página para que ele tenha o id gerado pelo MongoDB. Se isso não for feito, a aplicação gerará um erro parecido com o que exibe a Figura 2.3.4.

Figura 2.3.4

(node:67131) UnhandledPromiseRejectionWarning: CastError: Cast to ObjectId failed for value "undefined" at path "_id" for model "Cliente"

- Para ajustar o problema ilustrado, basta que façamos uso do id gerado pelo MongoDB uma vez que um cliente seja inserido. O primeiro passo para isso é registrar uma função que será executada assim que a inserção ocorrer. Ela recebe como parâmetro o documento criado, incluindo o seu id. A seguir, fazemos com que a resposta seja dada ao cliente (o app Angular) somente quando essa função for executada pois, assim, podemos devolver uma resposta que inclui o id de interesse. Veja a Listagem 2.3.5. Estamos no arquivo **app.js**, ou seja, no Back End.

Listagem 2.3.5

```
app.post ('/api/clientes', (req, res, next) => {  
  const cliente = new Cliente({  
    nome: req.body.nome,  
    fone: req.body.fone,  
    email: req.body.email  
  
  })  
  cliente.save().  
  then (clienteInserido => {  
    res.status(201).json({  
      mensagem: 'Cliente inserido',  
      id: clienteInserido._id  
    })  
  })  
});
```

- A seguir, a aplicação Angular precisa fazer uso do id recebido do Back End. Cabe ao serviço de manipulação de clientes (**clientes.service.ts**) fazer isso. Precisamos ajustar o tipo esperado pelo post (entre < e >) e associar o id ao objeto cliente. Aproveitamos também para construir o cliente, a princípio, com o valor null associado ao campo id, já que especificamos que ele é obrigatório. Veja a Listagem 2.3.6.

Listagem 2.3.6

```
adicionarCliente(nome: string, fone: string, email: string) {  
  const cliente: Cliente = {  
    id: null,  
    nome: nome,  
    fone: fone,  
    email: email,  
  };  
  this.httpClient.post<{mensagem: string, id: string}>('http://localhost:3000/api/clientes',  
cliente).subscribe(  
  (dados) => {  
    cliente.id = dados.id;  
    this.clientes.push(cliente);  
    this.listaClientesAtualizada.next([...this.clientes]);  
  }  
)  
}
```

2.4 (Adicionando funcionalidades de roteamento do Angular) Como o nome sugere, uma SPA (Single Page Application) possui uma única página. Conforme o usuário navega por ela, o navegador precisa renderizar somente trechos apropriados para a seção que ele deseja visualizar. Muitas partes podem ser idênticas em páginas diferentes. O Angular possui um mecanismo de rotas que permite que especifiquemos componentes a serem renderizados mediante requisições feitas de acordo com padrões pré estabelecidos. Por exemplo, podemos especificar que o componente **ListaComponent** deve ser renderizado quando o padrão **/lista** for acessado. O componente **PessoaComponent** deve ser renderizado quando o padrão **/pessoa** for acessado. E assim por diante. Nesta seção veremos como adicionar esse mecanismo a nossa aplicação.

- Para começar, vamos especificar um novo módulo que terá como finalidade lidar com as rotas da aplicação. Crie um novo arquivo chamado **app-routing.module.ts** (o nome pode ser qualquer um, na verdade, é só uma convenção) lado a lado com o arquivo **app.module.ts**.

- Um módulo é definido como uma classe a que aplicamos a anotação **NgModule**. O conteúdo inicial do arquivo **app-routing.module.ts** é dado na Listagem 2.4.1.

Listagem 2.4.1

```
import { NgModule } from '@angular/core';
@NgModule({
})
export class AppRoutingModule{
}
```

- Nosso módulo fará uso de um módulo próprio do Angular chamado **RouterModule**. As funcionalidades necessárias são todas implementadas por ele. Por essa razão, traga o símbolo **RouterModule** para o contexto. Ele pode ser importado de **@angular/router**. Veja a Listagem 2.4.2.

Listagem 2.4.2

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
@NgModule({
})
export class AppRoutingModule{
}
```

- Em geral, um módulo como esse armazena uma **lista de rotas**. Trata-se de uma lista cujos objetos mapeiam padrões (/clientes, por exemplo) a nomes de componentes (ClientesComponent, por exemplo). Cada objeto é do tipo **Route**. Faça a sua definição como mostra a Listagem 2.4.3.

Listagem 2.4.3

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
const routes: Routes = [
];
@NgModule({
})
export class AppRoutingModule{
}
```

- A primeira rota que definiremos mapeia a raiz da aplicação (associada a **path**) ao componente que exibe a lista de clientes (associado a **component**) . Veja a Listagem 2.4.4.

Listagem 2.4.4

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';
const routes: Routes = [
  { path: "", component: ClienteListaComponent }
];
@NgModule({
})
export class AppRoutingModuleModule{
}
```

- Uma nova rota poderia dar acesso a um componente que permite a inserção de clientes. Veja o exemplo da Listagem 2.4.5.

Listagem 2.4.5

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
const routes: Routes = [
  { path: "", component: ClienteListaComponent },
  { path: 'criar', component: ClienteInserirComponent }
];
@NgModule({
})
export class AppRoutingModuleModule{
}
```

- Note que ainda não fizemos uso do módulo de roteamento próprio do Angular. Para fazê-lo, basta adicioná-lo à coleção **imports** que pode ser definida nos metadados de nosso módulo. Em particular, chamamos o método **forRoot** para especificar o objeto de rotas a ser usado. Veja a Listagem 2.4.6.

Listagem 2.4.6

```
import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
const routes: Routes = [
  { path: '', component: ClienteListaComponent },
  { path: 'criar', component: ClienteInserirComponent }
];
@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ]
})
export class AppRoutingModule{

}
```

- Precisamos acessar o módulo de rotas (configurado com as rotas que especificamos) a partir do módulo principal de nossa aplicação, definido no arquivo **app.module.ts**. Para isso, o primeiro passo é especificá-lo na coleção **exports** nos metadados do módulo de roteamento. Veja a Listagem 2.4.7.

Listagem 2.4.7

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { ClienteListaComponent } from './clientes/cliente-lista/cliente-lista.component';
import { ClienteInserirComponent } from './clientes/cliente-inserir/cliente-inserir.component';
const routes: Routes = [
  { path: '', component: ClienteListaComponent },
  {path: 'criar', component: ClienteInserirComponent}
];
@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule{

}
```

- A seguir, no arquivo **app.module.ts**, onde o módulo principal da aplicação está definido, importamos nosso módulo de rotas, como ilustra a Listagem 2.4.8.

Listagem 2.4.8

```
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [
    AppComponent,
    ClienteInserirComponent,
    CabecalhoComponent,
    ClienteListaComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,...
  ]
})
export class AppModule { }
```

- Para utilizar o mecanismo de roteamento do Angular, especificamos um ponto em que os componentes devem ser renderizados. Esse ponto chama-se **router-outlet** (outlet pode ser traduzido como “saída”, “entrega”, “outorga”). Isso será feito no template do componente principal da aplicação, definido no arquivo **app.component.html**. Veja a Listagem 2.4.9.

Listagem 2.4.9

```
<app-cabecalho></app-cabecalho>
<main>
  <router-outlet></router-outlet>
</main>
```

- Acesse as URLs **localhost:4200** e **localhost:4200/criar**. Note que as páginas compartilham, por exemplo, a barra superior.

- Podemos adicionar links à aplicação que dão acesso a cada seção de interesse. Para isso, abra o arquivo **cabecalho.component.html**. Ajuste o seu conteúdo como na Listagem 2.4.10.

Listagem 2.4.10

```
<mat-toolbar color="primary">
  <span><a routerLink="/">Clientes</a></span>
  <ul>
    <li><a routerLink="/criar">Novo Cliente</a></li>
  </ul>
</mat-toolbar>
```

Em breve trataremos dos estilos.

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.