AngularJS - Conceito

O AngularJS, é um framework JavaScript de código aberto desenvolvido pela Google, que revolucionou a forma como as aplicações web são desenvolvidas.

Ele oferece uma estrutura robusta para criar interfaces de usuário dinâmicas e interativas, simplificando o desenvolvimento front-end.

AngularJS - Conceito

Em essência, o AngularJS é um framework que estende as capacidades do HTML, permitindo que você crie componentes personalizados e consiga vincular dados de forma bidirecional.

Isso significa que quando os dados mudam no modelo, a visualização (HTML) é atualizada automaticamente e vice-versa.

AngularJS - Vantagens

Produtividade: A estrutura modular e os recursos prontos para uso do AngularJS aceleram o desenvolvimento.

Manutenibilidade: O código organizado em componentes torna a aplicação mais fácil de entender e manter.

Teste: O AngularJS facilita a criação de testes unitários, garantindo a qualidade do software.

Comunidade: Uma grande comunidade de desenvolvedores oferece suporte, recursos e bibliotecas adicionais.

AngularJS - Estrutura

Componentes: Os blocos de construção do AngularJS. Cada componente tem seu próprio template (HTML), controller (lógica) e pode ter seus próprios serviços.

Diretivas: Extendem as tags HTML, permitindo criar elementos personalizados e a manipulação do DOM.

Expressões: Avaliam o JavaScript dentro do HTML, permitindo exibir dados dinamicamente.

AngularJS - Estrutura

Filtros: Formatam dados antes de exibi-los, como ordenar, filtrar e transformar strings.

Serviços: Contêm a lógica de negócios da aplicação, como fazer requisições HTTP ou acessar dados de um banco de dados.

Injeção de Dependência: Permite que os componentes recebam as dependências de que precisam, promovendo um código mais desacoplado e testável.

AngularJS - Arquitetura

O AngularJS segue o padrão Model-View-Controller (MVC):

Model: Representa os dados da aplicação.

View: É a interface do usuário, construída com HTML e diretivas.

Controller: Controla a lógica da aplicação e atualiza o modelo e a view.

AngularJS - Exemplo

Ex:

```
<!DOCTYPE html>
<html ng-app="minhaPrimeiraAplicacaoAngular">
<head>
                                                    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></
script>
</head>
<body>
  <div ng-controller="MeuControlador">
    <p>Nome: {{ nome }}</p>
    <button ng-click="mudarNome()">Mudar Nome</button>
  </div>
```

AngularJS - Exemplo

Ex:

```
<script>
    angular.module('minhaPrimeiraAplicacaoAngular', [])
      .controller('MeuControlador', function($scope) {
        $scope.nome = 'João';
        $scope.mudarNome = function() {
          $scope.nome = 'Maria';
        };
      });
  </script>
 </body>
</html>
```

AngularJS - Exemplo

No exemplo praticado, nós temos os seguintes componentes Angular:

ng-app define o módulo da aplicação.

ng-controller associa um controller à view.

{{ **nome** }} é uma expressão que exibe o valor da propriedade nome do controller.

ng-click é uma diretiva que executa uma função quando o botão é clicado.

Uso do NodeJS e do npm

Apesar do AngularJS ser um framework JavaScript que roda diretamente no navegador, a combinação com **Node.js** e **npm** traz uma série de benefícios para o desenvolvimento de aplicações web.

O Node.js, junto com o npm (Node Package Manager), fornece uma forma eficiente de gerenciar as dependências do seu projeto.

Uso do NodeJS e do npm

Através do npm, você pode instalar facilmente bibliotecas de terceiros, como frameworks de testes, ferramentas de build e outros módulos que complementam o AngularJS.

O Node.js permite criar um servidor local para desenvolver e testar sua aplicação. Isso facilita a simulação de um ambiente de produção e agiliza o fluxo de trabalho.

Uso do NodeJS e do npm

O npm mantém um registro preciso de todas as bibliotecas utilizadas em seu projeto, facilitando a instalação, atualização e remoção de pacotes.

O npm possui um vasto repositório de pacotes, o que significa que você provavelmente encontrará uma biblioteca para quase qualquer necessidade.

O npm permite definir scripts personalizados para automatizar tarefas como iniciar o servidor de desenvolvimento, executar testes e criar builds de produção.

Vantagens em usar o NodeJS e o npm

Produtividade: Automação de tarefas, gerenciamento eficiente de dependências e um ecossistema rico de bibliotecas.

Consistência: Um ambiente de desenvolvimento unificado para frontend e backend (se necessário).

Escalabilidade: Facilidade para desenvolver aplicações complexas e de grande porte.

Facilidade de manutenção: Código organizado e bem estruturado, facilitando a manutenção e colaboração em equipe.

Vantagens em usar o NodeJS e o npm

Imagine que você está desenvolvendo uma aplicação AngularJS que precisa fazer requisições a uma API.

Com o npm, você pode instalar facilmente bibliotecas como o Axios ou o Angular HTTPClient para facilitar essas requisições.

Além disso, você pode configurar um script npm para iniciar um servidor local com o Node.js, permitindo que você teste sua aplicação em tempo real.

Vantagens em usar o NodeJS e o npm

Embora o AngularJS seja um framework frontend, a combinação com Node.js e npm oferece um ecossistema completo para o desenvolvimento de aplicações web modernas.

Ao entender os benefícios dessa combinação, você estará mais bem preparado para desenvolver aplicações robustas e escaláveis.

Angular CLI - Definição

O Angular CLI é uma ferramenta de linha de comando (CLI) que simplifica significativamente o desenvolvimento de aplicações Angular.

Ele oferece uma série de comandos e recursos que automatizam tarefas repetitivas, agilizando o processo de criação, desenvolvimento e manutenção de projetos Angular.

Angular CLI – Vantagens de uso

Configuração Inicial Rápida: Cria um novo projeto Angular com toda a estrutura necessária em poucos comandos.

Geração de Componentes, Serviços e Outros: Permite criar rapidamente novos componentes, serviços, rotas e outros elementos da aplicação, economizando tempo e evitando erros manuais.

Build e Servidor de Desenvolvimento: Cria builds de produção otimizados e inicia um servidor de desenvolvimento local com recarga automática.

Angular CLI – Vantagens de uso

Testes: Integra-se com ferramentas de teste como o Karma e o Jasmine, facilitando a criação e execução de testes unitários e end-to-end.

Integração com outras ferramentas: Funciona bem com outras ferramentas populares do ecossistema Angular, como o Angular Material e o NgRx.

Padronização: Promove um estilo de codificação consistente e padronizado, facilitando a colaboração em equipe.

Angular CLI – Como instalar

Para fazer a instalação do Angular CLI, como ele se trata de uma biblioteca, precisaremos fazer uso do gerenciador de pacotes npm, através do comando abaixo:

npm install -g @angular/cli

Angular CLI – Como criar um projeto Angular

Para criarmos um projeto Angular com toda sua estrutura básica, utilizando o Angular CLI, basta executarmos o comando abaixo:

ng new meu-projeto

Ao executar o comando anterior, será criada uma pasta com o nome do projeto informado contendo todos os arquivos necessários para que o projeto criado funcione como uma aplicação web.

AngularJS – Desenvolvendo uma aplicação

Depois de criado o projeto ele já pode ser testado. Para tanto, basta executar o comando **ng serve** que será responsável por iniciar o servidor com toda estrutura do projeto funcionando.

Uma vez iniciado, basta abrir um browser de sua preferência e digitar a seguinte URL:

http://localhost:4200/

AngularJS – Desenvolvendo uma aplicação

Será exibida uma Página padrão do Angular. Para alterar essa página padrão, será feita uma substituição do código existente no arquivo **src/app/app.component.html** (podem apagar tudo que está nesse arquivo) pelo código abaixo:

Minha Agenda

<router-outlet></router-outlet>

Depois de efetuar a substituição do código, conforme mencionado, basta salvar o arquivo e constatar as mudanças no browser.

AngularJS – Desenvolvendo uma aplicação

A partir de agora, nós aprenderemos os conceitos do Angular na prática, através do desenvolvimento de uma aplicação responsável por gerenciar uma agenda.

Nessa aplicação, serão possíveis as seguintes funcionalidades:

Adicionar um contato na agenda

Editar um contato da agenda

Listar contatos registrados na agenda

Remover um contato da agenda

Autenticar na aplicação

AngularJS – Desenvolvendo um model

Para desenvolvermos nossa aplicação, será necessário criarmos o contigo fonte da entidade responsável por representar um contato, também chamada de model.

Um model ou entity, é uma classe que representa uma entidade do mundo real que precisará ser representada no sistema que está sendo desenvolvido.

Um model possui características (que são representados através de atributos na classe) e comportamentos (que são representados através de métodos na classe).

AngularJS – Desenvolvendo um model

Portanto, o passo seguinte consistirá na criação da classe responsável por representar um contato. Essa classe consiste em um model e, por questão de organização, será criada dentro do diretório **src/app/models**.

Para tanto será preciso executar o comando abaixo, de dentro do diretório **models**:

ng generate class contato --type=model

AngularJS – Desenvolvendo um model

A execução do comando anterior criou dentro do diretório **src/aap/models** dois arquivos.

Um deles é o **contato.model.ts** que corresponde à classe que representa o nosso contato.

O outro arquivo é o arquivo de testes desse contato.

AngularJS – Desenvolvendo um model

O conteúdo do arquivo contato.model.ts deverá ser substituído pelo código abaixo:

```
export class Contato {
    id: number;
    name: string;
    email: string;
    phoneNumber: string;
    constructor(id: number, name: string, email: string, phoneNumber: string){
        this.id = id;
        this.name = name;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }
}
```

AngularJS – Desenvolvendo um Service

Agora que nós já criamos nossa entidade contato, o próximo passo consiste na criação de uma classe que será responsável por fazer a ligação do nosso frontend com nossa aplicação backend, através de chamadas das apis desenvolvidas.

Essa classe, responsável por fazer a ligação do nosso frontend com nosso backend, através de requisições HTTP é chamada de Service ou Serviço.

AngularJS – Desenvolvendo um Service

Um serviço no Angular é como um trabalhador especializado que realiza tarefas específicas dentro da sua aplicação.

Ele é responsável por encapsular funcionalidades que são utilizadas em diversos componentes da sua aplicação, promovendo a reutilização de código e a separação de responsabilidades.

AngularJS – Desenvolvendo um Service

Para criarmos nossa classe responsável por fazer requisições HTTP com nosso backend, com objetivo de gerenciarmos as entidades contatos, nós criaremos uma pasta chamada **services** logo abaixo do diretório **app**.

De dentro desse diretório services, nós executaremos o comando abaixo:

ng generate service contato

AngularJS – Desenvolvendo um Service

Ao criarmos o serviço **contato** através do comando anterior, será criado um arquivo chamado **contato.service.ts** contendo a classe **ContatoService** com o seguinte código fonte:

```
import { Injectable } from '@angular/core';
@Injectable({
providedIn: 'root'
})
export class ContatoService {
constructor() { }
}
```

AngularJS – Desenvolvendo um Service

Nós vamos modificar o serviço que nós criamos no frontend chamado ContatoService para que ele seja capaz de invocar nossa api.

Mas antes, precisamos adicionar o provider **provideHttpClient** na lista de **providers** do arquivo **app.config.ts**.

Esse provider faz parte das bibliotecas do Angular e serve para efetuarmos requisições HTTP.

AngularJS – Desenvolvendo um Service

Arquivo **app.config.ts** depois da modificação:

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
        providers: [provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes), provideHttpClient()]
};
```

AngularJS – Desenvolvendo um Service

Agora que fizemos o import do provider **provideHttpClient**, chegou a vez de alterarmos nosso serviço ContatoService.

No nosso caso de uso em questão, nossa aplicação será capaz de adicionar um contato, editar um contato existente, remover um contato existente, listar contatos e obter um único contato a partir do seu identificador único.

Cada uma dessas funcionalidades, corresponde a um serviço disponível na aplicação backend e portanto, desenvolveremos um método para cada uma delas no nosso serviço de contato.

AngularJS – Desenvolvendo um Service

Segue abaixo como ficará o nosso serviço:

```
import { Injectable } from '@angular/core';
import { Contato } from '../models/contato.model';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
@Injectable({
providedIn: 'root'
})
```

AngularJS – Desenvolvendo um Service

Segue abaixo como ficará o nosso serviço:

```
export class ContatoService {

    private contatosUrl = 'http://localhost:5000/api/v1/contact';

    private headers = new HttpHeaders ({"Authorization" : "tokenJWT"})

    constructor(private http: HttpClient) { }
    … DEMAIS MÉTODOS
}
```

AngularJS – Desenvolvendo um Service

```
getContatos(): Observable<Contato[]> {
        return   this.http.get<Contato[]>(   `${this.contatosUrl}`,
{ "headers": this.headers }).pipe(
            map((contatos:any) => {
                return contatos;
            })
        );
}
```

AngularJS – Desenvolvendo um Service

```
getContatoById(id: number): Observable<Contato> {
        return  this.http.get<Contato>(`${this.contatosUrl}/${id}`,
{ "headers": this.headers }).pipe(
            map(contato => {
                return contato;
            })
        );
}
```

AngularJS – Desenvolvendo um Service

```
addContato(contatoParam: Contato) {
  let contatoJson = JSON.stringify(contatoParam);
    this.http.post<any>(`${this.contatosUrl}`, contatoJson, { "headers": this.headers }).subscribe({
        next: data => {
            return data;
        },
        error: error => {
            console.error('Houve um erro:', error);
        }
    });
}
```

AngularJS – Desenvolvendo um Service

```
editContato(contatoParam: Contato){
  let contatoJson = JSON.stringify(contatoParam);
      return    this.http.put<any>(`${this.contatosUrl}/${contatoParam.id}`,
contatoJson, { "headers" :  this.headers }).subscribe({
     next: data => {
        return data;
     },
     error: error => {
        console.error('Houve um erro:', error);
     }
  });
}
```

AngularJS – Desenvolvendo um Service

```
removeContato(id: number){
        this.http.delete<any>(`${this.contatosUrl}/${id}`,     {"headers": this.headers}).subscribe({
          next: data => {
              return data;
          },
          error: error => {
              console.error('Houve um erro:', error);
          }
      });
}
```

AngularJS – Desenvolvendo um Service

Percebam que no atributo headers, foi colocado um token válido. Em um outro momento faremos uma alteração para pegarmos esse token a partir de uma tela de autenticação.

Conforme dito anteriormente, foram desenvolvidos 5 métodos. Cada uma desses métodos faz invocação a uma das apis existentes no backend.

O primeiro método faz uma chamada a api que retorna todos os contatos do banco de dados.

AngularJS – Desenvolvendo um Service

O segundo faz uma chamada a api do backend que retorna um determinado contato a partir do seu id.

O terceiro método é responsável por chamar a api que salva um contato no banco de dados.

O quarto e quinto métodos fazem chamadas as apis que atualizam um contato existente no banco de dados e removem um contato existente no banco de dados respectivamente.

AngularJS – Desenvolvendo um Component

Uma vez que nós já temos esse elo de ligação entre nosso frontend e nosso backend, chegou a vez de criarmos nossos componentes.

Componentes no AngularJS são como peças de Lego que, quando combinadas, formam uma aplicação completa.

Cada componente encapsula uma parte da interface do usuário e a lógica associada a ela. Essa modularização torna o desenvolvimento mais organizado, reutilizável e fácil de manter.

AngularJS – Desenvolvendo um Component

Um componente no AngularJS é definido por:

Template: A parte visual do componente, escrita em HTML. É aqui que você define a estrutura e o layout da interface.

Controller: A parte lógica do componente, escrita em JavaScript. É aqui que você manipula os dados, responde a eventos e interage com outros componentes.

Diretivas: (Opcional) Extensões do HTML que permitem criar elementos personalizados e manipular o DOM de forma mais flexível.

AngularJS – Desenvolvendo um Component

Nesse projeto, nós teremos três telas (tela para inclusão / edição de um contato, tela para listagem de contatos e tela de login), e para cada uma dessas telas, nós desenvolveremos um componente correspondente.

Chegou a hora de criar o primeiro componente do projeto. Para isso, dentro do diretório **src/aap** será criada uma pasta com o nome **components** que será responsável por conter todos os componentes do projeto.

AngularJS – Desenvolvendo um Component

O comando seguinte será responsável por criar o primeiro componente do projeto em questão.

ng generate component <NOME-DO-COMPONENTE>

Ex: **ng generate component contato-list**

AngularJS – Desenvolvendo um Component

Depois de gerado o componente, chegou a hora de fazer a configuração para que este componente criado possa ser utilizado no projeto através de uma chamada no browser por uma URL.

Para tanto, será necessário adicionar uma rota na lista de rotas presente no arquivo **src/app/app.routes.ts**

```
const routes: Routes = [
{path: 'contato/list', component: ContatoListComponent}
];
```

AngularJS – Desenvolvendo um Component

Percebam, que foi criada uma rota chamada **contato/list** que redirecionará o browser para a página HTML do componente setado no atributo **component**.

Para testar o funcionamento desta rota, basta abrir o browser e digitar a URL

http://localhost:4200/contato/list

AngularJS – Desenvolvendo um Component

É importante lembrar que, para visualizar o projeto funcionando no browser, é necessário que o projeto tenha sido iniciado através do comando **ng serve**, conforme dito anteriormente.

Uma vez testado o funcionamento do componente, basta agora desenvolver o código fonte para que ele funcione conforme esperado.

No caso em questão, será desenvolvida uma página responsável por listar contatos de uma agenda, com seus respectivos nomes, endereços de e-mail e telefone.

AngularJS – Desenvolvendo um Component

Para tanto, será aberto o arquivo src/app/components/contato-list/contato-list.component.html e será substituído o código existente pelo seguinte código HTML.

```
<table *ngIf="contatos.length != 0">
  <thead>
    <tr>
      <th>Nome</th>
      <th>Email</th>
      <th>Telefone</th>
      <th>Ações</th>
    </tr>
  </thead>
```

AngularJS – Desenvolvendo um Component

```
    <tbody>
        <tr *ngFor="let contato of contatos;">
            <td>{{ contato.name }}</td>
            <td>{{ contato.email }}</td>
            <td>{{ contato.phoneNumber }}</td>
            <td>
<img class="edit" src="edit.png" (click)="editar(contato.id)">
<img class="remove" src="remove.png" (click)="remover(contato.id)">
            </td>
        </tr>
    </tbody>
</table>
```

AngularJS – Desenvolvendo um Component

Após essa alteração, será possível constatar que aparecerão os seguintes erros:

The `*ngIf` directive was used in the template, but neither the `NgIf` directive nor the `CommonModule` was imported

The `*ngFor` directive was used in the template, but neither the `NgFor` directive nor the `CommonModule` was imported.

Property 'editar' does not exist on type 'ContatoListComponent'.

Property 'remover' does not exist on type 'ContatoListComponent'.

AngularJS – Desenvolvendo um Component

Os dois primeiros erros acontecem porque estamos utilizando diretivas do módulo CommonModule do Angular, porém não importamos esse módulo no nosso componente. Para resolver o problema, basta fazermos o import na classe ContatoListComponent, ficando da seguinte forma:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
@Component({
  selector: 'app-contato-list',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './contato-list.component.html',
  styleUrl: './contato-list.component.css'
})
export class ContatoListComponent {}
```

AngularJS – Desenvolvendo um Component

Ao fazermos o import, os dois primeiros erros serão resolvidos, restando apenas os dois últimos. Porém um terceiro erro irá surgir e ficaremos então com os seguintes erros:

Property 'contatos' does not exist on type 'ContatoListComponent'

Property 'editar' does not exist on type 'ContatoListComponent'.

Property 'remover' does not exist on type 'ContatoListComponent'.

AngularJS – Desenvolvendo um Component

Este novo erro que surgiu acontece porque está sendo utilizada uma lista de contatos que não existe dentro do arquivo src/app/components/contato-list/contato-list.component.ts.

Para resolver esse erro, será então necessário criar essa lista. Essa lista será criada logo abaixo à definição da classe ContatoListComponent da seguinte forma:

```
contatos:Array<Contato> = [];
```

Depois de criado o atributo contatos, o erro deixará de ser exibido.

AngularJS – Desenvolvendo um Component

O segundo e o terceiro erro estão sendo exibidos pelo fato de estarem sendo utilizados métodos no HTML que ainda não foram criados no arquivo **src/app/components/contato-list/contato-list.component.ts**.

Será então necessário criar esses métodos dentro do arquivo, também dentro da classe **ContatoListComponent**.

AngularJS – Desenvolvendo um Component

Segue abaixo o código fonte dos métodos a serem desenvolvidos.

```
editar(id: number){
    this.router.navigateByUrl('/contato/edit', {state: { idContato: id }})
}
remover(id: number){
    this.contatoService.removeContato(id);
    this.atualizaContatos();
}
```

AngularJS – Desenvolvendo um Component

Depois dessa alteração, os erros 2 e 3 deixarão de ser exibidos no código fonte HTML do componente. Porém passarão a ser exibidos dois novos erros, só que desta vez no arquivo onde foram criados os métodos. Os erros exibidos serão os seguintes:

A propriedade 'router' não existe no tipo 'ContatoListComponent'.

A propriedade 'contatoService' não existe no tipo 'ContatoListComponent'.

AngularJS – Desenvolvendo um Component

Esses erros acontecem porque foram inseridos códigos dentro do método editar e no método remover que utilizam os objetos **router** e **contatoService**, porém esses objetos ainda não existem dentro da classe.

Será então necessário criar esses objetos e isso será feito no **construtor** da classe **ContatoListComponent**, que ainda não existe. Para criar esse **contrutor**, já com os objetos **router** e **contatoService**, basta adicionar o código abaixo dentro da classe, de preferência, logo abaixo da lista de contatos criada.

AngularJS – Desenvolvendo um Component

Construtor da classe **ContatoListComponent:**

constructor(private router: Router, private contatoService: ContatoService) {}

Ao inserir esse construtor na classe, serão exibidos os seguintes erros:

Não é possível encontrar o nome 'Router'.

Não é possível encontrar o nome 'ContatoService'.

AngularJS – Desenvolvendo um Component

Esses erros passaram a ser exibidos porque ainda não foi feito o **import** das classes **Router** e **ContatoService** no arquivo **src/app/components/contato-list/contato-list.component.ts**.

Basta então adicionar o **import** (que está logo abaixo) no início do arquivo.

```
import { Router } from '@angular/router';
import { Contato } from 'src/app/models/contato.model';
import { ContatoService } from 'src/app/services/contato.service';
```

AngularJS – Desenvolvendo um Component

Feito isso, o projeto não exibirá mais nenhum erro e o resultado das mudanças poderão ser vistas no browser. Lembrando que o projeto precisa ter sido iniciado através do comando ng serve.

OBS: É importante ressaltar que no código fonte HTML foram utilizadas duas imagens. São elas: **edit.png** e **remove.png**. Como essas imagens ainda não existem, elas não serão exibidas pelo navegador. É preciso fazer o download das imagens desejadas e colocá-las na pasta correspondente onde estão sendo buscadas.

AngularJS – Desenvolvendo um Component

Ao fazer isso, provavelmente elas serão exibidas no browser com um tamanho muito grande. Para corrigir esse problema, basta adicionar o código abaixo no arquivo **src/app/components/contato-list/contato-list.component.css**.

```
.edit{
    width:20px;
}
.remove{
    width:20px;
}
```

AngularJS – Desenvolvendo um Component

O próximo passo consiste na criação do método ngOnInit. Esse método é responsável por informar ao Angular o que deverá ser feito quando esse componente for criado.

No nosso caso específico, nós chamaremos o método **atualizaContatos** que invocará o método do serviço responsável por buscar todos os contatos no banco de dados para montarmos nossa tela de listagem.

AngularJS – Desenvolvendo um Component

O método **ngOnInit** ficará da seguinte forma:

```
ngOnInit(): void {
  this.atualizaContatos();
}
```

Podemos verificar que o método **ngOnInit** invoca um método chamado **atualizaContatos**, que ainda não foi criado. Precisamos então criar esse método.

AngularJS – Desenvolvendo um Component

O método **atualizaContatos** ficará da seguinte forma:

```
atualizaContatos(){
  this.contatoService.getContatos().subscribe({
      next: (contatos) => {
        this.contatos = contatos;
      },
      error: (erro) => {
        console.log(erro)
      }
    });
}
```

AngularJS – Desenvolvendo um Component

Feito isso, o código estará funcionando corretamente, inclusive a deleção de um registro. Só não funcionará a edição, pois ainda não foi criado o componente responsável por editar um contato e sua respectiva rota também não foi criada.

Ao fim dessas alterações, o código fonte deverá ficar da seguinte forma. Imports:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { Contato } from '../../models/contato.model';
import { Router } from '@angular/router';
import { ContatoService } from '../../services/contato.service';
```

AngularJS – Desenvolvendo um Component

Definição da classe:

```
@Component({
  selector: 'app-contato-list',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './contato-list.component.html',
  styleUrl: './contato-list.component.css'
})
export class ContatoListComponent {
    contatos:Array<Contato> = [];
    constructor(private router: Router, private contatoService: ContatoService) {}

    … DEMAIS MÉTODOS …
}
```

AngularJS – Desenvolvendo um Component

DEMAIS MÉTODOS:

```
ngOnInit(): void {
  this.atualizaContatos();
}
atualizaContatos(){
  this.contatoService.getContatos().subscribe({
      next: (contatos) => {
        this.contatos = contatos;
      },
      error: (erro) => {
        console.log(erro)
      }
    });
}
```

AngularJS – Desenvolvendo um Component

DEMAIS MÉTODOS:

```
editar(id: number){
    this.router.navigateByUrl('/contato/edit', {state: { idContato: id }})
}

remover(id: number){
    this.contatoService.removeContato(id);
    this.atualizaContatos();
}
```

AngularJS – Desenvolvendo um Component

Agora nós iremos criar um componente que será responsável por incluir um novo contato ou alterar um contato existente.

Para criar esse componente, será preciso executar o comando abaixo de dentro do diretório **src/app/components**.

ng generate component contato-edit

AngularJS – Desenvolvendo um Component

Depois de gerado o componente, chegou a hora de fazer a configuração para que este componente possa ser utilizado no projeto através de uma chamada no browser por uma URL.

Para tanto, será necessário adicionar uma rota na lista de rotas presente no arquivo **app.routes.ts**.

Atualmente já existe uma rota configurada conforme abaixo:

```
const routes: Routes = [
{path: 'contato/list', component: ContatoListComponent}
];
```

AngularJS – Desenvolvendo um Component

Será necessário adicionar mais uma rota e o código ficará da seguinte forma:

```
const routes: Routes = [
{path: 'contato/list', component: ContatoListComponent},
{path: 'contato/edit', component: ContatoEditComponent}
];
```

Para testar o funcionamento da nova rota criada, basta abrir o browser e digitar a URL

http://localhost:4200/contato/edit

AngularJS – Desenvolvendo um Component

Uma vez testado o funcionamento do componente, basta agora desenvolver o código fonte para que ele funcione conforme esperado.

Para tanto, será aberto o arquivo **src/app/components/contato-edit/contato-edit.component.html** e será substituído o código existente pelo seguinte código HTML.

AngularJS – Desenvolvendo um Component

Código fonte do arquivo **contato-edit.component.html:**

```
<form action="javascript:" [ngClass]="class_validate" [formGroup]="form_dados">
    <div class="form-row">
        <div class="form-group col-md-1"></div>
        <div class="form-group col-md-6">
            <label for="nome">Nome</label>
  <input type="text" class="form-control" id="name" formControlName="name" required>
        </div>
    </div>
    … CONTINUA
```

AngularJS – Desenvolvendo um Component

```
<div class="form-row">
    <div class="form-group col-md-1"></div>
    <div class="form-group col-md-6">
        <label for="email">E-mail</label>
                <input type="text" class="form-control" id="email"
formControlName="email" required>
    </div>
</div>
```

… CONTINUA

AngularJS – Desenvolvendo um Component

```
<div class="form-row">
    <div class="form-group col-md-1"></div>
    <div class="form-group col-md-6">
        <label for="telefone">Telefone</label>
        <input type="text" class="form-control" id="phoneNumber" formControlName="phoneNumber" required>
    </div>
</div>
```

… CONTINUA

AngularJS – Desenvolvendo um Component

```
    <div class="form-row">
        <div class="form-group col-md-1"></div>
            <div class="form-group col-md-6"><button class="btn btn-primary btn-salvar" type="submit" (click)="salvar()">Salvar</button>
        </div>
    </div>
</form>
```

AngularJS – Desenvolvendo um Component

Após essa alteração, será possível constatar que aparecerão os seguintes erros:

1. Can't bind to 'ngClass' since it isn't a known property of 'form'
2. Can't bind to 'formGroup' since it isn't a known property of 'form'.
3. Property 'class_validate' does not exist on type 'ContatoEditComponent'.
4. Property 'form_dados' does not exist on type 'ContatoEditComponent'
5. Property 'salvar' does not exist on type 'ContatoEditComponent'.

AngularJS – Desenvolvendo um Component

Para corrigirmos o primeiro erro, será necessário fazermos o import do módulo **CommonModule** na classe **ContatoEditComponent**, da mesma forma que fizemos na classe **ContatoListComponent**

Para corrigirmos o segundo erro, será necessário fazermos o import do módulo **ReactiveFormsModule** na classe **ContatoEditComponent.**

Para corrigirmos os demais erros, será necessário incluir as propriedades **class_validate** e **form_dados** e o método **salvar** dentro da classe **ContatoEditComponent**.

AngularJS – Desenvolvendo um Component

Import dos módulos **CommonModule** e **ReactiveFormsModule** na classe **ContatoEditComponent:**

```
import { CommonModule } from '@angular/common';
import { ReactiveFormsModule} from '@angular/forms';

@Component({
  selector: 'app-contato-edit',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './contato-edit.component.html',
  styleUrl: './contato-edit.component.css'
})
```

AngularJS – Desenvolvendo um Component

Inclusão dos atributos class_validate e form_dados e do método salvar:

```
class_validate = "needs-validation";
form_dados = new FormGroup({
    name: new FormControl('', Validators.required),
    email: new FormControl('', Validators.required),
    phoneNumber: new FormControl('', Validators.required)
});
salvar(){
    if(this.valida_campos_dados()){
        this.contato = Object.assign(this.form_dados.value)
    }
}
```

AngularJS – Desenvolvendo um Component

Além disso, é preciso incluir o import abaixo no arquivo src/app/components/contato-edit/contato-edit.component.ts:

```
import { FormControl, FormGroup, Validators } from '@angular/forms';
```

AngularJS – Desenvolvendo um Component

Após a realização dessas alterações, os erros citados anteriormente deixarão de existir e aparecerão dois novos erros no arquivo **src/app/components/contato-edit/contato-edit.component.ts**.

Os erros são os seguintes:

1. Property 'valida_campos_dados' does not exist on type 'ContatoEditComponent'.
2. Property 'contato' does not exist on type 'ContatoEditComponent'.

AngularJS – Desenvolvendo um Component

Essas propriedades foram incluídas dentro do método salvar, mas elas ainda não existem dentro da classe **ContatoEditComponent**.

O **valida_campos_dados** corresponde a um método que será criado enquanto a propriedade contato equivale a um objeto da classe.

AngularJS – Desenvolvendo um Component

Para criá-los, basta inserir o código abaixo dentro da classe **ContatoEditComponent**.

```
contato!: Contato;
valida_campos_dados(): boolean{
    if (this.form_dados.invalid) {
        this.class_validate = "was-validated";
        return false;
    }else{
        this.class_validate = "needs-validation";
        return true;
    }
}
```

AngularJS – Desenvolvendo um Component

O atributo **contato** pode ser criado logo acima do atributo **class_validate** enquanto que o método **valida_campos_dados()** pode ser criado logo acima do método **salvar**.

OBS: Importante lembrar que o atributo **contato** corresponde a uma instância da classe **Contato** e portanto é necessário fazer o **import** dessa classe no arquivo **src/app/components/contato-edit/contato-edit.component.ts**. Para tanto, basta incluir o código abaixo no início do arquivo, na seção de **imports**.

```
import { Contato } from 'src/app/models/contato.model';
```

AngularJS – Desenvolvendo um Component

O método salvar está apenas recuperando as informações do documento HTML e criando um objeto da classe Contato contendo essas informações.

Precisamos alterá-lo para que ele seja capaz de chamar o serviço responsável por adicionar um contato no banco de dados ou alterar um contato existente.

Para tanto, faremos a seguinte modificação no método salvar.

AngularJS – Desenvolvendo um Component

```
salvar(){
    if(this.valida_campos_dados()){
        let contato = Object.assign(this.form_dados.value)
        if(this.idContato){
            this.contato.name = contato.name;
            this.contato.email = contato.email;
            this.contato.phoneNumber = contato.phoneNumber;
            this.contatoService.editContato(this.contato);
        }else{
             this.contatoService.addContato(contato);
        }
        this.router.navigateByUrl('/contato/list');
    }
}
```

AngularJS – Desenvolvendo um Component

Podemos perceber que o método **salvar** está utilizando os atributos de classe **idContato**, router e **contatoService** que ainda não existem.

Precismos então criá-los. O atributo **idContato** será criado logo abaixo ao atributo **contato** já existente. Enquanto que os atributos **router** e **contatoService** serão criados no construtor da classe, ficando portanto da seguinte forma:

AngularJS – Desenvolvendo um Component

```
class_validate = "needs-validation";
contato!: Contato;
idContato!: number;
constructor(private router: Router, private contatoService: ContatoService) {
    const state = this.router.getCurrentNavigation()?.extras.state
    if(state && state['idContato']){
        const idContato = state['idContato'];
        if(idContato){
            this.idContato = idContato;
        }
    }
}
```

AngularJS – Desenvolvendo um Component

Não podemos nos esquecer de fazermos os imports dessas novas classes.

Agora, assim como fizemos na classe **ContatoListComponent**, nós criaremos aqui o método **ngOnInit**, que será responsável por informar ao Angular o que deverá ser feito quando esse componente for criado.

No nosso caso específico, nós verificaremos se o componente foi chamado para uma atualização ou para uma nova inserção de contato.

AngularJS – Desenvolvendo um Component

Caso, o componente tenha sido chamado para fazer uma atualização de um contato, nós invocaremos o serviço responsável por recuperar do banco de dados um contato existente a partir do seu id e setaremos esse contato como um atributo da classe e preencheremos o formulário com os dados desse contato.

Par tanto, precisaremos criar o método **preencheForm**, responsável por fazer esse procedimento.

AngularJS – Desenvolvendo um Component

```
ngOnInit(): void {
  if (this.idContato) {
    this.contatoService.getContatoById(this.idContato).subscribe({
      next: (contato) => {
        if (contato.name) {
          this.contato = contato;
          this.preencheForm(contato);
        }
      },
      error: (erro) => {
            console.log(erro)
      }
    });
  }
}
```

AngularJS – Desenvolvendo um Component

```
preencheForm(contato: Contato){
    this.f_dados.name.setValue(contato.name);
    this.f_dados.email.setValue(contato.email);
    this.f_dados.phoneNumber.setValue(contato.phoneNumber);
}

get f_dados() { return this.form_dados.controls; }
```

Com essas alterações, o componente **contato-edit.component** já está pronto, tanto para criar um novo contato, quanto para carregar um contato já existente.

AngularJS – Mecanismo de autenticação

Agora que nossa aplicação está funcionando, chegou a hora de implementarmos nosso mecanismo de segurança, com tela para autenticação.

Antes de mais nada, criaremos um novo serviço: **AuthService**. O serviço **AuthService** será responsável por chamar a api no backend responsável por fazer a autenticação e guardar na sessão do usuário a informação que o mesmo se encontra logado.

AngularJS – Mecanismo de autenticação

Para criar esses serviços, nós executaremos a partir da pasta service, os seguintes comandos:

ng generate service auth

AngularJS – Mecanismo de autenticação

Ao executarmos esse comando, será criado o arquivo **auth.service.ts** e seu respectivo arquivo de testes unitários.

auth.service.ts:

```
import { Injectable } from '@angular/core';
@Injectable({
providedIn: 'root'
})
export class AuthService {
    constructor() { }
}
```

AngularJS – Mecanismo de autenticação

Criaremos em seguida um novo model responsável por representar um usuário no sistema, com seu respectivo username e token. Para tanto, executaremos o comando abaixo de dentro da pasta models.

ng generate class user --type=model

Depois da execução do comando, será criado o arquivo **user.model.ts** com seu respectivo arquivo de testes unitários. Segue abaixo o conteúdo do arquivo criado:

user.model.ts:

```
export class User {
}
```

AngularJS – Mecanismo de autenticação

Alteraremos esse arquivo para incluirmos os atributos username e token e o construtor da classe.

```
export class User {
    username: string;
    token: string;
    constructor(username: string, token: string){
        this.username = username;
        this.token = token;
    }
}
```

AngularJS – Mecanismo de autenticação

Iremos agora alterar o serviço **AuthService**, para que ele seja capaz de efetuar o login e armazenar na sessão no navegador o usuário logado, bem como responder se um determinado usuário está logado na sessão.

Precisaremos desenvolver cinco métodos nessa classe: o método login será responsável chamar o endpoint do backend passando como parâmetro o usuário e a senha.

O método logout removerá da sessão do browser o token que foi salvo após a autenticação na aplicação.

AngularJS – Mecanismo de autenticação

O método isLoggedIn verificará se existe algum token na sessão do navegador e caso exista retornar true para indicar que o usuário está logado e retornará false quando não houver um token na sessão do navegador indicando que o usuário não está autenticado no sistema.

O método setToken armazenará na sessão do navegador um token passado como parâmetro.

E o método getToken retornará um token armazenado na sessão do navegador ou null, caso ele não exista.

AngularJS – Mecanismo de autenticação

Segue a seguir o código fonte da nossa classe **AuthService** depois de efetuadas as modificações citadas:

```
import { Injectable } from '@angular/core';
import { map, Observable } from 'rxjs';
import { HttpClient, HttpHeaders } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class AuthService {
 … CONTINUA
}
```

AngularJS – Mecanismo de autenticação

```
private loginUrl = 'http://localhost:5000/api/v1/login';
private headers = new HttpHeaders({'Content-Type': 'application/json'})

constructor(private http: HttpClient) {}

login(userParam: { username: any; password: string | null | undefined; }): Observable<any> {
  return this.http.post<any>(`${this.loginUrl}`,
    JSON.stringify({"username":userParam.username,"password":userParam.password}),{"headers":
this.headers}).pipe(
    map(data => {
      return data;
    })
    );
}
```

… CONTINUA

AngularJS – Mecanismo de autenticação

```
 isLoggedIn(): boolean {
  if(sessionStorage.getItem("token")){
     return true;
  }
  return false;
 }
setToken(token: string){
  sessionStorage.setItem("token", token);
}
getToken(): string {
  return sessionStorage.getItem("token")?? '';
}
logout() {
  sessionStorage.removeItem("token");
}
```

AngularJS – Mecanismo de autenticação

O próximo passo consiste na criação da nossa classe de segurança. Para tanto criaremos um diretório chamado security dentro da pasta src/app.

Em seguida, executaremos o comando abaixo de dentro da pasta security para criarmos nossa classe Access.

Ao executarmos o comando, será nos perguntado sobre qual interface gostaríamos de utilizar. Podemos teclar enter para selecionarmos a interface padrão **canActivate**.

ng generate guard access

AngularJS – Mecanismo de autenticação

Após a execução do comando, será criado o arquivo access.guard.ts dentro da pasta security, com o código fonte abaixo:

```
import { CanActivateFn } from '@angular/router';

export const accessGuard: CanActivateFn = (route, state) => {
  return true;
};
```

AngularJS – Mecanismo de autenticação

Podemos perceber que foi criada uma constante chamada accessGuard, que corresponde à um método chamado CanActivateFn que sempre retorna True.

É claro que precisaremos alterar esse método para que ele retorne true apenas quando o usuário estiver autenticado.

O código ficará da seguinte forma:

AngularJS – Mecanismo de autenticação

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from '../services/auth.service';

export const accessGuard: CanActivateFn = () => {
  const router = inject(Router);
  const service = inject(AuthService);
  if (service.isLoggedIn()) {
    return true;
  }
  router.navigate(['/login']);
  return false;
};
```

AngularJS – Mecanismo de autenticação

Com essa mudança, o método **canActivate** só retornará true se o usuário realmente estiver logado.

Além disso, ele redirecionará a aplicação para a tela de login, caso o usuário não esteja logado.

O passo seguinte consiste na criação do nosso componente responsável pela autenticação da nossa aplicação. Para criar esse componente, nós executaremos o comando seguinte, a partir da pasta Components:

ng generate component login

AngularJS – Mecanismo de autenticação

Depois de gerado o componente **LoginComponent**, nós faremos alterações no arquivo **login.component.html** para incluirmos o formulário de autenticação da nossa aplicação. Ele ficará da seguinte forma:

AngularJS – Mecanismo de autenticação

```
<h3>Autenticação</h3>
<h3 *ngIf="mensagem">{{ mensagem }}</h3>
<form [ngClass]="class_validate" [formGroup]="form_login" (ngSubmit)="logar()">
    <div class="form-group">
      <input type="text" matInput class="form-control" placeholder="usuário" formControlName="username"
required>
    </div>
    <div class="form-group">
      <input type="password" class="form-control" placeholder="senha" formControlName="password" required>
    </div>
    <div class="form-group">
      <input type="submit" mat-button value="Logar" class="btn float-right login_btn">
    </div>
</form>
```

AngularJS – Mecanismo de autenticação

Aparecerão alguns erros na nossa aplicação. Isso porque ainda não temos o método logar criado no nossa classe **LoginComponent**, nem os atributos **username**, **password**, **class_validate** e **mensagem**.

Além disso, assim como no componente de edição, precisamos fazer o import do **CommonModule** e do **ReactiveFormsModule**

Faremos então alterações no nosso arquivo **login.component.ts** que deverá ficar da seginte forma:

AngularJS – Mecanismo de autenticação

```
import { Component } from '@angular/core';
import { ReactiveFormsModule, FormGroup, FormControl, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { AuthService } from 'src/app/services/auth.service';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './login.component.html',
  styleUrl: './login.component.css'
})
export class LoginComponent {

    … CONTINUA ….
}
```

AngularJS – Mecanismo de autenticação

```
class_validate = "needs-validation";
mensagem = "";
form_login = new FormGroup({
    username: new FormControl('', Validators.required),
    password: new FormControl('', Validators.required)
});

constructor(private router: Router, private authService: AuthService) {
}
```

AngularJS – Mecanismo de autenticação

```
valida_campos_dados(): boolean{
    if (this.form_login.invalid) {
        this.class_validate = "was-validated";
        return false;
    }else{
        this.class_validate = "needs-validation";
        return true;
    }
}
```

AngularJS – Mecanismo de autenticação

```
logar(){
    if(this.valida_campos_dados()){
                        let user = {"username": this.form_login.get('username')?.value, "password":
this.form_login.get('password')?.value};
      if(! this.authService.isLoggedIn()){
        this.authService.login(user).subscribe({
          next: token => {
              if(token){
                this.authService.setToken(token[ 'token']);
                this.router.navigateByUrl('/contato/list');
              }
          },
          error: error => {
              console.error('Houve um erro:', error);
              console.log(error.error.message);
              this.mensagem = error.error.message;
          }
        });
      }else{
        this.router.navigateByUrl('/contato/list');
      }
    }
  }
```

AngularJS – Mecanismo de autenticação

Agora que nós já temos nosso componente de login, bem como nosso mecanismo de segurança devidamente configurado, precisaremos alterar o arquivo **app-routing.module.ts** para incluir a rota da página de login e a configuração do mecanismo de segurança nas demais rotas.

Depois das modificações, a lista de rotas deverá ficar da seguinte forma:

AngularJS – Mecanismo de autenticação

```
export const routes: Routes = [
            {path:   'contato/list',   canActivate:[accessGuard],
component: ContatoListComponent},
            {path:   'contato/edit',   canActivate:[accessGuard],
component: ContatoEditComponent},
    {path: 'login', component: LoginComponent}
];
```

AngularJS – Mecanismo de autenticação

Faremos agora a última alteração no nosso frontend.

Na classe **ContatoService**, todas as vezes que nós fazíamos uma chamada a uma de nossas apis, nós estávamos utilizando um headers com um token fixo.

Agora nós vamos alterar o nosso atributo headers para, ao invés de utilizar um token fixo, ele pegar o token que foi armazenado na sessão do navegador no momento da autenticação.

AngularJS – Mecanismo de autenticação

Atributo headers antigo na classe ContatoService:

```
Private headers = new HttpHeaders({"Authorization":"tokenJWT"})
```

Alterar para:

```
private headers = new HttpHeaders({"Authorization":""})
```

Nós precisaremos injetar um objeto da classe **AuthService** no construtor da nossa classe **ContatoService**. Dessa forma, o construtor da classe **ContatoService** deverá ficar da seguinte forma:

```
constructor(private http: HttpClient, private authService: AuthService) {
   this.headers = new HttpHeaders({"Authorization":this.authService.getToken()})
 }
```

AngularJS – Função para gerar HASH

Agora vai uma dica útil. Como desenvolver uma função que cria um HASH a partir de uma string.

Antes de mais nada precisamos entender que o HASH é um mecanismo matemático que gera uma cadeia de caracteres única a partir de uma outra cadeia de caracteres.

Existem alguns princípios básicos dos algoritmos de HASH que falaremos agora:

AngularJS – Função para gerar HASH

1º – Um algoritmo de hash sempre gerará o mesmo hash da mesma cadeia de caracteres

2º – Nunca haverão dois hashs iguais a partir de cadeias de caracteres diferentes

3º – Não será possível a partir de um hash, voltar para a cadeia de caracteres que o originou

AngularJS – Função para gerar HASH

Podemos citar 4 algoritmos de HASH:

MD5: Um dos mais antigos, mas considerado inseguro para muitas aplicações por ser suscetível a colisões.

SHA-1: Mais seguro que o MD5, mas também apresenta vulnerabilidades.

SHA-256: Considerado seguro para a maioria das aplicações.

SHA-512: Versão mais robusta do SHA-256, gerando hashes ainda maiores.

AngularJS – Função para gerar HASH

Mas afinal, como criamos uma função que cria um HASH a partir de uma cadeia de caracteres no Angular.

Antes de mais nada, precisamos instalar a biblioteca CryptoJS.

npm install crypto-js –save

Em seguida, precisaremos desenvolver nossa função. Neste projeto, nós criaremos uma pasta chamada util e dentro dela um arquivo chamado geradorHash.ts

AngularJS – Função para gerar HASH

No arquivo geradorHash.ts, colocaremos o código abaixo:

```
import * as CryptoJS from 'crypto-js';

export function gerarHash(senha: string) {
    return CryptoJS.SHA256(senha).toString();
}
```

Agora é só usar.

AngularJS – Mecanismo de autenticação

Ainda precisamos fazer algumas alterações na nossa aplicação frontend para que ela seja capaz de exibir mensagens de erros, como por exemplo, quando tentamos logar na aplicação com um login inválido.

Deixo essa alteração para vocês tentarem fazer.