DESENVOLVIMENTO WEB – FRONT-END

AULA 6

Prof. Mauricio Antonio Ferste

CONVERSA INICIAL

Nesta etapa, teremos um fechamento, buscando entender como conectar o *back-end* com o *front-end*. Trata-se de um dos pontos mais complicados. Pense que existe um limite, uma divisa para passar os parâmetros de um lado para outro.

TEMA 1 – REFORÇO SOBRE GET E POST

Para consumir APIs (Interfaces de Programação de Aplicativos), você pode seguir os seguintes passos:

- Identifique a API: primeiro, você precisa identificar a API que deseja consumir. Isso pode ser feito pesquisando a documentação da API, verificando se há uma documentação oficial, explorando recursos online ou entrando em contato com os desenvolvedores responsáveis.
- Obtenha uma chave de API (se necessário): alguns serviços de API exigem uma chave de autenticação para acessá-los. Se for o caso, você precisará obter uma chave de API registrando-se no serviço correspondente ou seguindo as instruções fornecidas pela API.
- Determine o método de autenticação: verifique qual método de autenticação é necessário para acessar a API. Alguns serviços podem usar autenticação baseada em token, OAuth, chaves de API ou outros métodos. Entenda os requisitos de autenticação e como incluí-los em suas solicitações para a API

1.1 JWT

Um ponto interessante é JWT. Assista ao vídeo <https://www.youtube.com/watch?v=sHyoMWnnLGU>. Acesso em: 19 set. 2023.

- Utilize uma biblioteca ou cliente HTTP: Para facilitar o consumo da API, muitas vezes é recomendado utilizar uma biblioteca ou cliente HTTP específico para a sua linguagem de programação. Essas bibliotecas fornecem métodos e funcionalidades pré-definidos para enviar solicitações HTTP e lidar com as respostas da API. Verifique se existe

uma biblioteca ou cliente HTTP disponível para a linguagem de programação escolhida e aprenda a utilizá-la.

- Realize as solicitações HTTP: com a biblioteca ou cliente HTTP configurado, você pode começar a fazer solicitações à API. Isso geralmente envolve construção de uma solicitação HTTP (como GET, POST, PUT, DELETE) com os parâmetros e cabeçalhos adequados, envio da solicitação para a URL da API e tratamento da resposta retornada.
- Analise a resposta: ao receber a resposta da API, você precisa analisar os dados retornados e extrair as informações relevantes para o seu aplicativo. Isso pode envolver a leitura e a manipulação de dados em formato JSON, XML ou outro formato definido pela API.
- Lide com erros e exceções: durante o consumo da API, é importante lidar com erros e exceções. Isso inclui tratar códigos de status HTTP não esperados, erros de autenticação, limites de taxa, entre outros. Certifique-se de implementar a lógica adequada para lidar com essas situações e fornecer mensagens de erro adequadas aos usuários do seu aplicativo.
- Teste e itere: à medida que você consome a API, é fundamental testar todas as funcionalidades e garantir que os dados estejam sendo recuperados e manipulados corretamente. Faça testes abrangentes para identificar possíveis problemas ou erros. Se necessário, faça iterações no código para melhorar a integração com a API.

1.2 Erros HTTP

Como já vimos, erros HTTP são códigos de status retornados por um servidor web para indicar o resultado de uma solicitação HTTP. Os erros mais comuns incluem:

- Erros do grupo 4xx: esses erros geralmente indicam problemas com a solicitação do cliente. Alguns exemplos são:
    - 400 Bad Request: indica que a solicitação do cliente não pôde ser entendida ou foi malformada.
    - 401 Unauthorized: indica que o cliente não está autorizado a acessar o recurso solicitado.

- 404 Not Found: indica que o recurso solicitado não foi encontrado no servidor.

- Erros do grupo 5xx: esses erros indicam problemas no servidor ao processar a solicitação. Alguns exemplos são:
  - 500 Internal Server Error: indica um erro genérico no servidor que pode ter várias causas.
  - 502 Bad Gateway: indica que o servidor atuou como um gateway ou proxy, mas recebeu uma resposta inválida do servidor upstream.

Para resumir um código de resposta HTTP, normalmente é utilizado um código de status. Cada código de status tem um significado específico, como os já mencionados. Ao lidar com respostas HTTP em seu código ou desenvolvimento web, você pode consultar a tabela de códigos de status HTTP para entender melhor o que cada código representa.

TEMA 2 – REFORÇO SOBRE POST, PUT, DELETE

Vamos reforçar os conceitos sobre os métodos HTTP POST, PUT e DELETE, amplamente utilizados para operações de criação, atualização e exclusão de recursos em aplicações web com arquitetura RESTful.

2.1 POST

O método HTTP POST é usado para enviar dados ao servidor com o objetivo de criar um novo recurso. Quando você envia uma requisição POST para uma URL, o servidor processa os dados enviados e cria um novo recurso com base nesses dados. Normalmente, a resposta do servidor contém informações sobre o novo recurso criado, como um identificador único (ID) ou uma URL para acessá-lo posteriormente.

Exemplo de requisição POST (em JSON):

```
POST /api/usuarios
Content-Type: application/json

{
  "nome": "João da Silva",
  "email": "joao@example.com",
```

```
  "idade": 30
}
```

2.2 PUT

O método HTTP PUT é utilizado para atualizar um recurso existente no servidor. Ao enviar uma requisição PUT para uma URL específica, você está informando ao servidor que deseja atualizar o recurso que corresponde àquela URL com os dados fornecidos na requisição.

É importante destacar que, ao usar o PUT, você deve fornecer todos os campos necessários para atualizar o recurso, mesmo que apenas um campo tenha sido alterado. Caso contrário, os campos não fornecidos na requisição podem ser definidos como nulos ou valores padrão no servidor.

Exemplo de requisição PUT (em JSON):

```
PUT /api/usuarios/1
Content-Type: application/json

{
  "nome": "João da Silva",
  "email": "joao.silva@example.com",
  "idade": 31
}
```

2.3 DELETE

O método HTTP DELETE é usado para solicitar a exclusão de um recurso específico no servidor. Ao enviar uma requisição DELETE para uma URL, você está instruindo o servidor a remover permanentemente o recurso correspondente àquela URL. O servidor pode responder com um código de status 204 no Content, para indicar que o recurso foi excluído com sucesso. No entanto, é possível que o servidor retorne outro código de status, como 200 OK, para indicar que a operação foi concluída com êxito.

```
uos/1
DELETE /api/usuarios/1
```

Neste caso, estamos trabalhando com uma API relacionada a usuários. Desejamos excluir o usuário com o identificador 1. A parte "/api/usuarios" indica que estamos lidando com um recurso relacionado a usuários na API. Portanto,

ao enviar uma solicitação DELETE para "/api/usuarios/1", estamos solicitando que o servidor remova o usuário com o ID 1 do banco de dados ou do armazenamento correspondente.

É importante notar que o uso correto dos métodos HTTP é fundamental para seguir os princípios do design RESTful e garantir que as operações de sua aplicação web sejam realizadas de forma consistente e semântica. Além disso, é importante garantir que as operações de atualização e exclusão sejam protegidas com autenticação e autorização adequadas, para evitar que usuários não autorizados realizem essas ações.

2.4 Na prática em nosso projeto do lado da API

Bem, para nosso projeto, temos de tomar algumas atitudes que o tornam o código mais legível. Primeiramente, do lado da API temos uma classe que reflete a entidade do banco (está em nosso projeto, consta aqui para entendimento, pode ser copiada para um componente).

```
import { Entity, PrimaryGeneratedColumn, Column } from
"typeorm";

@Entity()
export class User {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  firstName: string;

  @Column()
  lastName: string;

  @Column()
  age: number;

}
```

Para constar:

- import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";: esta linha de código importa as três principais decoradoras do TypeORM: Entity, PrimaryGeneratedColumn e Column. Elas serão usadas para definir e configurar a entidade de banco de dados "User".

- @Entity(): esta decoradora é usada acima da classe User e indica que essa classe representa uma entidade de banco de dados. Cada instância dessa classe representará uma linha na tabela correspondente no banco de dados.
- @PrimaryGeneratedColumn(): esta decoradora é usada acima da propriedade id e indica que essa coluna é a chave primária da tabela. Além disso, é configurada para que seja gerada automaticamente.

Para essa entidade, que reflete o banco, temos de ter alguém que a controle, que a comande, o que chamamos de controlador, conforme segue:

```
import { NextFunction, Request, Response } from "express"
import { User } from "../entity/User"
import { AppDataSource } from "../data-source"

export class UserController {

    private userRepository = AppDataSource.getRepository(User)

    async all(request: Request, response: Response, next:
NextFunction) {
        return this.userRepository.find()
    }

    async one(request: Request, response: Response, next:
NextFunction) {

        return this.userRepository.findOneBy({ id:
request.params.id })
    }

    async save(request: Request, response: Response, next:
NextFunction) {
        return this.userRepository.save(request.body)
    }

    async remove(request: Request, response: Response, next:
NextFunction) {
        let userToRemove = await this.userRepository.findOneBy({
id: request.params.id })
        await this.userRepository.remove(userToRemove)
    }

}
```

- import { NextFunction, Request, Response } from "express";: estas importações trazem as definições de tipos necessárias para trabalhar com

objetos Request, Response e NextFunction do Express, usados para manipular as requisições HTTP.

- import { User } from "../entity/User";: esta importação traz a classe User da entidade (model) definida no arquivo "../entity/User.ts". A classe User representa o modelo de dados do usuário no aplicativo.
- import { AppDataSource } from "../data-source";: esta importação traz a classe AppDataSource do arquivo "../data-source.ts". Supõe-se que esta classe forneça um acesso à fonte de dados, como a conexão com o banco de dados, por meio do TypeORM.
- private userRepository = AppDataSource.getRepository(User);: esta linha declara uma propriedade privada chamada userRepository e a inicializa com o repositório (classe do TypeORM) associado à entidade User. A classe AppDataSource é usada para obter esse repositório.
- async all(request: Request, response: Response, next: NextFunction) { ... }: este é um método assíncrono chamado *all,* que trata a rota para buscar todos os usuários. Quando essa rota é acessada (por exemplo, usando um método GET em "/users"), o método all é chamado para responder com todos os usuários do banco de dados.
- async one(request: Request, response: Response, next: NextFunction) { ... }: este é um método assíncrono que trata a rota para buscar um usuário específico. Quando essa rota é acessada (por exemplo, usando um método GET em "/users/1"), o método one é chamado para responder com os detalhes do usuário cujo ID é fornecido nos parâmetros da URL.
- async save(request: Request, response: Response, next: NextFunction) { ... }: este é um método assíncrono chamado *save*, que trata a rota para criar ou atualizar um usuário. Quando essa rota é acessada (por exemplo, usando um método POST ou PUT em "/users"), o método save é chamado para criar um novo usuário com os dados fornecidos no corpo da requisição (request.body), ou atualizar um usuário existente com esses dados.
- async remove(request: Request, response: Response, next: NextFunction) { ... }: este é um método assíncrono chamado *remove,* que trata a rota para excluir um usuário. Quando essa rota é acessada (por exemplo, usando um método DELETE em "/users/1"), o método remove

é chamado para excluir o usuário cujo ID é fornecido nos parâmetros da URL.

Em resumo, esse controlador é responsável por tratar as solicitações HTTP relacionadas aos usuários e interagir com o banco de dados por meio do TypeORM. Cada método do controlador (all, one, save, remove) corresponde a uma operação diferente que pode ser realizada em relação aos usuários da aplicação.

Agora falta expor o nosso código do lado da API. Para tanto, precisamos criar rotas REST:

```
import { UserController } from "./controller/UserController"

export const Routes = [{
    method: "get",
    route: "/users",
    controller: UserController,
    action: "all"
}, {
    method: "get",
    route: "/users/:id",
    controller: UserController,
    action: "one"
}, {
    method: "post",
    route: "/users",
    controller: UserController,
    action: "save"
}, {
    method: "delete",
    route: "/users/:id",
    controller: UserController,
    action: "remove"
}]
```

Este é o array chamado "Routes", que define as rotas da aplicação. Cada objeto dentro do array representa uma rota específica, com as seguintes propriedades:

- method: o método HTTP que a rota suporta, como "get", "post" ou "delete".
- route: a URL associada à rota, por exemplo "/users" ou "/users/:id" (com um parâmetro de ID).
- controller: o controlador que será responsável por lidar com as solicitações para essa rota, no caso o UserController.

- action: o método do controlador que será invocado quando a rota for acessada. Por exemplo, "all" chamará o método all do UserController, "one" chamará o método one, e assim por diante.

Em resumo, o array "Routes" está mapeando cada rota da aplicação a um controlador específico e a um método desse controlador que irá tratar a solicitação. Por exemplo, ao acessar a URL "/users" usando o método HTTP GET, o controlador UserController será chamado, e o método all dentro desse controlador será executado para retornar todos os usuários. De forma similar, ao acessar a URL "/users/:id" usando o método HTTP GET, o método one do controlador será executado para retornar um usuário específico com base no ID fornecido. As rotas também suportam os métodos HTTP POST e DELETE, para criar e excluir usuários, respectivamente.

2.5 Na prática em nosso projeto do lado WEB, criando a conexão de serviços

Nesse código, temos um serviço chamado HttpService, que é responsável por realizar requisições HTTP usando o Angular HttpClient. Vamos entender o que cada parte do código representa:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { IResultHttp } from '../interfaces/IResultHttp';


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


constructor(
private http: HttpClient) {


}


private createHeader(header?: HttpHeaders): HttpHeaders {


if (!header) {
```

```
header = new HttpHeaders();
}


header = header.append('Content-Type', 'application/json');
header = header.append('Accept', 'application/json');
header = header.append('Access-Control-Allow-Origin',"*")


const token = localStorage.getItem('faind:token');
if (token) {
header = header.append('x-token-access', token);
}


return header;
}


public get(url: string): Promise<IResultHttp> {
const header = this.createHeader();
console.log("header " + header)
console.log(url)
return new Promise(async (resolve) => {
try {
const res = await this.http.get(url, { headers: header
}).toPromise();
console.log("aqui")
resolve({ success: true, data: res, error: undefined });
} catch (error) {
resolve({ success: false, data: {}, error });
}
});
}


public post(url: string, model: any, headers?: HttpHeaders):
Promise<IResultHttp> {
const header = this.createHeader(headers);
return new Promise(async (resolve) => {
try {
const res = await this.http.post(url, model, { headers: header
}).toPromise();
resolve({ success: true, data: res, error: undefined });
} catch (error) {
resolve({ success: false, data: {}, error });
}
});
}
```

```
public delete(url: string): Promise<IResultHttp> {
const header = this.createHeader();
console.log("antes")
console.log(url)
return new Promise(async (resolve) => {
try {
const res = await this.http.delete(url, { headers: header
}).toPromise();
resolve({ success: true, data: res, error: undefined });
} catch (error) {
resolve({ success: false, data: {}, error });
}
});
}


}
```

- import { Injectable } from '@angular/core';: esta linha de código importa o decorador Injectable do Angular. O decorador Injectable é usado para permitir a injeção de dependência em uma classe.
- import { HttpClient, HttpHeaders } from '@angular/common/http';: esta linha de código importa a classe HttpClient e a classe HttpHeaders do módulo @angular/common/http. O HttpClient é usado para fazer requisições HTTP e o HttpHeaders é usado para configurar os cabeçalhos das requisições.
- import { IResultHttp } from '../interfaces/IResultHttp';: esta linha de código importa a interface IResultHttp do arquivo IResultHttp.ts, que provavelmente define a estrutura dos resultados das requisições HTTP.
- @Injectable({ providedIn: 'root' }): este é o decorador Injectable aplicado à classe HttpService. O parâmetro { providedIn: 'root' } indica que o Angular deve criar uma única instância compartilhada de HttpService em todo o aplicativo (singleton). Isso permite que o serviço seja injetado em outros componentes ou serviços.
- constructor(private http: HttpClient) { }: este é o construtor do serviço HttpService, executado quando uma instância do serviço é criada. Ele recebe uma instância de HttpClient como parâmetro, que será usada para fazer as requisições HTTP.
- private createHeader(header?: HttpHeaders): HttpHeaders { ... }: este é um método privado chamado createHeader, que é usado para criar os

cabeçalhos da requisição. Ele recebe um cabeçalho opcional como parâmetro, que pode ser fornecido quando necessário. Caso não seja fornecido, um novo cabeçalho é criado. Dentro do método, os cabeçalhos padrão são definidos, incluindo "Content-Type" e "Accept" como "application/json". Além disso, ele verifica se há um token de acesso salvo no localStorage do navegador. Se houver, adiciona-o ao cabeçalho como "x-token-access".

- public get(url: string): Promise<IResultHttp> { ... }: este é o método público get do serviço, usado para fazer uma requisição HTTP GET para a URL fornecida. Ele recebe a URL como parâmetro e retorna uma promessa (Promise)

2.6 Na prática em nosso projeto do lado WEB, outras classes

A classe BaseModel é uma classe abstrata em TypeScript. Classes abstratas não podem ser instanciadas diretamente, mas podem ser usadas como base para outras classes que estendem (herdam) delas. Vamos entender o que esta classe faz:

```
export abstract class BaseModel {
  id: string | undefined;
}
```

Assim, podemos implementar as classes concretas. Nesse código, temos uma classe chamada UsuarioModel que estende a classe abstrata BaseModel. Vamos entender o que cada parte do código representa:

```
import { BaseModel } from './baseModel';

export class UsuarioModel extends BaseModel {
  firstName: string | undefined;
  lastName: string | undefined;
  age: BigInt | undefined;
}
```

- import { BaseModel } from './baseModel';: esta linha importa a classe abstrata BaseModel do arquivo baseModel.ts. A classe UsuarioModel irá estender (herdar) dessa classe, o que significa que ela receberá todas as propriedades e métodos definidos em BaseModel.
- export class UsuarioModel extends BaseModel { ... }: esta é a declaração da classe UsuarioModel, que estende a classe abstrata BaseModel. A palavra-chave extends indica que UsuarioModel herda as características

de BaseModel. Isso significa que UsuarioModel terá a propriedade id, pois ela é definida em BaseModel.

- firstName: string | undefined;: esta linha declara uma propriedade chamada firstName na classe UsuarioModel. A propriedade firstName é do tipo string ou undefined, o que significa que pode conter um valor de texto ou não ter um valor definido.
- lastName: string | undefined;: esta linha declara uma propriedade chamada lastName na classe UsuarioModel. A propriedade lastName também é do tipo string ou undefined, permitindo que contenha um valor de texto ou não tenha um valor definido.
- age: BigInt | undefined;: esta linha declara uma propriedade chamada *age* na classe UsuarioModel. A propriedade age é do tipo BigInt ou undefined, o que significa que pode conter um valor inteiro muito grande ou não ter um valor definido.

Em resumo, a classe UsuarioModel é uma classe que representa um modelo de usuário com propriedades como firstName, lastName e age. Ela estende a classe abstrata BaseModel, que fornece a propriedade id. Ao herdar de BaseModel, UsuarioModel recebe automaticamente a propriedade id, além de definir suas próprias propriedades específicas. Essa abordagem é útil quando vários modelos (classes) têm propriedades comuns (como id neste caso), e quando desejamos evitar repetição de código ao herdar as características comuns de uma classe base.

Nesse código, temos uma classe abstrata chamada BaseService<T>, que serve como uma base para serviços que fazem requisições HTTP para uma API. Vamos entender cada parte do código:

```
import { environment } from './../../environments/environment';
import { HttpService } from './../services/http.service';
import { IResultHttp } from '../interfaces/IResultHttp';

export abstract class BaseService<T> {

  urlBase: string = '';

  constructor(
    public url: string,
    public http: HttpService) {
    this.urlBase = `${environment.url_api}/${this.url}`;
  }
```

```
}
```

- import { environment } from './../../environments/environment';: esta linha de código importa o objeto environment do arquivo environment.ts localizado no diretório environments. O objeto environment é utilizado para acessar variáveis de ambiente, como URLs de API, dependendo do ambiente em que a aplicação está sendo executada (por exemplo, ambiente de desenvolvimento, teste ou produção).
- import { HttpService } from './../services/http.service';: esta linha de código importa a classe HttpService do arquivo http.service.ts, que provavelmente é um serviço personalizado para realizar requisições HTTP.
- import { IResultHttp } from '../interfaces/IResultHttp';: esta linha de código importa a interface IResultHttp do arquivo IResultHttp.ts, que define a estrutura dos resultados das requisições HTTP.
- export abstract class BaseService<T> { ... }: esta é a declaração da classe abstrata BaseService<T>. A classe é definida com o tipo genérico <T>, o que significa que podemos definir um tipo específico para T quando estendemos essa classe.
- urlBase: string = '';: esta linha declara uma propriedade chamada urlBase do tipo string e inicializa seu valor como uma string vazia.
- constructor(public url: string, public http: HttpService) { ... este é o construtor da classe BaseService, executado quando uma classe que estende BaseService é instanciada. Ele recebe dois parâmetros: url (uma string) e http (um objeto do tipo HttpService). Dentro do construtor, a propriedade urlBase é configurada para conter a URL base da API, que é composta concatenando o valor de environment.url_api (a URL base da API definida no arquivo environment.ts) com o valor de this.url.
- this.urlBase = ${environment.url_api}/${this.url};: esta linha concatena a URL base da API (environment.url_api) com a URL específica do serviço (this.url), para formar a URL completa da API, e armazena o resultado em urlBase. A propriedade this.url foi definida no construtor e provavelmente representa uma parte específica da URL para o serviço (por exemplo, "/users" ou "/products").

Em resumo, a classe BaseService<T> fornece uma base comum para outros serviços que precisam fazer requisições HTTP para uma API. Ela utiliza a propriedade urlBase para armazenar a URL completa da API (que inclui a URL base da API definida em environment.url_api mais a parte específica do serviço). Isso permite que os serviços que estendem BaseService possam realizar requisições HTTP de forma mais fácil e consistente, aproveitando a URL base da API configurada de forma centralizada.

Agora podemos criar nosso serviço para expor na tela:

```
import { UsuarioModel } from './../model/usuarioModel';
import { Injectable } from '@angular/core';
import { BaseService } from '../base/base.service';
import { HttpService } from './http.service';
import { IResultHttp } from '../interfaces/IResultHttp';
import { environment } from './../../environments/environment';
import { Observable, Subject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class UsuarioService extends BaseService<UsuarioModel> {

  private loginSubject = new Subject<boolean>();

  constructor(public override http: HttpService) {
    super('users', http);
  }

  login(email: string, password: string): Promise<IResultHttp> {
    return this.http.post(`${environment.url_api}/users/auth`, {
email, password });
  }
}
```

- import { UsuarioModel } from './../model/usuarioModel';: esta linha de código importa a classe UsuarioModel do arquivo usuarioModel.ts. Presumivelmente, UsuarioModel é uma classe que representa o modelo de dados para usuários.
- import { Injectable } from '@angular/core';: esta linha de código importa o decorador Injectable do Angular. O decorador Injectable é usado para permitir a injeção de dependência em uma classe. Ao adicionar o decorador @Injectable({ providedIn: 'root' }), o serviço UsuarioService pode ser injetado em qualquer parte do aplicativo.

- import { BaseService } from '../base/base.service';: esta linha de código importa a classe BaseService do arquivo base.service.ts. Como vimos em um código anterior, BaseService é uma classe abstrata que fornece uma base comum para serviços que fazem requisições HTTP para uma API.
- import { HttpService } from './http.service';: esta linha de código importa a classe HttpService do arquivo http.service.ts. Presumivelmente, HttpService é um serviço personalizado que realiza requisições HTTP.
- import { IResultHttp } from '../interfaces/IResultHttp';: esta linha de código importa a interface IResultHttp do arquivo IResultHttp.ts, que define a estrutura dos resultados das requisições HTTP.
- import { environment } from './../../environments/environment';: esta linha de código importa o objeto environment do arquivo environment.ts, que contém variáveis de ambiente do Angular, como URLs de API, dependendo do ambiente em que o aplicativo está sendo executado.
- import { Observable, Subject } from 'rxjs';: esta linha de código importa as classes Observable e Subject do RxJS. Observable é usada para trabalhar com fluxos de dados assíncronos, enquanto Subject é um tipo de Observable que permite emitir e ouvir eventos.
- @Injectable({ providedIn: 'root' }): este é o decorador Injectable aplicado à classe UsuarioService. O parâmetro { providedIn: 'root' } indica que o Angular deve criar uma única instância compartilhada de UsuarioService em todo o aplicativo (singleton).
- export class UsuarioService extends BaseService<UsuarioModel> { ... }: esta é a declaração da classe UsuarioService, que estende a classe abstrata BaseService<UsuarioModel>. Isso significa que UsuarioService herda as características de BaseService, incluindo a propriedade urlBase e os métodos para fazer requisições HTTP.
- private loginSubject = new Subject<boolean>();: esta linha declara uma propriedade privada chamada loginSubject, que é um objeto do tipo Subject<boolean>. Esse objeto Subject será usado para emitir eventos relacionados ao login.
- constructor(public override http: HttpService) { ... }: este é o construtor da classe UsuarioService, executado quando uma instância de UsuarioService é criada. Ele recebe um parâmetro chamado http, que é do tipo HttpService. A palavra-chave public override é usada para indicar

que a propriedade http é uma propriedade herdada de BaseService, que está sendo substituída (override) na classe UsuarioService.

- Dentro do construtor, a classe super('users', http); é chamada, passando a string 'users' como o primeiro argumento e o objeto http como o segundo argumento. A chamada para super() invoca o construtor da classe BaseService, garantindo que a propriedade urlBase seja configurada corretamente.
- login(email: string, password: string): Promise<IResultHttp> { ... }: este é um método chamado login que recebe dois parâmetros email e password (ambos do tipo string). O método retorna uma promessa (Promise) que resolve com um objeto do tipo IResultHttp.

Dentro do método, uma requisição HTTP POST é feita para a URL ${environment.url_api}/users/auth, passando um objeto com as propriedades email e password como dados para a requisição. O resultado da requisição é retornado como uma promessa.

Em resumo, o UsuarioService é um serviço Angular que estende BaseService e fornece funcionalidades específicas para trabalhar com usuários. Ele herda a propriedade urlBase e os métodos HTTP de BaseService, tornando mais fácil fazer requisições HTTP para a API com a URL correta. O serviço também tem um método login que envia uma requisição HTTP POST para autenticar um usuário com base no email e senha fornecidos. Além disso, o serviço utiliza um Subject privado chamado loginSubject para emitir eventos relacionados ao login, que pode ser observado em outras partes do aplicativo.

Pronto! Ufa, deu trabalho, mas agora podemos usar o dado do serviço no componente que já vimos. Por exemplo, uma lista de usuários em tela.

```
import { Component, OnInit } from '@angular/core';
import { UsuarioService } from '../../services/usuario.service'
import { UsuarioModel } from '../../model/usuarioModel'
import { MatFormFieldModule} from '@angular/material/form-
field';
import { IUsuario } from '../../interfaces/IUsuario'
import { MatTableDataSource } from '@angular/material/table';
import { Router, ActivatedRoute } from '@angular/router';



@Component({
selector: 'app-usuarios',
```

```
templateUrl: './usuarios.component.html',
styleUrls: ['./usuarios.component.scss']
})


export class UsuariosComponent implements OnInit {
columns: string[] = ['id', 'Nome', 'Sobrenome'];
// origem dos dados
dataSource!: MatTableDataSource<IUsuario>;


constructor(private usuarioSrv: UsuarioService,
private router: Router,
private active: ActivatedRoute) {
}


ngOnInit() {
this.bind();
}


async bind() {
console.log("inicio")
const usuarios = await this.usuarioSrv.GetAll();
console.log ("----");
console.log(usuarios);
console.log ("----");
this.dataSource = new MatTableDataSource(usuarios.data);
}


async delete(usuario: UsuarioModel): Promise<void> {
const result = await this.usuarioSrv.delete(usuario.id);
this.bind();
this.router.navigateByUrl('/usuarios');
}
}
```

Nesse código, temos um componente chamado UsuariosComponent, responsável por exibir uma lista de usuários e realizar a exclusão de usuários através do serviço UsuarioService. Vamos entender o que cada parte do código representa:

- typescript import { Component, OnInit } from '@angular/core';: esta linha de código importa as classes Component e OnInit do Angular. Component é usada para definir componentes Angular, enquanto OnInit é uma

interface que contém o método ngOnInit(),executado após a inicialização do componente.

- import { UsuarioService } from '../../services/usuario.service';: esta linha de código importa o serviço UsuarioService do arquivo usuario.service.ts, que provavelmente é responsável por lidar com as operações relacionadas aos usuários.
- import { UsuarioModel } from '../../model/usuarioModel';: esta linha de código importa a classe UsuarioModel do arquivo usuarioModel.ts, que provavelmente é uma representação do modelo de dados para usuários.
- import { MatFormFieldModule } from '@angular/material/form-field';: esta linha de código importa o módulo MatFormFieldModule do Angular Material, que é usado para criar campos de formulário estilizados.
- import { IUsuario } from '../../interfaces/IUsuario';: esta linha de código importa a interface IUsuario do arquivo IUsuario.ts, que provavelmente define a estrutura de um usuário.
- import { MatTableDataSource } from '@angular/material/table';: esta linha de código importa a classe MatTableDataSource do Angular Material, que é usada para representar a origem dos dados de uma tabela.
- import { Router, ActivatedRoute } from '@angular/router';: esta linha de código importa as classes Router e ActivatedRoute do Angular, usadas para lidar com navegação e acesso a parâmetros de rota.
- @Component({ ... }): este é o decorador Component, aplicado à classe UsuariosComponent, que define o comportamento do componente.
- selector: 'app-usuarios', ...: o seletor do componente é definido como 'app-usuarios', o que significa que o componente pode ser usado em templates usando a tag <app-usuarios></app-usuarios>.
- templateUrl: './usuarios.component.html', ...: o arquivo de template do componente é definido como usuarios.component.html, que provavelmente contém a marcação HTML para a exibição da lista de usuários.
- styleUrls: ['./usuarios.component.scss'] ...: o arquivo de estilo do componente é definido como usuarios.component.scss, que provavelmente contém os estilos CSS para o componente.
- columns: string[] = ['id', 'Nome', 'Sobrenome'];: esta linha declara um array de strings chamado columns, que define as colunas a serem exibidas na

tabela de usuários. As colunas são identificadas pelos seus nomes (id, Nome e Sobrenome).

- dataSource!: MatTableDataSource<IUsuario>;: esta linha declara uma propriedade chamada dataSource, que é uma instância de MatTableDataSource<IUsuario>, que representa a origem dos dados da tabela de usuários.
- constructor(private usuarioSrv: UsuarioService, ...) { ... }: este é o construtor do componente UsuariosComponent, executado quando uma instância do componente é criada. Ele recebe três parâmetros injetados: usuarioSrv (uma instância de UsuarioService), router (uma instância de Router) e active (uma instância de ActivatedRoute).
- ngOnInit() { this.bind(); }: este é o método ngOnInit da classe UsuariosComponent, executado após a inicialização do componente. Ele chama o método bind() para carregar os usuários na tabela ao iniciar o componente.
- async bind() { ... }: este é o método bind da classe UsuariosComponent, responsável por carregar os usuários e atualizar a tabela. Ele utiliza o serviço UsuarioService para obter todos os usuários e, em seguida, cria uma nova instância de MatTableDataSource com os dados dos usuários para atualizar a tabela.
- async delete(usuario: UsuarioModel): Promise<void> { ... }: este é o método delete da classe UsuariosComponent, executado quando o usuário seleciona a opção de excluir um usuário na tabela. Recebe um parâmetro usuario (uma instância de UsuarioModel), que representa o usuário a ser excluído. O método utiliza o serviço UsuarioService para excluir o usuário, e depois chama o método bind() para atualizar a tabela após a exclusão. Em seguida, ele navega para a página `/ (a inicial).

TEMA 3 – REGRAS DE BOAS PRÁTICAS EM REST/CRUD/SOLID/ANGULAR

Vamos abordar as boas práticas para REST, CRUD, SOLID e Angular separadamente.

Boas Práticas em REST:

- Use verbos HTTP corretos para cada ação: GET (para ler dados), POST (para criar novos recursos), PUT/PATCH (para atualizar recursos existentes) e DELETE (para excluir recursos).
- Utilize substantivos plurais nas URLs para representar coleções de recursos (por exemplo, "/usuarios") e identificadores únicos para recursos individuais (por exemplo, "/usuarios/1").
- Use códigos de status HTTP apropriados para indicar o resultado da operação (por exemplo, 200 OK para sucesso, 201 Created para criação bem-sucedida, 404 Not Found para recurso não encontrado etc.).
- Faça uso de filtros, ordenação e paginação para lidar com grandes conjuntos de dados e melhorar a performance.
- Utilize versionamento na API para garantir a compatibilidade com versões anteriores ao realizar atualizações.

Boas práticas em CRUD (Create, Read, Update, Delete):

- Siga o princípio de responsabilidade única: cada método ou função deve ter uma tarefa específica.
- Mantenha o código limpo e organizado, evitando repetição de lógica.
- Utilize nomes de métodos que sejam claros e descritivos sobre o que fazem.
- Valide os dados de entrada antes de executar operações de criação e atualização.
- Ao implementar o Delete, utilize soft delete (marcar o recurso como excluído, em vez de removê-lo permanentemente), caso seja necessário manter histórico ou rastreamento.

Vejamos agora os **princípios SOLID**, acrônimo que representa cinco princípios importantes para o design de software orientado a objetos.

- Siga o guia de estilo oficial do Angular (Angular Style Guide) para padronizar a estrutura e nomenclatura do código.
- Separe o código em módulos, componentes, serviços e diretivas, para facilitar a manutenção e reutilização do código.
- Faça uso de lazy loading para carregar módulos sob demanda, melhorando o desempenho da aplicação.

- Utilize serviços para compartilhar lógica de negócio e dados entre componentes.
- Use o conceito de Observables e Reactive Programming para lidar com eventos assíncronos e comunicação entre componentes.
- Faça tratamento adequado de erros e exceções, para fornecer uma experiência de usuário melhor.
- Implemente testes unitários e de integração para garantir a qualidade do código.

3.1 REST (Representational State Transfer)

- Use verbos HTTP corretamente: utilize os verbos HTTP (GET, POST, PUT, DELETE) de acordo com as suas finalidades. GET para recuperar dados, POST para criar novos recursos, PUT para atualizar recursos existentes e DELETE para excluir recursos.
- Nomes de recursos significativos: utilize nomes de recursos que sejam descritivos e intuitivos, seguindo as convenções de nomenclatura.
- Utilize códigos de status apropriados: retorne códigos de status HTTP adequados para indicar o resultado da operação, como 200 para sucesso, 201 para criação bem-sucedida, 404 para recurso não encontrado etc.
- Versionamento de API: considere adicionar versionamento à sua API para lidar com mudanças futuras sem quebrar a compatibilidade com versões anteriores.
- Use autenticação e autorização: implemente mecanismos de autenticação e autorização para proteger suas APIs e garantir que apenas usuários autorizados acessem recursos.

3.2 CRUD (Create, Read, Update, Delete)

- Separação de responsabilidades: divida as operações de CRUD em camadas distintas, como a camada de persistência de dados (repositórios), serviços de negócio e controladores.
- Validação de entrada: realize validações adequadas nos dados de entrada para garantir que sejam consistentes e válidos antes de persisti-los.

- Utilize transações: se necessário, utilize transações para garantir a consistência dos dados em operações que envolvam múltiplas alterações.

3.3 SOLID (Princípios de Design Orientado a Objetos)

- Princípio da Responsabilidade Única (SRP): cada classe deve ter uma única responsabilidade.
- Princípio do Aberto/Fechado (OCP): as entidades de software devem estar abertas para extensão, mas fechadas para modificação.
- Princípio de Substituição de Liskov (LSP): as classes derivadas devem ser substituíveis por suas classes base.
- Princípio da Segregação de Interfaces (ISP): múltiplas interfaces específicas são melhores do que uma única interface genérica.
- Princípio da Inversão de Dependência (DIP): dependências devem ser baseadas em abstrações e não em implementações concretas.

- **S (Single Responsibility Principle – Princípio da Responsabilidade Única):** cada classe deve ter apenas uma responsabilidade, ou seja, ela deve ter apenas um motivo para ser alterada.
- **O (Open/Closed Principle – Princípio do Aberto/Fechado):** as classes devem ser abertas para extensão (por meio de herança, por exemplo) e fechadas para modificação. Isso promove um código mais flexível e evita modificações diretas em código existente.
- **L (Liskov Substitution Principle – Princípio da Substituição de Liskov):** uma instância de uma classe filha deve poder ser substituída por uma instância da classe mãe sem alterar a corretude do programa.
- **I (Interface Segregation Principle – Princípio da Segregação de Interfaces):** uma classe não deve ser forçada a depender de interfaces que não utiliza. É melhor ter várias interfaces específicas do que uma única interface genérica.
- **D (Dependency Inversion Principle – Princípio da Inversão de Dependência):** módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Além disso, abstrações não devem depender de detalhes; detalhes devem depender de abstrações.

3.4 Angular (Framework de Desenvolvimento Web)

- **Modularização**: organize o código em módulos, cada um com uma responsabilidade específica.
- **Components reutilizáveis:** crie componentes reutilizáveis para evitar duplicação de código e melhorar a manutenibilidade.
- **Injeção de dependência**: utilize a injeção de dependência para facilitar o teste e a modularidade do código.
- **Lazy Loading:** Utilize o Lazy Loading para carregar apenas os módulos necessários, melhorando o desempenho e a experiência do usuário.
- **Boa gestão de estado**: utilize técnicas de gerenciamento de estado, como o uso de serviços ou ferramentas como o NgRx, para manter a consistência do estado da aplicação.

3.5 Proteção de rotas em Angular

Em Angular, a proteção de rotas é uma prática importante para controlar o acesso a diferentes partes da aplicação com base nas permissões do usuário.

Existem várias abordagens para proteger rotas em Angular. Aqui estão algumas maneiras comuns de realizar isso:

Guards (Guardas de rota):

- **CanActivate:** este guard é usado para determinar se um usuário pode acessar uma rota específica. Ele verifica se determinadas condições são atendidas antes de permitir o acesso à rota.
- **CanActivateChild:** semelhante ao CanActivate, mas aplicado a rotas filhas aninhadas dentro de uma rota principal.
- **CanDeactivate:** permite verificar se um usuário pode sair de uma rota específica. É útil para exibir um aviso ou uma confirmação antes de sair de uma rota.
- **CanLoad:** usado para controlar o carregamento assíncrono de módulos. Verifica se um usuário tem permissão para carregar um módulo específico.
- **Role-based Authorization (autorização baseada em funções):** combine guards com informações de função ou permissão do usuário, para determinar se o acesso à rota deve ser concedido. Isso geralmente

envolve verificar se o usuário possui uma função específica ou permissões necessárias para acessar a rota.

- **Lazy loading de módulos:** ao usar o recurso de carregamento tardio (lazy loading) de módulos, você pode proteger rotas carregando módulos apenas quando um usuário tem permissão para acessá-los. Isso ajuda a reduzir o tamanho inicial da aplicação e permite um controle mais granular sobre o acesso às rotas.
- **Interceptors:** são usados para interceptar e modificar solicitações HTTP antes que elas sejam enviadas ao servidor. Você pode usar um interceptor para adicionar informações de autenticação, como tokens de acesso, a cada solicitação. Isso ajuda a garantir que apenas usuários autenticados possam acessar os recursos do servidor.

Essas são apenas algumas das técnicas comuns de proteção de rotas em Angular. A combinação dessas abordagens pode fornecer um sistema de controle de acesso eficiente e seguro em sua aplicação. Lembre-se de que a segurança também deve ser tratada no lado do servidor, para proteger os dados e as operações críticas.

TEMA 4 – TESTES UNITÁRIOS E AUTOMAÇÃO DE TESTES EM ANGULAR

Os testes unitários são uma prática de desenvolvimento de software que envolve a criação de testes automatizados para verificar se partes específicas do código, chamadas de *unidades*, funcionam corretamente de forma isolada. Essas unidades podem ser funções, métodos, classes ou até mesmo pequenos trechos de código.

O objetivo dos testes unitários é garantir que cada unidade de código individual funcione conforme o esperado, independentemente das outras partes do sistema. Isso ajuda a identificar e corrigir erros de forma rápida e eficiente, além de fornecer um nível de confiança e segurança ao realizar alterações ou melhorias no código existente.

- **Isolamento:** os testes unitários devem ser independentes uns dos outros e do ambiente externo. Isso significa que eles devem ser capazes de serem executados em qualquer ordem. Também não devem depender de recursos externos, como bancos de dados ou serviços web. Para alcançar

isso, é comum usar mocks (simulações) ou stubs (imitações) de dependências externas.

- **Cobertura abrangente:** os testes unitários devem cobrir várias situações e cenários possíveis para a unidade de código em questão. Isso inclui testar diferentes caminhos de execução, valores de entrada e resultados esperados. A cobertura abrangente ajuda a identificar erros e garantir um código mais robusto.
- **Automação:** os testes unitários devem ser automatizados, o que significa que eles podem ser executados repetidamente sem intervenção manual. Isso permite que eles sejam facilmente executados durante o desenvolvimento, com integração contínua ou em uma pipeline de entrega contínua. A automação dos testes ajuda a reduzir o tempo e os esforços necessários para validar a funcionalidade do código.
- **Frameworks e ferramentas:** existem diversos frameworks e ferramentas disponíveis para facilitar a criação e a execução de testes unitários em várias linguagens de programação. Alguns exemplos populares são o JUnit para Java, o pytest para Python e o NUnit para .NET. Essas ferramentas fornecem recursos para estruturar, organizar e executar os testes de forma eficiente.
- **Integração contínua:** os testes unitários são uma parte importante de uma estratégia de integração contínua, na qual o código é testado regularmente e de forma automatizada à medida que é desenvolvido. A integração contínua ajuda a identificar problemas de forma precoce, promovendo a entrega de software de maior qualidade e reduzindo os riscos de regressão.

Ao escrever testes unitários eficazes, é importante focar em casos de teste significativos, identificar os pontos críticos do código e garantir uma boa cobertura. Testes unitários bem elaborados proporcionam maior confiabilidade ao código, facilitam a manutenção e ajudam a criar uma base sólida para o desenvolvimento de software de alta qualidade.

TEMA 5 – SERVIÇOS DE NUVEM, LOJAS DE APLICATIVOS E DEPLOY

Os serviços de nuvem, também conhecidos como *computação em nuvem*, referem-se à entrega de recursos de computação, como servidores,

armazenamento, bancos de dados, redes, software e outros recursos, por meio da internet. Em vez de uma empresa possuir e manter fisicamente esses recursos, eles são fornecidos por provedores de serviços de nuvem, como Amazon Web Services (AWS), Microsoft Azure e Google Cloud Platform (GCP).

Os serviços de nuvem podem ser divididos em três modelos principais:

- **Infraestrutura como Serviço (IaaS):** fornece infraestrutura de TI, como máquinas virtuais, redes e armazenamento, permitindo que as empresas construam e gerenciem seus próprios ambientes virtuais.
- **Plataforma como Serviço (PaaS):** oferece uma plataforma completa para desenvolvimento, teste e implantação de aplicativos, sem a complexidade de gerenciar a infraestrutura subjacente.
- **Software como Serviço (SaaS):** disponibiliza aplicativos de software prontos para uso pela internet, geralmente acessados por meio de um navegador da web.

As lojas de aplicativos são plataformas online que permitem que os desenvolvedores de aplicativos publiquem e distribuam seus aplicativos para os usuários. Os usuários podem então baixar e instalar os aplicativos em seus dispositivos, como smartphones, tablets e computadores. Algumas das lojas de aplicativos mais populares são:

- **Google Play Store (Android):** loja de aplicativos oficial para dispositivos Android, gerenciada pelo Google.
- **Apple App Store (iOS):** loja de aplicativos oficial para dispositivos iOS, gerenciada pela Apple.
- **Microsoft Store (Windows):** loja de aplicativos para dispositivos com Windows 10, gerenciada pela Microsoft.
- **Mac App Store:** loja de aplicativos para Macs, gerenciada pela Apple.

5.1 Serviços em nuvem

Os serviços de nuvem são plataformas que fornecem recursos de computação, armazenamento, rede e outros serviços através da Internet. Eles permitem que as empresas e desenvolvedores hospedem, gerenciem e executem aplicativos e serviços sem a necessidade de infraestrutura física local. Alguns dos provedores de serviços de nuvem mais populares são: Amazon Web Services (AWS), Microsoft Azure e Google Cloud Platform (GCP). Esses serviços

oferecem uma ampla gama de recursos, incluindo hospedagem de aplicativos, bancos de dados, armazenamento de arquivos e serviços de mensagens.

5.2 Lojas de aplicativos

As lojas de aplicativos são plataformas online onde os usuários podem descobrir, baixar e instalar aplicativos em seus dispositivos móveis ou computadores. As lojas de aplicativos mais conhecidas são a App Store da Apple para dispositivos iOS, o Google Play Store para dispositivos Android e a Microsoft Store para dispositivos Windows. As lojas de aplicativos fornecem um ambiente seguro para os desenvolvedores distribuírem seus aplicativos, com recursos como avaliações de usuários, atualizações automáticas e processos de aprovação, para garantir a qualidade e a segurança dos aplicativos disponíveis.

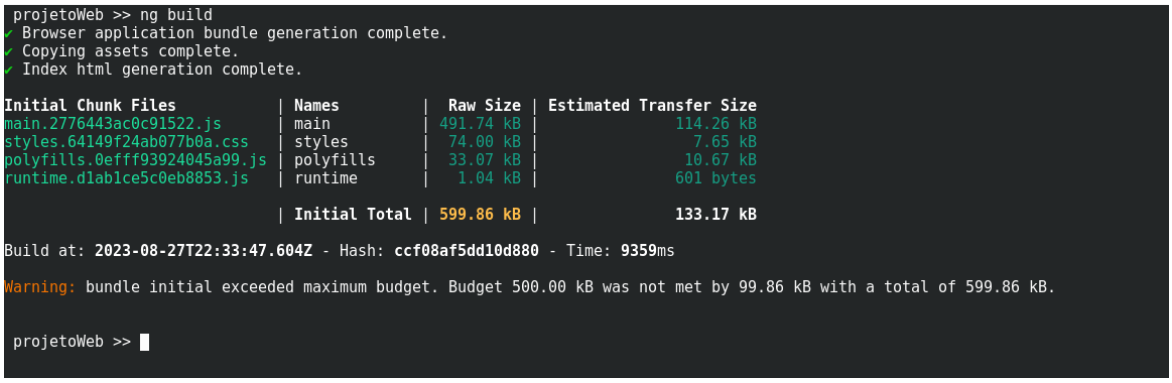
5.3 Deploy

O termo *deploy*, ou *implantação*, refere-se ao processo de disponibilizar um aplicativo ou serviço para uso em um ambiente de produção. Envolve a transferência de arquivos e configurações necessários para que o aplicativo possa ser executado em um servidor ou serviço de hospedagem. A implantação pode ser feita em diferentes ambientes, como servidores locais, servidores de nuvem ou plataformas de hospedagem gerenciadas. Existem várias formas de fazer o deploy de um aplicativo, incluindo uso de scripts de implantação, ferramentas de integração contínua/entrega contínua (CI/CD) e plataformas de orquestração de contêineres, como o Kubernetes. O objetivo é disponibilizar o aplicativo de forma eficiente, segura e escalável para que os usuários possam acessá-lo.

Primeiramente, temos de gerar o código na pasta dist. Para isso, fazemos um comando:

```
ng build
```

Ele deve gerar algum resultado, como o mostrado a seguir:



É preciso então acessar uma conta da Amazon, para uso do Amazon S3. Anote as credenciais de acesso (chave de acesso e chave secreta) associadas a essa conta. Essas informações serão usadas para autenticar o upload.

Instale a AWS CLI (Command Line Interface), seguindo as instruções no site oficial da AWS: <https://aws.amazon.com/cli/>.

Execute o comando aws. Configure no terminal e insira as credenciais de acesso (chave de acesso e chave secreta), além da região padrão.

No terminal, navegue até a pasta "dist" do seu projeto Angular, onde os arquivos de produção foram gerados. Use o seguinte comando para fazer o upload dos arquivos para o bucket do Amazon S3:

```
aws s3 sync . s3://nome-do-seu-bucket
```

Lembre-se de substituir "nome-do-seu-bucket" pelo nome real do seu bucket no comando acima. Após concluir essas etapas, os arquivos da pasta "dist" do seu aplicativo Angular estarão disponíveis no Amazon S3. Você poderá acessá-los por meio do URL do bucket ou ainda por meio de URLs específicas para os arquivos.

FINALIZANDO

Com esta etapa, fechamos o nosso estudo. É muito importante entender que todo ciclo de desenvolvimento depende do entendimento de arquitetura e tecnologia. Nossa construção de uma página web em Angular envolveu o uso da estrutura de desenvolvimento front-end do Angular, para criar uma aplicação web interativa e dinâmica. Usamos várias tecnologias Node para a estrutura básica do aplicativo.

A página foi desenvolvida através da criação de componentes reutilizáveis, serviços para lógica de negócios, roteamento para navegação entre páginas e manipulação de dados com formulários reativos. Utilizamos o

TypeScript para fornecer tipagem estática, o que traz vantagens para o desenvolvimento, como aproveitamento do sistema de módulos para modularizar o código. O ciclo de vida dos componentes permitiu controlar o comportamento durante as etapas de inicialização, atualização e destruição do componente.

Ao finalizar o desenvolvimento, a página web foi implantada em um servidor para que os usuários possam acessá-la pela internet. Tecnicamente, cumprimos nosso percurso, mas muito depende de interesse e aprofundamento individual.

Sucesso!