DESENVOLVIMENTO WEB – FRONT END

AULA 4

Prof. Mauricio Antonio Ferste

CONVERSA INICIAL

Vamos abordar 5 temas nesta etapa, que será baseada em vários padrões arquiteturais do Angular, como se segue:

- Injeção de dependências: a injeção de dependências é um padrão de design utilizado na programação orientada a objetos e no desenvolvimento de software. Envolve fornecer as dependências necessárias a um objeto ou componente de maneira externa, em vez de criá-las internamente. Isso ajuda a melhorar a modularidade, flexibilidade e testabilidade do código.
- Comunicação Publicador-Subscritor: a comunicação Publicador-Subscritor (editor-assinante) é um padrão de comunicação em que um componente chamado *publicador* emite eventos ou mensagens, enquanto outros componentes chamados *assinantes* se inscrevem para receber esses eventos. Isso permite uma comunicação assíncrona e desacoplada entre os componentes.
- Observables e Inscrição: Observable e Inscrição são conceitos relacionados à programação reativa. Um Observable é uma sequência de valores que podem ser observados. Os subscritores podem se inscrever para receber notificações quando novos valores são emitidos pelo Observable. A ação de se inscrever para receber essas notificações é chamada de *inscrição*.
- Formulários e formulários reativos: formulários e formulários reativos são tópicos relacionados ao desenvolvimento de interfaces de usuário interativas em aplicativos. Os formulários reativos são um método de construção de formulários em que os controlos do formulário são definidos como objetos em um modelo. Isso permite uma maior flexibilidade e recursos avançados de validação e manipulação de formulários em comparação com os formulários baseados em modelos.
- Gerenciamento e tratamento de erros: manipuladores: o gerenciamento e tratamento de erros são considerações importantes no desenvolvimento de software. Os manipuladores de erro são mecanismos ou blocos de código que lidam com erros quando eles ocorrem durante a execução do programa. Podem ser usados para capturar, registrar, notificar ou lidar

com erros de diferentes maneiras, garantindo a robustez e a estabilidade do aplicativo.

É importante estar preparado, pois agora começaremos o desenvolvimento propriamente dito. Ou seja, haverá trechos de código e explicações. Lembrem-se de que a prática é fundamental, e é essencial começar essa prática aqui.

TEMA 1 – INJEÇÃO DE DEPENDÊNCIAS

No Angular, os consumidores de dependência são as classes que precisam de uma determinada dependência para funcionar corretamente. Por exemplo, um componente pode depender de um serviço para buscar dados de um servidor. Essa dependência é solicitada por meio da injeção de dependência. Os provedores de dependência são as classes que fornecem as dependências solicitadas pelos consumidores. No exemplo mencionado anteriormente, o serviço seria o provedor de dependência, pois fornece os dados necessários ao componente.

O Angular utiliza o conceito de um Injetor para facilitar a interação entre consumidores e provedores de dependência. O Injetor é responsável por resolver as dependências solicitadas pelos consumidores. Ele verifica o registro de dependências para verificar se já existe uma instância disponível. Se não houver, o Injetor criará uma nova instância e a armazenará no registro.

Durante o processo de inicialização do aplicativo Angular, um injetor raiz é criado automaticamente. Esse injetor raiz é responsável por fornecer as dependências em todo o aplicativo. Além disso, é possível criar injetores adicionais conforme necessário em diferentes partes do aplicativo. Em resumo, a injeção de dependência no Angular permite que as classes consumidoras solicitem e recebam as dependências necessárias por meio do sistema de injeção fornecido pelo framework. Isso ajuda a promover a modularidade, a reutilização de código e a facilitar a manutenção dos aplicativos.

1.1 Fornecendo a dependência

Imagine que existe uma classe chamada *InternetService*, que estende uma classe BaseService e precisa atuar como dependência em um componente.

A primeira etapa é adicionar o decorador @Injectable para mostrar que a classe pode ser injetada.

Dica

A injeção de dependência é uma forma de inversão de controle – em vez de classes controlarem suas próprias dependências, elas trabalham com instâncias fornecidas pelo seu ambiente externo. Em termos mais concretos, injetar suas dependências simplifica alterar essas dependências no futuro.

O próximo passo é disponibilizá-lo, ou seja, fornecê-lo. Uma dependência pode ser fornecida em vários lugares: no nível do componente, usando o campo providers do decorador. Nesse caso, ele se torna disponível para todas as instâncias desse componente e de outros componentes e diretivas usados no modelo. Por exemplo:

```
import { InternetModel} from '../model/internetModel'
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 InternetService extends BaseService<InternetModel> {
  private loginSubject = new Subject<boolean>();
  constructor(public override http: HttpService) {
    super('internet', http);
  }
}
```

Ao registrar um provedor no nível do componente, você obtém uma nova instância do serviço com cada nova instância desse componente.

Nesse cenário, o serviço está disponível para todos os componentes, diretivas e pipes declarados neste NgModule ou em outro NgModule que esteja

dentro do mesmo ModuleInjector aplicável a este NgModule. Quando você registra um provedor com um NgModule específico, a mesma instância de um serviço fica disponível para todos os componentes, diretivas e canais aplicáveis. Por exemplo, temos abaixo o código de nosso projeto no arquivo app.module.ts. Esse código importa os módulos e, através da injeção de dependência com o @NgModule, disponibiliza para todo o sistema:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
import { MatToolbarModule} from '@angular/material/toolbar';
import { MatIcon, MatIconModule} from '@angular/material/icon';
import { MatListModule} from '@angular/material/list';
import { MatSidenavModule} from '@angular/material/sidenav';
import { UsuariosComponent } from
'./components/usuarios/usuarios.component';
import { QuestionarioComponent } from
'./components/questionario/questionario.component';
import { MatFormFieldModule} from '@angular/material/form-
field';
import { MatInputModule} from '@angular/material/input';
import { MatTableModule } from '@angular/material/table';
import { HttpClientModule } from '@angular/common/http';
import { QuestoesComponent } from
'./components/questoes/questoes.component';
import { UsuarioComponent } from
'./components/usuario/usuario.component';
import { FormsModule } from '@angular/forms';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
    UsuariosComponent,
    QuestionarioComponent,
    QuestoesComponent,
    UsuarioComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    MatToolbarModule,
    MatIconModule,
    MatListModule,
```

```
    MatSidenavModule,
    MatFormFieldModule,
    MatInputModule,
    MatTableModule,
    HttpClientModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Em Angular, a hierarquia de injetores refere-se à maneira como os serviços são fornecidos e compartilhados entre os componentes e módulos da aplicação. Angular utiliza o mecanismo de injeção de dependências para fornecer instâncias de serviços aos componentes que precisam deles. Quando um serviço é injetado em um componente, o Angular verifica o injetor para encontrar uma instância do serviço. O injetor é uma parte fundamental do sistema de injeção de dependências do Angular e é responsável por criar e fornecer as instâncias de serviço. A hierarquia de injetores segue uma ordem descendente. Isso significa que, quando um componente solicita um serviço, o Angular procura primeiro em seu próprio injetor local. Se o serviço não estiver disponível nesse injetor, o Angular continuará procurando no injetor do componente pai e assim por diante, subindo na árvore de componentes até encontrar um injetor que forneça o serviço ou alcançar o injetor raiz da aplicação (Angular, 2023).

Essa hierarquia em injetores permite que os serviços sejam compartilhados ou isolados em diferentes partes da aplicação, conforme necessário. Por exemplo, você pode ter um serviço que deve ser compartilhado por todos os componentes em um módulo específico e, ao mesmo tempo, ter outro serviço separado que é exclusivo de um componente específico. Em termos de Angular, temos:

- No início do arquivo, você importa os módulos e componentes necessários para a sua aplicação. Além dos módulos principais do Angular, como NgModule e BrowserModule, você também importa outros módulos específicos, como AppRoutingModule, BrowserAnimationsModule, MatToolbarModule, MatIconModule, MatListModule, MatSidenavModule, MatFormFieldModule,

MatInputModule, MatTableModule, HttpClientModule, FormsModule e ReactiveFormsModule. Esses módulos são utilizados para fornecer recursos como rotas, animações, estilos de material design, formulários, tabelas, acesso a serviços HTTP, entre outros.

- Os imports: nessa seção, você importa os módulos que serão utilizados por este módulo. Esses módulos são adicionados ao módulo raiz para que os componentes e serviços possam utilizá-los.
- Os providers: nessa seção, você pode fornecer serviços que ficarão disponíveis para toda a aplicação. No exemplo, está vazio, mas você pode adicionar serviços aqui para serem injetados em diferentes partes da aplicação.
- O bootstrap: nessa seção, você define o componente que será o componente raiz da aplicação. No exemplo, o componente raiz é o AppComponent, que será o componente principal da aplicação.

Com o AppModule definido dessa forma, ele está pronto para ser usado como o módulo raiz da sua aplicação Angular.

Existem vários exemplos interessantes como o existente em Angular (2023), em <https://angular.io/guide/dependency-injection>.

No nível raiz do aplicativo, o que permite injetá-lo em outras classes do aplicativo. Isso pode ser feito adicionando o providedIn: 'root'campo ao decorador:

```
No nível raiz do aplicativo, o que permite injetá-lo em outras
classes do aplicativo. Isso pode ser feito adicionando o
providedIn: 'root'campo ao decorador:

import { InternetModel} from '../model/internetModel'
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 InternetService extends BaseService<InternetModel>
{
  private loginSubject = new Subject<boolean>();
  constructor(public override http: HttpService) {
    super('internet', http);
```

```
  }
}
```

Quando você fornece o serviço no nível raiz, o Angular cria uma única instância compartilhada do serviço por toda a aplicação, usando a anotação @Injectable({ providedIn: 'root' }): A anotação @Injectable com a configuração providedIn: 'root' torna o serviço InternetService disponível como um singleton. Isso significa que haverá apenas uma instância compartilhada de InternetService em toda a aplicação.

1.2 Injetando a dependência

A maneira mais comum de injetar uma dependência é declará-la em um construtor de classe. Quando o Angular cria uma nova instância de um componente, diretiva ou classe de canal, ele determina quais serviços ou outras dependências essa classe precisa observando os tipos de parâmetro do construtor.

Quando o Angular descobre que um componente depende de um serviço, ele primeiro verifica se o injetor possui alguma instância existente desse serviço. Se uma instância do serviço solicitado ainda não existir, o injetor cria uma usando o provedor registrado e a adiciona ao injetor antes de retornar o serviço ao Angular.

Quando todos os serviços solicitados forem resolvidos e retornados, o Angular pode chamar o construtor do componente com esses serviços como argumentos.

1.3 Uso natural de serviços injetáveis

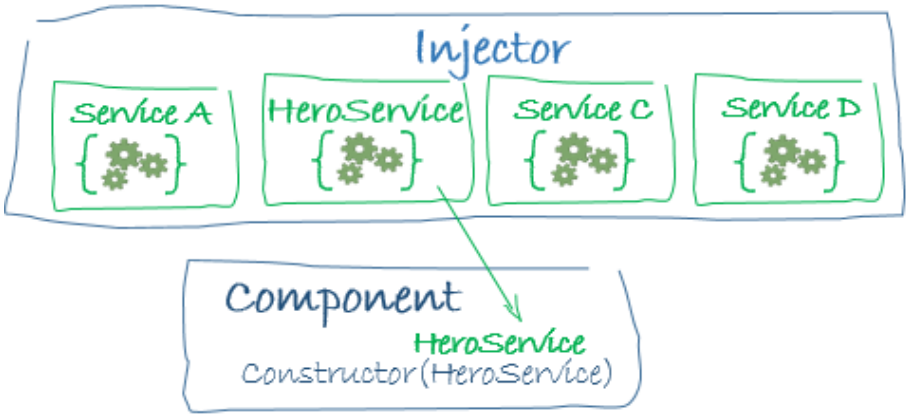
No Angular, os serviços são uma categoria ampla que abrange qualquer valor, função ou recurso de que um aplicativo precisa. Eles são geralmente implementados como classes com um propósito específico e bem definido.

Os componentes, por outro lado, são tipos de classe no Angular que têm a responsabilidade de lidar com a apresentação e a interação do usuário. Os componentes são responsáveis por fornecer propriedades e métodos para a ligação de dados e atuam como mediadores entre a visão (renderizada pelo modelo) e a lógica do aplicativo (geralmente referente ao modelo).

Ao separar a lógica de apresentação dos componentes em serviços, é possível alcançar maior modularidade e reutilização de código. Os serviços podem ser usados para executar tarefas específicas, como buscar dados do servidor, validar entrada do usuário ou autenticar. Essas tarefas de processamento são definidas em classes de serviço injetáveis, permitindo que sejam compartilhadas entre diferentes componentes.

Através do mecanismo de injeção de dependência (DI), o Angular facilita a disponibilização desses serviços aos componentes. Isso permite que diferentes provedores de serviços sejam injetados em diferentes circunstâncias, tornando o aplicativo mais adaptável.

Embora o Angular não imponha esses princípios, ele fornece recursos e ferramentas que facilitam a adoção dessas melhores práticas de separação de responsabilidades entre serviços e componentes. Isso promove a modularidade, a reutilização de código e a manutenibilidade do aplicativo.



Fonte: Angular, 2023.

1.4 Criando um serviço injetável

Para criar um novo serviço injetável no Angular usando o Angular CLI, você pode seguir os seguintes passos:

1. Abra um terminal ou prompt de comando.
2. Navegue até o diretório raiz do seu aplicativo Angular usando o comando cd (por exemplo, cd my-app).
3. Execute o seguinte comando para gerar um novo serviço:

```
ng generate service usuario
```

4. Isso criará uma classe chamada *usuario.service.ts* no diretório src/app. O nome *usuario* é um exemplo e você pode substituí-lo pelo nome do seu serviço.
5. O serviço recém-criado terá um decorator @Injectable() importado do pacote @angular/core, garantindo que ele possa ser injetado em outros componentes ou serviços.
6. Você pode implementar a lógica do serviço no arquivo usuario.service.ts. Por exemplo, você pode adicionar métodos para buscar dados de um servidor ou executar outras tarefas específicas do serviço.

O Angular CLI também atualizará automaticamente o arquivo app.module.ts para adicionar o serviço ao array providers, tornando-o disponível para injeção de dependência em outros componentes ou serviços.

Após criar o serviço, você pode injetá-lo em um componente usando a injeção de dependência padrão do Angular. Por exemplo, no arquivo app.component.ts, você pode importar o serviço e injetá-lo no construtor do componente.

Esses são os passos básicos para criar um novo serviço injetável no Angular usando o Angular CLI. A partir daí, você pode personalizar o serviço de acordo com as necessidades específicas do seu aplicativo.

TEMA 2 – COMUNICAÇÃO PUBLISHER-SUBSCRIBER

A comunicação Publisher-Subscriber, também conhecida como *pub-sub*, é um padrão de comunicação utilizado em sistemas distribuídos e arquiteturas de software. Nesse padrão, existem dois papéis principais: o Publisher (publicador) e o Subscriber (assinante). O Publisher é responsável por enviar mensagens ou eventos para um ou mais Subscribers. Ele não tem conhecimento direto sobre os Subscribers, apenas publica as mensagens em um canal centralizado ou tópico.

O Subscriber, por sua vez, se registra para receber mensagens específicas do Publisher. Ele se torna um observador desse canal ou tópico e será notificado sempre que uma mensagem relevante for publicada. O padrão pub-sub oferece várias vantagens. Em primeiro lugar, permite a comunicação assíncrona entre componentes e sistemas distribuídos, o que pode melhorar a escalabilidade e o desempenho. Além disso, o Publisher não precisa conhecer

ou se preocupar com os detalhes de implementação dos Subscribers, tornando o acoplamento mais fraco e promovendo a modularidade e o reuso de componentes.

Na prática, a implementação do padrão pub-sub pode variar dependendo da tecnologia ou do framework utilizado. No contexto do Angular, o Angular utiliza o padrão pub-sub em conjunto com a biblioteca RxJS para facilitar a comunicação assíncrona e o gerenciamento de fluxos de dados observáveis. No Angular, é comum usar o padrão pub-sub para implementar a comunicação entre componentes ou para transmitir eventos em um aplicativo. O serviço EventEmitter e a funcionalidade de Subject do RxJS são frequentemente utilizados para facilitar a comunicação pub-sub no Angular.

Em resumo, o padrão de comunicação Publisher-Subscriber, ou pub-sub, é um padrão que permite a comunicação assíncrona e desacoplada entre componentes ou sistemas distribuídos. No contexto do Angular, esse padrão é comumente utilizado para facilitar a comunicação entre componentes e transmitir eventos em um aplicativo.

2.1 EventEmitter no Javascript

O EventEmitter é um conceito comumente encontrado na programação orientada a eventos e é frequentemente usado em várias linguagens de programação e estruturas. Ele permite a implementação do padrão de design Observer, em que objetos (conhecidos como *ouvintes* ou *subscritores*) podem se registrar para receber notificações (conhecidas como *eventos*) quando uma determinada ação ou condição ocorre.

Em JavaScript, o EventEmitter é uma classe integrada fornecida pelo ambiente de execução do Node.js, que permite criar e gerenciar eventos personalizados. Também é comumente usado em várias estruturas e bibliotecas JavaScript, como EventEmitter2 ou o módulo EventEmitter do sistema de eventos baseado em navegador.

```
const EventEmitter = require('events');

// Criar uma nova instância do EventEmitter
const meuEmitter = new EventEmitter();
```

```
// Registrar um ouvinte de evento
meuEmitter.on('meuEvento', (arg) => {
  console.log('Evento ocorreu com argumento:', arg);
});

// Emitir o evento
meuEmitter.emit('meuEvento', 'Olá, mundo!');
```

Neste exemplo, criamos uma nova instância da classe EventEmitter chamada *meuEmitter*. Em seguida, registramos um ouvinte de evento para o evento chamado *'meuEvento'*. Quando o evento é emitido usando meuEmitter.emit('meuEvento', 'Olá, mundo!'), a função do ouvinte registrada é executada, imprimindo 'Evento ocorreu com argumento: Olá, mundo!' no console. É possível registrar vários ouvintes para o mesmo evento, e todos serão invocados quando o evento for emitido. O EventEmitter também fornece outros métodos para gerenciar eventos, como once (para ouvir um evento apenas uma vez), removeListener (para remover um ouvinte específico) e removeAllListeners (para remover todos os ouvintes de um determinado evento). Observe que o EventEmitter é específico do Node.js e pode não estar disponível em outros ambientes JavaScript, como navegadores da web.

Sugestão de leitura

Uma boa ideia de sugestão para ler e treinar seria utilizar o tutorial do próprio Node.js. Segue: <https://nodejs.dev/pt/learn/the-nodejs-event-emitter>. Ou pode também ser usado o exemplo on-line do W3Schools: <https://www.w3schools.com/nodejs/nodejs_events.asp>.

2.2 EventEmitter no Angular

No Angular, o conceito de *EventEmitter* também está presente, embora com algumas variações em relação à implementação padrão do Node.js. No Angular, o EventEmitter é uma classe específica do framework que permite a comunicação entre componentes por meio de eventos personalizados.

Aqui está um exemplo de como usar o EventEmitter no Angular:

1. Importe o módulo do EventEmitter:

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

2. Crie uma instância de EventEmitter dentro do componente:

```
export class MeuComponente {
  meuEvento = new EventEmitter<string>();
}
```

Neste exemplo, estamos criando uma instância de EventEmitter chamada *meuEvento*, que emitirá eventos contendo valores do tipo string.

```
this.meuEvento.emit('Olá, mundo!');
```

3. Registre um ouvinte para o evento no componente receptor:

```
export class OutroComponente {
  constructor() {
    meuComponente.meuEvento.subscribe((valor: string) => {
      console.log('Evento ocorreu com argumento:', valor);
    });
  }
}
```

Neste exemplo, estamos registrando um ouvinte para o evento meuEvento no construtor do componente OutroComponente. Quando o evento é emitido a partir de um componente que possui o meuEvento, o ouvinte é acionado e o valor emitido é capturado, imprimindo 'Evento ocorreu com argumento: Olá, mundo!' no console.

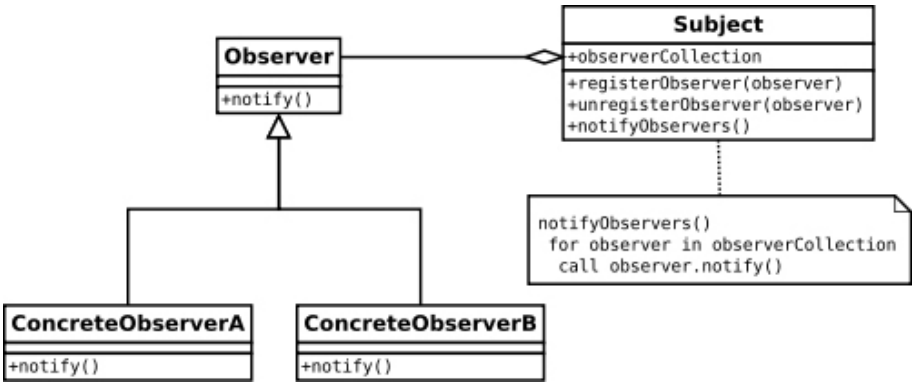
Em tempo, EventEmmiter é uma classe, ou seja, tem construtor, métodos e tudo mais. Ele é específico para a comunicação entre componentes dentro do framework e não é diretamente relacionado ao EventEmitter do Node.js. Ele é usado para criar e emitir eventos personalizados dentro de um aplicativo Angular, permitindo a interação e comunicação entre componentes.

Além disso, o EventEmitter no Angular também fornece métodos adicionais, como subscribe (para se inscrever em um evento), unsubscribe (para cancelar a inscrição em um evento) e outros métodos para gerenciar a emissão e manipulação de eventos, como visto em Angular (2023), em <https://angular.io/api/core/EventEmitter>.

TEMA 3 – OBSERVABLES E SUBSCRIBE

O padrão Observer é um padrão de projeto de software que permite que um objeto, conhecido como *subject* (sujeito), mantenha uma lista de seus dependentes, chamados de *observers* (observadores), e os notifique automaticamente de eventuais mudanças de estado. Essa notificação é geralmente realizada através da chamada de métodos específicos dos observadores. O padrão Observer é amplamente utilizado para implementar sistemas de tratamento de eventos distribuídos. Esse padrão desempenha um papel fundamental na arquitetura Model View Controller (MVC), sendo uma peça-chave nesse contexto. Na verdade, o padrão Observer foi implementado pela primeira vez no framework MVC de interface de usuário Smalltalk. Ele oferece uma maneira flexível e desacoplada de atualizar e manter a consistência entre as diferentes partes de um sistema. O padrão Observer é amplamente adotado e implementado em várias bibliotecas de programação e sistemas, incluindo a maioria das ferramentas de interface gráfica do usuário (GUI). Ele fornece uma abordagem eficiente para lidar com eventos e atualizações de estado, permitindo que os observadores sejam notificados de forma automática e reajam adequadamente às mudanças ocorridas no subject (Gamma et al., 2000).

Figura 2 – Uma ideia genérica do que é um padrão observer



Fonte: Gamma et al., 2000.

3.1 O padrão Observer

O padrão Observer é um dos padrões de projeto mais populares e amplamente utilizados no desenvolvimento de software. Ele pertence à categoria

dos padrões comportamentais, que se concentram em definir como objetos interagem e se comunicam entre si. O objetivo principal do Observer é estabelecer uma comunicação eficiente entre objetos, permitindo que eles sejam notificados sobre mudanças de estado ou eventos relevantes. Baseado em um relacionamento de dependência invertida, em que um objeto chamado de *subject* (sujeito) mantém uma lista de objetos chamados de *observers* (observadores). Esses observers estão interessados em acompanhar as mudanças no subject e se registram para receber notificações quando ocorrem alterações relevantes.

O subject é responsável por notificar os observers sobre as mudanças. Ele fornece métodos para adicionar, remover e notificar observers. Quando uma mudança ocorre, o subject percorre a lista de observers registrados e chama um método de atualização em cada um deles. Essa notificação permite que os observers reajam às mudanças e realizem as ações necessárias.

Os observers, por sua vez, implementam uma interface ou classe abstrata que define o contrato para receber notificações. Geralmente, eles possuem um método de atualização que é invocado pelo subject quando ocorre uma mudança relevante. Os observers podem acessar o estado atual do subject e tomar decisões com base nessa informação. Isso promove um baixo acoplamento entre os objetos, pois o subject não precisa conhecer os detalhes específicos dos observers, apenas a interface que eles implementam. Isso permite que o sistema seja mais flexível e fácil de manter, pois é possível adicionar ou remover observers sem modificar o subject.

Além disso, o padrão Observer ajuda a estabelecer um design modular e extensível. Novos observers podem ser adicionados sem afetar o código existente, o que facilita a adição de novos recursos ou funcionalidades ao sistema. Também permite que diferentes partes do sistema reajam de forma independente às mudanças, melhorando a escalabilidade e a flexibilidade do software.

Como já falamos, um exemplo comum de aplicação do padrão Observer é em sistemas de GUI (Interface Gráfica do Usuário), onde componentes gráficos como botões, menus ou caixas de seleção notificam outros componentes quando ocorrem eventos, como cliques ou mudanças de seleção. Os observers podem ser responsáveis por atualizar a interface, executar ações específicas ou processar os dados fornecidos pelos eventos.

Em resumo, o padrão Observer é uma poderosa ferramenta para estabelecer uma comunicação eficiente entre objetos em um sistema de software. Ele permite que objetos sejam notificados sobre mudanças relevantes, promove um baixo acoplamento e facilita a extensibilidade do sistema. Ao aplicar o padrão Observer de forma adequada, é possível construir sistemas mais flexíveis, modulares e fáceis de manter.

3.2 Observables e Subscribe no Angular

No Angular, os Observables e o método subscribe() são parte integrante do módulo RxJS (Reactive Extensions for JavaScript) e são usados para trabalhar com fluxos de dados assíncronos. Os Observables são objetos que representam uma sequência de valores que podem ser observados ao longo do tempo. O método subscribe() é usado para se inscrever em um Observable e receber notificações dos valores emitidos por ele.

O padrão Observables e Subscribe é um dos principais conceitos no Angular e em programação reativa. Ele é usado para tratar fluxos de eventos assíncronos, como solicitações HTTP, interações do usuário, atualizações de dados em tempo real, entre outros.

O padrão Observables é baseado no princípio de que uma fonte de dados emite eventos ao longo do tempo, e esses eventos podem ser observados e reagidos por meio de um objeto chamado *Observable*. O Observable representa o fluxo de dados e fornece métodos para se inscrever e receber notificações sempre que ocorrerem novos eventos.

Aqui estão os principais conceitos relacionados ao padrão Observables e Subscribe no Angular:

- Observable: um Observable é uma representação de um fluxo de eventos assíncronos. Ele pode emitir um ou vários valores ao longo do tempo. No Angular, os Observables são fornecidos pela biblioteca RxJS, que estende as capacidades do padrão Observables do JavaScript.
- Subscription: uma Subscription é o objeto retornado quando você se inscreve em um Observable. Ela representa a conexão entre o Observable e o código que irá receber e reagir aos eventos emitidos pelo Observable. A Subscription tem um método unsubscribe() que deve ser

chamado para cancelar a inscrição e liberar recursos quando não for mais necessário.

- Operadores: os operadores são métodos fornecidos pela biblioteca RxJS que permitem transformar, combinar, filtrar e manipular os fluxos de dados dos Observables. Eles permitem a criação de cadeias de operadores para manipular os eventos do Observable de acordo com as necessidades específicas.
- Observer: um Observer é um objeto que define como reagir aos eventos emitidos por um Observable. Ele possui três métodos principais: next(), que é chamado sempre que um novo valor é emitido pelo Observable; error(), que é chamado quando ocorre um erro no Observable; e complete(), que é chamado quando o Observable é concluído e não emitirá mais valores.

Aqui está um exemplo básico de como usar Observables e o método subscribe() no Angular:

1. Importe os módulos necessários:

```
import { Observable } from 'rxjs';
```

2. Crie um observable:

```
const meuObservable = new Observable((observer) => {
  observer.next('Primeiro valor');
  observer.next('Segundo valor');
  observer.next('Terceiro valor');
  observer.complete();
});
```

3. Subscreva o observable usando o método subscribe:

```
meuObservable.subscribe(
  (valor) => {
    console.log('Valor emitido:', valor);
  },
  (erro) => {
    console.error('Erro:', erro);
  },
```

```
  () => {
    console.log('Observable completo.');
  }
);
```

4. Neste exemplo, estamos chamando o método subscribe() no meuObservable. Passamos três funções de retorno para o método subscribe(): a primeira função é chamada quando um novo valor é emitido (next), a segunda função é chamada em caso de erro (error), e a terceira função é chamada quando o Observable é concluído (complete).

No exemplo acima, a função de retorno do next imprime cada valor emitido no console. A função de retorno do error é chamada caso ocorra algum erro durante a emissão dos valores. A função de retorno do complete é chamada quando o Observable é concluído.

Observables e o método subscribe() são amplamente utilizados no Angular para lidar com chamadas de API assíncronas, eventos de formulário, atualizações em tempo real, entre outros cenários em que a resposta ou os dados chegam de forma assíncrona. Eles fornecem uma maneira poderosa de trabalhar com fluxos de dados e reagir a eles de maneira eficiente e concisa.

De fato, existem várias diferenças significativas entre o AngularJS e o Angular 2+ (também conhecido como *Angular 2*, *Angular 4*, *Angular 5* etc.). Aqui estão algumas das diferenças mais notáveis:

- Linguagem de programação: o AngularJS é baseado em JavaScript, enquanto o Angular 2+ é baseado em TypeScript, que é um superset do JavaScript. O TypeScript adiciona recursos de tipagem estática, suporte a classes, interfaces e outros recursos avançados à linguagem.
- Arquitetura e Componentização: o Angular 2+ foi redesenhado para seguir uma arquitetura baseada em componentes. Em contraste, o AngularJS utiliza uma abordagem baseada em controladores e escopos ($scope). No Angular 2+, não há mais a necessidade de criar controladores e trabalhar com $scope, pois a lógica é encapsulada em componentes reutilizáveis.
- Orientação a dispositivos móveis: o Angular 2+ é projetado com foco no suporte a dispositivos móveis desde o início. Ele oferece recursos como

detecção de toque (touch), suporte a gestos e aprimoramentos de desempenho específicos para dispositivos móveis.

- Sintaxe e padrões: a sintaxe do Angular 2+ é bastante diferente do AngularJS. Alguns exemplos de diferenças incluem: ngFor se tornou *ngFor, camelCase é o padrão de nomeação para diretivas, e várias outras alterações foram feitas para melhorar a clareza e a consistência da sintaxe.
- Angular CLI: o Angular 2+ possui uma interface de linha de comando (CLI) poderosa e oficialmente suportada, que facilita a criação de novos projetos, componentes, serviços e muito mais. O CLI automatiza tarefas comuns de desenvolvimento e oferece um fluxo de trabalho eficiente.
- Serviços: no AngularJS, havia várias maneiras de definir um serviço, como factory, serviço, provider, constant e values. No Angular 2+, a definição de serviços é baseada em classes, em que um serviço é uma classe decorada com o @Injectable. Essa mudança simplifica e unifica a definição de serviços no Angular 2+.
- Desempenho e tamanho do pacote: o Angular 2+ introduziu melhorias significativas em termos de desempenho e tamanho do pacote em comparação com o AngularJS. O Angular 2+ utiliza um mecanismo de detecção de alterações mais eficiente, o que resulta em uma melhor performance. Além disso, o Angular 2+ possui um sistema de compilação e empacotamento (como o Angular CLI) que permite criar bundles otimizados e reduzir o tamanho final da aplicação.

Essas são apenas algumas das diferenças notáveis entre o AngularJS e o Angular 2+. O Angular 2+ trouxe várias melhorias e recursos avançados, tornando-o uma opção mais moderna e poderosa para o desenvolvimento de aplicativos da web em comparação com o AngularJS.

TEMA 4 – FORMULÁRIOS E REACTIVE FORMS

Em desenvolvimento de software, um formulário refere-se a uma interface de usuário que permite aos usuários inserir e enviar dados para um sistema. É uma maneira comum de coletar informações dos usuários, como detalhes de contato, preferências, configurações e muito mais. Os formulários geralmente são compostos por vários campos ou elementos, como caixas de texto, menus

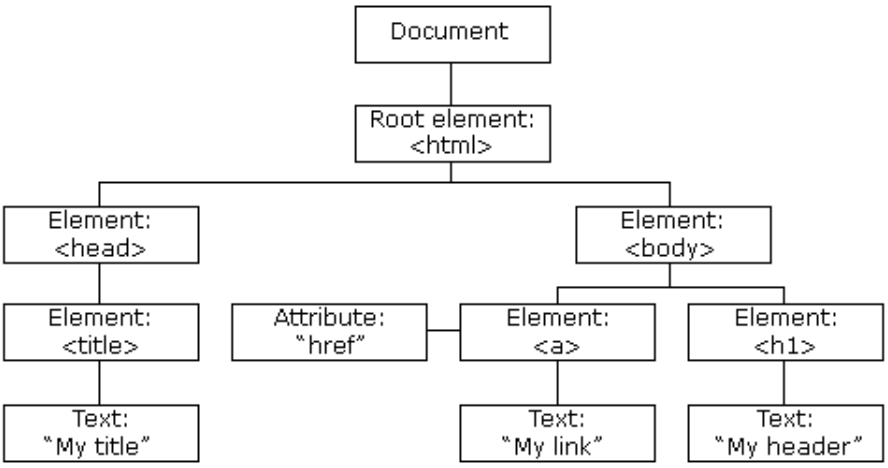
suspensos, botões de rádio, caixas de seleção e botões de envio. Os usuários preenchem esses campos com os dados solicitados e, em seguida, podem enviar o formulário, enviando os dados para um servidor ou processando-os localmente.

No desenvolvimento web, os formulários são criados usando linguagens de marcação como HTML e estilizados com CSS. Ao enviar um formulário, é comum usar linguagens de programação do lado do servidor, como PHP, Java, Python ou JavaScript, para processar os dados recebidos e executar as ações necessárias. Os formulários são uma parte essencial de muitos aplicativos e sites, pois fornecem uma maneira interativa de coletar informações dos usuários e permitem a interação entre o usuário e o sistema.

Em JavaScript, você pode trabalhar com formulários HTML e manipulá-los dinamicamente usando o DOM (Document Object Model). Existem várias formas de interagir com formulários em JavaScript, permitindo que você acesse e modifique os valores dos campos, valide os dados inseridos pelos usuários e responda a eventos relacionados aos formulários.

De acordo com o padrão W3C HTML DOM, tudo em um documento HTML é um nó:

- O documento inteiro é um nó de documento.
- Cada elemento HTML é um nó de elemento.
- O texto dentro dos elementos HTML são nós de texto.
- Cada atributo HTML é um nó de atributo (obsoleto).
- Todos os comentários são nós de comentário.



Fonte: W3C.

Importante

Utilizaremos a partir de agora em alguns pontos partes de código que disponibilizaremos nas gravações também. Entretanto, nenhum tipo de código deve ser considerado absoluto. Todo código tem variações e naturalmente muda com o tempo!

4.1 Formulários em Angular

Os formulários em Angular podem ser construídos usando duas abordagens: Reactive Forms (Formulários reativos) e Template-driven Forms (Formulários baseados em template). Ambas as abordagens têm seus próprios casos de uso e vantagens, e a escolha entre elas depende das necessidades do seu aplicativo.

- Reactive Forms (Formulários reativos)

Os Reactive Forms são uma abordagem baseada em código para a criação e manipulação de formulários no Angular. Eles são construídos usando classes TypeScript e fornecem uma maneira programática de gerenciar o estado e o comportamento do formulário. Com Reactive Forms, você cria uma representação do formulário no código, define validadores personalizados e utiliza Observables para rastrear e reagir a alterações nos valores dos campos. Com o uso desta alternativa, existe o controle total sobre a lógica do formulário no código: Com Reactive Forms, você tem controle total sobre a lógica do formulário no código TypeScript. Você cria uma representação do formulário como uma instância de classe usando o FormBuilder e tem acesso direto aos

campos, validadores, estado e comportamento do formulário. Isso permite maior flexibilidade e personalização na manipulação do formulário.

Validação avançada com validadores personalizados: Os Reactive Forms permitem que você crie validadores personalizados para realizar validações avançadas nos campos do formulário. Você pode criar validadores síncronos e assíncronos para verificar a validade dos dados inseridos. Isso inclui validações como verificação de formato de e-mail, validação de senha, verificação de campos obrigatórios e muito mais.

Como os Reactive Forms usam o padrão de projeto Observables para rastrear as alterações nos valores dos campos e no estado do formulário, cada campo do formulário é um Observable que você pode assinar para receber notificações sempre que houver alterações no valor. Isso permite reagir a essas alterações e atualizar a interface do usuário de acordo.

Dica: teste unitário

Basicamente, testes unitários ou testes de unidades aferem a qualidade do código em sua menor fração. Eles também podem ser identificados como testes de algoritmos e, por isso, não dependem do uso de recursos externos. Sendo assim, são os testes mais baratos da pirâmide de testes, como você verá a seguir. No entanto, geram menos valor para a empresa, visto que sozinhos eles não garantem a corretude do código.

Mas, em síntese, os testes unitários são os mais usados em qualquer projeto e, por isso, quem está começando a dar os primeiros passos na programação deve ficar atualizado sobre este tema, aprofundando os seus estudos.

A abordagem de Reactive Forms oferece uma melhor testabilidade em comparação com os Template-driven Forms. Como a lógica do formulário está no código TypeScript, você pode escrever testes unitários para verificar o comportamento do formulário, testar a validação dos campos, simular eventos de entrada e verificar as respostas do formulário. Isso permite uma cobertura de teste mais completa e garante a qualidade do código do formulário.

- Template-driven Forms (Formulários baseados em template)

Os Template-driven Forms são uma abordagem mais simples para criar formulários no Angular. Eles são construídos principalmente no template HTML e usam diretivas do Angular para lidar com a lógica do formulário. Com Template-driven Forms, o Angular infere automaticamente o modelo de dados do formulário com base nas diretivas usadas, e você pode adicionar validação usando atributos HTML padrão. Principais características dos Template-driven Forms:

Fácil configuração e menor quantidade de código: Os Template-driven Forms são mais fáceis de configurar, pois a lógica do formulário é tratada principalmente no template HTML. Você pode usar diretivas do Angular, como ngModel e ngForm, juntamente com atributos HTML padrão, para criar formulários de maneira rápida e fácil. Isso resulta em menos código necessário em comparação com os Reactive Forms.

Para fazer a validação básica usando atributos HTML: Nos Template-driven Forms, a validação básica pode ser adicionada aos campos do formulário usando atributos HTML padrão, como required, minlength, maxlength, pattern, entre outros. Essa validação é feita pelo navegador e pelo Angular automaticamente, garantindo que os dados inseridos pelos usuários atendam aos critérios básicos de validação.

Lógica do formulário tratada principalmente no template HTML: Com os Template-driven Forms, a lógica do formulário é tratada principalmente no template HTML. Você pode usar diretivas do Angular, como ngModel, ngForm e ngSubmit, para definir a estrutura do formulário e controlar o comportamento de envio. A manipulação de eventos, como cliques e envios de formulário, é feita diretamente no template HTML, tornando o código mais declarativo.

Menos flexibilidade em relação à lógica e validação personalizadas: Os Template-driven Forms têm menos flexibilidade em comparação com os Reactive Forms quando se trata de lógica e validação personalizadas. Embora você possa adicionar validações básicas usando atributos HTML, a criação de validações personalizadas e lógica mais complexa requer o uso de diretivas personalizadas ou funções auxiliares adicionais. Isso pode limitar um pouco a capacidade de personalizar a lógica do formulário.

Ambas as abordagens são suportadas pelo Angular e podem ser usadas de acordo com a complexidade do formulário e os requisitos do seu aplicativo. É

importante entender as diferenças entre elas e escolher a abordagem mais adequada para cada situação. É possível combinar ambas as abordagens em um único aplicativo, usando a abordagem que melhor se adequa a cada formulário específico. Isso permite uma flexibilidade maior e o uso das vantagens de cada abordagem quando necessário.

Abaixo segue o código de um app.module, mostrando as importações que ocorrerão, entre elas as de formulário, com componentes também:

AppModule em um aplicativo Angular. Esse módulo é responsável por importar e configurar os diversos módulos e componentes necessários para o funcionamento do aplicativo.

Aqui está uma explicação passo a passo do código fornecido:

- Importações:
  - NgModule, BrowserModule: Importações dAngular para definir e configurar módulo.
  - AppRoutingModule: Importaçãdmódulde roteamentdaplicativo.
  - AppComponent: Importaçãdcomponente principal daplicativo.
  - BrowserAnimationsModule: Importaçãdmódulde animações dnavegador.
  - MatToolbarModule, MatIconModule, MatListModule, MatSidenavModule, MatFormFieldModule, MatInputModule, MatTableModule: Importações dos módulos dAngular Material para utilizar componentes e recursos visuais.
  - HttpClientModule: ImportaçãdmódulHTTP dAngular para realizar requisições HTTP.
  - FormsModule, ReactiveFormsModule: Importações dos módulos de formulários dAngular para trabalhar com formulários reativos e template-driven.
- Declarações:
  - AppComponent, UsuariosComponent, QuestionarioComponent, QuestoesComponent, UsuarioComponent: Declaraçãdos componentes que serãutilizados naplicativo.
- Imports:
  - BrowserModule: Importaçãdmóduldnavegador necessáripara rodar aplicativnnavegador.

- AppRoutingModule: Importaçãdmódulde roteamentpara definir as rotas daplicativo.
- BrowserAnimationsModule: Importaçãdmódulde animações dnavegador para habilitar as animações naplicativo.
- MatToolbarModule, MatIconModule, MatListModule, MatSidenavModule, MatFormFieldModule, MatInputModule, MatTableModule: Importações dos módulos dAngular Material para utilizar os componentes visuais fornecidos pelMaterial Design.
- HttpClientModule: ImportaçãdmódulHTTP dAngular para realizar requisições HTTP naplicativo.
- FormsModule, ReactiveFormsModule: Importações dos módulos de formulários dAngular para utilizar recursos de formulários reativos e template-driven.

- Providers: Não há nenhum provedor de serviço especificado neste exemplo.
- Bootstrap: O componente AppComponent é definido como o componente raiz do aplicativo, que será iniciado quando o aplicativo for iniciado.

Esse código mostra uma configuração básica do módulo AppModule de um aplicativo Angular, importando e configurando os módulos necessários para o funcionamento correto do aplicativo, bem como declarando os componentes que serão utilizados.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
import { MatToolbarModule} from '@angular/material/toolbar';
import { MatIcon, MatIconModule} from '@angular/material/icon';
import { MatListModule} from '@angular/material/list';
import { MatSidenavModule} from '@angular/material/sidenav';
import { UsuariosComponent } from
'./components/usuarios/usuarios.component';
import { QuestionarioComponent } from
```

```
'./components/questionario/questionario.component';
import { MatFormFieldModule} from '@angular/material/form-field';
import { MatInputModule} from '@angular/material/input';
import { MatTableModule } from '@angular/material/table';
import { HttpClientModule } from '@angular/common/http';
import { QuestoesComponent } from
'./components/questoes/questoes.component';
import { UsuarioComponent } from './components/usuario/usuario.component';
import { FormsModule } from '@angular/forms';
import { ReactiveFormsModule } from '@angular/forms';



@NgModule({
declarations: [
AppComponent,
UsuariosComponent,
QuestionarioComponent,
QuestoesComponent,
UsuarioComponent
],
imports: [
BrowserModule,
AppRoutingModule,
BrowserAnimationsModule,
MatToolbarModule,
MatIconModule,
MatListModule,
MatSidenavModule,
MatFormFieldModule,
MatInputModule,
MatTableModule,
HttpClientModule,
FormsModule,
ReactiveFormsModule
],
```

```
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

TEMA 5 – GERENCIAMENTO E TRATAMENTO DE ERROS: HANDLERS

No contexto do gerenciamento e tratamento de erros em aplicações, os handlers (tratadores) são responsáveis por lidar com exceções e erros que ocorrem durante a execução do código. Eles permitem capturar, processar e responder aos erros de maneira adequada, garantindo que a aplicação continue funcionando corretamente e fornecendo informações úteis aos desenvolvedores e usuários.

Existem diferentes tipos de handlers que podem ser implementados em uma aplicação para lidar com erros de diferentes maneiras:

- Error Handlers (tratadores de erros): os error handlers são usados para capturar e tratar erros inesperados que ocorrem durante a execução da aplicação. Eles são geralmente implementados como funções que são chamadas quando uma exceção é lançada. O objetivo desses handlers é registrar o erro, exibir mensagens de erro significativas e, se possível, fornecer uma solução alternativa ou contornar o erro para manter a estabilidade da aplicação.
- Exception Handlers (tratadores de exceções): os exception handlers são usados para capturar e tratar exceções específicas que podem ocorrer em determinados trechos de código. Eles são implementados para lidar com situações excepcionais que podem ser previstas e tratadas de maneira adequada. Por exemplo, um exception handler pode ser usado para capturar uma exceção de divisão por zero e fornecer uma mensagem de erro personalizada ao usuário.
- Event Handlers (tratadores de eventos): os event handlers são usados para lidar com erros que ocorrem em resposta a eventos específicos na aplicação. Eles são implementados para capturar eventos de erro e executar ações específicas em resposta a esses eventos. Por exemplo, um event handler pode ser usado para lidar com erros de validação em

um formulário, exibindo mensagens de erro ao usuário ou destacando campos inválidos.

- HTTP Error Handlers (Tratadores de erros HTTP): Os HTTP error handlers são usados para lidar com erros específicos relacionados a solicitações HTTP. Eles são implementados para capturar respostas de erro de servidores HTTP, como códigos de status 404 (Não encontrado) ou 500 (Erro interno do servidor), e fornecer uma resposta apropriada ao cliente. Isso pode incluir redirecionar para uma página de erro personalizada, exibir mensagens de erro ou executar ações específicas com base no código de status retornado.

A implementação e o uso de handlers podem variar dependendo da linguagem de programação e do framework ou biblioteca utilizada. É importante considerar a estrutura da aplicação e os requisitos específicos ao implementar handlers adequados para o gerenciamento e tratamento de erros.

```
private handleError(error: HttpErrorResponse) {
  if (error.status === 0) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong.
    console.error(
      `Backend returned code ${error.status}, body was: `, error.error);
  }
  // Return an observable with a user-facing error message.
  return throwError(() => new Error('Something bad happened; please try again later.'));
}
```

5.1 Try-Except

Em TypeScript, o conceito de *try-catch* é semelhante ao utilizado em Python e é usado para lidar com exceções ou erros que podem ocorrer durante a execução de um programa. Ele permite que você capture e trate exceções

específicas de forma elegante, evitando que o programa seja interrompido abruptamente.

Aqui está a sintaxe básica de um bloco try-catch em TypeScript. Mais adiante, em etapa posterior, explicaremos sobre o método GET, que é usado para “pegar” algo em uma rede. Por enquanto, entenda que qualquer tipo de tratamento de erro deve ser feito da seguinte maneira:

```
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 });
     }
   });
 }
```

Aqui está como ele funciona:

- O código dentro do bloco try é executado.
- Se ocorrer uma exceção dentro do bloco try, o controle é imediatamente transferido para o bloco catch, e o objeto de exceção é capturado na variável error.
- O código dentro do bloco catch é executado, em que você pode tratar a exceção de acordo com suas necessidades.

Após o tratamento da exceção, a execução continua normalmente a partir do ponto seguinte ao bloco catch.

Neste exemplo, estamos tentando realizar uma divisão por zero, o que gera uma exceção Error. O bloco catch captura a exceção e imprime uma mensagem de erro. Note que não precisamos especificar o tipo da exceção no bloco catch em TypeScript. O objeto de exceção é capturado na variável error, e

você pode utilizá-lo para obter informações adicionais sobre a exceção, como a mensagem de erro.

É importante destacar que o uso do try-catch deve ser feito de forma cuidadosa e restrito a situações excepcionais. Geralmente, é recomendado capturar apenas as exceções específicas que você espera e tratá-las adequadamente, em vez de usar um bloco catch amplo que capture todas as exceções indiscriminadamente.

5.2 ErrorHandler

Um ErrorHandler é um componente ou conjunto de técnicas utilizadas para lidar com erros em um sistema ou aplicação. Sua função principal é capturar e gerenciar exceções e condições inesperadas que podem ocorrer durante a execução de um programa. Um ErrorHandler realiza várias tarefas importantes, como a captura de erros, o registro de informações relevantes sobre os erros ocorridos, o tratamento adequado dos erros, a notificação e alerta para a equipe responsável e o monitoramento e análise dos erros ocorridos. Ele desempenha um papel fundamental no desenvolvimento de software, contribuindo para a estabilidade, confiabilidade e manutenibilidade de um sistema. Um ErrorHandler bem implementado melhora a experiência do usuário, minimiza interrupções e falhas inesperadas e permite que os desenvolvedores identifiquem e resolvam problemas de forma eficiente.

FINALIZANDO

Nesta etapa, entendemos vários conceitos de *arquitetura* e de *Angular*.

Compreendemos o uso de injetores e componentes. Embora possam parecer complexos, são modernos e muito úteis, sendo fundamentais em qualquer framework moderno.

O termo *handler* também é muito importante. Com ele, vimos que podemos criar uma função ou rotina de código que é responsável por lidar com eventos, erros ou solicitações específicas em um programa ou sistema.

Também compreendemos que existem muitas chamadas *assíncronas* em resposta a eventos que ocorrem durante a execução do programa.

Enfim, esta é uma etapa extremamente importante para a construção e utilização de aplicações. Ela não nos fornece um produto pronto, mas nos apresenta diversas formas de utilização.

REFERÊNCIAS

ANGULAR. Disponível em: <https://www.angular.io>. Acesso em: 9 ago. 2023.

APACHE. Disponível em: <https://struts.apache.org>. Acesso em: 9 ago. 2023.

BALSAMIQ. Disponível em: <https://balsamiq.com>. Acesso em: 9 ago. 2023.

BETRYBE. Disponível em: <https://blog.betrybe.com/framework-de-programacao/angular>. Acesso em: 9 ago. 2023.

BIACHI, L. O que é Aplicação, Framework e Linguagem de programação qual a diferença? O que e qual devo escolher quando estou iniciando minha carreira? **LinkedIn**, 18 abr. 2023. Disponível em: <https://pt.linkedin.com/pulse/o-que-%C3%A9-aplica%C3%A7%C3%A3o-framework-e-linguagem-de-qual-diferen%C3%A7a-bianch>. Acesso em: 9 ago. 2023.

BOOTSTRAP. Disponível em: <https://getbootstrap.com>. Acesso em: 9 ago. 2023.

DJANGO. Disponível em: <https://www.djangoproject.com>. Acesso em: 9 ago. 2023.

FRAIN, B. **Responsive Web Design with HTML5 and CSS - Fourth Edition**: Build future-proof responsive websites using the latest HTML5 and CSS techniques. Packt, 2022.

FIGMA. Disponível em: <https://www.figma.com>. Acesso em: 9 ago. 2023.

GAMMA, E. et al. **Padrões de projeto**: soluções de software orientado a objetos. Porto Alegre: Bookman, 2000.

GENEXUS. Disponível em: <https://training.genexus.com/pt/aprendizagem/pdf/arquitetura-de-aplicativo-angular-pdf>. Acesso em: 9 ago. 2023.

GOULÃO, M. **Component-Based Software Engineering**: a Quantitative Approach. Universidade Nova de Lisboa, 2008.

INSOMNIA. Disponível em: <https://insomnia.rest>. Acesso em: 9 ago. 2023.

LARAVEL. Disponível em: <https://laravel.com>. Acesso em: 9 ago. 2023.

LEARN. Disponível em: <https://learn.microsoft.com/pt-br/windows/wsl/tutorials/wsl-vscode>. Acesso em: 9 ago. 2023.

NORMAN, D. A. **Design for a Better World**: Meaningful, Sustainable, Humanity Centered. Cambridge, Massachusetts: The MIT Press, 2023.

NPM – Node Package Manager. Disponível em: <https://www.npmjs.com>. Acesso em: 9 ago. 2023.

REACT. Disponível em: <https://www.react.dev>. Acesso em: 9 ago. 2023.

SPRING. Disponível em: <https://spring.io>. Acesso em: 9 ago. 2023.

SKETCH. Disponível em: <https://www.sketch.com/. Acesso em: 10 mai. 2023.

SCHMIDT, D. C.; WALLNAU, K. **Component-Based Software Engineering**: Putting the Pieces Together, por Clemens Szyperski. Addison Wesley, 2001.

TSCONFIG. Disponível em: <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html. Acesso em>. Acesso em: 9 ago. 2023.

TYPESCRIPT. Disponível em: <https://www.typescriptlang.org>. Acesso em: 9 ago. 2023.

VUEJS. Disponível em: <https://vuejs.org>. Acesso em: 9 ago. 2023.

WEBFOUNDATION. **World Wide Web Foundation**. Disponível em: <https://webfoundation.org>. Acesso em: 9 ago. 2023.

W3C. Disponível em: <https://www.w3schools.in/mvc-architecture>. Acesso em: 9 ago. 2023.