1. Introdução ao Angular

O que é o Angular?

O Angular é um framework de desenvolvimento web criado pelo Google que permite construir aplicativos web modernos e dinâmicos. Ele utiliza a linguagem TypeScript e oferece um conjunto de ferramentas poderosas para criar interfaces de usuário eficientes.

**Principais recursos do Angular**

Componentização

Data binding

Roteamento

Injeção de dependência

Testabilidade

Diferenças entre Angular e AngularJS

AngularJS vs. Angular

Motivos para a migração

Instalação e configuração do Angular

Para começar a desenvolver com Angular, você precisa configurar o ambiente de desenvolvimento. Isso inclui a instalação do Node.js, do Angular CLI e a criação de um novo projeto Angular.

Pré-requisitos

Antes de começar, certifique-se de que você tenha os seguintes pré-requisitos instalados em seu sistema:

Node.js: O Angular requer o Node.js. Baixe e instale a versão mais recente do Node.js no site oficial.

NPM (Node Package Manager): O NPM é instalado junto com o Node.js. Certifique-se de que o NPM esteja disponível em seu sistema.

Instalação do Angular CLI

O Angular CLI (Command Line Interface) é uma ferramenta que simplifica o desenvolvimento e a criação de projetos Angular. Para instalá-lo, abra o terminal ou prompt de comando e execute o seguinte comando:

npm install -g @angular/cli

Esse comando instala o Angular CLI globalmente em seu sistema, permitindo que você crie, desenvolva e gerencie projetos Angular.

Criando um Novo Projeto Angular

Com o Angular CLI instalado, você pode criar um novo projeto Angular. Execute o seguinte comando:

ng new nome-do-seu-projeto

Substitua nome-do-seu-projeto pelo nome que você deseja dar ao seu projeto. O Angular CLI fará algumas perguntas sobre configurações, como roteamento e folhas de estilo. Você pode responder às perguntas de acordo com suas necessidades.

Navegando até a Pasta do Projeto

Após a criação do projeto, navegue até a pasta do projeto usando o seguinte comando:

cd nome-do-seu-projeto

Executando o Aplicativo

Para iniciar o servidor de desenvolvimento e executar o aplicativo, use o seguinte comando:

ng serve

O servidor de desenvolvimento será executado e, por padrão, o aplicativo estará disponível em http://localhost:4200/. Abra seu navegador e acesse essa URL para ver seu aplicativo Angular em execução.

Outros Comandos Úteis

Para criar um novo componente, use o comando ng generate component nome-do-seu-componente.

Para criar um novo serviço, use o comando ng generate service nome-do-seu-servico.

Para criar um novo módulo, use o comando ng generate module nome-do-seu-modulo.

Lembre-se de que o Angular CLI é uma ferramenta poderosa que simplifica o desenvolvimento Angular, oferecendo comandos para criar rapidamente componentes, serviços, módulos e outras partes do aplicativo.

Esses são os passos básicos para instalar o Angular e criar um novo projeto. Você pode, então, começar a desenvolver seu aplicativo Angular e personalizar conforme suas necessidades. Certifique-se de explorar a documentação oficial do Angular (https://angular.io/) para obter mais informações sobre desenvolvimento e conceitos avançados.

2. Componentes

Criando componentes

Componentes são a estrutura básica de um aplicativo Angular. Vamos aprender como criar e usar componentes.

A criação de componentes é uma parte fundamental do desenvolvimento com o Angular. Os componentes são os blocos de construção de qualquer aplicativo Angular. Eles encapsulam uma parte específica da funcionalidade do aplicativo, incluindo seu próprio HTML, CSS e lógica. Aqui estão os passos para criar um componente Angular:

Usando o Angular CLI (Recomendado)

O Angular CLI (Command Line Interface) torna a criação de componentes extremamente fácil. Certifique-se de que você tenha o Angular CLI instalado, como mencionado nas etapas anteriores.

Abra o terminal ou prompt de comando.

Navegue até o diretório do seu projeto Angular (onde o arquivo angular.json está localizado).

Para criar um novo componente, execute o seguinte comando, substituindo nome-do-seu-componente pelo nome desejado para o seu componente:

ng generate component nome-do-seu-componente

Por exemplo, se você quiser criar um componente chamado "meu-componente", o comando será:

ng generate component meu-componente

O Angular CLI irá gerar automaticamente os arquivos do componente, bem como configurá-lo para você. O resultado deve ser algo parecido com:

CREATE src/app/meu-componente/meu-componente.component.html

CREATE src/app/meu-componente/meu-componente.component.ts

CREATE src/app/meu-componente/meu-componente.component.spec.ts

CREATE src/app/meu-componente/meu-componente.component.css

UPDATE src/app/app.module.ts

Isso cria um novo componente chamado "meu-componente" no diretório src/app. Ele inclui um arquivo HTML, um arquivo TypeScript, um arquivo de estilo CSS e um arquivo de teste.

Manualmente (Alternativa)

Se preferir criar um componente manualmente, você pode seguir estes passos:

Crie uma nova pasta para o seu componente em src/app. Por exemplo, meu-componente.

Dentro da pasta do seu componente, crie os seguintes arquivos:

meu-componente.component.ts: Este é o arquivo TypeScript do componente.

meu-componente.component.html: Este é o arquivo HTML do componente.

meu-componente.component.css (opcional): Este é o arquivo de estilo do componente.

No arquivo TypeScript do seu componente (meu-componente.component.ts), defina o componente, importando os módulos necessários e configurando o decorador @Component para definir as propriedades do componente, como o seletor, o template HTML, os estilos e a lógica do componente.

import { Component } from '@angular/core';

@Component({

selector: 'app-meu-componente',

templateUrl: './meu-componente.component.html',

styleUrls: ['./meu-componente.component.css']

})

export class MeuComponenteComponent {

// Lógica do componente aqui

}

No arquivo HTML do seu componente (meu-componente.component.html), defina o template HTML que será renderizado quando o componente for usado.

<p>Este é o meu componente.</p>

Adicione o componente ao módulo principal do seu aplicativo (geralmente app.module.ts) importando o componente e adicionando-o à matriz de declarações.

import { MeuComponenteComponent } from './meu-componente/meu-componente.component';

@NgModule({

declarations: [

// ... outros componentes

MeuComponenteComponent

],

// ...

})

Depois de criar o componente, você pode usá-lo em seus templates HTML, adicionar lógica e personalizá-lo de acordo com as necessidades do seu aplicativo. Certifique-se de que o componente esteja registrado no módulo para que o Angular saiba sobre ele

Ciclo de vida dos componentes

Os componentes no Angular têm um ciclo de vida bem definido, que permite executar lógica em estágios específicos durante a criação, atualização e destruição de um componente. Os principais métodos do ciclo de vida dos componentes Angular são os seguintes:

ngOnChanges(): Este método é chamado sempre que as propriedades de entrada de um componente são alteradas. Ele recebe um objeto SimpleChanges que contém informações sobre as alterações nas propriedades.

ngOnInit(): Este método é chamado uma vez, após a criação do componente e a inicialização das propriedades de entrada. É comumente usado para a inicialização de dados e para chamadas de serviços.

ngDoCheck(): Este método é chamado durante cada ciclo de detecção de mudanças. Ele permite que você realize verificações personalizadas para detectar alterações não detectadas pelo Angular.

ngAfterContentInit(): Chamado após o conteúdo do componente filho (projetado) ser inicializado.

ngAfterContentChecked(): Chamado após cada verificação de conteúdo, permitindo que você execute ações após a verificação do conteúdo do componente.

ngAfterViewInit(): Chamado após a visão (template) do componente e suas visualizações filho serem totalmente inicializadas.

ngAfterViewChecked(): Chamado após cada verificação de visualização, permitindo que você execute ações após a verificação das visualizações do componente.

ngOnDestroy(): Este método é chamado antes de um componente ser destruído. É usado para liberar recursos, cancelar assinaturas e limpar a memória para evitar vazamentos.

3. Templates e Data Binding

Em Angular, os conceitos de "Templates" (modelos) e "Data Binding" (vinculação de dados) são fundamentais para criar aplicativos dinâmicos e interativos. Vamos entender cada um deles:

Templates (Modelos):

Definição: Um template em Angular é uma representação visual da interface do usuário (UI) do seu aplicativo. Ele define como os elementos da interface, como HTML, CSS e às vezes JavaScript, serão renderizados no navegador.

Características:

HTML Aprimorado: Os templates em Angular são basicamente documentos HTML, mas com adições específicas do Angular, como diretivas, expressões e vinculação de dados.

Exemplo de um template simples em Angular:

<h1>{{titulo}}</h1>

<p \*ngIf="mostrarParagrafo">Este é um parágrafo condicional.</p>

<ul>

<li \*ngFor="let item of listaItens">{{item}}</li>

</ul>

Data Binding (Vinculação de Dados):

Certifique-se de que o módulo FormsModule está importado em seu módulo Angular para usar o ngModel.

Definição: Data Binding é o mecanismo que sincroniza automaticamente os dados do modelo (componente) com a interface do usuário. Ele permite que as alterações em um afetem o outro de forma bidirecional.

Tipos de Data Binding:

One-Way Binding (Vinculação de um único sentido):

Exemplo de Data Binding em Angular:

<input [(ngModel)]="nome" />

<p>Olá, {{nome}}!</p>

Interpolação

A interpolação em Angular é uma forma de realizar vinculação de dados em um template, permitindo que você incorpore expressões dinâmicas diretamente no texto do modelo. Essas expressões são avaliadas e os resultados são incorporados na exibição final do template. A sintaxe para interpolação em Angular usa as chaves duplas {{}}.

Aqui está um exemplo simples de interpolação em Angular:

<p>Olá, {{nome}}!</p>

Neste exemplo, {{nome}} é uma expressão Angular que será substituída pelo valor da propriedade nome do componente quando o template for renderizado. Se o valor de nome no componente for "João", a frase resultante será "Olá, João!".

A interpolação é uma forma rápida e direta de exibir valores dinâmicos no seu template sem a necessidade de criar elementos adicionais. Além disso, ela é frequentemente usada para exibir o resultado de expressões ou chamadas de métodos diretamente no conteúdo do template.

Além da interpolação com chaves duplas, como no exemplo acima, o Angular oferece outras formas de vinculação de dados, como o uso de diretivas como ngModel para vinculação bidirecional, e a vinculação de propriedades e eventos usando colchetes [] e parênteses (), respectivamente.

<!-- Exemplo de vinculação bidirecional com ngModel -->

<input [(ngModel)]="nome" />

<p>Olá, {{nome}}!</p>

Em resumo, a interpolação em Angular é uma técnica que permite incorporar dinamicamente valores de propriedades de componentes ou resultados de expressões diretamente no conteúdo do template, tornando a exibição de dados dinâmicos de maneira declarativa e concisa.

Property Binding

Property Binding é uma técnica em Angular que permite vincular propriedades de elementos HTML a expressões ou valores no componente. Ele é uma forma de vinculação de dados unidirecional, onde os dados fluem do componente para o template.

A sintaxe para Property Binding utiliza colchetes []. Aqui está um exemplo básico:

<img [src]="imagemUrl" alt="Descrição da imagem" />

Neste exemplo, a propriedade src da tag <img> está vinculada à expressão imagemUrl no componente. O valor de imagemUrl no componente será atribuído dinamicamente à propriedade src da tag <img>. Se imagemUrl no componente for uma URL de imagem, essa imagem será exibida.

Outro exemplo comum de Property Binding envolve a modificação de propriedades de estilo:

<button [style.backgroundColor]="corDeFundo">Clique-me</button>

Neste caso, a cor de fundo do botão é vinculada à propriedade corDeFundo do componente. Se o valor de corDeFundo for, por exemplo, "red", a cor de fundo do botão será alterada para vermelho.

Property Binding é especialmente útil para atualizar dinamicamente propriedades de elementos HTML com base nas mudanças nos dados do componente. Essa técnica é uma das formas fundamentais de se conseguir interatividade em aplicações Angular.

Em resumo, Property Binding em Angular é uma técnica que permite vincular dinamicamente propriedades de elementos HTML a expressões ou valores no componente, facilitando a atualização unidirecional dessas propriedades com base nas mudanças no estado do componente.

Event Binding

Event Binding é uma técnica em Angular que permite vincular eventos de elementos HTML a métodos no componente. Isso possibilita a execução de ações no componente em resposta a eventos do usuário, como cliques, teclas pressionadas, movimentos do mouse, entre outros.

A sintaxe para Event Binding utiliza parênteses (). Aqui está um exemplo básico:

<button (click)="onClick()">Clique-me</button>

Neste exemplo, o método onClick() do componente será chamado quando o botão for clicado. O evento de clique (click) está vinculado ao método do componente.

É possível passar informações do evento para o método do componente. Por exemplo, ao lidar com eventos de clique, você pode passar o objeto de evento como um argumento para o método:

<button (click)="onClick($event)">Clique-me</button>

No componente:

onClick(event: MouseEvent) {

// Faça algo com o objeto de evento, se necessário

console.log(event);

}

Outro exemplo comum envolve a vinculação de eventos de entrada, como ao pressionar uma tecla:

<input (keyup)="onKeyUp($event)" />

No componente:

onKeyUp(event: KeyboardEvent) {

// Faça algo com o objeto de evento, se necessário

console.log(event.key);

}

Event Binding é uma técnica crucial para criar interatividade em aplicações Angular, permitindo que ações no componente sejam desencadeadas por eventos do usuário. Combinado com Property Binding, isso oferece uma maneira poderosa de criar aplicativos dinâmicos e responsivos.

Vinculação Bidirecional (Two-Way Binding):

Two-Way Binding é uma técnica em Angular que combina Property Binding e Event Binding para permitir a sincronização bidirecional entre uma propriedade no componente e um elemento no template. Isso significa que as alterações na propriedade do componente são refletidas automaticamente no template, e vice-versa.

A sintaxe para Two-Way Binding utiliza a diretiva ngModel e a notação de parênteses e colchetes combinados [(ngModel)]. Aqui está um exemplo com um campo de entrada:

<input [(ngModel)]="nome" />

<p>Olá, {{nome}}!</p>

Neste exemplo, a propriedade nome do componente está vinculada bidirecionalmente ao campo de entrada. Quando o usuário digita algo no campo de entrada, a propriedade nome é automaticamente atualizada no componente, e qualquer alteração na propriedade nome no componente é refletida no campo de entrada.

Para usar Two-Way Binding com ngModel, você precisa importar o módulo FormsModule no seu módulo Angular:

import { FormsModule } from '@angular/forms';

@NgModule({

imports: [FormsModule],

// ...

})

export class SeuModulo { }

Essa técnica é frequentemente usada em formulários, onde a sincronização bidirecional entre o modelo do componente e os elementos do formulário é essencial.

É importante observar que Two-Way Binding com ngModel é eficaz principalmente para formulários simples. Para formulários mais complexos, pode ser necessário considerar abordagens reativas usando o módulo ReactiveFormsModule e formulários reativos em Angular.

Em resumo, Two-Way Binding em Angular é uma técnica que permite a sincronização bidirecional entre uma propriedade do componente e um elemento no template, facilitando a manipulação de dados em ambos os lados de forma eficiente.

Diretivas

Diretivas são um dos pilares fundamentais do Angular e são usadas para estender o HTML com funcionalidades adicionais e comportamentos personalizados. Elas permitem que você crie componentes reutilizáveis, manipule o DOM, adicione comportamentos dinâmicos aos elementos e muito mais. Existem dois tipos principais de diretivas em Angular: diretivas estruturais e diretivas de atributos.

Diretivas Estruturais:

As diretivas estruturais alteram a estrutura do DOM adicionando, removendo ou substituindo elementos. Elas são precedidas por um asterisco \* no HTML.

Exemplos comuns incluem ngIf (para adicionar ou remover elementos com base em uma condição) e ngFor (para iterar sobre uma lista e criar elementos correspondentes).

Exemplo de ngIf:

<div \*ngIf="mostrarElemento">

Este elemento será exibido se mostrarElemento for verdadeiro.

</div>

Exemplo de ngFor:

<ul>

<li \*ngFor="let item of listaItens">{{ item }}</li>

</ul>

Diretivas de Atributos:

As diretivas de atributos são usadas para alterar a aparência ou o comportamento de um elemento sem afetar sua estrutura. Elas são aplicadas como atributos nos elementos HTML.

Exemplos comuns incluem ngStyle (para aplicar estilos dinamicamente) e ngClass (para adicionar ou remover classes com base em condições).

Exemplo de ngStyle:

<div [ngStyle]="{'color': textColor, 'font-size': fontSize}">

Este texto terá cor e tamanho de fonte dinâmicos.

</div>

Exemplo de ngClass:

<div [ngClass]="{'destaque': destacar, 'negrito': negrito}">

Este texto terá classes CSS adicionadas dinamicamente.

</div>

Além das diretivas integradas, você pode criar suas próprias diretivas personalizadas em Angular para encapsular comportamentos específicos e promover a reutilização de código. Diretivas são uma parte essencial da abordagem declarativa do Angular, permitindo que você descreva como o DOM deve se comportar com base no estado e nas condições do aplicativo.

ngModel:

Essa diretiva é frequentemente usada em formulários para estabelecer uma ligação bidirecional entre os campos de entrada do formulário e as propriedades do componente. Permite que as alterações no campo de entrada atualizem automaticamente a propriedade no componente e vice-versa.

Exemplo:

<input [(ngModel)]="nome" />

Diretivas de Componentes:

Componentes: Em Angular, os componentes são diretivas com um template. Eles encapsulam a lógica, o comportamento e a aparência em um único bloco reutilizável.

Exemplo:

@Component({

selector: 'app-meu-componente',

template: '<p>Este é o conteúdo do meu componente.</p>'

})

export class MeuComponente {}

Estas são algumas das principais diretivas em Angular. Vale ressaltar que você também pode criar suas próprias diretivas personalizadas para atender a requisitos específicos do seu aplicativo, estendendo assim a funcionalidade do Angular.

ng-template

<ng-template> é um elemento HTML especial no Angular que é usado para declarar partes de um template que não serão renderizadas imediatamente quando a página for carregada. Em vez disso, o conteúdo dentro de uma <ng-template> será armazenado no DOM, mas não será exibido até ser ativado por alguma lógica no código Angular.

Existem algumas situações em que o <ng-template> é comumente utilizado:

ngIf com ng-template:

Às vezes, você precisa de uma estrutura condicional mais complexa do que a fornecida pela diretiva ngIf sozinha. Nesses casos, você pode usar o <ng-template> para criar blocos de conteúdo condicional.

Exemplo:

<ng-template [ngIf]="mostrarElemento">

<p>Este parágrafo será exibido se mostrarElemento for verdadeiro.</p>

</ng-template>

ngFor com ng-template:

Similar ao exemplo acima, você pode usar <ng-template> com a diretiva ngFor para criar estruturas mais complexas ao iterar sobre uma lista.

Exemplo:

<ng-template ngFor let-item [ngForOf]="listaItens">

<li>{{ item }}</li>

</ng-template>

Criação de Modelos Reutilizáveis:

<ng-template> é útil quando você deseja definir um bloco de conteúdo reutilizável que pode ser referenciado em várias partes do seu template.

Exemplo:

<ng-template #meuBloco>

<p>Este é um bloco de conteúdo reutilizável.</p>

</ng-template>

<div>

Conteúdo antes do bloco.

<ng-container [ngTemplateOutlet]="meuBloco"></ng-container>

Conteúdo depois do bloco.

</div>

Neste exemplo, o conteúdo do <ng-template> com a referência #meuBloco é inserido no DOM usando [ngTemplateOutlet]="meuBloco".

O <ng-template> é uma ferramenta poderosa para criar templates mais dinâmicos e flexíveis em Angular, especialmente quando você precisa lidar com estruturas condicionais ou repetitivas mais complexas.

ng-content

<ng-content> é uma diretiva no Angular que permite a criação de componentes mais flexíveis e reutilizáveis, permitindo a passagem de conteúdo do componente pai para o componente filho. Com <ng-content>, você pode criar componentes que encapsulam uma parte da estrutura do DOM, mas permitem que o conteúdo interno seja fornecido dinamicamente a partir do componente que o utiliza.

A principal utilização de <ng-content> é criar slots de conteúdo dentro de um componente, onde o conteúdo é inserido durante a utilização do componente. Aqui está um exemplo básico:

<!-- MeuComponente -->

<div class="meu-componente">

<h2>Título do Componente</h2>

<ng-content></ng-content>

</div>

Neste exemplo, <ng-content></ng-content> atua como um espaço reservado para o conteúdo que será fornecido quando o componente for utilizado. Agora, ao usar o componente em algum lugar:

<!-- Uso do MeuComponente -->

<app-meu-componente>

<p>Conteúdo personalizado vai aqui.</p>

</app-meu-componente>

O parágrafo <p> é inserido dentro de <ng-content> no componente MeuComponente. Assim, o componente pode encapsular uma estrutura específica, mas o conteúdo interno é determinado pelo uso do componente.

Você também pode ter múltiplos pontos de inserção de conteúdo usando seletores, o que é útil quando você deseja inserir diferentes tipos de conteúdo em áreas específicas do componente.

Exemplo com seletores:

<!-- MeuComponente com seletores -->

<div class="meu-componente">

<h2>Título do Componente</h2>

<ng-content select=".conteudo-topo"></ng-content>

<div class="conteudo-padrao">

<ng-content></ng-content>

</div>

</div>

<!-- Uso do MeuComponente com seletores -->

<app-meu-componente>

<p class="conteudo-topo">Conteúdo personalizado vai aqui no topo.</p>

<p>Conteúdo padrão vai aqui.</p>

</app-meu-componente>

Neste exemplo, o primeiro parágrafo com a classe conteudo-topo é selecionado pela primeira chamada de <ng-content>, enquanto o segundo parágrafo vai para a segunda chamada.

Atributo específico como seletor

No Angular, o ng-content é usado para criar slots de conteúdo em componentes. Se você quiser selecionar o conteúdo com base em um atributo específico, você pode usar a diretiva estrutural ng-content em conjunto com a classe ngProjectAs.

Aqui está um exemplo de como você pode usar um atributo como seletor no ng-content:

Suponha que você tenha um componente chamado meu-componente e deseja selecionar o conteúdo com base em um atributo chamado seletor:

// meu-componente.component.ts

import { Component } from '@angular/core';

@Component({

selector: 'meu-componente',

template: `

<div>

<ng-content select="[seletor]"></ng-content>

</div>

`,

})

export class MeuComponenteComponent {

}

No exemplo acima, o ng-content está usando a diretiva select para escolher apenas o conteúdo que tem um atributo chamado seletor. Isso significa que, ao usar meu-componente em outro lugar do seu aplicativo, apenas o conteúdo com o atributo seletor será projetado no local correto dentro do template do meu-componente.

<!-- app.component.html -->

<meu-componente>

<div seletor>Conteúdo com seletor</div>

<div>Conteúdo sem seletor</div>

</meu-componente>

Neste exemplo, apenas o primeiro <div> será projetado dentro do template de meu-componente, porque ele possui o atributo seletor. O segundo <div> será ignorado.

<ng-content> é uma ferramenta poderosa para criar componentes mais flexíveis e genéricos, permitindo uma melhor composição de interfaces de usuário em aplicações Angular.

Pipes

Em Angular, os pipes são transformadores visuais que você pode usar para formatar dados antes de exibi-los no modelo. Eles são semelhantes aos filtros em AngularJS. Os pipes são usados no modelo e podem ser encadeados para realizar transformações mais complexas. Aqui estão alguns exemplos de pipes em Angular:

Uppercase e Lowercase:

Transforma o texto para maiúsculas ou minúsculas.

<p>{{ 'Texto de Exemplo' | uppercase }}</p>

<p>{{ 'Texto de Exemplo' | lowercase }}</p>

Date:

Formata uma data de acordo com um formato específico.

<p>{{ today | date:'short' }}</p>

Currency: Formata um número como uma moeda.

<p>{{ price | currency:'USD':'symbol':'1.2-2' }}</p>

Percent:

Formata um número como uma porcentagem.

<p>{{ percentage | percent }}</p>

Slice:

Retorna uma parte de uma string ou array.

<p>{{ 'Hello World' | slice:0:5 }}</p>

JSON:

Converte um objeto em uma string JSON.

<p>{{ object | json }}</p>

Async:

Lida com valores assíncronos, especialmente útil com Observables.

<p>{{ asyncData | async }}</p>

Custom Pipe:

Você também pode criar seus próprios pipes personalizados para atender a requisitos específicos.

// Exemplo de um pipe personalizado para inverter uma string

@Pipe({ name: 'reverse' })

export class ReversePipe implements PipeTransform {

transform(value: string): string {

return value.split('').reverse().join('');

}

}

<p>{{ 'Angular' | reverse }}</p>

Estes são apenas alguns exemplos básicos de pipes no Angular. Eles podem ser combinados e personalizados para atender às necessidades específicas de formatação de dados no seu aplicativo Angular. Lembre-se de que os pipes não alteram os dados subjacentes, apenas a representação visual deles no modelo.

Controle de Fluxo

Comparação entre ngIf, ngFor e outras diretivas de controle.

<!-- Exemplo de ngIf -->

<div \*ngIf="isUserLoggedIn; else guestSection">

<p>Bem-vindo, usuário!</p>

</div>

<ng-template #guestSection>

<p>Faça login para acessar o conteúdo exclusivo.</p>

</ng-template>

<!-- Exemplo de ngFor -->

<ul>

<li \*ngFor="let item of items">{{ item.name }}</li>

</ul>

<!-- Exemplo de ngSwitch -->

<div [ngSwitch]="userRole">

<p \*ngSwitchCase="'admin'">Você tem acesso total.</p>

<p \*ngSwitchCase="'editor'">Você pode editar conteúdo.</p>

<p \*ngSwitchDefault>Você tem permissões padrão.</p>

</div>

ngIf - Controle de Fluxo Condicional

<div \*ngIf="isLoggedIn">

<p>Bem-vindo, {{ username }}!</p>

</div>

<div \*ngIf="!isLoggedIn">

<p>Faça login para acessar o conteúdo.</p>

</div>

ngIf-else e ngIf-then-else

<div \*ngIf="userRole === 'admin'; else regularUser">Olá, Admin!</div>

<ng-template #regularUser>Olá, Usuário Regular!</ng-template>

<div \*ngIf="isLoading; then loadingContent else regularContent"></div>

<ng-template #loadingContent>Aguarde, carregando...</ng-template>

<ng-template #regularContent>Conteúdo carregado com sucesso.</ng-template>

ngIf com Expressões

<div \*ngIf="userAge >= 18 && isUserLoggedIn">

<p>Você é maior de idade e está logado.</p>

</div>

ngFor

<ul>

<li \*ngFor="let item of items">{{ item.name }}</li>

</ul>

ngFor com Objetos e Iteráveis Complexos

<div \*ngFor="let user of users; let i = index">

<p>{{ i + 1 }}. {{ user.name }}</p>

</div>

ngFor com Filtros

<div \*ngFor="let item of items | filter: 'ativo'">{{ item.name }}</div>

Switch - Controle de Fluxo Multicondicional

<div [ngSwitch]="userRole">

<p \*ngSwitchCase="'admin'">Você tem acesso total.</p>

<p \*ngSwitchCase="'editor'">Você pode editar conteúdo.</p>

<p \*ngSwitchDefault>Você tem permissões padrão.</p>

</div>

ngSwitchCase e ngSwitchDefault

<div [ngSwitch]="userStatus">

<p \*ngSwitchCase="'ativo'">Seu status está ativo.</p>

<p \*ngSwitchCase="'inativo'">Seu status está inativo.</p>

<p \*ngSwitchDefault>Seu status é desconhecido.</p>

</div>

ngIf, ngFor e ngSwitch Juntos - Cenários Complexos

<div \*ngIf="isUserLoggedIn; else loginSection">

<!-- Se o usuário estiver logado -->

<div \*ngFor="let post of userPosts">

<h2>{{ post.title }}</h2>

<p>{{ post.content }}</p>

</div>

</div>

<ng-template #loginSection>

<!-- Se o usuário não estiver logado -->

<p>Faça login para visualizar os posts.</p>

</ng-template>

Em novembro deste ano, lançaremos a versão 17 do Angular com vários novos recursos de modelo, incluindo uma nova sintaxe integrada para fluxo de controle e visualizações diferidas. O novo fluxo de controle declarativo traz a funcionalidade de NgIf, NgFore NgSwitchpara a própria estrutura, e as visualizações diferidas permitem que os modelos carreguem dependências lentamente, em resposta a uma ou mais condições de gatilho configuráveis. Esta é a mudança mais significativa que já fizemos nos modelos Angular, e estamos trabalhando duro para dar os retoques finais nesses recursos.

<https://angular.io/guide/control_flow>

Template-Driven Forms no Angular

Importar FormModule

Ligação Bidirecional (Two-Way Binding) com ngModel

<!-- template-form.component.html -->

<form #userForm="ngForm" (ngSubmit)="onSubmit()">

<label for="username">Usuário:</label>

<input type="text" id="username" name="username" [(ngModel)]="user.username" required>

<label for="email">E-mail:</label>

<input type="email" id="email" name="email" [(ngModel)]="user.email" required>

<button type="submit">Enviar</button>

</form>

// template-form.component.ts

import { Component } from '@angular/core';

@Component({

selector: 'app-template-form',

templateUrl: './template-form.component.html',

})

export class TemplateFormComponent {

user = { username: '', email: '' };

onSubmit() {

console.log('Formulário enviado:', this.user);

}

}

Validação de Formulários em Template-Driven Forms

<!-- template-form.component.html -->

<form #userForm="ngForm" (ngSubmit)="onSubmit()">

<label for="username">Usuário:</label>

<input type="text" id="username" name="username" [(ngModel)]="user.username" required minlength="3" maxlength="15">

<div \*ngIf="userForm.controls['username'].invalid && (userForm.controls['username'].dirty || userForm.controls['username'].touched)">

<div \*ngIf="userForm.controls['username'].errors?.required">Campo obrigatório.</div>

<div \*ngIf="userForm.controls['username'].errors?.minlength">Mínimo de 3 caracteres.</div>

<div \*ngIf="userForm.controls['username'].errors?.maxlength">Máximo de 15 caracteres.</div>

</div>

<label for="email">E-mail:</label>

<input type="email" id="email" name="email" [(ngModel)]="user.email" required>

<button type="submit">Enviar</button>

</form>

Utilizando ngModel com Formulários Complexos

<!-- template-form.component.html -->

<form #userForm="ngForm" (ngSubmit)="onSubmit()">

<div ngModelGroup="personalInfo">

<label for="username">Usuário:</label>

<input type="text" id="username" name="username" [(ngModel)]="user.personalInfo.username" required>

<label for="email">E-mail:</label>

<input type="email" id="email" name="email" [(ngModel)]="user.personalInfo.email" required>

</div>

<div ngModelGroup="address">

<label for="street">Rua:</label>

<input type="text" id="street" name="street" [(ngModel)]="user.address.street" required>

<label for="city">Cidade:</label>

<input type="text" id="city" name="city" [(ngModel)]="user.address.city" required>

</div>

<button type="submit">Enviar</button>

</form>

Trabalhando com Seletores e Caixas de Verificação

<!-- template-form.component.html -->

<form #userForm="ngForm" (ngSubmit)="onSubmit()">

<label for="gender">Gênero:</label>

<select id="gender" name="gender" [(ngModel)]="user.gender" required>

<option value="male">Masculino</option>

<option value="female">Feminino</option>

</select>

<label for="newsletter">Receber Newsletter:</label>

<input type="checkbox" id="newsletter" name="newsletter" [(ngModel)]="user.newsletter">

<button type="submit">Enviar</button>

</form>

Manipulação de Estados e Reset de Formulários

<!-- template-form.component.html -->

<form #userForm="ngForm" (ngSubmit)="onSubmit()">

<!-- ... campos do formulário ... -->

<button type="submit">Enviar</button>

<button type="button" (click)="resetForm()">Resetar</button>

</form>

// template-form.component.ts

import { Component } from '@angular/core';

@Component({

selector: 'app-template-form',

templateUrl: './template-form.component.html',

})

export class TemplateFormComponent {

user = { /\* ... \*/ };

onSubmit() {

console.log('Formulário enviado:', this.user);

}

resetForm() {

this.user = { /\* ... \*/ };

// Também é possível resetar o estado do formulário usando ngForm

// this.userForm.resetForm();

}

}

Estilização e Experiência do Usuário com ngClass e ngSubmit

Adicionando Classes Dinamicamente com ngClass e Desativando o Botão de Envio com ngSubmit

<!-- template-form.component.html -->

<form #userForm="ngForm" (ngSubmit)="onSubmit()" [ngClass]="{ 'submitted': userForm.submitted, 'invalid': userForm.invalid }">

<!-- ... campos do formulário ... -->

<button type="submit" [disabled]="userForm.invalid">Enviar</button>

<button type="button" (click)="resetForm()">Resetar</button>

</form>

/\* styles.css \*/

.form-container {

width: 300px;

margin: 20px auto;

}

.invalid {

border: 2px solid #FF0000;

}

.submitted {

background-color: #E0E0E0;

}

Projeto Prático - Formulário de Cadastro de Usuário

<!-- registration-form.component.html -->

<form #registrationForm="ngForm" (ngSubmit)="onSubmit()" class="form-container">

<label for="username">Usuário:</label>

<input type="text" id="username" name="username" [(ngModel)]="user.username" required>

<label for="email">E-mail:</label>

<input type="email" id="email" name="email" [(ngModel)]="user.email" required>

<label for="password">Senha:</label>

<input type="password" id="password" name="password" [(ngModel)]="user.password" required minlength="8" pattern="(?=.\*\d)(?=.\*[a-z])(?=.\*[A-Z]).{8,}">

<div \*ngIf="registrationForm.controls['password'].invalid && (registrationForm.controls['password'].dirty || registrationForm.controls['password'].touched)">

<div \*ngIf="registrationForm.controls['password'].errors?.required">Senha obrigatória.</div>

<div \*ngIf="registrationForm.controls['password'].errors?.minlength">Mínimo de 8 caracteres.</div>

<div \*ngIf="registrationForm.controls['password'].errors?.pattern">A senha deve conter pelo menos um número, uma letra minúscula e uma letra maiúscula.</div>

</div>

<button type="submit" [disabled]="registrationForm.invalid">Registrar</button>

</form>

// registration-form.component.ts

import { Component } from '@angular/core';

@Component({

selector: 'app-registration-form',

templateUrl: './registration-form.component.html',

})

export class RegistrationFormComponent {

user = { username: '', email: '', password: '' };

onSubmit() {

console.log('Usuário registrado:', this.user);

}

}

Reactive Forms no Angular

Importar ReactiveFormsModule

Construção de Campos de Formulário

// reactive-form.component.ts

import { Component, OnInit } from '@angular/core';

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

@Component({

selector: 'app-reactive-form',

templateUrl: './reactive-form.component.html',

})

export class ReactiveFormComponent implements OnInit {

userForm: FormGroup;

constructor(private fb: FormBuilder) {}

ngOnInit() {

this.buildForm();

}

buildForm() {

this.userForm = this.fb.group({

username: ['', [Validators.required, Validators.minLength(3), Validators.maxLength(15)]],

email: ['', [Validators.required, Validators.email]],

password: ['', [Validators.required, Validators.minLength(8)]],

});

}

}

<!-- reactive-form.component.html -->

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" class="form-container">

<label for="username">Usuário:</label>

<input type="text" id="username" formControlName="username">

<div \*ngIf="userForm.get('username').hasError('required') && userForm.get('username').touched">Campo obrigatório.</div>

<div \*ngIf="userForm.get('username').hasError('minlength') && userForm.get('username').touched">Mínimo de 3 caracteres.</div>

<div \*ngIf="userForm.get('username').hasError('maxlength') && userForm.get('username').touched">Máximo de 15 caracteres.</div>

<label for="email">E-mail:</label>

<input type="email" id="email" formControlName="email">

<div \*ngIf="userForm.get('email').hasError('required') && userForm.get('email').touched">Campo obrigatório.</div>

<div \*ngIf="userForm.get('email').hasError('email') && userForm.get('email').touched">E-mail inválido.</div>

<label for="password">Senha:</label>

<input type="password" id="password" formControlName="password">

<div \*ngIf="userForm.get('password').hasError('required') && userForm.get('password').touched">Campo obrigatório.</div>

<div \*ngIf="userForm.get('password').hasError('minlength') && userForm.get('password').touched">Mínimo de 8 caracteres.</div>

<button type="submit" [disabled]="userForm.invalid">Enviar</button>

</form>

Validação de Formulários em Reactive Forms

// reactive-form.component.ts

import { Component, OnInit } from '@angular/core';

import { FormBuilder, FormGroup, Validators, AbstractControl } from '@angular/forms';

@Component({

selector: 'app-reactive-form',

templateUrl: './reactive-form.component.html',

})

export class ReactiveFormComponent implements OnInit {

userForm: FormGroup;

constructor(private fb: FormBuilder) {}

ngOnInit() {

this.buildForm();

}

buildForm() {

this.userForm = this.fb.group({

username: ['', [Validators.required, Validators.minLength(3), Validators.maxLength(15)]],

email: ['', [Validators.required, Validators.email]],

password: ['', [Validators.required, Validators.minLength(8)]],

confirmPassword: ['', [Validators.required]],

}, { validators: this.passwordMatchValidator });

}

passwordMatchValidator(control: AbstractControl) {

const password = control.get('password').value;

const confirmPassword = control.get('confirmPassword').value;

if (password !== confirmPassword) {

control.get('confirmPassword').setErrors({ passwordMismatch: true });

} else {

return null;

}

}

}

<!-- reactive-form.component.html -->

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" class="form-container">

<!-- ... campos do formulário ... -->

<label for="confirmPassword">Confirme a Senha:</label>

<input type="password" id="confirmPassword" formControlName="confirmPassword">

<div \*ngIf="userForm.hasError('passwordMismatch') && userForm.get('confirmPassword').touched">As senhas não coincidem.</div>

<button type="submit" [disabled]="userForm.invalid">Enviar</button>

</form>

Trabalhando com FormArrays e FormGroup Aninhados

// dynamic-form.component.ts

import { Component, OnInit } from '@angular/core';

import { FormBuilder, FormGroup, Validators, FormArray } from '@angular/forms';

@Component({

selector: 'app-dynamic-form',

templateUrl: './dynamic-form.component.html',

})

export class DynamicFormComponent implements OnInit {

userForm: FormGroup;

constructor(private fb: FormBuilder) {}

ngOnInit() {

this.buildForm();

}

buildForm() {

this.userForm = this.fb.group({

username: ['', [Validators.required, Validators.minLength(3), Validators.maxLength(15)]],

emails: this.fb.array([this.createEmailControl()]),

});

}

createEmailControl(): FormGroup {

return this.fb.group({

email: ['', [Validators.required, Validators.email]],

});

}

get emailControls(): FormArray {

return this.userForm.get('emails') as FormArray;

}

addEmailControl() {

this.emailControls.push(this.createEmailControl());

}

removeEmailControl(index: number) {

this.emailControls.removeAt(index);

}

onSubmit() {

console.log('Formulário enviado:', this.userForm.value);

}

}

<!-- dynamic-form.component.html -->

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" class="form-container">

<label for="username">Usuário:</label>

<input type="text" id="username" formControlName="username">

<div formArrayName="emails">

<div \*ngFor="let emailControl of emailControls.controls; let i = index" [formGroupName]="i">

<label for="email{{ i }}">E-mail {{ i + 1 }}:</label>

<input type="email" [id]="'email' + i" formControlName="email">

<button type="button" (click)="removeEmailControl(i)">Remover E-mail</button>

</div>

</div>

<button type="button" (click)="addEmailControl()">Adicionar E-mail</button>

<button type="submit" [disabled]="userForm.invalid">Enviar</button>

</form>

Controle Dinâmico de Estados e Observando Mudanças

// dynamic-state.component.ts

import { Component, OnInit } from '@angular/core';

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

@Component({

selector: 'app-dynamic-state',

templateUrl: './dynamic-state.component.html',

})

export class DynamicStateComponent implements OnInit {

userForm: FormGroup;

constructor(private fb: FormBuilder) {}

ngOnInit() {

this.buildForm();

this.observeFormChanges();

}

buildForm() {

this.userForm = this.fb.group({

username: ['', [Validators.required, Validators.minLength(3), Validators.maxLength(15)]],

isAdmin: [false],

});

}

observeFormChanges() {

this.userForm.valueChanges.subscribe(value => {

if (value.isAdmin) {

this.userForm.get('username').setValidators([Validators.required, Validators.minLength(5)]);

} else {

this.userForm.get('username').setValidators([Validators.required, Validators.minLength(3), Validators.maxLength(15)]);

}

this.userForm.get('username').updateValueAndValidity();

});

}

onSubmit() {

console.log('Formulário enviado:', this.userForm.value);

}

}

<!-- dynamic-state.component.html -->

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" class="form-container">

<label for="username">Usuário:</label>

<input type="text" id="username" formControlName="username">

<div \*ngIf="userForm.get('username').hasError('minlength') && userForm.get('username').touched">Mínimo de {{ userForm.get('isAdmin').value ? 5 : 3 }} caracteres.</div>

<label for="isAdmin">É Administrador?</label>

<input type="checkbox" id="isAdmin" formControlName="isAdmin">

<button type="submit" [disabled]="userForm.invalid">Enviar</button>

</form>

Customizando Controles e Validadores

// custom-control.component.ts

import { Component, OnInit } from '@angular/core';

import { FormBuilder, FormGroup, Validators, AbstractControl } from '@angular/forms';

@Component({

selector: 'app-custom-control',

templateUrl: './custom-control.component.html',

})

export class CustomControlComponent implements OnInit {

userForm: FormGroup;

constructor(private fb: FormBuilder) {}

ngOnInit() {

this.buildForm();

}

buildForm() {

this.userForm = this.fb.group({

username: ['', [Validators.required, this.customValidator]],

});

}

customValidator(control: AbstractControl): { [key: string]: any } | null {

const forbiddenUsername = /admin/.test(control.value);

return forbiddenUsername ? { 'forbiddenUsername': { value: control.value } } : null;

}

onSubmit() {

console.log('Formulário enviado:', this.userForm.value);

}

}

<!-- custom-control.component.html -->

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" class="form-container">

<label for="username">Usuário:</label>

<input type="text" id="username" formControlName="username">

<div \*ngIf="userForm.get('username').hasError('forbiddenUsername') && userForm.get('username').touched">Nome de usuário não permitido.</div>

<button type="submit" [disabled]="userForm.invalid">Enviar</button>

</form>

Projeto Prático - Formulário de Pesquisa Avançada (Continuação)

// search-form.component.ts

import { Component, OnInit } from '@angular/core';

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

import { SearchService } from './search.service';

import { debounceTime, switchMap } from 'rxjs/operators';

@Component({

selector: 'app-search-form',

templateUrl: './search-form.component.html',

})

export class SearchFormComponent implements OnInit {

searchForm: FormGroup;

searchResults: any[];

constructor(private fb: FormBuilder, private searchService: SearchService) {}

ngOnInit() {

this.buildForm();

this.observeFormChanges();

}

buildForm() {

this.searchForm = this.fb.group({

keyword: ['', [Validators.required]],

category: [''],

minPrice: [''],

maxPrice: [''],

});

}

observeFormChanges() {

this.searchForm.valueChanges

.pipe(

debounceTime(300),

switchMap(formValue => this.searchService.search(formValue))

)

.subscribe(results => {

this.searchResults = results;

});

}

onSubmit() {

console.log('Formulário enviado:', this.searchForm.value);

// Implementar lógica para lidar com os resultados da pesquisa

}

}

<!-- search-form.component.html -->

<form [formGroup]="searchForm" (ngSubmit)="onSubmit()" class="form-container">

<label for="keyword">Palavra-chave:</label>

<input type="text" id="keyword" formControlName="keyword" required>

<label for="category">Categoria:</label>

<select id="category" formControlName="category">

<option value="">Todas</option>

<option value="electronics">Eletrônicos</option>

<option value="clothing">Roupas</option>

<!-- ... outras opções ... -->

</select>

<label for="minPrice">Preço Mínimo:</label>

<input type="number" id="minPrice" formControlName="minPrice" min="0">

<label for="maxPrice">Preço Máximo:</label>

<input type="number" id="maxPrice" formControlName="maxPrice" min="0">

<button type="submit">Pesquisar</button>

</form>

<div \*ngIf="searchResults && searchResults.length > 0">

<h3>Resultados da Pesquisa:</h3>

<ul>

<li \*ngFor="let result of searchResults">{{ result.name }} - {{ result.price | currency }}</li>

</ul>

</div>

4. Roteamento

Configuração de rotas

Configure o sistema de roteamento para criar uma navegação em várias páginas em seu aplicativo.

Roteamento aninhado

Como criar rotas filhas para páginas aninhadas.

Passagem de parâmetros

Passe dados entre páginas usando parâmetros de rota.

Guardas de rota

Proteja rotas com guardas para controlar o acesso.

5. Formulários

Template-Driven Forms

Aprenda a criar formulários usando a abordagem baseada em templates.

Reactive Forms

Explore a criação de formulários reativos para maior controle e validação.

Validação de formulários

Como validar os dados de entrada em formulários.

Formulários dinâmicos

Crie formulários com campos dinâmicos que podem ser adicionados ou removidos dinamicamente.

6. Comunicação com o Servidor

HTTP Client

Utilize o módulo HttpClient para realizar solicitações HTTP para um servidor.

Realizando solicitações HTTP (GET, POST, PUT, DELETE)

Aprenda a realizar diferentes tipos de solicitações HTTP e manipular as respostas.

Interceptadores

Como usar interceptadores para adicionar funcionalidades globais a solicitações HTTP.

Tratamento de erros

Gerencie erros de solicitação HTTP e forneça feedback apropriado ao usuário.

7. Estado da Aplicação

Gerenciamento de estado com NgRx (opcional)

Introdução ao conceito de gerenciamento de estado com a biblioteca NgRx, que é baseada no padrão Redux.

Introdução ao Redux

Compreenda os princípios do Redux, que são fundamentais para o NgRx.

Ações, Redutores e Seletores

Como criar ações, redutores e seletores para gerenciar o estado da aplicação.

8. Testes

Testando componentes

Aprenda a escrever testes unitários para seus componentes Angular.

Testando serviços

Como testar serviços em Angular.

Testes end-to-end com Protractor (opcional)

Introdução ao teste end-to-end usando a ferramenta Protractor.

9. Melhores Práticas e Dicas

Estrutura de projeto recomendada

Organização de pastas e arquivos em um projeto Angular.

Organização de código

Mantenha seu código limpo e bem organizado seguindo as melhores práticas.

Otimização de desempenho

Dicas para melhorar o desempenho do aplicativo Angular.

Segurança

Práticas recomendadas para garantir a segurança do seu aplicativo.

10. Conclusão

Resumo dos principais conceitos