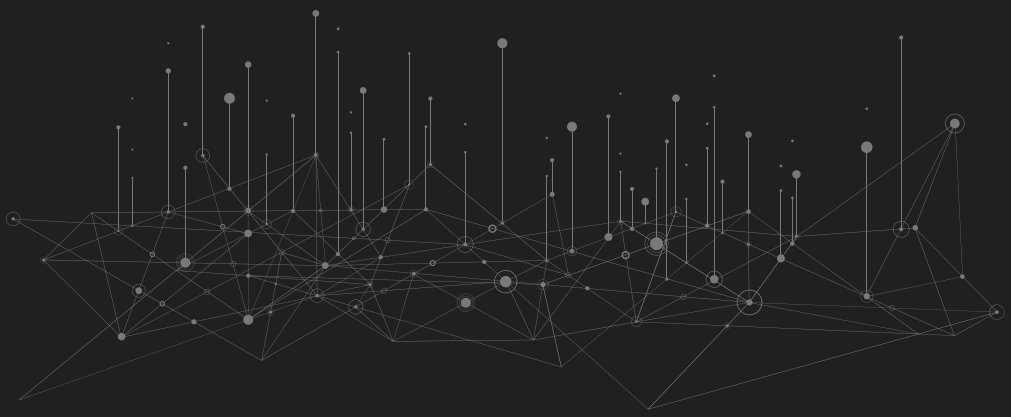
# Desmistificando o Angular

## Dos Fundamentos Teóricos à Construção de uma Aplicação Prática



Nicolas Cleik de Andrade

2025

# Agradecimentos

Este material nasceu de uma oportunidade especial, e por isso, gostaria de expressar minha mais profunda gratidão ao Professor **Josimar dos Santos**, meu mestre na disciplina de Desenvolvimento Web Front-End na **Universidade Tiradentes (UNIT)**.

Foi através do seu convite para apresentar sobre o framework Angular aos meus colegas, em 22 de outubro de 2025, que me senti motivado a aprofundar meus estudos e organizar este guia. O que começou como anotações pessoais transformou-se neste material de apoio, e isso só foi possível graças à sua confiança e incentivo.

Estou muito grato por me proporcionar esta experiência gratificante e por ser uma inspiração constante na minha jornada de aprendizado em desenvolvimento web.

**Instagram josimar**

https://www.instagram.com/prof.josimar.sts/

**LinkedIn josimar**

https://www.linkedin.com/in/josimarsts/

# Introdução: Desmistificando o Angular, Passo a Passo

Vamos ser sinceros: abrir a documentação de um framework como o Angular pela primeira vez pode ser intimidador. A gente se depara com tantos conceitos — componentes, services, data binding — que a pergunta "por onde eu começo?" é quase inevitável.

Eu sei bem como é, e foi pensando exatamente nisso que criei este material. Meu objetivo aqui não é apenas te mostrar um monte de código, mas te guiar em uma jornada lógica, transformando o complexo em algo compreensível. A ideia é que, ao final, você não só tenha construído algo, mas realmente entenda o "porquê" por trás de cada passo.

Ao concluir este guia, você terá em mãos não apenas o conhecimento teórico, mas a experiência prática de ter construído uma aplicação completa do zero. Você será capaz de:

- **Configurar um ambiente de desenvolvimento Angular** do início ao fim.
- **Entender a arquitetura** de um projeto profissional.
- **Criar e gerenciar componentes**, que são o coração do Angular.
- **Implementar um sistema de rotas** para criar uma aplicação de página única (SPA).
- **Dar vida aos seus componentes** com Data Binding (`ngModel`, `{{ }}`, `[ ]`, `( )`, `[()]`).
- **Organizar seu código** de forma limpa com Services.
- **Fazer componentes "conversarem"** entre si com `@Input` e `@Output`.
- **Salvar e carregar dados** para criar uma experiência real para o usuário.

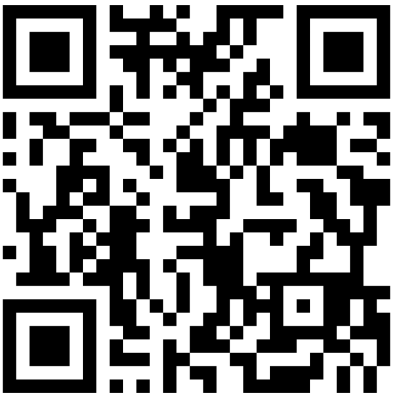
Para garantir que a gente não se perca no caminho, eu dividi nossa jornada em duas partes. Primeiro, em **"Desvendando o Angular"**, vamos construir uma base teórica sólida. Em seguida, no **"Tutorial Prático: Construindo uma To-Do List"**, vamos colocar a mão na massa e aplicar todo esse conhecimento.

Então, prepare seu café, abra seu editor de código e vamos juntos desmistificar o Angular!

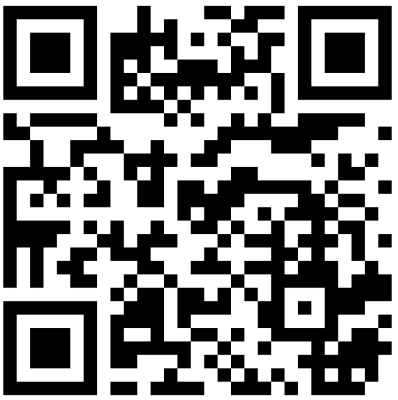
## Conecte-se com nós e Compartilhe sua Jornada

Este guia é o começo. A jornada de um dev é contínua, e eu adoraria acompanhar a sua e compartilhar mais dicas. Vamos nos conectar!

- **LinkedIn:** Para networking profissional, discussões mais aprofundadas sobre carreira e tecnologia.
    - www.linkedin.com/in/nicolascleik

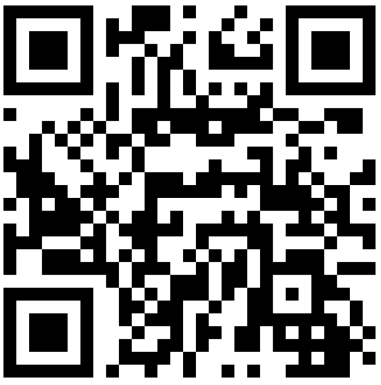


- **Instagram (@dev.cleik):** Para um conteúdo mais visual, dicas rápidas, os bastidores da minha jornada de estudos e projetos.
    - https://www.instagram.com/dev.cleik

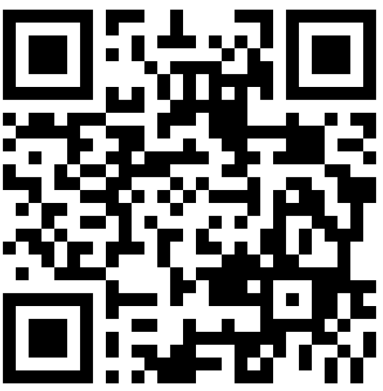


---

- **LinkedIn:** Para networking profissional, discussões mais aprofundadas sobre carreira e tecnologia.
    - https://www.linkedin.com/in/altemirfilho/



- **Instagram (@altemir.fh):** Para um conteúdo mais visual, dicas rápidas, os bastidores da minha jornada de estudos e projetos.
    - https://www.instagram.com/altemir.fh/



# Desvendando o Angular: O Guia Teórico Essencial

## 1. Requisitos e Instalação do Ambiente Angular

Antes de começar a desenvolver com Angular, é necessário preparar o seu ambiente de desenvolvimento. O processo é simples e envolve a instalação do Node.js e, em seguida, do Angular CLI.

### Passo 1: Instalar o Node.js

O Node.js é um ambiente de execução JavaScript que permite rodar JavaScript fora do navegador. Ele é um requisito fundamental, pois inclui o **npm** (Node Package Manager), que é o gerenciador de pacotes que usaremos para instalar o Angular.

- **Ação:** Baixe o instalador da versão LTS (Long Term Support) diretamente do site oficial do Node.js e siga as instruções de instalação para o seu sistema operacional.

**Verificação:** Após a instalação, abra o seu terminal (ou Prompt de Comando/PowerShell no Windows) e execute o seguinte comando para verificar se a instalação foi bem-sucedida:

```
Bash
node --version
```

- Este comando deve retornar a versão do Node.js que você acabou de instalar.

### Passo 2: Instalar o Angular CLI

O Angular CLI (Command Line Interface) é uma ferramenta de linha de comando que agiliza a criação e o gerenciamento de projetos Angular. Com ela, você pode criar projetos, componentes, serviços e muito mais com simples comandos.

**Ação:** Com o Node.js e o npm já instalados, execute o seguinte comando no terminal para instalar o Angular CLI:

```
Bash
npm install -g @angular/cli
```

- Vamos entender as partes deste comando:

- `npm`: É o **N**ode **P**ackage **M**anager, a ferramenta que gerencia as bibliotecas (pacotes) do seu projeto.
- `install`: O comando para instalar um novo pacote.
- `-g`: É uma abreviação para `--global`. Esta *flag* instrui o npm a instalar o pacote de forma **global** no seu sistema. Isso significa que você não precisará reinstalar o Angular CLI toda vez que criar um novo projeto.
- `@angular/cli`: É o nome oficial do pacote do Angular CLI.

**Verificação:** Após a conclusão da instalação, você pode verificar se o Angular CLI está funcionando corretamente com o comando:

```
Bash
ng --version
```

- Este comando exibirá um painel com a versão do Angular CLI instalada, junto com outras informações importantes sobre o seu ambiente Angular e Node.js.

## 2. Criando um Novo Projeto com a Angular CLI

Com o ambiente devidamente configurado (Node.js e Angular CLI instalados), o próximo passo é criar a estrutura da nossa aplicação. A CLI automatiza todo esse processo.

### Passo 1: Preparar o Terminal

Antes de executar qualquer comando, você precisa estar no local correto.

1. Escolha uma pasta em seu computador onde deseja que o projeto seja criado.
2. Abra seu terminal diretamente nessa pasta.
    - **Dica:** No VS Code, você pode abrir a pasta desejada e depois usar o atalho `Ctrl + '` (ou ir em `Terminal > Novo Terminal`) para abrir um terminal já no diretório certo.

### Passo 2: Executar o Comando de Criação

Com o terminal aberto na pasta correta, execute o seguinte comando:

```
Bash
ng new nome-do-projeto
```

- `ng`: É o comando que invoca a **Angular CLI**.
- `new`: É a instrução para criar uma nova aplicação.
- `nome-do-projeto`: É o nome que você dará à pasta e ao seu projeto.

### Passo 3: Responder às Perguntas da CLI

Após executar o comando, a CLI fará algumas perguntas para configurar seu projeto inicial:

**1. Qual formato de folha de estilo você gostaria de usar? (Which stylesheet format would you like to use?)** * Você pode escolher entre `CSS`, `Sass`, `Less`, entre outros. Para começar, **CSS** é a opção mais direta e recomendada.

**2. Deseja habilitar o Renderização do Lado do Servidor (SSR)? (Do you want to enable Server-Side Rendering?)** * Para esta introdução, a resposta deve ser **Não**. * Esta é uma técnica mais avançada. Ao escolher "Não", a CLI configurará o projeto como uma **SPA (Single Page Application)**, que é o padrão e o foco do nosso estudo.

Ao final, a CLI irá criar a pasta do projeto, instalar todas as dependências necessárias e gerar a estrutura de arquivos completa, pronta para ser executada.

## 3. Estrutura de um Projeto Angular

Ao criar um projeto com o comando `ng new`, a Angular CLI gera uma estrutura de pastas e arquivos muito bem organizada. Entender o propósito de cada um é fundamental para saber onde encontrar e modificar seu código. Vamos dividir a análise em três áreas principais: os arquivos de configuração na raiz, a pasta de código-fonte `src/`, e a pasta principal da aplicação `app/`.

### 3.1. Arquivos de Configuração (Raiz do Projeto)

Estes arquivos controlam o comportamento do projeto, das dependências e das ferramentas de desenvolvimento.

- `.editorconfig`: Define padrões de codificação (como indentação e espaçamento) para manter o código consistente entre diferentes editores e desenvolvedores.

- **`.gitignore`**: Um arquivo do Git que lista quais arquivos e pastas devem ser ignorados pelo controle de versão. Isso inclui:
    - Arquivos de cache e build
    - Arquivos específicos do ambiente de desenvolvimento (como do VS Code)
- **`angular.json`**: O principal arquivo de configuração da Angular CLI. É aqui que você define como o projeto deve ser construído (`build`), servido (`serve`), testado e otimizado.
- **`package.json`**: Arquivo padrão do ecossistema Node.js. Ele lista todas as bibliotecas externas (**dependências**) que seu projeto utiliza e define **scripts** para automatizar tarefas, como `npm run start`.
- **`package-lock.json`**: Gerado automaticamente, este arquivo "trava" a versão exata de cada dependência instalada, garantindo que o projeto funcione da mesma forma em qualquer máquina.
- **`tsconfig.json`**: Arquivo de configuração do TypeScript. Ele define as regras de como o código `.ts` deve ser compilado.
    - **`tsconfig.app.json`**: Configurações do TypeScript específicas para a aplicação.
    - **`tsconfig.spec.json`**: Configurações do TypeScript aplicadas apenas quando os testes unitários são executados.

## 3.2. A Pasta `src`: O Coração do Código-Fonte

A pasta `src` (source) é onde todo o código que você escreve para a sua aplicação reside.

- **`public/` (ou `assets/`)**: Contém arquivos estáticos que serão disponibilizados publicamente, como imagens, fontes e o ícone da aba do navegador, o **`favicon.ico`**.
- **`styles.css`**: Arquivo para estilização CSS global, que afeta a aparência de toda a aplicação.

- **`main.ts`**: O ponto de entrada (`entrypoint`) da aplicação. É o primeiro arquivo a ser executado e é responsável por "iniciar" (ou *bootstrapping*) o Angular.
    - *(**Nota:** O termo "bootstrap" aqui se refere ao processo de inicialização e não tem relação com o framework de CSS Bootstrap.)*
- **`index.html`**: É a única página HTML que o navegador realmente carrega. Sendo uma **SPA (Single Page Application)**, o Angular manipula o conteúdo desta página dinamicamente com JavaScript.
    - Dentro do `<body>`, você encontrará a tag `<app-root></app-root>`. Este é o "ponto de encaixe" onde o componente principal da sua aplicação será renderizado.

### 3.3. A Pasta `app`: Onde a Aplicação Vive

Dentro de `src/`, a pasta `app/` é o diretório central onde você organizará seus componentes, serviços e rotas.

- **O Componente Principal (`AppComponent`)**: Por padrão, o projeto já vem com o `AppComponent`, que é o primeiro componente a ser carregado. Ele é composto por 4 arquivos (`.html`, `.css`, `.ts`, `.spec.ts`) e seu seletor, `app-root`, é o que o conecta ao `index.html`.
- **`app.config.ts`**: Arquivo de configuração da aplicação Angular. É aqui que você "ativa" funcionalidades, como o sistema de rotas através do `provideRouter`.

**`app.routes.ts`**: O "mapa" ou "GPS" da sua aplicação. Neste arquivo, você define quais componentes devem ser renderizados com base na URL que o usuário acessa, criando as "páginas virtuais" da sua SPA.

```
TypeScript
// Exemplo de mapeamento de rota
export const routes: Routes = [
  {
    path: "home", // Quando a URL for /home...
    component: HomeComponent // ...renderize o HomeComponent.
```

```
  }
];
```

## 4. Executando o Projeto Angular

Depois que a estrutura do projeto foi criada, o próximo passo é iniciar o servidor de desenvolvimento para visualizar sua aplicação no navegador.

1. **Abra o Terminal:** Certifique-se de que seu terminal esteja aberto na pasta raiz do projeto que você acabou de criar.
    - Caso não esteja no caminho correto, você pode usar o comando:

```
Bash
cd nome-do-projeto
```

**Execute o Comando de Início:** Digite o seguinte comando:

```
Bash
npm run start
```

2. **Entendendo o Comando:**
    - Este comando executa o script chamado `start` que está definido no seu arquivo `package.json`.
    - Por padrão, o script `start` aciona o comando `ng serve` da Angular CLI.
    - O `ng serve` compila a aplicação, inicia um servidor de desenvolvimento e fica observando os arquivos do projeto em busca de alterações.
3. **Acessando a Aplicação:**
    - Após a compilação, o terminal informará o endereço local onde a aplicação está rodando, geralmente `http://localhost:4200/`.
    - O conteúdo inicial que você vê na tela é renderizado a partir do `app.component`.
4. **Live Reload:**

- Uma das grandes vantagens do servidor de desenvolvimento do Angular é o **Live Reload**. Qualquer alteração salva nos arquivos do projeto fará com que a página no navegador seja atualizada automaticamente.
- **Observação Importante:** Toda a interface da sua aplicação é renderizada dinamicamente pelo JavaScript. Se você desabilitar o JavaScript no navegador, a página ficará em branco, pois o conteúdo não está estaticamente no `index.html`.

## 5. Criando Novos Componentes

Componentes são os blocos de construção fundamentais de uma aplicação Angular. Para criá-los, utilizamos novamente a Angular CLI para garantir a padronização e agilidade.

### 5.1. O Comando `generate component`

No terminal, na raiz do seu projeto, utilize o comando `ng generate component`.

**Forma Completa:**

```
Bash
ng generate component components/home
```

**Forma Abreviada (mais comum):**

```
Bash
ng g c components/home
```

### 5.2. Boas Práticas de Organização

- **`components/home`**: Ao usar este caminho no comando, a CLI irá:
    1. Criar uma nova pasta chamada `components` dentro de `src/app/` (se ela não existir).
    2. Criar uma pasta `home` dentro de `components/`.

3. Gerar os quatro arquivos do componente (`.html`, `.css`, `.ts`, `.spec.ts`) dentro da pasta `home`.

- **Padrão da Comunidade:** É uma convenção e boa prática organizar todos os componentes reutilizáveis dentro de uma pasta `components`, localizada em `src/app/`. Isso mantém o projeto limpo e escalável.

## 6. A Estrutura dos Componentes

Cada componente gerado pela Angular CLI é um bloco de construção autossuficiente, composto por quatro arquivos que trabalham em conjunto. Cada um possui uma responsabilidade bem definida.

### 6.1. A Anatomia de um Componente: Os 4 Arquivos

1. **`.html` (O Template):**
    - Contém a estrutura HTML e a parte visual do componente. É aqui que você define as tags e a disposição dos elementos na tela.
2. **`.css` (O Estilo):**
    - Contém as regras de estilo (CSS) que se aplicam **exclusivamente** a este componente. O Angular encapsula o estilo, garantindo que o CSS de um componente não "vaze" e afete outros.
3. **`.spec.ts` (Os Testes):**
    - Arquivo para escrever testes unitários. Por padrão, a CLI já gera um teste básico que verifica se o componente está sendo criado corretamente.
4. **`.ts` (A Lógica):**
    - É o "cérebro" do componente. Este arquivo TypeScript contém a classe que define a lógica, os dados (propriedades) e os comportamentos (métodos) do componente.

### 6.2. O Decorador `@Component`: O Cérebro da Operação

No arquivo `.ts`, o que transforma uma simples classe em um componente Angular é o **decorador `@Component`**. Ele adiciona "metadados" essenciais que dizem ao Angular como o componente deve funcionar.

```
TypeScript
@Component({
  selector: 'app-home',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './home.component.html',
  styleUrl: './home.component.css'
})
export class HomeComponent { }
```

Vamos analisar suas propriedades principais:

- **`selector: 'app-home'`**:
    - Define o nome da "tag HTML" personalizada que usaremos para invocar este componente em outros templates. Por exemplo, o componente principal do projeto tem o seletor `'app-root'`, que é usado no `index.html`.
- **`standalone: true`**:
    - Esta é a abordagem moderna e padrão do Angular. Significa que o componente é autossuficiente e gerencia suas próprias dependências, sem a necessidade de ser declarado em um `NgModule` (a forma mais antiga de organização).
- **`imports: [ ... ]`**:
    - Como o componente é `standalone`, é neste array que declaramos tudo o que ele precisa "importar" para funcionar. Isso pode incluir outros componentes, diretivas (`*ngIf`, `@for`) ou módulos do Angular, como o `CommonModule` ou `RouterModule`.
- **`templateUrl: './home.component.html'`**:

    - Aponta para o arquivo de template (HTML) associado a este componente.

- `styleUrl: './home.component.css'`:
    - Aponta para o arquivo de estilo (CSS) associado a este componente.

## 7. Utilizando Componentes Através de Rotas

Depois de criar um componente, como fazemos para exibi-lo como uma página na nossa aplicação? A maneira principal é através do sistema de **Rotas** do Angular. O processo envolve dizer ao Angular qual componente carregar com base na URL que o usuário acessa.

### 7.1. Passo 1: Mapear a Rota no `app.routes.ts`

Primeiro, precisamos registrar nosso novo componente no "mapa" de rotas da aplicação.

1. Abra o arquivo `app.routes.ts`.
2. Adicione um objeto para a nova rota dentro do array `routes`. Este objeto associa um `path` (caminho da URL) ao `component` que deve ser renderizado.

**Exemplo: Configurando a Rota Raiz**

Para fazer o `HomeComponent` aparecer quando o usuário acessar a página inicial do site (ex: `http://localhost:4200/`), usamos um `path` vazio.

```
TypeScript
// No arquivo app.routes.ts
export const routes: Routes = [
  {
    path: "", // O path vazio "" representa a rota raiz
    component: HomeComponent
  }
];
```

Você pode adicionar quantas rotas precisar:

```
TypeScript
export const routes: Routes = [
  { path: "", component: HomeComponent },
   { path: "home", component: HomeComponent }, // Também carrega o
HomeComponent
  { path: "contato", component: ContatoComponent } // Uma nova "página"
];
```

*Se o usuário tentar acessar uma URL não mapeada, o Angular mostrará um erro no console do navegador.*

## 7.2. Passo 2: Definir o Ponto de Renderização com `<router-outlet>`

Agora que o Angular sabe qual componente carregar, precisamos dizer a ele **onde** na tela esse componente deve aparecer. Para isso, usamos uma tag especial.

1. Abra o arquivo `app.component.html` (o template do seu componente principal).
2. Adicione a tag `<router-outlet></router-outlet>`.

**O que o `<router-outlet>` faz?**

Ele funciona como um **espaço reservado** ou um "palco". O Angular irá renderizar dinamicamente o componente correspondente à rota ativa dentro desta tag. Isso permite que você tenha elementos fixos na página (como um cabeçalho e rodapé) e um conteúdo central que muda conforme a navegação.

```
HTML
<header>Meu Cabeçalho Fixo</header>

<main>
  <router-outlet></router-outlet>
</main>

<footer>Meu Rodapé Fixo</footer>
```

**Observação crucial:** Se você remover a tag `<router-outlet>` do seu template, o sistema de rotas não terá onde renderizar os componentes, e suas "páginas" não aparecerão, resultando em uma área em branco.

## 8. Estados: Dando "Memória" e Lógica aos Componentes

Um componente não é apenas uma estrutura visual estática; ele precisa ter uma "memória" para guardar informações dinâmicas, como dados de um formulário, se um menu está aberto ou se o usuário está logado. Essas informações são chamadas de **estados**.

Nós definimos os estados como atributos (propriedades) dentro da classe do componente, no arquivo `.ts`.

### 8.1. A Importância da Tipagem com TypeScript

É fundamental lembrar que o Angular utiliza TypeScript, que possui um sistema de **tipagem estática**, diferente do JavaScript puro. Isso significa que, ao declarar uma variável, definimos o tipo de dado que ela pode conter (texto, número, booleano), e esse tipo não pode ser alterado depois.

**JavaScript (Tipagem Dinâmica):**

```
JavaScript
let nome = "Nicolas"; // Começa como string
nome = 2025;          // Depois vira um número. Isso é permitido.
```

**TypeScript (Tipagem Estática):**

```
TypeScript
let nome: string = "Nicolas"; // O tipo é definido como string
nome = 2025;                  // ERRO! Não se pode atribuir um número a
uma string.
```

Essa característica do TypeScript torna o código mais seguro, legível e menos propenso a bugs.

### 8.2. Declarando Estados e Métodos

Dentro da classe do componente, podemos declarar os estados e os métodos (funções) que irão manipulá-los.

```
TypeScript
// Exemplo no home.component.ts
export class HomeComponent {
  // Declarando um estado (atributo) 'meu_Booleano' como booleano.
  meu_Booleano = false;

  // Criando um método para atualizar o estado.
  // O parâmetro 'valor' é explicitamente tipado como boolean.
  atualizaBoleano(valor: boolean) {
    this.meu_Booleano = valor;
  }
}
```

A tipagem garante que o método `atualizaBoleano` só aceite valores `true` ou `false`, prevenindo erros. Se o tipo do parâmetro não fosse especificado, o TypeScript apontaria um erro.

## 8.3. Uma Nota de Cuidado: O Tipo `any`

O TypeScript oferece um tipo especial chamado `any`, que serve como uma "válvula de escape" e desabilita a verificação de tipos para uma variável.

```
TypeScript
// Exemplo NÃO recomendado
atualizaBoleano(valor: any) { // 'valor' agora pode ser qualquer coisa
  this.meu_Booleano = valor; // Risco de atribuir um tipo incorreto, como
"Nicolas".
}
```

**Recomendação:** Evite usar `any`. Aderir à tipagem estrita torna o código mais fácil de entender e manter, pois fica claro quais tipos de dados são esperados em cada parte da aplicação.

## 9. Renderizando Dados Dinâmicos (Data Binding)

Agora que temos estados (dados) e lógica na nossa classe (`.ts`), como fazemos para exibi-los e interagir com eles no nosso template (`.html`)? A ponte que conecta esses dois mundos é o **Data Binding**.

É o mecanismo do Angular que sincroniza automaticamente os dados entre a classe (a lógica) e o template (a visualização), eliminando a necessidade de manipular o DOM manualmente.

### 9.1. Interpolação `{{ }}`: Exibindo Dados no Template

A forma mais simples de exibir o valor de uma propriedade da sua classe no HTML é usando a sintaxe de chaves duplas, chamada de **Interpolação**.

- **O que faz?** Pega o valor de uma propriedade no arquivo `.ts` e o transforma em texto no `.html`.
- **Quando usar?** Ideal para mostrar valores dentro de tags como `<h1>`, `<p>`, `<span>`, etc.

**Exemplo:**

```
TypeScript
// No home.component.ts
export class HomeComponent {
  nome = "Nicolas";
}
```

```
HTML
<p>Olá, {{ nome }}</p>
```

### 9.2. Property Binding `[ ]`: Controlando Atributos HTML

Quando precisamos alterar dinamicamente um atributo de uma tag HTML (como `id`, `disabled`, `src` de uma imagem), usamos o **Property Binding**. A sintaxe é envolver o atributo em colchetes `[ ]`.

- **O que faz?** Conecta o valor de uma propriedade da sua classe a um atributo de um elemento HTML.

**Exemplo:**

```
TypeScript
// No home.component.ts
export class HomeComponent {
  idButton = "meu-botao-principal";
}
```

```
HTML
// No home.component.html
<button [id]="idButton">Meu Botão</button>
```

- **Observação:** Essa sintaxe funciona para qualquer atributo, seja ele padrão do HTML (como `id`) ou um atributo customizado (como `[attr.meu-atributo]="valor"`).

## 9.3. Event Binding `( )`: Reagindo a Eventos do Usuário

Para que nosso componente reaja a ações do usuário, como um clique de mouse ou a digitação em um campo, usamos o **Event Binding**. A sintaxe é envolver o nome do evento do DOM em parênteses `( )`.

- **O que faz?** Executa um método da sua classe sempre que um evento específico acontece no HTML.
- **Importante:** Usamos os nomes dos eventos do DOM, mas sem o prefixo "on". Portanto, `onclick` vira `(click)`, `onmouseover` vira `(mouseover)`, etc.

**Exemplo:**

```
TypeScript
// No home.component.ts
export class HomeComponent {
  submit() {
    console.log("Hello World");
  }
}
```

```
HTML
// No home.component.html
<button (click)="submit()">Clique Aqui</button>
```

### 9.4. Two-Way Data Binding [()]: Ligação Bidirecional (Classe <-> Template)

E se quisermos uma **ligação nos dois sentidos**? Ou seja, quando a propriedade na classe muda, o template atualiza, **E** quando o valor no template muda (geralmente em um input), a propriedade na classe também é atualizada automaticamente? Para isso, usamos o **Two-Way Data Binding**, com a sintaxe `[()]` (apelidada de "banana in a box").

- **O que faz?** Combina Property Binding `[ ]` (Classe -> Template) e Event Binding `( )` (Template -> Classe) em uma única diretiva.
- **Uso Principal:** É mais comumente usado com a diretiva `ngModel` em elementos de formulário (`<input>`, `<select>`, `<textarea>`) para sincronizar o valor do campo com uma propriedade na classe. Requer a importação do `FormsModule`.

**Exemplo com `[(ngModel)]`:**

```
TypeScript
// No home.component.ts
import { FormsModule } from '@angular/forms'; // Necessário importar

@Component({
```

```
  selector: 'app-exemplo',
  standalone: true,
  imports: [FormsModule], // Adicionar FormsModule
  // ...
})
export class ExemploComponent {
  nomeUsuario: string = '';
}
```

```
HTML
// No home.component.html
<input [(ngModel)]="nomeUsuario" placeholder="Digite seu nome">
<p>Nome digitado: {{ nomeUsuario }}</p>
```

**Como funciona `[(ngModel)]`:**

1. `[ngModel]="nomeUsuario"`: O Property Binding define o valor inicial do input com o valor de `nomeUsuario`.
2. `(ngModelChange)="nomeUsuario = $event"`: O Event Binding (implícito) escuta por mudanças no input (`$event` contém o novo valor) e atualiza a propriedade `nomeUsuario`. A sintaxe `[()]` é um açúcar sintático para essa combinação.

**Uso em Componentes Customizados:** Você também pode implementar Two-Way Binding em seus próprios componentes (@Input + @Output com nome específico), mas `[(ngModel)]` é o caso de uso mais frequente.

## 9.5. Capturando o Contexto do Evento com `$event`

Se você precisar de informações sobre o evento em si (como os dados de um formulário, a posição do mouse ou qual tecla foi pressionada), pode passá-lo para seu método usando a palavra-chave especial `$event`.

**Exemplo:**

```
HTML
<button (click)="submit($event)">Clique e veja o evento no
```

```
console</button>
```

```
TypeScript
// No home.component.ts
submit(event: any) {
    console.log(event);  // Irá exibir o objeto completo do evento no
console
}
```

## 10. Loops e Condicionais: Controlando a Estrutura do HTML

Além de exibir dados, o Angular nos permite adicionar lógica diretamente no template para controlar *quais* elementos são renderizados e *quantas vezes*. Essas são as **diretivas estruturais**, pois elas manipulam a estrutura do DOM.

## 10.1. Renderização Condicional

Para exibir ou ocultar blocos de HTML com base em uma condição, usamos a sintaxe `@if`.

**1. Declarando o Estado no Componente (.ts)**

Primeiro, definimos as propriedades que controlarão a lógica no arquivo `home.component.ts`.

```
TypeScript
export class HomeComponent {
  nome = "Nicolas";
  deveMostrarTitulo = false;
}
```

**2. Usando a Sintaxe `@if` no Template (.html)**

Nas versões mais recentes do Angular, podemos usar blocos de controle de fluxo diretamente no HTML.

```
HTML
@if (deveMostrarTitulo) {
  <h1>Meu titulo no HTML</h1>
} @else if (nome === "Nicolas") {
  <h1>Meu titulo para Nicolas</h1>
} @else {
  <h1>A condição foi falsa</h1>
}
```

**Nota sobre Versões Antigas (`*ngIf`)**

Em projetos mais antigos, a sintaxe era diferente, usando a diretiva `*ngIf`.

- **Sintaxe:** `<h1 *ngIf="deveMostrarTitulo">Meu titulo no HTML</h1>`
- **Requisito:** Essa abordagem exigia a importação do `CommonModule` no array `imports` do decorador `@Component`.

## 10.2. Renderização de Listas com Loops

Para renderizar um bloco de HTML para cada item de uma lista (array), usamos a sintaxe `@for`.

**1. Declarando a Lista no Componente (.ts)**

Definimos o array que será percorrido.

```
TypeScript
export class HomeComponent {
  // Lista de strings simples
  listaDeItens = ["Livro", "Computador", "Tablet", "Monitor"];
}
```

**2. Usando a Sintaxe `@for` no Template (.html)**

```
HTML
```

```
@for (item of listaDeItens; track item) {
  <p>{{ item }}</p>
} @empty {
  <p>Não há itens na lista.</p>
}
```

**A Importância do `track`**

A propriedade `track` é **obrigatória** e fundamental para a performance. Ela ajuda o Angular a identificar cada elemento da lista de forma única. Quando a lista é modificada (itens são adicionados, removidos ou reordenados), o `track` permite que o Angular atualize a interface de forma otimizada, alterando apenas o que é necessário.

- **Para listas de valores simples (strings, numbers):** Use o próprio item como identificador: `track item`.

**Para listas de objetos:** Use uma propriedade única do objeto, como `id`.

```
TypeScript
// No home.component.ts
export class HomeComponent {
  listaDeItens = [{id: "1"}, {id: "2"}, {id: "3"}];
}
```

```
HTML
@for (item of listaDeItens; track item.id) {
  <p>Item com ID: {{ item.id }}</p>
}
```

## 11. Services: Compartilhando Lógica entre Componentes no Angular

No Angular, os *Services* (Serviços) são utilizados para compartilhar lógica, dados ou funcionalidades entre diferentes componentes.

Imagine que você tem vários componentes e precisa que todos executem a mesma função, como enviar dados para um servidor. Em vez de duplicar o código dessa função em cada componente (o que é uma má prática de programação), você centraliza essa lógica em um único serviço. Dessa forma, qualquer componente que precisar dessa funcionalidade pode simplesmente "injetar" e usar o serviço.

## Criando um Service

A maneira mais fácil de criar um serviço é usando o Angular CLI no terminal.

```
Bash
ng generate service services/enviaFormulario
```

Vamos entender o comando:

- `ng generate service`: Pede ao Angular CLI para gerar um novo arquivo de serviço.
- `services/enviaFormulario`: Define que o serviço deve ser criado dentro de uma pasta chamada `services` e o nome do arquivo será `enviaFormulario.service.ts`.

É possível simplificar o comando usando abreviações:

- `generate` pode ser abreviado para `g`.
- `service` pode ser abreviado para `s`.

O comando simplificado fica assim:

```
Bash
ng g s services/enviaFormulario
```

## Estrutura de um Service

Após a criação, você encontrará o arquivo `envia-formulario.service.ts` dentro da pasta `services`.

**services/envia-formulario.service.ts**

Dentro da classe do serviço, você pode adicionar os métodos que serão compartilhados.

```
TypeScript
import { Injectable } from '@angular/core';

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

  constructor() { }

  enviarInformacaoParaBackend(info: string) {
    console.log("Enviando para o backend:", info);
  }
}
```

- `@Injectable()`: Este é um "decorator" que marca a classe como um serviço que pode ser injetado em outros componentes. É semelhante ao `@Component` que decora as classes de componentes.
- `providedIn: 'root'`: Esta configuração disponibiliza uma única instância do serviço para toda a aplicação (injeção *singleton*). Isso significa que todos os componentes compartilharão o mesmo serviço.

## Utilizando um Service em um Componente

Para usar o serviço, você precisa injetá-lo no construtor do componente desejado.

**home.component.ts**

```
TypeScript
import { Component, inject } from '@angular/core';
import{             EnviaFormularioService            }            from
```

```
'../services/envia-formulario.service'; // Importe o serviço

@Component({
  // ... metadados do componente
})
export class HomeComponent {

  // Injeção do serviço
  private enviaFormularioService = inject(EnviaFormularioService);

  submit() {
    // Agora você pode chamar os métodos do serviço
        this.enviaFormularioService.enviarInformacaoParaBackend("enviando
informação");
  }
}
```

- Por padrão da comunidade, nos declaramos essa função como `private`, no caso sendo uma atributo privado restrito apenas naquela classe.

## 12. Como Enviar Informações de um Componente para o Outro

## @Input: Comunicação do Componente Pai para o Filho

O decorator `@Input()` permite que um componente filho receba dados de seu componente pai.

**Cenário:** Passar um valor do `app.component` (pai) para o `home.component` (filho).

### 1. Preparando o Componente Filho (`home.component.ts`)

No componente filho, declare uma propriedade com o decorator `@Input()` para indicar que ela receberá um valor de fora.

```
TypeScript
import { Component, Input } from '@angular/core';
```

```
@Component({
  // ...
})
export class HomeComponent {
  @Input() minhaPropsDeFora!: string;
   // A '!' informa ao TypeScript que esta propriedade será inicializada
pelo componente pai.
}
```

**2. Passando o Valor no Componente Pai (`app.component.html`)**

No template do componente pai, ao usar a tag do componente filho (`<app-home>`), passe o valor para a propriedade usando *property binding* `[ ]`.

```
HTML
<app-home minhaPropsDeFora="'um valor aleatorio'"></app-home>
```

**3. Exibindo o Valor no Filho (`home.component.html`)**

Agora você pode usar a propriedade no template do componente filho para exibir o valor recebido.

```
HTML
<h1>{{ minhaPropsDeFora }}</h1>
```

**Observação (Aliasing):** Você pode dar um "apelido" para a sua propriedade de `@Input` para tornar o nome no template do pai mais limpo.

`home.component.ts`

```
TypeScript
export class HomeComponent {
  @Input("name") teste!: string; // O nome no template do pai será "name"
```

```
}
```

**home.component.html**

```
HTML
<h1>{{ teste }}</h1> ```
```

**app.component.html**

```
HTML
<app-home [name]="'um valor aleatorio'"></app-home> ```
```

## @Output: Comunicação do Componente Filho para o Pai

O decorator `@Output()` permite que um componente Filho **emita um evento** com dados para o seu componente Pai. É a forma do Filho "avisar" o Pai que algo aconteceu e, opcionalmente, enviar dados junto.

- **Cenário:** O `home.component` (Filho) precisa enviar uma informação (o valor da variável `nome`) para o `app.component` (Pai) quando um botão for clicado.

### 1. Preparando o Componente Filho (`home.component.ts` e `.html`)

- **No arquivo `.ts`:** Crie uma propriedade com `@Output()` inicializada como um `new EventEmitter()`. Este será o "emissor" do evento. Defina também o método que chamará a função `.emit()` para disparar o evento.

```
Typescript

// No home.component.ts
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  // ... metadados
})
export class HomeComponent {
```

```
  nome = "Nicolas"; // Dado que o filho quer enviar

  // O EventEmitter<string> indica que emitirá um evento com um dado do
tipo 'string'
  @Output() emitindoValorName = new EventEmitter<string>();

  // Método que será chamado pelo botão no HTML do filho
  enviarNomeParaPai() {
    // Emite o valor da propriedade 'nome' através do EventEmitter
    this.emitindoValorName.emit(this.nome);
  }
}
```

- **No arquivo `.html` do Filho (`home.component.html`):** Adicione um elemento (como um botão) que, ao ser interagido (ex: `click`), chame o método que dispara o evento.

```
<button (click)="enviarNomeParaPai()">Enviar Nome para o Pai</button>
```

**2. Escutando o Evento no Pai (`app.component.html`)**

No template do componente Pai, use a sintaxe de **Event Binding** `( )` na tag do componente Filho. O nome do evento é o mesmo da propriedade `@Output` (`emitindoValorName`). Quando o evento for "ouvido", chame um método do Pai (ex: `logar`), passando a variável especial `$event` (que contém o dado emitido pelo Filho).

```
HTML
<app-home (emitindoValorName)="logar($event)"></app-home>
```

**3. Recebendo e Processando o Dado no Pai (`app.component.ts`)**

No componente Pai, crie o método (`logar`) que foi definido no template para receber e processar o dado vindo do Filho.

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

```
@Component({
  // ...
})
export class AppComponent {

  logar(eventoRecebido: string) {
    console.log("Evento recebido do filho: ", eventoRecebido);
  }
}
```

**Analisando o fluxo:**

1. O usuário clica no botão dentro do `home.component.html`.
2. O `(click)` chama o método `enviarNomeParaPai()` no `home.component.ts`.
3. O método `enviarNomeParaPai()` usa o `emitindoValorName.emit(this.nome)` para disparar um evento contendo o valor "Nicolas".
4. O `app.component.html`, que estava "escutando" por `(emitindoValorName)`, detecta o evento e chama seu próprio método `logar()`, passando o valor emitido (`$event`).
5. O método `logar()` no `app.component.ts` recebe a string "Nicolas" e a exibe no console.

## Debugando sua Aplicação Angular (O que fazer quando algo dá errado?)

Nenhum desenvolvedor, nem mesmo o mais experiente, escreve código perfeito na primeira tentativa. Encontrar e corrigir erros (um processo chamado "debugging" ou "depuração") é uma habilidade fundamental. Se sua aplicação não está se comportando como o esperado, aqui estão as ferramentas essenciais para investigar.

### 1.1 A Ferramenta Mais Simples: `console.log()`

A forma mais rápida e direta de entender o que está acontecendo no seu código é usar o `console.log()`. Você pode usá-lo dentro de qualquer método no seu arquivo `.ts` para imprimir o valor de uma variável no console do navegador.

- **Não sabe o que está sendo passado para uma função?** Use `console.log()`.
- **Não tem certeza se um `if` está sendo executado?** Coloque um `console.log()` dentro dele.
- **Quer ver como está sua lista de tarefas após adicionar um item?** Use `console.log()` no final do método `adicionarTarefa()`.

**Exemplo:**

```
TypeScript
deletarTarefa(id: number){
  console.log("Tentando deletar a tarefa com o ID:", id);
   this.listaDeTarefas = this.listaDeTarefas.filter(item => item.id !==
id);
  console.log("Estado da lista após a deleção:", this.listaDeTarefas);
  this.salvarTarefas();
}
```

### 1.2 As Ferramentas de Desenvolvedor do Navegador (F12)

Seu navegador (Chrome, Edge, Firefox) possui um conjunto de ferramentas poderosas para desenvolvedores, geralmente acessíveis pressionando a tecla **F12**.

- **Aba "Console":** É aqui que todos os seus `console.log()` aparecerão, juntamente com qualquer mensagem de erro que o Angular ou o JavaScript gerarem. **Sempre verifique o console primeiro se algo não funcionar!**

# Mini-exercícios para praticar a lógica com typescript

Nesta seção, vamos colocar em prática os conceitos de estados e interações básicas dentro de um componente Angular. Cada exercício focará em como usar propriedades da classe TypeScript e métodos para criar interfaces dinâmicas.

## Estados e Interação Básica (Dentro de Um Componente)

### Exercício 1: Contador Simples

**Objetivo:** Criar um componente simples que mantenha um contador numérico. Cada vez que um botão for clicado, o valor do contador deve aumentar em 1 e ser exibido na tela.

**Conceitos Praticados:**

- Declaração de uma propriedade (estado) na classe do componente.
- Criação de um método para modificar essa propriedade.
- Uso de **Event Binding** `(click)` para chamar o método a partir do template.
- Uso de **Interpolação** `{{ }}` para exibir o valor da propriedade no template.

**Código:**

**exercicios-estados.component.ts**

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

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  contador: number = 0;

  incrementar() {
    this.contador++;
```

```
  }
}
```

**exercicios-estados.component.html**

```
<p>Contador: {{ contador }}</p>
<button (click)="incrementar()">Incrementar</button>
```

**Explicação do Funcionamento:**

1. A propriedade `contador` é criada na classe TypeScript e começa com o valor `0`.
2. O método `incrementar()` simplesmente adiciona 1 ao valor atual de `contador` toda vez que é chamado.
3. No HTML, a interpolação `{{ contador }}` exibe o valor atual da propriedade `contador`.
4. O botão possui um Event Binding `(click)="incrementar()"`. Isso significa que, a cada clique, o método `incrementar()` da classe é executado.
5. Quando `incrementar()` atualiza o valor de `contador`, o Angular automaticamente detecta essa mudança e atualiza o valor exibido dentro do parágrafo `<p>` na tela.

## Exercício 2: Exibir Input

**Objetivo:** Criar um componente com um campo de input e um botão. O texto digitado pelo usuário no input deve ser armazenado em uma variável e, quando o botão for clicado, esse texto deve ser exibido em um parágrafo na tela.

**Conceitos Praticados:**

- Declaração de múltiplas propriedades (estados) na classe (`textoInput`, `textoExibido`).
- Uso de **Two-Way Data Binding** `[(ngModel)]` para conectar um input HTML a uma propriedade TypeScript.
- Necessidade de importar o `FormsModule` para usar `[(ngModel)]`.
- Criação de um método para transferir o valor de uma propriedade para outra (`mostrarTexto`).
- Uso de **Event Binding** `(click)` para acionar o método.

- Uso de **Interpolação** `{{ }}` para exibir o resultado.

**Código:**

`exercicios-estados.component.ts`

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  textoInput: string = '';
  textoExibido: string = '';

  mostrarTexto() {
    this.textoExibido = this.textoInput;
  }
}
```

`exercicios-estados.component.html`

```
<input [(ngModel)]="textoInput" placeholder="Digite algo..." />
<button (click)="mostrarTexto()">Mostrar Texto</button>

<p>{{ textoExibido }}</p>
```

**Explicação do Funcionamento:**

1. Duas propriedades são declaradas na classe: `textoInput` (para o valor do campo) e `textoExibido` (para o parágrafo). Ambas começam como strings vazias.

2. O `FormsModule` é importado no componente para habilitar o uso de `[(ngModel)]`.

3. No HTML, o `[(ngModel)]="textoInput"` cria uma ligação bidirecional:
    - Qualquer coisa digitada no `<input>` atualiza automaticamente a propriedade `textoInput`.
    - Se `textoInput` fosse alterada na classe, o valor do `<input>` também mudaria (embora não estejamos fazendo isso neste exemplo).
4. O botão tem um Event Binding `(click)="mostrarTexto()"`.
5. Quando o botão é clicado, o método `mostrarTexto()` é executado.
6. Dentro de `mostrarTexto()`, o valor atual da propriedade `textoInput` (que contém o que foi digitado) é copiado para a propriedade `textoExibido`.
7. O Angular detecta a mudança em `textoExibido` e atualiza o conteúdo do parágrafo `<p>{{ textoExibido }}</p>` na tela.

## Exercício 3: Calculadora Básica (Soma)

**Objetivo:** Criar um componente que permita ao usuário inserir dois números em campos de input, clicar em um botão para somá-los e exibir o resultado na tela. O resultado só deve aparecer após o cálculo ser realizado.

**Conceitos Praticados:**

- Declaração de propriedades numéricas (`numero1`, `numero2`, `resultado`) e booleanas (`mostrarResultado`).
- Uso de **Two-Way Data Binding** `[(ngModel)]` com inputs do tipo `number`.
- Necessidade de importar `FormsModule`.
- Criação de um método para realizar o cálculo (`somar`).
- Uso do operador unário `+` para garantir a conversão para número antes da soma.
- Uso de **Event Binding** `(click)` para acionar o método.
- Uso da diretiva estrutural `@if` para controlar a visibilidade do resultado.
- Uso de **Interpolação** `{{ }}` para exibir o resultado.
- Necessidade de importar `CommonModule` para usar `@if`.

**Código:**

**`exercicios-estados.component.ts`**

```ts
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  numero1: number = 0;
  numero2: number = 0;
  resultado: number = 0;
  mostrarResultado: boolean = false;

  somar() {
    if (this.numero1 == 0 && this.numero2 == 0) {
      this.mostrarResultado = false;
    } else {
      this.resultado = +this.numero1 + +this.numero2;
      this.mostrarResultado = true;
    }
  }
}
```

**exercicios-estados.component.html**

```html
<input type="number" [(ngModel)]="numero1" />
<input type="number" [(ngModel)]="numero2" />
<button (click)="somar()">Somar</button>

@if (mostrarResultado) {
  <p>Resultado: {{ resultado }}</p>
}
```

**Explicação do Funcionamento:**

1. As propriedades `numero1`, `numero2` e `resultado` são inicializadas com `0`. `mostrarResultado` começa como `false`.
2. Os `FormsModule` e `CommonModule` são importados para usar `[(ngModel)]` e `@if`, respectivamente.
3. Os inputs no HTML usam `[(ngModel)]` para ligar seus valores às propriedades `numero1` e `numero2`.
4. Quando o botão "Somar" é clicado (`(click)="somar()"`), o método `somar()` é executado.
5. O método `somar()` verifica se os valores dos inputs são válidos.
6. Se válidos, ele calcula a soma (usando `+` para garantir que sejam números) e armazena em `resultado`. Em seguida, define `mostrarResultado` como `true`.
7. Se inválidos, define `mostrarResultado` como `false`.
8. No HTML, o `@if (mostrarResultado)` verifica o valor dessa flag. Somente se for `true`, o parágrafo `<p>` é renderizado, exibindo o `resultado` calculado usando interpolação `{{ }}`.

## Exercício 4: Alterar Cor de Fundo

**Objetivo:** Criar um componente com três botões ("Vermelho", "Verde", "Azul") e uma `div`. Ao clicar em um dos botões, a cor de fundo da `div` deve ser alterada dinamicamente para a cor correspondente.

**Conceitos Praticados:**

- Declaração de uma propriedade (estado) na classe para armazenar a cor atual.
- Criação de um método que recebe um parâmetro (`novaCor`) para atualizar o estado da cor.
- Uso de **Event Binding** `(click)` para chamar o método, passando a cor desejada como argumento.
- Uso de **Property Binding** `[style.backgroundColor]` para vincular a propriedade da classe ao estilo CSS `background-color` da `div`.

- Necessidade de importar `CommonModule` para utilizar `[style.backgroundColor]`.

**Código:**

**exercicios-estados.component.ts**

```ts
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  corDeFundo: string = 'white';

  mudarCor(novaCor: string) {
    this.corDeFundo = novaCor;
  }
}
```

**exercicios-estados.component.html**

```html
<div>
  <button (click)="mudarCor('red')">Vermelho</button>
  <button (click)="mudarCor('green')">Verde</button>
  <button (click)="mudarCor('blue')">Azul</button>
</div>

<div [style.backgroundColor]="corDeFundo">
  A cor de fundo desta caixa vai mudar.
</div>
```

**Explicação do Funcionamento:**

1. A propriedade `corDeFundo` é inicializada com `'white'` na classe.
2. O método `mudarCor(novaCor)` atualiza o valor de `corDeFundo` com a `string` recebida.
3. No HTML, cada botão tem um `(click)` que chama `mudarCor()`, passando uma `string` de cor diferente ('red', 'green', 'blue').
4. A `div` utiliza o Property Binding `[style.backgroundColor]="corDeFundo"`. Isso significa que o estilo `background-color` da `div` será sempre igual ao valor atual da propriedade `corDeFundo`.
5. Quando um botão é clicado, `mudarCor()` atualiza `corDeFundo`, e o Angular automaticamente atualiza o estilo da `div` na tela, mudando sua cor de fundo.
6. O `CommonModule` precisa ser importado para que diretivas como `[style.backgroundColor]` funcionem corretamente em componentes standalone.

## Exercício 5: Contador de Caracteres

**Objetivo:** Criar um componente com uma área de texto (`<textarea>`) onde o usuário pode digitar. Abaixo da área de texto, exibir um contador que mostra o número de caracteres digitados em tempo real.

**Conceitos Praticados:**

- Declaração de uma propriedade (estado) na classe para armazenar o texto digitado (`texto`).
- Uso de **Two-Way Data Binding** `[(ngModel)]` para conectar a `<textarea>` à propriedade `texto`.
- Necessidade de importar o `FormsModule` para usar `[(ngModel)]`.
- Uso de **Interpolação** `{{ }}` para exibir dados dinâmicos no template.
- Acesso à propriedade `.length` de uma `string` para obter o número de caracteres.

**Código:**

**exercicios-estados.component.ts**

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

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

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  texto: string = '';
}
```

**exercicios-estados.component.html**

```
<textarea [(ngModel)]="texto" placeholder="Digite seu texto aqui..."></textarea>

<p>Caracteres digitados: {{ texto.length }}</p>
```

**Explicação do Funcionamento:**

1. A propriedade `texto` é inicializada como uma `string` vazia na classe.
2. O `FormsModule` é importado para permitir o uso de `[(ngModel)]`.
3. No HTML, a `<textarea>` usa `[(ngModel)]="texto"` para criar uma ligação bidirecional. Isso significa que:
    - Qualquer caractere digitado na `<textarea>` atualiza **imediatamente** o valor da propriedade `texto` na classe.
4. O parágrafo `<p>` usa a interpolação `{{ texto.length }}`. O Angular observa a propriedade `texto`.
5. Como `texto` é atualizada a cada caractere digitado (graças ao `[(ngModel)]`), o Angular recalcula `texto.length` e atualiza o valor exibido no parágrafo em tempo real, criando o efeito de contador instantâneo.

## Exercício 6: Gerador de Número Aleatório

**Objetivo:** Criar um componente que permita ao usuário definir um valor mínimo e um valor máximo através de inputs. Ao clicar em um botão, o componente deve gerar um número inteiro aleatório dentro desse intervalo (inclusive) e exibi-lo na tela. Implementar uma validação simples para garantir que o valor mínimo seja menor que o máximo.

**Conceitos Praticados:**

- Declaração de propriedades numéricas na classe (`minValor`, `maxValor`, `numeroGerado`).
- Declaração de uma propriedade booleana (`mostrarResultado`) para controle de exibição.
- Uso de **Two-Way Data Binding** `[(ngModel)]` com inputs do tipo `number`.
- Necessidade de importar `FormsModule`.
- Criação de um método (`gerarNumero`) que contém a lógica de validação e geração do número aleatório (`Math.random()`, `Math.floor()`).
- Uso de **Event Binding** `(click)` para acionar o método.
- Uso da estrutura `@if / @else` para exibir o resultado ou uma mensagem de erro/instrução.
- Uso de **Interpolação** `{{ }}` para exibir o número gerado.
- Necessidade de importar `CommonModule`.

**Código:**

**exercicios-estados.component.ts**

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './exercicios-estados.html',
```

```
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  minValor = 0;
  maxValor = 0;
  numeroGerado = 0;
  mostrarResultado = false;

  gerarNumero(){
    if (this.minValor < this.maxValor) {
      const min = this.minValor;
      const max = this.maxValor;

       this.numeroGerado = Math.floor(Math.random() * (max - min + 1)) +
min;
      this.mostrarResultado = true;
    } else {
      this.mostrarResultado = false;
    }
  }
}
```

exercicios-estados.component.html

```
<div>
  <label>Mínimo:</label>
  <input type="number" [(ngModel)]="minValor" />
</div>

<div>
  <label>Máximo:</label>
  <input type="number" [(ngModel)]="maxValor" />
</div>

<button (click)="gerarNumero()">Gerar Número</button>
```

```
@if (mostrarResultado) {
  <p>Número Gerado: {{ numeroGerado }}</p>
} @else {
    <p>Clique em gerar ou verifique se o valor mínimo é menor ao máximo.</p>
}
```

**Explicação do Funcionamento:**

1. As propriedades `minValor`, `maxValor` e `numeroGerado` são inicializadas com `0`. `mostrarResultado` começa como `false`.
2. Os `FormsModule` e `CommonModule` são importados.
3. No HTML, os inputs usam `[(ngModel)]` para ligar seus valores a `minValor` e `maxValor`.
4. Ao clicar no botão "Gerar Número", o método `gerarNumero()` é chamado.
5. O método verifica se `minValor` é estritamente menor que `maxValor`.
6. Se for, ele calcula um número inteiro aleatório usando `Math.random()` ajustado para o intervalo `[minValor, maxValor]` e armazena em `numeroGerado`. Define `mostrarResultado` como `true`.
7. Se `minValor` não for menor que `maxValor`, ele define `mostrarResultado` como `false`, escondendo qualquer resultado anterior.
8. O bloco `@if/@else` no HTML usa a flag `mostrarResultado` para decidir se exibe o parágrafo com o número gerado ou a mensagem de instrução/erro.

## Condicionais e Loops (`@if`, `@for`)

### Exercício 1: Lista Simples com @for

**Objetivo:** Criar um componente que tenha uma lista (array) de strings definida em sua classe TypeScript e use a diretiva `@for` no template HTML para renderizar cada item dessa lista dentro de uma `<ul>` (lista não ordenada).

**Conceitos Praticados:**

- Declaração de uma propriedade (estado) na classe do tipo array de strings (`string[]`).

- Uso da diretiva estrutural `@for` para iterar sobre os elementos de um array no template.
- Uso da cláusula `track` dentro do `@for` para otimização da renderização.
- Uso de **Interpolação** `{{ }}` para exibir cada item do array no template.
- Necessidade de importar o `CommonModule` para usar a diretiva `@for`.

**Código:**

**exercicios-estados.component.ts**

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  frutas: string[] = ['Maçã', 'Banana', 'Laranja', 'Uva', 'Morango'];
}
```

**exercicios-estados.component.html**

```
<ul>
  @for (fruta of frutas; track fruta) {
    <li>{{ fruta }}</li>
  }
</ul>
```

**Explicação do Funcionamento:**

1. Na classe TypeScript, a propriedade `frutas` é definida como um array contendo cinco strings.

2. O `CommonModule` é importado no componente para habilitar o uso da diretiva `@for`.

3. No HTML, a diretiva `@for (fruta of frutas; track fruta)` instrui o Angular a repetir o bloco de código seguinte (`<li>{{ fruta }}</li>`) para cada elemento do array `frutas`.
4. A cada repetição, a variável `fruta` recebe o valor do elemento atual do array (primeiro 'Maçã', depois 'Banana', e assim por diante).
5. A cláusula `track fruta` informa ao Angular como identificar unicamente cada item da lista (neste caso, usando o próprio valor da string), o que é essencial para otimizar atualizações futuras na lista.
6. Dentro do loop, a interpolação `{{ fruta }}` exibe o valor atual da variável `fruta` dentro de uma tag `<li>`.
7. O resultado final na tela é uma lista HTML (`<ul>`) com cinco itens (`<li>`), cada um contendo o nome de uma fruta.

## Exercício 2: Tabela Dinâmica com @for (Renderizando Objetos)

**Objetivo:** Criar um componente que tenha uma lista (array) de objetos definidos em sua classe TypeScript e use a diretiva `@for` no template HTML para renderizar as propriedades de cada objeto.

**Conceitos Praticados:**

- Definição de uma `interface` (`Produto`) para descrever a estrutura dos objetos (boa prática).
- Declaração de uma propriedade (estado) na classe do tipo array de objetos (`Produto[]`).
- Uso da diretiva estrutural `@for` para iterar sobre um array de objetos.
- Uso da cláusula `track` com uma **propriedade única** do objeto (como `produto.id`) dentro do `@for`, que é essencial ao iterar sobre objetos.
- Acesso às propriedades de cada objeto dentro do loop (ex: `produto.nome`, `produto.preco`).
- Uso de **Interpolação** `{{ }}` para exibir os valores das propriedades.
- Uso de **Pipes** (como o `number`) para formatar a exibição dos dados.
- Necessidade de importar o `CommonModule` para usar `@for` e pipes.

**Código:**

**exercicios-estados.component.ts**

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

interface Produto {
  id: number;
  nome: string;
  preco: number;
}

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  produtos: Produto[] = [
    { id: 1, nome: 'Laptop', preco: 3500.50 },
    { id: 2, nome: 'Teclado', preco: 150.75 },
    { id: 3, nome: 'Mouse', preco: 80.00 },
    { id: 4, nome: 'Monitor', preco: 1200.00 }
  ];
}
```

**exercicios-estados.component.html**

```
@for (produto of produtos; track produto.id) {
    <p>
            ID: {{ produto.id }} | Nome: {{ produto.nome }} | Preço: {{
produto.preco | number : '1.2-2' }}
    </p>
}
```

**Explicação do Funcionamento:**

1. A interface `Produto` define que cada item da nossa lista terá `id`, `nome` e `preco`.
2. Na classe, a propriedade `produtos` é um array que contém vários objetos seguindo essa estrutura.
3. O `CommonModule` é importado para permitir o uso de `@for` e do pipe `number`.
4. No HTML, o `@for (produto of produtos; track produto.id)` itera sobre o array `produtos`.
5. **`track produto.id`**: Ao iterar sobre objetos, é crucial usar `track` com uma propriedade que seja **única** para cada objeto (como o `id`). Isso permite ao Angular otimizar a atualização da lista de forma eficiente quando os dados mudam. Usar `track produto` (o objeto inteiro) pode ser menos performático.
6. Dentro do loop, a variável `produto` representa o objeto atual. Acessamos suas propriedades usando a notação de ponto (ex: `produto.nome`, `produto.preco`).
7. A interpolação `{{ }}` exibe os valores dessas propriedades.
8. **`produto.preco | number : '1.2-2'`**: Aqui usamos um **Pipe** (`|`). O pipe `number` formata o valor numérico `produto.preco`. O parâmetro `'1.2-2'` especifica o formato: no mínimo `1` dígito inteiro, e exatamente `2` dígitos decimais (ideal para exibir valores monetários).
9. O resultado na tela é uma sequência de parágrafos, cada um mostrando os detalhes de um produto.

## Exercício 3: Exibir/Ocultar com @if

**Objetivo:** Criar um componente com um botão e um elemento de texto (um parágrafo `<p>`). Ao clicar no botão, o texto deve alternar entre visível e oculto. O texto do próprio botão também deve mudar para indicar a próxima ação ("Ocultar Texto" ou "Mostrar Texto").

**Conceitos Praticados:**

- Declaração de uma propriedade booleana (`visibilidadeDoTexto`) na classe para controlar o estado de visibilidade.
- Criação de um método para inverter o valor dessa propriedade booleana (`inverterValrDeVisibilidade`).
- Uso de **Event Binding** `(click)` para chamar o método a partir do botão.
- Uso da diretiva estrutural `@if` para renderizar condicionalmente o parágrafo `<p>`.

- Uso da estrutura `@if / @else` para renderizar condicionalmente um dos dois botões (um com o texto "Ocultar", outro com "Mostrar").
- Necessidade de importar o `CommonModule` para usar `@if` e `@else`.

**Código:**

**exercicios-estados.component.ts**

```ts
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  visibilidadeDoTexto: boolean = true;

  inverterValrDeVisibilidade() {
    this.visibilidadeDoTexto = !this.visibilidadeDoTexto;
  }
}
```

**exercicios-estados.component.html**

```html
@if (visibilidadeDoTexto) {
  <button (click)="inverterValrDeVisibilidade()">Ocultar Texto</button>
} @else {
  <button (click)="inverterValrDeVisibilidade()">Mostrar Texto</button>
}

@if (visibilidadeDoTexto) {
  <p>Este texto pode ser escondido!</p>
}
```

**Explicação do Funcionamento:**

1. A propriedade `visibilidadeDoTexto` começa como `true`.

2. O `CommonModule` é importado para permitir o uso de `@if` e `@else`.
3. No HTML, o primeiro bloco `@if/@else` verifica o valor de `visibilidadeDoTexto`.
4. Como começa `true`, o primeiro botão (`<button>Ocultar Texto</button>`) é renderizado. Este botão, ao ser clicado (`(click)`), chama o método `inverterValrDeVisibilidade()`.
5. O segundo bloco `@if` também verifica `visibilidadeDoTexto`. Como é `true`, o parágrafo `<p>Este texto pode ser escondido!</p>` é renderizado.
6. **Quando o botão "Ocultar Texto" é clicado:**
    - O método `inverterValrDeVisibilidade()` é chamado.
    - `visibilidadeDoTexto` se torna `false`.
    - O Angular detecta a mudança e reavalia os blocos `@if`.
    - O primeiro bloco `@if/@else` agora renderiza o segundo botão (`<button>Mostrar Texto</button>`).
    - O segundo bloco `@if` avalia `visibilidadeDoTexto` como `false`, e o parágrafo `<p>` **não** é renderizado (desaparece da tela).
7. **Quando o botão "Mostrar Texto" é clicado:**
    - O processo se inverte: `visibilidadeDoTexto` volta a ser `true`, o botão "Ocultar Texto" reaparece e o parágrafo `<p>` é renderizado novamente.

## Exercício 4: Mostrar/Ocultar Detalhes

**Objetivo:** Criar um componente que exibe uma lista de itens. Cada item tem um nome (sempre visível) e uma descrição (inicialmente oculta). Um botão ao lado de cada nome permite ao usuário mostrar ou ocultar a descrição daquele item específico.

**Conceitos Praticados:**

- Definição de uma `interface` (`ItemComDetalhes`) para incluir uma propriedade booleana que controla a visibilidade (`detalhesVisiveis`).

- Declaração de um array de objetos (`items: ItemComDetalhes[]`) na classe.
- Uso da diretiva estrutural `@for` para iterar sobre o array de objetos.

- Uso de `track` com a propriedade `id` do objeto.
- Criação de um método (`inverterEstadoDeDetalhes`) que recebe o objeto `item` como parâmetro e inverte o valor da sua propriedade booleana `detalhesVisiveis`.
- Uso de **Event Binding** `(click)` para chamar o método, passando o objeto `item` atual do loop.
- Uso da diretiva estrutural `@if` para renderizar condicionalmente a descrição (`<p>`).
- Uso da estrutura `@if / @else` para renderizar condicionalmente o botão correto ("Mostrar Detalhes" ou "Ocultar Detalhes").
- Uso de **Interpolação** `{{ }}` para exibir o nome e a descrição.
- Necessidade de importar `CommonModule`.

**Código:**

**exercicios-estados.component.ts**

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

interface ItemComDetalhes {
  id: number;
  nome: string;
  descricao: string;
  detalhesVisiveis: boolean;
}

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  items: ItemComDetalhes[] = [
```

```
    { id: 1, nome: 'Item 1', descricao: 'Esta é a descrição detalhada do Item 1.', detalhesVisiveis: false },
    { id: 2, nome: 'Item 2', descricao: 'Esta é a descrição detalhada do Item 2.', detalhesVisiveis: false },
    { id: 3, nome: 'Item 3', descricao: 'Esta é a descrição detalhada do Item 3.', detalhesVisiveis: false }
  ];

  inverterEstadoDeDetalhes(item: ItemComDetalhes) {
    item.detalhesVisiveis = !item.detalhesVisiveis;
  }
}
```

**exercicios-estados.component.html**

```
<div>
  @for (item of items; track item.id) {
    <div>
      <span>{{ item.nome }}</span>

      @if (item.detalhesVisiveis) {
                <button  (click)="inverterEstadoDeDetalhes(item)">Ocultar Detalhes</button>
      } @else {
                <button  (click)="inverterEstadoDeDetalhes(item)">Mostrar Detalhes</button>
      }

      @if (item.detalhesVisiveis) {
        <p>{{ item.descricao }}</p>
      }
    </div>
    <hr> }
</div>
```

**Explicação do Funcionamento:**

1. A interface `ItemComDetalhes` agora inclui o campo `detalhesVisiveis`.

2. O array `items` na classe é inicializado com cada item tendo `detalhesVisiveis: false`.
3. O método `inverterEstadoDeDetalhes(item)` recebe o objeto `item` clicado e simplesmente inverte o valor da sua propriedade `detalhesVisiveis`.
4. No HTML, o `@for` percorre a lista.
5. O nome do item é sempre exibido.
6. O primeiro bloco `@if/@else` verifica a propriedade `item.detalhesVisiveis` do item *atual* do loop.
7. Se for `true`, o botão "Ocultar Detalhes" é renderizado. Ao ser clicado, ele chama `inverterEstadoDeDetalhes(item)`, passando o item atual, o que tornará `detalhesVisiveis false`.
8. Se for `false`, o botão "Mostrar Detalhes" é renderizado. Ao ser clicado, ele chama `inverterEstadoDeDetalhes(item)`, o que tornará `detalhesVisiveis true`.
9. O segundo bloco `@if` também verifica `item.detalhesVisiveis`.
10. Se for `true`, o parágrafo `<p>` com a `item.descricao` é renderizado. Se for `false`, ele não é renderizado (fica oculto).
11. O Angular detecta as mudanças na propriedade `detalhesVisiveis` de cada item e atualiza a interface (trocando o botão e mostrando/ocultando o parágrafo) automaticamente.

## Exercício 5: Desafio FizzBuzz

**Objetivo:** Criar um componente que permita ao usuário definir um limite numérico. Ao clicar em um botão, o componente deve gerar uma lista de números até esse limite e, para cada número, exibir:

- "FizzBuzz" se o número for múltiplo de 3 e 5.
- "Fizz" se o número for múltiplo de 3.
- "Buzz" se o número for múltiplo de 5.
- O próprio número caso contrário.

**Conceitos Praticados:**

- Declaração de propriedades (estados) na classe (`numeros`, `limiteInput`).
- Uso de **Two-Way Data Binding** `[(ngModel)]` para obter o limite do usuário.

- Necessidade de importar `FormsModule`.
- Criação de um método (`gerarLista`) para popular um array com números até o limite definido.
- Criação de um método (`getFizzBuzz`) que contém a lógica condicional do FizzBuzz (uso do operador módulo `%`).
- Uso da diretiva estrutural `@for` para iterar sobre o array de números.
- Chamada de um método da classe (`getFizzBuzz`) dentro da interpolação `{{ }}` no template.
- Uso do bloco `@empty` para feedback ao usuário.
- Necessidade de importar `CommonModule`.

**Código:**

**exercicios-estados.component.ts**

```ts
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  numeros: number[] = [];
  limiteInput = 0;

  gerarLista() {
    this.numeros = [];
    if (this.limiteInput !== null && this.limiteInput > 0) {
      for (let i = 1; i <= this.limiteInput; i++) {
        this.numeros.push(i);
```

```
      }
    }
  }

  getFizzBuzz(numero: number) {
    const isFizz = numero % 3 === 0;
    const isBuzz = numero % 5 === 0;

    if (isFizz && isBuzz) {
      return 'FizzBuzz';
    } else if (isFizz) {
      return 'Fizz';
    } else if (isBuzz) {
      return 'Buzz';
    } else {
      return numero.toString();
    }
  }
}
```

exercicios-estados.component.html

```
<label for="limite">Digite o limite:</label>
<input id="limite" type="number" [(ngModel)]="limiteInput" />

<button (click)="gerarLista()">Gerar FizzBuzz</button>

<div>
  <h3>Resultado FizzBuzz:</h3>
  @for (num of numeros; track num) {
    <p>{{ getFizzBuzz(num) }}</p>
}
@empty {
    <p>Nenhum número gerado ainda ou limite inválido.</p>
  }
</div>
```

**Explicação do Funcionamento:**

1. O componente inicializa com um array `numeros` vazio e `limiteInput` como `null`.
2. O usuário digita um número no `<input>`, que é armazenado em `limiteInput` via `[(ngModel)]`.
3. Ao clicar no botão "Gerar FizzBuzz", o método `gerarLista()` é chamado.
4. `gerarLista()` limpa o array `numeros` e, se `limiteInput` for válido, preenche `numeros` com a sequência de 1 até o limite.
5. O `@for` no HTML detecta a atualização do array `numeros`.
6. Para cada `num` no array, a expressão `{{ getFizzBuzz(num) }}` é avaliada.
7. A função `getFizzBuzz()` recebe o número, aplica a lógica FizzBuzz usando o operador módulo (`%`), e retorna a string apropriada ("Fizz", "Buzz", "FizzBuzz" ou o número).
8. O resultado retornado por `getFizzBuzz()` é exibido no parágrafo `<p>`.
9. Se `numeros` estiver vazio (porque `limiteInput` é inválido ou o botão ainda não foi clicado), o bloco `@empty` é exibido.

## Exercício 6: Mensagem de Status com @if/@else

**Objetivo:** Criar um componente que simula um status de login. Ele deve exibir uma mensagem de boas-vindas e um botão "Logout" se o usuário estiver "logado" (representado por uma variável booleana), ou uma mensagem pedindo login e um botão "Login" caso contrário. O botão deve alternar o estado de login.

**Conceitos Praticados:**

- Declaração de uma propriedade booleana (`estaLogado`) na classe para controlar o estado.
- Criação de um método para inverter o valor dessa propriedade booleana (`inverterCasoDeLogado`).
- Uso da diretiva estrutural `@if` juntamente com `@else` para renderizar blocos de HTML completamente diferentes com base na condição.
- Uso de **Event Binding** `(click)` para chamar o método de alternância a partir dos botões.
- Necessidade de importar `CommonModule` para usar `@if` e `@else`.

**Código:**

**exercicios-estados.component.ts**

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
  estaLogado = false;

  inverterCasoDeLogado() {
    this.estaLogado = !this.estaLogado;
  }
}
```

**exercicios-estados.component.html**

```
<div>
  @if (estaLogado) {
    <p>Bem-vindo, usuário!</p>
    <button (click)="inverterCasoDeLogado()">Logout</button>
  } @else {
    <p>Por favor, faça o login.</p>
    <button (click)="inverterCasoDeLogado()">Login</button>
  }
</div>
```

**Explicação do Funcionamento:**

1. A propriedade `estaLogado` é inicializada como `false`.

2. O `CommonModule` é importado para permitir o uso de `@if` e `@else`.
3. No HTML, o bloco `@if (estaLogado)` verifica o valor da propriedade.
4. Como `estaLogado` começa como `false`, o bloco `@else` é executado, renderizando o parágrafo "Por favor, faça o login." e o botão "Login".
5. Quando o botão "Login" é clicado:
    - O método `inverterCasoDeLogado()` é chamado.
    - `estaLogado` se torna `true`.
    - O Angular reavalia o bloco `@if/@else`.
    - Agora, o bloco `@if` é executado, renderizando o parágrafo "Bem-vindo, usuário!" e o botão "Logout".
6. Quando o botão "Logout" é clicado:
    - O método `inverterCasoDeLogado()` é chamado novamente.
    - `estaLogado` volta a ser `false`.
    - O Angular reavalia e o bloco `@else` é renderizado novamente.

## Exercício 7: Aplicar Classe Condicionalmente

**Objetivo:** Criar um componente que exibe uma lista de tarefas. Quando uma tarefa estiver marcada como "concluída" (baseado em uma propriedade booleana no objeto da tarefa), uma classe CSS específica deve ser aplicada ao item da lista para dar um feedback visual (por exemplo, riscar o texto). Permitir que o usuário clique em uma tarefa para alternar seu estado de conclusão.

**Conceitos Praticados:**

- Definição de uma `interface` (`Tarefa`) para a estrutura dos dados.
- Declaração de um array de objetos (`tarefas: Tarefa[]`) na classe.
- Uso da diretiva estrutural `@for` para iterar sobre o array de objetos.
- Uso de `track` com a propriedade `id` do objeto.
- Criação de um método (`alterarEstadoDaTarefa`) para modificar uma propriedade booleana do objeto.
- Uso de **Event Binding** `(click)` no item da lista para chamar o método.
- Uso de **Property Binding** na sintaxe `[class.nome-da-classe]="condicao"` para adicionar ou remover uma classe CSS dinamicamente.
- Uso de **Interpolação** `{{ }}` para exibir a descrição da tarefa.

- Necessidade de importar `CommonModule` para usar `@for` e `[class. ...]`.
- Definição de estilos CSS para a classe condicional.

**Código:**

**exercicios-estados.component.ts**

```ts
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

interface Tarefa {
  id: number;
  descricao: string;
  concluida: boolean;
}

@Component({
  selector: 'app-exercicios-estados',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './exercicios-estados.html',
  styleUrl: './exercicios-estados.css'
})
export class ExerciciosEstados {
tarefas: Tarefa[] = [
    { id: 1, descricao: 'Estudar Angular', concluida: false },
    { id: 2, descricao: 'Fazer compras', concluida: true },
    { id: 3, descricao: 'Lavar a louça', concluida: false },
    { id: 4, descricao: 'Ler um livro', concluida: true }
  ];

  alterarEstadoDaTarefa(tarefa: Tarefa){
    tarefa.concluida = !tarefa.concluida;
  }
}
```

**exercicios-estados.component.html**

```
<ul>
  @for (tarefa of tarefas; track tarefa.id) {
                      <li     [class.tarefa-concluida]="tarefa.concluida"
(click)="alterarEstadoDaTarefa(tarefa)">
      {{ tarefa.descricao }}
    </li>
  }
</ul>

<p style="color: gray;">Clique em uma tarefa para marcar/desmarcar como
concluída</p>
```

**exercicios-estados.component.css**

```
li {
  cursor: pointer;
  padding: 5px;
  margin-bottom: 5px;
  border: 1px solid #ccc;
}

li.tarefa-concluida {
  text-decoration: line-through;
  color: gray;
  background-color: #f0f0f0;
}
```

**Explicação do Funcionamento:**

1. A interface `Tarefa` e o array `tarefas` são definidos na classe.
2. O método `alterarEstadoDaTarefa` inverte o valor booleano da propriedade `concluida` do objeto `tarefa` que ele recebe.
3. O `CommonModule` é importado.
4. No HTML, o `@for` percorre o array `tarefas`.
5. A linha chave é `<li [class.tarefa-concluida]="tarefa.concluida" ...>`. O `[class.tarefa-concluida]` diz ao Angular: "Gerencie a classe CSS

`tarefa-concluida` para este elemento `<li>`". A parte `="tarefa.concluida"` diz: "Adicione a classe se `tarefa.concluida` for `true`, remova-a se for `false`".

6. O `(click)="alterarEstadoDaTarefa(tarefa)"` no `<li>` faz com que, ao clicar no item, o método seja chamado, passando a tarefa específica daquela linha.
7. A interpolação `{{ tarefa.descricao }}` exibe o texto da tarefa.
8. No CSS, a regra `li.tarefa-concluida` define como o item deve parecer *quando* a classe `tarefa-concluida` está aplicada a ele.
9. Quando o usuário clica em um `<li>`, o método `alterarEstadoDaTarefa` muda o valor de `tarefa.concluida`. O Angular detecta essa mudança, reavalia o `[class.tarefa-concluida]` e adiciona ou remove a classe CSS do `<li>`, fazendo com que o estilo definido no CSS seja aplicado ou removido instantaneamente.

# Tutorial Prático: Construindo uma To-Do List com Angular

Este guia irá detalhar, passo a passo, a construção de uma aplicação de lista de tarefas (To-do List) funcional, aplicando os principais conceitos do Angular.

## Seção 1: Preparação do Ambiente (O Alicerce)

Nesta primeira etapa, vamos criar a estrutura básica do nosso projeto, gerar nosso componente principal e configurá-lo para ser a página inicial da nossa aplicação.

### 1.1. Criando o Projeto Angular

Vamos começar utilizando a Angular CLI para criar um novo projeto. Abra seu terminal na pasta onde deseja salvar a aplicação e execute o comando:

```
Bash
ng new ToDoListAngular
```

Durante a configuração, a CLI fará duas perguntas:

- **Qual formato de folha de estilo você gostaria de usar?**
  - **Resposta:** Selecione **CSS**.
- **Deseja habilitar o Renderização do Lado do Servidor (SSR)?**
  - **Resposta:** Selecione **Não**. Isso configurará nosso projeto como uma **SPA (Single Page Application)**, que é o nosso foco.

### 1.2. Criando o Componente Principal (`Home`)

Com o projeto criado, o próximo passo é gerar o componente que conterá a lógica e a visualização da nossa lista de tarefas.

1. No terminal, certifique-se de estar dentro da pasta do projeto (`cd ToDoListAngular`).
2. Execute o seguinte comando para gerar o componente `home`:

```
Bash
ng g c components/home
```

- **Boa Prática:** Ao usar `components/home`, estamos instruindo a CLI a criar uma pasta `components` dentro de `src/app/` para organizar nossos componentes, e

dentro dela, a pasta `home` com os arquivos do nosso novo componente. Esta é uma convenção que mantém o projeto limpo e escalável.

### 1.3. Configurando a Rota Principal

Um componente recém-criado é como uma peça de LEGO fora da caixa; ele existe, mas ainda não faz parte da estrutura principal. Precisamos dizer ao Angular para exibir nosso `HomeComponent` como a página inicial da aplicação.

1. Abra o arquivo `src/app/app.routes.ts`. Este é o "mapa" da nossa aplicação.
2. Modifique o array `routes` para que a rota raiz (`path: ""`) aponte para o `HomeComponent`.

```
TypeScript
// No arquivo app.routes.ts

// 1. Importe o HomeComponent que acabamos de criar
import { HomeComponent } from './components/home/home.component';
import { Routes } from '@angular/router';

export const routes: Routes = [
  // 2. Configure a rota raiz para carregar o HomeComponent
  {
    path: "",
    component: HomeComponent
  }
];
```

Com estes passos, finalizamos a preparação. Nosso projeto agora está configurado para, ao ser iniciado, carregar o `HomeComponent` como a página principal. A partir daqui, podemos começar a construir a funcionalidade da nossa To-do List.

## Seção 2: Modelando Nossos Dados (A "Planta" da Tarefa)

Com a estrutura do projeto pronta, antes de construir a parte visual, uma boa prática é definir como serão os nossos dados. O que define uma "tarefa" na nossa aplicação? Para isso, usaremos um recurso do TypeScript chamado `interface`.

### 2.1. Uma Nota Importante Sobre Boas Práticas: Nomenclatura em Inglês

Antes de começarmos a codificar, é crucial alinhar uma convenção profissional. Neste tutorial, para fins didáticos e para facilitar a compreensão, usaremos nomes de variáveis e funções em português (como `listaDeTarefas`, `adicionarTarefa`, etc.).

No entanto, no mercado de trabalho e na comunidade de desenvolvimento de software, a convenção universal é escrever todo o código em **inglês**.

- **Por quê?**
    - **Colaboração Global:** Facilita o trabalho em equipes com pessoas de diferentes nacionalidades.
    - **Consistência:** O próprio Angular, suas bibliotecas e toda a documentação estão em inglês. Manter seu código no mesmo idioma cria consistência.
    - **Facilidade de Pesquisa:** É muito mais fácil encontrar soluções para problemas em fóruns como o Stack Overflow ao pesquisar por termos em inglês.

### 2.2. Definindo a Estrutura de uma Tarefa (A Interface `TodoItem`)

Uma `interface` é como um "contrato" ou uma "planta baixa" que define a forma de um objeto. Ela nos ajuda a garantir que todos os objetos de "tarefa" em nossa aplicação tenham sempre as mesmas propriedades, evitando erros.

1. Abra o arquivo `src/app/components/home/home.component.ts`.
2. **Acima** da linha do `@Component`, adicione o seguinte código para definir a nossa interface:

```
TypeScript
export interface TodoItem {
  id: number;
  task: string;
  finalizada: boolean;
}
```

- **O que este código faz?**
    - `export interface TodoItem`: Cria um novo "tipo" chamado `TodoItem`.
    - `id: number;`: Define que cada tarefa terá um identificador único, que será um número.
    - `task: string;`: Define que cada tarefa terá uma descrição, que será um texto.

- `finalizada: boolean;`: Define que cada tarefa terá um estado de conclusão, que será `true` ou `false`.

### 2.3. Criando os Estados no Componente

Agora que temos a "planta" da nossa tarefa, vamos criar as propriedades (estados) dentro da classe `HomeComponent` que irão armazenar os dados da nossa aplicação.

1. Ainda no arquivo `home.component.ts`, dentro da classe `HomeComponent`, adicione as seguintes propriedades:

```
TypeScript
export class HomeComponent {
  // 1. Uma lista para armazenar todas as nossas tarefas.
  //    Será um array de objetos, e cada objeto deve seguir o "contrato"
da interface TodoItem.
  listaDeTarefas: TodoItem[] = [];

  // 2. Uma variável para guardar o texto da nova tarefa que o usuário
está digitando.
  novaTarefa: string = '';

  // ... resto do código do componente ...
}
```

Com esta seção concluída, nosso componente agora tem uma estrutura de dados clara e as "gavetas" (variáveis) prontas para armazenar as tarefas. O próximo passo será construir a interface HTML para que o usuário possa adicionar novas tarefas.

## Seção 3: Construindo a Interface de Adição

Nesta seção, vamos criar o layout HTML básico para o título, o campo de texto (input) e o botão de adicionar. Em seguida, conectaremos o campo de texto ao estado do nosso componente usando `[(ngModel)]`.

### 3.1. O Layout HTML Básico

Primeiro, vamos adicionar a estrutura visual inicial ao nosso componente.

1. Abra o arquivo `src/app/components/home/home.component.html`.

2. Apague o conteúdo padrão (`<p>home works!</p>`) e substitua pelo seguinte código:

```
HTML
<div class="titulo">
  <h1>Lista de tarefas</h1>
</div>

<div class="adicionar_task">
  <input placeholder="Digite uma nova tarefa..." />
  <button>Adicionar Tarefa</button>
</div>
```

### 3.2. Ligação Bidirecional com `[(ngModel)]`

Agora, precisamos criar uma conexão "viva" entre o campo de input e a nossa propriedade `novaTarefa` no arquivo `.ts`. Para isso, usamos a diretiva `[(ngModel)]`, que implementa o **Two-Way Data Binding** (Ligação de Dados Bidirecional).

- **O que isso significa?** Significa que a comunicação acontece em dois sentidos, em tempo real:
    1. **Da classe para o template:** Se o valor de `novaTarefa` mudar na lógica, o texto no input será atualizado.
    2. **Do template para a classe:** Tudo que o usuário digitar no input será **automaticamente** salvo na propriedade `novaTarefa`.

1. Modifique a tag `<input>` no seu arquivo `home.component.html`:

```
HTML
<input [(ngModel)]="novaTarefa" placeholder="Digite uma nova tarefa..."
/>
```

Ao adicionar `[(ngModel)]="novaTarefa"`, você notará que seu editor de código provavelmente apontará um erro. Isso acontece porque, para usar essa diretiva, precisamos de um módulo específico.

### 3.3. Habilitando o `FormsModule`

A diretiva `[(ngModel)]` faz parte do `FormsModule` do Angular. Para que ela funcione, precisamos importar este módulo no nosso `HomeComponent`.

1. Abra o arquivo `src/app/components/home/home.component.ts`.
2. Adicione a importação do `FormsModule` no topo do arquivo.
3. Adicione o `FormsModule` ao array `imports` do decorador `@Component`.

```
TypeScript
// 1. Adicione a importação no topo
import { FormsModule } from '@angular/forms';
import { Component } from '@angular/core';

// (Aqui fica a sua interface TodoItem)

@Component({
  selector: 'app-home',
  standalone: true,
  // 2. Adicione o FormsModule ao array de imports
  imports: [FormsModule],
  templateUrl: './home.component.html',
  styleUrl: './home.component.css'
})
export class HomeComponent {
  listaDeTarefas: TodoItem[] = [];
  novaTarefa: string = '';
}
```

Após salvar o arquivo, o erro no `[(ngModel)]` deverá desaparecer. Agora, a interface de adição está pronta e conectada à lógica do nosso componente. O próximo passo será implementar a função para realmente adicionar a tarefa à lista.

### 3.4. Nota para o Futuro: Formulários Reativos (Reactive Forms)

É muito importante que você saiba que a abordagem que usamos com `[(ngModel)]` faz parte de uma estratégia do Angular chamada **Template-Driven Forms** (Formulários Guiados pelo Template). O nome vem do fato de que a maior parte da lógica é declarada diretamente no arquivo HTML. Essa abordagem é excelente para formulários simples e rápidos, como o nosso.

No entanto, para cenários mais complexos, como formulários com muitas validações dinâmicas ou lógicas de negócio sofisticadas, o Angular oferece uma abordagem mais poderosa e escalável chamada **Reactive Forms** (Formulários Reativos).

- **O que são?** Nos Formulários Reativos, a lógica e o estado do formulário são gerenciados de forma explícita no arquivo TypeScript (`.ts`), o que te dá um controle muito mais granular e facilita os testes.

**Conclusão:** O que você aprendeu com `[(ngModel)]` é fundamental e muito utilizado. Apenas saiba que, à medida que seus projetos crescerem, estudar os **Formulários Reativos** será o próximo passo natural para se tornar um desenvolvedor Angular ainda mais completo.

## Seção 4: Implementando a Lógica (Adicionar e Exibir Tarefas)

Nesta seção, vamos criar o método `adicionarTarefa()` em nosso arquivo TypeScript, conectá-lo ao botão "Adicionar Tarefa" e, finalmente, usar um loop `@for` para renderizar a lista de tarefas na interface.

### 4.1. A Lógica para Adicionar a Tarefa (`adicionarTarefa()`)

Primeiro, vamos criar a função que será responsável por adicionar a nova tarefa à nossa lista.

1. Abra o arquivo `src/app/components/home/home.component.ts`.
2. Dentro da classe `HomeComponent`, adicione o seguinte método:

```
TypeScript
adicionarTarefa(){
  // 1. Verifica se o input não está vazio (ignorando espaços em branco)
  if (this.novaTarefa !== '') {
    // 2. Cria um novo objeto de tarefa
    const itemTarefa: TodoItem = {
      id: Date.now(), // Usa o timestamp como um ID único
      task: this.novaTarefa,
      finalizada: false
    };

    // 3. Adiciona a nova tarefa à lista
    this.listaDeTarefas.push(itemTarefa);

    // 4. Limpa o campo de input para a próxima tarefa
    this.novaTarefa = '';
  }
}
```

### 4.2. Conectando o Botão ao Método

Agora que a lógica está pronta, precisamos "ligar" o nosso botão a este método. Para isso, usaremos o **Event Binding `(click)`**.

1. Abra o arquivo `src/app/components/home/home.component.html`.
2. Modifique a tag `<button>` para que ela chame o método `adicionarTarefa()` quando for clicada:

```
HTML
<button (click)="adicionarTarefa()">Adicionar Tarefa</button>
```

### 4.3. Exibindo a Lista de Tarefas com `@for`

Neste ponto, as tarefas já estão sendo adicionadas à lista `listaDeTarefas`, mas ainda não conseguimos vê-las. Vamos usar a diretiva `@for` para renderizar cada tarefa na tela.

1. Ainda no arquivo `home.component.html`, abaixo da `div class="adicionar_task"`, adicione o seguinte bloco de código:

```
HTML
<ul class="listaDeTarefasParaOUsuarioVisualizar">
  @for (tarefa of listaDeTarefas; track tarefa.id) {
    <li>
      <span>{{ tarefa.task }}</span>
    </li>
  }
</ul>
```

- **O que este código faz?**
    - `@for (tarefa of listaDeTarefas; track tarefa.id)`: Itera sobre cada objeto `tarefa` dentro do array `listaDeTarefas`. O `track tarefa.id` é essencial para que o Angular otimize a renderização da lista.
    - `<span>{{ tarefa.task }}</span>`: Para cada tarefa, exibe o valor da sua propriedade `task` na tela.

Neste momento, sua aplicação está totalmente funcional para adicionar e exibir tarefas. O próximo passo será adicionar a interatividade para concluir e deletar as tarefas existentes.

## Seção 5: Adicionando Interatividade (Concluir e Deletar)

Nesta seção, vamos adicionar um checkbox e um botão de apagar a cada item da lista. Em seguida, criaremos os métodos correspondentes na nossa classe e os conectaremos à interface para dar ao usuário controle total sobre suas tarefas.

### 5.1. Atualizando o Layout HTML

Primeiro, vamos adicionar os novos elementos (o checkbox e o botão "Apagar") ao nosso loop `@for` no arquivo de template.

1. Abra o arquivo `src/app/components/home/home.component.html`.
2. Modifique o bloco `@for` para incluir o `<input type="checkbox">` e o `<button>`:

```
HTML
<ul class="listaDeTarefasParaOUsuarioVisualizar">
  @for (tarefa of listaDeTarefas; track tarefa.id) {
    <li>
      <input type="checkbox" />
      <span>{{ tarefa.task }}</span>
      <button>Apagar</button>
    </li>
  }
</ul>
```

### 5.2. A Lógica para Concluir e Deletar Tarefas

Agora, vamos criar os métodos no nosso arquivo `.ts` que irão manipular o estado de cada tarefa.

1. Abra o arquivo `src/app/components/home/home.component.ts`.
2. Dentro da classe `HomeComponent`, adicione os dois novos métodos:

```
TypeScript
// Método para alternar o estado 'finalizada' de uma tarefa
tarefaCompletada(tarefa: TodoItem) {
   tarefa.finalizada = !tarefa.finalizada; // Inverte o valor booleano
atual
}

// Método para deletar uma tarefa com base no seu id
deletarTarefa(id: number){
```

```
  // Cria uma nova lista contendo apenas os itens cujo id é diferente do id passado
   this.listaDeTarefas = this.listaDeTarefas.filter(item => item.id !== id);
}
```

### 5.3. Conectando a Interface aos Métodos

Com a lógica pronta, o último passo é conectar nossos novos botões e checkboxes a esses métodos usando **Event Binding**.

1. Volte ao arquivo `src/app/components/home/home.component.html`.
2. Modifique o loop `@for` para adicionar os bindings `(change)` e `(click)`:

```
HTML
<ul class="listaDeTarefasParaOUsuarioVisualizar">
  @for (tarefa of listaDeTarefas; track tarefa.id) {
    <li>
      <input
        type="checkbox"
        [checked]="tarefa.finalizada"
        (change)="tarefaCompletada(tarefa)"
      />
      <span>{{ tarefa.task }}</span>
      <button (click)="deletarTarefa(tarefa.id)">Apagar</button>
    </li>
  }
</ul>
```

- **`[checked]="tarefa.finalizada"`**: Usa *Property Binding* para que o checkbox já venha marcado se a tarefa estiver concluída.
- **`(change)="tarefaCompletada(tarefa)"`**: Quando o estado do checkbox muda, chama o método `tarefaCompletada` e passa o objeto da tarefa atual.
- **`(click)="deletarTarefa(tarefa.id)"`**: Quando o botão é clicado, chama o método `deletarTarefa` e passa o `id` da tarefa atual.

### 5.4. Aplicando Estilos Condicionais

Para dar um feedback visual ao usuário, vamos fazer com que as tarefas concluídas apareçam riscadas. Faremos isso aplicando uma classe CSS dinamicamente.

1. Ainda no arquivo `home.component.html`, adicione o **Property Binding** `[class.completada]` à tag `<li>`:

```
HTML
<li [class.completada]="tarefa.finalizada">
  </li>
```

    - **O que isso faz?** Adiciona a classe CSS `completada` à tag `<li>` somente **SE** a propriedade `tarefa.finalizada` for `true`.

2. Abra o arquivo `src/app/components/home/home.component.css` e adicione o seguinte estilo:

```
CSS
.listaDeTarefasParaOUsuarioVisualizar li.completada span {
  text-decoration: line-through;
  color: #888;
}
```

Agora, sua aplicação possui as funcionalidades de CRUD (Create, Read, Update, Delete) completas. O próximo e último passo será tornar os dados persistentes.

## Seção 6: Tornando a Aplicação Persistente (Salvando Dados)

Nossa aplicação funciona bem, mas os dados vivem apenas na memória. Para criar uma experiência de usuário real, precisamos salvar as tarefas de forma que elas persistam mesmo que a página seja fechada ou atualizada. A forma mais simples de fazer isso, sem a necessidade de um banco de dados, é usando o `localStorage` do navegador.

### 6.1. O que é o `localStorage`?

Pense no `localStorage` como um pequeno "bloco de notas" ou "depósito" que cada site tem dentro do seu navegador. Ele nos permite salvar dados em formato de texto (`string`) que ficam armazenados no computador do usuário e não são apagados ao recarregar a página.

### 6.2. A Lógica para Salvar as Tarefas

A nossa estratégia será salvar a lista de tarefas inteira no `localStorage` sempre que ela for modificada (ao adicionar, concluir ou deletar uma tarefa).

1. Abra o arquivo `src/app/components/home/home.component.ts`.
2. Para evitar repetir código, vamos criar um **método auxiliar** chamado `salvarTarefas()`. Adicione este método dentro da classe `HomeComponent`:

```
TypeScript
salvarTarefas(){
   // localStorage só armazena texto, então convertemos nosso array de objetos para uma string no formato JSON.
                                    localStorage.setItem('listaDeTarefas', JSON.stringify(this.listaDeTarefas));
}
```

3. Agora, vamos chamar este método no final de cada função que modifica a lista: `adicionarTarefa()`, `tarefaCompletada()` e `deletarTarefa()`.

```
TypeScript
adicionarTarefa(){
  // ... (lógica para adicionar a tarefa) ...
  this.salvarTarefas(); // Salva a lista atualizada
}

tarefaCompletada(tarefa: TodoItem){
  tarefa.finalizada = !tarefa.finalizada;
  this.salvarTarefas(); // Salva a lista atualizada
}

deletarTarefa(id: number){
   this.listaDeTarefas = this.listaDeTarefas.filter(item => item.id !== id);
  this.salvarTarefas(); // Salva a lista atualizada
}
```

### 6.3. Entendendo o Ciclo de Vida do Componente (Lifecycle Hooks)

Antes de carregarmos nossos dados, precisamos entender um conceito fundamental do Angular: o **Ciclo de Vida do Componente**.

Pense em um componente como um ser vivo: ele **nasce** (é criado e inserido na tela), **vive** (sofre alterações, como receber novos dados) e **morre** (é destruído e removido da tela).

O Angular nos permite "enganchar" (hook) nossa própria lógica em cada um desses momentos-chave. Esses "ganchos" são chamados de **Lifecycle Hooks**. Eles são métodos

especiais que o Angular executa automaticamente em pontos específicos do ciclo de vida de um componente.

### 6.4. Carregando os Dados com `ngOnInit`

Salvar os dados é apenas metade da solução. Precisamos carregá-los de volta do `localStorage` sempre que a aplicação for iniciada. O lugar perfeito para fazer isso é dentro do **Lifecycle Hook `ngOnInit()`**.

- **O que é `ngOnInit()`?** É o gancho do ciclo de vida que o Angular executa **uma única vez**, logo após o componente "nascer" (ser criado e inicializado). É o momento ideal para realizar tarefas de configuração inicial, como buscar dados de uma API ou, no nosso caso, carregar dados do `localStorage`.

1. Ainda no arquivo `home.component.ts`, adicione o método `ngOnInit()` à sua classe:

```
TypeScript
ngOnInit() {
  // 1. Tenta buscar os dados salvos no localStorage.
  const dadosSalvos = localStorage.getItem('listaDeTarefas');

  // 2. Se encontrar algum dado...
  if (dadosSalvos){
    // 3. ...converte a string JSON de volta para um array de objetos e
carrega na nossa lista.
    this.listaDeTarefas = JSON.parse(dadosSalvos);
  }
}
```

Com isso, sua aplicação está completa! Ela agora não apenas permite o gerenciamento de tarefas, mas também salva o progresso do usuário localmente.

## Seção 7: Código Final da Aplicação

Para facilitar a revisão e garantir que tudo esteja funcionando corretamente, aqui estão os códigos completos dos principais arquivos que modificamos ao longo deste tutorial. Você pode usar esta seção para comparar com o seu projeto.

---

**`src/app/components/home/home.component.ts`**

```
TypeScript
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

export interface TodoItem {
    id: number;
    task: string;
    finalizada: boolean;
}

@Component({
    selector: 'app-home',
    standalone: true,
    imports: [FormsModule],
    templateUrl: './home.component.html',
    styleUrl: './home.component.css'
})
export class HomeComponent {
    listaDeTarefas: TodoItem[] = [];
    novaTarefa: string = '';

    ngOnInit(){
        const dadosSalvos = localStorage.getItem('listaDeTarefas');
        if (dadosSalvos) {
            this.listaDeTarefas = JSON.parse(dadosSalvos);
        }
    }

    salvarTarefas(){
                                    localStorage.setItem('listaDeTarefas',
JSON.stringify(this.listaDeTarefas));
    }

    adicionarTarefa(){
        if (this.novaTarefa !== '') {
            const itemTarefa: TodoItem = {
                id: Date.now(),
                task: this.novaTarefa,
                finalizada: false,
            };
            this.listaDeTarefas.push(itemTarefa);
            this.novaTarefa = '';
            this.salvarTarefas();
        }
    }
```

```
    tarefaCompletada(tarefa: TodoItem){
        tarefa.finalizada = !tarefa.finalizada;
        this.salvarTarefas();
    }

    deletarTarefa(id: number){
         this.listaDeTarefas = this.listaDeTarefas.filter(item => item.id
!== id);
        this.salvarTarefas();
    }
}
```

---

### src/app/components/home/home.component.html

```
HTML
<div class="titulo">
    <h1>Lista de tarefas</h1>
</div>

<div class="adicionar_task">
         <input  [(ngModel)]="novaTarefa"  placeholder="Digite  uma  nova
tarefa..." (keyup.enter)="adicionarTarefa()"/>
    <button (click)="adicionarTarefa()">Adicionar Tarefa</button>
</div>

<ul class="listaDeTarefasParaOUsuarioVisualizar">
    @for (tarefa of listaDeTarefas; track tarefa.id) {
        <li [class.completada]="tarefa.finalizada">
            <input
                type="checkbox"
                [checked]="tarefa.finalizada"
                (change)="tarefaCompletada(tarefa)"
            />
            <span>{{ tarefa.task }}</span>
            <button (click)="deletarTarefa(tarefa.id)">Apagar</button>
        </li>
    }
</ul>
```

---

### src/app/app.routes.ts

```
TypeScript
```

```
import { Routes } from '@angular/router';
import { HomeComponent } from './components/home/home.component';

export const routes: Routes = [
    { path: "", component: HomeComponent }
];
```

## Subseção 1: Resumo da Sua Conquista.

Parabéns! Se você chegou até aqui, você não apenas leu um tutorial, mas construiu uma aplicação Angular completa, funcional e do zero. Desde a configuração do ambiente até a persistência de dados, você passou por todas as etapas fundamentais do desenvolvimento front-end moderno.

Reserve um momento para reconhecer o conhecimento que você adquiriu. Ao longo deste guia, você aprendeu a:

- **Configurar um ambiente de desenvolvimento** profissional com Node.js e Angular CLI.
- **Entender a arquitetura** de um projeto Angular, desde seus arquivos de configuração até a estrutura de componentes.
- **Criar e gerenciar componentes**, os blocos de construção de qualquer aplicação Angular.
- **Implementar um sistema de rotas** para criar uma Single Page Application (SPA) com múltiplas "páginas" virtuais.
- **Dar vida aos componentes** com estados, lógica e o poder do Data Binding (`ngModel`, Interpolação, Property e Event Binding).
- **Manipular a estrutura do HTML** dinamicamente com as diretivas `@if` e `@for`.
- **Centralizar e reutilizar código** de forma profissional com os Services.
- **Criar comunicação direta** entre componentes com `@Input` e `@Output`.
- **Persistir dados** no navegador usando `localStorage` e o Lifecycle Hook `ngOnInit`.

Os conceitos que você praticou aqui não são apenas para uma "To-do List". Eles são a base para construir qualquer tipo de aplicação web, de pequeno a grande porte.

O segredo para a maestria é a prática contínua. Por isso, não pare por aqui. A aplicação que você construiu é o seu laboratório. As próximas seções de **Desafios** e **Sugestões de Novos Projetos** foram criadas exatamente para te incentivar a experimentar, quebrar, consertar e, acima de tudo, continuar aprendendo.

Você deu um passo gigante na sua jornada como desenvolvedor. Continue construindo!

## Subseção 2: A Prática Leva à Maestria (Desafios)

Parabéns! Você construiu uma aplicação Angular completa e funcional do zero. O conhecimento que você adquiriu aqui é a base para criar projetos muito mais complexos.

Agora, a melhor forma de solidificar o que aprendeu é praticando. Propomos três desafios para você aprimorar sua To-do List. Eles irão testar suas habilidades e te introduzir a práticas do dia a dia de um desenvolvedor front-end.

---

**Desafio 1: O Toque do Artista (Estilização com CSS)**

Atualmente, nossa aplicação é funcional, mas visualmente simples. Seu primeiro desafio é usar suas habilidades de CSS para dar vida e personalidade à To-do List.

- **Sua Missão:** Abra o arquivo `home.component.css` e estilize todos os elementos. Mude cores, fontes, espaçamentos, adicione bordas e talvez até animações sutis. Como você pode tornar a interface mais agradável e intuitiva apenas com CSS?

---

**Desafio 2: O Foco no Usuário (Melhorias de UX)**

Uma boa aplicação não é apenas funcional, ela é fácil e agradável de usar. Este desafio se concentra em pequenas melhorias que fazem uma grande diferença na experiência do usuário (UX).

- **Sua Missão:**
    1. **Botão Inteligente:** Como você pode usar o *Property Binding* `[disabled]` para desabilitar o botão "Adicionar Tarefa" quando o campo de input estiver vazio, evitando a criação de tarefas em branco?
    2. **Adicionar com "Enter":** A maioria dos usuários espera poder adicionar uma tarefa pressionando a tecla "Enter". Como você pode usar um *Event Binding* de teclado no input para acionar a função `adicionarTarefa()`?
    3. **Mensagem de Lista Vazia:** O que acontece quando não há tarefas? Atualmente, a lista fica em branco. Como você pode usar o bloco `@empty` dentro do seu loop `@for` para exibir uma mensagem amigável, como "Nenhuma tarefa cadastrada. Adicione uma nova!"?

---

**Desafio 3: O Arquiteto (Reutilização de Código)**

Este é o desafio mais avançado e o que mais se aproxima de uma prática profissional em projetos grandes. Atualmente, toda a lógica está no `HomeComponent`. E se quiséssemos reutilizar a aparência de um item da lista em outra parte do site?

- **Sua Missão:**
    1. Crie um novo componente chamado `TodoItemComponent` (`ng g c components/todo-item`).
    2. Mova a estrutura HTML (`<li>...</li>`) e a lógica de um único item da lista para este novo componente.

3. No `HomeComponent`, seu loop `@for` agora irá renderizar o seletor `<app-todo-item>` em vez do `<li>`.
4. **A grande questão:** Como você fará os componentes conversarem?
    - Use **`@Input()`** para o `HomeComponent` (Pai) passar os dados da tarefa **para** cada `TodoItemComponent` (Filho).
    - Use **`@Output()`** para o `TodoItemComponent` (Filho) notificar o `HomeComponent` (Pai) quando uma ação acontecer (ex: "fui completado!" ou "fui deletado!").

## Subseção 3: O Que Vem a Seguir? (Sugestões de Novos Projetos)

Parabéns por concluir a To-do List! Você agora tem uma base sólida nos fundamentos do Angular. O segredo para se tornar um desenvolvedor proficiente é continuar construindo. Cada novo projeto é uma oportunidade para solidificar o que você já sabe e aprender um novo conceito crucial.

A seguir, apresentamos uma lista de projetos organizados por nível de dificuldade para guiar seus próximos passos.

---

## Nível 1: Consolidando o Básico e Introduzindo APIs

O objetivo aqui é reforçar seu conhecimento sobre componentes e data binding, enquanto aprende a habilidade mais importante para um desenvolvedor front-end: consumir dados de uma API externa.

**1. Aplicativo de Clima (Weather App)**

- **O que é?** Uma aplicação simples onde o usuário digita o nome de uma cidade e a interface exibe a temperatura atual, a condição do tempo (ensolarado, chuvoso, etc.) e talvez uma previsão.
- **O que você vai aprender de novo?**
    - **Consumo de APIs com `HttpClient`:** A principal novidade. Você aprenderá a fazer requisições HTTP para buscar dados em um servidor externo.
    - **Gerenciamento de Estado Assíncrono:** Lidar com o tempo de espera da resposta da API, exibindo uma mensagem de "carregando..." para o usuário.
- **Dicas de Pesquisa:**
    - **Português:** "tutorial angular app de clima", "angular httpclient api tempo".
    - **Inglês:** "angular weather app tutorial", "angular httpclient tutorial".
    - **APIs Gratuitas:** Procure por "OpenWeatherMap API" ou "WeatherAPI".

**2. Buscador de Receitas ou Filmes (Movie/Recipe Finder)**

- **O que é?** Uma interface com um campo de busca. O usuário digita um termo (ex: "pizza" ou "Matrix") e a aplicação exibe uma lista de resultados. Ao clicar em um resultado, o usuário é levado para uma página com mais detalhes.
- **O que você vai aprender de novo?**

    - **Roteamento com Parâmetros:** Criar rotas dinâmicas (ex: `/filme/123`), onde `123` é o ID do filme, e aprender a capturar esse ID para buscar os detalhes específicos na API.
    - **Estruturas de Dados Mais Complexas:** Lidar com respostas de API que são arrays de objetos com mais informações.
- **Dicas de Pesquisa:**
    - **Português:** "angular app de filmes tutorial", "angular roteamento com parametros".
    - **Inglês:** "angular movie app tutorial", "angular routing with parameters".
    - **APIs Gratuitas:** "The Movie DB (TMDB) API", "TheMealDB API".

---

## Nível 2: Aprofundando em Formulários e Interatividade

Neste nível, o foco é em interações mais complexas, especialmente em como lidar com formulários de maneira robusta e escalável usando os **Formulários Reativos** do Angular.

**3. Clone do Front-end de um Blog**

- **O que é?** A interface de um blog onde você pode listar posts, clicar para ler um post completo e, o mais importante, ter uma página para criar um novo post através de um formulário com título e corpo de texto.
- **O que você vai aprender de novo?**
    - **Formulários Reativos (Reactive Forms):** A forma mais poderosa e profissional de gerenciar formulários no Angular.
    - **Validação de Formulários:** Aprender a adicionar regras de validação (ex: o título não pode estar vazio, o texto precisa ter um mínimo de caracteres).
- **Dicas de Pesquisa:**
    - **Português:** "angular formulários reativos tutorial", "validação de formulários angular".
    - **Inglês:** "angular reactive forms tutorial", "angular form validation".

**4. Front-end de um E-commerce Simples (Vitrine de Produtos)**

- **O que é?** Uma página que exibe uma lista de produtos. O desafio aqui é criar filtros interativos (por categoria, por faixa de preço) e uma barra de busca que atualize a lista de produtos em tempo real.
- **O que você vai aprender de novo?**
    - **Filtragem e Manipulação de Dados no Front-end:** Implementar lógicas para filtrar e ordenar arrays de objetos com base na interação do usuário.
    - **Componentização Avançada:** Praticar a criação de componentes reutilizáveis, como um `<app-card-de-produto>`.
    - **(Desafio Extra) Carrinho de Compras:** Criar um `Service` para gerenciar o estado do carrinho, uma excelente forma de entender o gerenciamento de estado global.
- **Dicas de Pesquisa:**
    - **Português:** "angular clone loja virtual", "filtrar dados angular", "angular

carrinho de compras service".
    - **Inglês:** "angular ecommerce tutorial", "angular filter data tutorial", "angular shopping cart service".

---

## Subseção 4: Compartilhe seu Sucesso!

Completou um ou todos os desafios? Você acaba de dar um passo gigante no seu desenvolvimento! Ficaríamos muito felizes em ver o resultado.

Tire um print da sua aplicação, compartilhe o link do seu repositório no GitHub ou até mesmo grave um pequeno vídeo mostrando as funcionalidades. Publique no **LinkedIn** ou em outras redes sociais marcando seu progresso. Isso não apenas inspira outros desenvolvedores, mas também mostra ao mercado sua dedicação e suas novas habilidades com Angular!

Use hashtags como `#Angular` `#FrontEnd` `#DesenvolvimentoWeb` `#MeuPrimeiroAppAngular` para que mais pessoas possam ver seu trabalho.

**Boa sorte e divirta-se codificando!**