Introdução

O Angular, desenvolvido pelo Google, é uma plataforma/framework que segue uma arquitetura do tipo MVC (Model-View-Controller) para facilitar a criação de aplicações web interativas. Com elementos pré-definidos como componentes, serviços, módulos e data binding, ele permite que desenvolvedores construam desde SPAs até aplicativos para dispositivos móveis e desktop.

Desde a versão 2 (2016), o Angular é escrito em TypeScript, sendo transpilado para JavaScript, e passou a adotar template HTML com marcações, componentes, serviços, módulos, metadados, data binding, diretivas e injeção de dependências como partes fundamentais de sua arquitetura.

Com o lançamento do Angular 20 (28 de maio de 2025), a plataforma se modernizou ainda mais, introduzindo reatividade baseada em Signals, detecção de mudanças zoneless, hidratação incremental para SSR, novas sintaxes de controle de fluxo, e diversos aprimoramentos no diagnóstico e na experiência do desenvolvedorEm termos gerais, o processo de desenvolvimento de aplicações Angular envolve os seguintes passos:

Criar templates HTML: Isso envolve a definição de trechos de código HTML com marcações Angular.

Escrever classes (componentes) para gerenciar os templates: Os componentes são responsáveis por controlar a lógica de exibição e interação do template HTML associado.

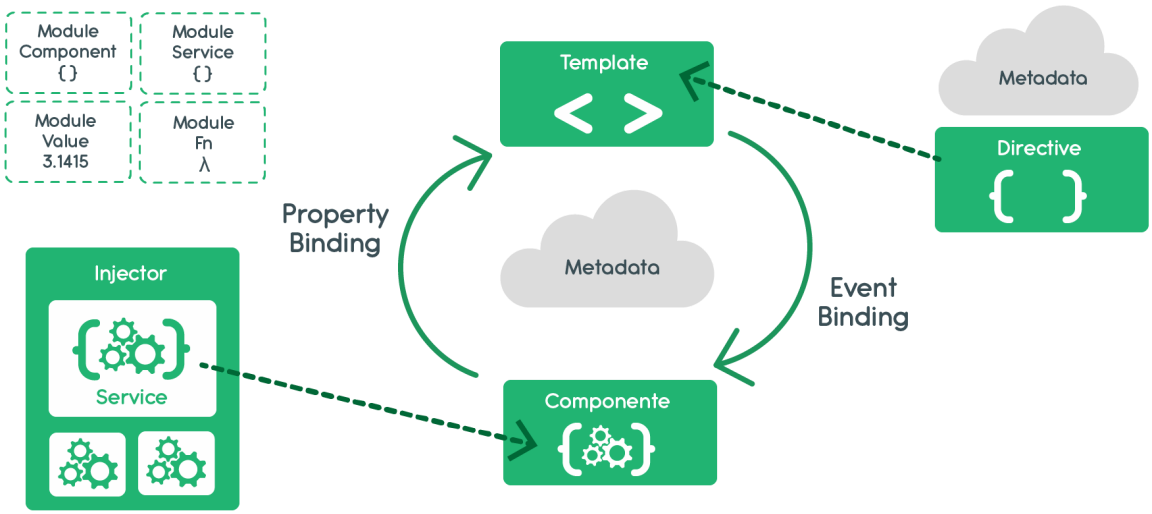
Adicionar lógica de negócio em serviços: Os serviços são usados para encapsular a lógica de negócio da aplicação, separando-a dos componentes para promover a reutilização e a modularidade.

Empacotar componentes e serviços em módulos: Os módulos ajudam a organizar e agrupar componentes e serviços relacionados, tornando o código mais organizado e mantendo a separação de preocupações.

Nas próximas seções, exploraremos cada um desses elementos de forma mais detalhada. Não se preocupe em memorizar todos os detalhes agora, pois abordaremos cada um deles ao longo do curso.

Principais itens Angular

De forma geral, a imagem abaixo representa uma aplicação Angular.



Módulos

Uma aplicação Angular é formada pelo menos por um módulo, o módulo raiz, tipicamente chamado de **AppModule**. Um módulo representa uma unidade coesa para o negócio ao qual a aplicação está sendo construída. Por exemplo, num sistema bancário, poderíamos dizer que a **área de relatórios** seria um módulo, contendo diversos componentes, já o **controle de contas** seria outro módulo.

Angular é formado por uma biblioteca de componentes e módulos que usamos dentro dos nossos. Por exemplo, em nossos módulos podemos usar o módulo angular abaixo:

```
import { BrowserModule } from '@angular/platform-browser';
```

Não há uma regra específica para se dividir a aplicação em módulos, mas dividir o problema negocial em unidades coesas é o caminho mais seguro.

Standalone Component

Um Standalone Component é um componente Angular que não precisa estar declarado em um módulo (NgModule) para ser usado.

Ele se declara como independente, com a propriedade standalone: true nos metadados do @Component.

Pode importar diretamente outros componentes, diretivas e pipes, sem depender da declaração dentro de NgModules.

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

import { CommonModule } from '@angular/common';


@Component({
  selector: 'hello-world',
  standalone: true,   //    Torna o componente independente
```

```
  imports: [CommonModule], // importa pipes/diretivas necessárias
  template: `<h1>Hello {{ name }}!</h1>`,
})
export class HelloWorldComponent {
  name = 'Angular 20';
}
```

Componentes

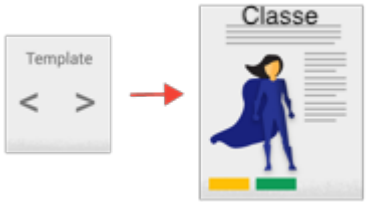
Um **componente** controla um trecho da tela (chamada visão). Por exemplo, podemos fazer um componente para listar todas as contas bancárias de um cliente, ou mesmo um componente que altere os dados de uma conta bancária. Definimos a lógica de aplicação do componente dentro de classes, escritas em Typescript. Um exemplo de classe pode ser vista abaixo (por enquanto, não se preocupe com os detalhes do código).

```
export class ContaListComponent implements OnInit {
  contas: Conta[];
  contaSelecionada: Conta;

  constructor(private service: ContaService) { }

  ngOnInit() {
    this.contas = this.service.getContas();
  }

  selecionarConta(conta: Conta) { this.contaSelecionada = conta; }

}
```

Essas classes interagem com a visão (tela) via uma API de propriedades e métodos da classe. O que se faz na tela normalmente se reflete nas propriedades da classe, e ao alterar as propriedades das classes as telas mostram essa alteração. Essas telas são desenvolvidas em templates, que veremos a seguir.

Templates

Um **template** é um trecho de tela (ou uma tela inteira) escrita em "HTML". A parte da visão do componente, ou seja, o que será mostrada para o usuário é definida no template. Perceba, pela figura abaixo, a interação entre o template e a classe que trata o template. Uma analogia para quem já programou/programa em JSF, guardadas as devidas proporções, seria que ***um Template está para uma Visão, assim como um Componente está para um Backing-Bean.*** Porém, é importante lembrar que a visão do JSF transforma-se no HTML que é mostrado no navegador do cliente, assim como o template em Angular, mas o backing-bean de JSF executa no servidor, diferentemente do Componente de Angular, que também executa no navegador do cliente (é JavaScript, como já sabemos).



Apesar da sintaxe básica do template ser HTML, na realidade há trechos de código que fazem parte da linguagem Angular, ou seja, não é um HTML puro. Por exemplo, no template abaixo listamos os clientes de uma conta.

```
<h2>Listagem de Contas</h2>

<p><i>Selecione uma conta</i></p>
<ul>
  <li *ngFor="let conta of contas" (click)="selecionarConta(conta)">
    {{conta.agencia}}-{{conta.numero}}
  </li>

</ul>
```

Perceba que há itens que não são HTML, mas sim de Angular. O *ngFor, o (click), as chaves {{}}. A propriedade **contas** vem da classe que trata essa tela, assim como o método selecionarConta().

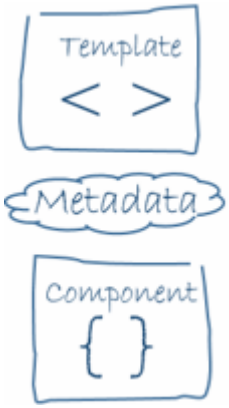
Metadados

Metadados é a forma de dizer ao Angular como processar uma classe. Uma classe em Typescript é apenas uma classe, até que você diga ao Angular que ela faz parte de um componente. Um exemplo de metadados é mostrado abaixo.

```
@Component({
  selector:    'conta-list',
  templateUrl: './conta-list.component.html',
  providers:  [ ContaService ]
})
export class ContaListComponent implements OnInit {
/* . . . */

}
```

Neste exemplo, usamos metadados para informar ao Angular que a classe ContaListaComponent é um componente (@Component), que o seletor (tag) deste componente será conta-list (ou seja, em HTML, ao usar <conta-list> será mostrada a tela de listagem de contas. Acabamos de criar uma nova tag em HTML ;)), que o template desse componente está no arquivo conta-list.component.html e que usamos o serviço ContaService. Maiores detalhes de quais propriedades usar nos metadados veremos depois. Perceba também que o formato passado para o @Componente é um JSON.

A junção de template, metadados e o componente descreve uma visão (tela).



Data binding

É importante que os frameworks implementem formas de interligar os dados que serão apresentados na tela (template) com as propriedades do componente que trata a tela (classe). O mecanismo que faz isso chamamos *data binding*. É uma forma de tratar dados muito poderosa em Angular, e a forma que foi feita se abstrai todos os detalhes complexos de tratamento de elementos em Javascript (DOM), ou seja, não precisamos mais ficar pegando diretamente os elementos de DOM e alterando na mão. Angular fará isso por você ;).

Basicamente, *data binding* realiza o fluxo de dados que vai da tela para o componente e que vem do componente para a tela. Vejamos um exemplo.

```
<li>{{conta.numero}}</li>
<conta-detalhe [conta]="contaSelecionada"></conta-detalhe>

<li (click)="contaSelecionada(conta)"></li>
```

No exemplo acima temos três formas de *data binding:*

1. **Interpolação de string**: ao fazer {{conta.numero}}, estou informando que é para mostrar na tela o valor dessa propriedade que está no Componente;
2. **Property binding**: ao fazer uso do [conta], estou informando que a contaSelecionada virá do componente (classe) e será chamado de **conta** no template;
3. **Event binding**: ao definir o (click), estou informando que ao ocorrer o evento click no elemento HTML será executado o método contaSelecionada, passando a conta como parâmetro. Perceba que esse tratamento de evento não está sendo feito diretamente em Javascript, mas que por trás dos panos será tratado o onClick.

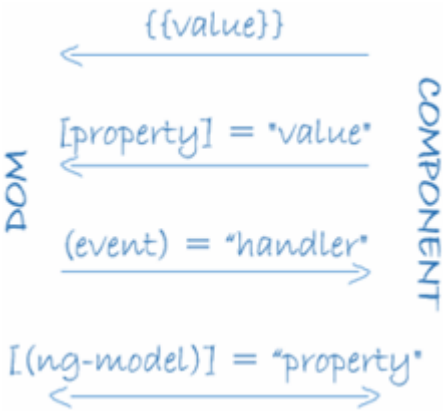
Perceba que o **property binding** define que um elemento terá o valor de uma propriedade do componente, ou seja, o valor flui da classe para tela. Já o **event binding** gera um fluxo da tela para o componente (classe).

Para simplificar, o Angular definiu o **two-way binding**, que é a junção entre **property binding** e **event binding**, como pode ser visto abaixo.

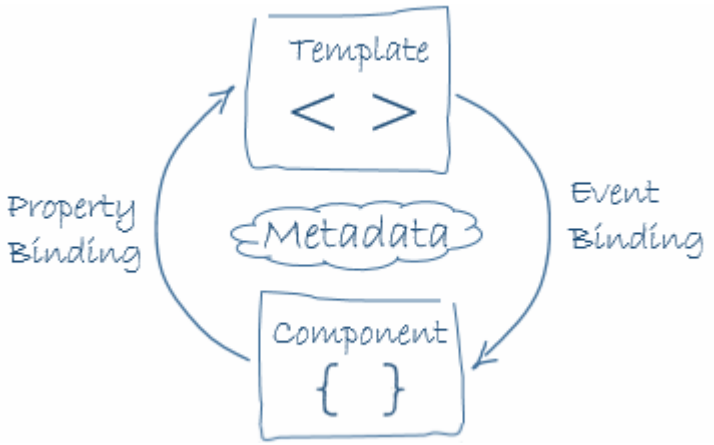
```
<input [(ngModel)]="conta.numero">
```

Por essa sintaxe, tanto está sendo feito o tratamento de evento (onChange) quanto a ligação com a propriedade do componente. Em resumo, se no componente for mudado o número da conta, na tela será mudado o valor, e se na tela mudar o valor, no componente será refletida

essa mudança. Essa sintaxe [()] é chamada de **caixa de bananas**. Ao fazer uso do [(ngModel)], estamos fazendo o two-day data binding. A imagem abaixo resume as formas de data binding.



A imagem abaixo mostra o fluxo de dados entre o template e o componente.



Além disso, o Angular 20 introduz reatividade via Signals, uma forma mais eficiente de reagir a mudanças sem depender do Zone.js

Diretivas

Diretivas são elementos de Angular que alteram o DOM Javascript, consequentemente alteram a visualização do template (tela). Basicamente há dois tipos de diretivas: **estruturais** e de **atributos**.

Angular 20 traz o controle de fluxo embutido, como @if, @for, @switch, substituindo sintaxes antigas e reduindo boilerplate

Diretivas estruturais alteram o layout (o template) alterando, inserindo ou removendo elementos do DOM. Exemplos desse tipo de diretiva podem ser vistos abaixo.

```
<li *ngFor="let conta of contas"></li>
```

```
<conta-detalhe *ngIf="contaSelecionada"></conta-detalhe>
```

O *ngFor gerará um <li> para cada conta que exista na propriedade contas do componente, e o *ngIf renderizará (mostrará) o detalhe de uma conta apenas se uma conta foi selecionada, ou seja, se contaSelecionada existir.

Diretivas de atributos mudam o layout (aparência) ou comportamento de elementos que já existem. Dentro dos templates, esse tipo de diretiva parecem atributos normais de html, por isso o nome.

Já vimos acima uma diretiva desse tipo, a ngModel, ao realizar o two-way databinding:

```
<input [(ngModel)]="conta.numero">
```

Serviços

Serviço engloba qualquer valor, função ou *feature* (característica) que sua aplicação precisa. Não há algo específico no Angular que diga que algo é um serviço, mas sua aplicação definirá diversos serviços para ajudar na comunicação com o backend, por exemplo.

Exemplos de serviços poderiam ser:

- Serviço de configuração da aplicação
- Serviço de log
- Serviço de dados para comunicação com o backend

Serviços estão por todo canto, e é peça chave em Angular. É importante deixar claro que componentes poderiam fazer o papel de serviços, indo, por exemplo, pegar dados no backend. Porém, nada deste tipo deve estar em componentes. Esses devem apenas melhorar a experiência do usuário, atualizando telas, mudando o visual etc. A parte mais ligada ao negócio de sua aplicação deveria estar em serviços. Abaixo vemos um exemplo de serviço, que pega dados de contas do backend (não se preocupe ainda com a sintaxe):

```
export class ContaService {
  private contas: Conta[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getContas() {
    this.backend.getAll(Conta).then( (contas: Conta[]) => {
      this.logger.log(`Carregadas ${contas.length} contas.`);
      this.contas.push(...contas);
    });
    return this.contas;
  }

}
```

Perceba que a ContaService tem um método chamado getContas() que carrega todas as contas do backend a partir do serviço BackendService. É comum um serviço usar o trabalho de outro, e a criação do objeto do serviço BackendService, por exemplo, é feito a partir do que chamamos de injeção de dependência, que é a forma de entregar uma nova instância de uma classe com todas as dependências necessárias.

O que vimos até agora é a base da programação Angular. Veremos cada um desses itens em detalhes a partir de agora.

Recursos

- Tutorial oficial do Angular: https://angular.io/tutorial. Esse texto é um resumo de parte desse tutorial: https://angular.io/guide/architecture
- Curso no Youtube: Não avaliei a qualidade, mas como introdução pode ser legal: https://www.youtube.com/watch?v=tPOMG0D57S0