## **1. Organização do código**

A organização do código-fonte é fundamental para a manutenção e escalabilidade de

projetos Angular.

### **Estrutura de pastas**

* **Estrutura por funcionalidade:** organize os arquivos por funcionalidade em vez de tipo. Por exemplo, em vez de ter uma pasta com todos os componentes, agrupe-os em uma pastas que faça sentido ao seu contexto, como por exemplo: serviços e módulos relacionados a essa funcionalidade.
* **Nomeação de arquivos:** adote um padrão de nomenclatura consistente e claro. Use o padrão nome-do-componente.component.ts para componentes, nome-do-servico.service.ts para serviços e assim por diante.

### **Estrutura dos módulos (Modules)**

* **Módulos de funcionalidade (Feature Modules):** divida sua aplicação em módulos de funcionalidade para evitar o carregamento excessivo de código desnecessário. Isso permite que o lazy loading seja aplicado.
* **Módulos compartilhados (Shared Modules):** crie um módulo compartilhado para componentes, diretivas e pipes que são usados em múltiplos módulos de funcionalidade. Isso evita a duplicação de código.
* **Módulos de roteamento (Routing Modules):** separe a configuração de rotas em um arquivo de módulo de roteamento dedicado. Por exemplo, dashboard-routing.module.ts.

## **2. Padrões de código**

A consistência na escrita do código facilita a leitura, o entendimento e a colaboração em equipe.

### **Componentes**

* **Responsabilidade única (Single Responsibility Principle):** cada componente deve ter uma única responsabilidade. Por exemplo, um componente deve ser responsável por exibir dados, enquanto outro é responsável por manipulá-los.
* **Componentes de apresentação e contêiner:** separe a lógica de negócios da apresentação da interface do usuário. Componentes de contêiner contêm a lógica e manipulam o estado, enquanto componentes de apresentação recebem os dados de entrada (@Input) e emitem eventos (@Output).

### **Serviços**

* **Serviços para lógica de negócio:** a lógica de negócio deve residir em serviços, e não em componentes. Isso mantém o código organizado e os serviços podem ser facilmente testados e reutilizados.
* **Serviços injetáveis:** use a injeção de dependência para fornecer serviços aos componentes. Use @Injectable({ providedIn: 'root' }) para serviços que precisam ser singleton em toda a aplicação.

## **3. Gerenciamento de estado**

O gerenciamento de estado é crucial para aplicações complexas.

### **Bibliotecas de gerenciamento de estado**

* **Utilize bibliotecas como o NGRX** para um gerenciamento de estado previsível. Essas bibliotecas centralizam o estado da aplicação em um único objeto, tornando-o mais fácil de rastrear e depurar.

## **4. Performance**

Otimizar a performance da aplicação é essencial para uma boa experiência do usuário.

### **Estratégias de carregamento**

* **Lazy Loading (Carregamento tardio):** use o lazy loading para carregar módulos sob demanda, ou seja, apenas quando o usuário navegar para a rota correspondente. Isso diminui o tempo de carregamento inicial.

### **Estratégias de detecção de mudanças**

* **OnPush Change Detection Strategy:** use a estratégia de detecção de mudanças **OnPush** para melhorar a performance. Com essa estratégia, a detecção de mudanças só é ativada quando as entradas do componente (@Input) mudam, ou quando um evento é emitido.

## **5. Testes**

Escrever testes é uma parte fundamental do desenvolvimento de software.

### **Tipos de testes**

* **Testes unitários:** garanta que cada parte de seu código (componentes, serviços, etc.) funcione como esperado de forma isolada.
* **Testes de integração:** verifique se as diferentes partes da sua aplicação funcionam juntas corretamente.

## **6. Documentação**

A documentação do código é vital para a colaboração em equipe.

**As Nove Regras do Object Calisthenics Aplicadas ao Angular**

### 

### **1. Um nível de indentação por método**

Essa regra é a base para a criação de **métodos curtos e focados**, que fazem apenas uma

coisa. Em vez de aninhar múltiplos if ou for loops, a lógica complexa deve ser extraída

para métodos auxiliares privados. Isso aumenta a legibilidade e facilita a depuração.

**Exemplo:** Imagine que você precisa processar uma lista de usuários e notificá-los.

TypeScript

// Código ruim: Múltiplos níveis de indentação

public processAndNotifyUsers(): void {

this.userService.getUsers().subscribe(users => {

if (users && users.length > 0) {

for (const user of users) {

if (user.isEligibleForNotification()) {

this.notificationService.notifyUser(user.id);

}

}

}

});

}

A versão acima tem muitos níveis de indentação, tornando-a difícil de ler e entender. Agora,

veja a versão refatorada.

TypeScript

// Código bom: Um nível de indentação por método

public processAndNotifyUsers(): void {

this.userService.getUsers().subscribe(users => {

this.handleUsers(users);

});

}

private handleUsers(users: User[]): void {

if (!this.isValidUserList(users)) {

return;

}

this.sendNotifications(users);

}

private isValidUserList(users: User[]): boolean {

return users && users.length > 0;

}

private sendNotifications(users: User[]): void {

users.forEach(user => {

if (user.isEligibleForNotification()) {

this.notificationService.notifyUser(user.id);

}

});

}

A segunda versão é mais clara e cada método tem um objetivo único e bem definido.

### **2. Não use a palavra-chave else**

O else pode ser substituído por **guard clauses** ou **polimorfismo**. Essa regra elimina

caminhos de execução alternativos explícitos, o que simplifica o fluxo de controle e reduz a

complexidade ciclomática.

**Exemplo:** Suponha que você tenha um método para definir um status.

TypeScript

// Código ruim: Uso do 'else'

public setStatus(value: string): void {

if (value === 'active') {

this.status = 'approved';

} else {

this.status = 'rejected';

}

}

O código acima é simples, mas imagine a complexidade com múltiplos if-else if-else.

A alternativa é usar um retorno antecipado.

TypeScript

// Código bom: Sem 'else'

public setStatus(value: string): void {

if (value === 'active') {

this.status = 'approved';

return;

}

this.status = 'rejected';

}

A segunda abordagem é mais direta e **reduz a ramificação** do código.

### **3. Evite o uso de primitivos**

Em vez de passar múltiplos parâmetros primitivos para um método, **encapsule-os em uma**

**classe ou objeto**. Isso torna a interface do método mais clara, menos propensa a erros de

ordem e mais fácil de estender no futuro.

**Exemplo:** Para criar um usuário, você poderia ter o seguinte método.

TypeScript

// Código ruim: Uso de primitivos

public createUser(firstName: string, lastName: string, email: string, role: string, isActive: boolean): void {

// Lógica para criar o usuário

}

O problema aqui é que a ordem dos parâmetros importa e a lista pode crescer. A solução é

criar um **Data Transfer Object (DTO)**.

TypeScript

// Código bom: Uso de objetos

interface UserDto {

firstName: string;

lastName: string;

email: string;

role: string;

isActive: boolean;

}

public createUser(userData: UserDto): void {

// Lógica para criar o usuário usando userData

}

Isso torna o método createUser mais robusto e auto-documentado.

### **4. Primeira classe de coleções**

Em vez de passar um array diretamente, crie uma classe que encapsule essa coleção e

adicione **métodos que operam sobre ela**. Isso centraliza a lógica relacionada à coleção e

a torna mais coesa.

**Exemplo:** Imagine que você tem uma lista de produtos.

TypeScript

// Código ruim: Manipulação direta de array

public calculateTotalValue(products: Product[]): number {

return products.reduce((total, p) => total + p.price, 0);

}

A lógica para o cálculo está fora do contexto da coleção. A aplicação da regra seria:

TypeScript

// Código bom: Primeira classe de coleções

class ProductCollection {

private products: Product[];

constructor(products: Product[]) {

this.products = products;

}

public getTotalValue(): number {

return this.products.reduce((total, p) => total + p.price, 0);

}

public getAvailableProducts(): Product[] {

return this.products.filter(p => p.isAvailable);

}

}

// Uso no componente

const productList = new ProductCollection(this.products);

const total = productList.getTotalValue();

Essa abordagem garante que a lógica da coleção permaneça no local correto.

### **5. Um ponto por linha**

Evite **cadeias de chamadas de método** (method chaining). Cada linha de código deve ter

no máximo um ponto (.). Isso evita os "trens de mensagem" e força a extração de

responsabilidades para variáveis intermediárias, tornando o código mais legível e fácil de

depurar.

**Exemplo:** Para obter o nome do usuário em maiúsculas a partir de um objeto aninhado.

TypeScript

// Código ruim: Múltiplos pontos

const userName = this.user.profile.name.toUpperCase();

O código acima é conciso, mas difícil de depurar se qualquer parte do caminho for null ou

undefined. A regra sugere a seguinte refatoração:

TypeScript

// Código bom: Um ponto por linha

const userProfile = this.user.profile;

const userName = userProfile.name;

const formattedName = userName.toUpperCase();

Com essa abordagem, você pode inspecionar o valor de cada variável intermediária.

### **6. Não use getters e setters**

Em vez de ter métodos que simplesmente retornam ou definem um valor (get e set), crie

**métodos que tenham comportamento**. Isso incentiva a criação de objetos que realmente

façam algo, em vez de serem apenas "recipientes de dados".

**Exemplo:** Uma classe para um Product.

TypeScript

// Código ruim: Apenas dados com getters e setters

class Product {

private \_price: number;

public getPrice(): number {

return this.\_price;

}

public setPrice(value: number): void {

this.\_price = value;

}

}

A lógica para manipular o preço fica em outro lugar. A regra sugere que a lógica seja

movida para dentro da classe.

TypeScript

// Código bom: Objeto com comportamento

class Product {

constructor(private price: number) {}

public applyDiscount(percentage: number): void {

this.price -= this.price \* (percentage / 100);

}

public getFinalPrice(): number {

return this.price;

}

}

Nesta versão, a classe Product **possui o comportamento** de "aplicar desconto", o que a

torna mais coesa.

### **7. Não use abreviações de classes/membros**

Use nomes de classes, métodos e variáveis que sejam **descritivos e claros**. Evite

abreviações que possam causar confusão, mesmo que pareçam óbvias no momento. O

código é lido com muito mais frequência do que é escrito.

**Exemplo:** Ao criar um serviço para buscar usuários.

TypeScript

// Código ruim: Abreviações

public getUsrById(id: number): Observable<User> { ... }

// Código bom: Nomes completos e claros

public getUserById(id: number): Observable<User> { ... }

Essa regra é sobre **comunicação**. O nome getUserById é inequivocamente claro para

qualquer pessoa que leia o código.

### **8. Evite mais de duas variáveis de instância**

Um componente ou serviço com muitas variáveis de estado (propriedades) é um sinal de

que ele está fazendo muitas coisas. Considere **refatorar** a lógica e extrair

responsabilidades para outros serviços ou classes.

**Exemplo:** Um componente com muitas variáveis de estado.

TypeScript

// Código ruim: Componente com muitas responsabilidades

export class UserProfileComponent {

public user: User;

public userPosts: Post[];

public isEditing: boolean;

public postLoading: boolean;

public errorMessage: string;

public notifications: Notification[];

// ... e mais

}

Esse componente é difícil de entender e manter. A solução é extrair as responsabilidades.

TypeScript

// Código bom: Componente com responsabilidades segregadas

export class UserProfileComponent {

public user: User;

public uiState: UIState; // uiState é uma classe que encapsula o estado de UI

constructor(

private postService: PostService,

private notificationService: NotificationService

) {}

}

A nova versão é mais limpa e direciona a lógica complexa para serviços dedicados.

### **9. Não use estáticos (static)**

Métodos e propriedades estáticas não podem ser facilmente testados, injetados ou

substituídos. Em vez de usá-los, prefira a **injeção de dependência** para fornecer a

funcionalidade necessária.

**Exemplo:** Um serviço de utilidades com um método estático.

TypeScript

// Código ruim: Método estático

class StringUtils {

public static toUpperCase(text: string): string {

return text.toUpperCase();

}

}

// Uso

const formattedText = StringUtils.toUpperCase('hello world');

O método acima não pode ser mockado em testes. A abordagem correta em Angular é usar

um serviço injetável.

TypeScript

// Código bom: Serviço injetável

@Injectable({

providedIn: 'root'

})

class StringFormatService {

public toUpperCase(text: string): string {

return text.toUpperCase();

}

}

// Uso em um componente ou serviço

constructor(private stringFormatService: StringFormatService) {}

public formatText(text: string): void {

const formattedText = this.stringFormatService.toUpperCase(text);

console.log(formattedText);

}

Com a injeção de dependência, a classe StringFormatService pode ser facilmente

substituída por um mock durante os testes.

A aplicação dessas regras, embora possa parecer rígida no começo, resulta em um código

**mais coeso, testável e de fácil manutenção**.

## **Programação defensiva**

## **1. Validação de Dados de Entrada (@Input)**

Quando um componente filho recebe dados de um componente pai através do decorador

@Input, é essencial garantir que esses dados são válidos antes de usá-los. Nunca assuma

que os dados de entrada estarão no formato ou valor esperado.

### 

### **Exemplo:**

Um que seu componente user-profile recebe um objeto user como entrada.

TypeScript

// Código não defensivo

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

@Component({

selector: 'app-user-profile',

template: `

<h2>{{ user.name }}</h2>

<p>Email: {{ user.email }}</p>

`

})

export class UserProfileComponent {

@Input() user: any;

}

**Problema:** Se o componente pai passar um objeto user nulo, user.name e user.email

causarão um erro de tempo de execução.

### **Solução Defensiva:**

Use o **Elvis operator (?.)** ou **ngIf** para fazer checagens de nulidade e garantir que a

renderização da tela ocorra somente se os dados estiverem disponíveis.

TypeScript

// Código defensivo

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

@Component({

selector: 'app-user-profile',

template: `

<div \*ngIf="user">

<h2>{{ user.name }}</h2>

<p>Email: {{ user.email }}</p>

</div>

<div \*ngIf="!user">

<p>Nenhum perfil de usuário disponível.</p>

</div>

`

})

export class UserProfileComponent implements OnInit {

@Input() user: any;

ngOnInit(): void {

if (!this.user) {

// Opcional: Lançar um erro ou registrar no console para depuração

console.error('O componente UserProfileComponent requer um objeto de usuário válido.');

}

}

}

O componente só tentará acessar as propriedades de user se ele

realmente existir.

## **2. Tratamento de Erros em Chamadas Assíncronas**

As chamadas a serviços externos (APIs) são a principal fonte de erros inesperados. É fundamental tratar esses erros para evitar que a aplicação quebre.

### **Exemplo:**

Uma chamada HTTP a um serviço de API.

TypeScript

// Código não defensivo

this.http.get<any>('/api/data').subscribe(data => {

this.data = data;

});

**Problema:** Se a chamada falhar (por exemplo, erro 404, 500, sem conexão), o

Observable não emitirá um valor e a lógica de erro não será tratada, podendo levar a um

estado inconsistente da aplicação.

### **Solução Defensiva:**

Use o operador catchError do RxJS para interceptar e lidar com erros.

TypeScript

// Código defensivo

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

this.http.get<any>('/api/data').pipe(

catchError(error => {

// Lógica para tratar o erro

console.error('Ocorreu um erro ao buscar os dados:', error);

// Retornar um Observable que lança um erro para a lógica de 'subscribe'

return throwError(() => new Error('Falha ao carregar os dados.'));

})

).subscribe({

next: data => {

this.data = data;

},

error: err => {

// Lógica para exibir uma mensagem de erro na UI

this.errorMessage = 'Não foi possível carregar os dados. Tente novamente mais tarde.';

}

});

Essa abordagem garante que, mesmo que a chamada à API falhe, a aplicação não irá

parar. Uma mensagem de erro pode ser exibida para o usuário, melhorando a experiência.

## **3. Gerenciamento de Subscrições (Subscriptions)**

É fácil criar vazamentos de memória em aplicações Angular se as subscrições a

Observables não forem canceladas.

### **Exemplo:**

Um componente que se inscreve em um Observable de um serviço.

TypeScript

// Código não defensivo (leva a vazamento de memória)

export class MyComponent implements OnInit {

ngOnInit() {

this.dataService.getData().subscribe(data => {

// ...

});

}

}

**Problema:** Quando o componente MyComponent é destruído, a subscrição ainda estará

ativa, o que pode causar vazamentos de memória e erros.

### **Solução Defensiva:**

Existem três maneiras comuns e seguras de gerenciar subscrições:

1. **Usar o pipe async no template:** O Angular gerencia automaticamente a subscrição e a desinscrição.
2. **Cancelar manualmente no ngOnDestroy:** Armazene a subscrição e use unsubscribe() no ciclo de vida do componente.
3. **Usar um Subject para gerenciar a vida útil da subscrição:** Uma técnica avançada para cenários mais complexos.

TypeScript

// Código defensivo (Opção 2 - Cancelamento manual)

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

import { Subscription } from 'rxjs';

export class MyComponent implements OnInit, OnDestroy {

private dataSubscription: Subscription;

ngOnInit() {

this.dataSubscription = this.dataService.getData().subscribe(data => {

// ...

});

}

ngOnDestroy() {

// Cancela a subscrição ao destruir o componente

if (this.dataSubscription) {

this.dataSubscription.unsubscribe();

}

}

}

## **4. Uso de Classes e Interfaces para Tipagem Forte**

A programação defensiva se beneficia enormemente da tipagem forte do TypeScript.

### **Exemplo:**

Em vez de usar tipos genéricos como any, use interfaces para definir a estrutura dos seus

dados.

TypeScript

// Código não defensivo

let user: any = {};

user.email = 123; // Nenhuma validação em tempo de compilação

### **Solução Defensiva:**

Defina uma interface para seu objeto user. O TypeScript irá garantir que o objeto siga a

estrutura definida.

TypeScript

// Código defensivo

export interface User {

id: number;

name: string;

email: string;

}

// O compilador irá alertar sobre o tipo incorreto

let user: User = {

id: 1,

name: 'John Doe',

email: 'john@example.com'

};

user.email = 123; // Erro de compilação: Type 'number' is not assignable to type 'string'.

Essa validação em tempo de compilação previne uma série de erros lógicos que só seriam

descobertos em tempo de execução.

### **Software Skills**

### **1. Comunicação**

A comunicação é a base de um bom trabalho em equipe. Um desenvolvedor precisa ser

capaz de se expressar de forma clara e objetiva.

* **Comunicação técnica:** Ser capaz de explicar conceitos técnicos complexos para colegas não técnicos e vice-versa.
* **Ouvir ativamente:** Prestar atenção às preocupações e ideias dos outros membros da equipe para entender os requisitos e os desafios de forma completa.

### **2. Colaboração**

Um desenvolvedor não trabalha isoladamente. A capacidade de colaborar é crucial para o

sucesso de um projeto.

* **Trabalho em equipe:** Contribuir ativamente para as discussões, compartilhar conhecimento e ajudar os colegas. O foco deve ser no sucesso do time, não apenas no individual.
* **Feedback construtivo:** Saber dar e receber feedback sobre o código (code reviews) de maneira profissional e respeitosa. Com o objetivo de melhorar a qualidade do código, não criticar as pessoas.
* **Flexibilidade:** Estar aberto a diferentes abordagens e disposto a adaptar-se a mudanças no projeto ou na forma de trabalhar da equipe.

### **3. Resolução de Problemas**

Em frontend, problemas surgem constantemente. Uma boa soft skill é a capacidade de

abordar esses desafios de forma lógica e colaborativa.

* **Mentalidade de solução:** Em vez de focar apenas no problema, um bom desenvolvedor busca ativamente soluções e alternativas.
* **Iniciativa:** Ser proativo na identificação de problemas antes que eles se tornem maiores e na busca por melhorias no código e nos processos.

### **4. Gestão de Tempo e Organização**

O gerenciamento do tempo impacta diretamente a capacidade de cumprir prazos e manter o

ritmo do projeto.

* **Priorização:** Saber o que é mais importante é focar nessas tarefas primeiro, especialmente quando há múltiplas responsabilidades.
* **Proatividade:** Não esperar instruções o tempo todo. Um desenvolvedor proativo antecipa necessidades e avança nas tarefas sem precisar de microgerenciamento.

### **5. Empatia e Inteligência Emocional**

Essas habilidades ajudam a construir um ambiente de trabalho positivo e a resolver

conflitos.

* **Empatia:** Tentar entender a perspectiva e os desafios dos outros membros da equipe.
* **Habilidade de negociação:** Em situações de conflito, ser capaz de negociar e chegar a um consenso que beneficie o projeto.

### **6. Curiosidade e Vontade de Aprender**

A tecnologia frontend evolui rapidamente. Um bom desenvolvedor de frontend tem que estar

sempre disposto a aprender.

* **Sede de conhecimento:** O desejo genuíno de aprender sobre novas tecnologias, frameworks e melhores práticas.
* **Adaptabilidade:** A capacidade de se ajustar a novas ferramentas e fluxos de trabalho com agilidade, sem resistência à mudança.