FASE 3 — ARQUITETURA FRONT-END: O Manual Definitivo de Engenharia Real

Resumo Executivo: A Transição para a Engenharia Real

A história do desenvolvimento front-end pode ser categorizada em três fases evolutivas distintas, cada uma respondendo à crescente complexidade das aplicações web e às demandas dos usuários por experiências mais ricas e interativas. A Fase 1, a era do scripting, foi caracterizada pela manipulação direta do Document Object Model (DOM) e pelo uso de bibliotecas utilitárias como jQuery, focadas em adicionar interatividade pontual a páginas estáticas. A Fase 2, a era dos Frameworks, introduziu estruturas robustas como Angular, React e Vue, que trouxeram o conceito de componentes, data binding e Single Page Applications (SPAs). No entanto, esta fase frequentemente resultou em um acoplamento excessivo entre a lógica de negócios e as especificidades das bibliotecas de UI, criando sistemas difíceis de testar e manter à medida que escalavam.

Atualmente, estamos ingressando na **Fase 3: Engenharia Real**. Esta fase transcende a escolha do framework. Ela aplica princípios de engenharia de software testados pelo tempo — como Clean Architecture, Domain-Driven Design (DDD) e Arquitetura Hexagonal — ao ambiente do navegador. Nesta nova era, o front-end não é tratado meramente como uma camada de apresentação ("telas"), mas como um sistema distribuído complexo, guiado por estado, que exige rigor arquitetural. Este relatório serve como um manual definitivo para navegar nesta transição, desconstruindo conceitos de alto nível em conhecimento técnico aplicável, projetado para transformar desenvolvedores em engenheiros de software completos.

Parte I: A Filosofia da Clean Architecture no Front-End

1.1 Além do Framework: A Inversão de Controle Arquitetural

No desenvolvimento front-end tradicional da Fase 2, o framework dita a estrutura da aplicação. Desenvolvedores constroem "apps Angular" ou "apps React", onde a lógica de negócios é pulverizada entre componentes visuais e serviços que misturam chamadas HTTP com regras de validação. A **Clean Architecture**, proposta por Robert C. Martin, inverte esse paradigma. Ela postula que o núcleo do software deve ser a **lógica de negócios** — as regras que definem o propósito da aplicação — e que o framework, o banco de dados e a interface do usuário são meros detalhes, mecanismos de entrega dessa lógica.[1]

A premissa central é que o software deve ser independente de frameworks. A arquitetura não deve dizer "isso é uma aplicação Angular"; ela deve gritar "isso é um sistema de gestão de

saúde" ou "isso é uma plataforma de e-commerce". O framework é uma ferramenta, não a própria aplicação. Ao adotar essa mentalidade, garantimos que a lógica vital da empresa sobreviva às mudanças tecnológicas inevitáveis, como a depreciação de bibliotecas ou atualizações disruptivas de versões.[2]

1.1.1 A Regra de Dependência

A regra mais crítica e inviolável na Clean Architecture é a **Regra de Dependência**. Visualizada como círculos concêntricos, esta regra dita que as dependências do código-fonte podem apontar apenas para **dentro**. Nada em um círculo interno pode saber qualquer coisa sobre algo em um círculo externo.

- **O Círculo Interno (Entidades e Domínio):** Este é o coração do sistema. Contém as regras de negócios corporativas, aquelas que existiriam mesmo se o sistema fosse executado manualmente com papel e caneta. As Entidades não sabem que estão rodando em um navegador, não sabem sobre requisições HTTP e não sabem sobre o Angular. Elas são TypeScript puro.
- **O Círculo Intermediário (Casos de Uso/Aplicação):** Contém as regras de negócios específicas da aplicação. Ele orquestra o fluxo de dados para e das entidades e direciona essas entidades para usar suas regras corporativas críticas para atingir os objetivos do caso de uso.
- **O Círculo Externo (Adaptadores de Interface e Frameworks):** Contém os detalhes de implementação. Aqui vivem os Componentes Angular, os serviços que fazem chamadas HttpClient, o acesso ao LocalStorage e as bibliotecas de UI.

Em uma arquitetura de "Fase 3", a Interface do Usuário (Componente Angular) depende do Caso de Uso, mas o Caso de Uso *nunca* depende da UI. Esse desacoplamento radical permite que a lógica de negócios seja testada em isolamento absoluto, reutilizada em diferentes plataformas (web, mobile, desktop) ou migrada para um novo framework sem a necessidade de reescrever as regras centrais.[3]

1.2 Os Três Pilares de um Front-End Limpo

Para implementar esses conceitos abstratos de forma tangível em uma aplicação moderna, estruturamos o sistema em três camadas primárias, cada uma com responsabilidades estritas e fronteiras claras.

1.2.1 A Camada de Apresentação (A "Pele")

Esta camada é responsável por como as coisas parecem e como o usuário interage com o sistema. No ecossistema Angular, isso consiste primariamente em **Componentes** e **Pipes**.

- **Responsabilidade:** Renderizar o HTML, capturar eventos do usuário (cliques, inputs, scroll) e exibir dados que foram processados pelas camadas inferiores.
- **Regra de Ouro:** Esta camada deve conter **zero lógica de negócios**. Ela deve ser "burra" e declarativa. Se a cor de um botão precisa mudar baseada no score de crédito de um

usuário, a camada de apresentação não calcula o score. Ela apenas pergunta à camada de lógica "Qual é o status?" e a lógica responde "Alto Risco". A camada de apresentação, então, traduz "Alto Risco" para a cor vermelha. Essa separação é crucial para evitar o anti-padrão de "Smart Components" que sabem demais, tornando-se difíceis de manter e testar.[1]

1.2.2 A Camada de Domínio (O "Cérebro")

Esta é a camada mais importante e frequentemente a mais negligenciada na Fase 2. Ela representa a realidade do negócio, modelada em software.

- **Entidades:** Objetos que possuem uma identidade única (como um Usuário com um ID específico) e um ciclo de vida. Diferente de simples estruturas de dados (DTOs), as Entidades na Fase 3 são ricas em comportamento. Elas contêm métodos que garantem a integridade dos seus dados (e.g., usuario.alterarSenha(...) que valida as regras de complexidade internamente), em vez de serem meros sacos de dados manipulados por serviços externos.[3]
- **Casos de Uso (Interactors):** Representam ações específicas que um usuário pode realizar no sistema (e.g., RegistrarUsuario, RealizarPedido, CancelarAssinatura). Frequentemente implementados como classes com um único método público (como execute()), eles encapsulam a orquestração necessária para cumprir uma tarefa de negócio, garantindo que as regras do domínio sejam respeitadas.[3]

1.2.3 A Camada de Dados (Os "Membros")

Esta camada lida com o "trabalho sujo" de recuperar e persistir dados, comunicando-se com o mundo exterior.

- **Repositórios:** Aqui ocorre a mágica da **inversão de dependência**. As interfaces dos repositórios são definidas na Camada de Domínio (o Domínio diz: "Eu preciso de uma maneira de buscar usuários"), mas a implementação concreta reside na Camada de Dados (a Infraestrutura diz: "Aqui está uma classe que busca usuários via API REST"). Isso garante que o Domínio não dependa da Camada de Dados, mas sim o contrário.[1]
- **Fontes de Dados (Data Sources):** Os clientes HTTP reais (como o HttpClient do Angular) ou wrappers para LocalStorage, IndexedDB ou Firebase. Eles lidam com a serialização, cabeçalhos de autenticação e tratamento de erros de rede.

1.3 Analogia: A Cozinha de um Restaurante Profissional

Para consolidar o entendimento dessas camadas para um público leigo ou em transição, podemos utilizar a analogia de um restaurante de alta gastronomia.

- **A Camada de Apresentação** é a **Equipe de Salão (Garçons)**. Eles interagem com o cliente, anotam os pedidos (inputs) e trazem os pratos prontos (outputs). Eles não cozinham. Se o menu muda, eles apenas entregam o novo cardápio; eles não precisam reaprender a cozinhar. Sua função é puramente de interface e experiência do cliente.[5]

- **A Camada de Domínio** é o **Chef e o Livro de Receitas**. O Chef sabe *como* preparar um hambúrguer gourmet (Lógica de Negócios). A Receita (Entidade) define o que *é* esse hambúrguer (ingredientes obrigatórios, ponto da carne). O Chef não se importa com quem é o garçom que trouxe o pedido, nem em qual mesa o cliente está sentado. Ele opera regras puras de culinária.[3]
- **A Camada de Dados** é o **Fornecedor e o Estoque**. O Chef precisa de tomates (Dados). Ele preenche um formulário de requisição (Interface do Repositório). O Fornecedor (Implementação da Camada de Dados) pega o caminhão, vai até a fazenda (API), busca os tomates e os entrega na cozinha. O Chef não se importa se o caminhão é Ford ou Volvo, ou qual estrada o fornecedor pegou, desde que os tomates cheguem frescos e conforme especificado na requisição.[7]

Parte II: Arquitetura Hexagonal (Portas e Adaptadores) no Angular

2.1 Desconstruindo o Hexágono

A Arquitetura Hexagonal, também conhecida como **Portas e Adaptadores** (Ports and Adapters), é uma variação específica da Clean Architecture que enfatiza a **simetria** das interfaces e o isolamento do núcleo da aplicação. Ela visualiza a aplicação não como camadas empilhadas (como na arquitetura n-camadas tradicional), mas como um núcleo central cercado por uma fronteira permeável apenas através de pontos de conexão específicos. A forma "hexagonal" é arbitrária; ela serve apenas para representar graficamente que a aplicação pode ter múltiplos lados ou facetas de interação com o mundo externo.[2]

O objetivo primordial é permitir que uma aplicação seja guiada por usuários, programas, testes automatizados ou scripts em lote, e que seja desenvolvida e testada isoladamente de seus eventuais dispositivos de execução e bancos de dados. Na prática do Angular, isso significa que podemos desenvolver e testar toda a lógica complexa da aplicação sem sequer abrir o navegador ou ter um backend funcional.[2]

2.1.1 O Núcleo (Inside)

Este é o santuário da lógica. Contém o **Modelo de Domínio** e os **Serviços de Aplicação**. É escrito em TypeScript puro e agnóstico. Ele não importa @angular/core, não usa HttpClient e não conhece bibliotecas de UI de terceiros. A pureza deste núcleo garante que ele seja perene e imune às "guerras de frameworks" ou mudanças de bibliotecas de infraestrutura.[2]

2.1.2 As Portas (As Interfaces)

As Portas são os "plugues" na fronteira do hexágono. Elas definem uma conversa ou contrato entre o núcleo e o mundo exterior.

- **Portas Primárias (Driving/Driver Ports):** São interfaces que permitem que o mundo

exterior "dirija" a aplicação. Elas expõem as funcionalidades que o núcleo oferece. Exemplo: Uma interface ITodoService ou TodoUseCase que a UI chama para adicionar um item à lista.

- **Portas Secundárias (Driven Ports):** São interfaces que a aplicação usa para falar com o mundo exterior. É o núcleo dizendo "Eu preciso que alguém salve isso para mim". Exemplo: Uma interface UserRepository que define métodos como save(user: User) ou findById(id: string).[2]

2.1.3 Os Adaptadores (As Implementações)

Os Adaptadores são as peças que conectam as Portas à realidade tecnológica.

- **Adaptadores Primários (Driving Adapters):** Um Componente Angular é um adaptador primário. Ele captura um evento do DOM (clique), converte-o em uma chamada de método e invoca a Porta Primária (Caso de Uso). Um Teste Unitário (Jest/Jasmine) também é um adaptador primário; ele invoca a mesma porta para verificar se a lógica funciona, sem renderizar pixels.
- **Adaptadores Secundários (Driven Adapters):** Um Serviço Angular que encapsula o HttpClient é um adaptador secundário. Ele implementa a interface da Porta Secundária (Repositório) e realiza a ação concreta de enviar um JSON via HTTP POST. Outro adaptador poderia implementar a mesma porta usando LocalStorage para permitir que a aplicação funcione offline.[2]

2.2 Implementando o Hexágono no Angular

O Angular é, talvez, o framework mais naturalmente adequado para a Arquitetura Hexagonal devido ao seu robusto sistema de **Injeção de Dependência (DI)**. O sistema de DI do Angular atua como o mecanismo que liga as Portas aos Adaptadores em tempo de execução, sem que o código fonte do domínio precise conhecer a implementação concreta.

2.2.1 Definindo a Porta (Classe Abstrata)

No TypeScript, interfaces são artefatos de tempo de compilação e desaparecem no JavaScript final. Isso as torna difíceis de usar como tokens para Injeção de Dependência. Portanto, na Engenharia Real da Fase 3, frequentemente utilizamos **Classes Abstratas** para definir nossas Portas. Elas servem tanto como contrato de tipagem quanto como token de injeção.[2]

TypeScript

```
// dominio/portas/user-repository.port.ts
export abstract class UserRepositoryPort {
```

```
  abstract getUser(id: string): Observable<User>;
  abstract saveUser(user: User): Observable<void>;
}
```

Esta classe abstrata reside na camada de Domínio. Ela não possui implementação lógica, apenas assinaturas.

2.2.2 Implementando o Adaptador (Infraestrutura)

A implementação concreta vive na camada de Infraestrutura, longe do domínio sagrado.

TypeScript

```
// infraestrutura/adaptadores/http-user-repository.adapter.ts
@Injectable()
export class HttpUserRepositoryAdapter implements UserRepositoryPort {
  constructor(private http: HttpClient) {}

  getUser(id: string): Observable<User> {
    // O adaptador também é responsável por Mapear o DTO da API para a Entidade de Domínio
    return this.http.get<UserDto>(`/api/users/${id}`).pipe(
      map(dto => UserMapper.toDomain(dto))
    );
  }
  //...
}
```

Note o uso de um Mapper. O adaptador atua como um tradutor, convertendo o dialeto da API (DTOs) para a linguagem pura do Domínio (Entidades). Isso impede que mudanças na API (como renomear um campo JSON) quebrem a lógica de negócios.[4]

2.2.3 A Conexão (Wiring)

No arquivo de configuração da aplicação (app.config.ts ou um Módulo), nós ligamos a Porta ao Adaptador.

TypeScript

```
// app.config.ts
providers:
```

Agora, quando um componente ou caso de uso solicita UserRepositoryPort, o Angular injeta HttpUserRepositoryAdapter. Se decidirmos mudar o backend para Firebase amanhã, criamos um FirebaseUserRepositoryAdapter, alteramos essa única linha de configuração, e todo o resto da aplicação continua funcionando perfeitamente sem alteração de código.[2]

2.3 Analogia: O Adaptador Universal de Viagem

Para ilustrar a potência desse padrão, imagine seu laptop (O Núcleo/Domínio). Ele precisa de energia para operar e possui um plugue com um formato específico (A Porta).

- **A Rede Elétrica** (O Sistema Externo/API) fornece eletricidade, mas as tomadas variam de país para país (Reino Unido, EUA, Brasil).
- **O Adaptador de Viagem** (O Adaptador) é a ponte. Ele se encaixa na parede (Sistema Externo) e fornece o formato de soquete que seu laptop espera (A Porta).
- Seu laptop funciona exatamente da mesma maneira em Londres, Nova York ou Tóquio. Ele não precisa ser refeito ou recabeado para tensões diferentes; o adaptador lida com a tradução da corrente e do formato físico.
- No software, se sua API muda de REST para GraphQL, você não reescreve sua aplicação. Você apenas troca o "Adaptador de Viagem" (cria um GraphQL Adapter) e o núcleo da aplicação continua operando inalterado.[10]

Parte III: Domain-Driven Design (DDD) no Front-End

3.1 Por que DDD no Navegador?

Historicamente, o Domain-Driven Design (DDD) era considerado território exclusivo do backend. O front-end era visto apenas como uma "casca" burra. No entanto, as Single Page Applications (SPAs) modernas são clientes ricos e complexos. Elas realizam validações intrincadas, cálculos de preços em tempo real, gestão de permissões e orquestração de fluxos de trabalho. Se tratarmos o front-end apenas como "telas", acabamos duplicando lógica de negócios de forma desorganizada, criando o anti-padrão de "Modelo Anêmico" e espalhando regras vitais em componentes de UI difíceis de testar.[6]

A Fase 3 reconhece que o navegador é um ambiente de execução legítimo para regras de negócio, exigindo a mesma disciplina de modelagem que o servidor.

3.2 Design Estratégico: Contextos Delimitados (Bounded Contexts)

O Design Estratégico lida com a arquitetura de alto nível e a organização de times e código. O conceito central é o de **Contextos Delimitados** (Bounded Contexts).

- **Definição:** Um limite explícito dentro do qual um modelo de domínio específico se aplica

e faz sentido. Grandes sistemas corporativos são complexos demais para um único modelo unificado.

- **Exemplo:** Em uma aplicação de E-commerce, a palavra "Produto" tem significados diferentes dependendo do contexto.
  - No **Contexto de Catálogo**, "Produto" é definido por título, descrição, imagens e avaliações.
  - No **Contexto de Inventário**, "Produto" é definido por SKU, localização no corredor do armazém, quantidade física e dimensões de empacotamento.
  - No **Contexto de Faturamento**, "Produto" é apenas um item de linha com valor, impostos e códigos fiscais.[13]
- **Aplicação no Front-End:** Em vez de uma pasta gigante src/app com subpastas técnicas (components, services), "fatiamos" a aplicação verticalmente por domínios funcionais.
  - projects/catalogo
  - projects/checkout
  - projects/minha-conta
    Cada um desses diretórios pode ser uma Biblioteca Angular distinta ou até um Micro-frontend, com suas próprias regras, modelos e componentes isolados.15

3.2.1 Linguagem Ubíqua (Ubiquitous Language)

O DDD enfatiza o uso rigoroso da mesma linguagem no código que os especialistas de negócio usam nas reuniões.

- Se os especialistas dizem "Cliente" e o código diz User, existe uma dissonância cognitiva que gera bugs.
- Se o negócio diz "Despachar Pedido", o método no código deve ser despacharPedido(), e não atualizarStatus(5).
- A linguagem ubíqua deve permear nomes de variáveis, classes, arquivos e até nomes de rotas na aplicação, reduzindo o custo de tradução mental e erros de interpretação.[15]

3.3 Design Tático: Os Blocos de Construção

O Design Tático fornece as ferramentas e padrões de código para construir o modelo dentro de um Contexto Delimitado.

3.3.1 Entidades vs. Objetos de Valor (Value Objects)

A distinção entre Entidade e Objeto de Valor é fundamental para modelar corretamente o comportamento do sistema.

- **Entidades:** São objetos definidos pela sua **identidade** contínua. Um Usuário é uma entidade porque, mesmo se ele mudar de nome, endereço ou e-mail, ele continua sendo o mesmo usuário (identificado por um ID 123). Entidades têm ciclo de vida e histórico.
- **Objetos de Valor:** São objetos definidos pelos seus **atributos**. Uma Cor (Vermelho) é um objeto de valor. Se você mudar a propriedade RGB de um objeto Cor, ele deixa de ser Vermelho e vira outra cor. Você não "edita" um objeto de valor; você o substitui. Eles são

imutáveis. No TypeScript, modelamos isso como Classes com propriedades readonly. Exemplos comuns incluem Dinheiro, CoordenadaGPS, IntervaloDeData.[12]

3.3.2 Modelos Ricos vs. Modelos Anêmicos

Este é o ponto de falha mais comum no desenvolvimento da Fase 2.

- **Modelo Anêmico (Anti-padrão):** Classes que contêm apenas dados (propriedades públicas) e nenhum comportamento. A lógica reside inteiramente em classes de "Serviço" externas que manipulam esses dados. Isso viola o encapsulamento da Orientação a Objetos.

```
TypeScript
// Anêmico: O objeto não se protege
class User {
  id: string;
  isActive: boolean;
}
class UserService {
  activate(user: User) {
    // A lógica de validação está fora do objeto
    user.isActive = true;
  }
}
```

- **Modelo Rico (Engenharia Real):** A classe contém os dados *e* a lógica para manipular esses dados. Isso impõe regras de validade e consistência.

```
TypeScript
// Rico: O objeto garante sua própria integridade
class User {
  private _isActive: boolean;

  constructor(private id: string, isActive: boolean, private isBanned: boolean) {
    this._isActive = isActive;
  }

  activate() {
    if (this.isBanned) {
        throw new Error("Não é possível ativar um usuário banido.");
    }
    this._isActive = true;
  }
}
```

Isso garante que um User nunca possa estar em um estado inválido dentro do sistema,

aumentando drasticamente a robustez.[6]

3.4 Mapeamento de Contexto (Context Mapping)

Como os diferentes Contextos Delimitados conversam entre si no front-end?

- **Shared Kernel (Núcleo Compartilhado):** Uma biblioteca compartilhada (e.g., pacote npm interno) contendo tipos universais como Money, EmailAddress ou DateRange. Deve ser usado com parcimônia extrema para evitar acoplamento rígido entre times distintos.[15]
- **Anti-Corruption Layer (ACL):** Quando o contexto de "Checkout" precisa de dados do "Sistema Legado de Inventário", ele não deve usar os objetos bagunçados do sistema legado diretamente. Ele constrói uma camada (ACL) que traduz os dados "sujos" do legado para objetos limpos e semanticamente corretos do contexto de "Checkout" antes de usá-los. Isso protege o novo código de ser contaminado pela dívida técnica de sistemas antigos.[15]

Parte IV: State-Driven UI — O Paradigma Moderno (Signals + RxJS)

4.1 A Mudança de Paradigma: Imperativo vs. Declarativo

A transição para a Fase 3 é marcada por uma mudança fundamental na forma como pensamos a interação da UI.

- **Imperativo (Event-Driven):** Focado no "Como". "Quando o botão for clicado, chame o serviço, espere a resposta, pegue os dados, atribua à variável lista, depois mude a variável isLoading para false." O desenvolvedor microgerencia cada passo e transição de estado.[17]
- **Declarativo (State-Driven):** Focado no "O Que". "A Lista *é* uma transformação dos Dados do servidor. O Indicador de Carregamento *é* visível sempre que o status da Requisição for 'Pendente'." Na Engenharia Real, nós definimos as relações e o estado; a UI é simplesmente um reflexo (uma projeção) desse estado atual. Nós não atualizamos a UI manualmente; nós atualizamos o estado, e a UI reage matematicamente.[17]

4.2 Angular Signals: A Nova Primitiva de Reatividade

Os **Angular Signals** representam a maior mudança arquitetural na história do Angular. Eles resolvem o problema fundamental do Zone.js. O Zone.js opera "monkey-patching" em eventos do navegador para detectar mudanças, o que tem custos de performance e às vezes dispara detecções de mudança desnecessárias em toda a árvore de componentes. Os Signals introduzem a **reatividade granular**, permitindo que o Angular saiba *exatamente* qual parte do modelo mudou e atualize apenas o nó de texto ou atributo correspondente no DOM, sem verificar a árvore inteira.[20]

4.2.1 Primitivas dos Signals

- **Writable Signal:** Um wrapper de valor que você pode definir ou atualizar. É a fonte da verdade. const count = signal(0);
- **Computed Signal:** Um valor derivado de outros signals. const double = computed(() => count() * 2);. Ele é **preguiçoso (lazy)** (só calcula quando lido) e **memoizado** (cacheado). Se count não mudar, double não recalcula, economizando processamento. Ele cria um grafo de dependência dinâmico e automático.[20]
- **Effect:** Um efeito colateral que roda quando um signal muda. effect(() => console.log(count()));. Usado para logging, sincronização manual com o DOM ou integração com bibliotecas não-reativas. Deve ser usado com cautela.[23]

4.2.2 A Analogia da Planilha Eletrônica

Os Signals funcionam exatamente como o **Excel ou Google Sheets**.

- **Célula A1** contém o valor 10 (Writable Signal).
- **Célula B1** contém a fórmula =A1*2 (Computed Signal).
- Se você atualiza A1, B1 atualiza automaticamente. Você não precisa escrever um script para "avisar B1 para atualizar". A dependência é inerente à definição.
- Se a Célula C1 depende de B1, ela também atualiza em cascata. Esse "grafo de dependência" garante que apenas as células (componentes) necessárias sejam recalculadas, levando a ganhos massivos de performance e previsibilidade.[21]

4.3 Signals + RxJS: A Arquitetura de Interoperabilidade

Um equívoco comum na Fase 3 é achar que Signals substituem o RxJS. Na "Engenharia Real", eles são tecnologias complementares que resolvem problemas diferentes.

- **RxJS** é ideal para **Fluxos de Eventos no Tempo** (Event Streams). Ele brilha em lidar com assincronismo complexo, cancelamento de requisições, debouncing, throttling e WebSockets. É baseado no modelo **Push** (Empurrar).
- **Signals** são ideais para **Gerenciamento de Estado Síncrono**. Eles brilham em renderizar a UI, derivar dados e garantir consistência visual. São baseados no modelo **Pull** (Puxar) com notificação de mudança.[24]

4.3.1 O Padrão Ouro: Stream to Signal

O padrão arquitetural mais poderoso no Angular moderno é usar RxJS para o "processamento complexo de eventos" (a camada de infraestrutura/transporte) e converter o resultado final em um Signal para consumo fácil e performático na UI (a camada de apresentação).

TypeScript

```
// Camada de Orquestração (RxJS)
// Lidamos com a complexidade do tempo: debounce, cancelamento de requisições anteriores
private searchResults$ = this.searchTerm$.pipe(
  debounceTime(300),
  distinctUntilChanged(),
  switchMap(term => this.api.search(term)),
  catchError(err => of())
);

// Camada de Estado (Signal)
// toSignal converte o fluxo Observable em um valor Signal síncrono para o template
// O template não precisa do pipe | async e não precisa lidar com null/undefined manualmente
public results = toSignal(this.searchResults$, { initialValue: });
```

Isso nos dá o poder dos operadores RxJS (Debounce, SwitchMap, Retry) com a simplicidade e performance dos Signals no template, eliminando o pipe | async e o gerenciamento manual de subscrições.[23]

4.4 Gerenciando Estado Complexo (Service-Based Store)

Para muitas aplicações, bibliotecas completas de gerenciamento de estado como NgRx (Redux) podem introduzir complexidade desnecessária (boilerplate). A Fase 3 popularizou o padrão **Service-with-Signals** (O padrão "Service Store").

1. **Private Writable Signals:** Mantêm os dados brutos dentro do serviço, inacessíveis externamente.
2. **Public Read-Only Signals:** Expõem os dados para os componentes usando .asReadonly() ou computed(). Isso impõe um fluxo de dados unidirecional — os componentes não podem bagunçar o estado diretamente; eles devem chamar métodos (ações) no serviço.[27]

Parte V: Guia de Implementação e Refatoração (De Anti-Padrões a Padrões)

5.1 Anti-Padrão 1: O "God Component"

- **Sintoma:** Um arquivo de componente com mais de 500 linhas de código. Ele faz chamadas HTTP diretas, contém lógica de validação de formulário, manipula arrays de dados e controla a visibilidade de modais. Ele sabe tudo e faz tudo.[29]
- **A Correção (Refatoração):** Aplicar a separação **Smart vs. Dumb** (Container vs. Presentational).

    - **Smart Component (Container):** O orquestrador. Ele injeta serviços, obtém Observables/Signals e passa dados para baixo via [inputs]. Ele ouve eventos dos filhos via (outputs). Ele não tem estilos complexos nem muito HTML.
    - **Dumb Component (Presentational):** Puramente visual. Ele não tem dependências (não injeta serviços). Ele só conhece seus Inputs e Outputs. Isso o torna altamente reutilizável e fácil de testar visualmente (Storybook).[5]

5.2 Anti-Padrão 2: A Sopa de Serviços (Service Soup)

- **Sintoma:** Um UserService que tem 50 métodos e 2000 linhas. Ele autentica, salva configurações, formata datas, calcula idade do usuário e busca histórico de compras. Tornou-se um depósito de lixo para qualquer coisa relacionada a "usuário".[31]
- **A Correção:** Aplicar DDD e Segregação de Responsabilidade.
    - Extrair **Casos de Uso** (LoginUserUseCase, UpdateProfileUseCase) como classes separadas e focadas.
    - Extrair **Serviços de Domínio** para lógica pura (AgeCalculator).
    - O UserService deve idealmente desaparecer ou tornar-se apenas uma Fachada (Facade) para orquestrar esses atores menores.[1]

5.3 Anti-Padrão 3: O Tipo "Any" e Interfaces Anêmicas

- **Sintoma:** Uso de any ou interfaces que espelham exatamente o JSON do backend (e.g., user_addr_street), poluindo o código frontend com a nomenclatura do banco de dados legado.
- **A Correção:** Uso estrito de **Mappers**. Nunca permita que um DTO (Data Transfer Object) do backend vaze para a camada de aplicação ou apresentação.
    - Crie uma interface User limpa na camada de Domínio (addressStreet).
    - Crie um UserMapper na camada de Infraestrutura que converte UserDto (da API) para User (Domínio).
    - Se a API mudar user_addr_street para u_str, você altera apenas uma linha no Mapper, e não em 50 arquivos espalhados pela aplicação.[4]

5.4 Fatiamento Vertical e Bibliotecas de Funcionalidade

Em vez de organizar o projeto por camadas técnicas horizontais:

- src/app/components
- src/app/services
- src/app/models

A Engenharia Real organiza por **Funcionalidade (Bounded Context)** vertical:

- src/app/features/checkout/components
- src/app/features/checkout/services
- src/app/features/checkout/models

Isso garante que o código relacionado ao "Checkout" esteja co-localizado (princípio da coesão). Se você precisar deletar a funcionalidade de Checkout, você deleta a pasta e tudo vai embora. Na abordagem por camadas, você teria que caçar arquivos em 5 pastas diferentes, deixando para trás "código morto".[15]

5.5 Tabelas de Comparação Arquitetural

Comparativo de Modelos de Estado

Característica	Observable (RxJS)	Signal (Angular)	Uso Recomendado (Fase 3)
Paradigma	Push (Evento)	Pull (Valor)	Híbrido: RxJS para Eventos, Signals para Estado.
Natureza	Assíncrono (Fluxo no tempo)	Síncrono (Valor atual)	Signals para o Template (Sync), RxJS para API (Async).
Dependências	Manual (subscribe / async)	Automática (Grafo de Dependência)	Signals eliminam vazamento de memória em Views.
Operadores	Poderosos (switchMap, debounce)	Simples (computed, effect)	RxJS para orquestração complexa, Signals para derivação simples.
Detecção de Mudança	Zone.js (Verifica tudo)	Granular (Verifica só o que mudou)	Signals permitem aplicações "Zone-less" de alta performance.

Comparativo de Arquitetura de Camadas

Camada	Conceitos Chave	Tecnologias	Dependências

		Típicas	Permitidas
Apresentação	Views, Event Handling, Formatação	Componentes, HTML, CSS, Pipes	Depende do Domínio e Aplicação.
Aplicação	Casos de Uso, Orquestração	Services (Facades), State Management	Depende do Domínio.
Domínio	Entidades, Value Objects, Regras, Portas	Classes TypeScript Puras, Interfaces	**Depende de NADA** (O centro do universo).
Infraestrutura	Chamadas API, Cache, Mappers, DTOs	HttpClient, LocalStorage, Firebase	Depende do Domínio (Implementa as Portas).

Conclusão: O Novo Padrão de Engenharia

A Arquitetura Front-End da Fase 3 não se trata apenas de aprender uma nova sintaxe ou atualizar a versão do Angular; trata-se de adotar uma nova mentalidade profissional. É o reconhecimento de que o navegador é um alvo de implantação para sistemas distribuídos sofisticados, e não apenas um visualizador de documentos.

Ao adotar a Clean Architecture, protegemos nossa lógica preciosa da volatilidade dos frameworks de UI, garantindo longevidade.
Ao adotar a Arquitetura Hexagonal, desacoplamos nossa aplicação das APIs de backend, tornando-nos resilientes a mudanças de infraestrutura e permitindo testes robustos.
Ao adotar o Domain-Driven Design, garantimos que nosso software modele fielmente os problemas de negócio do mundo real, criando uma "Linguagem Ubíqua" compartilhada com os stakeholders e eliminando erros de tradução.
Ao adotar a State-Driven UI com Signals, alavancamos as capacidades modernas do navegador para uma renderização preditiva de alta performance, simplificando drasticamente o código de interface.
Este manual fornece a planta baixa. A jornada da "Engenharia Real" começa quando você para de escrever código apenas para "fazer funcionar" e começa a projetar sistemas para "durar, escalar e evoluir". O futuro do front-end pertence aos engenheiros que dominam não apenas as ferramentas, mas os princípios que as governam.

Referências citadas

1. Clean Architecture: Simplified and In-Depth Guide | by DrunknCode | Medium, acessado em dezembro 27, 2025, https://medium.com/@DrunknCode/clean-architecture-simplified-and-in-depth-guide-026333c54454
2. “Ports and Adapters” vs “Hexagonal Architecture” - is it the same pattern? - Angular.love, acessado em dezembro 27, 2025, https://angular.love/ports-and-adapters-vs-hexagonal-architecture-is-it-the-same-pattern/
3. Domain layer | App architecture - Android Developers, acessado em dezembro 27, 2025, https://developer.android.com/topic/architecture/domain-layer
4. Where to map to domain from Data layer in MVVM Clean Architecture? - Stack Overflow, acessado em dezembro 27, 2025, https://stackoverflow.com/questions/79417502/where-to-map-to-domain-from-data-layer-in-mvvm-clean-architecture
5. Making Dumb Components Smart: Refactoring - Simple Programmer, acessado em dezembro 27, 2025, https://simpleprogrammer.com/dumb-components-smart-refactoring/
6. Anaemic Domain Model vs. Rich Domain Model | Ensono, acessado em dezembro 27, 2025, https://www.ensono.com/insights-and-news/expert-opinions/anaemic-domain-model-vs-rich-domain-model/
7. Difference B/W Inversion of Control(IOC) & Dependency Injection (DP) & Dependency Inversion(DI) | by Chiluka Vinayak | Medium, acessado em dezembro 27, 2025, https://medium.com/@chilukavinayak.p/difference-b-w-inversion-of-control-ioc-dependency-injection-dp-dependency-inversion-di-55deaaa4a104
8. Inversion of Control vs Dependency Injection vs Dependency Inversion - C# Corner, acessado em dezembro 27, 2025, https://www.c-sharpcorner.com/article/inversion-of-control-vs-dependency-injection-vs-dependency-inversion/
9. How to implement Hexagonal architecture in frontend (Javascript/Typescript) - GitHub, acessado em dezembro 27, 2025, https://github.com/juanm4/hexagonal-architecture-frontend
10. Hexagonal Architecture in A Nutshell - DEV Community, acessado em dezembro 27, 2025, https://dev.to/muhammadazis/hexagonal-architecture-in-a-nutshell-22bd
11. Hexagonal architecture pattern - AWS Prescriptive Guidance, acessado em dezembro 27, 2025, https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/hexagonal-architecture.html
12. Anemic Domain Model vs. Rich Domain Model | by Matthias Schenk ..., acessado em dezembro 27, 2025, https://medium.com/@inzuael/anemic-domain-model-vs-rich-domain-model-787

52b46098f
13. Bounded Context - Martin Fowler, acessado em dezembro 27, 2025, https://www.martinfowler.com/bliki/BoundedContext.html
14. Bounded Context | DevIQ, acessado em dezembro 27, 2025, https://deviq.com/domain-driven-design/bounded-context/
15. DDD in Angular & Frontend Architecture: What you need to know ..., acessado em dezembro 27, 2025, https://www.angulararchitects.io/blog/all-about-ddd-for-frontend-architectures-with-angular-co/
16. Luca Mezzalira - Micro-frontends anti-patterns - WeAreDevelopers, acessado em dezembro 27, 2025, https://www.wearedevelopers.com/videos/299/micro-frontends-anti-patterns
17. Imperative vs Declarative Programming - ui.dev, acessado em dezembro 27, 2025, https://ui.dev/imperative-vs-declarative-programming
18. A Basic Introduction to Declarative vs. imperative Programming | by Jonathan Paul-Reuven, acessado em dezembro 27, 2025, https://medium.com/@jonathan.paul-reuven/a-basic-introduction-to-declarative-vs-imperative-programming-a518ff33b196
19. Mastering Angular Signals: A Complete Guide Beyond RxJS - DEV ..., acessado em dezembro 27, 2025, https://dev.to/romain_geffrault_10d88369/mastering-angular-signals-a-complete-guide-beyond-rxjs-30mo
20. Signals • Overview • Angular, acessado em dezembro 27, 2025, https://angular.dev/guide/signals
21. How Angular Signals Transformed My State Management - And Yours Can Too, acessado em dezembro 27, 2025, https://dev.to/karol_modelski/how-angular-signals-transformed-my-state-management-and-yours-can-too-4i93
22. Angular Signals State Management - Perficient Blogs, acessado em dezembro 27, 2025, https://blogs.perficient.com/2025/06/19/angular-signals-state-management/
23. Angular Signals: A Comprehensive Guide - DEV Community, acessado em dezembro 27, 2025, https://dev.to/hassantayyab/mastering-angular-signals-a-comprehensive-guide-1dac
24. Signals vs. Observables, what's all the fuss about? - Builder.io, acessado em dezembro 27, 2025, https://www.builder.io/blog/signals-vs-observables
25. Angular Signals vs RxJS Observables: A Practical Comparison - Djamware, acessado em dezembro 27, 2025, https://www.djamware.com/post/68941a3bdc39375fe804f0b3/angular-signals-vs-rxjs-observables-a-practical-comparison
26. RxJS interop with Angular signals, acessado em dezembro 27, 2025, https://angular.dev/ecosystem/rxjs-interop
27. Best Practices for Angular State Management - DEV Community, acessado em dezembro 27, 2025,

https://dev.to/devin-rosario/best-practices-for-angular-state-management-2pm1
28. Dependent state with linkedSignal - Angular, acessado em dezembro 27, 2025, https://angular.dev/guide/signals/linked-signal
29. Steps to refactor Large components, more than 400 lines : r/Angular2 - Reddit, acessado em dezembro 27, 2025, https://www.reddit.com/r/Angular2/comments/10e8foy/steps_to_refactor_large_components_more_than_400/
30. Angular Smart Components vs Presentational Components - Angular University blog, acessado em dezembro 27, 2025, https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/
31. Introduction to services and dependency injection - Angular, acessado em dezembro 27, 2025, https://v17.angular.io/guide/architecture-services
32. Angular Architecture Guide To Building Maintainable Applications at Scale | Nx Blog, acessado em dezembro 27, 2025, https://nx.dev/blog/architecting-angular-applications