

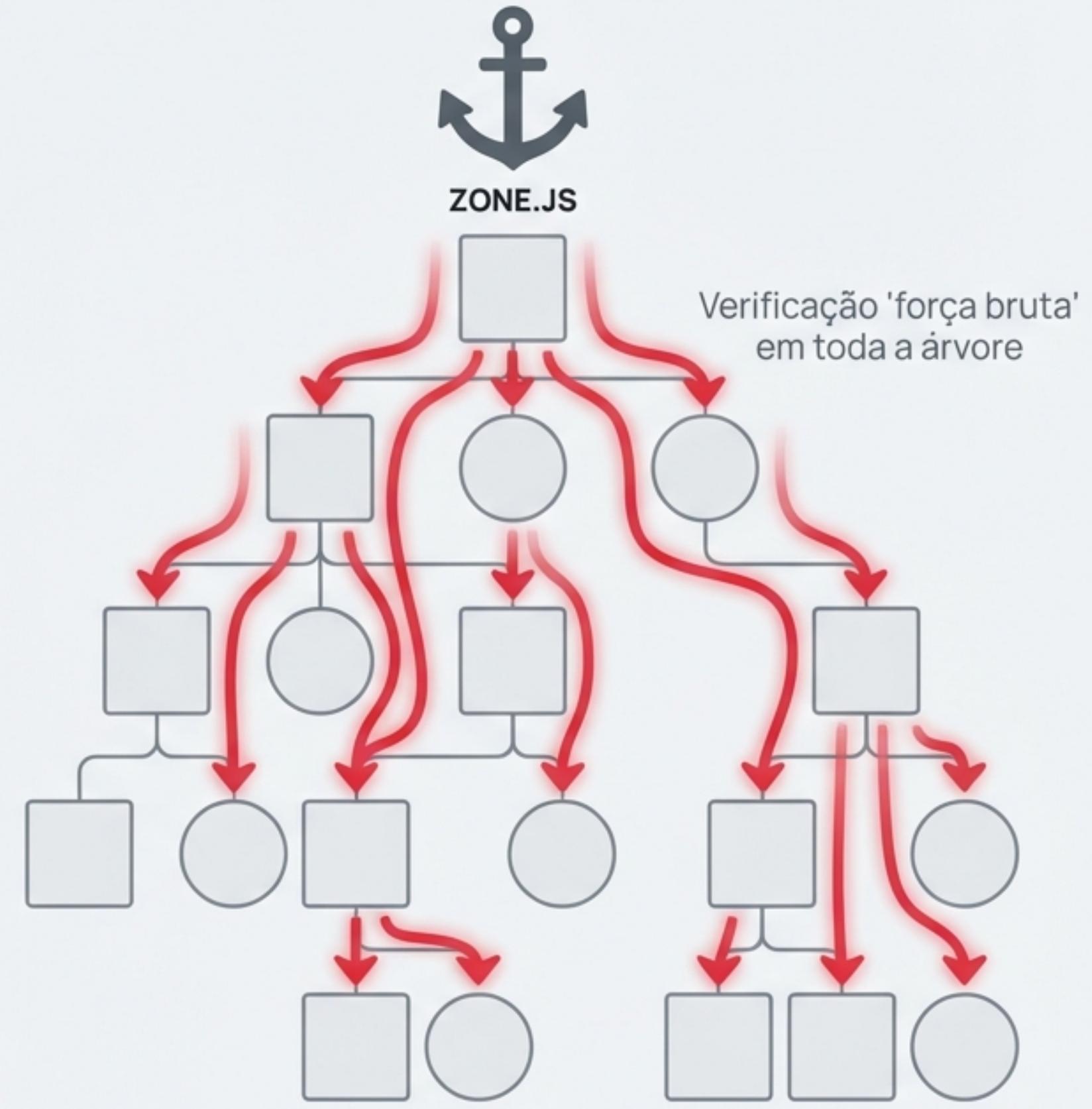
A Revolução Silenciosa do Angular

De Detecção Global a Reatividade Granular:
Um Manual Definitivo sobre Signals.

A introdução dos Signals representa a transformação arquitetural mais profunda no Angular desde a versão 2.
Não é uma nova API, é um novo motor.

O Paradigma Anterior: A Conveniência e o Custo do Zone.js

- **Mecanismo:** Zone.js intercepta eventos assíncronos (setTimeout, Promises, DOM) para disparar a detecção de mudanças de forma 'mágica'.
- **Problema Fundamental:** O 'dirty checking' global percorre toda a árvore de componentes, do topo à base, para encontrar mudanças.
- **Impacto Técnico:**
 - Complexidade computacional de **$O(n)$** , onde n é o número de componentes.
 - Performance limitada em aplicações complexas.
 - 'Stack traces' poluídos, dificultando o debugging.



Pense em Células, Não em Árvores: A Intuição por Trás dos Signals

A1	B1
10	20
C1	D1
30	50

=A1+B1

Step 1 Visualization

A1	B1
10 15	20
C1	D1
30 35	50

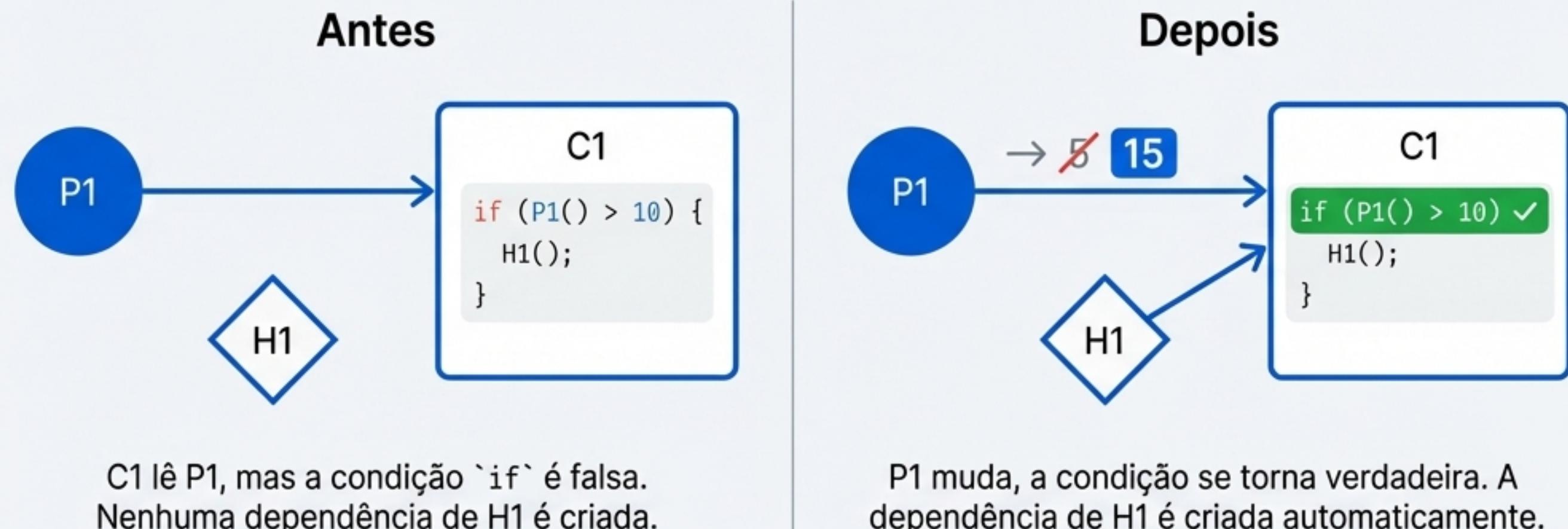
Step 2 Visualization

A mudança flui apenas através das dependências declaradas, garantindo eficiência máxima.

O Cérebro da Operação: O Grafo de Dependência Dinâmico

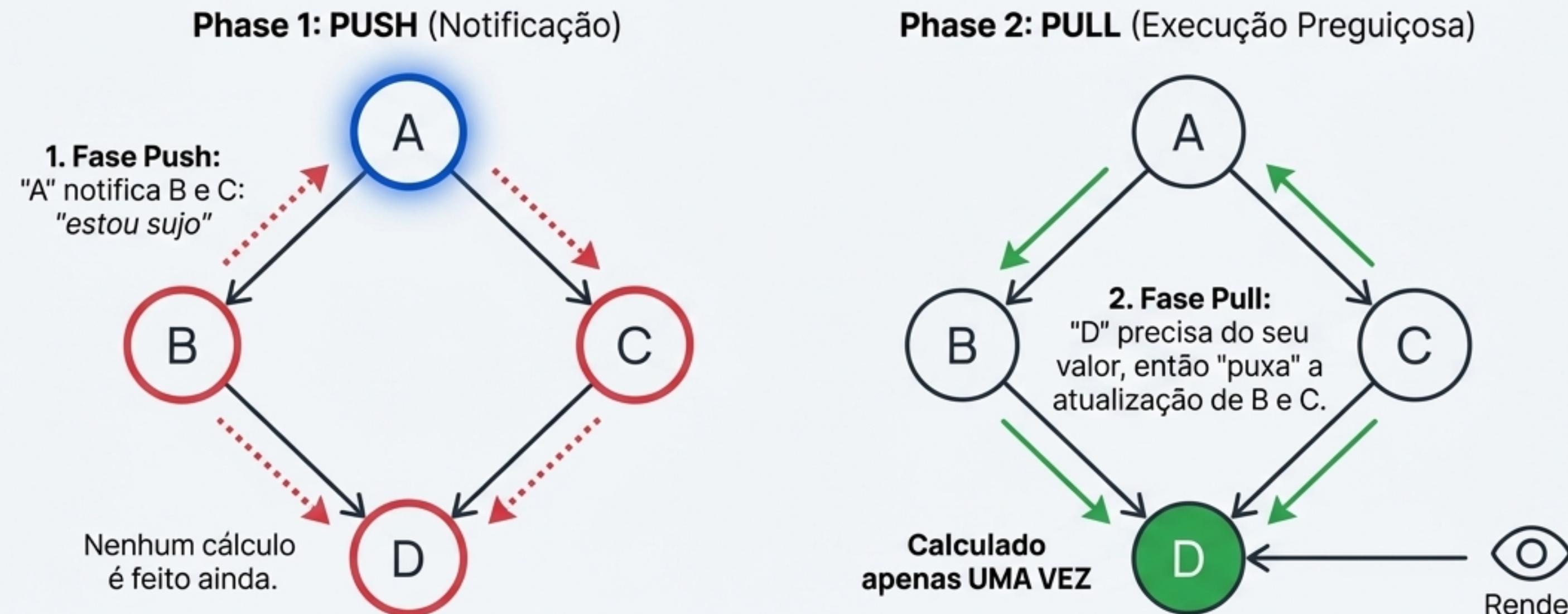
O sistema constrói e desconstrói relações em tempo de execução, sem necessidade de arrays de dependência manuais.

- **Produtores (Producers):**
Nós que detêm valores
(ex: WritableSignal)
- **Consumidores (Consumers):**
Nós que dependem de valores
(ex: template, effect)
- ◇ **Nós Híbridos:**
Atuam como ambos
(ex: computed)



Rastreamento Automático: Uma dependência é criada quando um consumidor lê um produtor. Se uma lógica condicional ('if') impede a leitura, a dependência é removida, prevenindo 'over-reactivity'.

Sem 'Glitches': Garantindo Consistência Atômica com Push/Pull



O Problema do Diamante: Se **A** alimenta **B** e **C**, e ambos alimentam **D**, uma mudança em **A** poderia fazer **D** recalcular duas vezes em um sistema ingênuo, causando estados inconsistentes (glitches).

A Solução Híbrida do Angular: O algoritmo Push/Pull garante que **D** é calculado apenas uma vez, após todas as suas dependências terem sido atualizadas.

As Ferramentas Fundamentais: `signal`, `computed` e `effect`



`signal` (Estado Gravável)

A fonte da verdade. Armazena estado que pode ser modificado.

API: ` `.set(valor)` , ` `.update(fn)`

****Ponto Chave**:** A mutação de objetos internos não dispara a notificação; use ` `.update()` com imutabilidade.

```
const count = signal(0);
count.set(1);
```



`computed` (Estado Derivado)

Valor derivado de outros sinais.

Características: Memoização (caching) e Avaliação Preguiçosa (lazy evaluation).

****Ponto Chave**:** Sempre prefira `computed` para derivar estado; nunca use `effect` para isso.

```
const double = computed(() =>
  count() * 2);
```



`effect` (Ação Colateral)

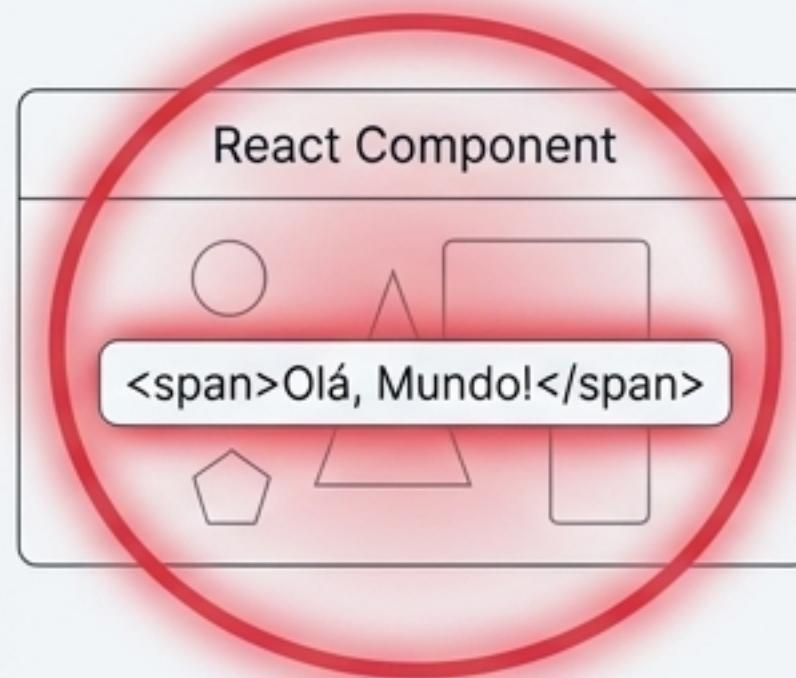
Interage com o 'mundo exterior' (DOM, logs, localStorage). Não retorna valor.

Características: Agendado assincronamente e executa apenas uma vez após múltiplas mudanças síncronas.

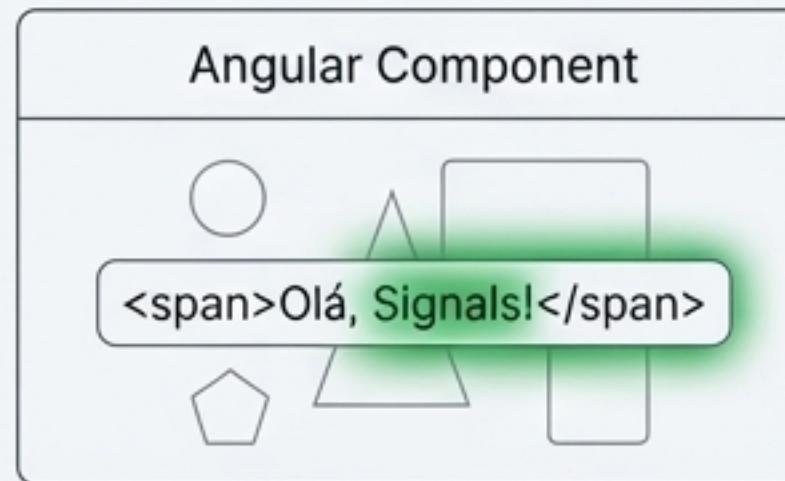
****Ponto Chave**:** Evite usar para sincronizar estado entre sinais.

```
effect(() => console.log(count()));
```

Granularidade Cirúrgica vs. Re-renderização em Bloco



Re-renderização do Componente



Atualização Cirúrgica do DOM

Característica	Angular Signals	React Hooks (<code>'useState'</code> / <code>'useEffect'</code>)
Unidade de Atualização	O nó específico do DOM (ex: o texto em um <code></code>).	O componente inteiro é re-renderizado.
Mecanismo	Reatividade Fina , sem VDOM diffing.	Virtual DOM diffing.
Dependências	Rastreamento automático e dinâmico .	Declaração manual em arrays de dependência.
Fonte de Erros	Menor chance de bugs de dependência.	Erros comuns com dependências obsoletas (stale closures).

O Melhor de Dois Mundos: Estado Síncrono vs. Eventos Assíncronos

A relação é de **complementaridade**, não de substituição.

Signals são para estado, RxJS é para eventos.



Use Signals para:

- ✓ Estado local de componente.
- ✓ Estado derivado para a UI (via `computed`).
- ✓ Integração direta com o template (sem `async` pipe).

Use RxJS para:

- ✓ Eventos complexos e orquestração de tempo (debounce, switchMap).
- ✓ Comunicação com WebSockets.
- ✓ Gerenciamento de fluxos de eventos contínuos.

Fechando o Ciclo: APIs Avançadas para Ergonomia

`linkedSignal`

Resolve: Estado que depende de outro, mas também precisa ser gravável. Expressa o padrão 'reset on change' declarativamente, eliminando `effects` imperativos.

Antes (com `effect`)

```
// Propenso a erros de sincronia
effect(() => {
  country(); // registra dependência
  city.set(DEFAULT_CITIES[country()]);
});
```



Depois (com `linkedSignal`)

```
// Declarativo e seguro
const city = linkedSignal(
  () => DEFAULT_CITIES[country()]
);
```

Mais limpo e declarativo

`Resource API` (Experimental)

Resolve: Código boilerplate para carregar dados assíncronos, gerenciando estados de `isLoading`, `error` e `value` em um único objeto reativo.

Antes (manual)

```
// Três sinais para gerenciar
const data = signal(null);
const isLoading = signal(true);
const error = signal(null);
// Lógica em ngHoInit...
```



Gerenciamento de estado unificado

Depois (com `Resource API`)

```
// Um único objeto reativo
const user = resource(
  (userId$) => this.api.getUser(userId$())
);
// Acessa user.value(), user.isLoading()...
```

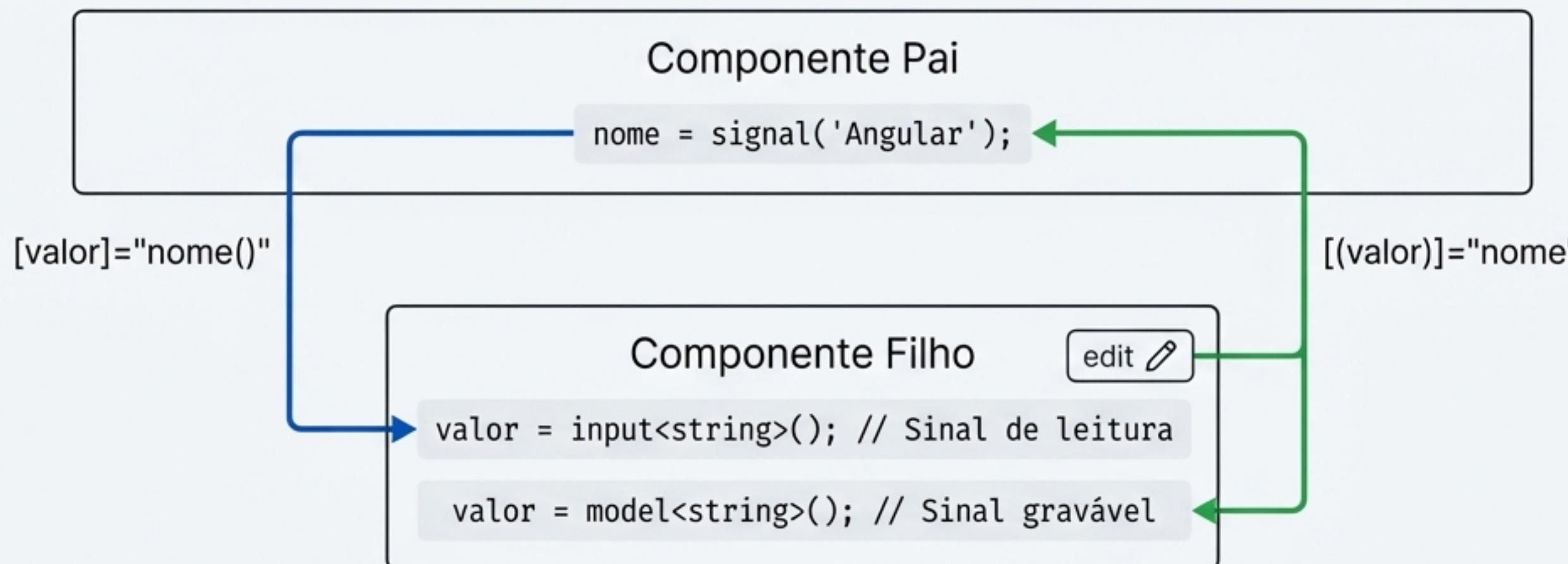
Reatividade de Ponta a Ponta: `input()` e `model()`

`input()`

Substitui o decorador `@Input`. Transforma propriedades de entrada de componentes em sinais de *leitura*. Garante que os dados já entram no componente dentro do sistema reativo.

`model()`

Habilita o *two-way binding* nativo com Signals. Expõe um sinal *gravável* que, quando alterado internamente, emite um evento de mudança para o componente pai.



Impacto

Simplifica drasticamente a criação de componentes de formulário e a comunicação entre componentes, eliminando a necessidade de saídas de eventos (`@Output`) para padrões de `(change)`.

O Objetivo Final: A Promessa de um Futuro ‘Zoneless’

Benefícios da Remoção do Zone.js:

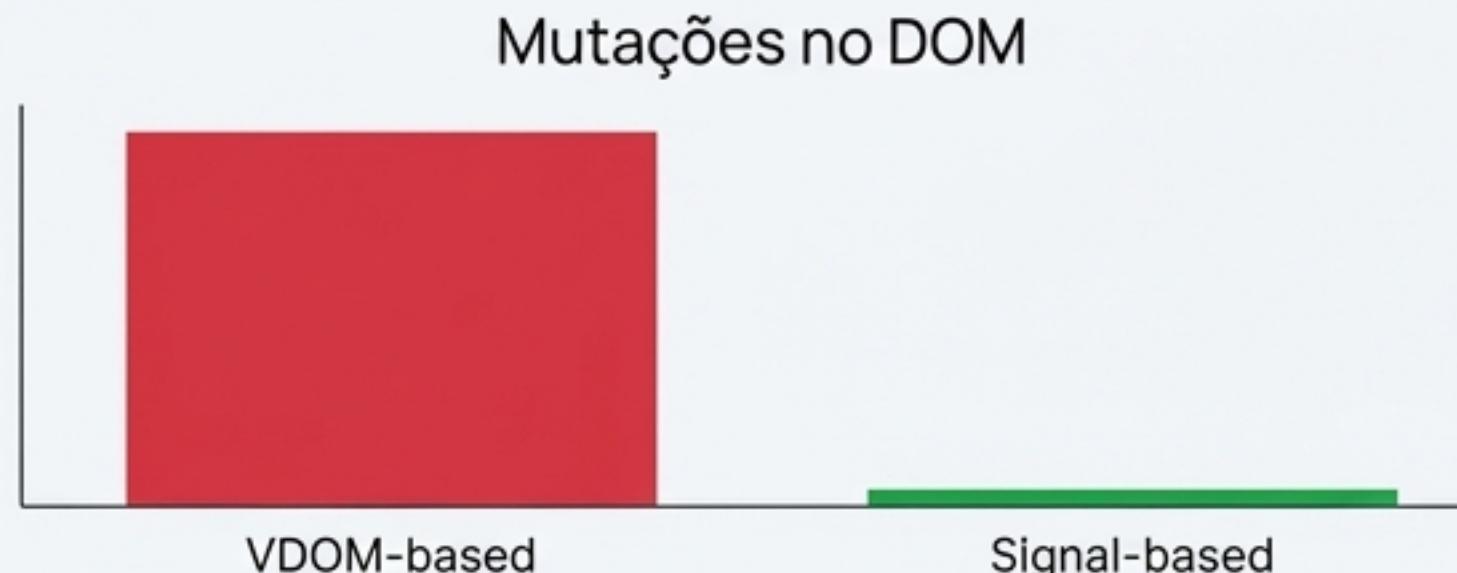
- Menor Tamanho de Bundle:** Redução do peso inicial da aplicação.
- Inicialização Mais Rápida (TTI):** Sem a necessidade de *monkey-patching* das APIs do navegador.
- `Stack Traces` Limpos:** Debugging mais fácil, sem a ofuscação das camadas internas do Zone.
- Controle Explícito:** A detecção de mudanças é disparada apenas por atualizações de sinais ou eventos, tornando o fluxo de dados mais previsível.



Os Números Não Mentem: O Impacto Real da Reatividade Fina

99.9%

Redução nas mutações do DOM
(cenários de tabelas massivas)



>70%

Redução no uso de memória heap



****A Conclusão Reforçada**:** A complexidade da atualização muda de $O(n)$ para $O(1)$ no caso ideal.

Checklist para Produção: Escrevendo Código Reativo Idiomático



Prefira `input()` sobre `@Input`: Garante que os dados externos entrem no fluxo reativo imediatamente.



Use `OnPush` sempre: Com Signals, a estratégia `OnPush` torna-se trivial de gerenciar e maximiza a performance.



Mantenha o Estado Plano (Flat State): Evite sinais profundamente aninhados. Prefira múltiplos sinais atômicos.



Use a `Resource API` para Requisições HTTP: Substitua o padrão manual de `ngOnInit` + `subscribe`.



Depure com Angular DevTools: Utilize a visualização do grafo de sinais para entender as dependências e evitar recomputações desnecessárias.

Armadilhas a Evitar: Anti-Padrões Comuns e Suas Soluções

Anti-Padrão 1: `Effect Hell` (Inferno de Efeitos)



Erro

Usar `effect` para propagar dados entre sinais.

```
// ERRADO
effect(() => {
  b.set(a() * 2);
});
```



Solução

Sempre use `computed` para dados derivados.

```
// CERTO
const b = computed(() => a() * 2);
```

Anti-Padrão 2: Mutação de Objetos



Erro

Mudar uma propriedade interna (`user().name = 'Novo'`). A referência não muda.

```
// ERRADO
user().name = 'Novo'; // Não notifica
user.set(user());
```



Solução

Use `update()` para criar uma nova referência (imutabilidade).

```
// CERTO
user.update(u => ({...u, name: 'Novo'}));
```

Anti-Padrão 3: Leitura Desprotegida em Efeitos



Erro

Ler um sinal em um `effect`, criando uma dependência indesejada.

```
// ERRADO
effect(() => {
  // Cria dependência em 'someSignal'
  sendAnalytics(someSignal());
});
```



Solução

Use `untracked()` para ler o valor sem criar dependência.

```
// CERTO
effect(() => {
  sendAnalytics(untracked(someSignal()));
});
```

Dominar Signals é Dominar o Futuro do Angular

A mudança para Signals não é apenas sobre performance, mas sobre a criação de um modelo mental superior para a engenharia de UI. Ela traz clareza, previsibilidade e robustez para aplicações complexas.

A arquitetura Zoneless e Signal-First definirá a próxima década de desenvolvimento de aplicações web escaláveis. A adoção desta tecnologia entrega um controle sem precedentes sobre o ciclo de vida e a renderização da aplicação.

Documentação Oficial
do Angular Signals



NotebookLM