FASE 2 — SIGNALS E REATIVIDADE GRANULAR: O MANUAL DEFINITIVO DE ARQUITETURA

1. Introdução: A Mudança de Paradigma na Engenharia Web

A introdução dos *Signals* (Sinais) no ecossistema Angular representa a transformação arquitetural mais profunda desde a reescrita do framework do AngularJS para a versão 2. Não se trata meramente de uma nova sintaxe ou de uma API utilitária adicional; trata-se de uma substituição fundamental do motor de detecção de mudanças, migrando de um modelo de verificação global e implícita para um modelo de **Reatividade Granular** (*Fine-Grained Reactivity*). Este relatório analisa exaustivamente essa transição, dissecando os mecanismos internos, as implicações de performance e o impacto no desenvolvimento de interfaces modernas, servindo como um manual definitivo tanto para engenheiros experientes quanto para aqueles que estão adentrando este novo paradigma.[1]

A motivação primária para esta evolução reside nas limitações intrínsecas do modelo baseado em Zone.js, que serviu ao Angular por mais de uma década. Embora o Zone.js tenha proporcionado uma experiência de desenvolvimento conveniente ao abstrair a detecção de mudanças — interceptando eventos assíncronos como setTimeout, Promise e interações do DOM — ele impôs um teto de performance. O mecanismo de *dirty checking* (verificação de sujeira) global exige que o framework percorra a árvore de componentes do topo à base para identificar mutações de estado, resultando em uma complexidade computacional de O(n), onde *n* é o número de componentes na aplicação. A Reatividade Granular, viabilizada pelos Signals, inverte essa lógica para um modelo O(1) no melhor caso, onde o dado notifica diretamente a visão que necessita de atualização, ignorando a hierarquia de componentes.[2]

---

2. Fundamentos Teóricos da Reatividade Granular

Para compreender os Signals, é imperativo dissociar o conceito de "reatividade" da implementação baseada em streams (como RxJS) ou em Virtual DOM (como React). A Reatividade Granular opera sob princípios distintos de ciência da computação, focados na sincronização de estado e na minimização de computação redundante.

2.1 O Modelo Mental: A Metáfora da Planilha Eletrônica

A analogia mais precisa para explicar Signals a um "leigo técnico" é o funcionamento de uma planilha eletrônica, como o Excel. Considere três células: A1, B1 e C1. Se definirmos que A1

contém o valor 10 e B1 contém 20, e configurarmos C1 com a fórmula =A1+B1, o valor exibido em C1 será 30. No momento em que o usuário altera A1 para 15, C1 atualiza instantaneamente para 35.

Neste cenário, C1 não precisa ser "instruído" imperativamente a recalcular; a relação de dependência declarada garante a sincronização. Mais importante ainda, o Excel não recalcula todas as células da planilha, apenas aquelas que dependem direta ou indiretamente de A1. Este é o cerne da Reatividade Granular: um grafo de dependência onde a mudança flui apenas através das arestas conectadas, garantindo eficiência máxima.[1]

2.2 O Grafo de Dependência Dinâmico

Diferentemente de sistemas que exigem declaração manual de dependências (como o array de dependências do useEffect no React), os Signals constroem seu grafo de dependência em tempo de execução. O sistema categoriza os elementos em dois papéis fundamentais:

- **Produtores (Producers):** Nós que detêm valores e emitem notificações de mudança. Um WritableSignal é um produtor puro.
- **Consumidores (Consumers):** Nós que dependem de valores de produtores. Um template HTML ou um effect são consumidores puros.
- **Nós Híbridos:** Um computed signal atua simultaneamente como consumidor (lê outros sinais) e produtor (emite um valor derivado).[5]

O rastreamento é dinâmico e automático. Quando um consumidor lê um sinal dentro de um contexto reativo, uma aresta é criada no grafo. Se a lógica condicional alterar quais sinais são lidos (por exemplo, um if que deixa de ler o sinal B porque o sinal A é falso), a aresta de dependência com B é removida automaticamente. Isso previne o "over-reactivity" (reatividade excessiva), onde cálculos ocorrem desnecessariamente baseados em dados obsoletos ou irrelevantes para o estado atual.[5]

2.3 O Problema do Diamante e o Algoritmo Push/Pull

Um desafio clássico em sistemas reativos é o "Problema do Diamante" (*The Diamond Problem*). Imagine um grafo onde um sinal A alimenta os sinais B e C, e ambos B e C alimentam um sinal D. Se A mudar, B e C serão atualizados. Em um sistema reativo ingênuo (puramente *Push*), D poderia ser notificado duas vezes: uma vez quando B atualiza e outra quando C atualiza. Isso não apenas desperdiça ciclos de CPU, mas pode levar a estados inconsistentes temporários (glitches), onde D é calculado com o novo valor de B mas o valor antigo de C.[3]

O Angular resolve isso implementando um algoritmo híbrido **Push/Pull**:

1. **Fase Push (Notificação):** Quando o sinal A muda, ele envia uma notificação "rápida" para B e C dizendo apenas "estou sujo" (*dirty*). Essa notificação se propaga até D. Nenhum cálculo é feito nesta fase; apenas bandeiras de estado são alteradas.

2. **Fase Pull (Execução Preguiçosa):** Quando o valor de D é efetivamente necessário (por exemplo, para renderizar o template), D verifica seu estado. Se estiver marcado como "sujo", ele solicita os valores de B e C. B e C, por sua vez, recalculam seus valores baseados em A.
3. **Garantia Glitch-Free:** Como D só executa o cálculo final após solicitar os dados atualizados de suas dependências, ele é calculado apenas uma vez, garantindo consistência atômica do estado.[8]

---

3. Os Primitivos Técnicos: Anatomia dos Signals no Angular

O Angular expõe a reatividade granular através de três primitivos principais: signal, computed e effect. A compreensão profunda de suas características de memória e ciclo de vida é crucial para a implementação correta.

3.1 Writable Signals (Sinais Graváveis)

O WritableSignal é a unidade básica de estado. Ele armazena um valor e fornece uma interface para leitura e escrita. A leitura é feita invocando o sinal como uma função (getter), o que registra o contexto atual como um dependente.

A escrita deve ser feita através de métodos explícitos para garantir a integridade do fluxo de dados:

- **set(value):** Substitui o valor completamente. É útil para tipos primitivos ou substituição total de objetos.
- **update(fn):** Aceita uma função que recebe o valor atual e retorna o novo valor. É a abordagem recomendada para estados complexos, pois garante que a atualização seja baseada no valor mais recente de forma atômica.

A mutabilidade direta de objetos internos (ex: array.push) não dispara a notificação de mudança por padrão, pois os Signals utilizam verificação de igualdade referencial (Object.is). Para arrays e objetos, a prática recomendada é tratar os dados como imutáveis, retornando novas referências através do método update, o que assegura a propagação correta da reatividade.[1]

3.2 Computed Signals (Sinais Computados)

Sinais computados representam valores derivados. Eles são a personificação do princípio declarativo: definem *o que* é o dado, não *como* ou *quando* atualizá-lo.

Características técnicas distintivas:

1. **Memoização (Caching):** O valor de um computed é armazenado em cache. Se o sinal

for lido múltiplas vezes sem que suas dependências mudem, a função de derivação não é reexecutada; o valor em cache é retornado.
2. **Avaliação Preguiçosa (Lazy Evaluation):** Se um computed é definido mas nunca lido (por exemplo, pertence a um componente que não está na tela), sua lógica de cálculo nunca será executada, economizando recursos computacionais.
3. **Somente Leitura:** Não possuem métodos set ou update. Seu valor é estritamente uma função de suas dependências.[6]

3.3 Effects (Efeitos Colaterais)

Enquanto computed gerencia dados, effect gerencia comportamentos. Um efeito é uma operação que executa em resposta a mudanças nos sinais, mas não retorna um valor para o grafo reativo. Eles são utilizados para interagir com o "mundo exterior" ao sistema reativo, como manipular o DOM manualmente, realizar logs, ou sincronizar dados com localStorage.

Os efeitos no Angular são agendados assincronamente. Isso significa que se múltiplos sinais mudarem sincronicamente em sequência, o efeito rodará apenas uma vez após a estabilização das mudanças (coalescência), prevenindo thrashing na UI. É fundamental evitar o uso de efeitos para sincronizar estado (ex: "quando A mudar, atualize B"), pois isso reintroduz os problemas de rastreamento e complexidade que os sinais visam resolver. Para derivação de estado, sempre se deve preferir computed.[10]

---

4. Reatividade Granular: O Diferencial Comparativo

A introdução dos Signals coloca o Angular em competição direta com os modelos de reatividade de outros frameworks modernos. A análise comparativa revela as nuances de performance e ergonomia de desenvolvimento.

4.1 Angular Signals vs. React Hooks (useState/useEffect)

A diferença fundamental reside na granularidade da atualização e no rastreamento de dependências.

O React utiliza um modelo de "Re-renderização por Componente". Quando um estado muda via useState, o componente inteiro (e potencialmente seus filhos) é reexecutado para gerar uma nova árvore de Virtual DOM. O React então compara (diffing) essa nova árvore com a anterior para aplicar mudanças ao DOM real. Embora otimizado, esse processo gera *overhead* de memória e CPU. Além disso, o React exige que o desenvolvedor declare manualmente as dependências em useEffect e useMemo, o que é uma fonte frequente de bugs e vazamentos de memória se o array de dependências estiver incorreto.[12]

O Angular, com Signals, adota a "Reatividade Fina". O framework não precisa reavaliar o componente inteiro. Se um sinal muda, apenas o nó específico do DOM (ex: um texto dentro

de um) é atualizado. Não há necessidade de Virtual DOM diffing para essas atualizações. As dependências são rastreadas automaticamente, eliminando a necessidade de arrays de dependência manuais e prevenindo erros de dependências obsoletas (stale closures).[2]

4.2 Angular Signals vs. RxJS

A relação entre Signals e RxJS é de complementaridade, não de substituição total. RxJS é uma biblioteca baseada em eventos (*Push*), ideal para orquestrar fluxos assíncronos complexos ao longo do tempo (ex: debouncing, throttling, switchMap). Signals são primitivos de estado (*Push/Pull*), ideais para gerenciar valores síncronos e renderização de UI.

A tabela abaixo sintetiza a aplicabilidade de cada tecnologia no Angular moderno [16]:

Cenário de Uso	Tecnologia Recomendada	Justificativa Técnica
Estado Local de Componente	**Signals**	API mais simples, sem necessidade de subscrição/desubscrição manual, integração direta com template.
Estado Derivado (Cálculos)	**Computed**	Memoização automática, avaliação preguiçosa, sintaxe declarativa.
Eventos Complexos (ex: Busca)	**RxJS**	Operadores como debounceTime e switchMap são imbatíveis para coordenar tempo e cancelamento.
Fluxos WebSocket	**RxJS**	Natureza de "stream" contínuo de eventos encaixa-se no modelo Push do RxJS.
Renderização na View	**Signals**	Elimina a necessidade do AsyncPipe, reduz change detection global.

4.3 Angular vs. SolidJS e Vue

É importante reconhecer que o design dos Angular Signals foi fortemente influenciado pelo SolidJS, criado por Ryan Carniato. O SolidJS foi pioneiro na reatividade granular moderna sem Virtual DOM, compilando componentes para instruções imperativas precisas. O Angular adota princípios similares, mas os integra em um framework "batteries-included". O Vue.js também utiliza um sistema de reatividade baseado em proxies (Vue 3 Composition API), que é conceitualmente próximo aos Signals, mas o Angular se diferencia pela integração profunda com sua injeção de dependência e pelo sistema híbrido Push/Pull rigoroso para evitar glitches.[19]

---

5. Fase 2: APIs Avançadas e Integração Arquitetural

Com a maturação dos Signals nas versões 18 e 19 do Angular, surgiram APIs avançadas que resolvem lacunas iniciais de ergonomia e funcionalidade, consolidando a "Fase 2" da reatividade.

5.1 linkedSignal: Resolvendo o Estado Dependente Gravável

Um padrão recorrente no desenvolvimento de UI é a necessidade de um estado que seja dependente de outro, mas que também possa ser editado pelo usuário. Considere um formulário onde a seleção de um "País" deve resetar a seleção de "Cidade" para um valor padrão, mas o usuário deve poder alterar a "Cidade" livremente depois.

Anteriormente, isso exigia efeitos colaterais (effect) propensos a erros para sincronizar os dois estados. O linkedSignal introduz um primitivo específico para isso. Ele cria um sinal gravável que possui uma "ligação" com um sinal fonte. Quando a fonte muda, o linkedSignal reseta seu valor baseado em uma função de computação. Se a fonte não mudar, ele se comporta como um WritableSignal normal.[22]

Isso permite expressar padrões de "reset on change" (resetar ao mudar) de forma puramente declarativa, eliminando condições de corrida e efeitos imperativos.

5.2 Resource API: A Ponte para o Assíncrono

Até recentemente, carregar dados assíncronos (HTTP) e exibi-los em Signals exigia muito código boilerplate (transformar Observable em Signal, gerenciar estados de loading/erro manualmente). A Resource API (experimental no v19) resolve isso integrando promessas e observables diretamente ao mundo dos Signals.

A função resource aceita um carregador assíncrono (loader) e um objeto de parâmetros reativos (params). Quando os sinais dentro de params mudam, o resource dispara

automaticamente o carregador (similar a um switchMap no RxJS). O retorno é um objeto contendo sinais para value, isLoading, error e status.

Exemplo de impacto arquitetural:
Em vez de gerenciar três variáveis separadas (data, loading, error) e subscrições manuais, o desenvolvedor declara um único recurso. O template pode então reagir a esses estados de forma limpa. Isso padroniza o tratamento de assincronicidade, reduzindo drasticamente a superfície de bugs relacionados a condições de corrida em requisições HTTP.24

5.3 Inputs como Signals e Model Inputs

A substituição do decorador @Input pela função input() transforma as propriedades de entrada dos componentes em sinais de leitura. Isso fecha o ciclo de reatividade: os dados entram no componente como sinais, são transformados por computed, e consumidos pelo template, sem nunca sair do contexto reativo.

Além disso, os model inputs (model()) introduzem suporte nativo para *two-way binding* com sinais. Diferente do input() que é somente leitura, o model() expõe um sinal gravável que, quando alterado internamente, emite um evento de mudança para o pai, simplificando drasticamente a criação de componentes de formulário reutilizáveis.[27]

---

6. Performance e o Futuro Zoneless (Sem Zonas)

O impacto final da adoção de Signals é a viabilidade de aplicações "Zoneless" (Sem Zone.js). A remoção do zone.js resulta em:

1. **Redução do Bundle:** O zone.js é uma biblioteca pesada que precisa ser carregada antes de qualquer outra coisa. Sua remoção diminui o peso inicial da aplicação.
2. **Melhoria na Inicialização:** Sem a necessidade de fazer *monkey-patching* em todas as APIs do navegador, o tempo de TTI (Time to Interactive) diminui.
3. **Stack Traces Limpos:** Erros não são mais ofuscados pelas múltiplas camadas de execução interna do Zone, facilitando o debugging.[29]

Benchmarks sintéticos comparando arquiteturas baseadas em VDOM (como React) contra arquiteturas de reatividade fina (como Solid/Angular Signals) mostram que a abordagem de sinais pode reduzir as mutações no DOM em até 99.9% em cenários de atualizações massivas de tabelas, e reduzir o uso de memória heap em mais de 70%.[2] Isso ocorre porque os Signals eliminam a alocação de memória necessária para criar árvores virtuais de comparação a cada ciclo de renderização.

Para habilitar o modo Zoneless experimental, o Angular fornece o provedor provideExperimentalZonelessChangeDetection(). Neste modo, a detecção de mudanças não é mais disparada magicamente por qualquer evento; ela é disparada explicitamente por atualizações de sinais, eventos de template, ou pelo pipe async. Isso força uma disciplina de

arquitetura onde o fluxo de dados deve ser explícito, resultando em aplicações mais previsíveis e performáticas.[31]

---

7. Melhores Práticas e Anti-Padrões: O Guia de Sobrevivência

A transição para Signals exige disciplina para evitar recriar problemas antigos com novas ferramentas.

7.1 Anti-Padrões Comuns

- **Effect Hell (Inferno de Efeitos):** Usar effect para propagar dados entre sinais (ex: "quando A mudar, setar B"). Isso quebra o fluxo unidirecional e torna o rastreamento de dados impossível. Sempre use computed para dados derivados.[10]
- **Leitura Desprotegida em Efeitos:** Ler um sinal dentro de um efeito apenas para obter seu valor, sem querer reagir a ele. Solução: utilizar a função untracked() para ler o valor sem criar uma dependência no grafo.[1]
- **Mutação de Objetos:** Tentar atualizar um sinal de objeto mudando uma propriedade interna (user().name = 'Novo'). Como a referência do objeto não mudou, o sinal não notifica os dependentes. Solução: usar user.update(u => ({...u, name: 'Novo'})) ou linkedSignal com comparadores de igualdade customizados.[11]

7.2 Checklist para Produção

[34]

1. **Prefira input() sobre @Input:** Garante que dados externos entrem no fluxo reativo imediatamente.
2. **Use OnPush sempre:** Com Signals, a estratégia OnPush torna-se trivial de gerenciar e maximiza a performance da view.
3. **Flat State (Estado Plano):** Evite sinais aninhados profundamente. Prefira múltiplos sinais atômicos ou objetos planos para facilitar a granularidade das atualizações.
4. **Resource para HTTP:** Substitua o padrão manual de ngOnInit + subscribe pelo uso declarativo da resource API para carregar dados iniciais.
5. **Debug com Angular DevTools:** Utilize as novas ferramentas de visualização do grafo de sinais para entender quem depende de quem e evitar ciclos ou recomputações desnecessárias.

---

8. Conclusão

A "Fase 2" dos Signals no Angular não é apenas uma atualização incremental; é a consolidação de um modelo mental superior para desenvolvimento de interfaces. Ao combinar a ergonomia declarativa dos Signals com a performance cirúrgica da reatividade granular, o Angular resolve o dilema histórico entre facilidade de uso e otimização de recursos.

Para o profissional, dominar Signals é dominar o futuro da plataforma. O modelo baseado em Zone.js será gradualmente relegado ao legado, enquanto a arquitetura *Zoneless* e *Signal-First* definirá a próxima década de aplicações web escaláveis. O diferencial competitivo não está apenas na performance bruta, mas na clareza e previsibilidade que este modelo traz para a engenharia de software frontend complexa. A adoção desta tecnologia exige estudo e mudança de hábitos, mas entrega em troca um controle sem precedentes sobre o ciclo de vida e a renderização da aplicação.

Referências citadas

1. Signals • Overview • Angular, acessado em dezembro 27, 2025, https://angular.dev/guide/signals
2. I Benchmarked Signals vs Virtual DOM — Here's What I Found ..., acessado em dezembro 27, 2025, https://dev.to/mike_hanol_e21eef42461b5e/i-benchmarked-signals-vs-virtual-dom-heres-what-i-found-3do7
3. Stop Thinking in Trees: Why Angular Signals Require Graph Thinking | by Syam Pradeep Reddy | Dec, 2025 | Medium, acessado em dezembro 27, 2025, https://medium.com/@syampradeep.ca/stop-thinking-in-trees-why-angular-signals-require-graph-thinking-3098fea33b8b
4. Beyond Signals: The Next Big Shift in Web Reactivity by Ryan Carniato - GitNation, acessado em dezembro 27, 2025, https://gitnation.com/contents/beyond-signals
5. Dependency Graph in Angular Signals - DEV Community, acessado em dezembro 27, 2025, https://dev.to/oz/dependency-graph-in-angular-signals-2jon
6. Signals in Angular: deep dive for busy developers - Angular.love, acessado em dezembro 27, 2025, https://angular.love/signals-in-angular-deep-dive-for-busy-developers/
7. Reactive algorithms: How Angular took the right path, acessado em dezembro 27, 2025, https://medium.com/coreteq/reactive-algorithms-how-angular-took-the-right-path-c90e9f0183c2
8. Angular: Why Signals Are a Game-Changer? | by Vincent BOURDEIX | Medium, acessado em dezembro 27, 2025, https://medium.com/@vbourdeix/angular-why-signals-are-a-game-changer-aa2752987809
9. Angular Signals, acessado em dezembro 27, 2025, https://v17.angular.io/guide/signals

10. 6 Common effect() Mistakes in Angular Signals — and How to Fix Them | by Krunal vekariya, acessado em dezembro 27, 2025, https://medium.com/@krunalvekariya12345/6-common-effect-mistakes-in-angular-signals-and-how-to-fix-them-7cf21b911d69
11. Avoid These Angular Signals Mistakes - A Must-Read for Every Developer, acessado em dezembro 27, 2025, https://dev.to/codewithrajat/avoid-these-angular-signals-mistakes-a-must-read-for-every-developer-3d72
12. Angular Signals vs. React useState: Key Similarities and Differences | by Maham Cheema, acessado em dezembro 27, 2025, https://medium.com/@maham.cheema91/angular-signals-vs-react-usestate-key-similarities-and-differences-22fa437b1404
13. Angular Signals vs. React: A Deep Dive into Reactivity | by Sreekumar P - Medium, acessado em dezembro 27, 2025, https://sreekumarp.medium.com/angular-signals-vs-react-a-deep-dive-into-reactivity-3c6835f92945
14. Fine-Grained Reactivity in React: How Preact Signals Do It | by Sparsh Malhotra - Medium, acessado em dezembro 27, 2025, https://medium.com/@sparshmalhotraaa/fine-grained-reactivity-in-react-how-preact-signals-do-it-a282943b2bd8
15. The Angular Signals Revolution: Rethinking Reactivity - AppSignal Blog, acessado em dezembro 27, 2025, https://blog.appsignal.com/2025/09/17/the-angular-signals-revolution-rethinking-reactivity.html
16. Signals vs RxJS in Angular 17: Comparison Guide - NareshIT, acessado em dezembro 27, 2025, https://nareshit.com/blogs/signals-vs-rxjs-angular-17-comparison-guide
17. Angular Signals vs RxJS — Do We Still Need Observables? - DEV Community, acessado em dezembro 27, 2025, https://dev.to/rohit_singh_ee84e64941db7/angular-signals-vs-rxjs-do-we-still-need-observables-3of3
18. Will Signals replace RxJs? - Angular Experts, acessado em dezembro 27, 2025, https://angularexperts.io/blog/signals-vs-rxjs/
19. Signals: Fine-grained Reactivity for JavaScript Frameworks - SitePoint, acessado em dezembro 27, 2025, https://www.sitepoint.com/signals-fine-grained-javascript-framework-reactivity/
20. Ryan Carniato on SolidJS: Web Innovation | gotopia.tech - GOTO Conferences, acessado em dezembro 27, 2025, https://gotopia.tech/articles/235/ryan-carniato-solidjs-javascript-frameworks
21. Comparing reactivity models | CS 484: Secure Web Application Development, acessado em dezembro 27, 2025, https://www.cs.uic.edu/~ckanich/cs484/f24/readings/chapter-2-client-side-web-development/other-reactivity-models/index.html
22. Angular linkedSignal(): The Missing Link in Signal-Based Reactivity, acessado em dezembro 27, 2025, https://blog.angular-university.io/angular-linkedsignal/

23. Dependent state with linkedSignal - Angular, acessado em dezembro 27, 2025, https://angular.dev/guide/signals/linked-signal
24. The Angular Resource API. Introduction | by Preston Lamb | ngconf | Nov, 2025 - Medium, acessado em dezembro 27, 2025, https://medium.com/ngconf/the-angular-resource-api-fe8fa2b538bf
25. Mastering Angular's Resource API: A Practical Guide - DEV Community, acessado em dezembro 27, 2025, https://dev.to/manthanank/mastering-angulars-resource-api-a-practical-guide-3k08
26. Angular's Resource APIs Are Broken - Let's Fix Them!, acessado em dezembro 27, 2025, https://angular.schule/blog/2025-10-rx-resource-is-broken/
27. Model inputs - Angular, acessado em dezembro 27, 2025, https://v17.angular.io/guide/model-inputs
28. Accepting data with input properties - Angular, acessado em dezembro 27, 2025, https://angular.dev/guide/components/inputs
29. How Angular 20.2 Replaces Zone.js for Better Performance, acessado em dezembro 27, 2025, https://javascript-conference.com/blog/angular-20-zoneless-mode-performance-migration-guide/
30. Zoneless • Angular, acessado em dezembro 27, 2025, https://angular.dev/guide/zoneless
31. Angular Change Detection with Zoneless - DEV Community, acessado em dezembro 27, 2025, https://dev.to/soumayaerradi/angular-change-detection-with-zoneless-413f
32. Angular v21 Goes Zoneless by Default: What Changes & Why It's Faster - Push-Based, acessado em dezembro 27, 2025, https://push-based.io/article/angular-v21-goes-zoneless-by-default-what-changes-why-its-faster-and-how-to
33. Stop Misusing Effects! Linked Signals Are the Better Alternative! - Angular Experts, acessado em dezembro 27, 2025, https://angularexperts.io/blog/stop-misusing-effects/
34. Angular Code Review Checklist: All Steps Included - Redwerk, acessado em dezembro 27, 2025, https://redwerk.com/blog/angular-code-review-checklist-all-steps-included/
35. Angular Signals: Best Practices - Medium, acessado em dezembro 27, 2025, https://medium.com/@eugeniyoz/angular-signals-best-practices-9ac837ab1cec