Nem todo sistema precisa de um banco de dados: uma decisão arquitetural contextual

Persistência simples, trade-offs reais e quando "planilha como storage" faz sentido

1. Introdução — O problema antes da solução

Na prática profissional, sobretudo em projetos pequenos, MVPs e sistemas internos, é comum observar um padrão recorrente: a adoção de arquiteturas completas — frequentemente descritas informalmente, no meio técnico, como "canhão para matar formiga" — com backend estruturado, banco de dados transacional, ORM, migrações e camadas adicionais, mesmo quando o problema a ser resolvido é essencialmente simples. Esse comportamento não decorre de má prática deliberada, mas frequentemente do uso acrítico de "arquiteturas de referência" replicadas sem reflexão contextual.

Esse fenômeno cria uma tensão clara entre complexidade arquitetural e necessidade real. Em muitos cenários, o custo de manter infraestrutura, lidar com deploys, backups, observabilidade e governança técnica supera o valor entregue pelo sistema — especialmente quando os requisitos envolvem apenas persistência básica, acesso via API e baixa concorrência.

O ponto central deste artigo não é questionar a relevância de bancos de dados tradicionais — eles existem por razões técnicas sólidas. O que se propõe aqui é uma reflexão diferente: há contextos em que um banco de dados transacional é excesso, e em que outras formas de persistência podem atender melhor ao problema imediato.

A tese defendida é a seguinte: nem todo sistema precisa nascer com um banco de dados transacional tradicional. Em determinados cenários, soluções de persistência simples, orientadas a workflow humano e acessadas por API — como Google Sheets combinado com Google Apps Script — podem representar uma decisão arquitetural consciente, desde que suas limitações sejam explícitas, bem delimitadas e assumidas.

2. Desenvolvimento — Fundamentação técnica

Do ponto de vista da arquitetura de software, essa tese não rompe com princípios clássicos; ao contrário, está alinhada a eles. Em Arquitetura Limpa, Robert C. Martin defende que mecanismos de persistência devem ser tratados como detalhes de implementação, e não como o núcleo da aplicação. A lógica de negócio deve existir independentemente da tecnologia usada para armazenar dados, permitindo que essa tecnologia seja substituída conforme o contexto e a maturidade do sistema.

Essa abordagem dialoga diretamente com dois princípios amplamente aceitos na engenharia de software:

- YAGNI (You Ain't Gonna Need It): não introduzir infraestrutura antes que ela seja estritamente necessária.
- KISS (Keep It Simple, Stupid): simplicidade como valor técnico, não como improviso.

Aplicados à arquitetura, esses princípios indicam que iniciar um sistema com uma infraestrutura pesada, sem necessidade concreta, pode ser tão prejudicial quanto negligenciar requisitos críticos em sistemas de alta complexidade.

Persistência não é sinônimo de banco de dados

Um ajuste conceitual importante precisa ser explicitado: persistência e banco de dados transacional não são termos equivalentes. Um banco transacional oferece garantias fortes — como controle de concorrência, integridade referencial e propriedades ACID — que são indispensáveis em muitos domínios. Já soluções como Google Sheets, quando acessadas via API, oferecem persistência tabular com acesso programático, adequada apenas a determinados tipos de problema claramente restritos e bem delimitados.

O argumento aqui não é de equivalência funcional, mas de adequação contextual.

3. Prova de conceito — A arquitetura em prática

Em alto nível, a arquitetura proposta é simples:

- um cliente (web ou mobile) envia uma requisição HTTP
- uma camada de API processa regras básicas
- uma camada de persistência tabular armazena ou retorna dados

Diagrama conceitual do fluxo:

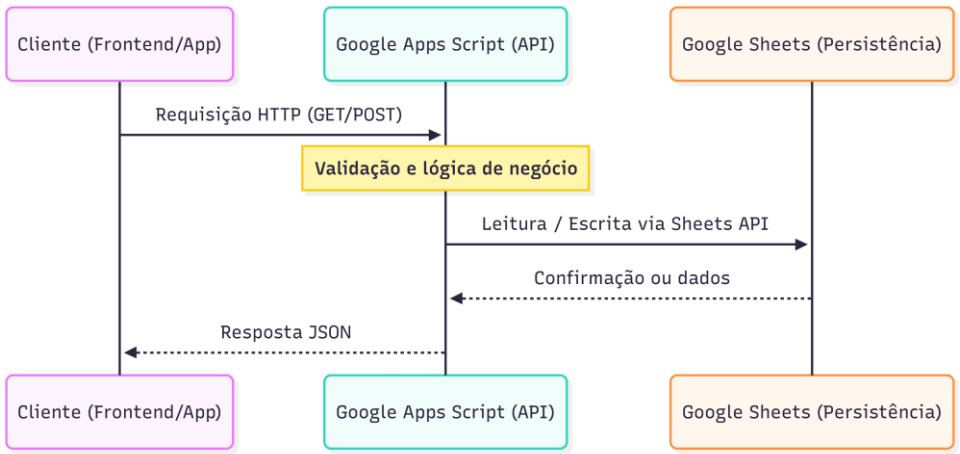

Figura 1 — Fluxo de requisição simples

O valor dessa arquitetura não está em substituir backends tradicionais, mas em reduzir fricção em contextos onde:

- a lógica de negócio é simples
- a inspeção humana dos dados é útil
- a escala é baixa
- o custo de manter infraestrutura é desproporcional

4. Exemplo mínimo — API simplificada (exemplo deliberadamente didático)

O trecho abaixo ilustra o mecanismo, não uma implementação de produção:

```
function doPost(e) {
  const API_TOKEN = 'MEU_TOKEN_SECRETO';

  if (e.parameter.token !== API_TOKEN) {
    return ContentService
      .createTextOutput(JSON.stringify({ error: 'Unauthorized' }))
      .setMimeType(ContentService.MimeType.JSON);
  }

  const sheet = SpreadsheetApp
    .openById('SPREADSHEET_ID')
    .getSheetByName('dados');
```

```
  const body = JSON.parse(e.postData.contents);
  sheet.appendRow([new Date(), body.nome, body.email]);

  return ContentService
    .createTextOutput(JSON.stringify({ status: 'ok' }))
    .setMimeType(ContentService.MimeType.JSON);
}
```

Esse exemplo existe para demonstrar:

- acesso HTTP
- validação mínima
- separação entre lógica e persistência
- viabilidade técnica da abordagem

5. Limitações reais — não-ACID na prática

A ausência de garantias ACID não é apenas um detalhe teórico. Na prática, isso pode resultar em:

- race conditions em escritas simultâneas
- inconsistência temporal em leituras
- bugs intermitentes difíceis de reproduzir

Esses riscos não invalidam a abordagem, mas delimitam claramente seu campo de aplicação. Em alguns casos, mecanismos como locks e validações adicionais podem mitigar parte do problema, mas não eliminam a natureza não transacional da solução.

6. Segurança, governança e fator humano (e o custo disso)

Outro aspecto crítico é organizacional. Soluções baseadas em planilhas permitem edição por usuários não técnicos, o que pode ser vantagem (workflow humano), mas também risco:

- alterações manuais indevidas
- exclusões acidentais
- quebra de esquema

- compartilhamento incorreto

Por isso, governança mínima é indispensável: permissões restritas, validação no backend e controle de acesso. E vale explicitar um ponto: governança também é custo — só que, em certos cenários, é um custo menor e mais direto do que operar um stack completo.

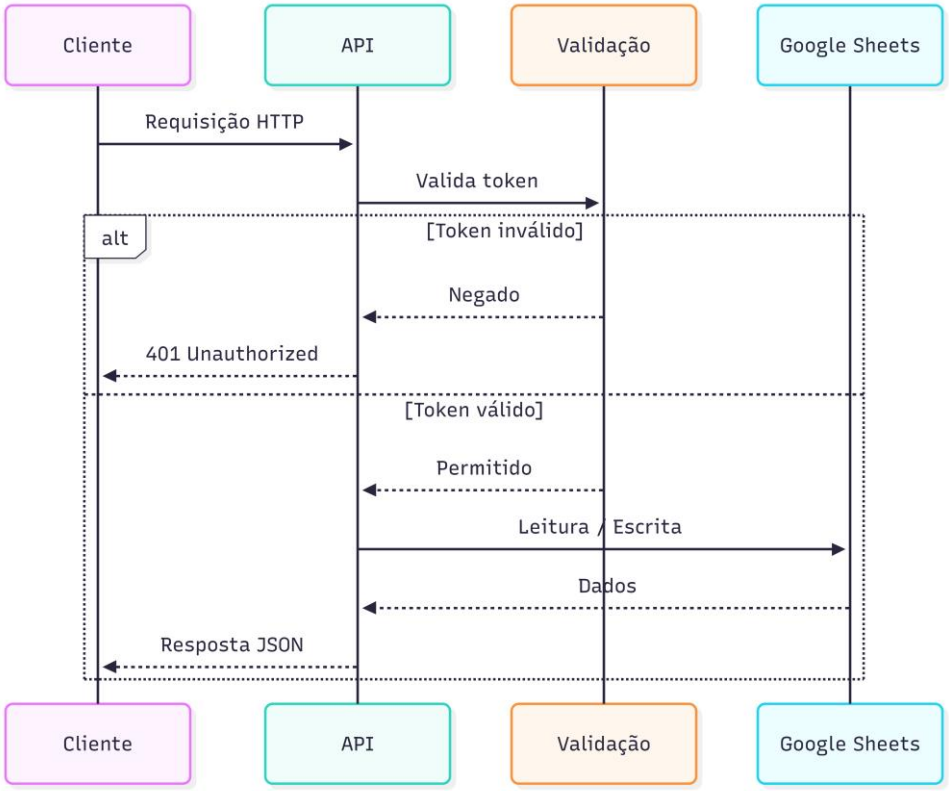

Figura 2 — Fluxo com validação

Simplicidade arquitetural não elimina responsabilidade técnica.

7. Lock-in e alternativas no mesmo espaço arquitetural

Essa abordagem cria dependência do ecossistema Google, com quotas e políticas próprias. Em alguns contextos, isso pode ser aceitável; em outros, não. É importante reconhecer alternativas no mesmo espaço de baixa fricção:

- Airtable
- Firebase
- Supabase (modo mínimo)

- Notion (com ressalvas)

O ponto central permanece: a escolha deve ser contextual, não dogmática.

8. Critérios objetivos de decisão

Faz sentido quando:

- há poucos usuários simultâneos
- existe baixa taxa de escrita
- os dados têm baixa criticidade
- há necessidade de inspeção humana
- o foco é agilidade e baixo custo operacional

Evite quando:

- existem transações complexas
- há concorrência elevada
- os dados são críticos ou sensíveis
- há necessidade de SLA rigoroso e controle fino

9. Conclusão — Fechamento

A tese se mantém: há sistemas que não precisam nascer com um banco de dados transacional tradicional. O que eles precisam é de persistência adequada ao problema, com clareza sobre limites, riscos e custos.

Antes de perguntar "qual banco de dados usar?", talvez a pergunta mais produtiva seja:

"Qual nível de persistência esse problema realmente exige agora?"

Arquitetura madura não é maximalista. Ela é contextual, consciente e evolutiva.