Explorando o PostgREST como Solução de Interoperabilidade para Sistemas Legados

Gustavo Klehm[1], **Evandro Kuszera**[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Dois Vizinhos – PR – Brazil

gustavoklehm014@gmail.com, evandrokuszera@utfpr.edu.br

Abstract. *The integration of heterogeneous systems constitutes a recurring challenge in corporate environments, especially when they involve legacy applications developed using discontinued technologies. This work evaluates the technical feasibility of PostgREST as an interoperability solution between a legacy ERP system built in Visual FoxPro 9 and a mobile application developed in React Native, both directly accessing a PostgreSQL database. The study identifies issues related to duplication of SQL queries and proposes centralizing data access through SQL views automatically exposed as RESTful APIs by PostgREST. Quantitative metrics demonstrate an average reduction of 68% in lines of code in VFP9 and 63% in JavaScript for a sample of five representative queries. The work contributes by demonstrating the challenges of modernizing legacy systems through database-driven APIs.*

Resumo. *A integração entre sistemas heterogêneos constitui desafio recorrente em ambientes corporativos, especialmente quando envolvem aplicações legadas desenvolvidas em tecnologias descontinuadas. Este trabalho avalia a viabilidade técnica do PostgREST como solução de interoperabilidade entre um sistema ERP legado em Visual FoxPro 9 e um aplicativo móvel em React Native, ambos acessando diretamente um banco PostgreSQL. O estudo identifica problemas de duplicação de consultas SQL e propõe centralização do acesso a dados através de views SQL expostas automaticamente como APIs RESTful pelo PostgREST. Métricas quantitativas demonstram redução média de 68% em linhas de código no VFP9 e 63% no JavaScript para amostra de cinco consultas representativas. O trabalho contribui demostrando os desafios de modernização de sistemas legados via APIs database-driven.*

1. Introdução

A interoperabilidade entre sistemas heterogêneos constitui desafio técnico e gerencial relevante em ambientes corporativos que combinam aplicações desenvolvidas em diferentes épocas, linguagens e paradigmas tecnológicos [Chen et al. 2008]. Quando múltiplas aplicações necessitam acessar os mesmos dados corporativos, a ausência de uma camada padronizada de integração frequentemente resulta em duplicação de lógica de acesso, inconsistências entre implementações e elevado custo de manutenção [Brooke and Paige 2001, Khadka et al. 2014].

As APIs REST consolidaram-se como abordagem predominante para integração entre sistemas devido à simplicidade arquitetural, uso de protocolos HTTP padronizados

e independência de plataforma [Fielding 2000]. Entretanto, o desenvolvimento manual de APIs RESTful demanda esforço significativo de implementação, manutenção e versionamento [Masse 2011]. Ferramentas de geração automática de APIs a partir de bancos de dados relacionais emergem como alternativa para reduzir esse esforço, particularmente em cenários onde a estrutura do banco de dados já está consolidada e normalizada.

Este trabalho investiga o PostgREST, ferramenta que gera automaticamente APIs RESTful completas a partir de esquemas PostgreSQL, expondo tabelas, *views* e funções como endpoints HTTP sem necessidade de código intermediário [PostgREST Contributors 2024]. O estudo foi motivado por um problema real em uma empresa brasileira de médio porte cujo sistema *Enterprise Resource Planning* (ERP) legado em Visual FoxPro 9 (VFP9) e aplicativo móvel em *React Native* acessam diretamente banco PostgreSQL, resultando em 4.013 consultas SQL apenas no módulo de Produtos. Como contribuição, este artigo avalia o uso do PostgREST como camada de interoperabilidade entre sistemas (novo e legado), a fim de padronizar e reduzir a duplicação de código de acesso a base de dados.

2. Fundamentação Teórica

Esta seção apresenta os conceitos fundamentais usados neste trabalho, cobrindo arquitetura REST, interoperabilidade de sistemas, modernização de aplicações legadas e geração automática de APIs.

2.1. Arquitetura REST e APIs RESTful

REST (*Representational State Transfer*) é um estilo arquitetural definido por Fielding [Fielding 2000], baseado em princípios como separação cliente-servidor, comunicação *stateless*, interface uniforme e uso de URIs e representações padronizadas. APIs RESTful aplicam esses princípios sobre HTTP, mapeando operações CRUD para verbos do protocolo e utilizando formatos como JSON [Masse 2011]. No modelo de maturidade de Richardson [Richardson and Ruby 2008], tais APIs variam do uso básico do HTTP (Nível 0) ao uso avançado de hipermídia (Nível 3). O PostgREST situa-se principalmente no Nível 2, oferecendo URIs baseadas em tabelas ou *views* e suporte aos métodos HTTP, ainda que sem HATEOAS (Hypermedia as the Engine of Application State) completo.

2.2. Interoperabilidade em Sistemas de Software

Interoperabilidade, conforme a ISO/IEC 25000 [ISO 2014], refere-se à capacidade de sistemas trocarem e utilizarem informações. Chen et al. [Chen et al. 2008] definem três níveis dessa capacidade: técnico (protocolos e redes), sintático (formato dos dados) e semântico (significado compartilhado). Em ambientes heterogêneos, esses desafios se intensificam, como apontam Jardim-Goncalves et al. [Jardim-Goncalves et al. 2013], reforçando a importância de mecanismos padronizados de integração. Abordagens como Service-Oriented Architecture (SOA) e APIs REST promovem acoplamento fraco e comunicação uniforme [Papazoglou and van den Heuvel 2007]. No contexto deste trabalho, o PostgREST cobre sobretudo os níveis técnico (HTTP/JSON) e sintático (esquema relacional), enquanto o nível semântico depende do desenho das *views* SQL.

2.3. Geração Automática de APIs a partir de Bancos de Dados

No desenvolvimento *database-driven*, o banco de dados funciona como fonte primária de verdade, orientando a construção das aplicações a partir de seu esquema [Ambler 2003]. Ferramentas de geração automática de APIs adotam a abordagem *schema-first*, criando interfaces diretamente a partir da estrutura do banco [PostgREST Contributors 2024]. Entre essas soluções estão PostgREST, Hasura, Supabase e Prisma, que diferem quanto ao uso de REST, GraphQL ou Object-Relational Mapping (ORM). O PostgREST se destaca pela leveza arquitetural, aderência aos princípios REST e integração nativa com mecanismos de segurança do PostgreSQL, como RLS. Estudos como o de Scheible et al. [Scheible et al. 2024] demonstram sua aplicabilidade prática, embora apontem limitações em customização e autenticação avançada — aspectos também identificados neste trabalho.

3. Trabalhos Relacionados

Scariott [Scariott 2025] compara PostgREST, Directus e NocoDB por meio de benchmarks focados em desempenho, escalabilidade e compatibilidade com bancos normalizados, indicando que o PostgREST apresenta menor latência em consultas diretas (15 ms contra 28 ms do Directus), embora ofereça menos flexibilidade para transformações complexas. Enquanto o estudo conclui que o PostgREST é mais adequado para cenários CRUD simples, com o Directus se destacando pela interface administrativa, este trabalho avança além de um ambiente comparativo controlado ao investigar a viabilidade do PostgREST em um cenário real de integração com sistemas legados.

Štefancová [Štefancová 2018] avalia o uso do PostgREST para expor grandes volumes de dados de séries temporais no CERN, demonstrando escalabilidade ao servir 2,4 bilhões de registros via API. O estudo aponta desafios como configuração de *Row-Level Security* (RLS) e ausência de mecanismos nativos de *rate limiting*.

Scheible et al. [Scheible et al. 2024] apresentam um *data provider* para React-Admin baseado em PostgREST, demonstrando compatibilidade entre as operações da biblioteca (`getList`, `getOne`, etc.) e os endpoints da ferramenta, embora relatem limitações relacionadas à autenticação JWT e à ausência de versionamento nativo. Diferentemente desse enfoque voltado à integração com interfaces administrativas, este trabalho analisa o uso do PostgREST para interoperabilidade entre sistemas legados desktop (Visual FoxPro 9) e aplicações mobile (React Native), um contexto ainda não explorado pelas pesquisas anteriores.

4. Usando PostgREST para Interoperabilidade entre Sistemas

Esta seção apresenta o estudo de caso na qual o PostgREST é utilizado como camada de interoperabilidade, centralizando acesso a dados corporativos e disponibilizando-os via API RESTful padronizada.

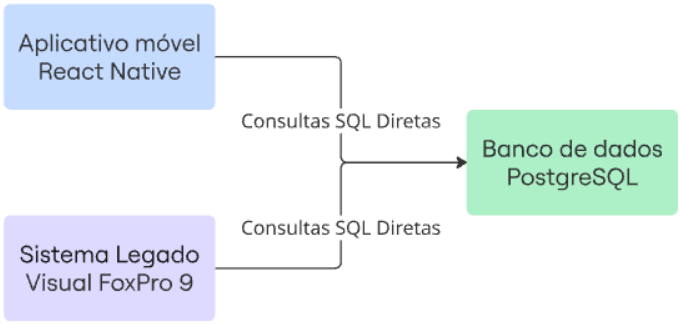
4.1. Cenário do Estudo de Caso

O cenário selecionado considera um ERP em operação há 27 anos, um banco PostgreSQL com 515 tabelas, a convivência entre um sistema desktop legado (VFP9) e um aplicativo mobile moderno (React Native), além de significativa duplicação de consultas SQL — 4.013 em VFP9 e 203 e React. Dentre os diversos módulos do ERP, foi selecionado o

módulo de Produtos, por concentrar o maior volume de regras de negócio e participação de quase todos os fluxos críticos do sistema. Esse módulo tem 8 tabelas principais, com 85 colunas na tabela principal (`PCCDITE0`).

A Figura 1 apresenta arquitetura vigente, caracterizada por múltiplos pontos de acesso direto ao banco e duplicação de lógica entre plataformas.

Figura 1. Arquitetura atual: aplicações acessam o banco diretamente, cada uma mantendo sua própria lógica de negócio e consultas SQL.

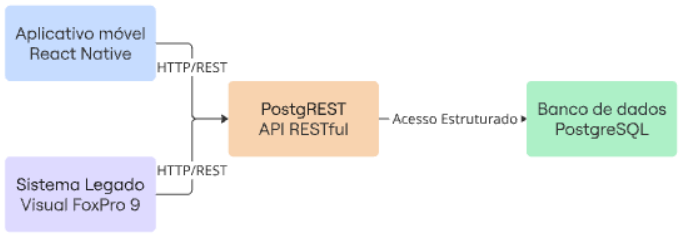


Fonte: Elaborado pelo autor (2025).

4.2. Arquitetura Proposta

A Figura 2 ilustra a arquitetura proposta, na qual o PostgREST atua como intermediário, consolidando consultas em *views* SQL e expondo-as via *endpoints* RESTful.

Figura 2. Arquitetura proposta: PostgREST como camada central de interoperabilidade entre aplicações e banco de dados.



Fonte: Elaborado pelo autor (2025).

4.3. Metodologia do Estudo de Caso

O estudo de caso foi executado em etapas, sendo:

Etapa 1 — Migração dos Dados para Ambiente de Testes: Criou-se uma réplica local do banco de produção via backup/restore para uso no experimento, validando a integridade das consultas principais.

Etapa 2 — Desenvolvimento de Views SQL: O objetivo foi centralizar consultas repetidas em *views*, mantendo compatibilidade funcional com consultas originais. Um subconjunto de consultas foi selecionado para o estudo de caso, seguindo critérios de frequência de uso, representatividade e duplicação entre as duas aplicações. Nessa etapa, cinco consultas foram selecionadas e convertidas para *views*.

Etapa 3 — Configuração do PostgREST: todas as views criadas foram inseridas em uma esquema do banco de dados chamado *api*. O PostgREST foi configurado para expor o esquema `api` via porta 4444. O mecanismo de autenticação foi desabilitado no PostgREST por estar fora do escopo do trabalho.

Etapa 4 — Execução de Testes e Coleta de Métricas Quantitativas: Aplicaram-se duas métricas principais: **4.1) Linhas de Código (LOC)** - comparação entre trechos VFP9/JS originais e suas versões equivalentes utilizando fetch() para chamar o endpoint PostgREST. Calculou-se média e redução percentual do número de linhas antes de depois de usar PostgREST. Para calcular LOC foram consideradas apenas as linhas que continham instruções efetivas, desconsiderando completamente linhas em branco ou compostas apenas por comentários. A partir dessa medição bruta, foi obtida a média entre os trechos analisados e calculada a redução percentual em relação às versões equivalentes que utilizam fetch() para acessar o endpoint do PostgREST. **4.2) Pontos de Acesso** - comparação entre número total de consultas nos códigos analisados e a quantidade de *views* necessárias no PostgREST.

Etapa 5 — Análise Qualitativa: Avaliou-se manutenibilidade com base em critérios da literatura (Pressman e Maxim, 2016), como localização da lógica, risco de divergência e custo cognitivo. A comparação antes/depois foi sintetizada em tabela.

5. Implementação do Estudo de Caso

Esta seção detalha processo de conversão de consultas SQL dispersas para *views* centralizadas expostas via PostgREST.

A primeira etapa é localização da consulta original em ambas aplicações. Após, essa consulta é implementada como uma *view* no banco de dados, armazenada no esquema *api*. Testes manuais são realizados para verificar a equivalência de dados retornados pela *view* criada. Abaixo um exemplo de *view* implementada no estudo de caso.

```
CREATE OR REPLACE VIEW api.produtos_similares AS
SELECT
  p.codigo,
  p.similarm AS similar_mestre,
  p.descricao
FROM pccdite0 p
WHERE (
    -- Caso 1: produto aponta para mestre diferente
    (p.codigo <> p.similarm
     AND NOT EXISTS (
         SELECT 1 FROM pccdite0 p2
         WHERE p2.codigo = p.similarm
           AND p2.similarm = p.similarm
     ))
    OR
    -- Caso 2: produto mestre mas tem aplicacao distinta
    (p.codigo = p.similarm
     AND p.produtoaplicacao IS DISTINCT FROM p.codigo
     AND NOT EXISTS (
         SELECT 1 FROM pccdite0 p3
         WHERE p3.similarm = p.codigo
     ))
);
```

Listing 1. View de produtos similares

O PostgREST é configurado para expor automaticamente as *views* armazenadas no esquema *api*. Um *endpoint* com o mesmo nome da *view* é gerado automaticamente.

```
GET http://localhost:4444/produtos_similares?codigo=eq.001234
```

Resposta JSON:

```
[
  {
    "codigo": "001234",
    "similar_mestre": "BRONZIA",
    "descricao": "BRONZIA DE BIELA STD"
  },
  ...
]
```

A próxima etapa consiste em modificar as chamadas para as consultas em ambas as aplicações (Visual FoxPro 9 e React), para usar os *endpoints* expostos pelo PostgREST. As Figuras4 e 3 mostram trechos de códigos introduzidos nas aplicações no lugar das chamadas usando diretamente as consultas SQL.

Figura 3. Código JavaScript chamando o endpoint PostgREST.

```
import fetch from "node-fetch";

fetch("http://localhost:4444/vw_produtos_similares")
  .then(res => res.json())
  .then(data => console.log(data));
```

Figura 4. Código VFP9 consultando endpoint PostgREST.

```
loHttp = CREATEOBJECT("WinHttp.WinHttpRequest.5.1")
loHttp.Open("GET", "Http://localhost:4444/vw_produtos_similares", .F.)
loHttp.Send()

lcJson = loHttp.ResponseText
? lcJson
```

Por fim, as etapas descritas acima são repetidas para as demais consultas selecionadas no estudo de caso, não sendo repetidas aqui por limitação de espaço.

6. Resultados e Discussão

Esta seção apresenta resultados quantitativos e qualitativos obtidos após a implementação do PostgREST como camada de interoperabilidade.

6.1. Redução de Pontos de Acesso

A Tabela 1 apresenta a análise de duplicação das consultas SQL do módulo Produtos, comparando o código legado em Visual FoxPro 9 com a camada mobile em React Native, antes do uso do PostgREST. O ERP em VFP9 tem 528 arquivos `.prg` contendo 4.013 consultas SQL ao módulo Produtos. A versão em React Native 3 arquivos `.js` contendo 203 consultas SQL ao mesmo módulo. Isso totaliza 4.216 pontos de acesso dispersos. Para estimar quantas consultas são realmente distintas, analisou-se uma amostra de 10% das instruções presentes em cada tecnologia e extrapolou-se o resultado para o total identificado.

Os dados mostram que as 4.216 consultas mapeadas correspondem a aproximadamente 138 consultas diferentes, resultando em uma taxa média de duplicação de 31:1. Esse nível de repetição indica forte espalhamento de lógica SQL pelo código, aumentando

Tabela 1. Análise de duplicação de consultas SQL no módulo Produtos

Sistema	Arquivos	Consultas Totais	Estimativa de Distintas*
Visual FoxPro 9 (Sistema Legado)	528	4.013	˜122
React Native	3	203	˜16
Total	531	4.216	˜138
* Baseado em análise de amostra de 10% das consultas			

Fonte: Dados extraídos do código-fonte dos sistemas, elaborados pelo autor (2025).

o risco de inconsistências e elevando o custo de manutenção, já que a alteração de uma regra pode exigir modificações em dezenas de locais distintos.

A partir dessa evidência, a Tabela 1 reforça a necessidade de centralizar as consultas em views e expô-las via PostgREST, reduzindo redundâncias e padronizando o acesso aos dados. Com o uso do PostgREST como camada de interoperabilidade, cada consulta distinta torna-se uma *view* SQL. Dessa forma, são necessárias apenas 138 *views*, uma *view* por consulta distinta. Como resultado, do total de 4.216 pontos de acesso foram criados 138 *endpoints* (um por *view*), o que representa uma redução de **96,7%** em termos de pontos de acesso.

6.2. Redução de Linhas de Código

A Tabela 2 apresenta a comparação de linhas de código entre as consultas originais (VFP9 e JavaScript) e suas versões reescritas utilizando views SQL e chamadas via fetch() para o PostgREST. A análise foi feita a partir de uma amostra de cinco consultas representativas, selecionadas por cobrirem padrões comuns do módulo.

Tabela 2. Comparação de linhas de código por consulta (amostra n=5). Legenda: Q1-Q5 = consultas selecionadas; LOC = Lines of Code; DP = Desvio Padrão; Red. = Redução percentual.

ID	LOC VFP9	LOC JS	LOC View SQL	LOC Fetch	Red. VFP9	Red. JS
Q1	12	10	10	4	67%	60%
Q2	15	14	13	5	67%	64%
Q3	18	14	10	5	72%	64%
Q4	9	8	5	3	67%	62%
Q5	13	11	7	4	69%	64%
Média	**13,4**	**11,4**	**9,0**	**4,2**	**68%**	**63%**
DP	**3,4**	**2,7**	**3,2**	**0,8**	**2,1%**	**1,8%**

Fonte: Dados extraídos do código-fonte dos sistemas, elaborados pelo autor (2025).

Os resultados mostram reduções médias de 68% no VFP9 e 63% no JavaScript, com baixo desvio-padrão, indicando consistência entre os casos analisados. Embora parte da lógica permaneça nas *views* SQL (média de 9 LOC), essa complexidade deixa de estar espalhada pelo código e passa a ser centralizada no banco. Por ser uma amostra pequena, os valores devem ser interpretados como indicativos, mas já evidenciam o impacto estrutural da substituição de consultas inline por endpoints RESTful expostos pelo PostgREST.

Limitação reconhecida: Amostra reduzida (n=5) impede inferência estatística robusta. Testes de significância (t-test) não foram aplicados devido a tamanho amostral insuficiente. Resultados devem ser interpretados como indicativos, não conclusivos.

6.3. Análise Qualitativa de Manutenibilidade

Comparação estruturada de características de manutenção baseada em critérios da ISO/IEC 25010 [ISO 2011] e Pressman [Pressman and Maxim 2016]:

Tabela 3. Análise comparativa qualitativa de manutenibilidade

Critério	Antes	Com PostgREST	Análise
Localização da Regra	Dispersa em 531 arquivos (VFP9 + JS)	Centralizada em view SQL única	Redução de esforço de busca; mudança propagada automaticamente via API
Propagação de Mudanças	Alteração manual em N pontos (até 31 por consulta)	Alteração única na view; clientes inalterados	Alinhado com princípio DRY (*Don't Repeat Yourself*) [Hunt and Thomas 1999]
Testes de Regressão	Necessários em VFP9 E JavaScript	Concentrados no banco (testes de view)	Redução de superfície de teste
Documentação	Código em múltiplos locais, sem padrão	API autodocumentada (OpenAPI)	PostgREST gera Swagger automaticamente
Curva de Aprendizado	Conhecimento em VFP9 + SQL + JS	SQL padrão + HTTP básico	Tecnologias mais universais
Risco de Divergência	Alto (implementações independentes)	Eliminado (fonte única de verdade)	Consistência garantida por design

Fonte: Elaborado pelo autor (2025).

Discussão: Centralização alinha-se com princípios de engenharia de software estabelecidos, particularmente DRY (*Don't Repeat Yourself*) [Hunt and Thomas 1999] e *Single Source of Truth*. Contudo, desloca complexidade para camada SQL, exigindo expertise em PostgreSQL (*views, functions, RLS*).

6.4. Limitações Identificadas

A análise do uso do PostgREST evidenciou algumas limitações relevantes. Em termos de autenticação, embora a ferramenta suporte JWT, sua configuração depende de serviços externos para emissão de tokens e de uma infraestrutura de gerenciamento de chaves, o que pode elevar a complexidade em ambientes sem expertise em segurança.

Quanto à lógica de negócio, o PostgREST é adequado para exposições diretas de dados, mas operações mais complexas — como integrações externas, processamento intensivo ou tarefas assíncronas — precisam ser deslocadas para stored procedures ou para uma camada de serviços intermediária, introduzindo trade-offs entre simplicidade e acoplamento.

Também se observa a ausência de mecanismos nativos de versionamento de API: alterações no esquema do banco podem quebrar clientes imediatamente, exigindo estratégias complementares como schemas paralelos ou views de compatibilidade.

Por fim, a integração com sistemas desktop legados, como Visual FoxPro 9, apresenta desafios adicionais devido à ausência de suporte nativo a HTTP e JSON, demandando componentes externos e maior esforço de implementação. Embora viável, essa integração tende a ser mais trabalhosa e limita o alcance imediato da solução em ambientes com legados profundos.

7. Conclusão

Este trabalho investigou a viabilidade técnica do PostgREST como solução de interoperabilidade entre sistemas legados e modernos, aplicado a empresa brasileira de médio porte.

O estudo foi motivado por problema de duplicação massiva de consultas SQL (4.013 no ERP Visual FoxPro 9 e 203 no aplicativo React Native) resultante de acesso direto ao banco de dados por múltiplas aplicações heterogêneas.

Os resultados demonstram que PostgREST é tecnicamente viável para centralizar acesso a dados corporativos via APIs RESTful geradas automaticamente a partir de *views* PostgreSQL. A consolidação das consultas dispersas no código para views SQL reduziu em aproximadamente 96,7% o número de pontos de acesso ao banco de dados, passando de 4.216 consultas encontradas no código para cerca de 138 *views* distintas. Além disso, na análise detalhada de uma amostra de cinco consultas, observou-se redução média de 68% nas linhas de código em VFP9 e 63% nas versões equivalentes em JavaScript.

Ademais, a centralização das consultas ao banco elimina divergências entre implementações e reduz o risco de comportamentos inconsistentes entre sistemas legados e aplicativos modernos. A manutenção também torna-se mais direta, uma vez que alterações nas regras de negócio ficam concentradas em *views* SQL. Além disso, a API gerada passa automaticamente a contar com documentação em formato OpenAPI, facilitando inspeção, integração e governança técnica.

As limitações identificadas, contudo, mostram que a adoção do PostgREST exige algumas considerações adicionais. O uso de autenticação JWT depende de infraestrutura externa, como Auth0 ou API Gateways. Consultas mais complexas podem demandar *stored procedures* ou uma camada intermediária para encapsular regras que excedem o escopo das *views*. O versionamento de APIs não é suportado nativamente, sendo necessário recorrer a esquemas paralelos. Por fim, a integração com Visual FoxPro 9 apresenta maior verbosidade devido ao tratamento manual de JSON e ao uso de componentes COM para requisições HTTP.

Como trabalho futuro pretende-se realizar validação em ambiente de produção, com monitoramento contínuo de desempenho, execução de testes de carga e avaliação longitudinal da manutenibilidade. Outra frente relevante consiste em estudos comparativos entre PostgREST e ferramentas como Hasura e Supabase, considerando produtividade, custo operacional e curva de aprendizado — aspectos que podem revelar critérios mais objetivos para adoção em diferentes contextos. Também pretende-se aplicar a abordagem a outros módulos do ERP e replicando o estudo em diferentes organizações para avaliar sua generalização. Além disso, futuras pesquisas podem avançar em temas técnicos como automação da conversão de consultas para views, definição de critérios formais para seleção de ferramentas database-to-API e desenvolvimento de estratégias mais robustas de versionamento da API.

Referências

(2011). Iso/iec 25010: Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models.

(2014). Iso/iec 25000: Systems and software engineering – systems and software quality requirements and evaluation (square) – guide to square.

Ambler, S. W. (2003). *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley.

Brooke, C. and Paige, R. F. (2001). Organisational scenarios and legacy systems. *International Journal of Information Management*, 21(5):365–384.

Chen, D., Doumeingts, G., and Vernadat, F. (2008). Architectures for enterprise integration and interoperability: Past, present and future. *Computers in Industry*, 59(7):647–659.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley.

Jardim-Goncalves, R., Romero, D., and Grilo, A. (2013). Systematisation of interoperability body of knowledge: The foundation for enterprise interoperability as a science. *Enterprise Information Systems*, 7(1):7–32.

Khadka, R., Saeidi, A., Jansen, S., and Hage, J. (2014). How do professionals perceive legacy systems and software modernization? *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 36–47.

Masse, M. (2011). *REST API Design Rulebook*. O'Reilly Media.

Papazoglou, M. P. and van den Heuvel, W.-J. (2007). *Service Oriented Architectures: Approaches, Technologies and Research Issues*. Springer.

PostgREST Contributors (2024). Postgrest documentation. Available at: `https://postgrest.org` (accessed 2025-11-23).

Pressman, R. S. and Maxim, B. R. (2016). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 8 edition.

Richardson, L. and Ruby, S. (2008). Restful web services. O'Reilly Media.

Scariott, F. (2025). Comparative evaluation of automatic api generation tools: Postgrest, directus and nocodb. In *Anais do Simpósio Brasileiro de Sistemas de Informação (SBSI)*. To appear.

Scheible, J. et al. (2024). Postgrest data provider for react-admin: Bootstrap the development of database driven admin frontends. *SoftwareX*, 25:100699.

Štefancová, V. (2018). Using postgrest for time series data access at cern. In *Proceedings of CHEP 2018*.