UNIVERSIDADE FEDERAL DE ALFENAS

EDUARDO GUERREIRO ROCHA

REENGENHARIA DE UM SISTEMA *WEB* EDUCACIONAL: MIGRAÇÃO PARA ARQUITETURA *REST* COM PADRÃO *REPOSITORY* E *NODE.JS*

ALFENAS/MG

2025

EDUARDO GUERREIRO ROCHA

REENGENHARIA DE UM SISTEMA *WEB* EDUCACIONAL: MIGRAÇÃO PARA ARQUITETURA *REST* COM PADRÃO *REPOSITORY* E *NODE.JS*

Trabalho de Conclusão de Curso apresentado(a) como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal de Alfenas.

Orientador: Prof. Dr. Luiz Eduardo da Silva

ALFENAS/MG

2025

Sistema de Bibliotecas da Universidade Federal de Alfenas
Biblioteca Unidade Educacional Santa Clara

Rocha, Eduardo Guerreiro .
Reengenharia de um sistema web educacional : Migração para arquitetura REST com padrão Repository e Node.js / Eduardo Guerreiro Rocha. - Alfenas, MG, 2025.
30 f. : il. -

Orientador(a): Luiz Eduardo da Silva.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Federal de Alfenas, Alfenas, MG, 2025.
Bibliografia.

1. Reengenharia. 2. Sistemas web. 3. API REST. 4. Padrão Repository. 5. Node.js. I. Silva, Luiz Eduardo da, orient. II. Título.

Ficha gerada automaticamente com dados fornecidos pelo autor.

EDUARDO GUERREIRO ROCHA

REENGENHARIA DE UM SISTEMA *WEB* EDUCACIONAL: MIGRAÇÃO PARA ARQUITETURA *REST* COM PADRÃO *REPOSITORY* E *NODE.JS*

O Presidente da banca examinadora abaixo assina a aprovação do Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal de Alfenas.

Aprovada em: 01 de dezembro de 2025

Prof. Dr. Luiz Eduardo da Silva
Universidade Federal de Alfenas

Assinatura:

gov.br
Documento assinado digitalmente
LUIZ EDUARDO DA SILVA
Data: 16/12/2025 18:33:04-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Fellipe Guilherme Rey de Souza
Universidade Federal de Alfenas

Assinatura:

Prof.ª Dr.ª Mariane Moreira de Souza
Universidade Federal de Alfenas

Assinatura:

RESUMO

A evolução das ferramentas educacionais é essencial para acompanhar as mudanças no ambiente de aprendizado. A plataforma *Tales Ludos*, que se destaca como um recurso educacional aberto para criação de jogos, enfrentou o desafio de modernizar sua infraestrutura tecnológica a fim de melhorar a sinergia entre suas frentes de desenvolvimento. Este artigo relata a transição estratégica e a refatoração do *backend* da aplicação, migrando de um ambiente PHP para *Node.js* sob o modelo de *API REST* e padrão *Repository*, com o intuito de otimizar a comunicação entre camadas. A metodologia compreendeu inicialmente a análise do sistema original e o planejamento da reengenharia em três eixos: a adoção de uma *API REST*, a implementação do padrão *Repository* com injeção de dependências e a migração tecnológica para *Node.js*. Em seguida, realizou-se a implementação da nova arquitetura em camadas e sua documentação. Os resultados demonstram que a nova arquitetura tornou o sistema mais modular, flexível e preparado para integração com múltiplos clientes. Embora tenha aumentado a complexidade conceitual, a refatoração facilitou a manutenção evolutiva e os testes automatizados. Conclui-se que a adoção dessas práticas foi crucial para aumentar a sustentabilidade técnica da plataforma, alinhando-a às boas práticas contemporâneas.

Palavras-chave: Reengenharia; Sistemas *Web*; *API REST*; Padrão *Repository*; *Node.js*.

ABSTRACT

The evolution of educational tools is essential to keep pace with changes in learning environments. The *Tales Ludos* platform, an open educational resource for game creation, faced the challenge of modernizing its technological infrastructure in order to improve synergy between its development fronts. This article reports the strategic transition and refactoring of the application *backend*, migrating from a PHP environment to *Node.js* under a *REST API* model and the *Repository* pattern, with the goal of optimizing communication between layers. The methodology initially comprised an analysis of the original system and the planning of the re-engineering along three main axes: the adoption of a *REST API*, the implementation of the *Repository* pattern with dependency injection, and the technological migration to *Node.js*. Next, the new layered architecture was implemented and documented. The results show that the new architecture made the system more modular, flexible and prepared for integration with multiple clients. Although it increased conceptual complexity, the refactoring facilitated evolutionary maintenance and automated testing. It is concluded that the adoption of these practices was crucial to strengthen the technical sustainability of the platform and align it with contemporary best practices.

Keywords: Software Re-engineering; Web Systems; REST API; Repository Pattern; Node.js

SUMÁRIO

1 INTRODUÇÃO

A crescente integração das Tecnologias de Informação e Comunicação (TICs) na sociedade contemporânea tem impulsionado transformações significativas em diversos setores, incluindo a educação (Klein, 2020). No Brasil a pandemia de COVID-19 acelerou a adoção de tecnologias educacionais, com muitas instituições implementando rapidamente soluções digitais para garantir a continuidade do ensino e da aprendizagem em ambientes virtuais. Ferramentas como plataformas de aprendizagem *online*, sistemas de gestão de conteúdo, aplicativos educacionais e recursos digitais interativos foram adotados para responder à crise sanitária, evidenciando o potencial das TICs para impulsionar a inovação e a adaptação no campo educacional (Veloso B. e Mill, 2024).

Um dos campos que mais se destaca na intersecção entre tecnologia e educação é a gamificação. Ao integrar elementos lúdicos em ambientes de aprendizagem, a gamificação busca estimular o engajamento e a motivação dos alunos, tornando o processo educativo mais eficaz e prazeroso (Deterding, 2011). Parte do destaque se deve ao desenvolvimento de diversas ferramentas que aplicam com êxito essa estratégia, como o Duolingo, que utiliza desafios, recompensas e progressão em níveis para promover o aprendizado de idiomas de forma divertida (Duolingo, 2024). Outras plataformas, como o *Khan Academy*, incentivam o progresso dos alunos por meio de pontos, *rankings* e emblemas que reconhecem esforços e conquistas (Academy, 2024). Até mesmo jogos comerciais foram adaptados a esse contexto, como *Minecraft*, que, em sua versão Educativa, cria ambientes de aprendizagem imersivos e colaborativos dentro do próprio jogo (Short, 2012).

Nesse cenário, surge a plataforma Tales Ludos (CAPES e UNIFAL,), um sistema online e gratuito projetado para a criação de jogos educacionais. Alinhada ao conceito de Recursos Educacionais Abertos (REA), a plataforma democratiza o acesso à criação de jogos, permitindo que educadores e alunos, mesmo sem conhecimentos técnicos em programação, desenvolvam e compartilhem seus próprios jogos de forma colaborativa. Essa abordagem permite a criação de jogos personalizados para diferentes áreas do conhecimento e níveis de ensino, fomentando a cultura do compartilhamento e da colaboração em prol do desenvolvimento educacional, conforme aprofundado no manual teórico-metodológico publicado (Silva, 2025). Além disso, a plataforma oferece um repositório de jogos criados pela comunidade, incentivando o reuso e a adaptação de materiais, em consonância com os princípios dos REA (UNESCO, 2012).

Entretanto, o crescimento contínuo da plataforma e a demanda por novas funcionalidades evidenciam limitações na arquitetura adotada. No atual modelo, todas as camadas do sistema (interface, lógica de negócio e persistência de dados) são fortemente acopladas, o que dificulta a evolução e manutenção do código (Fowler, 2018). Sendo assim, a correção de erros, adição de novos módulos e integração de tecnologias mais recentes tornam-se processos lentos e suscetíveis a falhas, reduzindo a escalabilidade e a sustentabilidade da aplicação (Rajlich, 2005). A longo prazo, essa arquitetura pode limitar a capacidade da plataforma de se adaptar às novas tecnologias e demandas dos usuários.

Para enfrentar esses desafios, este trabalho propõe a aplicação de um processo de refatoração, prática amplamente reconhecida na engenharia de *software* que visa melhorar a estrutura interna do código sem alterar seu comportamento externo, tornando-o mais compreensível, flexível e fácil de manter (Fowler, 2018). A refatoração se encontra dentro do contexto da manutenção evolutiva, na qual o software é modificado de forma planejada para se adaptar a novas necessidades, melhorar desempenho e garantir sua longevidade (Rajlich, 2005).

Especificamente, a proposta consiste em refatorar o backend da aplicação para um modelo de *API REST* (IBM, 2025) desenvolvida em *Node.js* (Foundation, 2025). Essa abordagem permite que o sistema seja reorganizado em camadas desacopladas, nas quais o *frontend* em *Vue.js* (Vue.js, 2025) consome serviços expostos via *HTTP*(*Hypertext Transfer Protocol*) de maneira padronizada. O estilo arquitetural *REST* (*Representational State Transfer*) permite uma comunicação leve, escalável e interoperável entre cliente e servidor, favorecendo a manutenção, o reuso e a integração com outras aplicações (Meshram, 2021). Dessa forma, a refatoração representa não apenas uma mudança tecnológica, mas uma evolução estrutural que torna o Tales Ludos mais escalável, manutenível e alinhado às boas práticas contemporâneas de desenvolvimento de *software*.

O trabalho está organizado da seguinte forma: na próxima seção, apresentamos o referencial teórico utilizado nesta pesquisa; na terceira seção, descrevemos a metodologia aplicada; na quarta seção, detalhamos a análise da arquitetura original e o processo de refatoração; e, por fim, são discutidos os resultados e conclusões obtidos.

2 REVISÃO BIBLIOGRÁFICA

2.1 REENGENHARIA E REFATORAÇÃO

A reengenharia de *software* é definida como o processo de examinar e transformar um sistema existente para aprimorar sua estrutura e desempenho, mantendo sua funcionalidade original (Chikofsky; Cross, 1990). A refatoração, descrita por (Fowler, 2018), representa uma etapa dentro desse processo, voltada especificamente à melhoria da estrutura interna do código sem alterar seu comportamento externo. Ambas as práticas visam facilitar a manutenção, reduzir a complexidade e facilitar futuras evoluções do sistema, sendo essenciais em contextos de modernização tecnológica.

2.2 ARQUITETURA CLIENTE-SERVIDOR COM *APIS REST*

A arquitetura cliente-servidor estabelece a separação entre a interface do usuário (cliente) e o processamento de dados (servidor), permitindo o desenvolvimento modular e a comunicação por meio de protocolos padronizados (Tanenbaum; Steen, 2017). Nesse contexto, a arquitetura *REST*, proposta por Fielding (Pires, 2015), consolidou-se como um padrão para o desenvolvimento de *APIs* (*Application Programming Interfaces*) voltadas à comunicação entre sistemas distribuídos. O *REST* define princípios baseados em recursos identificados por *URL's*, operações padronizadas (como *GET*, *POST*, *PUT* e *DELETE*) e troca de dados em formato leve, geralmente *JSON*(*Javascript Object Notation*) (Bray, 2017).

2.3 PADRÃO EM CAMADAS E O MODELO *REPOSITORY*

O padrão em camadas organiza o *software* em níveis hierárquicos de responsabilidade, permitindo isolamento entre lógica de negócio, persistência de dados e interface de usuário. Essa abordagem facilita o reuso, os testes e a manutenção do código, além de contribuir para a clareza arquitetural (Bass; Clements; Kazman, 2021). Dentro desse paradigma, o *Repository Pattern* atua como uma camada intermediária entre a aplicação e o mecanismo de persistência, abstraindo o acesso aos dados e centralizando as operações de leitura e escrita. Esse padrão reduz o acoplamento entre as entidades de domínio e o banco de dados, promovendo uma arquitetura mais coesa, modular e testável (Microsoft, 2025a).

3 METODOLOGIA

Esta seção apresenta, em linhas gerais, o caminho metodológico adotado para a reengenharia do Tales Ludos. A pesquisa caracteriza-se como um estudo de caso de natureza aplicada, em que um sistema web educacional existente foi tomado como objeto para experimentação de uma nova arquitetura baseada em *API REST*, padrão *Repository* e *Node.js*.

O processo foi organizado em quatro etapas principais: análise do sistema original, planejamento da refatoração, implementação da nova arquitetura e documentação das decisões técnicas. Na etapa de análise, buscou-se identificar limitações estruturais e de acoplamento no sistema monolítico. Na etapa de planejamento, foram definidos os eixos de reengenharia e as tecnologias que responderiam às restrições encontradas. Em seguida, a implementação colocou em prática essas decisões, com a criação da *API REST*, a introdução do padrão *Repository* e a migração para *Node.js*. Por fim, a documentação consolidou diagramas, contratos de serviço e registros técnicos, com o objetivo de facilitar futuras manutenções e reduzir a curva de aprendizado de novos desenvolvedores.

Essas etapas têm como propósito registrar decisões, dificuldades e impactos observados durante a transformação arquitetural, de modo a oferecer um material de referência para evoluções posteriores do sistema. A Seção 4 detalha o processo de reengenharia aplicado ao Tales Ludos, descrevendo o que foi efetivamente realizado em cada uma dessas etapas.

4 REENGENHARIA DO TALES LUDOS

4.1 ANÁLISE DO SISTEMA ORIGINAL

O sistema foi originalmente desenvolvido como uma aplicação monolítica, mantendo *frontend* e *backend* no mesmo repositório, compartilhando recursos, lógica e estruturas internas. Essa arquitetura foi muito eficiente no início, devido à sua fácil implementação em aplicações de menor escala. No entanto, os recentes planos de expansão funcional e interoperabilidade com novas interfaces, como versões *mobile* e para *tablet* educativo, evidenciaram limitações causadas por esse forte acoplamento.

A análise exploratória do código apontou uma sobrecarga na camada de controle, que concentrava responsabilidades distintas: definição da lógica de negócios, validação de informações e acesso direto ao banco de dados. Esse acúmulo, além de violar princípios clássicos da Engenharia de *Software* como o *Single Responsibility Principle* (*SRP*) e o *Dependency Inversion Principle* (*DIP*) (Martin, 2017), reduzia na prática o potencial evolutivo da plataforma. Concretamente, a ausência de mecanismos de injeção de dependência tornava a substituição e extensão de componentes internos mais complexa, a lógica de domínio, intrinsecamente dependente do fluxo definido pelos controladores, comprometia a reutilização de código, e, por fim, a mistura das regras de negócio com o acesso aos dados dificultava a criação de testes unitários e a utilização de *mocks* (massa de dados usados em testes).

Assim, foi constatada a necessidade de uma abordagem arquitetural menos acoplada, capaz de promover a separação sistemática entre as camadas, delimitar as responsabilidades e proporcionar um ecossistema adequado para testes, escalabilidade horizontal e integração futura com múltiplas interfaces e dispositivos.

4.2 PLANEJAMENTO DA REFATORAÇÃO

A partir da análise conduzida, foram definidos três eixos estratégicos de reengenharia: adoção de uma *API REST*, implementação do padrão *Repository* com injeção de dependências e migração do *backend* para *Node.js*. Cada escolha respondeu diretamente às limitações diagnosticadas e teve expectativas claras associadas à sua implantação.

4.2.1 Servidor independente usando *API REST*

A presença de um monólito com forte acoplamento entre *backend*, *frontend* e camada de armazenamento impunha barreiras à escalabilidade e restringia a interoperabilidade. Assim, optou-se pela transformação do *backend* em uma *API REST*, promovendo comunicação via *HTTP* padronizada, com dados trafegando em formato *JSON*.

Expectativas: escalabilidade horizontal, maior flexibilidade arquitetural e interoperabilidade multiplataforma.

4.2.2 Desacoplamento e Injeção de Dependência com *Repository*

A sobrecarga dos controladores e a criação manual de dependências exigiam a adoção de uma abordagem menos acoplada. Foi introduzido o padrão *Repository*, abstraindo lógica de domínio, lógica de aplicação e acesso a dados. Esse padrão cria uma camada especializada para acesso ao banco, com serviços intermediando regras de negócio, enquanto controladores apenas administram requisições e respostas. Além disso, substituiu-se a injeção manual por mecanismos de injeção de dependências, permitindo que classes recebam suas dependências via construtor e facilitando os testes, uma vez que facilita a substituição dos repositórios por *mocks*.

Expectativas: padronização de responsabilidades, melhoria na manutenção, maior capacidade de reutilização de código e testes unitários.

4.2.3 Unificação de *Stacks* com *Node.Js*

A construção de uma *API REST* e a adoção do padrão *Repository* motivaram a migração para o *Node.js*, tanto pela alta compatibilidade com arquiteturas orientadas a eventos e aplicações em tempo real quanto pelo alinhamento tecnológico com o *frontend*, que utiliza *Vue.js*. A utilização da linguagem *JavaScript* em ambos os lados promove um ecossistema unificado, reduz a duplicidade conceitual e facilita a integração entre camadas. Estudos da indústria apontam vantagens do *Node.js* em aplicações assíncronas, operações intensivas e sistemas colaborativos (Pessôa, 2022).

Para além disso, *frameworks* como o *Express.js* simplificam o desenvolvimento de *APIs* modernas ao oferecer mecanismos de roteamento nativos, que permitem mapear *URLs* e verbos *HTTP* diretamente para funções de controle. Essa abordagem organiza o código em módulos com responsabilidades bem definidas, facilita a evolução da *API* e contribui para uma manutenção mais previsível. No ambiente de desenvolvimento, o suporte a ferramentas de *hot reload* possibilita que o servidor seja recarregado automaticamente a cada alteração no código fonte, encurtando o ciclo de *feedback*, favorecendo a experimentação rápida e reduzindo o risco de erros decorrentes de reinicializações manuais (Docs, 2025).

Expectativas: maior aderência a modelos reativos e em tempo real, unificação do *stack* tecnológico e maior escalabilidade operacional.

4.3 IMPLEMENTAÇÃO DA NOVA ARQUITETURA

Após a implementação, os resultados observados foram analisados e comparados às expectativas iniciais, destacando ganhos e custos operacionais.

4.3.1 *API REST*

A reformulação da arquitetura confirmou sua capacidade de escalabilidade e suporte multiplataforma. O sistema tornou-se apto a ser consumido por clientes diversos, consolidando um modelo de acesso unificado. Contudo, o esforço de implementação foi superior ao inicialmente previsto.

No contexto dessa migração, adotou-se a biblioteca *Axios*, um cliente *HTTP* baseado em *Promises*, amplamente utilizado em aplicações *web* para realizar requisições assíncronas a serviços *REST*, simplificando o envio e o recebimento de dados, bem como o tratamento de erros de comunicação. Com o uso do *Axios*, as chamadas diretas foram substituídas por requisições *HTTP* padronizadas, e também se fez necessário criar *endpoints* específicos para servir arquivos armazenados anteriormente em diretórios locais, incluindo imagens e recursos vinculados à pasta *storage*. Assim, embora o retorno técnico tenha sido positivo, a adoção da *API REST* demandou refatoração na camada do *frontend* e ajustes de infraestrutura.

Além disso, a *API* passou a expor seus recursos por meio de *URIs* baseadas em substantivos no plural, como */journeys*, */areas* e */contacts*, utilizando de forma consistente os méto-

dos *HTTP* para expressar operações de leitura, criação, atualização e remoção. Também foram adotadas *URIs* hierárquicas, como */users/{userId}/journeys* e */journeys/{journeyId}/games*, para representar relacionamentos entre entidades. Essa organização reforça a aderência às boas práticas do padrão *REST* e facilita a compreensão e a evolução da interface por outras equipes de desenvolvimento.

4.3.2 *Repository*

A adoção do padrão *Repository* e a divisão do *backend* em Entidades, Controladores, Interfaces de Serviço, Implementações de Serviço e Repositórios otimizaram a manutenção do código, facilitaram testes automatizados e promoveram a reutilização modular.

Entretanto, essa arquitetura possui uma curva de aprendizagem maior que a anterior, pois exige dos desenvolvedores o entendimento claro das funções e interações entre cada nível do *backend*. Nesse sentido, e visando orientar aqueles que irão interagir com a arquitetura implantada, detalham-se, abaixo, as camadas implementadas e como elas se conectam entre si e com os demais componentes do sistema:

- **Entidades (*Entities*):** As entidades representam os modelos de domínio e definem a estrutura dos dados persistidos no banco de dados. Modelam os registros, armazenando desde atributos até regras básicas do próprio dado, como validações simples ou conversões. Interações: são utilizadas diretamente pelos repositórios para consultas e persistência e podem ser retornadas por serviços após o processamento.

- **Interfaces de Serviço (*Services*):** As interfaces de serviço atuam como contratos formais que definem o conjunto de operações oferecidas pela camada de negócios, sem expor sua implementação concreta. Garantem padronização e permitem substituição ou expansão dos serviços sem impacto nas demais camadas. Interações: são injetadas nos controladores; suas implementações dependem dos repositórios para executar operações de dados.

- **Serviços (*Services Implementations*):** Os serviços encapsulam as regras de negócio e coordenam o fluxo interno da aplicação. Eles utilizam repositórios para acessar dados, aplicam validações e orquestram retornos coerentes para os controladores. Também servem como ponto de integração entre domínio e regras operacionais, pois, por exemplo, um *middleware* de autenticação pode validar permissões antes de acessar métodos do

serviço. Interações: recebem dependências via injeção, acessam repositórios, retornam resultados para controladores e podem acionar *middlewares* quando necessário para validações contextuais.

- **Repositórios (*Repositories*):** Os repositórios constituem a camada responsável por interagir com a fonte de dados. São responsáveis por criar, recuperar, atualizar e deletar registros, isolando a lógica de acesso ao banco do restante da aplicação. Além disso, padronizam o comportamento, permitindo a troca ou simulação da fonte de dados, como o uso de *mocks* em testes. Interações: são chamados diretamente pelos serviços e fazem uso de entidades.

- **Controladores (*Controllers*):** Os controladores são o ponto de entrada da *API* e interagem diretamente com o cliente. Eles recebem as requisições *HTTP*, validam apenas parâmetros básicos, acionam os serviços para tratar as operações e definem o formato da resposta enviada ao cliente. Nesta camada não deve existir lógica de negócio. Interações: recebem serviços via injeção, acionam *middlewares* antes da execução, como autenticação *JWT*, e nunca acessam repositórios diretamente.

Esse aumento de robustez ocasionou aumentos paralelos na complexidade de leitura do fluxo, na exigência de maturidade técnica e na importância de uma documentação clara.

4.3.3 *Node.js*

O uso de *Node.js* reforçou os benefícios esperados: alinhamento com a arquitetura *REST*, ecossistema padronizado com o *frontend* e maior fluidez na integração entre módulos. A compatibilidade com padrões modernos e a existência de *frameworks* que oferecem *APIs* integradas demonstraram ser uma escolha coerente para interfaces interativas e comunicação assíncrona.

Ainda assim, a expectativa de redução significativa da curva de aprendizado não se confirmou plenamente, devido à tipagem fraca nativa do *JavaScript* e à necessidade de lidar com programação assíncrona e promessas, conceitos que podem aumentar a complexidade para equipes acostumadas a linguagens síncronas, como o *PHP*. Nesse contexto, a adoção gradual de *TypeScript* configura-se como uma evolução natural da solução, uma vez que introduz tipagem estática opcional, melhora o suporte a ferramentas de desenvolvimento e reduz a probabilidade

de erros em tempo de execução, sem exigir a troca do ecossistema já consolidado (Microsoft, 2025b). Contudo, a migração completa da base de código para *TypeScript* demandaria um esforço adicional de configuração, reescrita de módulos e adaptação da equipe, o que extrapola o escopo temporal deste trabalho e, por isso, é tratada como melhoria planejada para ciclos futuros de evolução da plataforma.

4.4 DOCUMENTAÇÃO E REGISTRO DO PROCESSO

O processo de reengenharia foi documentado com registros técnicos das evoluções de código e diagramas *UML* (*Unified Modeling Language*) representando as novas estruturas internas. Foram produzidas a documentação dos *endpoints* da *API* e diagramas de classe, componentes e sequência. Essa abordagem busca garantir a continuidade do ciclo de desenvolvimento, reduzir riscos na troca de equipe e consolidar fundamentos sólidos para escalabilidade futura do sistema.

4.4.1 Documentação dos *Endpoints* da *API*

A documentação de *endpoints* estabelece o contrato de comunicação entre *frontend* e *backend*. Esta seção descreve os principais *endpoints* disponíveis na nova *API* da plataforma, organizados por funcionalidade. Cada *endpoint* inclui o método *HTTP*, o caminho e uma breve descrição.

A nomenclatura das rotas segue boas práticas associadas ao estilo *REST*, com *URIs* baseadas em recursos no plural (*/journeys*, */areas*, */contacts*) e o uso dos métodos *HTTP* para representar as operações. Além disso, são utilizadas *URIs* hierárquicas, como */users/{userId}/journeys* e */journeys/{journeyId}/games*, para explicitar relacionamentos entre entidades e facilitar a navegação pela *API*.

A primeira categoria é de Autenticação, que gerencia o ciclo de vida de usuários e sessões, utilizando *JWT* para segurança. A Tabela 1 apresenta os *endpoints* pertencentes a esta categoria.

Tabela 1 – Endpoints de Autenticação

Endpoint	Descrição
`POST /auth/register`	Registro de novo usuário
`POST /auth/login`	*Login do usuário*
`POST /auth/forgot-password`	Solicitar redefinição de senha
`POST /auth/reset-password`	Redefinir senha
`POST /auth/logout`	*Logout do usuário*

Fonte: Do autor.

A segunda categoria, de *endpoints* de Jornada, controla as operações *CRUD* (*Create, Read, Update and Delete*) de jornadas educativas, incluindo *download* e distribuição de recursos associados. Estes estão documentados na Tabela 2.

Tabela 2 – Endpoints de Jornada

Endpoint	Descrição
`GET /journeys`	Listar jornadas públicas
`GET /journeys/:journeyId`	Ver jornada específica
`GET /users/:userId/journeys`	Listar jornadas de um usuário
`POST /journeys`	Criar nova jornada
`PUT /journeys/:journeyId`	Atualizar jornada
`DELETE /journeys/:journeyId`	Excluir jornada
`GET /users/:userId/journey-exports/:journeyId`	*Download da jornada exportada*
`GET /users/:userId/files/:fileName`	Servir imagem associada à jornada

Fonte: Do autor.

A categoria de *endpoints* de Jogos manipula dados específicos dos jogos associados às jornadas, como marcas, cenários e desafios, serializados em *JSON*, conforme demonstrado na Tabela 3.

Tabela 3 – Endpoints de Jogo

Endpoint	Descrição
`GET /journeys/:journeyId/games`	Buscar jogo associado à jornada para edição
`PUT /journeys/:journeyId/games`	Atualizar dados do jogo associado

Fonte: Do autor.

Por fim, os *endpoints* de Áreas e Contato disponibilizam recursos estáticos de classificação temática e um canal de comunicação com a equipe responsável pela plataforma, como apresentado na Tabela 4.

Tabela 4 – Endpoints de Áreas e Contato

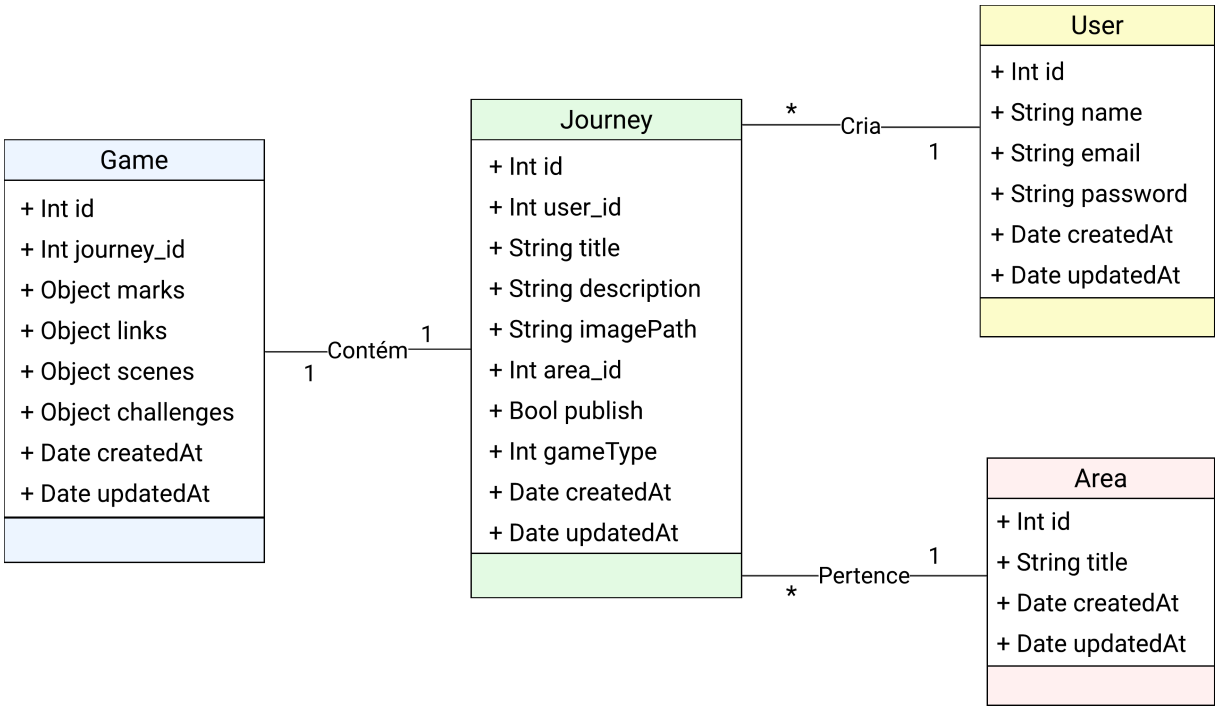
Endpoint	Descrição
`GET /areas`	Listar áreas disponíveis
`POST /contacts`	Enviar mensagem de contato

Fonte: Do autor.

4.4.2 Diagrama de Classes das Entidades Principais

O diagrama de classes define o modelo de domínio do sistema, representando entidades fundamentais e seus relacionamentos. A Figura 1 ilustra as quatro entidades principais: *User* (usuário), *Area* (área temática), *Journey* (jornada educativa) e *Game* (jogo), com cardinalidades que refletem as regras de negócio, em que um usuário cria múltiplas jornadas e cada jornada contém um jogo e pertence a uma área temática.

Figura 1 – Diagrama de Classes das entidades principais



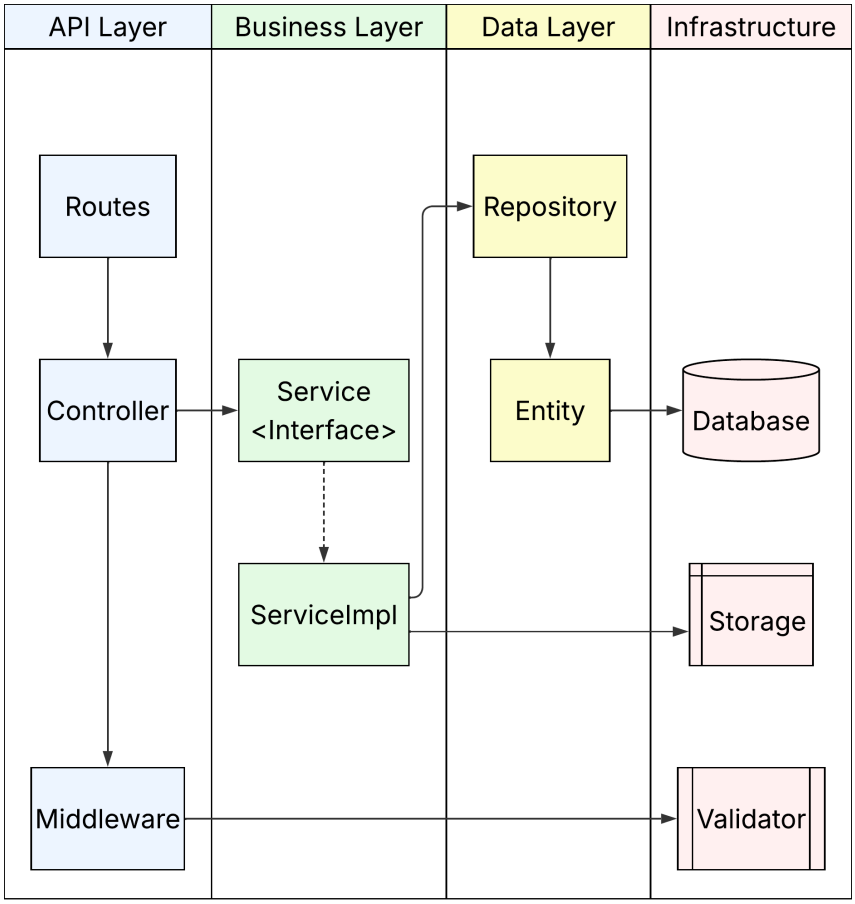
Fonte: Do autor.

4.4.3 Diagrama de Componentes da Arquitetura

O diagrama de componentes representa a arquitetura em camadas adotada na reengenharia, demonstrando a separação de responsabilidades entre módulos. A Figura 2 mostra quatro

camadas concêntricas: *API Layer* (rotas e controladores), *Business Layer* (serviços e lógica), *Data Layer* (repositórios e entidades) e *Infrastructure* (banco de dados, armazenamento local *storage* e autenticação *JWT*).

Figura 2 – Diagrama de Componentes da Arquitetura



Fonte: Do autor.

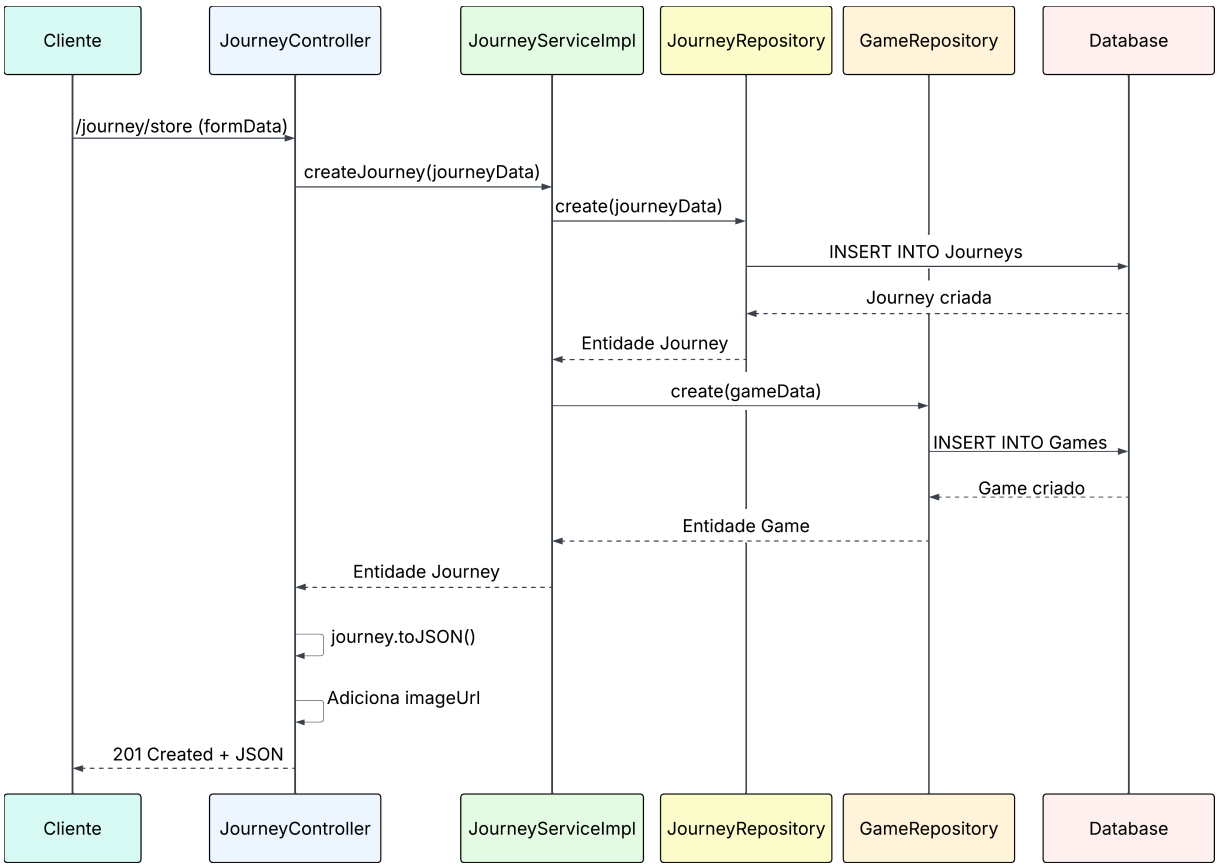
4.4.4 Diagrama de Sequência

O diagrama de sequência detalha o fluxo de interação entre componentes durante determinado processo do sistema. A Figura 3 exemplifica o processo de criação de uma jornada educacional, mostrando a comunicação sequencial entre *Controller*, *Service*, *Repository* e Banco de Dados, evidenciando o padrão de passagem de dados entre cada camada.

Em termos práticos, o fluxo tem início quando o cliente envia uma requisição para o endpoint */journey/store* com os dados do formulário, que são recebidos pelo *JourneyController* e repassados ao serviço responsável pela criação da jornada. O *JourneyServiceImpl* persiste a entidade *Journey* por meio do *JourneyRepository*, que realiza a inserção na tabela *Journeys* e

devolve a entidade criada, e em seguida cria o jogo associado utilizando o *GameRepository*, que grava os dados em *Games*. Após obter as entidades de domínio, o serviço retorna a jornada para o controlador, que serializa o objeto em *JSON*, adiciona a *imageUrl* correspondente e responde ao cliente com o status *201 Created* e o corpo em formato *JSON*.

Figura 3 – Diagrama de Sequência



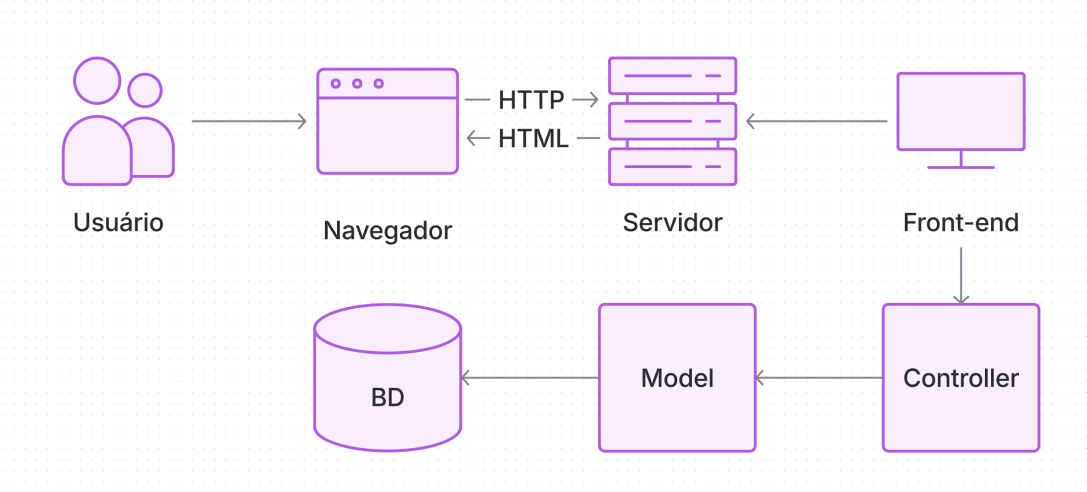
Fonte: Do autor.

5 ANÁLISE DOS RESULTADOS

A refatoração do sistema Tales Ludos, originalmente desenvolvido em *Laravel PHP*, para *Node.js* teve como objetivo primário resolver o forte acoplamento entre controladores, modelos e interface. Para evidenciar o impacto e os benefícios da reengenharia, a análise a seguir compara as arquiteturas original e refatorada, focando primeiro na estrutura base e, em seguida, no potencial de escalabilidade.

Inicialmente, a Figura 4 e a Figura 5 ilustram a mudança arquitetural central.

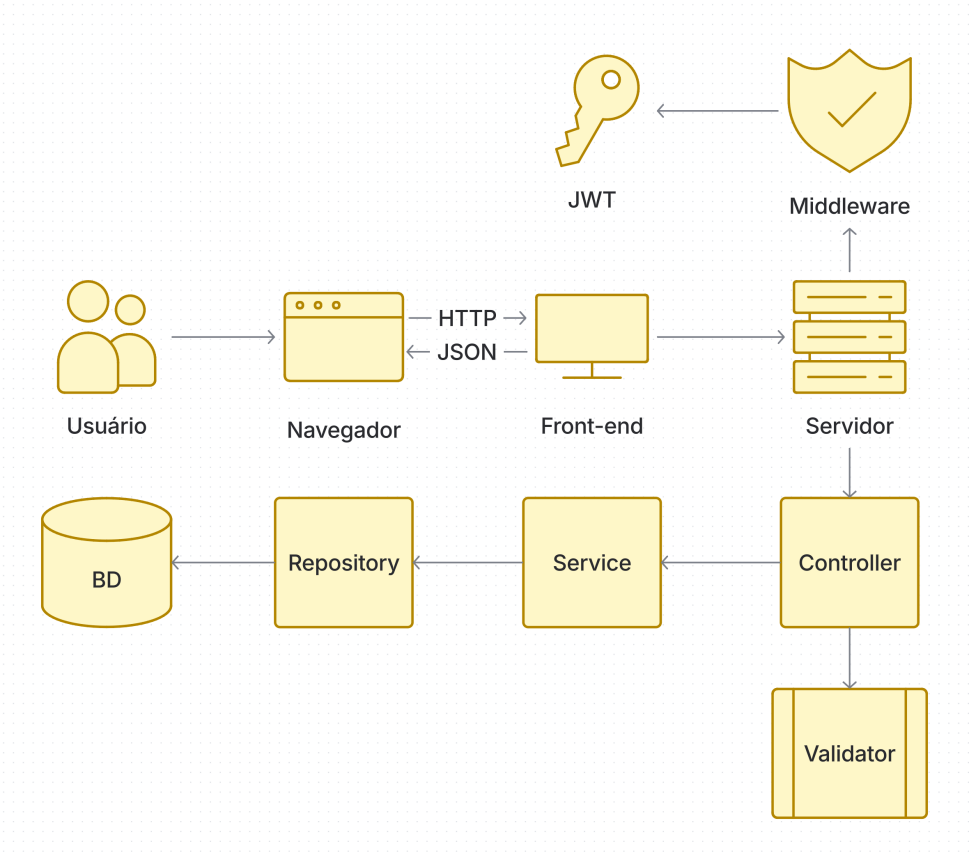
Figura 4 – Diagrama da arquitetura original



Fonte: Do autor.

O diagrama da arquitetura original (Figura 4) evidencia um padrão *MVC* monolítico clássico. O cliente (Navegador) envia requisições *HTTP* ao Servidor, que processa a lógica (*Controller* e *Model*) e é responsável por renderizar a própria interface (*Front-end*), devolvendo o *HTML* completo. A característica marcante é o alto acoplamento: a lógica de apresentação (*View*) está intrinsecamente ligada às regras de negócio e ao acesso a dados, ambos residindo na mesma aplicação no servidor.

Figura 5 – Diagrama da arquitetura refatorada



Fonte: Do autor.

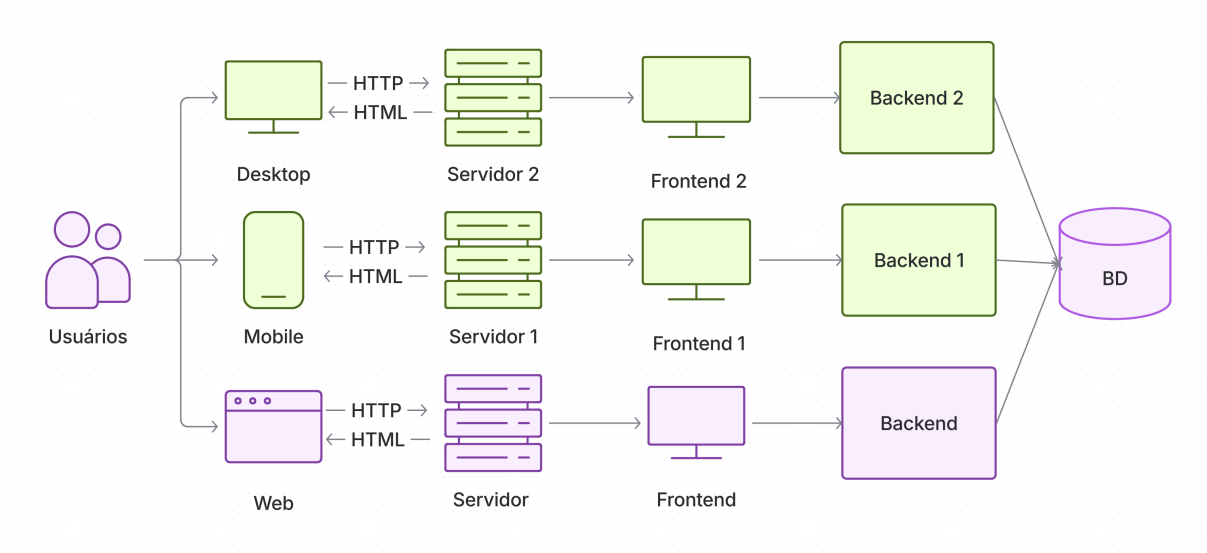
Em contrapartida, a arquitetura refatorada (Figura 5) implementa uma separação de responsabilidades clara, transformando o *back-end* em uma *API REST* independente. O *Front-end* torna-se uma entidade desacoplada, e a comunicação passa a ser feita via *HTTP* através de dados *JSON*. Internamente, o *back-end* adota padrões mais robustos: o fluxo de requisição é protegido por *Middleware JWT* (autenticação), tratado por *Controllers*, validado por um Validador, processado por *Services* (camada de negócio) e acessa o Banco de Dados através do padrão *Repository*. Este último abstrai a lógica de acesso a dados, aumentando a modularidade e a testabilidade do sistema.

O impacto estratégico dessa mudança, contudo, torna-se mais evidente ao analisar o esforço necessário para escalar o sistema, como demonstrado na comparação entre a Figura 6 e a Figura 7.

A Figura 6 ilustra a principal deficiência da abordagem monolítica diante da expansão. Ao introduzir novos clientes (*Mobile* e *Desktop*), a arquitetura original exige a criação de novos *back-ends* completos para cada plataforma. Isso não apenas eleva o custo de desenvolvimento, mas gera redundância e um alto risco de inconsistência na lógica de negócio, uma vez que

qualquer alteração precisaria ser replicada em três bases de código distintas (*Backend*, *Backend* 1 e *Backend* 2).

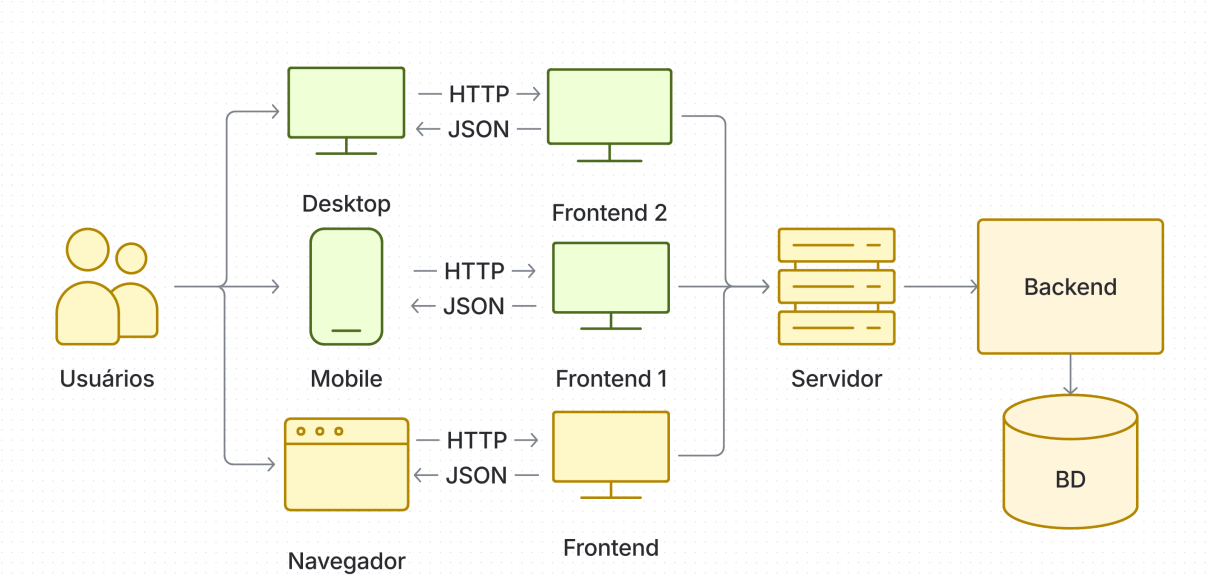
Figura 6 – Cenário de Escalabilidade na arquitetura original



Fonte: Do autor.

A Figura 7, por sua vez, demonstra o ganho fundamental da arquitetura de *API*. As novas plataformas são introduzidas apenas como novos clientes (*Frontend* 1, *Frontend* 2), mas todos consomem o mesmo e único *back-end*. A lógica de negócios, segurança e validações são implementadas uma única vez e centralizadas. O esforço de desenvolvimento para novas plataformas é, portanto, drasticamente reduzido, limitando-se basicamente à criação da nova interface. Isso resulta diretamente em uma manutenibilidade centralizada e uma escalabilidade horizontal muito mais eficiente.

Figura 7 – Cenário de Escalabilidade na arquitetura refatorada



Fonte: Do autor.

Em suma, os diagramas evidenciam que a reengenharia foi uma decisão estratégica que habilita o sistema educacional a evoluir para múltiplas plataformas com custos e prazos otimizados. Além dos ganhos arquiteturais de consistência e facilidade de manutenção futura, a migração para *Node.js* alinhou o *back-end* ao ecossistema *JavaScript* já utilizado no *front-end*, melhorando a gestão de dependências. Embora testes quantitativos de desempenho não tenham sido o foco, observou-se qualitativamente uma redução perceptível no tempo de manutenção e maior clareza estrutural no código refatorado.

6 CONCLUSÃO

A refatoração da plataforma Tales Ludos, de um sistema monolítico acoplado para uma arquitetura baseada em *API REST*, representou um avanço significativo tanto em termos estruturais quanto operacionais. A migração para *Node.js*, associada à adoção do padrão *Repository*, resultou em uma organização interna mais modular e coerente, permitindo a separação clara entre responsabilidades, o que impactou diretamente na escalabilidade, na clareza do código e no potencial de evolução contínua do sistema. Essa transformação também favoreceu a redução do acoplamento entre camadas e a padronização de fluxos internos, elementos essenciais para aplicações que demandam alta manutenibilidade ao longo do tempo.

Sob uma perspectiva científica, os resultados obtidos reforçam a relevância das práticas de reengenharia e refatoração como estratégias eficientes para estender o ciclo de vida de sistemas legados, especialmente aqueles que já possuem usuários ativos e funcionalidades críticas. A impossibilidade de reescrita total de um sistema não deve ser vista como impedimento para modernizações profundas: ao contrário, este estudo demonstra que abordagens progressivas, sustentadas por princípios arquiteturais bem definidos, são capazes de gerar melhorias substanciais sem comprometer a continuidade operacional. Além disso, a adoção de uma arquitetura orientada a recursos (*REST*) mostrou-se especialmente adequada para promover interoperabilidade, favorecer testes e possibilitar o consumo uniforme da *API* por diferentes plataformas, reforçando a robustez da aplicação e sua flexibilidade frente a cenários educacionais em constante transformação.

Os resultados alcançados confirmam ainda que a arquitetura proposta trouxe benefícios concretos na gestão do projeto. Com uma *API* independente e camadas desacopladas, tornou-se possível planejar integrações futuras, facilitar a incorporação de novos desenvolvedores na equipe e estabelecer bases sólidas para o desenvolvimento de novas interfaces, como aplicações móveis, executáveis para *desktop* ou, até, *tablets* educativos. Assim, o trabalho demonstrou que a modernização de sistemas educacionais, quando conduzida com rigor metodológico, contribui diretamente para sua sustentabilidade técnica e longevidade.

6.1 TRABALHOS FUTUROS

A evolução da plataforma Tales Ludos não se encerra com a refatoração apresentada, pelo contrário, a nova arquitetura abre espaço para um conjunto de melhorias graduais. Em

primeiro plano, destacam-se aprimoramentos de boas práticas na própria base de código, com a migração progressiva da camada de *backend* para *TypeScript*. A introdução de tipagem estática nas interfaces, serviços e repositórios tende a tornar o modelo de dados mais explícito, reduzir erros em tempo de compilação e favorecer refatorações seguras, mantendo, ao mesmo tempo, a compatibilidade com o ecossistema atual baseado em *Node.js* e na *API REST* já consolidada.

A partir dessa base mais robusta, será pertinente avançar na expansão externa da plataforma, explorando a criação de novas interfaces, como aplicativos *mobile* e versões executáveis, bem como integrações com outros sistemas e ambientes educacionais. Ao reutilizar a *API REST* como ponto central de comunicação, essas extensões podem oferecer experiências enriquecidas em diferentes dispositivos e contextos de uso, o que tende a elevar tanto a quantidade de dados trafegados quanto o número de operações simultâneas no ambiente de produção.

Nesse cenário de maior carga, tornam-se prioritárias melhorias internas de desempenho e qualidade. Destaca-se a implementação de mecanismos de *cache*, como o uso de *Redis*, para reduzir latências em operações repetitivas e otimizar o tempo de resposta de serviços críticos, especialmente consultas frequentes de jornadas e recursos estáticos. Em paralelo, a adoção sistemática de *DTOs* (*Data Transfer Objects*) pode padronizar entradas e saídas da *API*, reduzir ambiguidades no contrato entre cliente e servidor e facilitar testes automatizados. Complementarmente, o desenvolvimento contínuo de testes unitários e de integração, apoiado na arquitetura em camadas e na injeção de dependências, tende a ampliar a confiabilidade da plataforma diante de futuras modificações.

Por fim, a consolidação desse ecossistema demanda uma ampliação da documentação e da avaliação empírica. A utilização de ferramentas como *Swagger/OpenAPI* pode sistematizar a descrição dos *endpoints*, facilitar a integração com equipes externas e apoiar a governança da *API*. Em complemento, a realização de experimentos quantitativos com usuários e cenários reais de uso permitirá mensurar indicadores como desempenho, tempo de resposta, engajamento em jornadas educacionais e percepção de usabilidade, fornecendo evidências objetivas sobre os impactos pedagógicos e técnicos das mudanças arquiteturais.

De forma geral, essas perspectivas reforçam o compromisso da plataforma com boas práticas de engenharia de *software* e com a consolidação de um ecossistema educacional escalável, sustentável e preparado para acompanhar as transformações tecnológicas e pedagógicas dos próximos anos. Nesse sentido, destaca-se também o papel do Tales Ludos enquanto Recurso Educacional Aberto (REA), que fomenta a criatividade e permite que educadores e estudantes

participem ativamente da construção de experiências de aprendizagem mais significativas, reafirmando o propósito social de promover inclusão e fortalecer a cultura do compartilhamento no contexto educacional.

REFERÊNCIAS

ACADEMY, K. **About Khan Academy**. 2024. Https://www.khanacademy.org/about. Acesso em: 25 Jun. 2024.

BASS, L.; CLEMENTS, P.; KAZMAN, R. Software architecture in practice. **Addison-Wesley Professional**, 2021.

BRAY, T. **The JavaScript Object Notation (JSON) Data Interchange Format,**. 2017. Https://datatracker.ietf.org/doc/html/rfc8259. Acesso em: 20 out. 2025.

CAPES e UNIFAL. **Plataforma Tales Ludos**. http://talesludos.bcc.unifal-mg.edu.br/. Acesso em: 05 dez. 2025.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: a taxonomy. **IEEE Software**, 1990.

DETERDING, S. e. a. From game design elements to gamefulness: defining "gamification". **Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments**, 2011.

DOCS, M. W. **Introdução Express/Node – Aprendendo desenvolvimento web**. 2025. Https://developer.mozilla.org/pt-BR/docs/Learn$_{w}eb_{d}evelopment/Extensions/Server-side/Express_{N}odejs/Introduction.Acessoem$: 20out.2025.

DUOLINGO. **About Duolingo**. 2024. Https://www.duolingo.com/info. Acesso em: 25 Jun. 2024.

FOUNDATION, N. **About Node.js – Run JavaScript Everywhere**. 2025. Https://nodejs.org/about. Acesso em: 20 out. 2025.

FOWLER, M. Refactoring: improving the design of existing code. **Addison-Wesley Professional**, 2018.

IBM. **O que é API REST?** 2025. Https://www.ibm.com/br-pt/think/topics/rest-apis. Acesso em: 20 out. 2025.

KLEIN, D. R. e. a. Tecnologia na educação: evolução histórica e aplicação nos diferentes níveis de ensino. **Educere - Revista da Educação da UNIPAR**, 2020.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [S.l.]: Prentice Hall, 2017.

MESHRAM, S. U. Evolution of modern web services – rest api with its architecture and design. **International Journal of Research in Engineering, Science and Management**, 2021.

MICROSOFT. **Projetando a camada de persistência de infraestrutura**. 2025. Https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design. Acesso em: 28 out. 2025.

MICROSOFT. **TypeScript para programadores JavaScript**. 2025. Https://www.typescriptlang.org/pt/docs/handbook/typescript-in-5-minutes.html. Acesso em: 08 dez. 2025.

PESSôA, C. O que é node.js: vantagens e usos na programação. **Alura**, 2022.

PIRES, P. F. e. a. Plataformas para a internet das coisas. **Minicursos SBRC – Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**, 2015.

RAJLICH, V. Software evolution and maintenance. **Proceedings of the Future of Software Engineering (FOSE)**, 2005.

SHORT, D. Teaching scientific concepts using a virtual world – minecraft. **Teaching Science**, 2012.

SILVA, L. E. **Manual Tales Ludos: criação de jogos educacionais digitais**. [S.l.]: Pimenta Cultural, 2025. Disponível em: https://www.pimentacultural.com/wp-content/uploads/2025/06/eBook_manual-tales-ludos.pdf. Acesso em: 10 out. 2025.

TANENBAUM, A. S.; STEEN, M. V. Distributed systems. **CreateSpace Independent Publishing Platform**, 2017.

UNESCO. **Declaração REA de Paris em 2012**. 2012. Https://unesdoc.unesco.org/ark:/48223/pf0000246687. Congresso Mundial sobre Recursos Educacionais Abertos (REA), Paris, 20–22 jun. 2012. Acesso em: 10 abril 2025.

VELOSO B. E MILL, D. Educação a distância e ensino remoto: Oposição pelo vértice. **Revista Portuguesa de Educação**, 2024.

VUE.JS. **Vue.js – The Progressive JavaScript Framework**. 2025. Https://vuejs.org/. Acesso em: 20 out. 2025.