ANÁLISE ARQUITETURAL E REFATORAÇÃO DE APLICAÇÕES WEB: Um Estudo de Caso da Aplicação *Tem Lógica*

Davi José Lucena Luiz

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2025

Davi José Lucena Luiz

ANÁLISE ARQUITETURAL E REFATORAÇÃO DE APLICAÇÕES WEB: Um Estudo de Caso da Aplicação *Tem Lógica*

Relatório Técnico apresentado ao curso Ciência da Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Lincoln David Nery e Silva

Maio de 2025

Catalogação na publicação
Seção de Catalogação e Classificação

L953a Luiz, Davi Jose Lucena.
Análise arquitetural e refatoração de aplicações web: um estudo de caso da aplicação tem lógica / Davi Jose Lucena Luiz. - João Pessoa, 2025.
71 f. : il.

Orientação: Lincoln David Nery e Silva.
TCC (Graduação) - UFPB/CI.

1. Avaliação de desempenho. 2. Arquitetura de software. 3. Aplicação web. 4. Evolução do sistema. 5. Manutenibilidade. I. Silva, Lincoln David Nery e. II. Título.

UFPB/CI CDU 004.4

Elaborado por MICHELLE DE KASSIA FONSECA BARBOSA - CRB-738

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da computação intitulado ***ANÁLISE ARQUITETURAL E REFATORAÇÃO DE APLICAÇÕES WEB: Um Estudo de Caso da Aplicação Tem Lógica*** de autoria de Davi José Lucena Luiz, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Lincoln David Nery e Silva
Universidade Federal da Paraíba

Prof. Dra. Yuska Paola Costa Aguiar
Universidade Federal da Paraíba

Prof. Dra. Danielle Rousy Dias Ricarte
Universidade Federal da Paraíba

Coordenador(a) do Departamento de informática
Giorgia de Oliveira Mattos
CI/UFPB

João Pessoa, 09 de maio de 2025

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

RESUMO

O trabalho em questão trata da análise e reestruturação da aplicação Tem Lógica, uma plataforma educacional voltada para o ensino de matemática no ensino fundamental por meio de jogos online interativos. A aplicação permite que professores criem atividades personalizadas e acompanhem o desempenho dos alunos. Desenvolvida inicialmente com *HTML*, *CSS* e *JavaScript* no *front-end*, *Node.js* no *back-end* e *MySQL* no armazenamento, a plataforma enfrentava problemas de desempenho e manutenção devido à ausência de um planejamento arquitetural adequado. Este trabalho realizou a identificação de falhas, análise da arquitetura, refatoração de trechos críticos de código e proposição de melhorias estruturais, como o uso de *cookies* para identificar falhas de conexão e o desacoplamento entre camadas. Como resultado, a aplicação torna-se mais estável, modular e preparada para futuras evoluções, com melhoria significativa na manutenibilidade e na performance geral do sistema.

Palavras-chave: Avaliação de Desempenho. Arquitetura de Software. Aplicação Web. Evolução do Sistema. Manutenibilidade.

ABSTRACT

This work focuses on the analysis and restructuring of the Tem Lógica application, an educational platform aimed at teaching mathematics to elementary school students through interactive online games. The application allows teachers to create personalized activities and monitor students' performance. Initially developed using *HTML*, *CSS*, and *JavaScrip*t on the *front-end*, *Node.js* on the *back-end*, and *MySQL* for data storage, the platform faced performance and maintenance issues due to the lack of proper architectural planning. This study involved identifying system failures, analyzing the architecture, refactoring critical code sections, and proposing structural improvements such as the use of cookies to detect connection issues and the decoupling of system layers. As a result, the application became more stable, modular, and better prepared for future developments, with significant improvements in maintainability and overall system performance.

Key-words: Maintainability. Performance Evaluation. Software Architecture. System Evolution. Web Application.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

API – *Application Programming Interface*

CD – *Continuous Delivery*

CI – *Continuous Integration*

CSS – *Cascading Style Sheets*

HTML – *Hypertext Markup Language*

HTTP – *HyperText Transfer Protocol*

HTTPS – *HyperText Transfer Protocol Secure*

JS – *JavaScript*

JSON – *JavaScript Object Notation*

LAVID – *Laboratório de Aplicações de Vídeo Digital*

RDBMS – *Relational Database Management System*

SGDB – *Sistema de Gerenciamento de Banco de Dados*

SQL – *Structured Query Language*

Sumário

1 INTRODUÇÃO

A plataforma Tem Lógica é uma ferramenta educativa voltada para o ensino do pensamento computacional por meio de atividades e jogos online, proporcionando uma experiência interativa que torna o estudo da matemática mais envolvente e acessível. Ela permite que os professores criem e personalizem roteiros de atividades, acompanhem o progresso dos alunos e gerem relatórios detalhados, enquanto os alunos se beneficiam de uma abordagem dinâmica e motivadora.

Durante o desenvolvimento, no qual tive a oportunidade de participar, dificuldades emergiram com a adição de novas funcionalidades sem um planejamento voltado para a evolução do sistema. Instabilidades, como quedas frequentes do banco de dados, falta de integração entre ambientes de teste e produção e o acoplamento entre *front-end* e *back-end*, destacaram a necessidade de intervenções que garantisse a continuidade e a robustez do serviço.

A participação prévia no desenvolvimento da plataforma, permitiu observar desde então diversas limitações estruturais que impactavam sua estabilidade e manutenção. No entanto, os erros apresentados neste trabalho foram identificados a partir de um estudo técnico mais recente, conduzido localmente de forma manual. Esse estudo utilizou inspeção direta da interface, análise do código-fonte e observação de *logs* gerados em ambiente de desenvolvimento.

Em resposta, as correções propostas neste trabalho focaram em melhorar a disponibilidade e a estabilidade do sistema, assegurando que novos recursos pudessem ser implementados sem comprometer o funcionamento da plataforma. Como resultado, as melhorias buscam oferecer um desempenho mais consistente e uma estrutura mais modular, facilitando futuras evoluções.

Neste sentido, o trabalho tem como objetivo demonstrar uma análise da arquitetura e propor melhorias na aplicação web Tem Lógica, desenvolvida no Laboratório de Aplicações de Vídeo Digital (LAVID).

1.1 Tema

Diante do constante avanço tecnológico e das crescentes demandas por plataformas web robustas e eficientes, desenvolver uma aplicação que funcione adequadamente, sem falhas e gargalos, tem se tornado essencial. A plataforma Tem Lógica, abordada neste estudo de caso, visa atender essas necessidades por meio da adoção de boas práticas como a separação de camadas, detecção de falhas de conexão, o tratamento de exceções no lado do servidor e a organização do código com padrões modulares.

1.2 Problema

O problema em questão aborda as principais falhas e problemas encontrados no desenvolvimento da plataforma Tem Lógica, incluindo o acoplamento e dependência entre as camadas, erros de lógica de código, falhas de integração, autenticação e desempenho inadequado. A abordagem para resolver esses problemas envolve a análise da arquitetura da aplicação, com foco em entender e identificar as causas da instabilidade. Muitos desses problemas são comuns em diversas plataformas e aplicações web. A resolução pode variar em dificuldade dependendo do tamanho, complexidade e outros fatores, mas é possível corrigir e implementar soluções que garantem o funcionamento adequado do sistema.

1.3 Objetivo geral

O trabalho tem como objetivo realizar uma análise arquitetural da aplicação web Tem Lógica e propor melhorias por meio da refatoração do código-fonte. As ações buscam reduzir o acoplamento entre camadas, melhorar a organização estrutural da aplicação, facilitar a manutenção e promover uma evolução mais sustentável do sistema. Além disso, fornece uma avaliação comparativa de desempenho entre a versão original e a versão refatorada, a fim de apresentar os impactos das mudanças implementadas.

1.4 Objetivos específicos

No que diz respeito aos objetivos específicos, pretende-se:

- Apresentar um estudo com base em análises e testes da plataforma Tem Lógica para identificar possíveis erros, falhas e *bugs* antes de implementar e propor melhorias;
- Analisar cada problema encontrado, descrevendo sua ocorrência e impacto na aplicação como um todo;
- Propor melhorias de código de forma a melhorar a eficiência, desempenho e manutenção;
- Realizar comparativos através de imagens e diagramas, fluxo de funcionamento da plataforma antes e depois das melhorias.

1.5 Estrutura do relatório técnico

Este trabalho foi estruturado na forma de relatório técnico para oferecer uma visão abrangente do estudo sobre as melhorias de desempenho na aplicação Tem Lógica. No Capítulo 1, são apresentadas a introdução ao tema e a sua importância, a definição do problema, as premissas e hipóteses, bem como os objetivos gerais e específicos do trabalho.

O Capítulo 2 apresenta uma visão geral da plataforma Tem Lógica, abordando seu propósito pedagógico, principais funcionalidades, tipos de usuários e fluxos de interação. Também são descritos aspectos técnicos e a evolução do sistema ao longo do projeto.

O Capítulo 3 descreve o processo de refatoração da plataforma Tem Lógica, abordando problemas identificados no código, como acoplamento e falhas de tratamento de erros. Apresenta também as soluções propostas e implementadas, visando melhorar a estrutura, a manutenibilidade e a escalabilidade do sistema.

O Capítulo 4 apresenta os principais resultados da análise técnica da plataforma Tem Lógica, com foco na identificação de problemas estruturais. São discutidas propostas e modificações aplicadas para melhorar a escalabilidade, a manutenção e o desempenho do sistema.

Finalmente, o Capítulo 5 apresenta as conclusões obtidas a partir da análise e reformulação da arquitetura da plataforma. Destaca-se a importância da separação entre camadas para melhorar a manutenção, os testes e a escalabilidade do sistema. Também são sugeridas direções para futuros aprimoramentos.

Referências bibliográficas serão incluídos nas seções finais, fornecendo materiais suplementares, como códigos e diagramas, listando as fontes consultadas durante o desenvolvimento do trabalho.

2 APRESENTAÇÃO DO TEM LÓGICA

Neste capítulo, será realizada uma análise da plataforma Tem Lógica, destacando suas principais funcionalidades. Inicialmente, será apresentada uma visão geral da aplicação, descrevendo como ela é utilizada por alunos e professores através de casos de uso.

2.1 O que é o Tem Lógica?

O Tem Lógica é uma plataforma que oferece uma solução para explorar o pensamento computacional através do aprendizado matemático de alunos do ensino fundamental, utilizando atividades e jogos interativos online. Esses recursos, documentados no livro O desenvolvimento das bases do pensamento computacional e a matemática: uma relação que tem lógica (NERY E SILVA; RÊGO; RÊGO, 2022), incluem formas geométricas, sequências numéricas, repetições e tarefas de completar números, criando um ambiente de aprendizado dinâmico e envolvente para os estudantes. O objetivo deste capítulo é apresentar a plataforma Tem Lógica desde as suas funcionalidades, usuários e fluxos de funcionamento.

Utilizando diagramas e imagens, este trabalho demonstra a arquitetura do Tem Lógica e analisa o funcionamento de cada componente e a comunicação entre as funções presentes nas diferentes partes do sistema. Embora o foco principal esteja no serviço oferecido pela plataforma, a análise busca assegurar que as soluções propostas não comprometam a experiência ou a usabilidade do usuário.

2.2 Desenvolvimento

A plataforma Tem Lógica foi desenvolvida no contexto do projeto Paraíba Humana e Inteligente (PHI), em parceria com o Laboratório de Aplicações de Vídeo Digital (LAVID). Inicialmente, a proposta envolvia apenas a criação de alguns jogos educativos isolados. No entanto, com o avanço do desenvolvimento, esses jogos foram organizados em categorias, segmentadas de acordo com o ano escolar dos alunos.

Com a expansão da aplicação, foram implementadas novas funcionalidades, como a criação de atividades personalizadas por parte dos professores e o monitoramento do desempenho dos alunos por meio de relatórios. Esse crescimento exigiu a divisão do trabalho entre diferentes frentes: *front-end*, *back-end*, banco de dados e processos de *deploy*.

Durante minha participação no projeto, atuei principalmente no desenvolvimento de interfaces e na implementação de novas funcionalidades na camada de *front-end*. Na fase final do desenvolvimento e entrega da plataforma, concentrei esforços na correção de *bugs* e no suporte à equipe, especialmente na criação e integração de novos jogos.

2.3 Usuários

A plataforma Tem Lógica é voltada para dois grupos de usuários: alunos do ensino fundamental e professores. Os alunos utilizam a plataforma para desenvolver o pensamento matemático por meio de jogos e atividades interativas. Eles acessam desafios matemáticos, resolvem problemas lógicos e avançam nas fases conforme seu desempenho.

Os professores, por sua vez, têm acesso a ferramentas para criar e gerenciar atividades personalizadas para seus alunos. A plataforma permite que eles acompanhem o desempenho dos estudantes por meio de relatórios detalhados, identificando acertos, dificuldades e progresso individual. Além disso, os professores podem configurar tarefas de acordo com o nível de ensino e o conteúdo abordado.

Vale destacar que o aluno pode jogar sem necessariamente estar associado a uma atividade criada pelo professor. Ele pode ingressar na plataforma e escolher um jogo dentro da categoria de algum ano. No entanto, existe uma outra maneira dele jogar, por meio da criação de uma atividade realizada pelo professor.

2.4 Apresentando suas funcionalidades gerais

2.4.1 Fluxo de visão do aluno

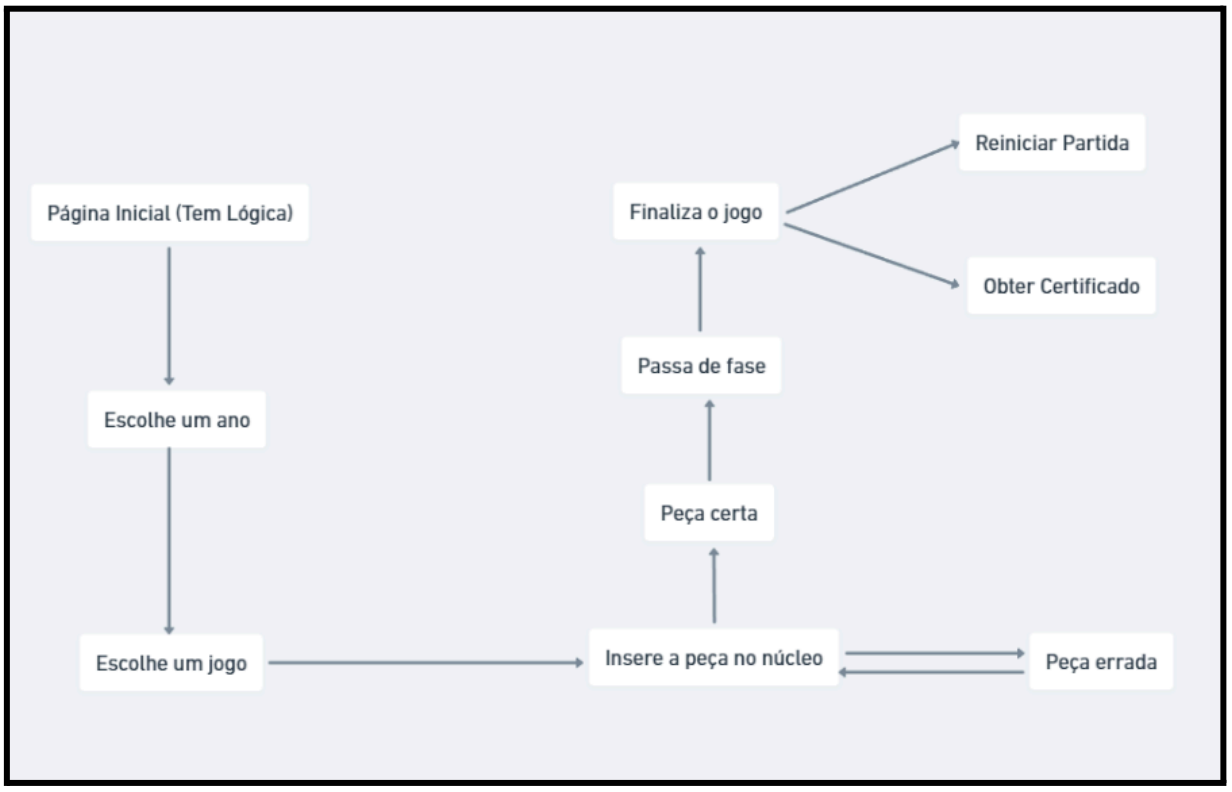

Figura 1: Fluxo do aluno interagindo com os jogos (Fonte: O próprio autor).

A jornada do aluno na plataforma segue um fluxo estruturado, proporcionando uma experiência de aprendizado interativo. Conforme ilustrado na Figura 1, o processo tem início na Página Inicial, onde o aluno seleciona o ano letivo correspondente e insere seu nome para personalizar a experiência. Esse passo inicial permite que o sistema registre o progresso do aluno e adapte os desafios conforme o nível selecionado.

Após essa etapa, o aluno escolhe um dos jogos disponíveis para iniciar sua atividade. Cada jogo apresenta desafios específicos que exigem a resolução de problemas por meio da inserção de peças em um "Núcleo" exibido na tela. A mecânica do jogo incentiva a tentativa e o erro, permitindo que o aluno desenvolva habilidades lógicas ao longo da experiência.

Quando uma peça é corretamente posicionada, o aluno avança para a próxima fase, encontrando desafios progressivamente mais complexos. Se a peça inserida estiver errada, o sistema permite novas tentativas até que a resposta correta seja encontrada.

Ao concluir todas as fases do jogo, o aluno finaliza sua jornada dentro daquela atividade e tem duas opções: reiniciar a partida para reforçar seu aprendizado ou obter um certificado como reconhecimento pelo seu desempenho.

2.4.2 Fluxo de visão do professor

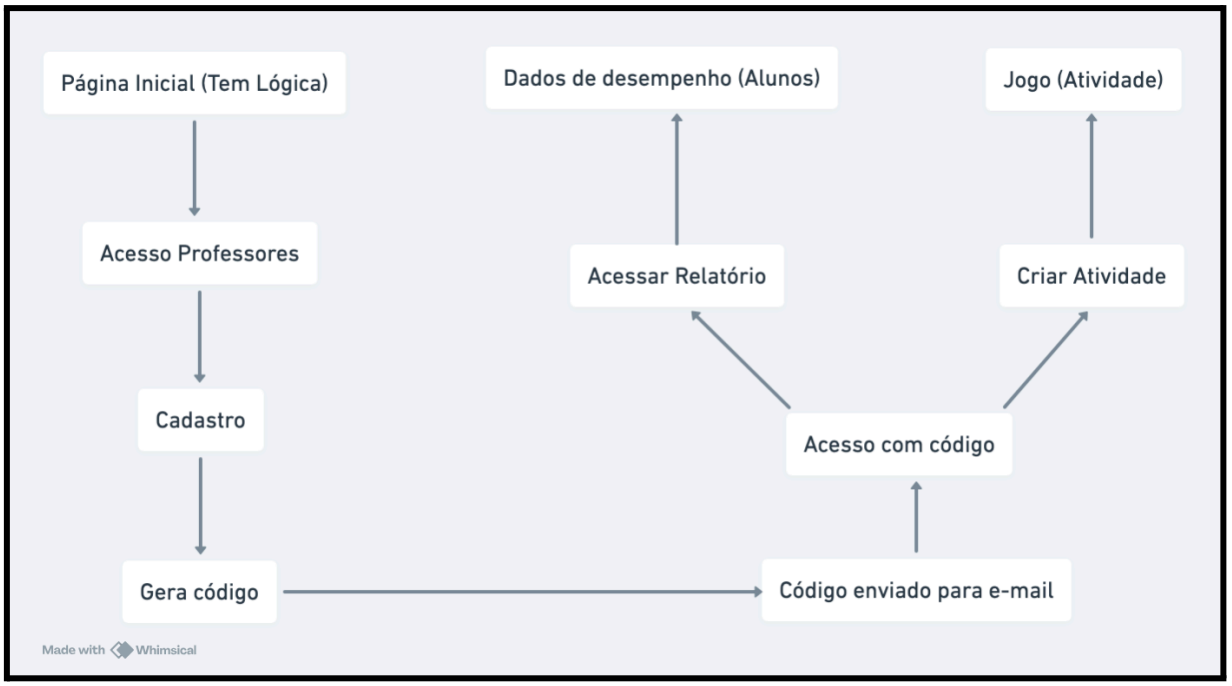

Figura 2: Fluxo do professor realizando a criação das tarefas (Fonte: O próprio autor).

A jornada do professor tem um papel fundamental no acompanhamento e na personalização da experiência dos alunos. No fluxo dessa visão, ilustrado na Figura 2, a Página Inicial apresenta uma opção específica para professores, onde é possível realizar um cadastro e gerar um código de acesso. Esse código é essencial para vincular os alunos às atividades criadas pelo próprio professor e garantir um acompanhamento estruturado.

Após gerar o código, o professor pode utilizá-lo para criar atividades dentro do jogo , personalizando os desafios conforme os objetivos pedagógicos desejados. Isso possibilita uma abordagem mais flexível e adaptada ao ritmo de aprendizado dos alunos.

Além da criação de atividades, o professor tem acesso a uma ferramenta essencial para o monitoramento do progresso dos estudantes: os relatórios de desempenho. Esses relatórios contêm informações detalhadas sobre o desempenho individual e coletivo dos alunos, permitindo que o professor identifique dificuldades específicas e ajuste as atividades de forma estratégica.

Para garantir a segurança e integridade dos dados, o acesso aos relatórios requer um código específico, que é enviado por e-mail. Esse processo assegura que apenas os professores autorizados possam visualizar as informações acadêmicas dos alunos.

A plataforma, ao oferecer essa estrutura, permite que os professores atuem de forma mais ativa e eficaz na mediação do aprendizado, personalizando tarefas, acompanhando o progresso e ajustando estratégias para melhorar o desempenho dos alunos.

2.4.3 Na visão do aluno - Casos de uso

Nesta seção serão apresentados os casos de uso referentes ao fluxo e interação na visão do aluno. Os casos de uso apresentados neste trabalho estão identificados por códigos numéricos para facilitar a organização e a referência ao longo do texto. Por exemplo, UC01 representa o Caso de Uso 1, UC02 o Caso de Uso 2, e assim sucessivamente.

UC01 – Selecionar o ano escolar

- **Ator Principal**: Aluno
- **Descrição**: O aluno acessa a tela inicial da plataforma Tem Lógica e seleciona o ano escolar correspondente (entre o 1º e o 6º ano), Figura 3. Cada série é representada por uma cor distinta: verde, amarelo, azul ou vermelho. A seleção do ano define quais jogos estarão disponíveis na próxima etapa.
- **Pré-condições**:
    - O aluno acessou corretamente a plataforma.
    - A tela inicial foi carregada com sucesso.
- **Fluxo Principal de Eventos**:
    - O sistema exibe a tela com as opções de anos escolares.
    - O aluno visualiza as opções disponíveis.
    - O aluno clica na opção desejada (de 1º a 6º ano).
    - O sistema carrega e exibe os jogos disponíveis para o ano selecionado.
- **Fluxos Alternativos**:
    - 3a. Caso o aluno não selecione nenhum ano, o sistema permanece na mesma tela aguardando a interação.
- **Pós-condições**:
    - O sistema exibe corretamente a lista de jogos referentes ao ano escolar

selecionado.

Figura 3: Tela de seleção de ano (Fonte: O próprio autor).

UC02 – Escolher um jogo

- **Ator Principal**: Aluno
- **Descrição**: Após selecionar o ano escolar, o aluno acessa uma lista de jogos educativos relacionados àquele ano, representado na Figura 4. Cada jogo é voltado para práticas matemáticas, com diferentes temáticas como sequências, formas geométricas e operações simples.
- **Pré-condições**:
  - O aluno selecionou corretamente um ano escolar.
- **Fluxo Principal de Eventos**:
  - O sistema exibe a lista de jogos referentes ao ano escolhido.
  - O aluno visualiza os jogos disponíveis.
  - O aluno clica sobre o jogo desejado.
  - O sistema carrega a interface do jogo.
- **Pós-condições**:
  - O sistema inicia a execução do jogo selecionado.

Figura 4: Tela de opções de jogos (Fonte: O próprio autor).

UC03 – Interagir com o jogo

- **Ator Principal**: Aluno
- **Descrição**: O aluno interage com o Jogo (Figura 5) escolhendo e arrastando peças de uma barra inferior para uma área chamada "Núcleo" (Figura 6), com o objetivo de completar a sequência apresentada. Após montar a sequência, o aluno clica em "Terminei" para verificar a resposta.
- **Pré-condições**:
    - O aluno selecionou o Jogo da Repetição.
- **Fluxo Principal de Eventos**:
    - O sistema exibe a interface do jogo.
    - O aluno observa a sequência a ser completada.
    - O aluno arrasta as peças para a área do Núcleo.
    - O aluno clica em "Terminei".
    - O sistema verifica a resposta.
- **Fluxos Alternativos**:
    - Se a resposta estiver correta, o sistema exibe mensagem de parabenização e avança para a próxima fase (figura 7).
    - Se a resposta estiver incorreta, o sistema exibe uma mensagem de erro e solicita nova tentativa (figura 8).
- **Pós-condições**:

- O aluno é direcionado para a próxima fase ou permanece na mesma até acertar.

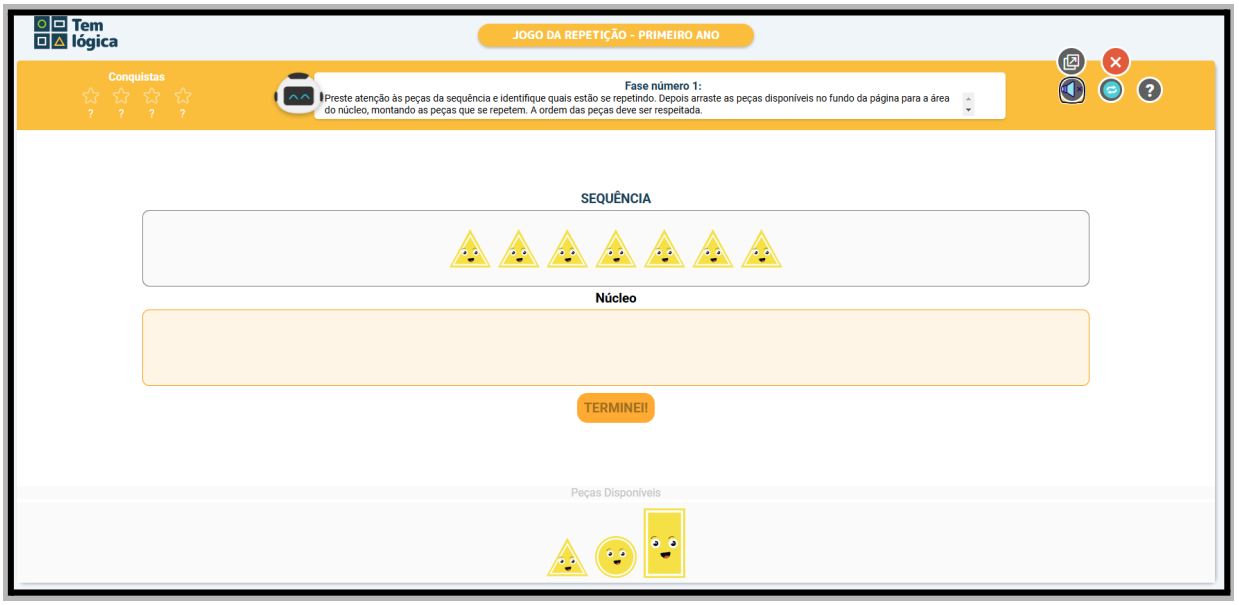

Figura 5: Tela do jogo da repetição (Fonte: O próprio autor).

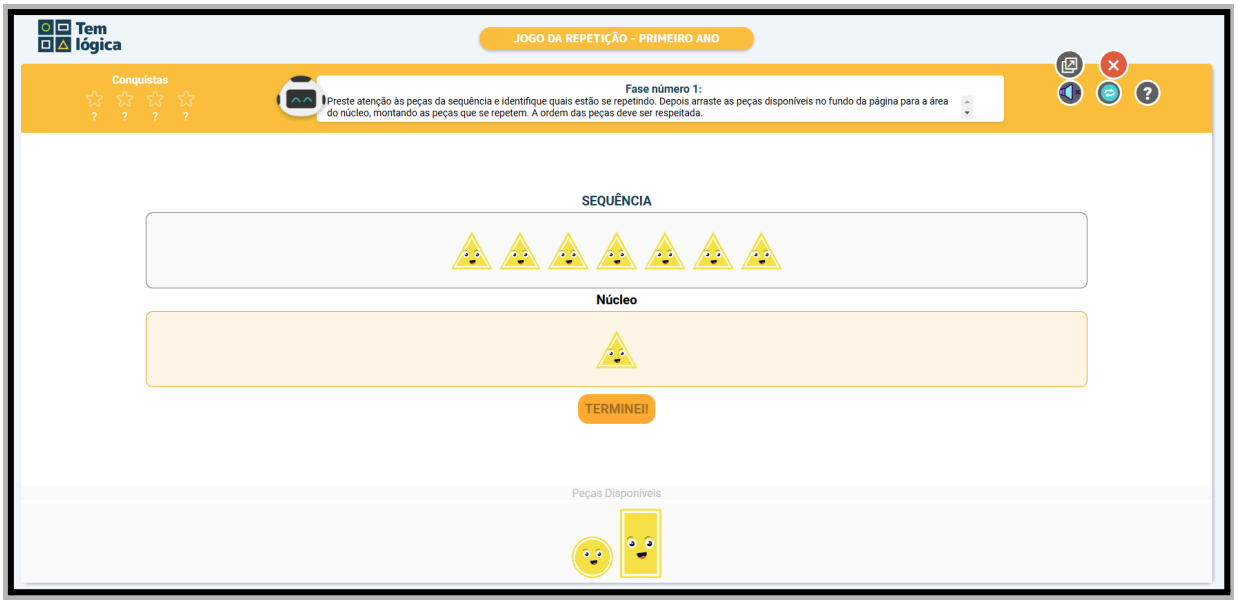

Figura 6: Selecionando e arrastando a peça para área do núcleo (Fonte: O próprio autor).

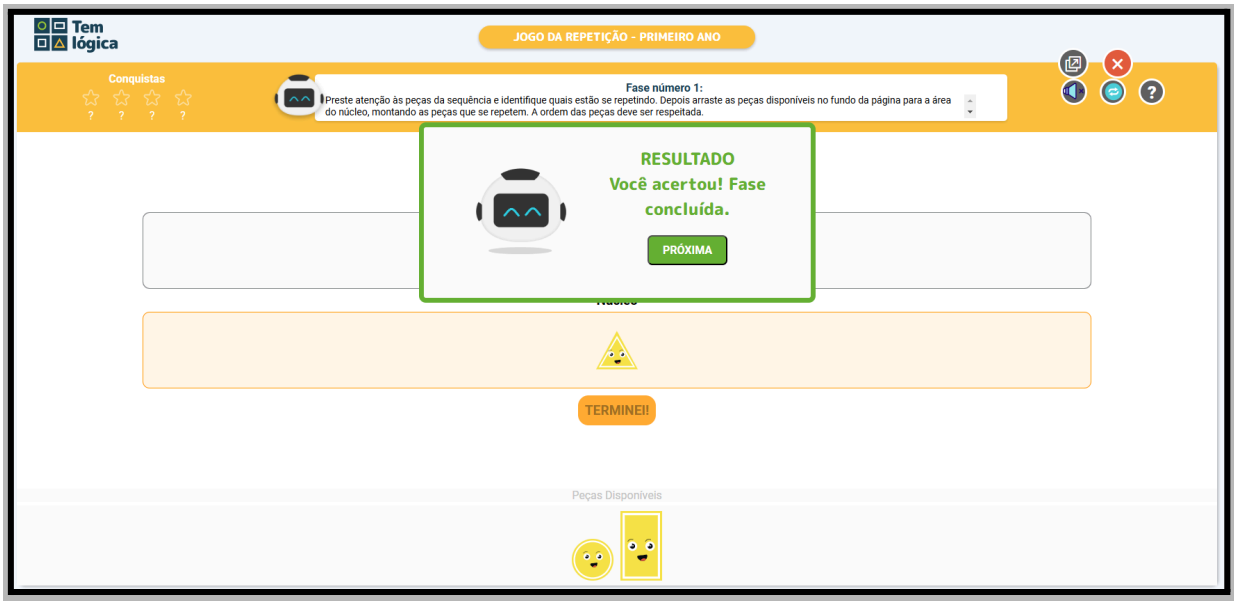

Figura 7: Janela para parabenizar o estudante pela seleção da peça correta (Fonte: O próprio autor).

Figura 8: Janela para avisar ao estudante sobre a peça errada (Fonte: O próprio autor).

2.4.4 Na visão do professor - Casos de uso

UC04 – Cadastrar-se como professor

- **Ator Principal**: Professor
- **Descrição**: O professor acessa a tela de cadastro e preenche um formulário com nome e e-mail, conforme ilustrado na Figura 9. Após isso, clica em "Gerar Novo Código" e recebe um código de acesso por e-mail para autenticar-se na plataforma.

- **Pré-condições**:
    - O professor acessou a tela de *login*/cadastro da plataforma.
- **Fluxo Principal de Eventos**:
    - O sistema exibe o formulário de cadastro.
    - O professor insere nome e e-mail.
    - O professor clica em "Gerar Novo Código".
    - O sistema envia o código por e-mail.
    - O professor insere o código e acessa a plataforma.
- **Pós-condições**:
    - O professor está autenticado e pode criar ou acessar atividades.

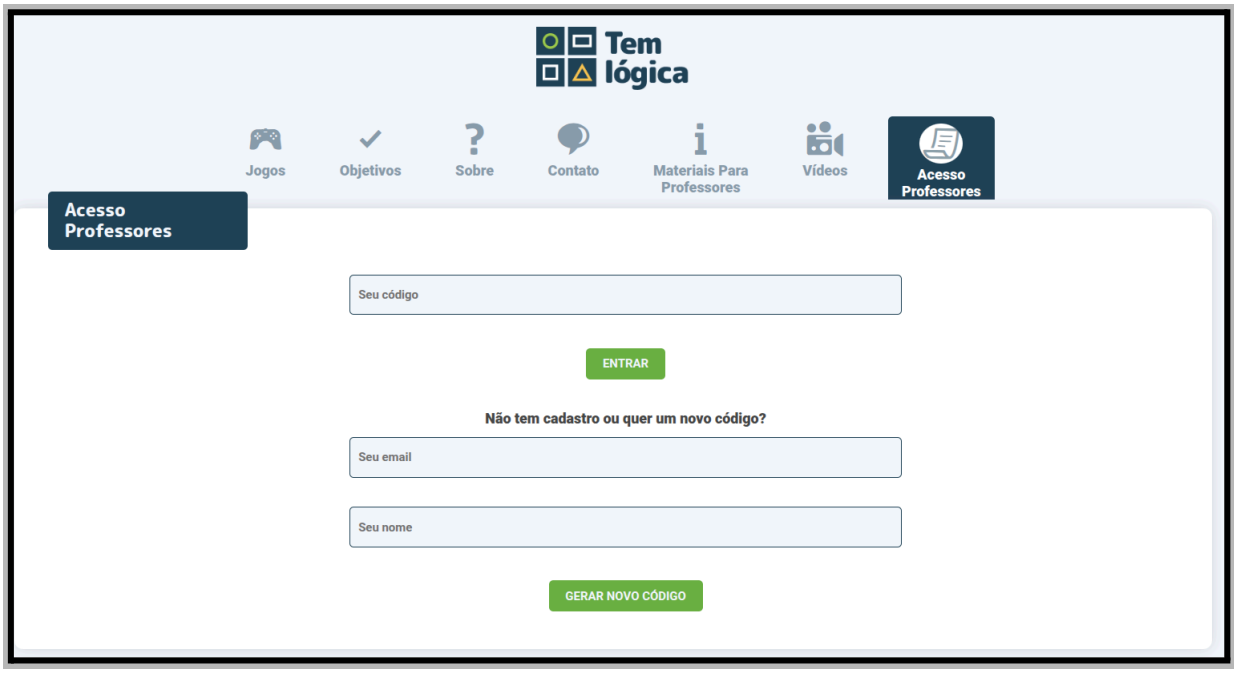

Figura 9: Tela para o professor gerar o código de acesso (Fonte: O próprio autor).

UC05 – Criar atividade

- **Ator Principal**: Professor
- **Descrição**: O professor acessa a opção "Criar Atividade" (Figura 10) e preenche um formulário com nome, e-mail, escola, turma, jogo selecionado, tempo da atividade, quantidade de fases e observações (Figura 11). Um link é gerado para compartilhamento com os alunos.
- **Pré-condições**:

- O professor está autenticado na plataforma.

- **Fluxo Principal de Eventos**:
  - O professor clica em "Criar Atividade".
  - O sistema exibe o formulário de criação.
  - O professor preenche os dados solicitados.
  - O sistema gera um link compartilhável para acesso à atividade.
- **Pós-condições**:
  - O link é enviado ao aluno, que pode acessar diretamente a atividade criada.

Figura 10: Tela para o professor acessar o painel de relatórios e criar atividades (Fonte: O próprio autor).

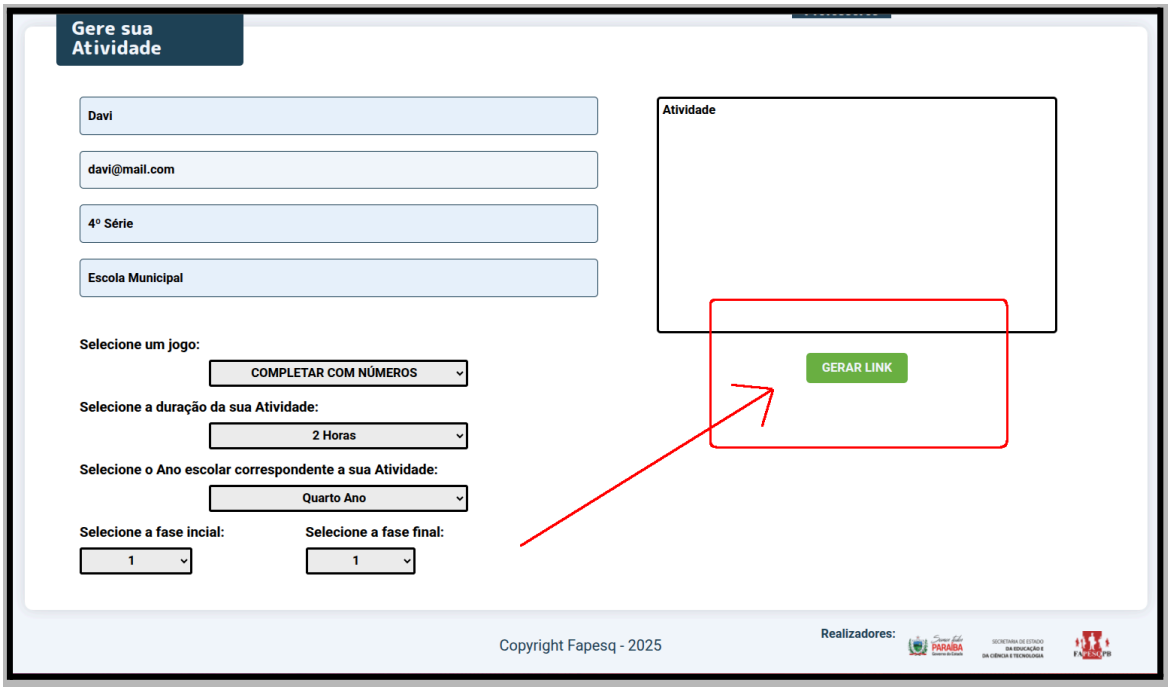

Figura 11: Tela de criação de atividades (Fonte: O próprio autor).

UC06 – Acessar relatórios de atividades

- **Ator Principal**: Professor
- **Descrição**: O professor acessa a opção "Acessar Relatório" (Figura 10) e visualiza os dados de desempenho dos alunos, como quantidade de acertos por jogo e atividade, organizados em tabelas, conforme exibido na Figura 12.
- **Pré-condições**:
  - O professor está autenticado na plataforma.
  - Pelo menos uma atividade já foi criada.
- **Fluxo Principal de Eventos**:
  - O professor clica em "Acessar Relatório".
  - O sistema exibe os dados referentes ao desempenho dos alunos.
- **Pós-condições**:
  - O professor tem acesso a métricas de desempenho para avaliação pedagógica.

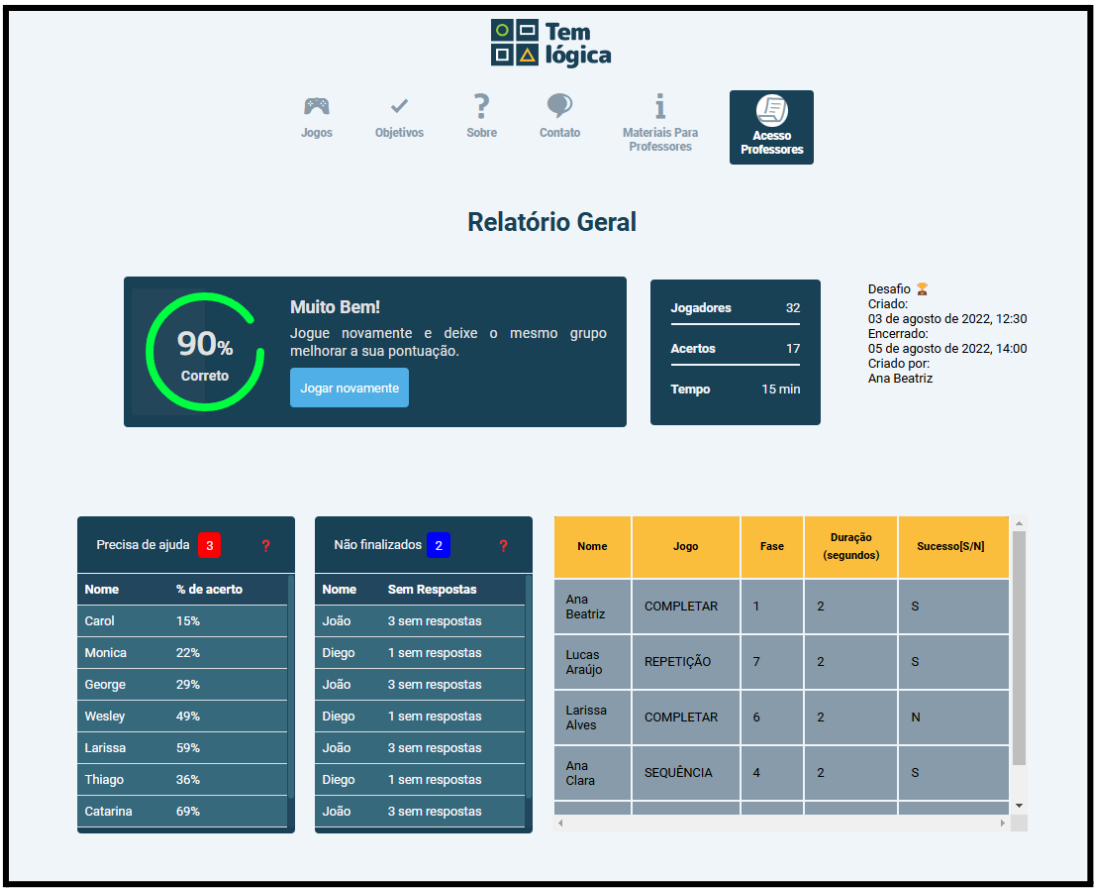

Figura 12: Tela de relatórios (Fonte: O próprio autor).

2.5 Arquitetura inicial proposta

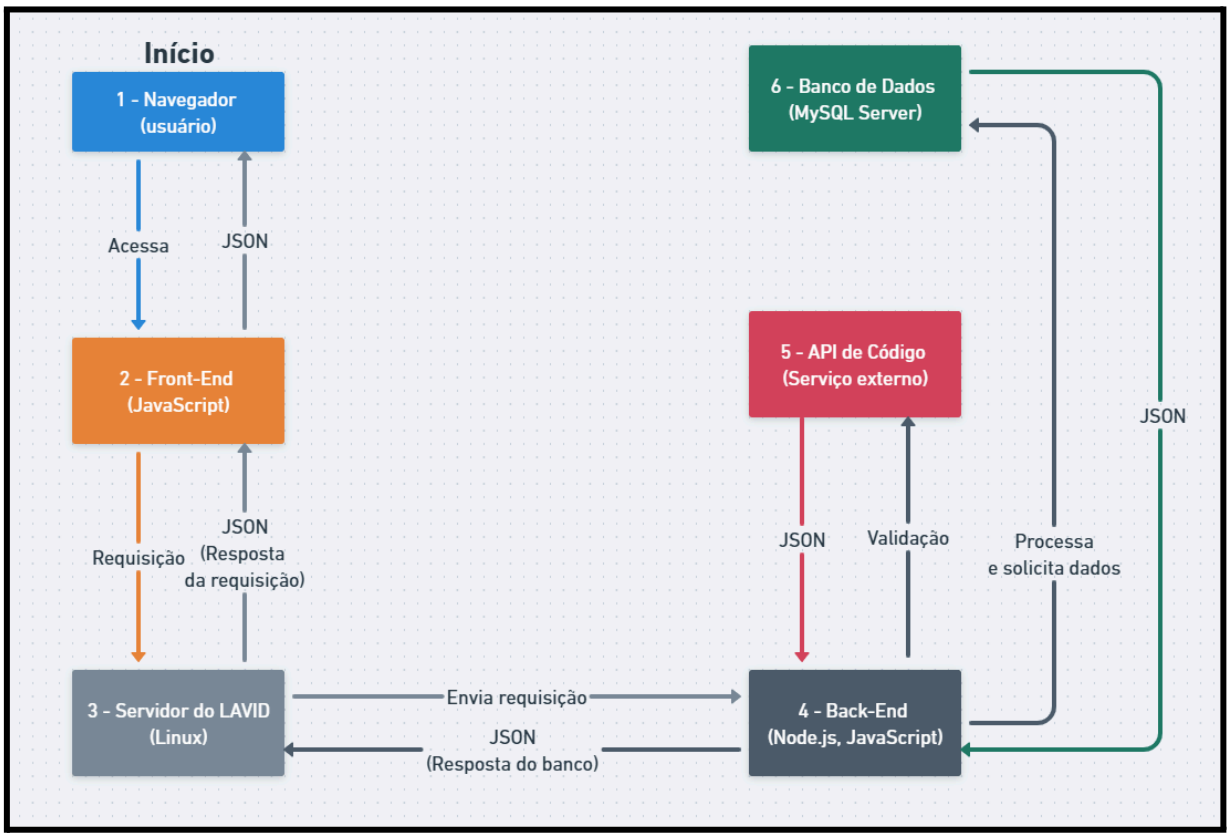

Figura 13: Arquitetura do fluxo de funcionamento do Tem Lógica (Fonte: O próprio autor).

A aplicação Tem Lógica é composta por diversas camadas, conforme ilustrado na Figura 13, que operam de forma integrada desde o *front-end* até o banco de dados.

A camada de *front-end* (bloco 2 da Figura 13) representa a interface visual da aplicação, sendo o ponto de interação direta com os usuários por meio de um navegador (bloco 1). Essa camada é responsável por capturar as ações do usuário e enviar requisições ao servidor para que essas ações sejam processadas.

O bloco 3 representa o local onde a plataforma se encontrava hospedada ainda em sua fase de desenvolvimento, esse servidor era responsável por estabelecer comunicações com outras camadas e repassar as requisições ao *back-end*.

A camada de *back-end* (bloco 4) processa as requisições recebidas do *front-end*, realiza a validação junto a serviços externos (bloco 5 – API de código) e se comunica com o banco de dados (bloco 6). Além disso, gerencia bibliotecas e dependências da aplicação e implementa as regras de negócio. Também é responsável por retornar ao *front-end* as respostas apropriadas, possibilitando ao usuário visualizar os resultados de suas interações.

O banco de dados armazena todas as informações essenciais da plataforma, como dados dos jogos, taxas de acertos e erros, horários das atividades, professores responsáveis e tempo destinado a cada tarefa.

2.6 Tecnologias envolvidas

Neste tópico são apresentadas as tecnologias empregadas no desenvolvimento da plataforma e algumas nomenclaturas usadas, além de sugestões de ferramentas e recursos compatíveis com a arquitetura recomendada, a ser discutida nos capítulos seguintes. Compreender essa base tecnológica é fundamental para contextualizar as decisões de arquitetura e justificar as propostas de refatoração e melhoria aplicadas ao sistema.

Serão abordadas as tecnologias de *front-end*, responsáveis pela interface e interação com o usuário, bem como as de *back-end*, que processam dados, integram serviços e garantem o funcionamento interno da aplicação. A análise destaca como cada tecnologia impacta a funcionalidade, desempenho e manutenibilidade da plataforma.

2.6.1 JavaScript

JavaScript (JS) é a linguagem de programação mais amplamente utilizada para o desenvolvimento web. É uma linguagem multiparadigma que oferece suporte a várias abordagens de escrita e organização de código. Além de ser interpretada, *JavaScript* possui tipagem dinâmica, o que significa que não exige definição explícita dos tipos de variáveis. Suporta estilos de programação imperativa, funcional e orientada a objetos, proporcionando grande flexibilidade aos desenvolvedores. O *JavaScript* surgiu com objetivo de proporcionar a criação de páginas Web interativas e dinâmicas.

Esta linguagem de programação tem como principal objetivo fornecer funcionalidades às plataformas web, desde a regra de negócio operando no processamento e armazenamento de dados em constante comunicação com o banco de dados, redirecionamento de usuários para as rotas corretas até o design e criação da interface que será visualizada pelo usuário. O *JavaScript* torna a aplicação utilizável, funcional e com recursos necessários que garantem o seu funcionamento.

No desenvolvimento da plataforma Tem Lógica, o *JavaScript* teve seu papel na criação de páginas web em conjunto com *HTML* e *CSS* fornecendo controle e interação, estilizações dinâmicas de botões de acordo com a ação efetuada pelo usuário. Além da criação da lógica de funcionamento de cada jogo presente na plataforma e envio de informações.

2.6.2 HTML

HTML (*HyperText Markup Language*) é a linguagem fundamental para a criação de páginas web, sendo composta por *tags* que definem a estrutura e o conteúdo dos documentos. Essa tecnologia é utilizada na camada *front-end* das aplicações web, pois define o esqueleto das páginas que são visualizadas e acessadas pelos usuários. *HTML* permite que elementos sejam organizados e localizados na interface, garantindo uma experiência de navegação intuitiva, facilitando a interação com os componentes na tela e servindo como base para a estrutura visual do sistema.

2.6.3 CSS

CSS (*Cascading Style Sheets*) é uma linguagem de estilo utilizada para controlar a apresentação visual de documentos *HTML*. Com o *CSS*, é possível definir o *layout*, as cores, as fontes e outros aspectos visuais de uma página web, separando o conteúdo da sua apresentação. Isso facilita a manutenção e o desenvolvimento de sites com uma aparência consistente e atraente. Esta ferramenta teve como principal objetivo a estilização da plataforma envolvendo as páginas, definição do padrão de cores utilizadas, botões, peças, imagens de cada jogo, fontes, *backgrounds*, espaçamento entre textos e outros.

2.6.4 MySQL

O *MySQL* é um *SGDB* ou Sistema de Gerenciamento de Banco de Dados Relacional (RDBMS), utilizando a linguagem *SQL* (*Structured Query Language*) para manipular dados. Na plataforma do Tem Lógica foi especificamente utilizado para o armazenamento de dados de alunos, professores, informações sobre os jogos, atividades criadas, relatórios e desempenho dos alunos.

2.6.5 SQL

A linguagem *SQL*, por sua vez, é utilizada para acessar e gerenciar bancos de dados relacionais, possibilitando a execução de consultas, inserção, atualização e exclusão de dados, bem como a administração de esquemas de banco de dados em *RDBMS* como *MySQL*, *PostgreSQL* e *SQL Server*. O *SQL* foi utilizado para realizar consultas ao banco de dados com o objetivo de recuperar informações a respeito dos jogos disponíveis na

plataforma, dados sobre alunos e professores, histórico de atividades e para eventuais geração de relatórios.

2.6.6 Node.js

O *Node.js* é uma plataforma de desenvolvimento que permite a execução de código *JavaScript* no lado do servidor, possibilitando a construção de aplicações web escaláveis e em tempo real. Ele é construído sobre o motor *V8* do *Google Chrome* e oferece um ambiente de execução leve e eficiente. Durante o desenvolvimento da plataforma Tem Lógica, o *Node.js* foi utilizado na camada *back-end* que é responsável por definir as regras de negócio do sistema, gerenciar a comunicação com o banco de dados, controlar o fluxo dos jogos e gerenciar o servidor, que processa as requisições feitas pelo cliente (navegador) e retorna as respostas adequadas, garantindo o funcionamento da lógica de aplicação e fornecendo dados para o *front-end*.

2.6.7 Banco de dados

Banco de dados é um sistema de armazenamento que permite a criação, o armazenamento e a manipulação de dados de forma eficiente. É utilizado para armazenar informações que podem ser acessadas, gerenciadas e atualizadas de forma rápida e segura. Dentro do Tem Lógica o banco de dados tem sua principal contribuição em servir como uma fonte de armazenamento de informações, onde é possível recuperar essas informações sempre que necessário. Seja para validar os dados de um usuário que está tentando se autenticar ou para exibir informações devido o tipo de requisição realizada.

2.6.8 GitHub

O *GitHub* é uma plataforma de hospedagem de código-fonte que permite aos desenvolvedores colaborar em projetos utilizando o controle de versão *Git*. Ele facilita o gerenciamento de código, possibilitando que várias pessoas contribuam e revisem alterações. Além de ser um repositório central para projetos, o *GitHub* oferece ferramentas para documentação e integração com outros serviços, promovendo o trabalho colaborativo em software. Durante o desenvolvimento do Tem Lógica o *GitHub* foi utilizado como a principal plataforma de versionamento de código.

2.6.9 GitHub Actions

GitHub Actions faz parte do GitHub, basicamente é uma funcionalidade que permite a automação de fluxos de trabalho para desenvolvimento de software. Com ela, é possível configurar pipelines de integração contínua e entrega contínua de código, automatizando tarefas como testes, compilações e deploys (processo de disponibilizar uma aplicação para uso em um ambiente). Isso ajuda a simplificar e acelerar processos repetitivos, melhorando a produtividade e a qualidade do código.

2.6.10 Netlify

Netlify é uma plataforma de hospedagem de sites e aplicações web. Ela facilita o processo de implantação e manutenção de sites, permitindo integração com repositórios de código para realizar deploys automáticos. A *Netlify* oferece também ferramentas para otimização de desempenho, segurança e configuração de funcionalidades avançadas.

2.6.11 JSON

JSON (*JavaScript Object Notation*), ou Notação de Objetos *JavaScript*, é um formato de dados amplamente adotado por sua simplicidade e facilidade de uso em diferentes plataformas. Ele organiza informações em pares de chave e valor, o que torna o armazenamento e a transmissão de dados diretos e intuitivos. Sua versatilidade permite que diversas linguagens de programação o utilizem sem necessidade de adaptação complexa. Em aplicações web, *JSON* desempenha um papel crucial na troca de dados entre o cliente e o servidor, facilitando a comunicação entre diferentes partes do sistema e garantindo que informações sejam enviadas e recebidas de maneira rápida e eficiente.

2.6.12 Deploy

O *deploy* (implantação) é o processo de disponibilizar uma aplicação ou sistema para que ele seja acessado e utilizado por usuários reais. Basicamente, é o momento em que o software, após ser desenvolvido e testado, fica disponível para acesso por meio de um site, aplicativo ou outra plataforma.

2.6.13 Integração Contínua e Entrega Contínua (CI/CD)

A integração contínua (CI) é uma prática de desenvolvimento de software em que as alterações no código feitas pelos desenvolvedores são frequentemente reunidas e testadas automaticamente para garantir que funcionem bem juntas. Já a entrega contínua (CD) leva esse conceito um passo além, automatizando o envio dessas alterações aprovadas para o ambiente de produção onde a aplicação está sendo disponibilizada para usuários finais, tornando o processo de entrega mais rápido, eficiente e confiável.

2.6.14 Index.js

O arquivo *index.js* é comumente utilizado como ponto de entrada principal em aplicações web e *back-end* desenvolvidas com *Node.js*. Nele, são realizadas as configurações iniciais do servidor, como o gerenciamento de rotas, o controle de conexões com banco de dados, a inicialização de middlewares e a definição das regras de negócio principais. O *index.js* é responsável por orquestrar a estrutura da aplicação, garantindo que todos os módulos e serviços necessários sejam carregados de maneira organizada e funcional para o correto funcionamento do sistema.

2.6.15 Local Storage

O *Local Storage* é uma tecnologia de armazenamento local disponível nos navegadores web, que permite guardar dados de maneira persistente no lado do cliente. As informações armazenadas no *Local Storage* permanecem disponíveis mesmo após o navegador ser fechado, possibilitando que aplicações web mantenham estados, preferências de usuário e pequenos volumes de dados sem a necessidade de comunicação constante com o servidor. É uma ferramenta útil para otimizar a experiência do usuário, proporcionando carregamento mais rápido e menos dependente de requisições externas.

2.6.16 Cookies

Cookies são pequenos arquivos de dados armazenados no navegador do usuário que

permitem a troca de informações entre o cliente e o servidor de maneira automática em cada requisição *HTTP* ou *HTTPS*. Eles são amplamente utilizados para gerenciar sessões de usuário, guardar preferências, autenticar acessos e personalizar a experiência de navegação. Por serem enviados em conjunto com as requisições, os cookies facilitam a manutenção de estados entre diferentes acessos e interações do usuário dentro de uma aplicação web. Essas informações ficam disponíveis do lado do *front-end* e do *back-end*.

2.6.17 Staging

Staging é um ambiente intermediário no processo de desenvolvimento de software, utilizado para validar e testar aplicações em condições semelhantes às do ambiente de produção. Ele permite que desenvolvedores e equipes de qualidade verifiquem se novas funcionalidades, correções e atualizações funcionam corretamente antes de serem disponibilizadas aos usuários finais.

2.6.18 Logs

Logs são registros automáticos gerados por aplicações para documentar eventos e atividades durante sua execução. Esses registros são fundamentais para o monitoramento do sistema, diagnóstico de falhas, auditoria de ações e análise de desempenho. Em ambientes de desenvolvimento e produção, os logs fornecem informações detalhadas sobre requisições, respostas, erros, alertas e comportamento do sistema, auxiliando na identificação rápida de problemas.

2.6.19 Git

Git é um sistema de controle de versão distribuído amplamente utilizado no desenvolvimento de software. Ele permite que desenvolvedores acompanhem mudanças no código-fonte, colaborem de forma eficiente e mantenham diferentes versões de um mesmo projeto. Com o *Git*, é possível criar ramificações (*branches*), mesclar alterações, reverter erros e trabalhar paralelamente em diferentes partes da aplicação.

2.6.20 Branch

Branch, ou ramificação, é um recurso do *Git* que permite a criação de linhas paralelas de desenvolvimento dentro de um mesmo projeto. Com as *branches*, é possível desenvolver novas funcionalidades, corrigir erros ou testar ideias sem interferir diretamente na versão principal do código. Após a validação das alterações, a *branch* pode ser integrada ao código principal.

2.6.21 Commit

Commit é uma ação do sistema de controle de versão *Git* que registra alterações feitas em arquivos do projeto. Cada commit representa um ponto no histórico de desenvolvimento, contendo uma mensagem descritiva sobre o que foi modificado, facilitando o rastreamento das mudanças ao longo do tempo. Os *commits* são essenciais para manter a organização do projeto, permitir reversões em caso de erros e promover a colaboração entre desenvolvedores.

2.6.22 Console

O *console* é uma interface de linha de comando que permite aos desenvolvedores interagir diretamente com o sistema, enviar comandos e visualizar mensagens de saída. Ele é amplamente utilizado durante o desenvolvimento para exibir mensagens de erro, executar scripts, inspecionar dados e monitorar o comportamento da aplicação em tempo real.

2.6.23 Endpoints

Endpoints são pontos de acesso a funcionalidades de uma *API* (Interface de Programação de Aplicações), permitindo a comunicação entre o cliente (*front-end*) e o servidor (*back-end*). Cada *endpoint* representa uma *URL* específica que executa determinada ação, como salvar dados, retornar informações ou atualizar registros.

2.6.24 Middlewares

Middlewares são funções intermediárias executadas durante o ciclo de vida de uma requisição *HTTP* em aplicações web. Elas atuam entre a recepção da requisição e a resposta enviada ao cliente, podendo realizar diversas tarefas como autenticação, validação de dados, registro de *logs* e controle de acesso.

2.7 Resultados preliminares

Durante a análise da aplicação Tem Lógica, antes da realização das refatorações propostas, foi possível identificar uma série de limitações que comprometem o desempenho, a organização estrutural e a evolução da plataforma. Esses problemas surgiram principalmente devido à ausência de um planejamento arquitetural consistente desde o início do projeto, o que levou a decisões técnicas que, com o tempo, se mostraram ineficazes frente ao crescimento da aplicação.

Um dos principais pontos observados foi a integração dependente entre as camadas de *front-end* e *back-end*, dificultando a separação de responsabilidades e tornando a aplicação mais suscetível a falhas em cadeia. A comunicação entre essas camadas era feita de forma direta, com pouca abstração, o que prejudicava a flexibilidade na introdução de novas funcionalidades.

Durante o desenvolvimento de novos jogos na plataforma, frequentemente não havia um modelo estruturado ou um padrão de código a ser seguido. A prática comum era reutilizar trechos de jogos já existentes, copiando diretamente seus códigos para novos arquivos. Embora isso agilizasse a criação de funcionalidades, também resultava em código duplicado, estrutura desorganizada e ausência de aplicação de padrões de projeto.

Adicionalmente, a constante requisição ao banco de dados a cada nova interação do usuário sobrecarregava a aplicação, resultando em instabilidade e lentidão perceptível durante o uso. Essas consultas frequentes não eram otimizadas e muitas vezes envolviam dados que poderiam ser armazenados temporariamente no navegador do usuário por meio de estratégias de *cache*, o que não foi implementado inicialmente.

Outro problema recorrente foi a necessidade de reinicializações manuais do servidor, em razão de falhas de conexão com o banco de dados. Isso demonstrava uma falta de mecanismos automáticos de recuperação ou de detecção de falhas, além da ausência de *logs* estruturados que pudessem facilitar a análise e correção de erros.

Do ponto de vista do código-fonte, foi identificado um número elevado de trechos duplicados e lógica redundante, especialmente no *front-end*, o que dificultava a manutenção e aumentava a possibilidade de erros em atualizações futuras. Além disso, havia um uso inconsistente de bibliotecas e *frameworks*, com versões desatualizadas e má configuração de dependências.

A aplicação também apresentava problemas na organização dos ambientes de desenvolvimento, testes e produção. Em muitos casos, o código era implantado diretamente

no ambiente final sem a devida validação prévia, o que comprometia a confiabilidade do sistema em operação.

Esses resultados preliminares evidenciaram que, embora a aplicação já estivesse funcional e disponível para uso, sua estrutura interna carecia de uma base sólida para permitir a manutenção contínua e a expansão futura.

3 PROCESSO DE REFATORAÇÃO

Durante o período em que participei ativamente do desenvolvimento da plataforma Tem Lógica, foi possível acompanhar uma série de evoluções e modificações no sistema. Entretanto, grande parte dessas mudanças ocorreram sem um planejamento estruturado, o que acabou acarretando problemas técnicos.

As primeiras falhas foram identificadas de forma empírica, principalmente durante testes manuais feitos em ambiente de homologação e, em alguns casos, em produção. As ocorrências mais críticas envolviam a queda do banco de dados, que exigia a reinicialização manual do servidor para que a aplicação voltasse a operar normalmente.

Esses erros apontavam um problema de integridade do sistema, resultado de uma combinação de fatores, incluindo estrutura de código pouco planejada, falta de monitoramento automatizado, e ausência de ambientes de teste e produção devidamente separados.

Além disso, durante a análise do código-fonte e ao realizar novas implementações, tornou-se evidente a presença de código duplicado, versões desatualizadas de bibliotecas e ausência de lógica unificada para controle de sessão e autenticação. Essas observações foram possíveis por meio de inspeções manuais de código, revisão de *commits* no repositório *Git* e acompanhamento de logs de erro gerados no servidor.

Embora não tenha sido utilizado um sistema formal de análise estática, ferramentas básicas como o *console* do navegador, *logs* do terminal e comparações entre *branches* no *GitHub* foram fundamentais para localizar trechos críticos e inconsistentes.

Os principais pré-requisitos adotados para iniciar a refatoração incluíram a definição de boas práticas mínimas, como:

- Modularização do código;
- Separação entre responsabilidades de *front-end* e *back-end*;
- Atualização das bibliotecas essenciais;
- Redução de redundância e padronização de nomenclaturas.

O objetivo dessa refatoração busca melhorar o código nos seguintes aspectos:

- Manutenibilidade: Permitindo que alterações futuras fossem feitas de forma mais segura e compreensível;

- Desempenho: Reduzindo a quantidade de requisições desnecessárias ao banco de dados;
- Estabilidade: Evitando falhas recorrentes de conexão;
- Escalabilidade: Preparando a aplicação para suportar novas funcionalidades sem comprometer seu funcionamento.

Em resumo, os problemas enfrentados não tinham origem única. Foram consequência de falhas durante seu desenvolvimento, envolvendo a estrutura do código e ausência de processos padronizados. A refatoração surgiu, portanto, como uma etapa essencial para estabelecer uma base técnica mais sólida e confiável para a evolução da plataforma.

3.1 Processo de identificação de erros

A identificação dos erros apresentados ocorreu durante a fase de testes funcionais da plataforma, realizados em ambiente de desenvolvimento local. As falhas foram detectadas a partir da navegação simulada no sistema, com a execução de interações típicas de um usuário comum, utilizando o perfil de aluno.

O processo foi conduzido manualmente, por meio da inspeção de elementos no navegador, análise de mensagens de erro exibidas no *console*, bem como a observação de *logs* gerados no servidor. Além disso, foram analisados trechos específicos do código-fonte e o histórico de versões nos repositórios *Git*, o que permitiu rastrear a origem de alguns problemas recorrentes.

Apesar da ausência de ferramentas automatizadas de teste ou integração contínua, a abordagem adotada foi suficiente para mapear os principais pontos críticos da aplicação, os quais são detalhados na seção seguinte. Esses problemas revelaram padrões de dependência entre componentes e fragilidades estruturais que impactam diretamente a estabilidade e a manutenibilidade do sistema.

3.1.1 Erros identificados

Durante a realização de testes com o perfil de aluno, foram identificados erros e comportamentos inesperados. Um exemplo recorrente ocorreu ao selecionar um ano na plataforma: a aplicação tentava acessar uma rota de *API*, mas falhava, pois não havia conexão ativa com o banco de dados, o *back-end* ou a própria *API* (figura 17).

Esse tipo de erro foi identificado em ambiente de desenvolvimento local, durante a verificação manual das funcionalidades básicas da plataforma.

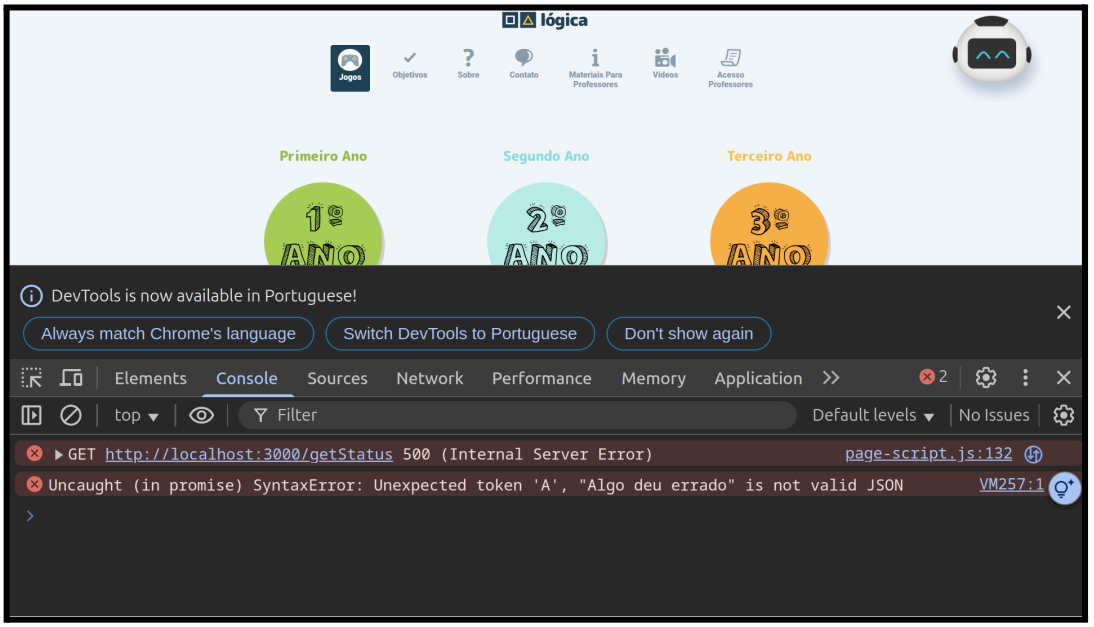

Figura 14: Falha emitida ao tentar escolher uma opção de ano (Fonte: O próprio autor).

Essas falhas acontecem devido à ausência de tratamento de exceções e ao acoplamento entre camadas, especialmente entre o *front-end* e os serviços de *back-end*. Isso significa que funcionalidades básicas da interface dependiam diretamente da resposta imediata do servidor e do banco de dados para renderizar corretamente, o que tornava o sistema frágil e pouco tolerante a falhas.

Quando, por exemplo, o banco de dados estava *offline* ou a *API* inacessível, a aplicação simplesmente quebrava, sem apresentar mensagens de erro amigáveis ou alternativas automáticas em caso de erro.

Além disso, algumas interações da interface, como o acesso ao jogo ilustrado na Figura 4 ou a tentativa de arrastar uma peça (Figura 5), não funcionavam adequadamente. Em alguns casos, mesmo quando as ações eram parcialmente concluídas, a aplicação emitia mensagens de erro, revelando dependências entre componentes.

No contexto desses testes, "acoplamento com o banco de dados" refere-se à dependência direta e imediata que a camada de interface possuía com a resposta do banco. Por exemplo, ao inicializar a aplicação sem uma instância do banco conectada, diversas funcionalidades não carregavam, e elementos visuais apareciam incompletos ou

desconfigurados (figura 15). Isso pode indicar que a renderização da interface não estava isolada da lógica de dados, o que é uma prática indesejada do ponto de vista arquitetural.

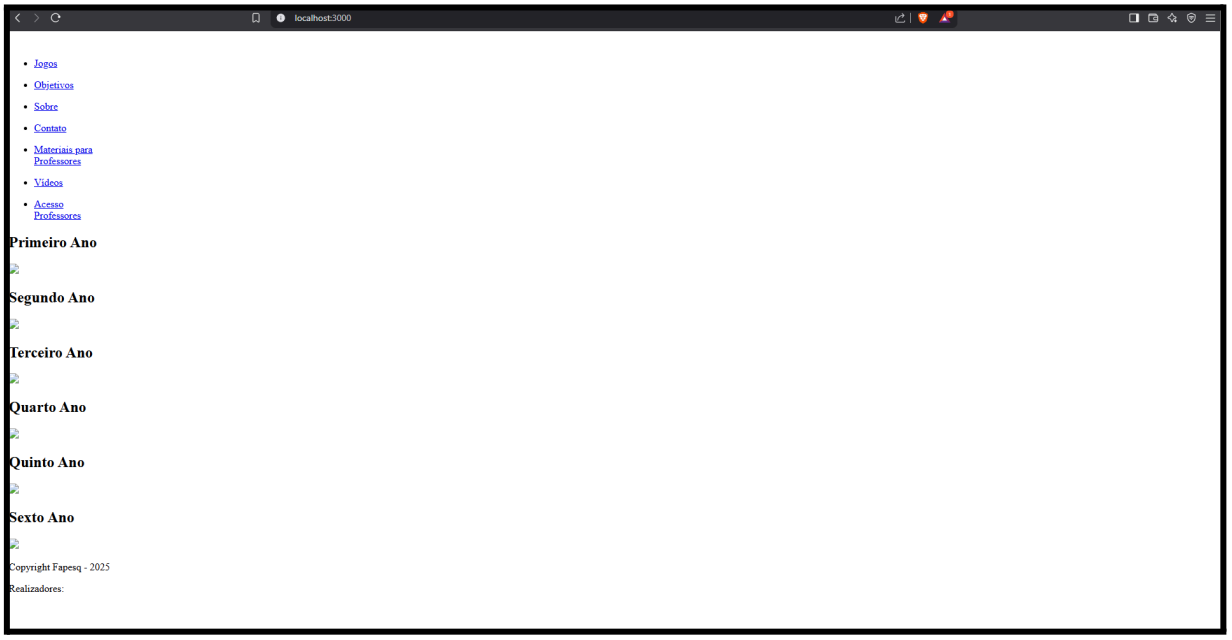

Figura 15: Tela inicial da plataforma sem conexão com o banco de dados (Fonte: O próprio autor).

Retomando a arquitetura da visão do aluno, ilustrada na Figura 1, observou-se que entre o bloco da “Página Inicial” e a “Escolha do Ano” existe uma conexão direta com as rotas do servidor. A tentativa de acessar uma categoria específica, como os anos escolares, gerava imediatamente uma requisição ao *back-end*, sem verificação prévia de estado ou tratamento de falhas. Um exemplo desse erro é o ilustrado na figura 14.

Esse comportamento reforça a hipótese de que o *front-end* estava acoplado à lógica de negócio, o que dificultava a modularidade e comprometia a resiliência da aplicação.

O termo "eficiência de comunicação", citado anteriormente, refere-se aqui à capacidade do sistema de responder de forma rápida, com o menor uso possível de recursos. No caso da plataforma, observou-se que a comunicação entre as camadas estava mal otimizada, com múltiplas requisições redundantes e sem controle adequado de estado, o que afetava negativamente a experiência fornecida pela plataforma.

Isso pode ser observado nas imagens apresentadas neste capítulo, nas quais funcionalidades básicas, como o acesso a um jogo, resultam em erros. Além disso, a tentativa de obter o código de autenticação do professor falha devido à ausência de comunicação com a *API* externa (Figura 20).

A identificação dos erros também foi facilitada pela análise de trechos de código, como os presentes nas Figuras 16 a 19. Por exemplo, a função *checkStatus* (Figura 17), chamada ao clicar em um ano (Figura 16), realiza uma requisição para */getStatus* (Figura 18), a qual redirecionava para outra função responsável pela validação da sessão do usuário (Figura 19), que por fim executava o método *sessao.getStatus*. Esse encadeamento direto entre elementos do *front-end* e funções do *back-end* demonstra a ausência de uma camada intermediária de abstração ou controle, reforçando o problema da integração camadas.

```
<section class="sub-section" id="section-sequencia-home">
    <h2 class="section-title" id="color1">Primeiro Ano</h2>
    <a onclick='checkStatus("Primeiro ano")' class="section-link" class="primeiro-ano" id="primeiro">
        <img class="section-link-img" src="img\svg\1.svg">
    </a>
</section>
```

Figura 16: Função *checkStatus* que verifica a situação da sessão do aluno (Fonte: O próprio autor).

```
async function checkStatus(ano){
  var anoAluno = ano;

  if(!anoAluno){
    console.log('não recebi o ano')
  }

  let resultado = await (await fetch('/getStatus'))
  resultado = await resultado.json();
  criarModalLogin(resultado.nome, anoAluno);
}
```

Figura 17: Função *checkStatus* (Fonte: O próprio autor).

```
routerDefault.get('/getstatus',(req,res) => {
    return res.json(sessao.getStatus(req))
})
```

Figura 18: Função de redirecionamento para verificação do status da sessão do usuário (Fonte: O próprio autor).

```
sessao.getStatus = (req) => {
  let log;
  req.session.logado == true ? log = {logado : true, ano: req.session.ano, nome:req.session.nome}:
  log = {logado : false, ano: req.session.ano};

  return log
}
```

Figura 19: Função que verifica o status da sessão (Fonte: O próprio autor).

A identificação desses pontos críticos foi essencial para compreender os gargalos da aplicação e fundamentar as soluções propostas, que visam reestruturar o sistema de forma mais modular, escalável e tolerante a falhas.

Durante os testes feitos na parte do sistema voltada para os professores, foi possível perceber que, ao informar seus dados — como nome e e-mail — e aguardar o código de acesso, esse código nem sempre chegava.

Isso acontecia porque o sistema não verificava corretamente se o código foi realmente enviado, não oferecia uma forma de reenviar automaticamente caso houvesse falha, nem conferia se o código salvo no sistema era o mesmo enviado por e-mail. A falta dessas checagens acabava prejudicando a confiabilidade do processo de *login* do professor.

Nas Figuras 20 e 21, podemos ver exemplos desses problemas:

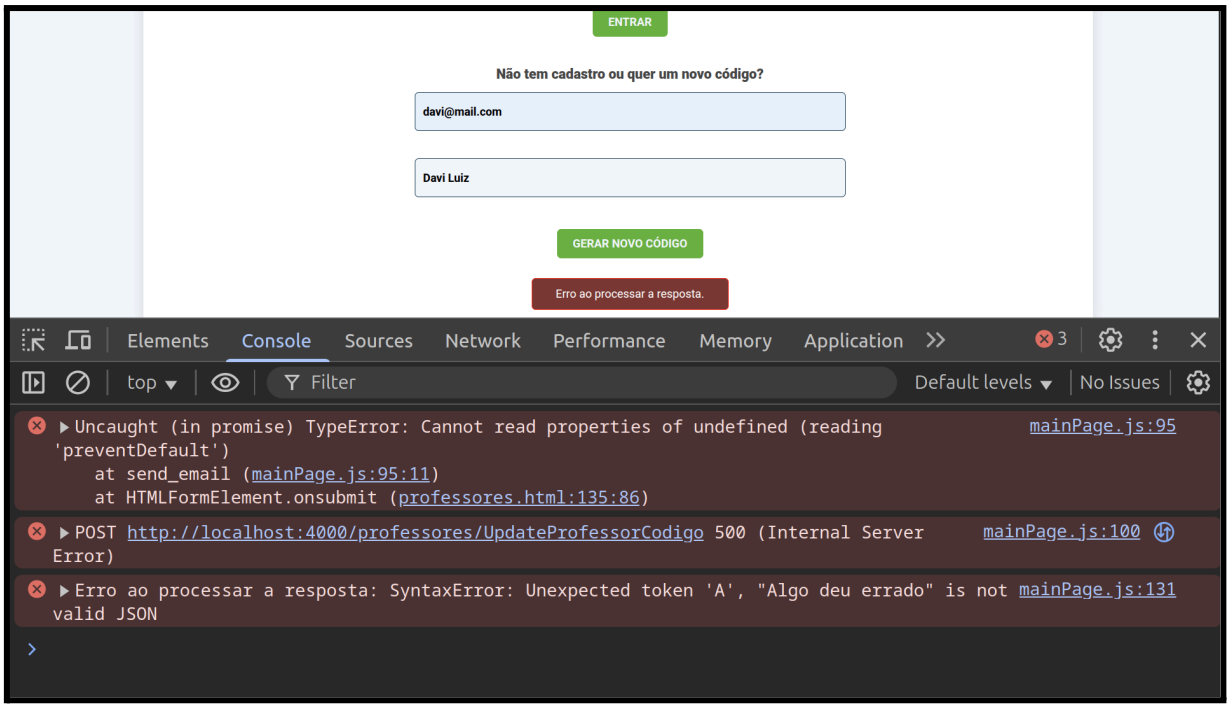

Figura 20: Erro gerado ao tentar criar um código de acesso (Fonte: O próprio autor).

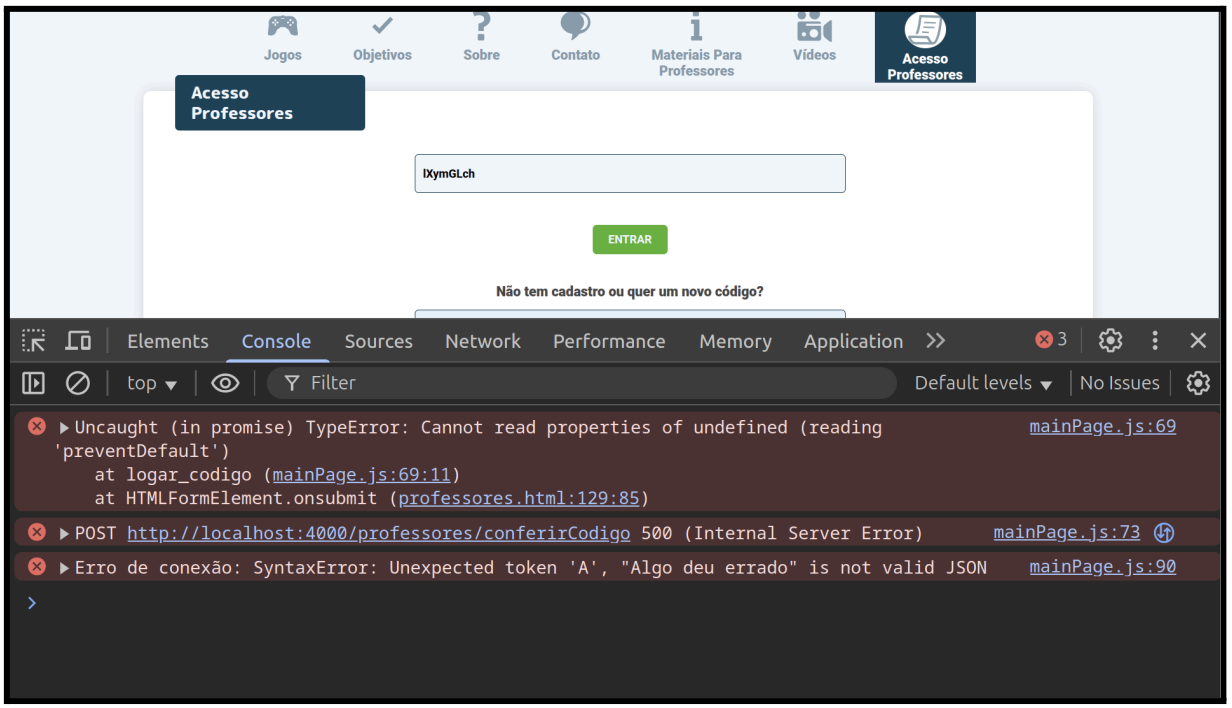

Figura 21: Erro gerado ao tentar entrar na plataforma do professor com o código de acesso (Fonte: O próprio autor).

Na primeira (Figura 20), o sistema mostra um erro indicando que não conseguiu se conectar com o servidor, o que sugere que o sistema estava fora do ar ou com problemas técnicos. Já na segunda (Figura 21), o erro indica que a conexão com o banco de dados falhou, e também que as regras de verificação do código não funcionaram corretamente. Isso impedia o professor de acessar a plataforma e realizar tarefas como criar atividades ou visualizar relatórios dos alunos.

A Figura 22 mostra um trecho do código da plataforma que apresenta alguns erros importantes. Um dos problemas está no uso da função *form.addEventListener('submit', event.preventDefault());*. Essa linha deveria impedir que o formulário fosse enviado automaticamente, mas foi escrita de forma incorreta. O *preventDefault()* está sendo executado imediatamente, quando na verdade ele deveria ser chamado apenas quando o botão de envio fosse realmente clicado. Isso faz com que o comportamento esperado do formulário não funcione corretamente.

Outro problema está no tratamento da resposta do servidor. Quando o servidor responde com erro, a função *setMsgResponse(response)* é chamada diretamente, sem verificar qual foi o erro nem mostrar uma mensagem explicativa para o usuário ou para o desenvolvedor que estiver realizando manutenção e testes na plataforma. Isso torna difícil saber o que exatamente de errado aconteceu.

Por fim, na última linha do código, há uma tentativa de usar o *dados.text()*. No entanto, a variável dados nunca foi definida no código, o que provavelmente causará uma falha na execução do sistema. O correto seria usar *response.text()*, já que *response* é a variável que armazena a resposta do servidor. Esse tipo de erro pode fazer com que o sistema trave ou simplesmente não demonstre nenhuma informação no *console*, dificultando a correção por parte do desenvolvedor.

Figura 22: Bloco com verificação incorreta do código gerado para o professor (Fonte: O próprio autor).

A Figura 23 apresenta uma versão mais elaborada e bem estruturada do código responsável por enviar as informações do formulário de *login* do professor. O processo começa com o uso da função *event.preventDefault()*, que impede o envio automático do formulário. Isso evita que a página recarregue, permitindo que o restante do código seja executado de forma mais controlada e eficiente.

Em seguida, a variável *codigo*, que representa o código digitado pelo professor, é capturada diretamente do campo do formulário. Esse valor é então enviado ao servidor utilizando o método *fetch* com o tipo *POST*, que é o método usado para enviar dados a outras partes do sistema (neste caso, ao *back-end* da plataforma). Os dados são organizados no formato *JSON*, que é um padrão muito comum para trocar informações entre sistemas.

Depois que a requisição é feita, o código verifica a resposta do servidor. Se o *status* da resposta for 200, isso significa que tudo correu bem e, nesse caso, o professor é redirecionado para a página apropriada. Se o *status* não for esse, o sistema reconhece que houve um erro e uma mensagem é mostrada ao usuário.

Por fim, o código também utiliza um bloco *try catch*, que serve para capturar e tratar qualquer erro inesperado, como falhas na rede ou respostas malformadas. Isso garante que, mesmo que algo dê errado, o sistema possa reagir de forma controlada, sem travar ou causar comportamentos confusos para o usuário.

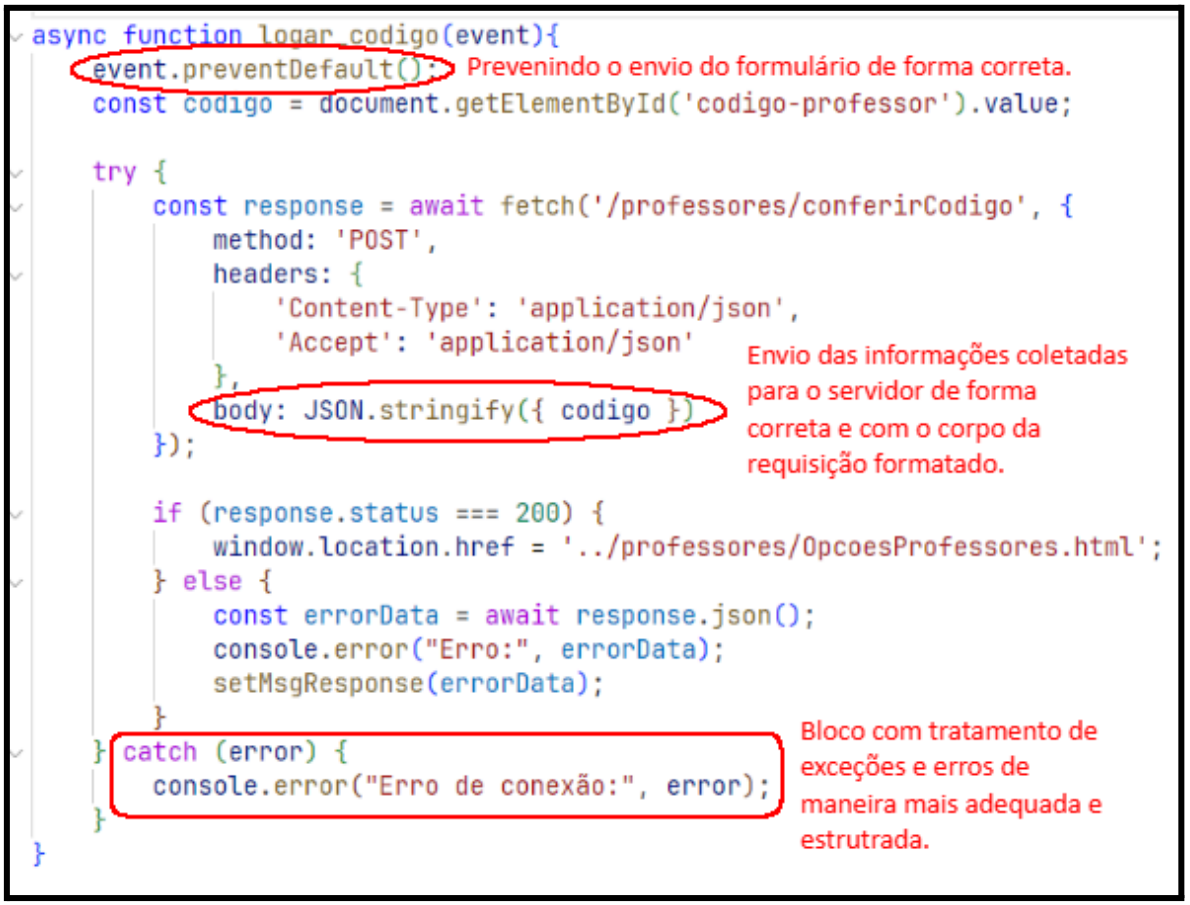

Figura 23: Código com verificação correta do código gerado para o professor (Fonte: O próprio autor).

É importante destacar que, devido a falhas técnicas presentes na funcionalidade do fluxo do professor, não foi possível realizar testes diretos e completos dessa funcionalidade durante o desenvolvimento deste trabalho. Essas limitações técnicas impediram a execução plena do fluxo esperado, tornando inviável a validação prática por meio da aplicação em funcionamento.

Como consequência, a identificação e análise dos erros relacionados a essa funcionalidade foram restritas ao estudo do código-fonte e à simulação do fluxo previsto. Essa abordagem permitiu documentar detalhadamente como o processo ideal deveria ocorrer, garantindo uma compreensão clara e estruturada do funcionamento esperado da plataforma.

3.2 Soluções candidatas

Nesta seção, serão apresentadas as soluções candidatas elaboradas a partir da identificação e análise dos principais problemas enfrentados pela plataforma Tem Lógica. Para isso, foi conduzida uma pesquisa técnica detalhada sobre as camadas que compõem o sistema — *front-end*, *back-end* e banco de dados — com o objetivo de diagnosticar os pontos críticos de acoplamento e outras fragilidades que comprometem a manutenção e evolução da plataforma.

Durante esse processo, foram identificadas algumas falhas, como problemas na autenticação de usuários, lentidão no desempenho da aplicação, erros na lógica de programação e dificuldades de integração entre as camadas. Um dos principais obstáculos observados foi o acoplamento entre o *front-end*, o *back-end* e o banco de dados, que exigia a execução simultânea de todas essas partes para possibilitar alterações ou testes, restringindo a autonomia do desenvolvimento e aumentando a complexidade da manutenção.

Como resposta a esse cenário, foram propostas e implementadas estratégias de desacoplamento entre as camadas, permitindo que cada uma opere de forma mais independente. Essa abordagem foi acompanhada por um processo de refatoração do código, com o intuito de reorganizar a estrutura do sistema sem comprometer suas funcionalidades existentes.

Complementando essas ações, foram aplicados métodos de teste e validação, com o objetivo de assegurar a eficácia das soluções e garantir que as melhorias introduzidas não comprometam o funcionamento da plataforma. Tais medidas contribuíram para aumentar a estabilidade, a segurança e a escalabilidade do sistema, tornando-o mais robusto e sustentável a longo prazo.

3.2.1 Diagnóstico e propostas de melhorias

A metodologia adotada nesta etapa baseou-se na identificação e análise de erros observados durante a execução de testes na plataforma. O objetivo principal foi diagnosticar falhas recorrentes, compreender suas causas e aprofundar a investigação técnica para identificar a raiz dos problemas. A partir desse diagnóstico, foram elaboradas propostas de melhorias, tanto no nível do código-fonte e das funcionalidades quanto na própria arquitetura da aplicação.

Os testes foram realizados em um ambiente de desenvolvimento local, envolvendo todas as camadas da aplicação. Para isso, utilizou-se uma máquina com sistema operacional *Windows 11 Pro*, configurada para simular o funcionamento completo da plataforma.

Cada teste foi conduzido com a participação direta do autor deste trabalho. As sugestões de melhorias apresentadas foram fundamentadas em pesquisas acadêmicas e em boas práticas reconhecidas na engenharia de software, tais como:

- Separação de responsabilidades, visando a modularização do sistema para facilitar manutenção e evolução;
- Refatoração incremental, respeitando a integridade funcional do sistema durante as alterações estruturais;
- Aplicação de testes manuais, conforme os princípios do Test-Driven Development (TDD), para garantir estabilidade e prevenir regressões;
- Adoção de arquitetura em camadas, visando o desacoplamento entre a interface do usuário, a lógica de negócio e a persistência de dados;
- Adoção de princípios de design limpo (Clean Code), visando a legibilidade, simplicidade e clareza do código, com nomes descritivos, funções curtas e baixa complexidade ciclomática.

Essas práticas forneceram o embasamento técnico necessário para propor soluções alinhadas às necessidades da plataforma e aderentes aos princípios de qualidade em desenvolvimento de software, como manutenção, escalabilidade e desempenho.

3.2.2 Diagnóstico do acoplamento entre as camadas da arquitetura

Na arquitetura atual da plataforma Tem Lógica, um dos principais desafios identificados é o acoplamento inadequado entre o *front-end* e o *back-end*, caracterizado principalmente pela inserção de lógica de *back-end* diretamente em arquivos do *front-end*, o que fere os princípios de separação de responsabilidades e modularidade. Essa prática compromete a manutenção do código, dificulta a evolução da plataforma e pode gerar dependências indesejadas entre componentes que deveriam operar de forma isolada.

Além disso, observou-se uma dependência de um banco de dados *MySQL* instalado localmente, sem a disponibilidade de uma instância mínima em ambiente remoto para testes realistas. Isso limita a capacidade de realizar testes distribuídos, dificulta o trabalho colaborativo entre desenvolvedores e aumenta a complexidade na configuração do ambiente de desenvolvimento.

Atualmente, a aplicação, o *back-end* e o banco de dados funcionam juntos em um único ambiente local, como ilustrado na Figura 24. Nessa configuração, o *front-end* — construído com *HTML*, *CSS* e *JavaScript* — possui trechos de código que fazem chamadas diretas ao servidor *back-end* desenvolvido em *Node.js*. Por exemplo, quando o usuário realiza alguma ação, uma função como *postData()* é acionada, enviando uma solicitação para o servidor.

O *back-end*, ao receber essa solicitação, se conecta diretamente ao banco de dados *MySQL* que também está instalado localmente, realizando operações como salvar ou buscar informações. Após isso, o resultado é devolvido ao *front-end*, que apresenta a resposta ao usuário — informando, por exemplo, se a operação deu certo ou não.

Apesar de funcional, essa configuração local é a única disponível para testes e validações, não existindo ambientes intermediários (como *staging* ou homologação) que possibilitem avaliar as novas implementações em condições mais próximas da produção. Essa limitação compromete a confiabilidade dos testes, pois impede a identificação de falhas que só se manifestam em contextos reais de uso, dificultando a validação segura de funcionalidades antes da publicação em ambiente de produção.

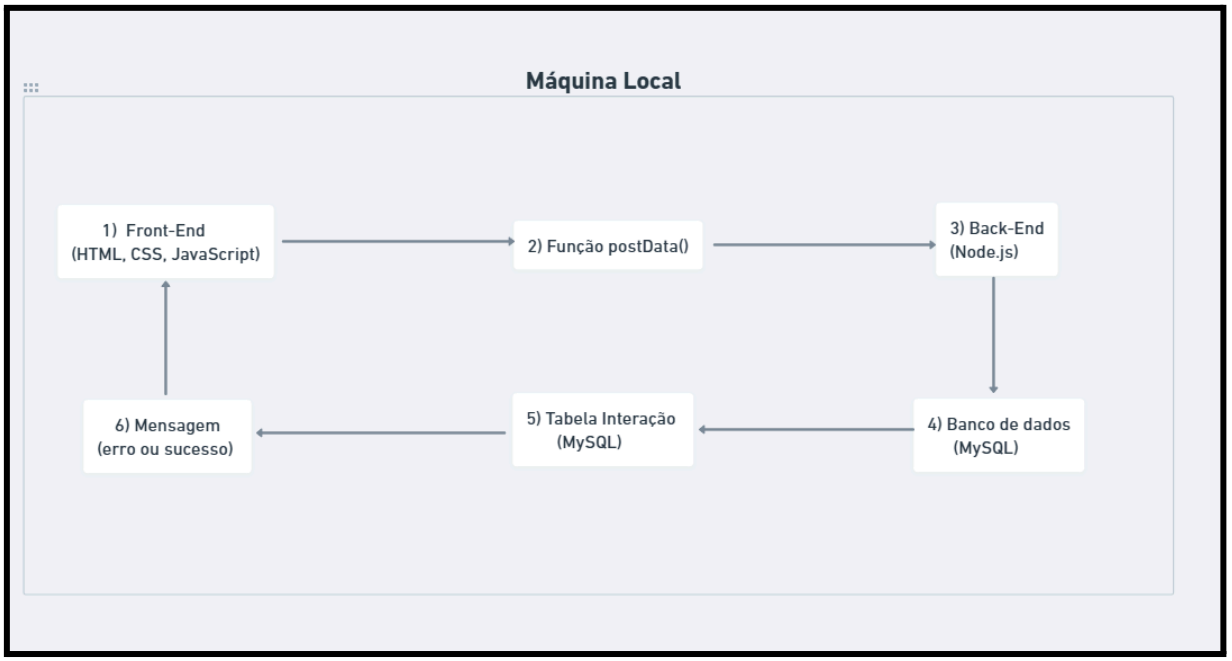

Figura 24: Arquitetura da aplicação Tem Lógica simulando a ativação de uma função rodando em ambiente local (Fonte: O próprio autor).

Essa configuração apresenta algumas desvantagens, principalmente relacionadas à forma como os componentes do sistema estão integrados e ao ambiente limitado de testes. A necessidade de instalar e configurar o banco de dados localmente dificulta o processo de entrada para novos colaboradores e pode gerar inconsistências entre os ambientes de desenvolvimento, especialmente quando há diferenças na versão do banco ou nas permissões de acesso.

Além disso, observa-se que há uma dependência entre o *front-end* e o *back-end*, com trechos de lógica de negócio e de acesso ao banco de dados inseridos diretamente nos arquivos de interface. Embora a arquitetura não seja formalmente monolítica, essa interligação entre camadas prejudica a separação de responsabilidades, dificultando a manutenção e a evolução da plataforma. Isso faz com que falhas ou modificações em uma camada impactem diretamente outras partes do sistema, tornando o desenvolvimento mais suscetível a erros.

Outro ponto crítico é a ausência de um ambiente remoto para testes e validação. Toda a aplicação depende exclusivamente do ambiente local de cada desenvolvedor, sem uma infraestrutura de homologação que simula o comportamento real da plataforma em produção. Essa limitação dificulta a identificação antecipada de falhas e o teste de novas funcionalidades em condições mais próximas ao uso real, o que compromete a qualidade das entregas.

Embora o processamento local possa oferecer maior velocidade de execução em muitos casos, essa abordagem também apresenta desafios quando se trata de simular cenários de carga, concorrência e conectividade, aspectos essenciais em aplicações educacionais com múltiplos usuários. A inexistência de um banco de dados remoto, ainda que com baixo desempenho, impede testes mais realistas e impede a preparação para uma possível transição para ambientes produtivos mais robustos.

Em resumo, a interdependência entre as camadas da aplicação e a falta de um ambiente de testes mais robusto dificultam tanto a manutenção quanto a escalabilidade da plataforma Tem Lógica. Essas limitações reforçam a importância de evoluir a arquitetura da aplicação para uma estrutura mais desacoplada e independente, segura e preparada para o crescimento da base de usuários.

3.2.3 Local do acoplamento entre o back-end e o front-end

A utilização do *Node.js* foi essencial para estruturar o *back-end* da plataforma, possibilitando o gerenciamento de rotas e o processamento de requisições oriundas tanto do *front-end* quanto do banco de dados. Com essa tecnologia, foram implementadas funcionalidades como autenticação de usuários, recuperação de códigos de acesso, cadastro de usuários, criação de atividades e consulta de informações armazenadas.

Após a criação das rotas e integração com o banco de dados, os dados gerados pela aplicação passaram a ser registrados e consultados diretamente por meio de requisições sincronizadas entre *front-end* e *back-end*. No entanto, a arquitetura adotada estabeleceu uma relação direta entre os arquivos de interface e a lógica de negócio, com a camada de apresentação chamando funções *back-end* diretamente a partir do *front-end*. Esse padrão comprometeu a separação de responsabilidades.

O acoplamento se manifesta especificamente na forma como o *front-end* depende do comportamento exato das rotas do *back-end*, sem uma camada intermediária ou abstração. Como resultado, alterações em *endpoints* ou na estrutura de dados exigem mudanças imediatas no *front-end*, tornando o desenvolvimento mais sensível a falhas e dificultando a manutenção evolutiva.

Essa dependência rígida entre as camadas impacta diretamente a escalabilidade da aplicação, além de aumentar o esforço necessário para testes, refatorações e integração de novas funcionalidades. A identificação desse ponto crítico evidenciou a necessidade de modularização, promovendo uma arquitetura mais flexível, capaz de desacoplar as camadas

por meio de boas práticas como a adoção de contratos (*APIs* bem definidas), controle de versionamento e utilização de camadas intermediárias (*middlewares*) para abstrair regras de negócio.

3.2.4 O que contribuiu para o acoplamento entre front-end e back-end?

O acoplamento identificado entre o *front-end* e o *back-end* da plataforma Tem Lógica teve como principal causa a comunicação direta entre essas camadas. Durante a execução dos jogos, informações como horários de início e término das atividades, taxas de erro e acerto, entre outros dados de desempenho, eram enviadas diretamente do *front-end* para rotas do *back-end* via requisições *POST*. Esse modelo contribuiu para a forte dependência estrutural entre as partes do sistema.

Outro fator relevante foi a execução local tanto do *back-end* quanto do banco de dados, sem *deploy* em servidores ou plataformas que permitissem sua operação independente. A ausência de ambientes separados resultou em uma arquitetura rígida e fortemente interligada, dificultando a modularidade e comprometendo a escalabilidade da aplicação.

A introdução da integração entre as camadas — antes inexistente — intensificou essa dependência. O *back-end* passou a processar os dados recebidos dos jogos antes de armazená-los no banco de dados, o que, sem o suporte de ambientes distribuídos ou máquinas virtuais, aumentou o impacto do acoplamento.

Além disso, a ausência de mecanismos automatizados para atualizar e publicar novas versões da plataforma dificultou o controle do que era testado e do que era disponibilizado para os usuários. Sem esse tipo de organização, tornou-se mais difícil manter versões separadas para testes e para uso real, o que também atrapalhou a separação adequada entre as diferentes partes do sistema.

Um exemplo prático dessa situação é a função *postData* apresentada anteriormente, responsável por enviar ao *back-end* informações como a quantidade de vezes que peças foram arrastadas, acertos dos usuários, horários de início e fim das atividades, além do monitoramento das áreas onde as peças eram colocadas nos jogos. Cada uma dessas interações gerava consultas ao banco de dados, elevando a carga sobre o sistema e comprometendo seu desempenho.

A funcionalidade de arrasto de peças, presente em todos os jogos, tornou-se um ponto crítico: cada movimento do usuário resultava em uma nova requisição ao banco de dados, sobrecarregando o sistema. Embora a exclusão dessa funcionalidade pudesse aliviar o banco,

tal mudança afetaria negativamente a experiência do usuário e a proposta pedagógica da plataforma.

Para mitigar essas limitações, recomenda-se uma reformulação da arquitetura. Isso inclui a separação entre *front-end* e *back-end* em ambientes independentes, desde a separação concreta de códigos elaborando com clareza qual código pertence a qual camada. Deploys em plataformas como *AWS*, *Vercel* ou *Google Cloud*, a implementação de *APIs* para modularizar a comunicação entre as camadas, o uso de banco de dados remoto para eliminar a dependência da máquina local em testes que demandam casos reais e a adoção de pipelines de entrega contínua de código para facilitar a atualização do sistema.

Adicionalmente, sugere-se o armazenamento temporário dos dados diretamente no navegador do usuário durante a execução das atividades, com envio posterior ao banco de dados ao final de cada interação. Essas medidas visam reduzir a dependência e integração entre as camadas com o objetivo de melhorar o desempenho e garantir maior escalabilidade à plataforma.

4 RESULTADOS

Este capítulo apresenta os principais resultados obtidos a partir da análise da plataforma Tem Lógica. São descritas as observações realizadas sobre sua estrutura técnica, usabilidade e funcionalidades, com base em testes e exploração prática do sistema. A partir dessa análise, foram identificados pontos críticos e elaboradas propostas de melhoria.

Em uma arquitetura ideal para a plataforma, todas as camadas — *front-end*, *back-end* e banco de dados — devem operar de forma independente, permitindo que alterações ou atualizações em uma delas não impactem diretamente as demais. Essa separação entre as camadas é fundamental para facilitar a manutenção, aumentar a performance e garantir um desempenho consistente da aplicação. As melhorias e sugestões apresentadas a seguir buscam não apenas resolver os problemas identificados, mas também preparar a plataforma para demandas futuras.

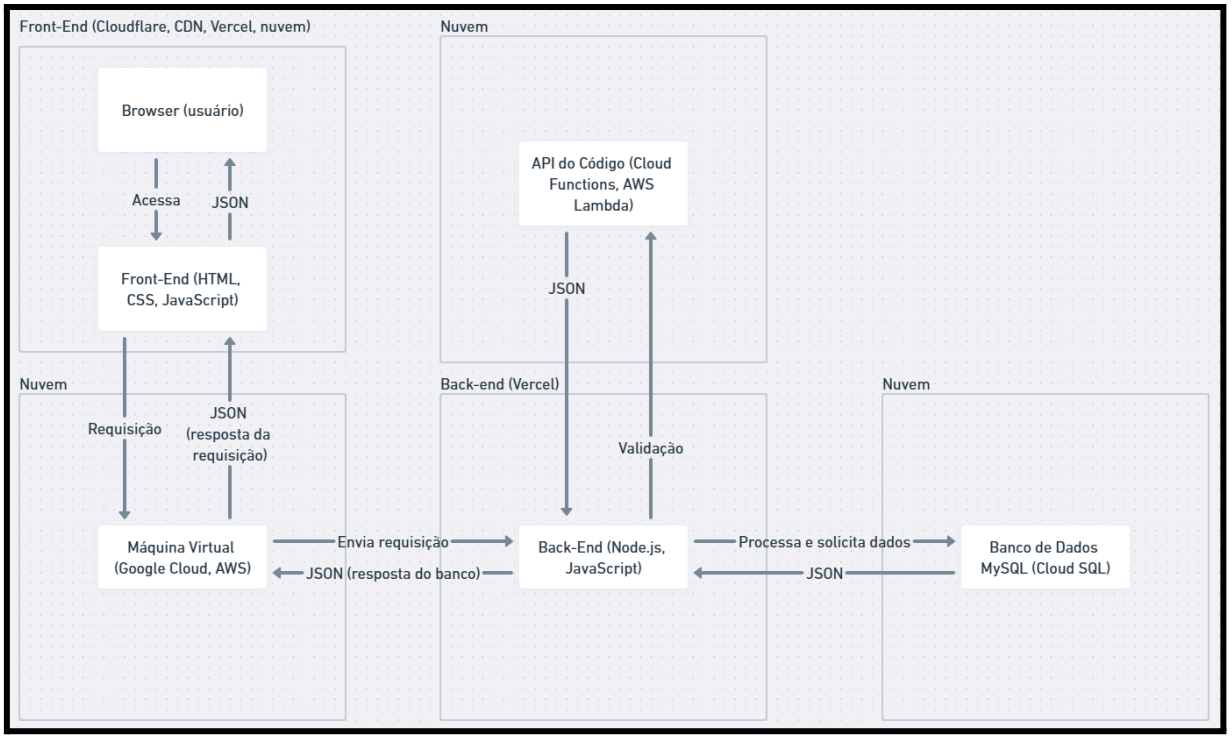

Figura 25: Arquitetura ideal da aplicação Tem Lógica (Fonte: O próprio autor).

A arquitetura ideal está representada na Figura 25. Nela, o *front-end* pode ser hospedado em plataformas como *Netlify*, *Vercel* ou *Cloudflare*, separadamente do *back-end* e do banco de dados. A independência entre camadas, neste contexto, refere-se à capacidade de cada uma operar, ser atualizada ou mantida sem que isso exija mudanças nas demais. Ou seja,

uma camada não deve depender diretamente da lógica interna ou da infraestrutura das outras para funcionar.

Atualmente, a camada de *front-end* está acoplada ao *back-end* não apenas pelo uso direto das rotas de comunicação para envio de dados, mas também pela ausência de uma separação clara entre os trechos de código responsáveis por cada camada. As requisições ao servidor são configuradas e executadas diretamente nos arquivos da interface, o que compromete a modularidade e dificulta a manutenção.

Da mesma forma, o *back-end* depende de uma instância local do banco de dados, reflexo da inexistência de um ambiente adequado para testes e simulações de funcionalidades em condições reais de uso. Em uma arquitetura aprimorada, o banco de dados deveria estar hospedado em uma solução de nuvem, como o *Cloud SQL* do *Google Cloud*, o que permitiria conexões padronizadas, mais robustas e independentes das máquinas locais. Essa estrutura mais modular garantiria maior flexibilidade, escalabilidade e facilidade de manutenção do sistema.

Além da separação entre camadas, é essencial também isolar funcionalidades que hoje contribuem para o acoplamento da aplicação. Um exemplo é o sistema de pontuação do jogador: atualmente, os pontos são enviados em tempo real para o *back-end* e já armazenados no banco de dados. Uma alternativa mais eficiente seria manter temporariamente essas pontuações no *front-end* e realizar a sincronização com o *back-end* apenas em momentos específicos. Por exemplo, o sistema poderia guardar localmente as últimas 10 pontuações do usuário, permitindo consultas rápidas sem depender do servidor a todo momento.

Essa mesma lógica poderia ser aplicada ao avanço de fases. Anteriormente, as fases eram gerenciadas no próprio *front-end*, eliminando a necessidade de comunicação constante com o *back-end*. Retomar essa abordagem, mesmo que parcialmente, poderia reduzir significativamente a quantidade de requisições feitas ao servidor, otimizando a performance da plataforma.

A plataforma enfrenta desafios decorrentes da forma como as camadas foram estruturadas e interligadas. O problema não está no fato de trabalhar localmente, pois é perfeitamente possível utilizar servidores locais em desenvolvimento. No entanto, o que se observou foi a falta de ambientes adequados para testes e validações de funcionalidades fora do ambiente de produção, o que resultava em dependências técnicas desnecessárias.

Por exemplo, para testar uma funcionalidade isolada do *back-end*, era obrigatório manter o banco de dados local ativo, pois o código não previa simulações ou dados fictícios. Além disso, como o *front-end* e o *back-end* compartilhavam arquivos e configurações, não

havia uma separação adequada entre suas responsabilidades, o que dificultava testes independentes de cada camada.

Esse modelo de acoplamento direto e ausência de ambientes de *staging* ou homologação compromete o fluxo de trabalho: as funcionalidades eram testadas localmente e, em seguida, já publicadas diretamente em produção. Sem ferramentas reais de teste e monitoramento intermediário, algumas atualizações foram disponibilizadas com erros ou comportamentos inesperados.

Uma solução viável seria a adoção de ambientes específicos para cada camada da aplicação, hospedando o *front-end*, o *back-end* e o banco de dados em serviços separados. Além disso, seria importante utilizar ferramentas que automatizam o processo de testes e publicação, como o *GitHub Actions*.

Com esse tipo de recurso, toda vez que uma nova funcionalidade for criada ou modificada, o sistema pode executar testes automáticos para verificar se está tudo funcionando corretamente, antes mesmo que a atualização chegue aos usuários. Esse processo, conhecido como *CI/CD* (Integração Contínua e Entrega Contínua), ajuda a evitar erros na versão final da plataforma, garantindo mais segurança, estabilidade e facilidade para aplicar futuras melhorias.

4.1 Modificações para evitar o acoplamento entre camadas

Durante o processo de desenvolvimento da plataforma, identificou-se que algumas funcionalidades estavam fortemente acopladas entre si, dificultando a realização de testes, ajustes e melhorias em partes isoladas do sistema. Com base nisso, foi realizada uma análise no código-fonte para localizar pontos em que funções, chamadas de dados e elementos de interface estavam excessivamente dependentes uns dos outros.

A partir dessa análise, foram aplicadas modificações estratégicas com o objetivo de reduzir esse acoplamento. Foram utilizados blocos de controle e variáveis específicas que permitissem o uso de dados simulados ou locais durante o desenvolvimento e os testes. Com isso, o *front-end* passou a operar em determinados contextos sem depender diretamente de respostas do *back-end* ou do banco de dados.

Essas mudanças trouxeram mais flexibilidade ao processo de desenvolvimento, permitindo, por exemplo, que determinadas funcionalidades fossem validadas isoladamente antes de serem integradas ao sistema completo. A diminuição da dependência direta entre as

camadas facilitou a manutenção do código, tornando mais simples corrigir erros ou adaptar partes da plataforma sem comprometer outras funcionalidades.

4.2 Passo a passo realizado

Como parte das melhorias implementadas na plataforma, foi necessário desenvolver uma maneira de identificar se o banco de dados está ativo, permitindo que o sistema se adapte mesmo em situações em que ele esteja fora do ar. O primeiro passo foi criar um mecanismo de comunicação entre o *front-end* e o *back-end*, capaz de verificar essa condição.

A solução encontrada foi o uso de *cookies*. *Cookies* são pequenos arquivos que ficam armazenados no navegador do usuário e servem para guardar informações temporárias, como preferências, dados de sessão ou *status* do sistema. Uma vantagem dos *cookies* é que, diferente do *localStorage* (que só funciona no *front-end*), eles também podem ser acessados pelo *back-end*. Isso permite que tanto o navegador quanto o servidor "leiam" o mesmo dado e se comportem de acordo com ele, melhorando a comunicação entre as camadas do sistema.

Após definir essa abordagem, o próximo passo foi revisar o arquivo principal da aplicação, o *index.js*. Esse arquivo é responsável por controlar ações importantes, como a tentativa de conexão com o banco de dados, a criação de sessões de usuário (ou seja, guardar informações enquanto a pessoa navega pelo site) e o redirecionamento para a tela inicial do Tem Lógica.

Foram realizadas modificações nesse código para que o sistema conseguisse funcionar de forma mais flexível. Agora, por exemplo, se o banco de dados não estiver disponível, o sistema pode exibir uma mensagem de aviso ou funcionar de forma limitada com dados simulados, sem que a plataforma pare completamente ou apresente erros graves. Isso representa um avanço importante no sentido de tornar o Tem Lógica mais robusto, estável e preparado para situações reais de uso e testes em fase de desenvolvimento.

4.3 Criação dos cookies

Com base na análise anterior, foi implementado um *cookie* específico para indicar se o banco de dados está ativo ou inativo. Esse processo de criação pode ser visualizado na Figura 26, enquanto a verificação do cookie está demonstrada na Figura 27.

O objetivo desse *cookie* é simples: ele funciona como um sinalizador, informando ao sistema — tanto no *front-end* quanto no *back-end* — o *status* da conexão com o banco de

dados. Assim, a plataforma pode adaptar seu funcionamento conforme a disponibilidade do banco.

As Figuras 28 e 29 mostram na prática como o sistema responde, exibindo mensagens diferentes quando o banco está disponível ou indisponível. Essa estratégia contribui para tornar a plataforma mais estável e capaz de lidar com falhas de conexão sem interromper totalmente seu funcionamento.

```
function createCookie(name, value) {
    app.use(cookieParser());
    app.use((req, res, next) => {
      res.cookie(name, value, { maxAge: 900000 });
      next();
    });
}
```

Figura 26: Bloco de código que executa a criação do cookie (Fonte: O próprio autor).

```
async function main() {
    try {
        await startApp();
        console.log("[ISCONNECTED] " + isConnected);

        if (isConnected === true) {
            createCookie('db_connected', true);
```

Figura 27: Bloco de código que verifica se o banco está ativo e sinaliza por meio da criação do cookie (Fonte: O próprio autor).

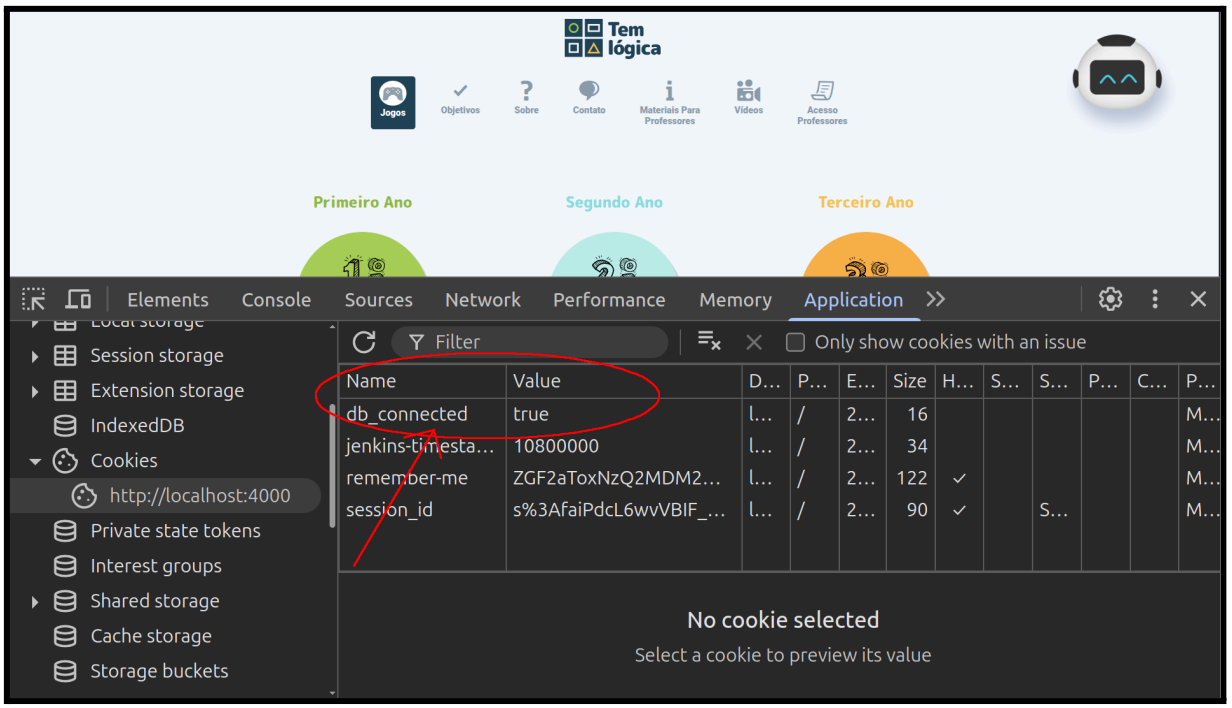

Figura 28: Saída na console informando que o banco de dados está conectado (Fonte: O próprio autor).

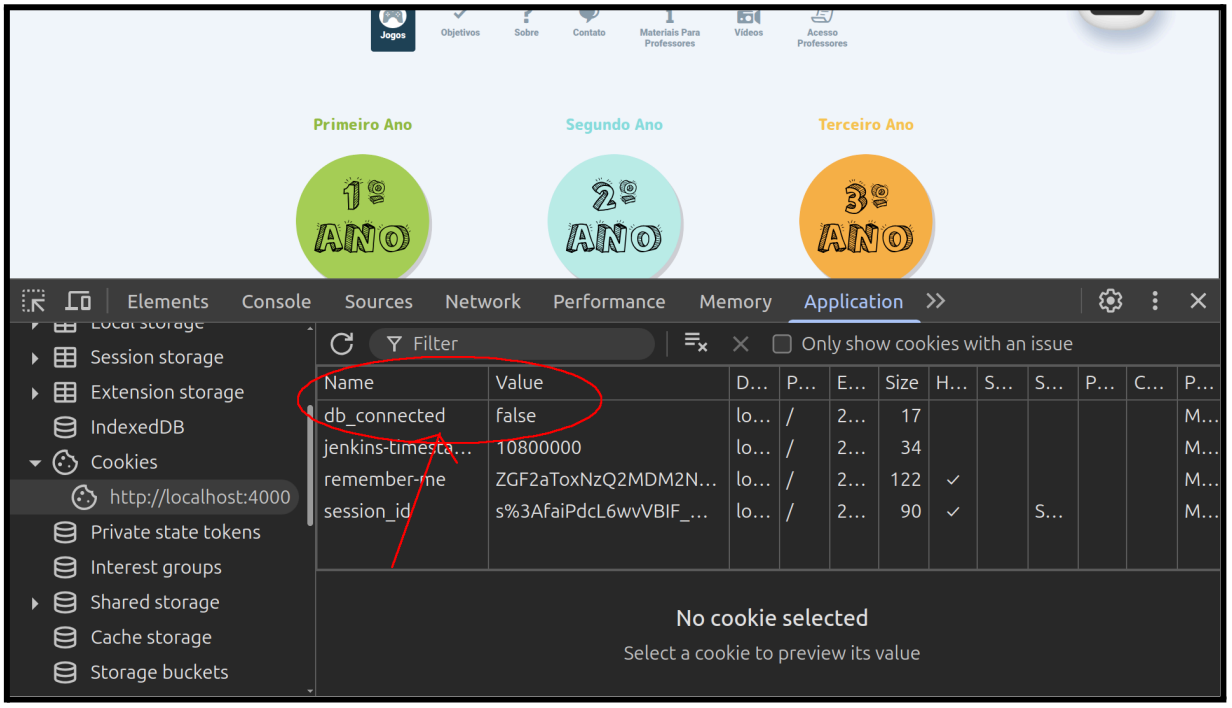

Figura 29: Saída na console informando que o banco de dados está desconectado (Fonte: O próprio autor).

Caso o cookie indique que o banco de dados está ativo, um trecho específico do código é executado, como mostrado nas Figuras 30 a 36. Esse trecho é responsável por

realizar a conexão com o banco, iniciar a sessão do usuário e, em seguida, direcioná-lo para a página principal da plataforma Tem Lógica.

```
async function main() {
    try {
        await startApp();
        console.log("[ISCONNECTED] " + isConnected);

        if (isConnected === true) {
            createCookie('db_connected', true);

            var option = {
                host: process.env.DATABASE_HOST,
                user: process.env.DATABASE_USER,
                password: process.env.DATABASE_PASSWORD,
                database: process.env.DATABASE_NAME,
                waitForConnections: true,
                connectionLimit: 100,
            }
```

Figura 30: Trecho de código que faz a verificação das variáveis do banco de dados caso ele esteja ativo (Fonte: O próprio autor).

```
optStore = {
    host: process.env.DATABASE_HOST,
    user: process.env.DATABASE_USER,
    password: process.env.DATABASE_PASSWORD,
    database: process.env.DATABASE_NAME,
    waitForConnections: true,
    connectionLimit: 100,
    createDatabaseTable: false,
    schema: {
        tableName: 'sessions',
        columnNames: {
            session_id: 'session_id',
            expires: 'expires',
            data: 'data'
        }
    }
};

var mysql, sessionStore;
mysql = connection.createPool(option);
sessionStore = new mysqlStore(optStore, mysql);
```

Figura 31: Trecho de código responsável por criar a sessão do banco de dados e estabelecer a conexão (Fonte: O próprio autor).

```
app.use(session({
    name: "session_id",
    resave: false,
    saveUninitialized: true,
    store: sessionStore,
    secret: "sioajffen",
    cookie: {
        maxAge: TWO_HOURS,
        sameSite: 'strict',
        secure: false

    },
    rolling: true
}));
```

Figura 32: Bloco de código que configura e preenche as variáveis de sessão do banco de dados para a conexão (Fonte: O próprio autor).

```
const dbConnected = JSON.parse(getCookie('db_connected'));
console.log("Cookie db_connected: " + dbConnected);
```

Figura 33: Checagem e exibição do status de conexão do banco de dados (Fonte: O próprio autor).

```
if (dbConnectedValue) {
    getFasesPorAno();
}
```

Figura 34: Bloco de código que redireciona o usuário para a seleção de jogos caso o banco de dados sinalize que está com conexão aberta (Fonte: O próprio autor).

```
function getCookie(name) {
    const cookieString = decodeURIComponent(document.cookie);
    const cookies = cookieString.split('; ');

    for (const cookie of cookies) {
        const [cookieName, cookieValue] = cookie.split('=');

        if (cookieName === name) {
            return cookieValue;
        }
    }

    return null;
}
```

Figura 35: Função responsável por capturar o cookie que aponta o status de conexão do banco de dados (Fonte: O próprio autor).

Por outro lado, se o *cookie* indicar que o banco de dados está inativo, outro trecho do código será executado, conforme mostrado na Figura 36. Nessa situação, a lógica cria um novo cookie informando que o banco está fora do ar e redireciona o usuário para uma página estática, que opera de forma independente, sem necessidade de conexão com o banco de dados.

```
} else {
    createCookie('db_connected', false);

    app.use(logger);
    app.use(express.static('./public_html'));


    app.use(express.urlencoded({extended:false}));
    app.use(express.json());
    app.use(formParser.array());


    app.set('trust proxy', true);
    app.use('/professores', routerProfessores);
    app.use('/atividade', routerAtividade);
    app.use(routerDefault);
    server = app.listen(process.env.PORT || 4000, () => console.log('App disponivel na http://localhost:4000'));
}
```

Figura 36: Trecho de código responsável por redirecionar o usuário para uma página sinalizando que não existe conexão entre banco de dados e aplicação. (Fonte: O próprio autor).

Essa página estática, ilustrada na Figura 37, opera de forma totalmente independente do banco de dados, permitindo que a plataforma funcione mesmo quando o banco estiver inativo. Nessa versão da aplicação, as funcionalidades visuais, como a movimentação de peças, permanecem ativas, mas sem realizar registros ou consultas. Isso evita erros de execução e facilita os testes e ajustes no código, sem a necessidade de preencher o banco com dados temporários ou simular requisições em um ambiente incompleto.

Figura 37: Página de exibição que informa a não conexão do banco de dados com a aplicação Tem Lógica (Fonte: O próprio autor).

Essa abordagem tem como principal objetivo facilitar os testes das funcionalidades da plataforma, sem a necessidade de manter o banco de dados conectado o tempo todo, como era exigido anteriormente. Essa mudança é especialmente útil durante o desenvolvimento, pois evita forçar o uso do banco em momentos em que ele não é realmente necessário.

Ao permitir que a aplicação funcione de forma independente, os desenvolvedores conseguem testar e ajustar as funcionalidades com mais liberdade, focando no visual, na lógica e na experiência do usuário, sem precisar de dados reais ou configurações complexas do banco.

Além disso, elimina a necessidade de inserir dados falsos só para testar a aplicação e depois apagá-los. O processo fica mais rápido, prático e organizado.

5 CONCLUSÃO

O desenvolvimento deste trabalho permitiu identificar limitações importantes na estrutura da plataforma Tem Lógica, especialmente relacionadas à dependência entre suas camadas. Observou-se que, inicialmente, o *front-end* dependia diretamente do *back-end* e do banco de dados para funcionar, o que dificultava a manutenção, a realização de testes e o desenvolvimento de novas funcionalidades.

A proposta de separação entre as camadas permitiu que cada parte da aplicação funcionasse de maneira mais independente. Com isso, tornou-se possível modificar e testar implementações no *front-end* sem a necessidade de ativar o banco de dados ou o servidor..

As melhorias implementadas visam justamente tornar a plataforma mais estável, organizada e preparada para crescer. Ao eliminar acoplamentos desnecessários, reduziu-se o risco de falhas em cadeia e aumentou-se a flexibilidade para futuras modificações.

Para trabalhos futuros, recomenda-se a adoção de uma estratégia de testes mais robusta, incluindo testes automatizados que ajudem a garantir o bom funcionamento de cada parte do sistema de forma isolada e integrada. A criação de ambientes de teste separados do ambiente de produção também será essencial para validar mudanças sem comprometer o uso da plataforma pelos usuários.

REFERÊNCIAS

[1] NERY E SILVA, Lincoln David; RÊGO, Rogéria Gaudencio do; RÊGO, Thaís Gaudencio do (Coord.). **O desenvolvimento das bases do pensamento computacional e a matemática: uma relação que tem lógica**. [João Pessoa]: [Ideia Editora], 2022.

[2] REFACTORING GURU. **Padrões de projeto de design**. [S.l.]: [Refactoring Guru], [2024]. Disponível em: https://refactoring.guru/pt-br/design-patterns. Acesso em: 22 abr. 2025.

[3] SANTOS, Marina Gabriela do Amaral; SOUZA, Maurício R. de A.; FIGUEIREDO, Eduardo. **Padrões de Projeto em Java: Um Estudo Prático sobre a Utilização e Benefícios. In: I Workshop sobre Aspectos Sociais, Humanos e Econômicos de Software** (WASHES 2016). Belo Horizonte: Universidade Federal de Minas Gerais (UFMG), 2016.

[4] AMAZON WEB SERVICES. **O que são microsserviços?** [S.l.]: [Amazon Web Services], [2025]. Disponível em: https://aws.amazon.com/pt/microservices/. Acesso em: 22 abr. 2025.

[5] MOZILLA DEVELOPER NETWORK. **Using HTTP cookies**. [S.l.]: [Mozilla Foundation], [2025]. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies. Acesso em: 22 abr. 2025.

[6] GITHUB. **How to build a CI/CD pipeline with GitHub Actions in four simple steps**. [S.l.]: [GitHub], [2025]. Disponível em: https://github.blog/enterprise-software/ci-cd/build-ci-cd-pipeline-github-actions-four-steps/. Acesso em: 22 abr. 2025.

[7] OLIVEIRA, Euller Henrique Bandeira. **Arquitetura de Microsserviços: Um Estudo de Caso. 2023**. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Universidade Federal de Uberlândia, Uberlândia, 2023.

[8] AMAZON WEB SERVICES. **Como escolher uma solução de banco de dados da AWS**. [S.l.]: [Amazon Web Services], [2025]. Disponível em: https://aws.amazon.com/pt/getting-started/decision-guides/databases-on-aws-how-t

o-choose/. Acesso em: 22 abr. 2025.

[9] GOOGLE CLOUD. **Visão geral do Cloud SQL**. [S.l.]: [Google Cloud], [2025]. Disponível em: https://cloud.google.com/sql/docs/introduction?hl=pt-br. Acesso em: 22 abr. 2025.

[10] GOOGLE CLOUD. **O que é um banco de dados na nuvem?** [S.l.]: [Google Cloud], [2025]. Disponível em: https://cloud.google.com/learn/what-is-a-cloud-database?hl=pt-BR. Acesso em: 22 abr. 2025.

[11] PELEGRINI, Leonardo B.; CORINO, Marcos J. Vissoto; SILVA, Roger Sá da. **Um estudo de caso sobre a utilização de containers para aplicações em nuvem de alta disponibilidade**. Veranópolis: Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul – Campus Avançado Veranópolis, 2023.

[12] NETO, Mauro Pires Moreira; AUGUSTO, Vinicius da Silva e Sousa. **Padrões para produção de aplicações utilizando Microsserviços**. 2021. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Instituto Federal de Educação, Ciência e Tecnologia de Goiás, Goiânia, 2021.

[13] MOZILLA DEVELOPER NETWORK. **JavaScript**. [S.l.]: [Mozilla Foundation], [2025]. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Acesso em: 22 abr. 2025.

[14] MOZILLA DEVELOPER NETWORK. **HTML**. [S.l.]: [Mozilla Foundation], [2025]. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/HTML. Acesso em: 22 abr. 2025.

[15] MYSQL. **MySQL**. [S.l.]: [Oracle Corporation], [2025]. Disponível em: https://www.mysql.com/. Acesso em: 22 abr. 2025.

[16] NETLIFY. **Netlify**. [S.l.]: [Netlify], [2025]. Disponível em: https://www.netlify.com/. Acesso em: 22 abr. 2025.

[17] DOCKER. **Containerize an application**. [S.l.]: Docker, [2024]. Disponível em: https://docs.docker.com/get-started/workshop/02_our_app/. Acesso em: 4 maio 2025.