



# UNIFACS

UNIVERSIDADE SALVADOR

LAUREATE INTERNATIONAL UNIVERSITIES\*

Usabilidade, Desenvolvimento Web, Mobile e Jogos

## **Relatório de aplicação Web: Query Games**

Andrey Barroso Costa – 1272123464

Leonardo Carvalho de Oliveira – 1272121364

Pablo Henrique Carvalho dos Santos – 1272121863

Vitor Souza dos Santos – 1272121538

Yana Barreto Luiz Simina – 1272219964

SALVADOR – BA

2023

## SUMÁRIO

1	Introdução: .....	3
2	Objetivo:.....	4
3	O que é um Wireframe? .....	5
4	Figma e Heurísticas de Nielsen. ....	6
4.1	Correspondência entre o sistema e o mundo real: .....	6
4.2	Controle e liberdade do usuário: .....	7
4.3	Prevenção de erros: .....	8
4.4	Reconhecimento em vez de lembrança:.....	9
4.5	Estética e design minimalista:.....	10
4.6	Ajude os usuários a reconhecer, diagnosticar e se recuperar de erros: .....	10
4.7	Ajuda e documentação:.....	11
5	Conclusão Parte I .....	12
6	API .....	13
7	Front-end .....	25
8	Conclusão parte II .....	31
9	Referências: .....	32

## **1 Introdução:**

Com a desenvoltura da tecnologia, a demanda no mundo dos jogos está cada vez mais exigente, os usuários são rigorosos quando se fala algo que certamente dominam. O compartilhamento de opiniões sempre existiu no meio digital, muitas vezes através de redes sociais que abrangem diversos assuntos. Através das buscas por comunidades de “Gamers” onde possam compartilhar experiências e opiniões como também consumir o conteúdo de outra pessoa, têm-se a criação da Aplicação Web “Query Games”, que busca a aproximação desses usuários, a facilidade em conhecer o enredo de um jogo através da opinião de um jogador e as suas diferentes perspectivas sobre o modo como aquele game se impõe ao usuário, poupando que comece um jogo que certamente não gostaria.

Dedicada ao gerenciamento eficiente do catálogo de jogos de um usuário, esta aplicação será uma solução para catalogar, organizar e acessar detalhes do jogo, disponibilizando uma plataforma online. Será desenvolvida seguindo as práticas de desenvolvimento web e utilizando as tecnologias atuais para garantir uma experiência de usuário otimizada. Os usuários poderão registrar, avaliar e acessar informações sobre seus jogos, incluindo descrições, imagens e dados. Ao decorrer deste relatório poderá entender a forma como essa aplicação será apresentada para o mercado, como foi o seu rascunho e toda a sua interface desenvolvida buscando cumprir as normas das Heurísticas de Nielsen.

## **2 Objetivo:**

Este projeto visa desenvolver uma aplicação web inovadora e centrada no usuário, priorizando a usabilidade e acessibilidade para garantir uma experiência positiva para uma ampla variedade de públicos. A intenção é criar uma plataforma intuitiva que atenda não apenas às necessidades funcionais, mas também se destaque pela facilidade de uso, permitindo que usuários com diferentes níveis de habilidade possam interagir de maneira eficaz.

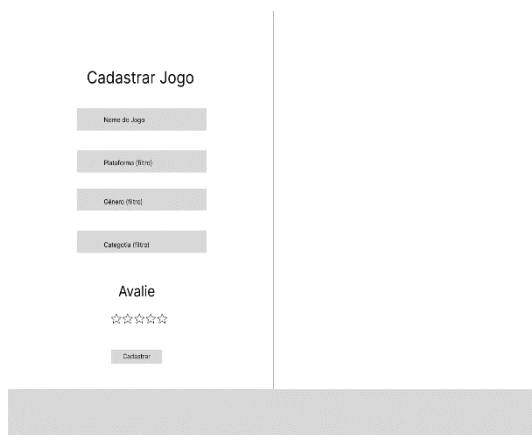
Além disso, a aplicação será projetada para ser escalável e flexível, permitindo futuras expansões e integrações sem comprometer a segurança ou a usabilidade. A arquitetura será cuidadosamente planejada, utilizando tecnologias modernas e boas práticas de desenvolvimento para garantir a manutenção e evolução sustentáveis da aplicação ao longo do tempo.

Dessa forma, o projeto busca criar uma aplicação web abrangente que não apenas atenda às expectativas funcionais dos usuários, mas também estabeleça novos padrões em termos de usabilidade.

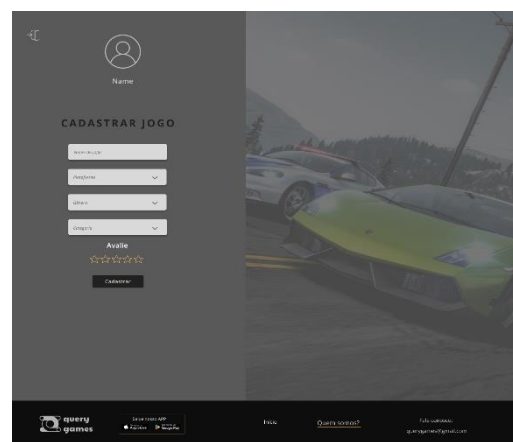
### 3 O que é um Wireframe?

Wireframe é um protótipo da página de um site ou aplicativo, ou seja, antes da elaboração de um layout cria-se um rascunho como visto nas **Figura 1** e **Figura 2**. Desta forma conseguimos enxergar como o produto final vai ficar sem mexer no que está pronto, apenas reajustando o esboço quantas vezes for necessário. O wireframe serve como um guia para a auxiliar na diagramação da página. Observe a Figura 3 e Figura 4.

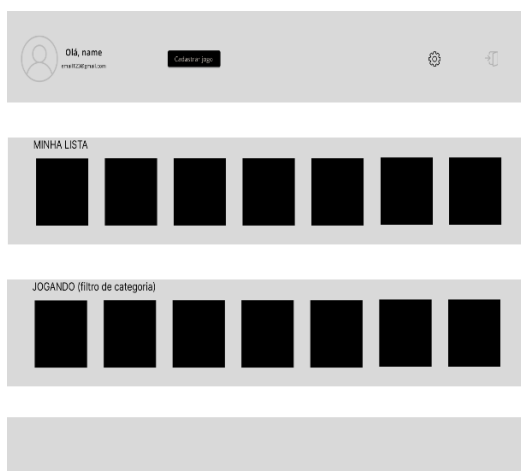
**Figura 1** - Tela "Cadastrar Jogo - Wireframe"



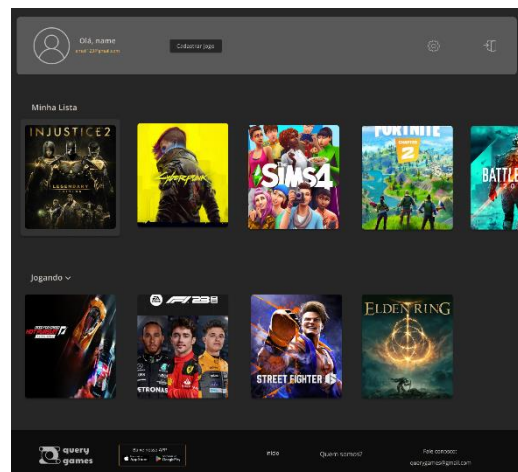
**Figura 2** - Tela "Cadastrar Jogo - Figma"



**Figura 3** - Tela "Usuário - Wireframe"



**Figura 4** - Tela "Tela Usuário - Figma"



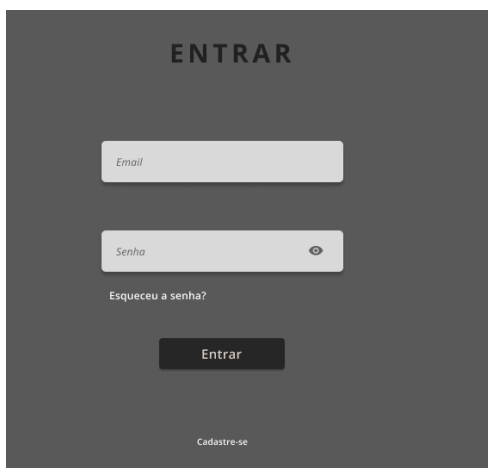
## 4 Figma e Heurísticas de Nielsen.

### 4.1 Correspondência entre o sistema e o mundo real:

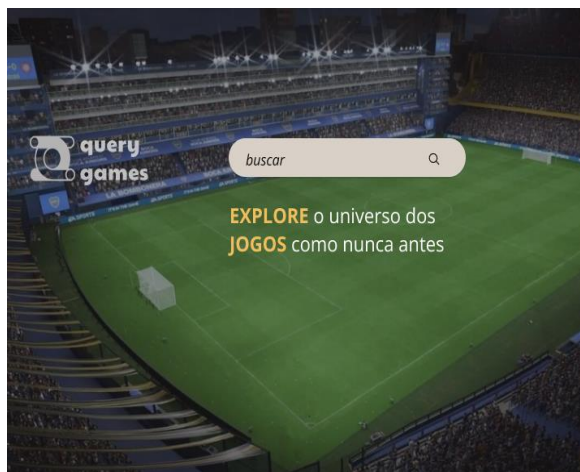
De acordo com Nielsen o design deve falar a língua dos usuários, utilizar frases, imagens, e conceitos familiares facilita para que o usuário se identifique com a aplicação. Certificar-se que o usuário possa entender o significado sem precisar procurar uma definição para algum elemento é essencial.

Na aplicação utilizou-se a linguagem correspondente com o vocabulário dos usuários. Com palavras, frases e conceitos familiares como mostrado na **Figura 5** o botão “Entrar” e na **Figura 6** o campo “Buscar”, consegue-se fazer com que se sintam mais confiantes nas tomadas de decisões em qualquer momento da navegação, além de tudo os ícones presentes na interface facilitam o entendimento. Para a avaliação dos jogos, usa-se estrelas completamente intuitivas como visto na **Figura 7**, os usuários facilmente associam a quantidade de estrelas preenchidas com qualidade e aprovação. O usuário esqueceu a senha? Facilmente consegue-se identificar o campo em que ele pode resolver o seu problema, com a simples pergunta: “Esqueceu a senha?” presente também na **Figura 5**.

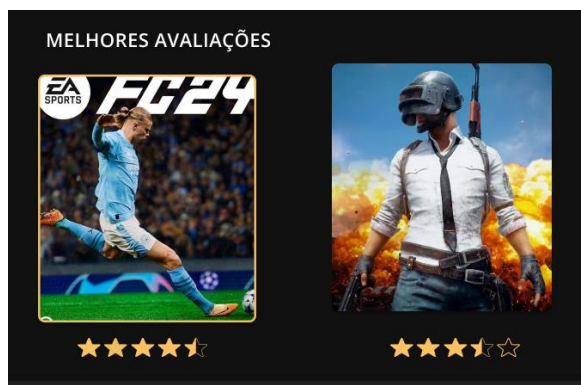
**Figura 5** - Tela "Entrar"



**Figura 6** - Campo "Buscar"



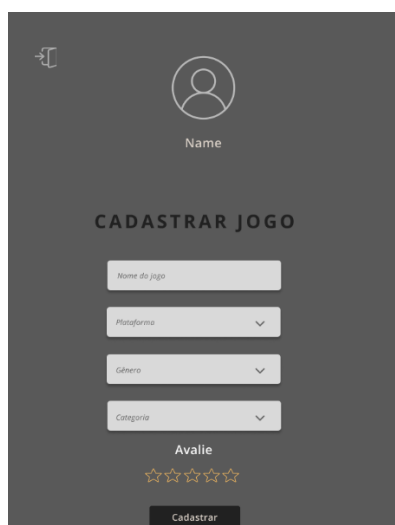
**Figura 7** – Tela “Avaliações Jogos”



#### 4.2 Controle e liberdade do usuário:

Empoderar os usuários com controle sobre suas ações e oferecer opções de saída claramente definidas é fundamental. Isso não apenas aumenta a sensação de controle do usuário, mas também reduz a ansiedade, permitindo uma experiência mais personalizada, com a opção de catalogar e avaliar seus jogos como visto na **Figura 8** ou até mesmo cadastrar uma plataforma.

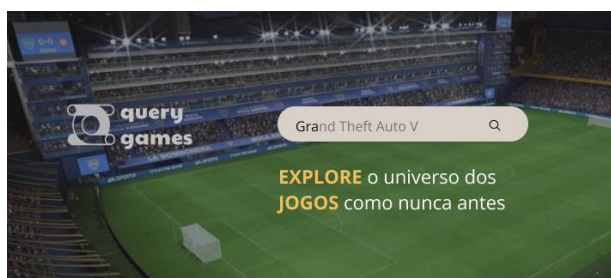
Nas telas optou-se pela “saída de emergência”, os usuários podem executar ações por engano, por isso precisam de um meio de retorno rápido e fácil que sejam claramente identificados, como por exemplo os ícones de "Porta de saída", presente em todas as telas para facilitar o encaminhamento de um local para outro.

**Figura 8** - Tela "Cadastrar Jogo"

#### 4.3 Prevenção de erros:

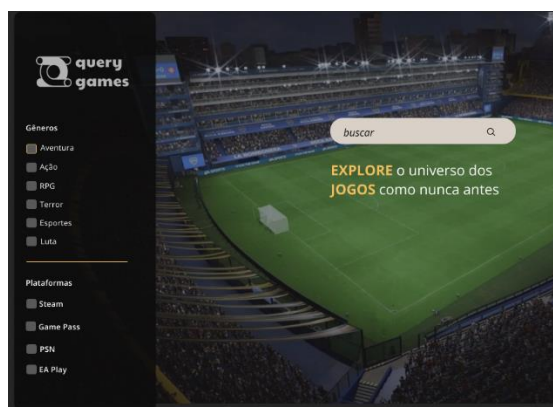
Evitar que os usuários cometam erros é tão crucial quanto fornecer meios fáceis de corrigi-los quando ocorrem. A aplicação dessa heurística visa reduzir a probabilidade de equívocos, garantindo uma experiência mais suave e sem frustrações, o erro mais comum entre eles é o da escrita, quando o usuário tentar colocar o e-mail é necessário que ele coloque o “@”, caso contrário o sistema emite uma mensagem e bordas visuais informando que tem algo errado, dando-lhe a possibilidade de correção. Assim como não conseguir achar algum jogo porque não sabe a sua escrita ao certo, com isso decidiu-se então que após o usuário digitar algumas letras, automaticamente a aplicação reconhecerá o nome e completará a palavra. Observe a **Figura 9**.



**Figura 9 - Campo "Buscar"**

#### 4.4 Reconhecimento em vez de lembrança:

Ao minimizar a carga cognitiva exigida dos usuários, facilitamos a navegação e a compreensão do sistema. A prioridade é atribuída ao reconhecimento de elementos, em vez de forçar os usuários a lembrar informações, tornando a experiência mais eficiente. Utilizou-se então o “Menu Hambúrguer” que contém informações de gênero e plataforma, essa heurística torna as opções visíveis para serem reconhecidas deixando ao critério do usuário o que irá o favorecer no momento. Isso torna a tela inicial mais limpa e leve visualmente, percebe na **Figura 10**.

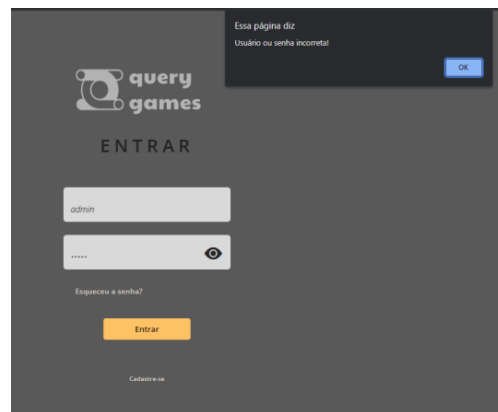
**Figura 10 - Menu Hambúrguer**

#### 4.5 Estética e design minimalista:

A estética é sobre como a aparência influencia a usabilidade. Um design limpo e minimalista não apenas melhora a estética, mas também facilita a localização de informações, promovendo uma experiência mais agradável. Buscou-se focar no essencial, a interface não contém informações irrelevantes, o conteúdo e o design repassam a ideia de uma aplicação mais limpa e objetiva com a padronização de todos os botões, como "Entrar" (presente na **Figura 5**) e "Cadastrar" (veja a **Figura 8**) ou o "Menu Hambúrguer" presente na **Figura 10**, que só é aberto ao comando do usuário, deixando a tela inicial mais objetiva.

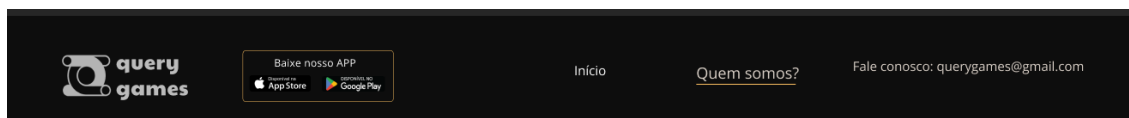
#### 4.6 Ajude os usuários a reconhecer, diagnosticar e se recuperar de erros:

De acordo com as Heurísticas de Nielsen, as mensagens de erro devem ser mostradas em linguagem simples sem todos aqueles códigos de erro, apenas indicando com precisão o problema e sugerindo uma solução de forma eficaz. É importante que ele tenha acesso a diagnósticos da situação, entendendo qual erro aconteceu e como é possível solucioná-lo. Pensando nisso, criou-se uma sinalização ao topo da tela (posição que chama a atenção) uma mensagem de cor destacada que facilmente o usuário conseguirá entender. Sem a presença de códigos e em linguagem simples indica-se o problema apresentado facilitando a resolução do empecilho. Veja na **Figura 11**.

**Figura 11** – Mensagem de Erro

#### 4.7 Ajuda e documentação:

De acordo com as heurísticas aplicadas sempre pretendemos que não haja dificuldade para a navegação, entretanto precisa-se evitar que o usuário fique sem compreender qualquer conteúdo apresentado, para isso criou-se a ajuda ao usuário mesmo sem uma solicitação, no *footer* (presente na **Figura 12**) da página está presente a opção “Email” para o contato, atendendo aos nossos clientes de forma individualizada, dando mais atenção aos seus problemas.

**Figura 12** - *Footer* das páginas

## 5 Conclusão Parte I

Em conclusão, este relatório reflete a jornada de desenvolvimento da aplicação web "Query Games" em resposta à crescente demanda por soluções inovadoras no mundo dos jogos. Reconhecendo a exigência dos usuários e a importância do compartilhamento de experiências no universo digital, a aplicação foi criada com o objetivo de aproximar a comunidade de gamers, oferecendo facilidade na descoberta de novos jogos e na troca construtiva de opiniões. Ao longo do desenvolvimento, o projeto manteve um foco sólido na experiência do usuário, seguindo as Heurísticas de Nielsen como um guia fundamental. A interface foi projetada com atenção à correspondência com o mundo real, garantindo que a linguagem e os elementos utilizados fossem familiares e intuitivos para os usuários. A ênfase na liberdade e controle do usuário se manteve firme ao design, proporcionando aos usuários a capacidade de catalogar, avaliar e personalizar suas interações na plataforma. A presença de opções de saída claras e ações reversíveis contribuem para uma experiência mais personalizada e livre de ansiedades.

O processo de criação incorporou a prática de wireframes, permitindo uma visão prévia do layout e facilitando ajustes iterativos. Além disso, o design minimalista adotado não apenas aprimora a estética, mas também contribui para a usabilidade, proporcionando uma experiência mais agradável e focada. A abordagem para lidar com erros foi cuidadosamente considerada, seguindo as diretrizes de Nielsen para mensagens de erro simples e claras, visando ajudar os usuários a reconhecer, diagnosticar e se recuperar de possíveis equívocos. Finalmente, a presença de recursos de ajuda e documentação, como o contato via e-mail no *footer* da página, reflete o compromisso contínuo com a satisfação do usuário, proporcionando suporte adicional quando necessário. Dessa forma, a aplicação web "Query Games" não apenas se propõe a ser uma solução eficaz para o gerenciamento de catálogos de jogos, mas também aspira a estabelecer novos padrões em termos de usabilidade e experiência do usuário no universo dos games online.

## 6 API

A adoção de padrões de projeto é essencial no desenvolvimento de software para promover a manutenibilidade, escalabilidade e reutilização de código. Um dos padrões amplamente utilizados é a separação das responsabilidades entre o repositório e o controlador, uma prática que proporciona uma arquitetura mais organizada e modular. A decisão de empregar essa abordagem específica reflete uma busca por um código mais limpo e coeso, capaz de lidar de maneira eficiente com a complexidade crescente de sistemas modernos.

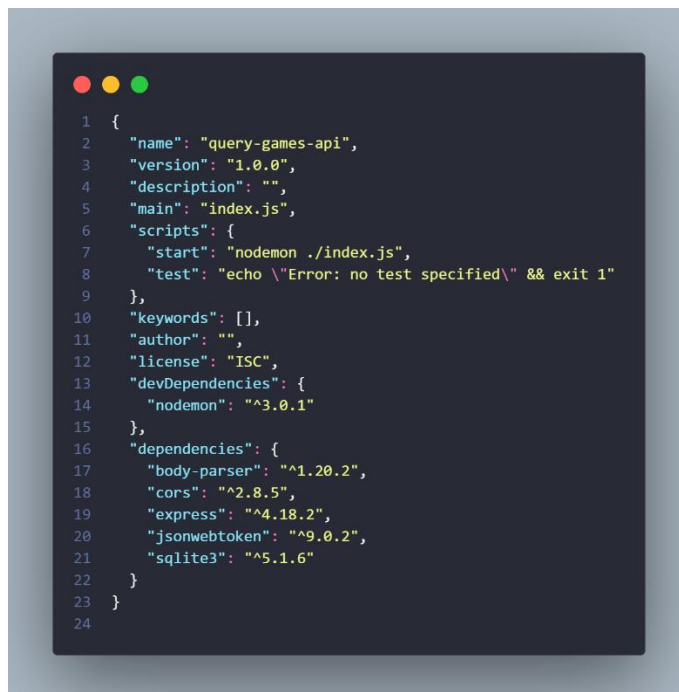
Ao separar o repositório, responsável pelo acesso e manipulação de dados, do controlador, que gerencia as interações e lógicas de negócio, obtemos benefícios significativos. A manutenção do código torna-se mais simples, uma vez que alterações nos requisitos de acesso aos dados não impactam diretamente a lógica da aplicação. Além disso, a modularidade resultante facilita a realização de testes unitários e a implementação de novos recursos de forma mais ágil.

A separação do repositório do controlador também promove a reusabilidade do código, pois os componentes podem ser adaptados e integrados em diferentes partes do sistema. Isso contribui para a construção de uma base sólida e flexível, permitindo expansões futuras com menor risco de introduzir erros não previstos. Em última análise, a escolha desse padrão de projeto reflete o comprometimento com boas práticas de desenvolvimento, visando a criação de software robusto, fácil de dar manutenção e adaptável às demandas em constante evolução.

O pontapé inicial para a implementação do projeto requer a execução do comando “npm install” no terminal dentro da pasta “backend”. Esse comando desempenha a crucial função de realizar o download das bibliotecas essenciais para a execução adequada do projeto, conforme meticulosamente especificado no arquivo package.json (conforme representado na figura 13). Esta etapa inaugural é vital para a preparação do ambiente de desenvolvimento, garantindo a disponibilidade de todas as dependências necessárias para o funcionamento correto do sistema. A iniciativa de utilizar o gerenciador de pacotes npm demonstra a intenção de estabelecer um ambiente controlado e uniforme, onde as versões

das bibliotecas são gerenciadas de maneira eficiente, minimizando possíveis conflitos e inconsistências.

Figura 13 -




```
1 {
2   "name": "query-games-api",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "start": "nodemon ./index.js",
8     "test": "echo \\\"Error: no test specified\\\" && exit 1"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "devDependencies": {
14    "nodemon": "^3.0.1"
15  },
16  "dependencies": {
17    "body-parser": "^1.20.2",
18    "cors": "^2.8.5",
19    "express": "^4.18.2",
20    "jsonwebtoken": "^9.0.2",
21    "sqlite3": "^5.1.6"
22  }
23 }
24
```

Em seguida, a execução do comando "npm start" constitui um script concebido para iniciar o projeto, utilizando a biblioteca Nodemon, a qual simplifica o desenvolvimento no ambiente Node.js ao reiniciar automaticamente o servidor em caso de detecção de alterações no código. Essa ferramenta, inserida estrategicamente no projeto, visa otimizar o ciclo de desenvolvimento, eliminando a necessidade de reinicializações manuais e contribuindo para uma abordagem mais eficiente e produtiva na construção do projeto.

O sistema utiliza o banco de dados SQLite devido à sua natureza embarcada, sendo gerado no início do projeto, e pela sua facilidade de integração. As configurações do banco são definidas no arquivo database.js no qual são incluídos códigos SQL para a criação de tabelas, caso não existam, e inserção de registros. Essa escolha visa simplificar o gerenciamento do banco de dados no contexto do projeto acadêmico, garantindo uma configuração eficiente e integrada às necessidades específicas da aplicação.

No arquivo `index.js`, é realizado o processo de criação e inicialização de um servidor HTTP, onde é especificado que ele operará na porta 3000 (conforme indicado na Figura 14).

Figura 14 – `index.js`



```
1 const http = require('http')
2 const app = require('./src/app');
3
4 const port = parseInt(process.env.PORT) || 3000;
5 const server = http.createServer(app);
6 server.listen(port, () => {
7   console.log(`Server started on port ${port}...`);
8 });
```

No arquivo `app.js` (Figura 15), são realizadas as configurações fundamentais do projeto acadêmico. Nele, o módulo Express é importado para a criação do aplicativo, o body-parser é utilizado para processar os dados provenientes do corpo das requisições e as rotas, previamente definidas no arquivo `routes.js` (conforme indicado na Figura 16), são incorporadas ao aplicativo.

Figura 15 – `app.js`



```
1 const express = require('express');
2 const bodyParser = require('body-parser');
3 const routes = require("./routes");
4
5 const app = express();
6 app.use(bodyParser.urlencoded({extended: false}));
7 app.use(express.json());
8 app.use(routes);
9
10 module.exports = app;
```

Figura 16 - Rotas

```

1  const express = require('express');
2  const authenticationController = require('./controllers/authentication-controller');
3  const userController = require('./controllers/user-controller');
4  const platformController = require('./controllers/platform-controller');
5  const categoryRepository = require('./controllers/category-controller');
6  const gameRepository = require('./controllers/game-controller');
7  const scoreRepository = require('./controllers/score-controller');
8  const profileRepository = require('./controllers/profile-controller');
9  const roleRepository = require('./controllers/role-controller');
10
11  const routes = express();
12
13  routes.post('/authentication', authenticationController.postAuth);
14
15  routes.get('/users', userController.get);
16  routes.get('/users/games/:id', userController.getGamesById);
17  routes.get('/users/:id', userController.getById);
18  routes.post('/users', userController.post);
19  routes.put('/users/:id', authenticationController.verifyToken, userController.putById);
20  routes.delete('/users/:id', authenticationController.verifyToken, userController.deleteById);
21
22  routes.get('/platforms', platformController.get);
23  routes.get('/platforms/:id', platformController.getById);
24  routes.post('/platforms', authenticationController.verifyToken, platformController.post);
25  routes.put('/platforms/:id', authenticationController.verifyToken, platformController.putById);
26  routes.delete('/platforms/:id', authenticationController.verifyToken, platformController.deleteById);
27
28  routes.get('/categories', categoryRepository.get);
29  routes.get('/categories/:id', categoryRepository.getById);
30  routes.post('/categories', authenticationController.verifyToken, categoryRepository.post);
31  routes.put('/categories/:id', authenticationController.verifyToken, categoryRepository.putById);
32  routes.delete('/categories/:id', authenticationController.verifyToken, categoryRepository.deleteById);
33
34  routes.get('/games', gameRepository.get);
35  routes.get('/games/:id', gameRepository.getById);
36  routes.post('/games', authenticationController.verifyToken, gameRepository.post);
37  routes.put('/games/:id', authenticationController.verifyToken, gameRepository.putById);
38  routes.delete('/games/:id', authenticationController.verifyToken, gameRepository.deleteById);
39
40  routes.get('/scores', scoreRepository.get);
41  routes.get('/scores/:id', scoreRepository.getById);
42  routes.get('/scores/games/:id', scoreRepository.getByGameId);
43  routes.post('/scores', authenticationController.verifyToken, scoreRepository.post);
44  routes.put('/scores/:id', authenticationController.verifyToken, scoreRepository.putById);
45  routes.delete('/scores/:id', authenticationController.verifyToken, scoreRepository.deleteById);
46
47  routes.get('/profiles', profileRepository.get);
48  routes.get('/profiles/:id', profileRepository.getById);
49  routes.post('/profiles', authenticationController.verifyToken, profileRepository.post);
50  routes.put('/profiles/:id', authenticationController.verifyToken, profileRepository.putById);
51  routes.delete('/profiles/:id', authenticationController.verifyToken, profileRepository.deleteById);
52
53  routes.get('/roles', roleRepository.get);
54  routes.get('/roles/:id', roleRepository.getById);
55  routes.post('/roles', authenticationController.verifyToken, roleRepository.post);
56  routes.put('/roles/:id', authenticationController.verifyToken, roleRepository.putById);
57  routes.delete('/roles/:id', authenticationController.verifyToken, roleRepository.deleteById);
58
59  module.exports = routes;

```

Na rota de autenticação do projeto, foi implementado um endpoint do tipo POST no qual são fornecidos, no corpo da requisição, o e-mail e a senha do usuário. Na respectiva controller de autenticação, foi desenvolvido uma função que verifica no banco de dados



a existência desse usuário. Se não for encontrado, o sistema responde indicando credenciais inválidas. No caso de um usuário existente, a função gera um token personalizado, retornando informações cruciais como o tipo do token, o próprio token e a sua duração em milissegundos. Esse processo visa garantir a segurança e a eficácia da autenticação no contexto do projeto acadêmico.

Figura 17 – Autenticação JWT

```

1  const userRepository = require('../repositories/user-repository');
2  const profileRepository = require('../repositories/profile-repository');
3  const jwt = require('jsonwebtoken');
4
5  const SECRET = 'hf293uh4g90234hg92-384hg2-9034nc-742-37c02-734-0239u40-239u0-237u4-023u-450n3h-';
6
7  async function postAuth(req, res) {
8    const authContent = req.body;
9    const user = await userRepository.findByEmailAndPassword(authContent.email, authContent.password);
10   if (user === null || user === undefined) {
11     res.status(401).json({message: 'Invalid Credentials!!!'}).end();
12   }
13   return;
14   let profileName = null;
15   if (user.profileId !== null) {
16     const profile = await profileRepository.findById(user.profileId);
17     profileName = profile.name;
18   }
19   const timeExpire = '1h';
20   const token = jwt.sign({userId: user.id, username: user.name, profile: profileName}, SECRET, {subject: user.email, issuer: 'Query Games API', expiresIn: timeExpire});
21   res.status(200).json({type: 'Bearer', token: token, expire: timeExpire});
22 }
23
24 function verifyToken(req, res, next) {
25   const authorization = req.headers['authorization'];
26   if (!authorization) {
27     res.status(401).json({message: 'Invalid Token!!!'}).end();
28     return;
29   }
30   const token = authorization.split(' ')[1];
31   jwt.verify(token, SECRET, (err, decoded) => {
32     if (err) {
33       res.status(401).json({message: 'Invalid Token!!!'}).end();
34       return;
35     }
36     req.logged = {
37       id: decoded.userId,
38       name: decoded.username,
39       profile: decoded.profile,
40     };
41     next();
42   });
43 }
44
45 module.exports = {postAuth, verifyToken};

```

De maneira geral, as controllers no âmbito do projeto são encarregadas de processar as requisições, enquanto os repositórios desempenham a função de realizar consultas no banco de dados. Cada controller depende de um repository correspondente, delineando uma abordagem organizacional essencial para a estrutura do projeto. A síntese do sistema pode ser resumida a essa relação fundamental, com exceção de nuances relacionadas às regras de negócio. Notavelmente, o usuário com o perfil "Administrador" possui permissões abrangentes, contrastando com o usuário "Cliente" que detém permissões restritas no sistema.

Tendo como exemplo o arquivo “user-controller.js” a função “get” é uma função assíncrona que retorna todos os usuários do sistema utilizando a função “findAll” do

arquivo “user-repository.js” que por sua vez retorna uma “Promise” realizando uma consulta no banco de dados para recuperar todos os registros da tabela “users”. A função “get” também realiza um loop pelos usuários retornando, se existente, um id de “profile” e realiza outra função assíncrona pra obter o perfil correspondente.

Figura 18 – Função “get” (user-controller.js)

```
1  async function get(req, res) {
2    const users = await userRepository.findAll();
3    for(let i =0; i < users.length; i++) {
4      const profileId = users[i].profileId;
5      if (profileId !== null) {
6        const profile = await profileRepository.findById(profileId);
7        users[i]['profile'] = profile.name;
8      }
9    }
10   res.json(users);
11 }
```

Figura 19 – Função “findAll” (user-repository.js)

```
1  async function findAll() {
2    return new Promise((resolve, reject) => {
3      const users = [];
4      db.each('SELECT * FROM users ORDER BY id', (err, row) => {
5        if (err) {
6          console.error('Occurred an error with find all users!');
7          reject(err);
8        }
9        users.push(row);
10     }, (err, count) => {
11       if (err) reject(err);
12       resolve(users);
13     });
14   });
15 }
```

A função “getById” busca de usuário por ID passado como parâmetro na requisição. Ela utiliza os repositórios assíncronos `userRepository` e `profileRepository` para recuperar informações do usuário, incluindo o nome do perfil se existir. Se o usuário não for encontrado, retorna um erro 404. A função trata exceções, registrando mensagens de erro e retornando uma resposta de erro interno (status 500) em caso de problemas.

Figura 20 - Função “getById” (user-controller.js)

```

1  async function getById(req, res) {
2    try {
3      const user = await userRepository.findById(req.params.id);
4      if (!user) {
5        return res.status(404).json({ error: "User not found" });
6      }
7      if (user.profileId !== null) {
8        const profile = await profileRepository.findById(user.profileId);
9        if (profile) {
10         user['profile'] = profile.name;
11       } else {
12         console.warn('Profile not found for ID ${user.profileId}');
13         user['profile'] = "Profile not found";
14       }
15     }
16     res.json(user);
17   } catch (error) {
18     console.error("Error getting user by ID", error);
19     res.status(500).json({ error: "Internal server error" });
20   }
21 }

```

Figura 21 – Função “findById” (user-repository.js)

```

1  async function findById(id) {
2    return new Promise((resolve, reject) => {
3      const stmt = db.prepare('SELECT * FROM users WHERE id = ?', [id]);
4      stmt.get((err, row) => {
5        if (err) {
6          console.error('Occurred an error with find user by id!');
7          reject(err);
8        }
9        resolve(row);
10      });
11      stmt.finalize();
12    });
13 }

```

A função “getGamesById” retorna os jogos cadastrados pelo usuário específico. Ela utiliza o método assíncrono “findGamesById” do arquivo “userRepository.js” que por sua vez retorna uma “Promise” realizando uma consulta no banco de dados obtendo os jogos associados ao ID do usuário passado. Após aguardar a conclusão dessa operação assíncrona, a função envia a lista de jogos encontrada como uma resposta JSON ao cliente que fez a requisição.

Figura 22 - Função “getGamesById” (user-controller.js)



```

1  async function getGamesById(req, res) {
2    const games = await userRepository.findGamesById(req.params.id);
3    res.json(games);
4  }

```

Figura 23 – Função “findGamsById” (user-repository.js)



```

1  async function findGamesById(id) {
2    return new Promise((resolve, reject) => {
3      const games = [];
4      db.each('SELECT G.id, G.name as game, P.name as platform, C.name as category, S.note FROM users U
5      INNER JOIN user_games UG ON (U.id = UG.userId) INNER JOIN games G on (UG.gameId = g.id)
6      INNER JOIN platforms P on (G.platformId = P.id) INNER JOIN games_categories GC on (G.id = GC.gameId)
7      INNER JOIN categories C on (GC.categoryId = C.id) LEFT JOIN scores S on (G.id = s.gameId) WHERE U.id = ${id}', (err, row) => {
8        if (err) {
9          console.error('Occurred an error with find all games created from user!');
10         reject(err);
11        }
12        games.push(row);
13      }, (err, count) => {
14        if (err) reject(err);
15        resolve(games);
16      });
17    });
18  }

```

A função “post” é uma rota de requisição HTTP do tipo POST que lida com a criação de novos usuários. Ao receber uma requisição, a função utiliza o método assíncrono “findByEmail” do repositório “userRepository” para verificar se já existe um usuário com o mesmo endereço de e-mail fornecido no corpo da requisição. Se um usuário já existir, a função retorna um erro 400 indicando que o e-mail já está registrado. Caso contrário, utiliza o método assíncrono “insert” para adicionar o novo usuário ao banco de dados, e retorna o usuário recém-criado com um código de status 201, indicando sucesso na

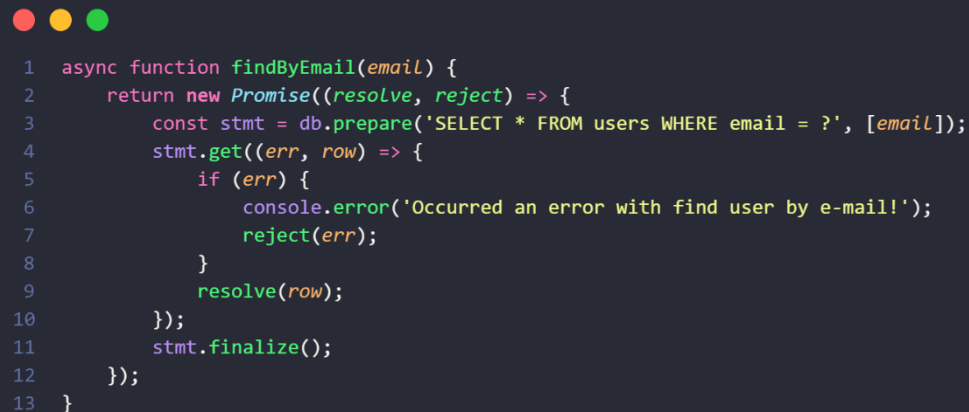
criação. Em caso de qualquer erro durante o processo, a função trata exceções, registrando mensagens de erro e retornando uma resposta de erro interno (status 500) ao cliente.

Figura 24 – Função “post” (user-controller.js)



```
1  async function post(req, res) {
2    try {
3      const existingUser = await userRepository.findByEmail(req.body.email);
4      if (existingUser) {
5        return res.status(400).json({ error: 'Email already registered.' });
6      }
7      const user = await userRepository.insert(req.body);
8      res.status(201).json(user);
9    } catch (error) {
10     console.error(error);
11     res.status(500).json({ error: 'Internal server error.' });
12   }
13 }
```

Figura 25 – Função findByEmail (user-repository.js)



```
1  async function findByEmail(email) {
2    return new Promise((resolve, reject) => {
3      const stmt = db.prepare('SELECT * FROM users WHERE email = ?', [email]);
4      stmt.get((err, row) => {
5        if (err) {
6          console.error('Occurred an error with find user by e-mail!');
7          reject(err);
8        }
9        resolve(row);
10      });
11      stmt.finalize();
12    });
13 }
```

Figura 26 – Função insert (user-repository.js)



```

1  async function insert(user) {
2    return new Promise((resolve, reject) => {
3      const stmt = db.prepare('INSERT INTO users(name, email, dateOfBirth, password, profileId) VALUES(?, ?, ?, ?, ?)');
4      stmt.bind([user.name, user.email, user.dateOfBirth, user.password, user.profileId]);
5      stmt.run(err => {
6        if (err) {
7          console.error('Occurred an error with insert user!');
8          reject(err);
9        }
10     });
11     stmt.finalize();
12     const stmt2 = db.prepare('SELECT seq FROM sqlite_sequence WHERE name = "users"');
13     stmt2.get((err, row) => {
14       resolve(findById(row ? row['seq'] + 1 : 1));
15     });
16     stmt2.finalize();
17   });
18 }

```

A função “putById” representa uma rota de requisição HTTP do tipo PUT, destinada à atualização de informações de um usuário com base no ID fornecido na requisição. Inicialmente, a função utiliza o método assíncrono “findById” do repositório “userRepository” para obter o usuário correspondente ao ID fornecido. Se o usuário não existir, a função retorna um erro 404 indicando que o usuário não foi encontrado. Em seguida, verifica se o usuário possui um perfil associado e se o ID do perfil no corpo da requisição é diferente do perfil existente no banco de dados. Se essas condições forem verdadeiras, a função retorna um erro 400 indicando um perfil inválido para o usuário. Caso contrário, utiliza o método assíncrono update do repositório para realizar a atualização do usuário com as informações fornecidas no corpo da requisição. Finalmente, retorna uma resposta de sucesso com o código de status 204, indicando que a operação foi realizada com sucesso, mas não há conteúdo a ser retornado no corpo da resposta.

Figura 27 – Função “putById” (user-controller.js)



```

1  async function putById(req, res) {
2    const user = await userRepository.findById(req.params.id);
3    if (!user) {
4      res.status(404).json({message: 'User not found!'});
5      return;
6    }
7    if (user.profileId !== null && user.profileId !== req.body.profileId) {
8      res.status(400).json({message: 'Invalid profile to user!'});
9      return;
10   }
11   await userRepository.update(req.body);
12   res.status(204).json();
13 }

```

Figura 28 – Função “update” (user-repository.js)

```

1  async function update(user) {
2    return new Promise((resolve, reject) => {
3      const stmt = db.prepare('UPDATE users set name = ?, dateOfBirth = ?, password = ?, profileId = ? WHERE id = ?');
4      stmt.bind([user.name, user.dateOfBirth, user.password, user.profileId, user.id]);
5      stmt.run(err => {
6        if (err) {
7          console.error('Occurred an error with update user!');
8          reject(err);
9        }
10       resolve();
11     });
12     stmt.finalize();
13   });
14 }

```

A função “deleteById” representa uma rota de requisição HTTP do tipo DELETE, destinada à exclusão de um usuário com base no ID fornecido na requisição. Antes de prosseguir com a exclusão, a função verifica se o perfil do usuário logado possui a permissão de administrador; caso contrário, retorna um erro 403 indicando falta de permissão para realizar a operação. Em seguida, utiliza o método assíncrono findById do repositório userRepository para obter informações sobre o usuário que será excluído. Se o usuário não for encontrado, a função retorna um erro 404 indicando que o usuário não existe. Caso contrário, utiliza o método assíncrono “deleteById” do repositório para efetuar a exclusão do usuário com base no ID obtido. A resposta ao cliente indica o sucesso da operação com um código de status 204, indicando que a operação foi bem-sucedida, mas não há conteúdo a ser retornado no corpo da resposta. Essa função inclui uma verificação adicional de permissão para garantir que apenas administradores podem executar a exclusão de usuários.

Figura 29 – Função “deleteById” (user-controller.js)

```

1  async function deleteById(req, res) {
2    if (req.logged.profile !== 'Administrador') {
3      return res.status(403).json({message: 'You not has permission to execute this operation!'});
4    }
5    const user = await userRepository.findById(req.params.id);
6    if (!user) {
7      res.status(404).json({message: 'User not found!'});
8      return;
9    }
10   await userRepository.deleteById(user.id);
11   res.status(204).json()
12 }

```

Figura 30 – Função “deleteById” (user-repository.js)

A screenshot of a code editor with a dark background and light-colored text. The code is written in JavaScript and defines an asynchronous function named deleteById. The function takes an 'id' parameter and returns a new Promise. Inside the Promise, a database statement is prepared to delete a user by ID. The statement is then executed, and an error is handled by logging a message and rejecting the promise. Finally, the statement is finalized and the promise is resolved.

```
1  async function deleteById(id) {  
2      return new Promise((resolve, reject) => {  
3          const stmt = db.prepare('DELETE FROM users WHERE id = ?');  
4          stmt.bind([id]);  
5          stmt.run(err => {  
6              if (err) {  
7                  console.error('Occurred an error with delete user!');  
8                  reject(err);  
9              }  
10             resolve();  
11         });  
12         stmt.finalize();  
13     });  
14 }
```

Documentação da API:

<https://documenter.getpostman.com/view/26587593/2s9YXmWzLW>

Coleção do Postman:

<https://mega.nz/file/PwJyRZQY#tpDrjSoxIzVOqkslDh97a5gJmhuCPVT8MCiGg6PIwWQ>



## 7 Front-end

Para iniciar a aplicação front-end, primeiro, navegue até o diretório "frontend" usando o comando "cd frontend". Em seguida, execute "npm install" para instalar as dependências do projeto. Após a conclusão da instalação, inicie a aplicação utilizando o comando "npm start". Isso iniciará o projeto na porta 3006, permitindo o acesso à aplicação. Certifique-se de que o servidor esteja em execução para garantir um funcionamento adequado.

A configuração inicial para comunicação do front-end com a API é realizada no arquivo "config.js" onde é utilizado a biblioteca "Axios" para criar uma instância chamada api. Essa instância é configurada com uma base URL definida como "http://localhost:3000/". A biblioteca Axios é comumente utilizada para realizar requisições HTTP em aplicações JavaScript, como em aplicações web front-end ou back-end. Ao criar essa instância, facilita-se o envio de requisições para o servidor hospedado em "http://localhost:3000/", simplificando o código ao fornecer uma única instância configurada com a URL base desejada.

Dentro do arquivo "routes.jsx", as rotas são configuradas usando a biblioteca "react-router-dom". No arquivo, há uma função que verifica a presença de um token antes de permitir a navegação para as rotas protegidas. A rota padrão do sistema ("/") redireciona automaticamente para a rota "/login". Se um usuário não autenticado tentar acessar rotas protegidas, ele será redirecionado para a página de login, garantindo assim a segurança e controle de acesso na aplicação.

Na página de cadastro de usuário, ao submeter o formulário de cadastro, são realizadas diversas validações. Estas incluem a verificação de campos preenchidos, validação da data de nascimento, onde o ano deve estar no intervalo entre 1900 e 2018, e a validação do tamanho da senha, que deve conter no mínimo 8 caracteres. Além disso, há a validação da confirmação de senha, onde os campos "senha" e "confirmação de senha" são comparados para garantir que sejam iguais antes de enviar a requisição para a API. Após passar por essas validações, a requisição é enviada com os dados do formulário de cadastro no corpo da requisição. Caso o e-mail fornecido no formulário já exista, o usuário recebe uma mensagem de erro indicando a necessidade de escolher um e-mail

único. Essas validações garantem a integridade dos dados submetidos e uma experiência de usuário mais robusta.

Na página de login, ao submeter o formulário, são conduzidas verificações essenciais, abrangendo a validação de campos preenchidos e a autenticação de credenciais. Após a conclusão dessas verificações, a requisição é encaminhada para a rota de autenticação da API. Uma vez autenticado com sucesso, a API retorna um token de autenticação, o qual é armazenado localmente no `localStorage`. Subsequentemente, o usuário é redirecionado automaticamente para a página "home", proporcionando uma transição suave após a autenticação bem-sucedida. Este processo assegura a segurança e a gestão eficiente das credenciais de autenticação na aplicação.

A página "home" é uma interface simplificada, apresentando de forma fictícia dados relacionados a jogos. Neste espaço, os usuários encontram informações simuladas sobre diversos jogos, proporcionando uma experiência atraente e intuitiva.

Na barra de navegação, oferecemos acesso simplificado às páginas "home", "meu perfil", "catalogar jogo" e "cadastrar plataforma". Além disso, implementamos um botão dedicado para efetuar o logout do sistema. Ao acionar esse botão, o sistema remove o token de autenticação associado ao usuário armazenado no `localStorage`, proporcionando uma saída segura e encaminhando-o de volta para a página de login. Esse mecanismo de logout garante a efetiva desconexão do usuário, contribuindo para a segurança e privacidade dentro da aplicação.

Na página de cadastro de plataforma, após a validação dos campos obrigatórios, é efetuada uma requisição para a API, especificamente na rota de plataformas, transmitindo os dados do formulário de cadastro de plataforma no corpo da requisição. No cabeçalho da requisição, são incluídos o token recuperado do `localStorage` e o tipo correspondente do token. Em caso de sucesso, a API retorna uma confirmação ao usuário, informando que a plataforma foi cadastrada com êxito.

Na página de catalogar jogo, no menu suspenso referente às plataformas e categorias no formulário de cadastro de jogo, são exibidas as opções previamente cadastradas. Ao usuário, basta selecionar as opções desejadas. No corpo da requisição, são incluídos o

nome do jogo, o ID da plataforma e o ID da categoria escolhidos. No cabeçalho da requisição, são enviados o token obtido do localStorage e o tipo correspondente do token, garantindo a autenticação adequada. Ao passar pelas validações dos campos obrigatórios, a aplicação retorna ao usuário uma confirmação indicando que o jogo foi catalogado com sucesso.

Na página "meu perfil", são exibidos todos os jogos cadastrados pelo usuário, proporcionando também a funcionalidade de atribuir uma nota ao jogo mediante a interação com estrelas, implementada por meio da biblioteca "react-simple-star-rating". A ação de avaliação é tratada na função "handleRating", que, dentro de um bloco try, inicia com uma requisição GET na rota "score/games" da API, passando o ID do jogo como parâmetro. Caso não exista uma nota atribuída ao jogo, uma requisição POST é enviada para a rota "score", incluindo a nota, o ID do jogo e o ID do usuário (obtido do token armazenado no localStorage) no corpo da requisição, e o token e seu tipo no cabeçalho. Se já houver uma nota associada ao jogo, uma requisição PUT é realizada na rota "score/id", com o ID da nota como parâmetro e o ID da nota, o valor da nota, o ID do jogo e o ID do usuário no corpo da requisição.

Além disso, a página apresenta a função "handleUpdateGame" que utiliza a biblioteca SweetAlert2 para exibir os dados do jogo selecionado, permitindo ao usuário realizar alterações. Adicionalmente, a função "handleDeleteGame" gerencia a exclusão do jogo, apresentando um SweetAlert para confirmar a decisão do usuário. Após a confirmação, uma requisição DELETE é enviada para a rota "games" da API, com o ID do jogo como parâmetro.

Figura 31 – Função “handleRating” (catalogGames)

```
1  const handleRating = async (rate, gameId) => {
2    if (rate >= 1 && rate <= 5) {
3      setRating(rate);
4    }
5
6    try {
7      const response = await api.get(`scores/games/${gameId}`);
8
9      if (!response.data) {
10       await api.post(
11         "scores",
12         {
13           note: rate,
14           gameId: gameId,
15           userId: userData.userId,
16         },
17         {
18           headers: {
19             Authorization: `Bearer ${localStorage.getItem("token")}`,
20           },
21         }
22       );
23     } else {
24       await api.put(
25         `scores/${response.data.id}`,
26         {
27           id: response.data.id,
28           note: rate,
29           gameId: gameId,
30           userId: userData.userId,
31         },
32         {
33           headers: {
34             Authorization: `Bearer ${localStorage.getItem("token")}`,
35           },
36         }
37       );
38     }
39     setRating(rate);
40   } catch (error) {
41     console.error("Erro ao enviar pontuação para a API:", error);
42   }
43   };
```

Figura 31 – Função “handleUpdateGame” (catalogGames)

```

1  async function handleUpdateGame(gameId) {
2    try {
3      const gameResponse = await api.get(`games/${gameId}`);
4      const gameData = gameResponse.data;
5      const platformsResponse = await api.get("platforms");
6      const categoriesResponse = await api.get("categories");
7      const platforms = platformsResponse.data;
8      const categories = categoriesResponse.data;
9
10     const swalResult = await Swal.fire({
11       title: "Atualizar Jogo",
12       html: `
13         <div style="margin-bottom: 10px;">
14           <label for="swal-input1">Nome:</label>
15           <input id="swal-input1" class="swal2-input" value="${gameData.name}">
16         </div>
17         <div style="margin-bottom: 10px;">
18           <label for="swal-input2">Plataforma:</label>
19           <select id="swal-input2" class="swal2-input">
20             ${platforms
21               .map(
22                 (platform) =>
23                   `<option value="${platform.id}" ${
24                     platform.id === gameData.platformId ? "selected" : ""
25                   }>${platform.name}</option>`
26               )}
27           </select>
28         </div>
29         <div style="margin-bottom: 10px;">
30           <label for="swal-input3">Categorias:</label>
31           <select id="swal-input3" class="swal2-input">
32             ${categories
33               .map(
34                 (category) =>
35                   `<option value="${category.id}" ${
36                     category.id === gameData.categories[0].id ? "selected" : ""
37                   }>${category.name}</option>`
38               )}
39           </select>
40         </div>
41       `,
42       focusConfirm: false,
43       showCancelButton: true,
44       confirmButtonText: "Sim",
45       cancelButtonText: "Não",
46     });
47
48     if (swalResult.isConfirmed) {
49       const updatedName = document.getElementById("swal-input1").value;
50       const updatedPlatformId = parseInt(document.getElementById("swal-input2").value);
51       const updatedCategories = parseInt(document.getElementById("swal-input3").value);
52       const payload = {
53         id: gameId,
54         name: updatedName,
55         platformId: updatedPlatformId,
56         categories: [
57           {
58             id: updatedCategories,
59           },
60         ],
61       };
62       const updateResponse = await api.put(
63         `games/${gameId}`,
64         payload,
65         {
66           headers: {
67             Authorization: `Bearer ${localStorage.getItem("token")}`,
68           },
69         }
70       );
71       if (updateResponse.status === 204) {
72         const updatedUserGames = userGames.map((game) =>
73           game.id === gameId ? { ...game, ...updateResponse.data } : game
74         );
75         setUserGames(updatedUserGames);
76         Swal.fire({
77           title: "Atualizado!",
78           text: "O jogo foi atualizado com sucesso.",
79           icon: "success",
80           showConfirmButton: false,
81         });
82         setTimeout(function() {
83           window.location.reload();
84         }, 2000);
85       } else {
86         Swal.fire("", "O jogo não foi atualizado.", "error");
87       }
88     } else {
89       Swal.fire("", "Atualização cancelada.", "info");
90     }
91   } catch (error) {
92     console.error("Erro ao atualizar o jogo", error);
93     Swal.fire("", "Ocorreu um erro ao atualizar o jogo.", "error");
94   }
95 }
96

```

Figura 32 – Função handleDeleteGame (catalogGames)

```
1  async function handleDeleteGame(gameId) {
2    try {
3      const result = await Swal.fire({
4        title: "Tem certeza?",
5        text: "Esta ação é irreversível!",
6        icon: "warning",
7        showCancelButton: true,
8        confirmButtonColor: "#d33",
9        cancelButtonColor: "#3085d6",
10       confirmButtonText: "Sim, exclua!",
11       cancelButtonText: "Cancelar",
12     });
13
14     if (result.isConfirmed) {
15       const response = await api.delete(`games/${gameId}`, {
16         headers: {
17           Authorization: `Bearer ${localStorage.getItem("token")}`,
18         },
19       });
20
21       if (response.status === 204) {
22         const updatedUserGames = userGames.filter(
23           (game) => game.id !== gameId
24         );
25         setUserGames(updatedUserGames);
26
27         Swal.fire("Excluído!", "O jogo foi excluído com sucesso.", "success");
28       } else {
29         Swal.fire("", "O jogo não foi excluído.", "error");
30       }
31     } else {
32       Swal.fire("", "Exclusão cancelada.", "info");
33     }
34   } catch (error) {
35     console.error("Erro ao excluir o jogo", error);
36     Swal.fire("", "Ocorreu um erro ao excluir o jogo.", "error");
37   }
38 }
```

## 8 Conclusão parte II

Em conclusão, a adoção de padrões de projeto, como a separação entre repositório e controlador, desempenha um papel fundamental no desenvolvimento de software, promovendo a manutenibilidade, escalabilidade e reutilização de código. A estrutura modular resultante facilita a realização de testes, implementação de novos recursos e manutenção do código, especialmente em sistemas complexos.

No contexto específico do projeto apresentado, a organização do back-end reflete a busca por boas práticas, desde a configuração inicial do ambiente de desenvolvimento até a implementação das rotas e controladores. A escolha do banco de dados SQLite e a integração eficiente visam simplificar o gerenciamento de dados, adequando-se às necessidades do projeto acadêmico.

A comunicação entre o front-end e o back-end é estabelecida de maneira consistente, com destaque para a utilização da biblioteca Axios e a configuração cuidadosa das rotas. A implementação das páginas no front-end reflete a preocupação com a experiência do usuário, incorporando validações robustas e oferecendo funcionalidades intuitivas, como a avaliação de jogos e a interação com o sistema.

Em resumo, a abordagem adotada no projeto não apenas atende aos requisitos técnicos, mas também reflete um compromisso com a segurança, eficiência e usabilidade. Ao seguir práticas organizacionais e arquiteturais sólidas, o desenvolvimento do software é orientado para a construção de sistemas robustos e adaptáveis, capazes de enfrentar os desafios das demandas em constante evolução.

## 9 Referências:

NIELSEN, Jacob; NORMANDO, Dom Nielsen Norman Group. nngroup, 1994. Disponível em: <https://www.nngroup.com/articles/ten-usability-heuristics/>. Acesso em: 21/10/2023 às 19:47h.

<https://www.npmjs.com/>

<https://nodejs.org/docs/latest/api/>

<https://www.npmjs.com/package/express>

<https://www.npmjs.com/package/body-parser>

<https://www.npmjs.com/package/cors>

<https://jwt.io/>

<https://www.npmjs.com/package/sqlite3>

<https://github.com/TryGhost/node-sqlite3>

<https://www.npmjs.com/package/react-router-dom>

<https://www.npmjs.com/package/react-simple-star-rating>

<https://www.npmjs.com/package/jwt-decode>

<https://www.npmjs.com/package/axios>

<https://sweetalert2.github.io/>

<https://react.dev/>