

**UNIFACS - UNIVERSIDADE SALVADOR**

**UC – USABILIDADE, DESENVOLVIMENTO WEB, MOBILE E JOGOS**

Douglas Dórea da Silva Melo - RA: 1272215019

Marcos Vinicius de Jesus Galvão - RA: 12722112437

Ricardo Brito da Silva - RA: 12722129482

Wilson Pinheiro dos Santos Neto - RA: 12723110270

GERENCIADOR DE CATÁLOGOS DE JOGOS:

PLATAFORMA DOMINUS PRO

SALVADOR

2023

Douglas Dórea da Silva Melo - RA: 1272215019

Marcos Vinicius de Jesus Galvão - RA: 12722112437

Ricardo Brito da Silva - RA: 12722129482

Wilson Pinheiro dos Santos Neto - RA: 12723110270

GERENCIADOR DE CATÁLOGOS DE JOGOS:

PLATAFORMA DOMINUS PRO

Este relatório tem como finalidade apresentar e documentar a visão final e dos processos de desenvolvimento da plataforma Dominus Pro.

Orientadores: Adailton de Jesus Cerqueira Junior e Lucas Silva dos Santos.

SALVADOR

2023

## Sumário

Introdução.....	4
Informações Gerais do Projeto.....	5
Wireframes.....	5
Telas Login e Registro.....	8
Tela Gerenciar Plataforma e Gerenciar Jogos.....	11
Tela Adicionar e Editar Jogos.....	14
Tela Documentação.....	17
Como executar a aplicação:.....	19
Rodando o backend (servidor):.....	19
Rodando o frontend (cliente):.....	21
Arquitetura, estratégia e algoritmos utilizados:.....	22
Servidor (API):.....	23
Rota:.....	24
Controller:.....	25
DTO (Data Transfer Object):.....	26
Use Case (Caso de Uso):.....	27
Repository (Repositório):.....	28
Fluxo da API:.....	29
Integrando Back-end x Front-end.....	30
Bibliografia.....	32

## Introdução

O presente relatório visa apresentar uma visão abrangente do projeto, incorporando informações essenciais. Além disso, evidenciaremos os wireframes das telas do sistema, oferecendo uma visualização inicial das interfaces propostas. Juntamente com isso, apresentaremos protótipos de alta fidelidade, aprimorando a compreensão visual do design proposto e mostrando a tela final.

Um aspecto crucial deste relatório é a análise dos princípios de usabilidade incorporados no processo de construção das telas e do sistema como um todo. A usabilidade é uma dimensão fundamental para a eficácia de qualquer aplicação, influenciando diretamente a experiência do usuário. Ao longo do documento, serão fornecidas argumentações detalhadas sobre como esses princípios foram integrados, destacando as decisões de design e interação que visam otimizar a navegabilidade e a compreensão do sistema por parte do usuário.

Assim, este relatório foi concebido com a intenção de ser um guia para compreender a fundo o projeto, desde suas bases conceituais até as representações visuais tangíveis.

## Informações Gerais do Projeto

A ideia da Dominus Pro é a criação dos jogos por usuário, nesse sentido a aplicação será como uma vitrine, mostrando todos os jogos adicionados pelo usuário com uma avaliação pessoal da sua experiência no jogo.

A Dominus Pro tem a proposta de ser intuitiva e de fácil navegação permitindo que até os usuários mais leigos consigam utilizar sem maiores problemas. Dentro da plataforma os usuários cadastram plataformas (essas que serão compartilhadas entre todos os usuários da aplicação) que serão utilizadas para cadastrar o jogo.

A Dominus Pro possui diversas validações para que não permita que uma plataforma seja modificada caso ela já possua um jogo associado por algum usuário, preservando os dados e a experiência do usuário na aplicação.

## Wireframes

De acordo com os wireframes, as nossas páginas foram idealizadas para terem um design minimalista, com layouts padronizados para que o usuário tenha uma melhor navegabilidade.

Os protótipos inseridos nesse documento sofreram alguns ajustes em suas resoluções e por consequência perderam um pouco da sua qualidade, para mitigar esse problema, segue o link para o google drive, onde encontra-se as imagens em uma qualidade melhor.

Link:

[https://drive.google.com/drive/folders/1JbEkBSEzJCE61CmHt5uqt2vMilFaW-eM?usp=drive\\_link](https://drive.google.com/drive/folders/1JbEkBSEzJCE61CmHt5uqt2vMilFaW-eM?usp=drive_link)

## Protótipos de Alta Fidelidade e Heurísticas

As heurísticas de Nielsen dão uma base para melhorar a usabilidade e a experiência do usuário dentro de uma aplicação e com base em algumas dessas heurísticas, desenvolvemos a Dominus Pro. Os protótipos inseridos nesse documento sofreram alguns ajustes e por consequência perderam um pouco da sua qualidade, para mitigar esse problema segue o link para o figma:

Link:

<https://www.figma.com/file/kASuCvsgDfucbRyCtRtgme/A3---Usabilidade%2C-desenvolvimento-web%2C-mobile-e-jogos?type=design&node-id=0-1&mode=design&t=D7IPJx7VROGEzr3h-0>

Esse tópico foi dividido em evidências e heurísticas, em cada imagem, vão ter números em vermelho que vão destacar quais heurísticas estão sendo evidenciadas para facilitar o entendimento para o leitor.

# Telas Login e Registro

- Wireframes

LOGO

Login

E-mail:

Senha:

Registrar

Já tem uma conta?  
[Faça login](#)

Site Title

LOGO

Login

Email

Senha

Entrar

Não tem conta?  
[Cadastre-se](#)

LOGO

Registro

Usuário:

E-mail:

Nome:

Senha:

Registrar

Já tem uma conta?  
[Faça login](#)

Site Title

LOGO

Registro

Usuário:

E-mail:

Nome:

Senha:

Registrar

Já tem uma conta?  
[Faça login](#)



- Protótipo



## REGISTRO

Usuário 3, 4, 5

Email

Nome

Senha

**Registrar**

Já tem uma conta? [Faça login](#)



## LOGIN

Email 3, 4, 5

Senha Preencha este campo. 1, 6, 7

**Entrar**

Não tem uma conta? [Cadastre-se](#)

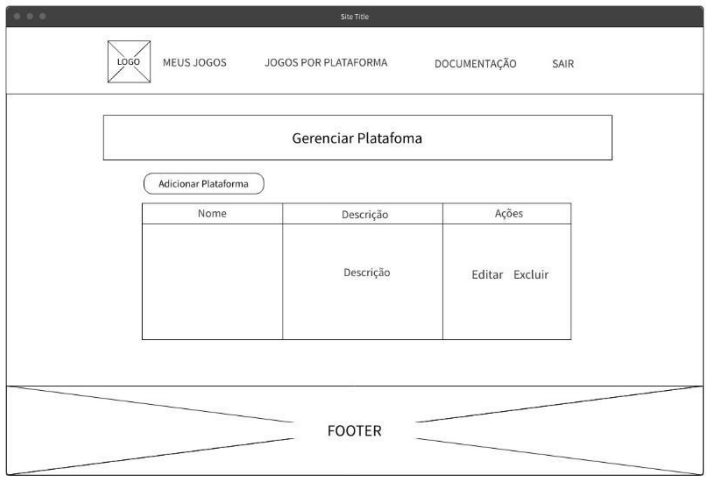
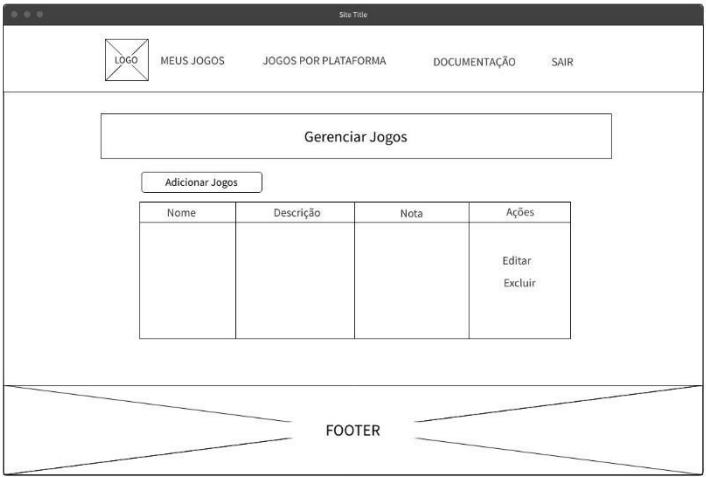
Heurísticas:

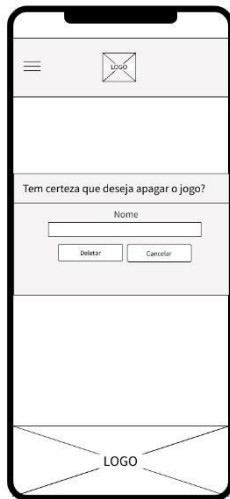
Visibilidade do Status do Sistema (1), Prevenção de Erros (7), Ajuda aos usuários a reconhecerem, diagnosticar e corrigir erros (6): Durante toda navegação nesta página, o usuário consegue se localizar com nomes claros e sugestivos. Essa página conta com mensagens de erros explícitas (atendendo também os padrões de segurança para uma aplicação) para que o usuário consiga entender o que está fazendo de errado e conseguir se recuperar durante a navegação na página.

Consistência e padrões (3), Reconhecimento em vez de memorização (4), Estética e design minimalista (5): As telas não têm informações desnecessárias, possuem um design simples, elegante e sugestivo. O design dos formulários das telas de registro e login são iguais, dando uma experiência sugestiva ao usuário, esse design é usual por diversas plataformas fazendo os usuários reconhecerem e ao se cadastrar na aplicação conseguirem gravar o fluxo (que é intuitivo) e fazer o login sem maiores problemas.

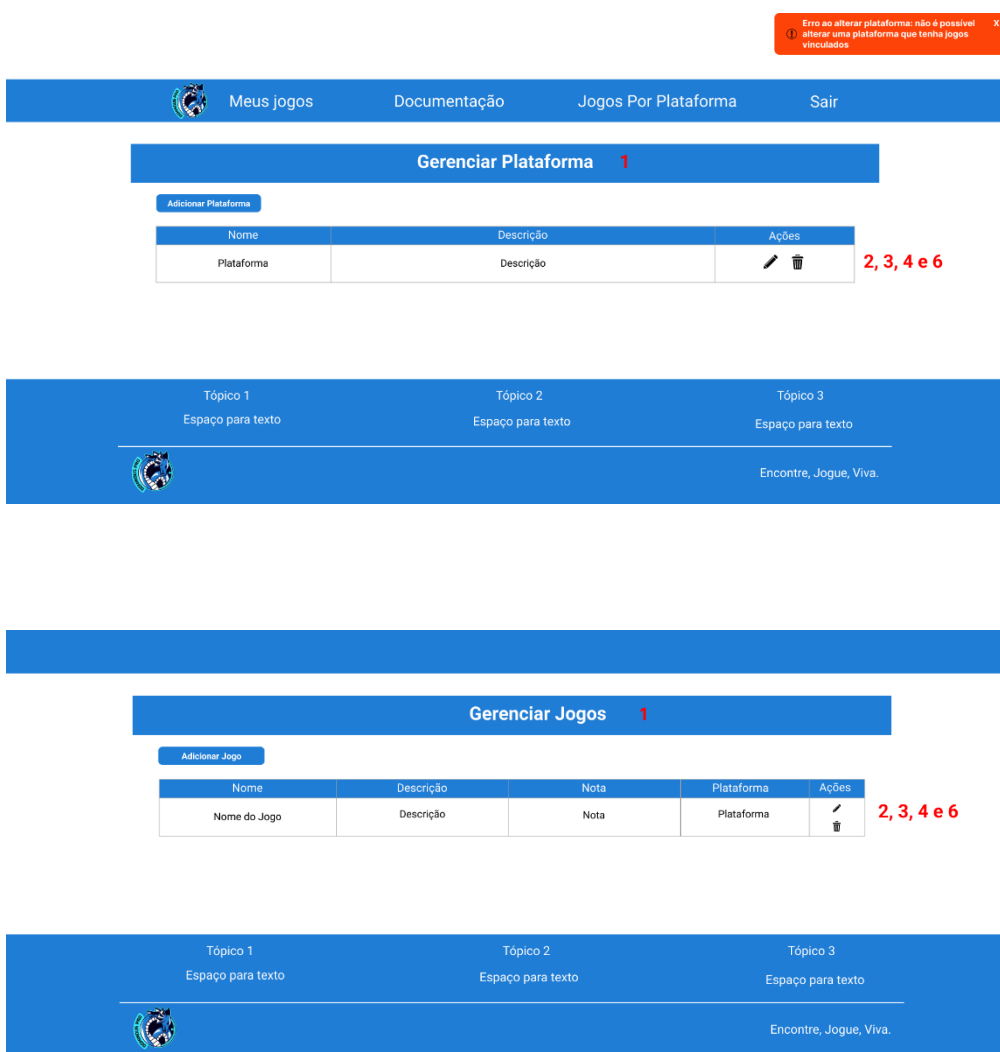
# Tela Gerenciar Plataforma e Gerenciar Jogos

- Wireframe:





- Protótipo:



Tem certeza que deseja excluir este jogo? 1

Nome do Jogo

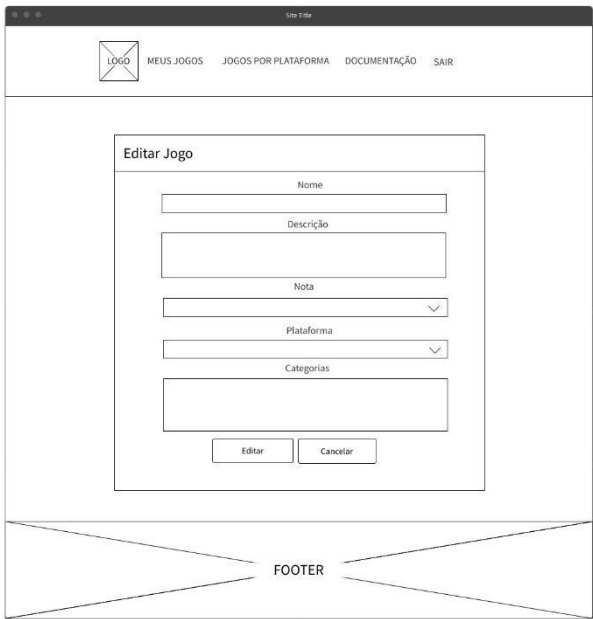
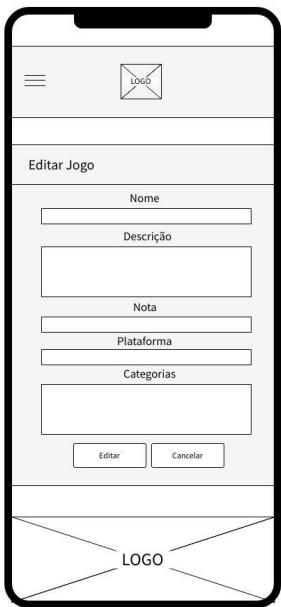
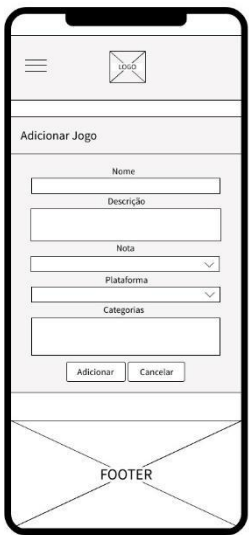
Excluir Cancelar

### Heurísticas:

Visibilidade do Status do Sistema (1), Consistência e padrões (3), Correspondência entre sistema e mundo real (2) Reconhecimento em vez de memorização (4), estética e design minimalista (5), Reconhecimento em vez de recordação (6), Ajuda os usuários a reconhecer, diagnosticar e recuperar erros (9): Durante toda navegação nesta página, o usuário consegue se localizar com nomes claros e sugestivos. As telas de gerenciamento foram desenvolvidas seguindo padrões consistentes, é possível notar que como são telas com intuitos similares (gerenciar) possuem o mesmo layout e organização fazendo com que o usuário não tenha que reaprender um novo fluxo em todas as telas da aplicação. É possível notar também que existem ícones na tabela que remetem ao uso em outros sistemas. As telas foram desenvolvidas para serem simples e direta, sem muitas informações desnecessárias dando um ar elegante e direto para aplicação, além de que caso algo dê errado, o erro será mostrado ao usuário em uma linguagem simples e objetiva das ações que causaram o erro e com isto o usuário conseguirá se recuperar do erro.

# Tela Adicionar e Editar Jogos

- Wireframes:



- Protótipo:

## Heurísticas:

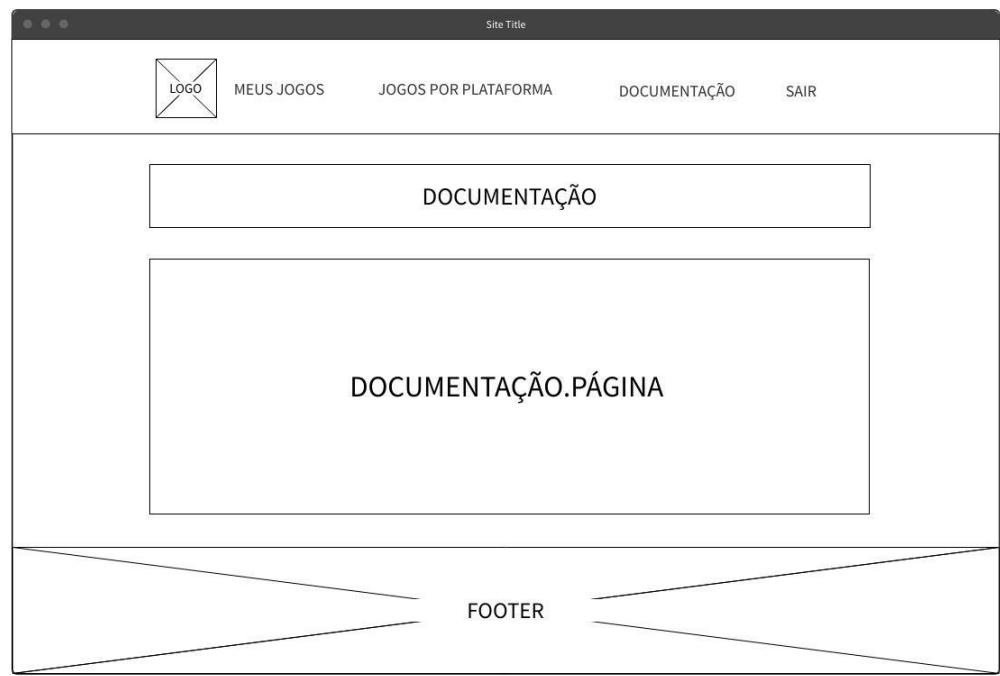
Visibilidade do Status do Sistema (1), Prevenção de Erros (7), Ajuda aos usuários a reconhecerem, diagnosticar e corrigir erros (6): Durante toda navegação nesta página, o usuário consegue se localizar com nomes claros e sugestivos. Essa página conta com mensagens de erros explícitas para que o usuário consiga entender o que está fazendo de errado e conseguir se recuperar durante a navegação na página.

Consistência e padrões (3), Reconhecimento em vez de memorização (4), Estética e design minimalista (5): As telas não têm informações desnecessárias, possuem um design simples, elegante e sugestivo. O design dos formulários nas telas é igual, dando uma experiência sugestiva ao usuário, esse design é usual por diversas plataformas fazendo os usuários reconhecerem, as cores dos botões também são sugestivas utilizando o vermelho para cancelar e o azul para adicionar.

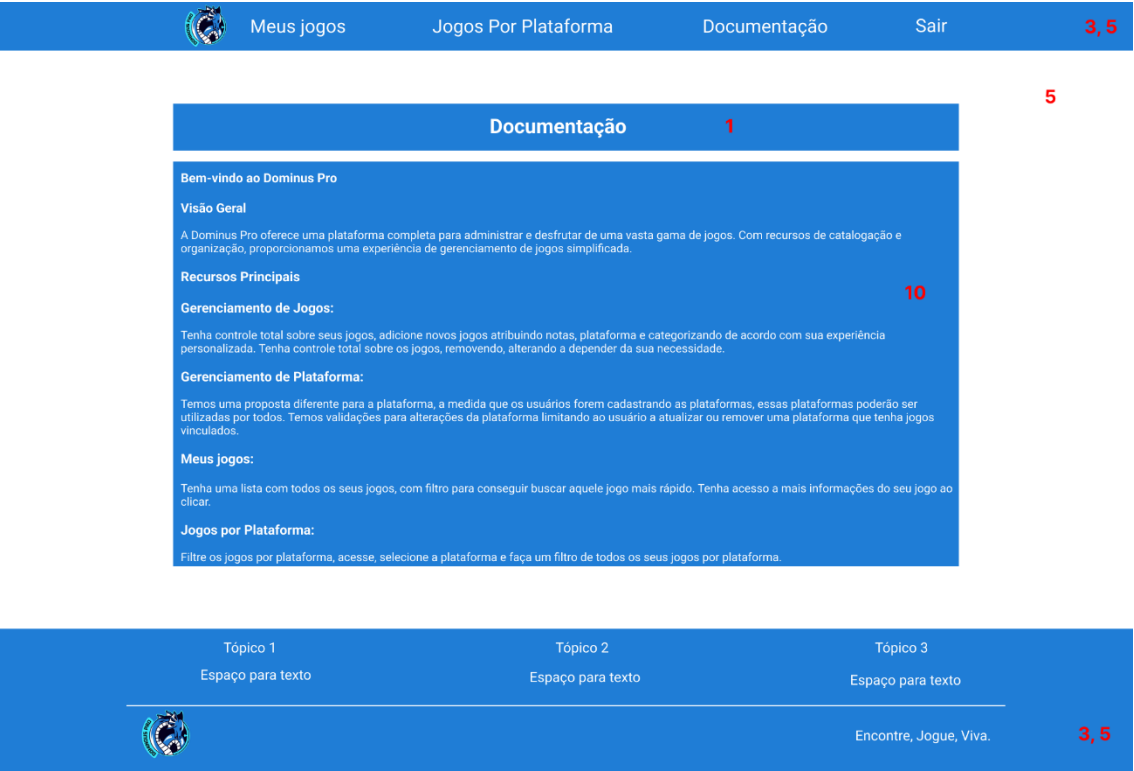


# Tela Documentação

- Wireframe:



- Protótipo:



Heurísticas:

Visibilidade do Status do Sistema (1), Consistência e padrões (3), Estética e design minimalista (5), Ajuda e Documentação (10): As telas não têm informações desnecessárias, possui um design simples, elegante e sugestivo, além de manter o padrão utilizado em toda a aplicação.

Esta tela foi desenvolvida com o intuito de auxiliar o usuário a resolver suas dúvidas, tornando-o mais independente de suporte.

Como executar a aplicação:

Este projeto é dividido em duas partes:

1. Backend (Servidor);
2. Frontend (Cliente);

**O front-end precisa que o back-end esteja sendo executado para funcionar.**

- Pré-requisitos:

Antes de começar, você vai precisar ter instalado em sua máquina as seguintes ferramentas: [Git](#), [Node.js](#), [MYSQL](#), Além disto é bom ter um editor para trabalhar com o código como [Visual Studio Code](#) e uma ferramenta para banco de dados como [DBEAVER](#).

## Rodando o backend (servidor):

- 1- Certifique-se de ter a versão **v18.17.1** do Node instalada, certifique-se de já possuir os pré-requisitos instalados.
- 2- Clone esse repositório: `git clone https://github.com/wilson6g/a3-usabilidade.git`;
- 3- Acesse a pasta do projeto que acabou de clonar e navegue até a pasta do back-end (`src/back-end`);
- 4- Instale as dependências utilizando: `npm install` ou `yarn install` (varia de acordo com o gerenciador de pacotes instalado na máquina do usuário);
- 5- Crie um arquivo chamado `“.env”` na raiz do projeto;
- 6- Copie os dados do `“.env-example”` e cole no `“.env”` que acabou de criar e mude para os dados do seu ambiente;
- 7- `APP_PORT=` insira a porta que deseja rodar o servidor, geralmente usa-se a `"3000"`.
- 8- `DATABASE_HOST=` aqui geralmente usa-se `"localhost"`.
- 9- `DATABASE_NAME=` insira o nome do banco de dados que você quer criar, ele vai gerar o banco pelo nome dessa variável.
- 10- `DATABASE_USERNAME=` insira o usuário do banco de dados, geralmente é `"root"`.
- 11- `DATABASE_PASSWORD=` insira a senha do seu banco de dados.
- 12- `DATABASE_PORT=` insira a porta, geralmente é: `"3306"`.
- 13- `DATABASE_SYNCHRONIZE=` insira `true` na primeira vez que for rodar o projeto para criar o banco e a tabela e depois coloque como `false`.
- 14- Execute a aplicação em modo de desenvolvimento `npm run dev` **OU** `yarn dev`

# O servidor iniciará na porta que foi inserida dentro do arquivo `.env` no campo `APP_PORT` por padrão.

**Observação:** É importante salientar que após iniciar o servidor pela primeira vez com o `“DATABASE_SYNCHRONIZE”` ativo (`true`) e a estrutura do banco de dados seja criada, recomendamos que desligue o servidor e mude essa variável de ambiente para falso (`false`) e rode novamente.

### Rodando o frontend (cliente):

- 1- Certifique-se de ter a versão **v18.17.1** do Node instalada, certifique-se de já possuir os pré-requisitos instalados.
- 2- Clone esse repositório: `git clone https://github.com/wilson6g/a3-usabilidade.git`;
- 3- Acesse a pasta do projeto que acabou de clonar e navegue até a pasta do front-end (`src/front-end`);
- 4- Instale as dependências utilizando: `npm install` ou `yarn install` (varia de acordo com o gerenciador de pacotes instalado na máquina do usuário);
- 5- Crie um arquivo chamado `“.env”` na raiz do projeto;
- 6- Copie os dados do `“.env-example”` e cole no `“.env”` que acabou de criar e mude para os dados do seu ambiente;
- 7- `VITE_BASE_URL= http://localhost:"Porta definida na aplicação Back-end"` .
- 8- Execute a aplicação em modo de desenvolvimento `npm run dev` **OU** `yarn dev`
- 9- O cliente iniciará na porta:5173 por padrão - acesse `http://localhost:5173`

**Observação:** Essa aplicação só vai funcionar perfeitamente com o back-end em node rodando em paralelo, ou seja, ao mesmo tempo.

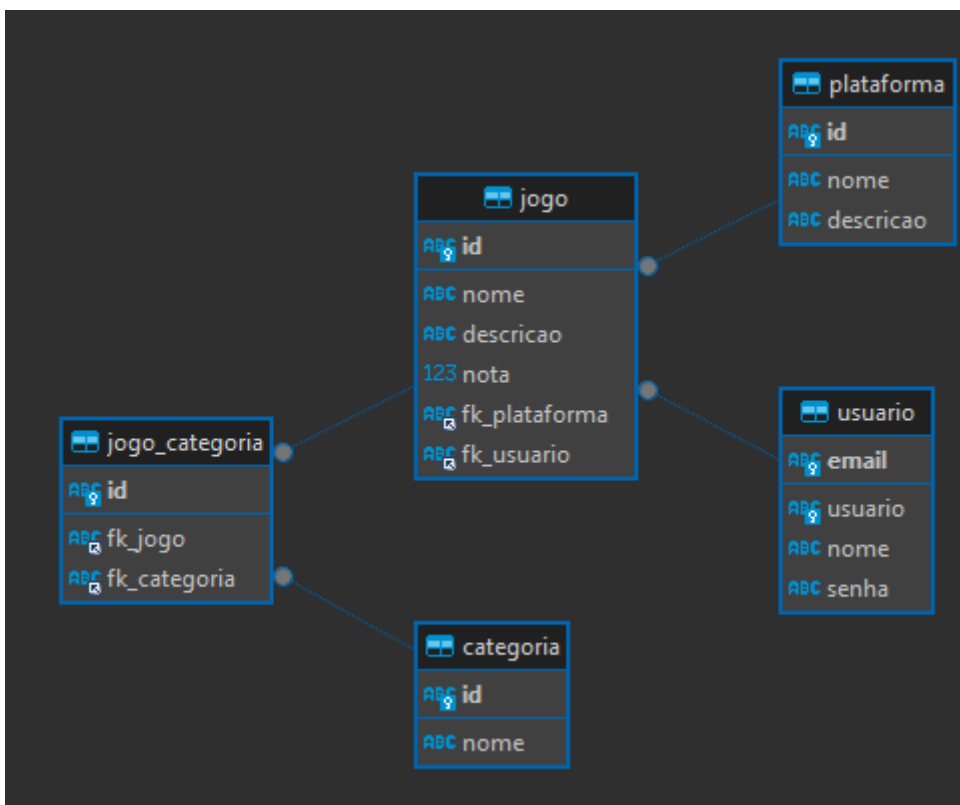
## Arquitetura, estratégia e algoritmos utilizados:

- Banco de Dados

Para o desenvolvimento da aplicação, utilizamos o banco MYSQL pela facilidade de integração com tecnologias existentes e pela velocidade e desempenho.

- Modelo Entidade Relacionamento

A aplicação foi desenvolvida seguindo a ideia de cadastrar vários jogos por usuário e cada jogo ter uma plataforma específica, a ideia também é que um jogo possa ter diversas categorias.



## Servidor (API):

A aplicação foi desenvolvida em Node.JS utilizando uma arquitetura em camadas, essa abordagem foi escolhida para manter o código com uma alta manutenibilidade e com o objetivo de separar o máximo de responsabilidade possível.

A estrutura final do projeto e suas responsabilidades foram divididas da seguinte forma:

- controller: Responsável por receber inputs e validar os dados para serem enviados ao caso de uso;
- dto: Ele vai devolver um objeto com os atributos (chave, valor) necessários para o caso de uso;
- routes: Gerencia as rotas (endpoints) da aplicação, criando requisições (POST, GET, PUT, DELETE) e mostrando o fluxo o qual deve ser seguido.
- use-cases: Armazena os casos de uso da aplicação. Cada caso de uso é uma interação específica do usuário, encapsulando a regra de negócio;
- util: Contém funções que fornecem funcionalidades comuns reutilizáveis em toda aplicação.
- repository: lida com a interação com o banco de dados.
- config: faz a conexão do banco de dados com a API.

server.js: arquivo que funciona como ponto de entrada da aplicação, onde é configurado o servidor e conecta todas as partes.

Rota:


A rota faz o direcionamento das requisições HTTP para as funções com base na URL (PATH) e método HTTP.

```
1  const express = require("express");
2  const jogoCategoriaRoutes = express();
3  const validarCampoBuscarMeuJogo =
4    require("../controllers/meus-jogos-controller").validarCampoBuscarMeuJogo;
5  const validarCamposAlterarMeuJogos =
6    require("../controllers/meus-jogos-controller").validarCamposAlterarMeuJogos;
7  const validarCamposCriarMeuJogo =
8    require("../controllers/meus-jogos-controller").validarCamposCriarMeuJogo;
9
10 const criarMeusJogosCategoriaUseCase = require("")
11
12 const buscarMeuJogoUseCase = require("../use-case/meus-jogos-use-cases/buscar-meus-jogos-use-case");
13 const criarMeusJogosUseCase = require("../use-case/meus-jogos-use-cases/criar-meus-jogos-use-case");
14 const listarMeusJogosUseCase = require("../use-case/meus-jogos-use-cases/listar-meus-jogos-use-case");
15
16 jogoCategoriaRoutes.post("/meus-jogos", async (request, response) => {
17   try {
18     const input = request.body;
19
20     const inputValido = validarCamposCriarMeuJogo(input);
21
22     if (!inputValido) {
23       throw {
24         message: "Os dados de entrada não são válidos.",
25         status: 422,
26       };
27     }
28
29     const meuJogo = await criarMeusJogosUseCase(input);
30   }
```



## Controller:

A função do controller dentro dessa aplicação é para validação de dados, cada caso de uso, vai ter um controller específico que vai fazer uma validação dos dados de entrada e pode ou retornar um erro ou um DTO.

```
1  const validarUUID = require("../util/validar-uuid");
2  
3  function validarCamposCriarCategoria({ nome }) {
4      if (nome.length === 0) {
5          throw {
6              message: `O campo 'nome' não deve ser vazio.`,
7              status: 422,
8          };
9      } else if (nome.length > 32) {
10         throw {
11             message: `O campo 'nome' não pode ter mais de 32 caracteres.`,
12             status: 422,
13         };
14     } else if (typeof nome !== "string") {
15         throw {
16             message: `O campo 'nome' deve ser do tipo string.`,
17             status: 422,
18         };
19     }
20     return true;
21 }
22
23 function validarCampoBuscarCategoria(id) {
24     const uuidValido = validarUUID(id);
25
26     if (!uuidValido) {
27         throw {
28             message: `O campo 'id' não é do tipo uuid, portanto, não é válido.`,
29             status: 422,
30         };
31     }
32
33     return true;
34 }
35
36 module.exports = { validarCamposCriarCategoria, validarCampoBuscarCategoria };
37
```

## DTO (Data Transfer Object):

O DTO é um objeto de transferência de dados, nessa aplicação, ele vai receber os dados no controller caso não tenha nenhum problema e vai retornar um objeto com os dados necessários para um caso de uso em específico.

```
1 function alterarJogoCategoriaDTO(input) {
2   return {
3     id: input.jogo_categoria.map(
4       (jogo_categoria) => jogo_categoria.jogo_categoria_id
5     ),
6     fk_jogo: input.id,
7     fk_categoria: input.categorias.map((categoria) => categoria.categoria_id),
8   };
9 }
10
11 module.exports = alterarJogoCategoriaDTO;
12
```

```
1 function apagarJogoCategoriaDTO({ id, categorias }) {
2   return {
3     fk_jogo: id,
4     fk_categoria: categorias,
5   };
6 }
7
8 module.exports = apagarJogoCategoriaDTO;
9
```

```
1 const crypto = require("crypto");
2
3 function criarJogoCategoriaDTO({ fk_jogo, fk_categoria }) {
4   return {
5     id: crypto.randomUUID(),
6     fk_jogo,
7     fk_categoria,
8   };
9 }
10
11 module.exports = criarJogoCategoriaDTO;
12
```

## Use Case (Caso de Uso):

O caso de uso vai receber os dados (DTO) da rota após todas as validações e vai ser a camada que vai ter comunicação com o repositório (que interage com o banco de dados), ele também vai ser responsável por validar regras de negócios da aplicação e aplicar a lógica dos casos de uso.

```
1  const alterarJogoDTO = require("../dto/jogo-dto/alterar-jogo-dto");
2  const alterarJogoRepository = require("../repository/jogo-repository/alterar-jogo-repository");
3  const buscarJogoPorNomeEUsuarioRepository = require("../repository/jogo-repository/buscar-jogo-por-nome-e-usuario-repository");
4  const alterarJogoCategoriaUseCase = require("../jogo-categoria-use-cases/alterar-jogo-categoria-use-case");
5  const buscarJogoPorIdUseCase = require("../buscar-jogo-id-use-case");
6
7  async function alterarJogoUseCase(input) {
8    try {
9      const jogo = alterarJogoDTO(input);
10
11      const jogoExistente = await buscarJogoPorIdUseCase(input);
12
13      const jogoExistenteNome = await buscarJogoPorNomeEUsuarioRepository(
14        jogo.nome,
15        jogo.fk_usuario
16      );
17
18      if (!jogoExistente) {
19        const error = new Error("O jogo não existe.");
20        error.statusCode = 404;
21        throw error;
22      } else if (jogoExistenteNome && jogo.id !== jogoExistenteNome?.id) {
23        const error = new Error("O jogo já existe.");
24        error.statusCode = 400;
25        throw error;
26      }
27
28      await alterarJogoCategoriaUseCase(jogoExistente, jogo);
29
30      const jogoAtualizado = await alterarJogoRepository(jogo.id, jogo);
31
32      return jogoAtualizado;
33    } catch (error) {
34      const statusCode = error.statusCode || 500;
35      throw {
36        message: `Erro ao atualizar o jogo: ${error.message}`,
37        status: statusCode,
38      };
39    }
40  }
41
42  module.exports = alterarJogoUseCase;
43
```

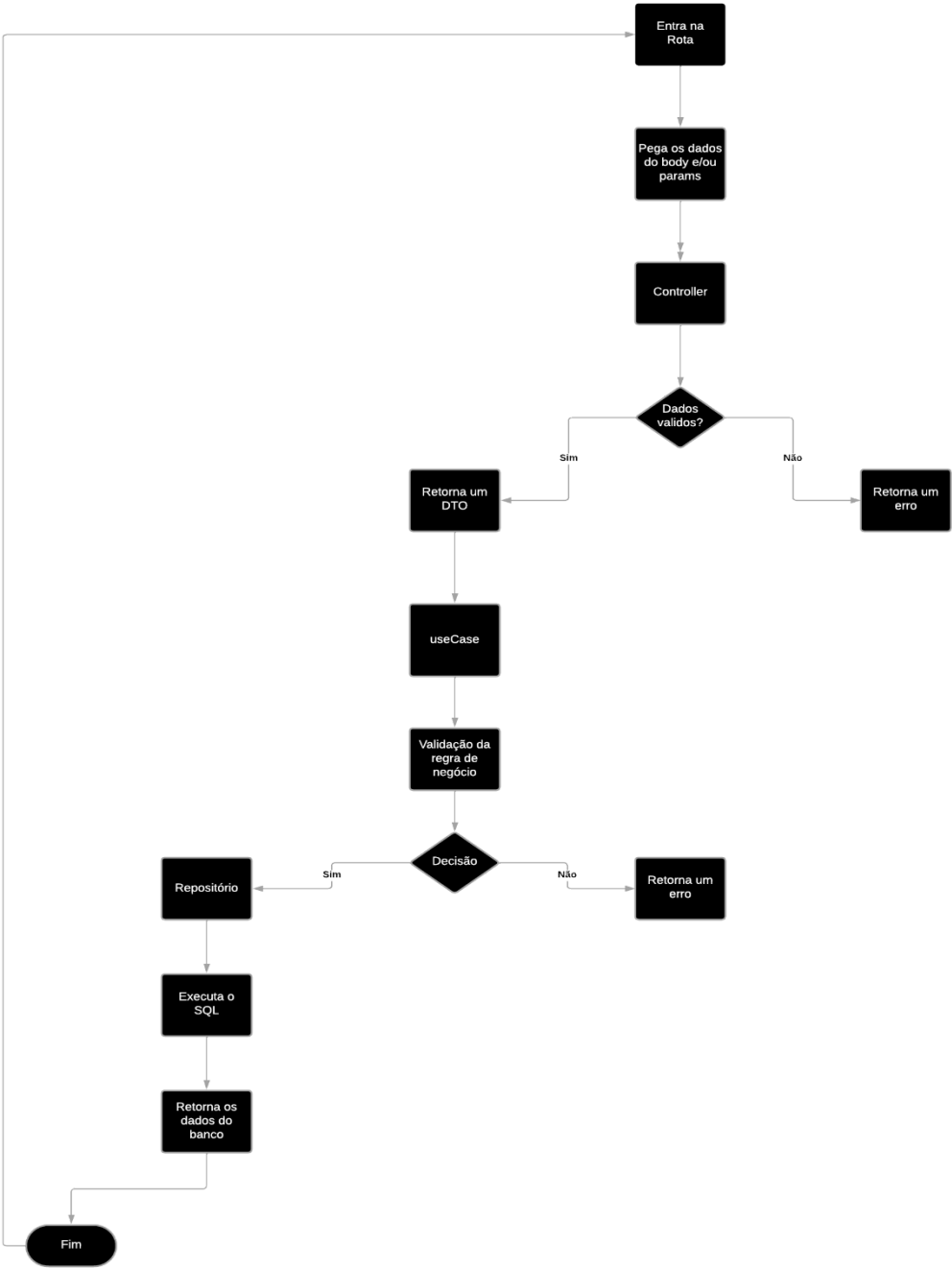
## Repository (Repositório):

O repositório nesse contexto é a camada de comunicação direta com o banco de dados, nesta camada que vai ter a consulta SQL e vai acontecer a comunicação com o banco de dados. Essa camada vai ser responsável por desempenhar o papel de interagir com o banco de dados e realizar alguma operação do CRUD (Create, Read, Update, Delete).

```
1  const database = require("../config/database");
2
3  async function buscarUsuario(coluna, input) {
4      try {
5          const [rows, fields] = await database.query(
6              `SELECT * FROM usuario WHERE ${coluna} = ?`,
7              [input]
8          );
9
10         return rows;
11     } catch (error) {
12         throw error;
13     }
14 }
15
16 module.exports = buscarUsuario;
17
```

```
1  const database = require("../config/database");
2
3  async function criarUsuario({ usuario, nome, email, senha }) {
4      try {
5          await database.query(
6              "INSERT INTO usuario (usuario, nome, email, senha) VALUES (?, ?, ?, ?)",
7              [usuario, nome, email, senha]
8          );
9
10         // Consultar o banco de dados para retornar os dados do usuário inserido
11         const [rows] = await database.query(
12             "SELECT * FROM usuario WHERE usuario = ?",
13             [usuario]
14         );
15
16         // Retornar o usuário inserido
17         return rows;
18     } catch (error) {
19         throw error;
20     }
21 }
22
23 module.exports = criarUsuario;
24
```

Fluxo da API:



## Integrando Back-end x Front-end

```
.env
1 VITE_BASE_URL= http://localhost:4000
```

No arquivo `.env`, é onde contém a base da URL da API, essa é uma boa prática no mercado de desenvolvimento, por exemplo, se mudasse o domínio do back-end só precisaria mudar a URL no `.env` e já refletiria para toda aplicação.

```
1 import axios from "axios";
2
3 const BASE_URL = import.meta.env.VITE_BASE_URL || "http://localhost:4000";
4
5 const instance = axios.create({
6   baseURL: BASE_URL,
7   headers: {
8     "Access-Control-Allow-Origin": "*",
9     "Content-Type": "application/json; charset=UTF-8",
10  },
11 });
12
13 export { instance };
14
```

No arquivo `"/shared/axios/axios.js"` é onde a instância é criada e exportada.

Na linha 3 é onde está sendo importada a URL criada no arquivo `.env`

Na **const instance**, é onde está sendo criado o objeto que será exportado para aplicação, onde como parâmetros estão sendo passado o `BASE_URL` e os headers, onde informa que está recebendo acesso de todas as origens e informando que o tipo de conteúdo que irá ser recebido será do tipo JSON.

```

1  import { instance } from "../../axios/axios";
2
3  async function loginService(input) {
4    const output = await instance.post("/login", input);
5
6    return output.data;
7  }

```

Na função login-service temos uma função que ao ser chamada irá fazer a requisição para o path "/login" da API e assim que tiver o retorno da API irá retornar os dados.

```

const onSubmit = async (event) => {
  event.preventDefault();
  try {
    const { data } = await loginService(values);

    if (data && data?.auth) {
      toast.success(data.message);
      localStorage.setItem("token", data?.token);
      setUserEmail(values.email);
      navigate("/library");
    }
  } catch (error) {
    toast.error(error.response.data.error);
  }
};

```

Assim que os dados forem retornados, vai ser verificado se o data existe e se o data.auth é verdadeiro. Se for verdadeiro, será retornada uma mensagem de sucesso, depois será adicionado o token no local storage que será utilizado posteriormente para fazer requisições na API.

## Bibliografia

<https://brasil.uxdesign.cc/10-heur%C3%ADsticas-de-nielsen-para-o-design-de-interface-58d782821840>

<https://www.npmjs.com/package/mysql2>

<https://expressjs.com/pt-br/>

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

<https://vitejs.dev/guide/>