UNIVERSIDADE SALVADOR – UNIFACS

Ciência da Computação

Equipe:

Manoel Duran - RA: 1272214667,

Nicolas Batista Lima - RA: 1272217454,

Raphael Oliveira - RA: 1272211685,

Hélio Júnior - RA: 12722129211

Data: 22/11/2023

RELATÓRIO DE DESENVOLVIMENTO WEB E USABILIDADE

Salvador, BA

2023

Introdução

Este projeto tem o objetivo de criar uma plataforma de jogos onde um usuário possa cadastrar plataformas, categorias e jogos. Nos jogos, é possível adicionar uma nota e várias categorias com status diferentes.

Design de Experiência do Usuário e Interface:

A fim de projetar uma interface que não gere momentos de insegurança para o usuário, que deixe claro quais serão os resultados de suas ações e garantir que o mesmo realize todas as tarefas de forma simples e eficiente, a equipe desenvolveu o software com base em diversas heurísticas de Nielsen. Dentre elas, as que mais se destacaram de maneira evidente no layout inicial foram:

- Consistência e padrões: Desde o início, a equipe se reuniu para pensar na estética e padrão da plataforma, visando manter a consistência entre as telas, tornando-a mais intuitiva e facilitando a experiência do usuário.
- Prevenção de Erros: Também pensando na Experiência de Usuário (UX), foi implementado com base nessa heurística as verificações de erros em toda a plataforma.
- Estética e Design Minimalista: Também é importante salientar que a Interface de Usuário (UI) foi desenhada com base na heurística de estética e design minimalista, diminuindo a quantidade de informações na tela e permitindo consequentemente com que os usuários tenham mais clareza e foco para entender o que deve for transmitido.
- Feedback ao usuário: A todo momento, deixamos em evidência todas as ações realizadas pelo usuário com notificações informando o que foi realizado dentro do site.

Geral:

A API é feita em Express e o frontend foi feito em React. Usamos a arquitetura MVC com partes de clean code e com princípios do SOLID no backend e os princípios de design de composição do React.

Infraestrutura:

Utilizamos o banco de dados SQLite. O arquivo src/infrastructure/database/`connection.js` `contém a classe DatabaseConnection, responsável` pela conexão com o banco de dados. Essa classe possui o método "`exec`" usado primariamente para executar comandos SQL para inserção, atualização e deleção de dados, além dos métodos "`query`", que utilizamos para executar consultas SQL que retornam várias linhas, e a "`queryOne`", para consultas que retornam apenas 1 linha.

O arquivo src/infrastructure/database/queries/migrations/`init.js` contém uma lista de comandos SQL utilizados para a criação do esquema de banco de dados, contendo as devidas relações e restrições. O arquivo src/infrastructure/database/queries/fixtures/`fixtures.js contém os comandos` SQL `responsáveis` pela inserção de dados no banco. Os comandos de criação das tabelas e inserção de dados iniciais no banco são executados durante a inicialização da classe DatabaseConnection, estando a invocação dos métodos necessários no construtor da classe.

Backend:

Para esse projeto, o backend segue um padrão bem organizado, dividindo claramente as responsabilidades em diferentes camadas:

1. Model: Utilizado para definir os atributos das entidades.
2. Repository: Utilizado para realizar a comunicação direta com o banco de dados, invocando os métodos `exec`, `query` e `queryOne` da instância de `DatabaseConnection` que foi injetada no construtor do próprio repositório. Todas elas, recebem uma query especifica que fica na pasta `/queries` do módulo.
3. Service: Responsável pela implementação das regras de negócio. Utiliza-se de uma instância de um repositório que é injetada em seu construtor.
4. Controller: Utilizado para lidar com a camada de transporte, sendo responsável pelo tratamento de erros e chamando os métodos do Service injetado em seu construtor.
5. Module: Utilizado para injeção de dependência entre as classes e invocação da função que declara as rotas do módulo.
6. Routes: Utilizado para as rotas do controller. As rotas recebem os middlewares `validationMiddleware` **e** `validationUrlParam` de acordo com a necessidade de cada rota nas requisições HTTP: GET, POST, PUT e DELETE.

Middlewares:

A aplicação possui 3 middlewares principais:

1. errorHandlingMiddleware

   Responsável pelo tratamento de erros. Esse middleware é executado no final da cadeia de middlewares de cada rota quando o bloco “catch” da rota é executado. Caso a instância do erro possua o campo “statusCode”, ele é

enviado como status na resposta. Caso contrário, a resposta emite um status 500 (Internal Server Error).

2. validateUrlParam

   Responsável pela validação de parâmetros passados via URL. Como nossa aplicação espera apenas parâmetros de ID no caminho da url, esse middleware valida se o parâmetro passado é um inteiro. Caso contrário, retorna um erro 400 (Bad Request).

3. validationMiddleware

   Essa função recebe uma função de validação, que deve ser uma outra função que deve checar se o corpo da requisição (POST ou PUT) está de acordo com o esperado para a rota.
   Retorna uma função closure, que executa a função de validação antes de chamar o próximo handler da rota ou trata um eventual erro de validação, retornando um erro 400 (Bad Request).

Frontend:

No arquivo `apiClient.js`, estamos facilitando a comunicação entre o nosso frontend e o backend por meio de uma abstração dos verbos HTTP comuns: GET, POST, PUT e DELETE. Isso é alcançado através da criação de um serviço dedicado para realizar requisições à API do backend de forma clara e organizada.

A função principal, `fetchFromBaseUrl`, age como um construtor genérico de requisições. Ela encapsula a lógica comum de fazer uma requisição HTTP, incluindo a construção do URL completo, a definição dos headers padrão e o tratamento de erros.

Essa estrutura facilita a manutenção, já que qualquer modificação na forma como as requisições são feitas pode ser feita de maneira centralizada no

`fetchFromBaseUrl` ou na criação dos métodos específicos no `apiClient`, também evitando repetição de código.

Rotas:

Para a implementação das rotas da aplicação, foi utilizada a biblioteca react-router. Para melhor organização, é utilizada uma pasta “pages”, que contém os arquivos com os componentes que devem ser renderizados em cada uma das rotas, que estão declaradas em src/routes/Routes.jsx.

Componentes:

Para melhor organização e maior facilidade na manipulação de código, separamos os pedaços da interface de usuário em vários componentes, todos presentes na pasta “components”.

Os componentes foram separados de forma a facilitar o gerenciamento de estado, de forma a evitar a passagem de estado exagerada entre diferentes componentes através de propriedades (“props” ou “prop drilling”). Alguns dos componentes são pequenas partes da interface de usuário que são reutilizadas extensivamente, como “FormField” e “Button”, de forma a evitar repetição de código.

Alguns dos formulários são também reutilizados para criação e edição de recursos, ficando o componente mais acima na árvore de renderização responsável por passar uma função a ser executada quando o formulário for submetido.

Finalmente, um simples componente AuthGuard é utilizado nas rotas protegidas. Esse componente checa se o usuário está autenticado, e caso contrário, redireciona o usuário para a tela de login.

Hooks:

Foram criados hooks customizados para evitar repetição de código e aderir às boas práticas do desenvolvimento em React. Os hooks customizados criados tem,

em geral, a responsabilidade de buscar dados da api e retorná-los, de forma a evitar o uso excessivo de useEffect diretamente nos componentes da aplicação.

authContext:

É utilizado um contexto (“Context”) do React para gerenciamento do estado de autenticação do usuário. O context foi a escolha ideal para esse cenário pois permite o compartilhamento de estado em toda a árvore de componentes, permitindo assim que os componentes que precisam de informação sobre o usuário possa acessar essa informação sem a necessidade de prop drilling.

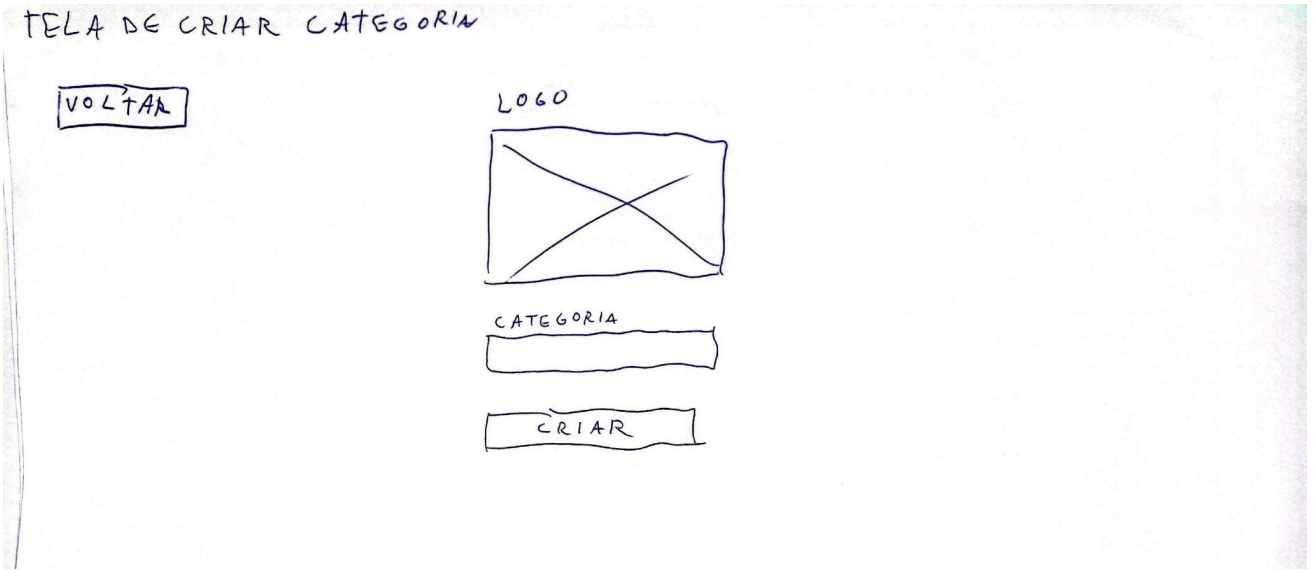
Conclusão:

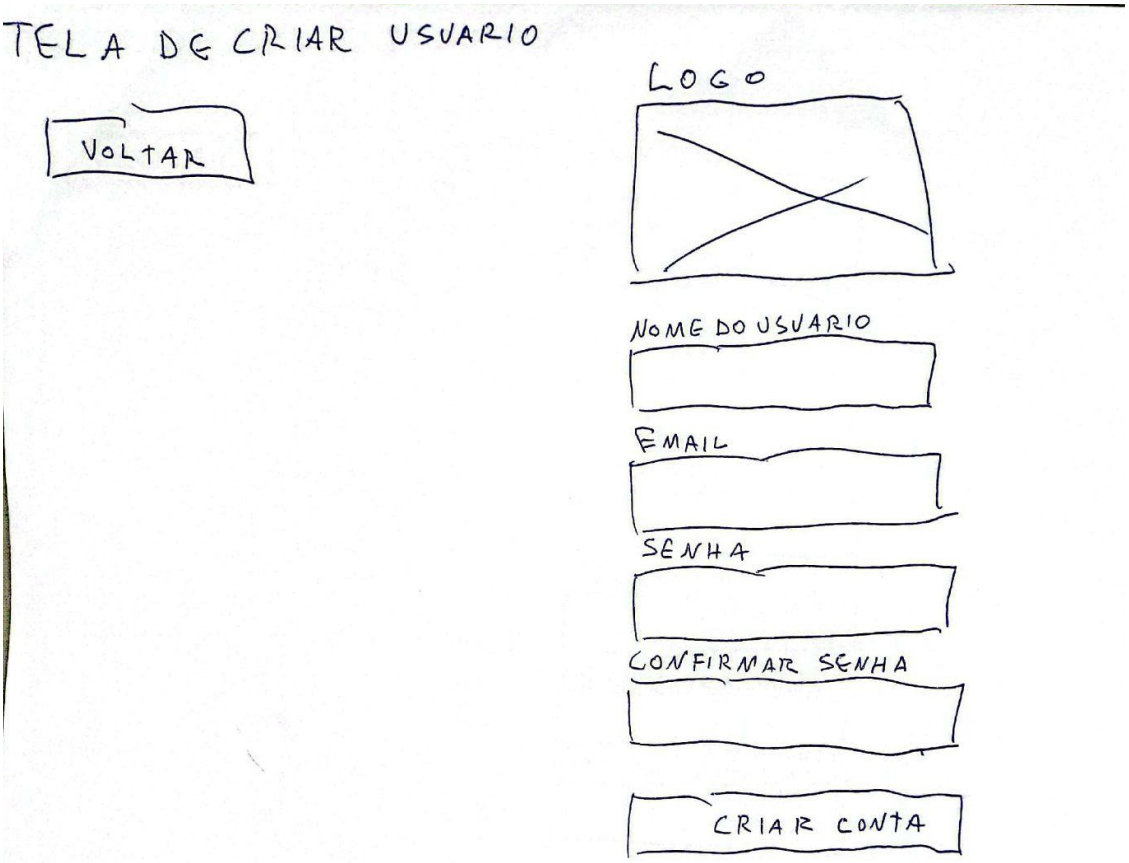
O nosso projeto de desenvolvimento de uma plataforma de jogos se destaca por sua forte ênfase na usabilidade e na experiência do usuário, aplicando heurísticas de Nielsen, como consistência, prevenção de erros, estética e design minimalista, e feedback ao usuário.

Essa estruturação contribui significativamente para a manutenção do sistema, já que as alterações podem ser feitas de maneira modular e sem afetar outras partes do código. Além disso, a clareza na definição das responsabilidades de cada camada tanto no back end quanto no frontend promove uma melhor escalabilidade e compreensão do sistema na totalidade.

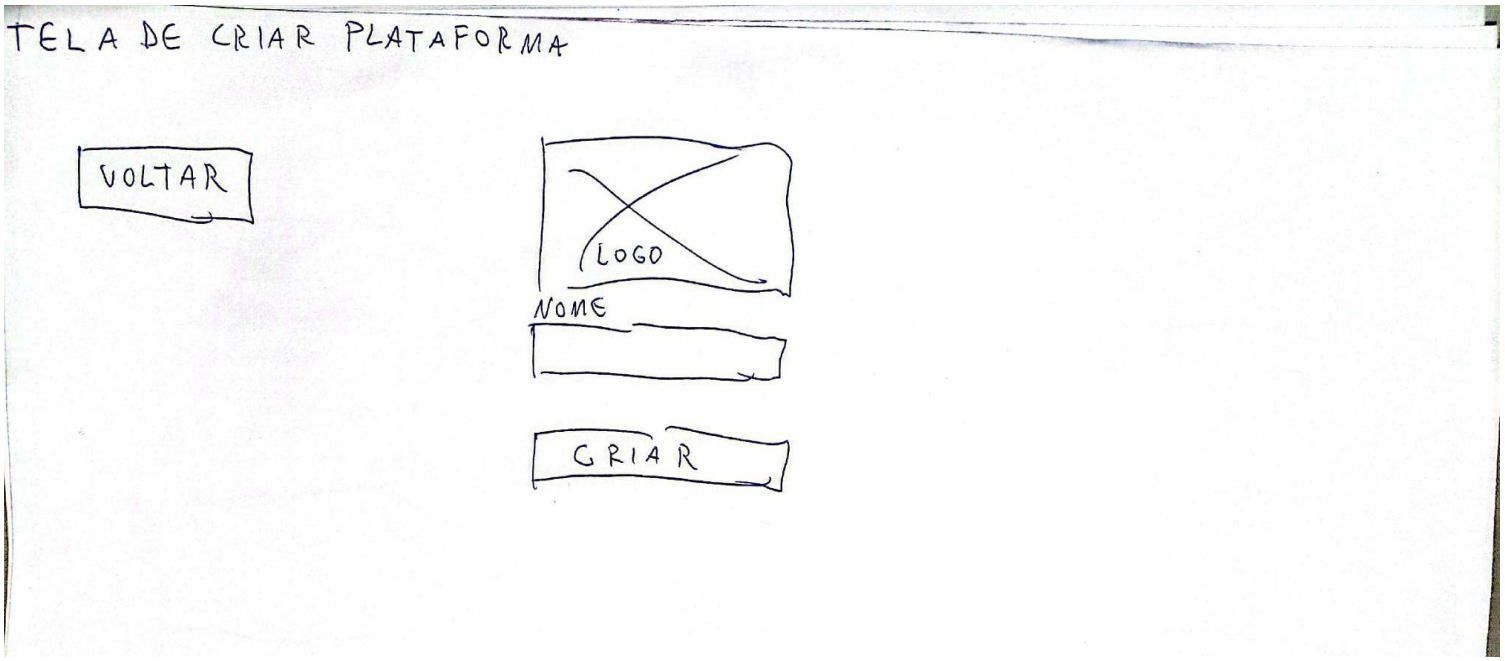
Em resumo, a abordagem adotada tanto no back end quanto no frontend é bastante organizada, seguindo boas práticas de desenvolvimento e promovendo a escalabilidade, manutenção e compreensão do código.

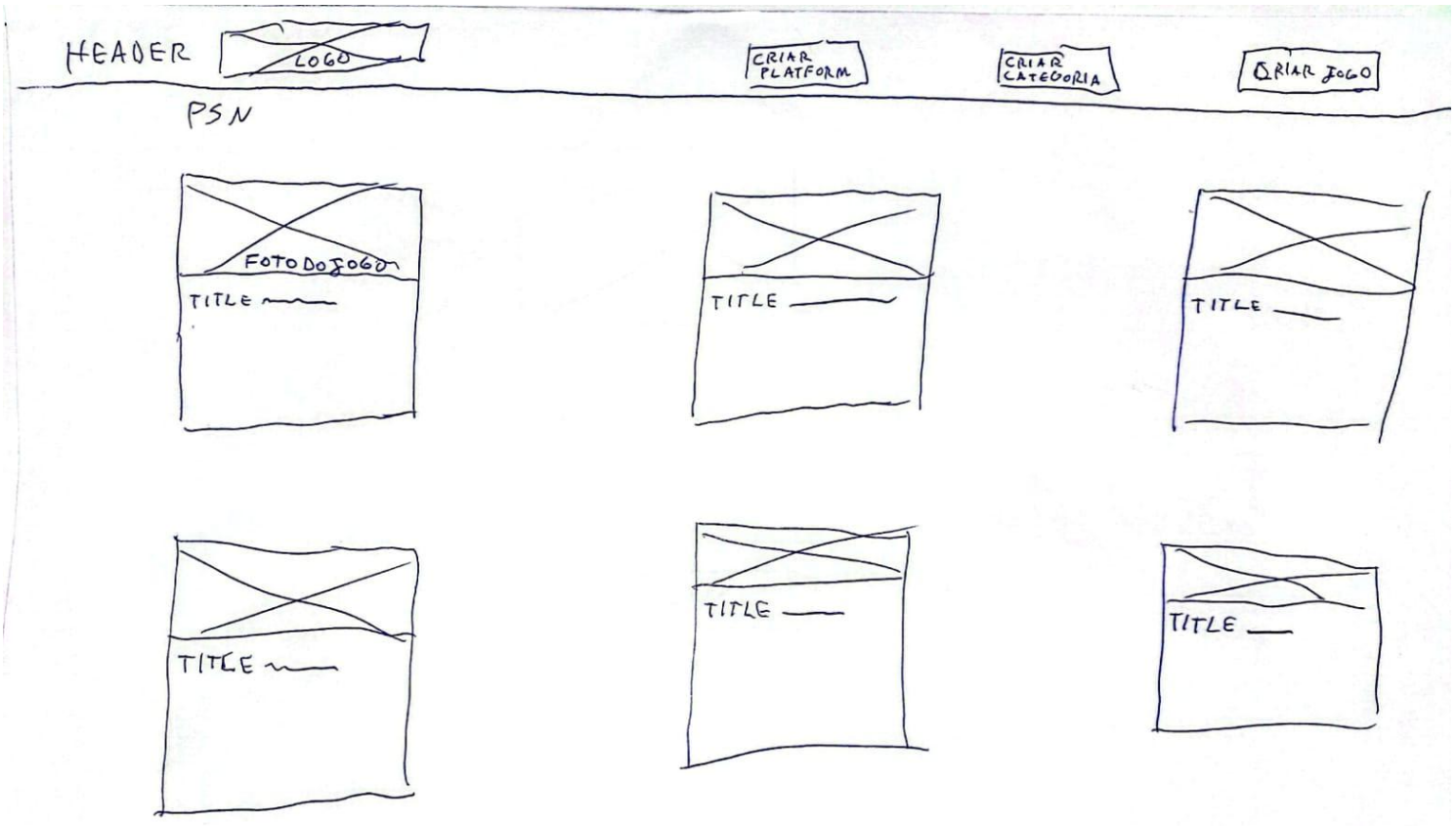
Wireframes:


TELA DE CRIAR CATEGORIA
VOLTAR
LOGO
CATEGORIA
CRIAR


TELA DE CRIAR USUARIO
VOLTAR
LOGO
NOME DO USUARIO
EMAIL
SENHA
CONFIRMAR SENHA
CRIAR CONTA

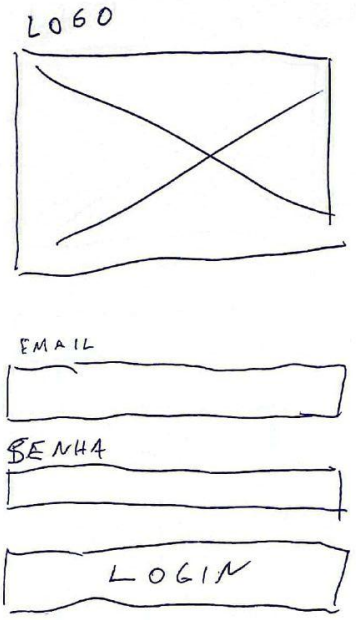
TELA DE CRIAR PLATAFORMA



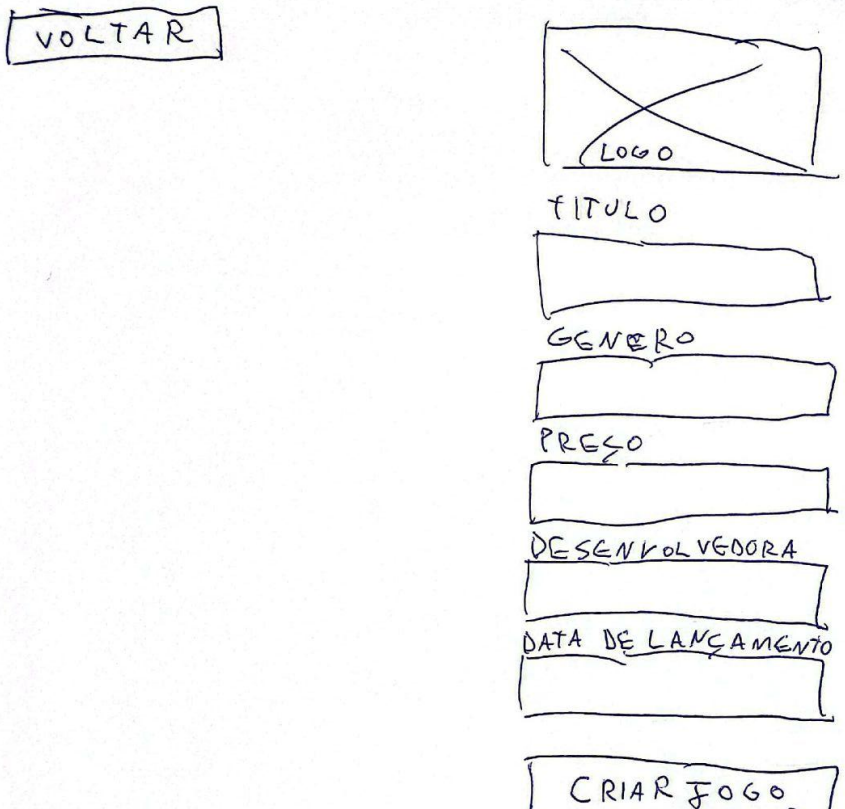


TELA DE PLATAFORMAS

TELA DE LOGIN

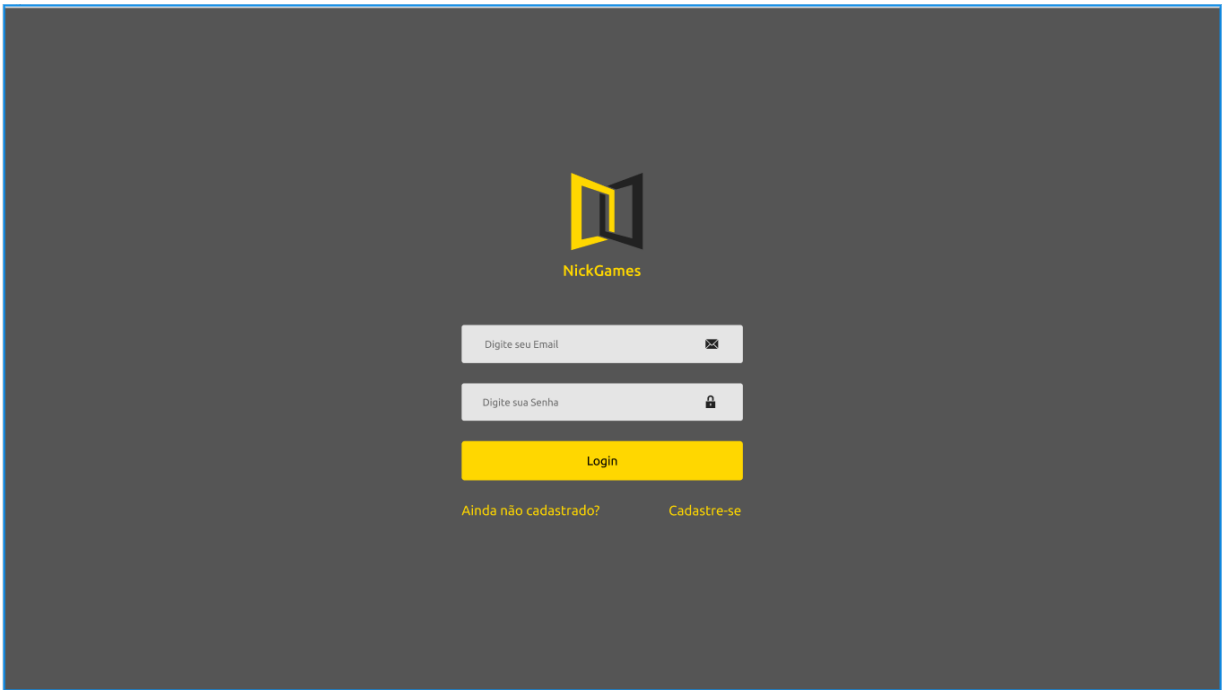


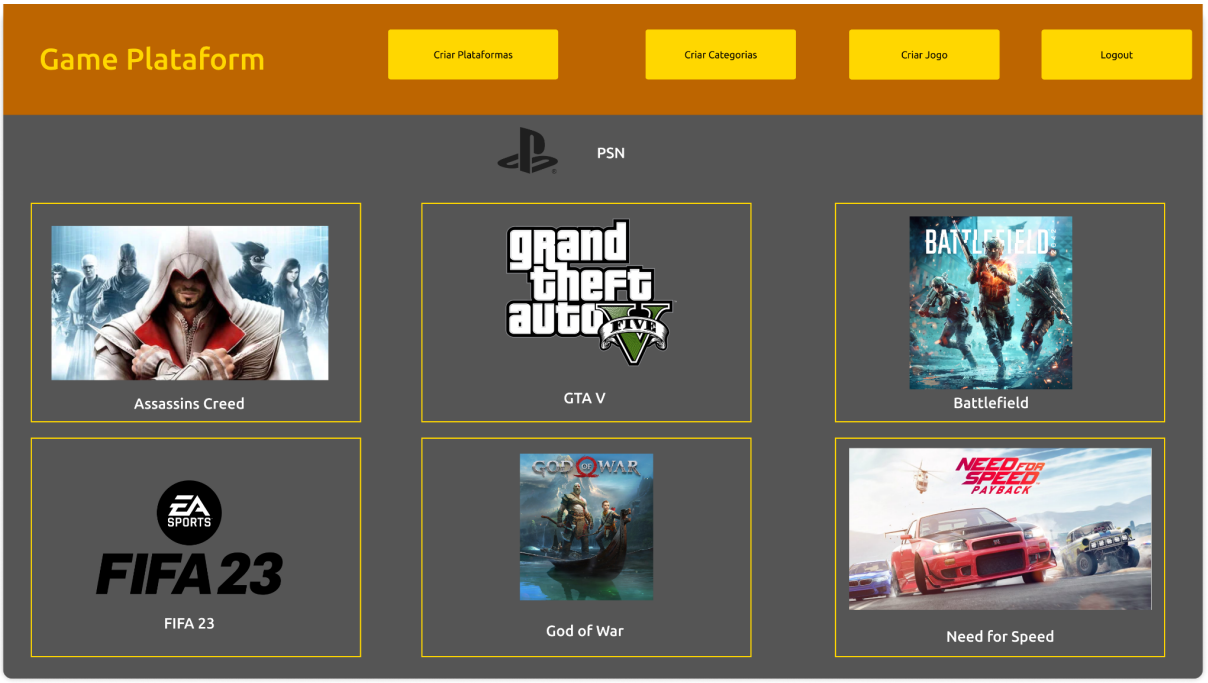
TELA DE CRIAR JOGO

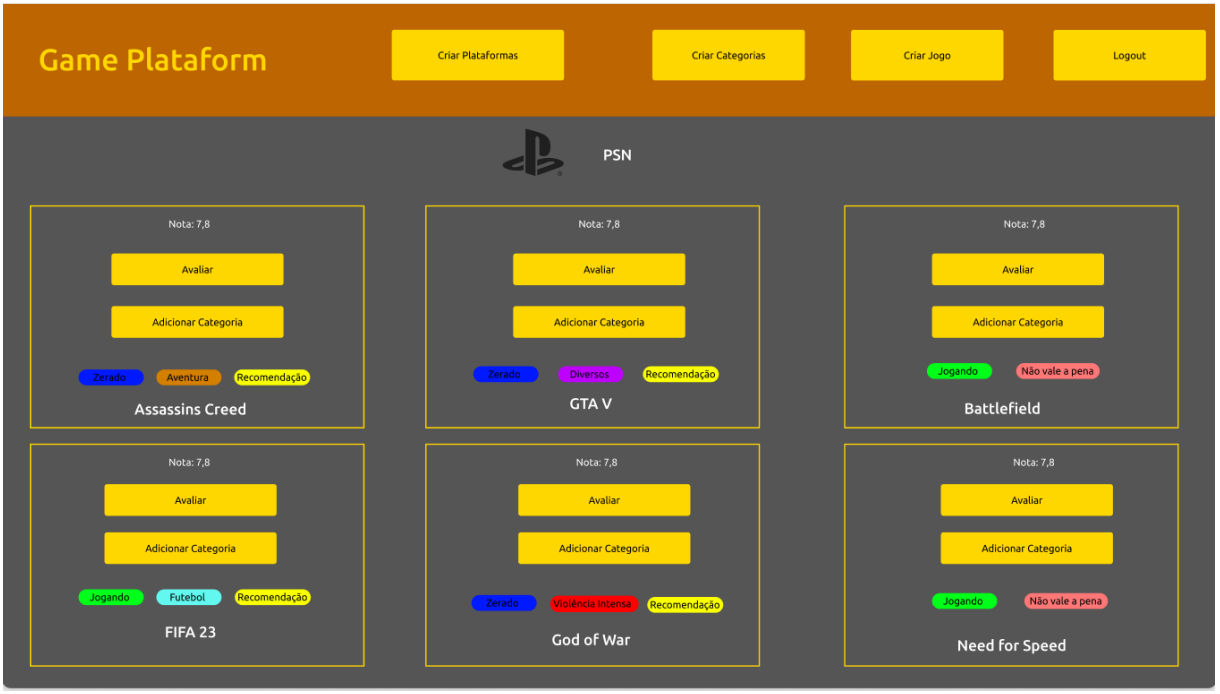


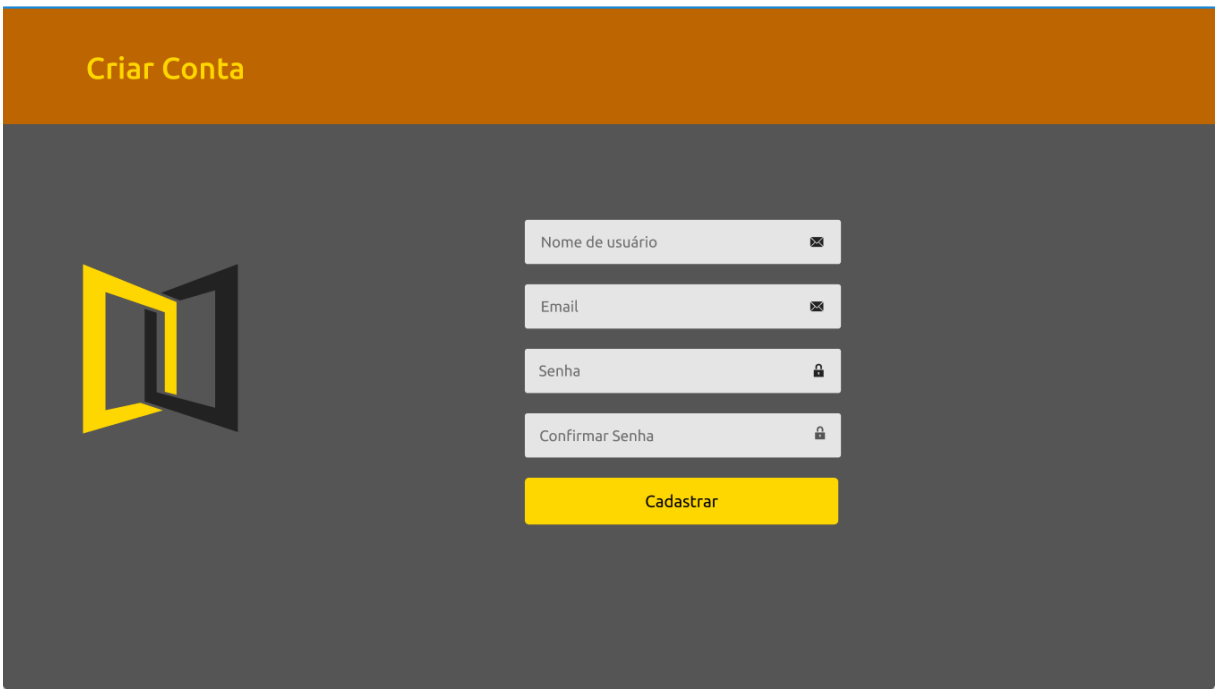
Telas do Figma:

[(Clique aqui para acessar)]






Game Plataform
Criar Plataformas
Criar Categorias
Criar Jogo
Logout
PSN
Nota: 7,8
Avaliar
Adicionar Categoria
Zerado
Aventura
Recomendação
Assassins Creed
Nota: 7,8
Avaliar
Adicionar Categoria
Zerado
Diversos
Recomendação
GTA V
Nota: 7,8
Avaliar
Adicionar Categoria
Jogando
Não vale a pena
Battlefield
Nota: 7,8
Avaliar
Adicionar Categoria
Jogando
Futebol
Recomendação
FIFA 23
Nota: 7,8
Avaliar
Adicionar Categoria
Zerado
Violência Intensa
Recomendação
God of War
Nota: 7,8
Avaliar
Adicionar Categoria
Jogando
Não vale a pena
Need for Speed


Criar Conta
Nome de usuário
Email
Senha
Confirmar Senha
Cadastrar

Criar Plataforma

Nome

Criar

Criar Categoria

Nome

Criar


Criar Jogo
Titulo
Gênero
Preço
Desenvolvedora
Data de Lançamento
Criar