Black Box Web: Implementação de um Jogo de Lógica com Bot e Modo 1v1 em Arquitetura MVC

Lucas Barão Machado*, Vitor Philip Starucka**Curso de Ciências da Computação, Universidade Federal de Santa Catarina (UFSC)

***Resumo*—Este artigo descreve o desenvolvimento de uma aplicação Web denominada *Black Box — Bot & 1v1*, que implementa a versão digital do jogo de tabuleiro *Black Box*. O sistema permite que usuários registrem perfis, joguem contra um bot ou em modo *pass-and-play* (1v1) e acompanhem um ranking baseado na menor pontuação obtida. A aplicação foi desenvolvida utilizando arquitetura Modelo–Visão–Controlador (MVC), com *frontend* em React, *backend* em Node.js e persistência de dados em banco não relacional. São apresentados a fundamentação teórica, os materiais e métodos empregados, a estrutura do projeto, bem como os resultados obtidos em termos de funcionalidade, usabilidade e discussão de vulnerabilidades e soluções de segurança.**

***Palavras-chave*—Programação para Web, Jogos Digitais, React, Node.js, Arquitetura MVC, Segurança em Aplicações Web.**

***Abstract*—This paper presents the development of a Web application called *Black Box — Bot & 1v1*, a digital implementation of the *Black Box* board game. The system allows users to register profiles, play against a bot or in a local pass-and-play mode, and track a ranking based on the lowest score achieved. The application follows a Model–View–Controller (MVC) architecture, with a React frontend, a Node.js backend, and a NoSQL database for persistence. Theoretical background, materials and methods, project structure, experimental results, and a discussion on security vulnerabilities and countermeasures are presented.**

***Index Terms*—Web Development, Digital Games, React, Node.js, MVC Architecture, Web Security.**

I. Introdução

O JOGO de tabuleiro *Black Box* é um jogo de lógica e dedução em que um jogador dispara raios nas bordas de uma grade para inferir a posição de átomos ocultos no interior do tabuleiro. Apesar de sua mecânica simples, trata-se de um jogo com elevado potencial educativo, pois estimula o raciocínio lógico, a formulação de hipóteses e a análise de resultados. A digitalização desse tipo de jogo permite ampliar o acesso, facilitando o uso em ambientes educacionais e em dispositivos pessoais conectados à Internet.

A. Motivação

No contexto da disciplina de Programação para Web, buscou-se um projeto que integrasse conceitos de *frontend*, *backend*, persistência e segurança, mantendo ao mesmo tempo um caráter lúdico e interativo. A escolha do jogo *Black Box* se mostrou adequada, pois envolve uma lógica de simulação de raios relativamente simples, mas suficiente para exigir o uso estruturado de estado, tratamento de eventos e organização do código em camadas.

Universidade Federal de Santa Catarina (UFSC). Correspondência ao autor: Lucas Barão Machado (email: vitor.starucka@grad.ufsc.br).

B. Problema

O problema abordado consiste em projetar e implementar uma aplicação Web responsiva que: (i) replique com fidelidade as regras do jogo *Black Box*; (ii) permita partidas contra um bot e partidas 1v1 no modelo *pass-and-play*; (iii) gerencie perfis de usuários com avatar; (iv) registre partidas e exiba um ranking com as melhores pontuações; (v) seja desenvolvida seguindo boas práticas de arquitetura, segurança e manutenibilidade.

C. Trabalhos relacionados

Existem implementações on-line do jogo *Black Box* em formulários simples ou como jogos de navegador, geralmente focadas apenas na lógica básica, sem autenticação nem ranking persistente. Trabalhos em desenvolvimento de jogos Web em React e Node.js, como os exemplos fornecidos pela documentação oficial de bibliotecas JavaScript e tutoriais de aplicações no modelo *single-page application* (SPA), abordam aspectos de componentização e gerenciamento de estado, mas não focam o jogo *Black Box* em si. Estudos sobre arquiteturas MVC em aplicações Web também serviram de referência para a organização em camadas adotada neste trabalho.

D. Contribuição do trabalho

A contribuição principal está na implementação de uma versão Web do *Black Box* que:

- oferece dois modos de jogo (bot e 1v1) integrados em uma mesma interface;
- registra histórico de partidas e permite o acompanhamento de ranking com base na menor pontuação;
- utiliza arquitetura MVC com clara separação entre modelo, controlador e visão;
- discute vulnerabilidades típicas de aplicações Web e as contramedidas adotadas na implementação.

E. Organização do trabalho

O restante deste artigo está organizado da seguinte forma. A Seção II apresenta a fundamentação teórica relacionada ao desenvolvimento Web, ao jogo *Black Box* e à arquitetura MVC. A Seção III descreve os materiais e métodos empregados, incluindo frameworks, tecnologias, estrutura de código e roteiro de instalação. A Seção IV discute os resultados obtidos, a estrutura do Document Object Model (DOM), o padrão MVC e as vulnerabilidades observadas. Por fim, a Seção V apresenta as conclusões e possibilidades de trabalhos futuros.

II. Fundamentação Teórica

A fundamentação teórica deste trabalho abrange três frentes principais: desenvolvimento Web, o jogo *Black Box* enquanto problema de dedução lógica e a arquitetura Modelo–Visão–Controlador.

A. Desenvolvimento Web com SPA

Aplicações Web modernas têm adotado o modelo de *single-page application* (SPA), em que a navegação ocorre predominantemente no lado do cliente, por meio de componentes reutilizáveis e gerenciamento de estado. A literatura de desenvolvimento Web apresenta abordagens para criação de interfaces declarativas com manipulação eficiente da árvore DOM. Essa abordagem facilita a composição de elementos de interface como tabuleiros, painéis de controle e formulários de login.

B. Jogo Black Box e simulação de raios

O jogo de tabuleiro *Black Box* foi popularizado como jogo de lógica em que um jogador posiciona átomos no interior de uma matriz e outro jogador dispara raios pelas bordas, observando três tipos de resposta: *hit*, *reflection* e *deflection*. A partir dessas respostas, o jogador procura deduzir a posição dos átomos. A literatura de jogos de lógica destaca o papel pedagógico desse tipo de problema na formação de habilidades de raciocínio dedutivo.

C. Arquitetura Modelo–Visão–Controlador (MVC)

A arquitetura MVC propõe a separação da aplicação em três camadas: o Modelo, responsável pela lógica de negócio e acesso a dados; a Visão, que corresponde à interface com o usuário; e o Controlador, responsável por orquestrar requisições e respostas. No projeto em estudo, o React implementa a camada de visão, enquanto o *backend* em Node.js com Express expõe uma API REST que representa a camada de controlador. Os modelos de usuário, partida e gravação de vídeo compõem a camada de modelo, sendo persistidos em banco de dados não relacional.

III. Materiais e Métodos

Esta seção descreve as tecnologias empregadas, a metodologia de desenvolvimento, os softwares utilizados e o roteiro de instalação e configuração do sistema.

A. Frameworks, tecnologias e APIs

A aplicação foi organizada em dois módulos principais: *frontend* e *backend*.

1) Frontend: O *frontend* foi desenvolvido em React com suporte da ferramenta Vite para criação e empacotamento do projeto. Foram utilizados:

- React e React Hooks para gerenciamento de estado local;
- CSS modularizado em arquivos dedicados (`layout.css` e `gameboard.css`);
- Componentes reutilizáveis para tabuleiro (`GameBoard`), perfil (`AvatarPicker`), gravação de partidas (`RecordingControls`) e listagens (`MyGames` e `Ranking`);
- Fetch API para comunicação com o *backend*.

2) Backend: O *backend* foi implementado em Node.js, utilizando o framework Express para definição de rotas e controle de requisições HTTP. Foram definidos, entre outros, os seguintes recursos REST:

- rota de autenticação (`/api/login` e `/api/register`);
- rota para registro de partidas (`/api/games`);
- rota para consulta de partidas do usuário;
- rota para consulta do ranking global.

A persistência foi realizada em banco NoSQL (por exemplo, MongoDB), com modelos para usuário, jogo e vídeo associado à partida.

3) APIs adicionais: A aplicação permite o envio de links de vídeo associados à partida, possibilitando que uma jogada gravada seja posteriormente acessada a partir da interface *Minhas partidas*. Essa funcionalidade foi pensada para integração com plataformas de hospedagem de vídeo, como YouTube, por meio do armazenamento apenas da URL.

B. Metodologia de desenvolvimento

A metodologia adotada foi incremental, com ciclos curtos de implementação, teste manual e refatoração. Inicialmente foi implementada a lógica do tabuleiro no lado do cliente, incluindo o cálculo do caminho dos raios, armazenamento de tentativas e pontuação. Em seguida, adicionou-se a camada de autenticação e o registro de partidas no *backend*. Por fim, foram incluídos o ranking, a gravação de links de vídeo e ajustes de layout, incluindo suporte a modo escuro (*dark mode*).

C. Softwares e códigos implementados

Foram utilizados os seguintes softwares:

- Node.js (versão LTS) e npm para gerenciamento de dependências;
- Vite para criação e *bundling* do *frontend*;
- Editor de código Visual Studio Code;
- Git para controle de versão;
- Navegador Google Chrome para testes.

O código-fonte foi organizado conforme a Listagem 1, em que se apresenta, de forma resumida, a estrutura de diretórios.

```
BLACKBOX-1V1-VITE/
       client/
            node_modules/
            src/
                 components/
                      AuthPanel.jsx
                      AvatarPicker.jsx
                      Chat.jsx
                      GameBoard.jsx
                      HighScore.jsx
                      MyGames.jsx
                      ProfilePanel.jsx
                      Ranking.jsx
                      RecordingControls.jsx
                 api.js
                 App.jsx
                 gameboard.css
                 index.css
                 layout.css
                 main.jsx
            index.html
            package.json
            package-lock.json
            vite.config.js
            README.md

       server/
            node_modules/
            src/
                 config/
                      db.js
                 controllers/
                      authController.js
                      gameController.js
                      userController.js
                 middleware/
                      auth.js
                 models/
                      AdminConfig.js
                      Game.js
                      User.js
                      Video.js
                 routes/
                      authRoutes.js
                      gameRoutes.js
                      userRoutes.js
                      videoRoutes.js
                 sockets/
                      index.js
                 uploads/
                 app.js
                 server.js
            .env
            package.json
            package-lock.json
            process.env.JWT_SECRET

       README.md
```

Listing 1. Estrutura de diretórios do projeto

Códigos completos encontram-se disponibilizados no repositório público:

- **Repositório principal:** https://github.com/b4ra0/INE5646-WEB

D. Roteiro de instalação e execução

O roteiro resumido de instalação e configuração é apresentado a seguir.

1) Clonagem do repositório:

```
git clone https://github.com/b4ra0/INE5646-WEB.git
cd AP
```

Listing 2. Clonagem do repositório

2) Instalação das dependências do backend:

```
cd server
npm install
# configurar variaveis de ambiente (.env) com URL do banco e secret
npm start
```

Listing 3. Instalação do backend

3) Instalação das dependências do frontend:

```
cd ../client
npm install
npm run dev
```

Listing 4. Instalação do frontend

Após a execução dos comandos anteriores, a aplicação fica disponível em http://localhost:5173. O link público de implantação (quando hospedado em servidor remoto) deve ser inserido nesta seção, por exemplo:

- **Link da aplicação Web:** https://blackbox-web.exemplo.com

IV. Resultados

A. Árvore DOM e estrutura do projeto

A *single-page application* é estruturada a partir de um componente raiz `App`, que agrega a barra superior com saudação e botão de logout, o painel de perfil, a seção de gravação de partidas, o tabuleiro e as listas de partidas e ranking. De forma simplificada, a estrutura do DOM pode ser representada como:

- `div.page-root`
  - `div.top-row`
  - `div.card.profile-card`
  - `div.card` (gravação)
  - `div.board-wrapper` (`GameBoard`)
  - `div.two-lists` (`MyGames` e `Ranking`)

O componente `GameBoard` instancia a grade 8×8, os botões de borda e a área de histórico de raios e estatísticas.

B. Padrão MVC na implementação

A arquitetura MVC pode ser visualizada da seguinte forma:

- **Modelo:** esquemas de usuário, jogo e vídeo no banco de dados; cada modelo representa uma coleção com seus respectivos atributos (por exemplo, `nickname`, `score`, `mode`, `opponent`, `createdAt`);
- **Controlador:** funções em Express que recebem requisições HTTP, validam dados, consultam ou atualizam o banco e retornam respostas em formato JSON;
- **Visão:** componentes React responsáveis pela renderização do tabuleiro, formulários de login/cadastro, listagem de partidas e ranking.

A Listagem 5 ilustra, de forma reduzida, um controlador de registro de partida.

```
async function saveGame(req, res) {
  try {
    const { mode, score, opponentNickname } = req.
        body;
    const userId = req.user.id; // obtido via
        middleware de auth

    const game = await Game.create({
      user: userId,
      mode,
      score,
      opponentNickname
    });

    return res.status(201).json({ gameId: game._id
        });
  } catch (err) {
    return res.status(500).json({ error: 'Erro ao
        salvar partida.' });
  }
}
```

Listing 5. Exemplo simplificado de controlador de partidas

C. Screens de tela e experiência de uso

A Figura 1 apresenta a interface principal do tabuleiro com o modo de jogo ativado, destacando: (i) botões para seleção de modo (*vs Bot* e *1v1*); (ii) instruções dinâmicas para o jogador de acordo com o papel de *setter* e *shooter*; (iii) tabuleiro central com bordas numeradas; e (iv) histórico de raios e estatísticas à esquerda e à direita.

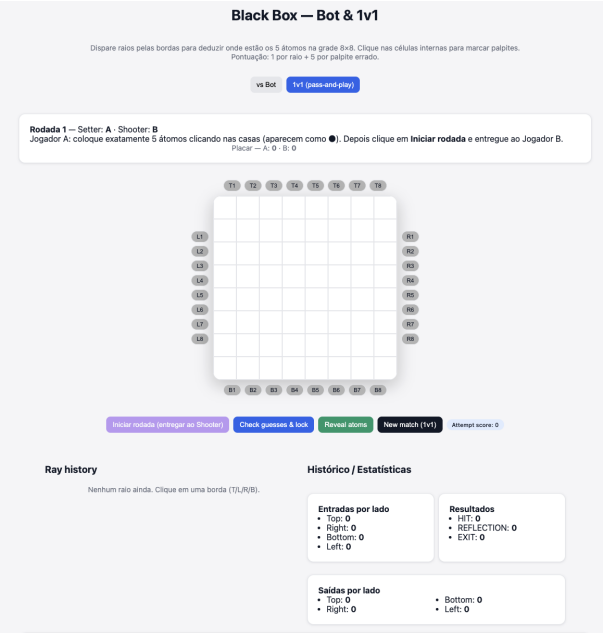

Figura 1. Tela principal de jogo do *Black Box — Bot & 1v1*.

A Figura 2 mostra a tela de ranking, em que a melhor pontuação (menor valor) aparece em destaque, bem como modo de jogo, oponente e data/hora da partida. O ranking apresenta as cinco melhores partidas registradas para o critério de menor pontuação.

Figura 2. Tela de ranking com as cinco melhores partidas.

D. Problemas encontrados

Ao longo do desenvolvimento foram identificados problemas como:

- inconsistências no cálculo da trajetória dos raios, corrigidas com a revisão sistemática da lógica de desvio e reflexão;
- dificuldades de alinhamento responsivo do tabuleiro com os botões de borda, mitigadas com o uso de *flexbox* e ajuste fino de espaçamentos em `gameboard.css`;
- erros na integração entre o *frontend* e o *backend* devido a formatos de JSON divergentes, resolvidos com a definição de um contrato de API mais rígido.

E. Vulnerabilidades e soluções de segurança

Em termos de segurança, a aplicação está sujeita às vulnerabilidades típicas de sistemas Web. Foram consideradas as seguintes medidas:

- **Validação de entrada:** todos os campos de formulário são validados tanto no *frontend* quanto no *backend*, reduzindo a chance de injeções de código;
- **Armazenamento de senhas:** as senhas são armazenadas em formato cifrado (por exemplo, com *hash* bcrypt), evitando a exposição de credenciais mesmo em caso de vazamento de base;
- **Autenticação baseada em token:** a sessão do usuário é representada por token assinado, reduzindo o risco de sequestro de sessão;
- **CORS e cabeçalhos de segurança:** o servidor foi configurado para aceitar origens confiáveis e para definir cabeçalhos de segurançã (por exemplo, *Content-Security-Policy*) quando aplicável.

Ainda que não tenha sido realizado um teste de intrusão formal, a revisão do código considerou boas práticas consolidadas em relação à segurança de aplicações Web.

V. Conclusão

Este artigo apresentou o desenvolvimento da aplicação Web *Black Box — Bot & 1v1*, que digitaliza o jogo de tabuleiro *Black Box* e integra recursos de autenticação, ranking, gravação de partidas e suporte a modos de jogo contra bot e *pass-and-play*. A aplicação foi implementada com arquitetura MVC, *frontend* em React e *backend* em Node.js, com persistência em banco NoSQL.

Os resultados obtidos mostram que a lógica de simulação de raios e cálculo de pontuação foi implementada com sucesso, a interface oferece boa experiência de uso e o ranking permite

acompanhar as melhores partidas. Em termos de segurança, foram adotadas medidas de validação de dados e armazenamento seguro de senhas, ainda que exista espaço para testes automatizados de vulnerabilidades.

Como trabalhos futuros, considera-se a implementação de partida on-line em tempo real, com comunicação via WebSocket, a criação de um módulo de análise de desempenho do jogador e a internacionalização da interface para outros idiomas.

APÊNDICE A
LISTAGEM DE ARQUIVOS PRINCIPAIS

Este apêndice apresenta um resumo dos arquivos que desempenham papel central na implementação do sistema *Black Box — Bot & 1v1*. O objetivo é ilustrar, por meio de trechos de código, como a lógica do jogo, os controladores, os modelos de dados e os componentes de interface se articulam dentro da arquitetura MVC (Modelo–Visão–Controlador). As listagens completas encontram-se no repositório oficial do projeto.

A. *Lógica do tabuleiro e simulação dos raios*

A simulação dos raios é o núcleo lógico do jogo: a cada disparo em uma borda, o sistema precisa calcular se o raio sofrerá *hit*, *reflection* ou *exit*, de acordo com a posição dos átomos. Essa lógica está concentrada no componente `GameBoard.jsx` e em funções auxiliares de cálculo.

A Listagem 6 apresenta uma versão simplificada da função de disparo de raios.

```
/**
 * Simula o disparo de um raio a partir de uma borda.
 * @param {Array<{r:number,c:number}>} atoms - posições dos átomos
 * @param {number} entryIdx - índice da borda clicada (T/L/R/B)
 * @returns {Object} tipo de resultado (hit, exit, reflection) e caminho percorrido
 */
export function shootRay(atoms, entryIdx) {
  // converte a borda selecionada em posição inicial (r,c) e direção
  let { r, c, dir } = entryFromEdgeIndex(entryIdx);

  // histórico do caminho percorrido (útil para animação e depuração)
  const visited = [];

  while (true) {
    // avança uma célula na direção atual
    r += dir.dr;
    c += dir.dc;

    // raio saiu do tabuleiro (EXIT)
    if (!isInside(r, c)) {
      return {
        type: "exit",
        exitIndex: edgeIndexFromRC(r, c),
        visited
      };
    }

    visited.push([r, c]);

    // colisão direta com átomo (HIT)
    if (atoms.some(a => a.r === r && a.c === c)) {
      return { type: "hit", visited };
    }

    // cálculo de desvio ou reflexão em função de átomos vizinhos
    const turn = computeDeflection({ r, c }, atoms, dir);

    // reflexão: o raio retorna pela mesma borda
    if (turn.reflect) {
      return { type: "reflection", visited };
    }

    // desvio: altera direção do raio, mantendo-o dentro da grade
    if (turn.newDir) {
      dir = turn.newDir;
    }
  }
}
```

Listing 6. Função central de simulação de raios no tabuleiro

B. *Controladores do backend*

No servidor, a lógica principal está concentrada em controladores Express, responsáveis por receber requisições HTTP, validar os dados, interagir com os modelos do banco de dados e retornar respostas em formato JSON.

A Listagem 7 mostra o controlador responsável por registrar uma nova partida, incluindo modo de jogo, pontuação, apelido do oponente e, opcionalmente, um link de vídeo associado.

```
/**
 * Controlador respons vel por salvar uma nova
     partida no banco.
 * Espera que o middleware de autentica  o
     preencha req.user.id.
 */
exports.saveGame = async function (req, res) {
  try {
    const { mode, score, opponentNickname, videoUrl
        } = req.body;

    const game = await Game.create({
      user: req.user.id,        // refer ncia ao
          usu rio autenticado
      mode,                     // "bot" ou "pvp"
      score,                    // pontua  o total
          da rodada
      opponentNickname,         // opcional no modo
          contra bot
      videoUrl,                 // URL de grava  o
          da partida (YouTube, etc.)
      createdAt: new Date()
    });

    return res.status(201).json({ gameId: game._id
        });
  } catch (err) {
    // em caso de erro inesperado, retorna c digo
        500 gen rico
    return res.status(500).json({ error: "Erro ao
        salvar partida." });
  }
};
```

Listing 7. Controlador de registro de partidas no backend

C. Modelo de usuário

Os modelos (camada *Model* do MVC) representam as entidades persistidas no banco de dados. O modelo de usuário é fundamental para autenticação, associação das partidas ao perfil correto e exibição de informações no ranking e na seção de perfil.

A Listagem 8 apresenta o schema simplificado do usuário.

```
const UserSchema = new mongoose.Schema({
  nickname: {
    type: String,
    required: true,
    unique: true         // impede dois usu rios com
        o mesmo apelido
  },
  passwordHash: {
    type: String,
    required: true       // senha armazenada apenas
        na forma de hash
  },
  avatarUrl: {
    type: String,
    default: null        // URL da imagem escolhida
        no AvatarPicker
  },
  createdAt: {
    type: Date,
    default: Date.now
  }
});
```

Listing 8. Modelo de usuário utilizado na aplicação

D. Componente de ranking no frontend

Na camada de visão (Visão do MVC), o componente `Ranking.jsx` é responsável por consultar a API, ordenar as partidas pela menor pontuação e exibir as cinco melhores na interface, conforme ilustrado na Figura 2.

A Listagem 9 mostra o trecho que busca os dados e renderiza a tabela.

```
/**
 * Componente respons vel por exibir o Top 5 de partidas
 * com menor pontua  o registrada.
 */
useEffect(() => {
  fetch("/api/games/highscore")
    .then(res => res.json())
    .then(data => {
      // garante que apenas as 5 melhores partidas sejam exibidas
      setTopGames(data.slice(0, 5));
    });
}, []);

return (
  <table className="ranking-table">
    <thead>
      <tr>
        <th>#</th>
        <th>Jogador</th>
        <th>Score</th>
        <th>Modo</th>
        <th>Oponente</th>
        <th>Data</th>
      </tr>
    </thead>
    <tbody>
      {topGames.map((g, i) => (
        <tr key={g._id} className={i === 0 ? "ranking-row-first" : ""}>
          <td>{i + 1}</td>
          <td>{g.userNickname}</td>
          <td>{g.score}</td>
          <td>{g.mode}</td>
          <td>{g.opponentNickname || "-"}</td>
          <td>{formatDate(g.createdAt)}</td>
        </tr>
      ))}
    </tbody>
  </table>
);
```

Listing 9. Trecho do componente de Ranking no frontend

E. Gravação de partidas e associação de vídeos

Uma funcionalidade adicional do sistema é a possibilidade de associar um link de gravação em vídeo à partida recém-finalizada, permitindo que o usuário reveja o jogo posteriormente. Essa funcionalidade é oferecida pelo componente `RecordingControls.jsx`, que envia ao backend a URL inserida pelo jogador.

A Listagem 10 mostra o trecho responsável por essa interação.

```
/**
 * Envia ao backend a URL de v deo associada partida.
 * A API registra o v nculo entre gameId e videoUrl.
 */
function saveVideo() {
  fetch("/api/video/upload", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      gameId,   // identificador da partida salva
      videoUrl  // link de grava  o (por exemplo, YouTube)
    })
  });
}
```

Listing 10. Trecho do componente de gravação de partidas