Tutorial Completo: Desenvolvendo o Backend de um Cassino Online com Node.js/Express e Docker

Introdução

Bem-vindo ao guia completo para desenvolver o backend de um cassino online! Este tutorial foi cuidadosamente elaborado para desenvolvedores que, como você, estão começando no mundo do desenvolvimento de sistemas complexos e desejam uma abordagem prática e detalhada. Nosso objetivo é construir um backend robusto, escalável e seguro, utilizando tecnologias modernas e as melhores práticas de desenvolvimento.

Neste guia, você aprenderá a configurar seu ambiente de desenvolvimento do zero, utilizando ferramentas essenciais como Docker para containerização, Node.js/Express para a lógica de negócios, PostgreSQL como banco de dados, e VSCode como sua principal ferramenta de trabalho. Abordaremos desde a instalação e configuração de cada ferramenta até a estruturação do projeto e a implementação das primeiras funcionalidades.

Pré-requisitos

Para acompanhar este tutorial, você precisará de:

- Um computador com **Windows 10** (ou superior).
- Conexão com a internet para download das ferramentas e dependências.
- Noções básicas de linha de comando (CMD ou PowerShell).
- Conhecimento básico de JavaScript e Node.js (não obrigatório, mas ajuda).

Filosofia do Tutorial

Nosso foco será em:

1. **Minimizar instalações locais**: Utilizaremos o Docker para isolar a maioria das dependências do projeto, evitando poluir seu sistema operacional com instalações desnecessárias.
2. **Maximizar o uso do VSCode**: O Visual Studio Code será sua central de comando, integrando diversas funcionalidades para otimizar seu fluxo de trabalho.
3. **Abordagem passo a passo**: Cada etapa será detalhada, com explicações claras e comandos precisos, garantindo que você possa seguir sem dificuldades.
4. **Arquitetura clara**: Entenderemos a estrutura do sistema antes de mergulhar no código.

Vamos começar!

1. Configuração do Ambiente de Desenvolvimento

O primeiro passo é preparar seu ambiente. A peça central aqui será o Docker, que nos permitirá rodar nosso banco de dados, o backend Node.js e outras ferramentas em contêineres isolados, garantindo que seu ambiente de desenvolvimento seja consistente e fácil de gerenciar.

1.1. Instalação do Docker Desktop no Windows 10

O Docker Desktop é a maneira mais fácil de começar a usar o Docker no Windows. Ele inclui o Docker Engine, Docker CLI, Docker Compose, Kubernetes e o Docker Desktop UI.

Passo 1: Verifique os requisitos do sistema

Para instalar o Docker Desktop no Windows 10, seu sistema deve atender aos seguintes requisitos:

- **WSL 2 backend**: Windows 10 64-bit: Home ou Pro versão 2004 ou superior, com Build 19041 ou superior. (Recomendado)
- **Hyper-V backend e Containers do Windows**: Windows 10 64-bit: Pro, Enterprise ou Education (Build 15063 ou superior). (Alternativo, se WSL 2 não for possível)

Recomendamos fortemente o uso do WSL 2 (Windows Subsystem for Linux 2) para uma melhor performance e integração com o Docker. Se você ainda não tem o WSL 2 configurado, siga os passos abaixo. Caso já tenha, pule para o **Passo 3**.

Passo 2: Habilitar WSL 2 e instalar uma distribuição Linux (se necessário)

Abra o PowerShell como Administrador e execute os seguintes comandos:

1. **Habilitar o Subsistema Windows para Linux**:

   dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

2. **Habilitar o recurso de Plataforma de Máquina Virtual**:

   dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart

3. **Reinicie seu computador** para que as alterações entrem em vigor.

4. **Definir o WSL 2 como sua versão padrão**:

wsl --set-default-version 2

5. **Instalar uma distribuição Linux (ex: Ubuntu)**: Abra a Microsoft Store, procure por

'Ubuntu' e instale-a. Após a instalação, abra o Ubuntu pela primeira vez para configurar seu nome de usuário e senha.

Passo 3: Baixar o instalador do Docker Desktop

Acesse o site oficial do Docker e baixe o instalador do Docker Desktop para Windows:

https://docs.docker.com/desktop/install/windows-install/

Passo 4: Instalar o Docker Desktop

Execute o instalador baixado (`Docker Desktop Installer.exe`). Siga as instruções na tela. Certifique-se de que a opção

`Use WSL 2 instead of Hyper-V (recommended)` esteja marcada durante a instalação. Após a instalação, o Docker Desktop será iniciado automaticamente.

Passo 5: Verificar a instalação do Docker

Abra o PowerShell ou o Prompt de Comando e execute:

docker --version

docker compose version

Você deverá ver as versões do Docker e do Docker Compose, indicando que a instalação foi bem-sucedida. Além disso, o ícone do Docker Desktop na bandeja do sistema (canto inferior direito da tela) deve estar verde, indicando que o Docker está em execução.

1.2. Configuração do Visual Studio Code (VSCode)

O VSCode será seu ambiente de desenvolvimento principal. Ele oferece uma integração excelente com Docker, Git e Node.js, o que o torna a escolha ideal para este projeto.

Passo 1: Instalar o VSCode

Se você ainda não tem o VSCode instalado, baixe-o do site oficial:

https://code.visualstudio.com/download

Execute o instalador e siga as instruções. Recomendamos marcar a opção "Adicionar ao PATH" durante a instalação para poder abrir o VSCode diretamente do terminal.

Passo 2: Instalar Extensões Essenciais do VSCode

Para otimizar sua experiência de desenvolvimento, instale as seguintes extensões no VSCode. Abra o VSCode, vá para a aba de Extensões (Ctrl+Shift+X) e pesquise por:

- **Docker**: Essencial para interagir com seus contêineres diretamente do VSCode. Permite visualizar, iniciar, parar e gerenciar imagens e contêineres.
- **ESLint**: Para garantir a qualidade do código JavaScript/Node.js, identificando problemas e aplicando padrões de estilo.
- **Prettier - Code formatter**: Para formatar automaticamente seu código, mantendo um estilo consistente em todo o projeto.
- **GitLens — Git supercharged**: Aprimora as capacidades do Git no VSCode, mostrando quem alterou cada linha de código, histórico de commits, etc.
- **PostgreSQL**: Para interagir com o banco de dados PostgreSQL diretamente do VSCode (opcional, mas útil).
- **REST Client**: Permite enviar requisições HTTP diretamente de arquivos `.http` ou `.rest` no VSCode, substituindo a necessidade de abrir o Postman para testes simples.

1.3. Configuração do Git

O Git é fundamental para controle de versão. Embora o VSCode tenha integração com Git, é importante ter o Git instalado globalmente.

Passo 1: Instalar o Git

Baixe o instalador do Git para Windows no site oficial:

https://git-scm.com/download/win

Execute o instalador e siga as instruções. As opções padrão geralmente são suficientes. Certifique-se de que a opção "Git from the command line and also from 3rd-party software" esteja selecionada para que o Git esteja disponível no seu PATH.

Passo 2: Configurar o Git

Após a instalação, abra o PowerShell ou o Prompt de Comando e configure seu nome de usuário e e-mail. Isso será usado para identificar seus commits:

git config --global user.name "Seu Nome"

git config --global user.email "seu.email@example.com"

1.4. Estrutura Inicial do Projeto e Docker Compose

Agora que as ferramentas básicas estão instaladas, vamos criar a estrutura inicial do seu projeto de backend e configurar o Docker Compose para orquestrar o Node.js e o PostgreSQL.

Passo 1: Criar o diretório do projeto

Crie uma pasta para o seu projeto em um local de sua preferência. Por exemplo:

mkdir C:\Projetos\casino-backend

cd C:\Projetos\casino-backend

Passo 2: Inicializar o repositório Git

Dentro da pasta `casino-backend`, inicialize um novo repositório Git:

git init

Passo 3: Criar o arquivo `docker-compose.yml`

Este arquivo definirá os serviços (contêineres) que compõem sua aplicação. Crie um arquivo chamado `docker-compose.yml` na raiz do seu projeto (`C:\Projetos\casino-backend\docker-compose.yml`) com o seguinte conteúdo:

```
version: '3.8'

services:

  db:

    image: postgres:13-alpine

    restart: always

    environment:

      POSTGRES_DB: casino_db
```

```
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
  ports:
    - "5432:5432"
  volumes:
    - db_data:/var/lib/postgresql/data
backend:
  build:
    context: .
    dockerfile: Dockerfile
  restart: always
  environment:
    NODE_ENV: development
    DATABASE_URL: postgres://user:password@db:5432/casino_db
    JWT_SECRET: supersecretjwtkey
  ports:
    - "3000:3000"
  depends_on:
    - db
  volumes:
    - .:/app
    - /app/node_modules
```

volumes:

db_data:

Explicação do `docker-compose.yml`:

- `version: '3.8'`: Define a versão da sintaxe do Docker Compose.
- `services`: Lista os serviços que serão executados.
  - `db`: Nosso serviço de banco de dados PostgreSQL.
    - `image: postgres:13-alpine`: Usa a imagem oficial do PostgreSQL na versão 13 (a versão `alpine` é menor e mais rápida).
    - `restart: always`: Garante que o contêiner do banco de dados sempre reinicie se parar.
    - `environment`: Variáveis de ambiente para configurar o PostgreSQL (nome do banco, usuário, senha).
    - `ports: - "5432:5432"`: Mapeia a porta 5432 do seu host para a porta 5432 do contêiner, permitindo que você acesse o banco de dados de fora do contêiner (ex: com DBeaver).
    - `volumes: - db_data:/var/lib/postgresql/data`: Persiste os dados do banco de dados em um volume nomeado (`db_data`), garantindo que seus dados não sejam perdidos se o contêiner for removido.
  - `backend`: Nosso serviço de backend Node.js.
    - `build: context: . dockerfile: Dockerfile`: Indica que o Docker deve construir a imagem do backend a partir do `Dockerfile` localizado no diretório atual (`.`).
    - `restart: always`: Garante que o contêiner do backend sempre reinicie.
    - `environment`: Variáveis de ambiente para o backend (ambiente de desenvolvimento, URL do banco de dados, chave secreta JWT).
    - `ports: - "3000:3000"`: Mapeia a porta 3000 do seu host para a porta 3000 do contêiner, onde o servidor Express.js estará rodando.
    - `depends_on: - db`: Garante que o serviço `db` seja iniciado antes do serviço `backend`.
    - `volumes: - .:/app - /app/node_modules`: Mapeia o diretório atual do seu projeto (`.`) para o diretório `/app` dentro do contêiner. O volume `/app/node_modules` é para evitar que os `node_modules` do host sobrescrevam os do contêiner, garantindo que as dependências sejam instaladas dentro do contêiner.
- `volumes`: Define os volumes nomeados usados pelos serviços.

Passo 4: Criar o arquivo `Dockerfile`

Na raiz do seu projeto (`C:\Projetos\casino-backend\Dockerfile`), crie o arquivo `Dockerfile` com o seguinte conteúdo:

```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD [ "npm", "start" ]
```

Explicação do `Dockerfile`:

- **`FROM node:18-alpine`**: Define a imagem base para o nosso contêiner. Usamos uma imagem Node.js na versão 18 (LTS) baseada em Alpine Linux, que é leve.
- **`WORKDIR /app`**: Define o diretório de trabalho dentro do contêiner.
- **`COPY package*.json ./`**: Copia os arquivos `package.json` e `package-lock.json` (se existir) para o diretório de trabalho. Isso é feito antes do `npm install` para aproveitar o cache do Docker. Se esses arquivos não mudarem, a camada de `npm install` não será reconstruída.
- **`RUN npm install`**: Instala as dependências do Node.js.
- **`COPY . .`**: Copia todo o restante do código-fonte do seu projeto para o diretório de trabalho dentro do contêiner.
- **`EXPOSE 3000`**: Informa ao Docker que o contêiner escutará na porta 3000. Isso é apenas uma documentação; o mapeamento real da porta é feito no `docker-compose.yml`.
- **`CMD [ "npm", "start" ]`**: Define o comando que será executado quando o contêiner for iniciado. Assumimos que você terá um script `start` no seu `package.json`.

Passo 5: Criar o arquivo `package.json`

Na raiz do seu projeto (`C:\Projetos\casino-backend\package.json`), crie o arquivo `package.json` com o seguinte conteúdo inicial:

```
{
```

```
  "name": "casino-backend",
  "version": "1.0.0",
  "description": "Backend for an online casino application",
  "main": "src/server.js",
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "Manus AI",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

Explicação do `package.json`:

- **`name`**, **`version`**, **`description`**: Metadados do seu projeto.

- **`main: "src/server.js"`**: Define o ponto de entrada principal da sua aplicação Node.js.
- **`scripts`**: Define scripts de conveniência.
    - **`start`**: Comando para iniciar a aplicação em produção.
    - **`dev`**: Comando para iniciar a aplicação em desenvolvimento com `nodemon` (que reinicia o servidor automaticamente a cada alteração de arquivo).
- **`dependencies`**: Dependências de produção (aqui, apenas `express`).
- **`devDependencies`**: Dependências de desenvolvimento (aqui, `nodemon`).

Passo 6: Criar o diretório `src` e o arquivo `server.js`

Crie a pasta `src` na raiz do seu projeto (`C:\Projetos\casino-backend\src`) e, dentro dela, crie o arquivo `server.js` com o seguinte conteúdo:

```
const express = require('express');

const app = express();

const PORT = process.env.PORT || 3000;

app.use(express.json());

app.get('/', (req, res) => {

  res.send('Bem-vindo ao Backend do Cassino Online!');

});

app.listen(PORT, () => {

  console.log(`Servidor rodando na porta ${PORT}`);

});
```

Este é um servidor Express.js básico que responde com uma mensagem de boas-vindas na rota raiz.

Passo 7: Criar o arquivo `.gitignore`

Na raiz do seu projeto (`C:\Projetos\casino-backend\.gitignore`), crie o arquivo `.gitignore` para que o Git ignore arquivos e pastas que não devem ser versionados:

```
node_modules/
```

.env

.DS_Store

build/

dist/

logs/

*.log

.env.

1.5. Iniciando os Contêineres com Docker Compose

Agora que todos os arquivos de configuração estão no lugar, você pode iniciar sua aplicação.

Passo 1: Abrir o terminal no VSCode

Abra o VSCode na pasta do seu projeto (`C:\Projetos\casino-backend`). Você pode fazer isso abrindo o VSCode e indo em `File > Open Folder...` ou, se você adicionou o VSCode ao PATH, navegando até a pasta no terminal e digitando `code .`.

Dentro do VSCode, abra o terminal integrado (Ctrl+Shift+`).

Passo 2: Construir e iniciar os serviços

No terminal do VSCode, execute o seguinte comando:

docker compose up --build

- **`docker compose up`**: Inicia os serviços definidos no `docker-compose.yml`.
- **`--build`**: Garante que as imagens dos contêineres sejam construídas (ou reconstruídas) antes de iniciar os serviços. Isso é importante na primeira vez ou quando você altera o `Dockerfile`.

Você verá a saída dos logs de ambos os serviços (`db` e `backend`). Se tudo estiver correto, você deverá ver uma mensagem como `Servidor rodando na porta 3000` nos logs do serviço `backend`.

Passo 3: Acessar a aplicação

Abra seu navegador e acesse:

http://localhost:3000

Você deverá ver a mensagem "Bem-vindo ao Backend do Cassino Online!". Isso confirma que seu backend Node.js está rodando dentro de um contêiner Docker e acessível do seu host.

Para parar os contêineres, pressione `Ctrl+C` no terminal onde o `docker compose up` está rodando. Para parar e remover os contêineres, redes e volumes (exceto os volumes nomeados como `db_data` neste caso, que persistem os dados), use:

docker compose down

1.6. Conectando ao Banco de Dados com DBeaver

O DBeaver é uma ferramenta universal de banco de dados que permite gerenciar e interagir com diversos tipos de bancos de dados, incluindo PostgreSQL. Embora o VSCode tenha extensões para PostgreSQL, o DBeaver oferece uma interface mais completa para gerenciamento de dados e esquemas.

Passo 1: Baixar e instalar o DBeaver Community Edition

Acesse o site oficial do DBeaver e baixe a versão Community Edition para Windows:

https://dbeaver.io/download/

Execute o instalador e siga as instruções.

Passo 2: Conectar ao PostgreSQL

1. Certifique-se de que seu contêiner `db` esteja rodando (`docker compose up -d` para rodar em segundo plano).
2. Abra o DBeaver.
3. Clique em `Database > New Database Connection` (ou o ícone de plugue).
4. Na lista de bancos de dados, selecione `PostgreSQL` e clique em `Next >`.
5. Preencha os detalhes da conexão:
    - **Host**: `localhost` (ou `127.0.0.1`)
    - **Port**: `5432`
    - **Database**: `casino_db`
    - **Username**: `user`
    - **Password**: `password`

6. Clique em `Test Connection...` para verificar se a conexão funciona. Se for a primeira vez, ele pode pedir para baixar o driver JDBC para PostgreSQL. Confirme.
7. Se o teste for bem-sucedido, clique em `Finish`.

Agora você pode explorar o esquema do banco de dados, executar queries SQL e gerenciar seus dados através do DBeaver.

1.7. Testando APIs com Postman (ou REST Client no VSCode)

O Postman é uma ferramenta popular para testar APIs REST. Alternativamente, você pode usar a extensão REST Client no VSCode para testes mais simples, sem sair do seu ambiente de desenvolvimento.

Opção 1: Usando Postman

Passo 1: Baixar e instalar o Postman

Acesse o site oficial do Postman e baixe o aplicativo para Windows:

https://www.postman.com/downloads/

Execute o instalador e siga as instruções.

Passo 2: Criar uma nova requisição

1. Abra o Postman.
2. Clique em `+` para criar uma nova aba de requisição.
3. Selecione o método `GET`.
4. No campo URL, digite `http://localhost:3000`.
5. Clique em `Send`.

Você deverá receber uma resposta com o status `200 OK` e o corpo da resposta "Bem-vindo ao Backend do Cassino Online!".

Opção 2: Usando REST Client no VSCode

Se você instalou a extensão REST Client, pode criar um arquivo `.http` ou `.rest` no seu projeto. Por exemplo, crie `requests.http` na raiz do projeto:

GET http://localhost:3000/

Accept: application/json

Clique no link "Send Request" que aparece acima da requisição no VSCode. A resposta será exibida em uma nova aba no VSCode.

Próximos Passos

Com seu ambiente configurado e as ferramentas essenciais prontas, você está preparado para começar a desenvolver as funcionalidades do seu backend de cassino. Na próxima seção, abordaremos a implementação da estrutura de autenticação e gerenciamento de usuários, seguindo a arquitetura definida.

2. Desenvolvimento do Código Base

Agora que seu ambiente está configurado, vamos desenvolver a estrutura base do backend do cassino. Esta seção cobrirá a criação de modelos de dados, controladores, rotas, middleware de autenticação e as primeiras funcionalidades essenciais.

2.1. Estrutura de Pastas e Organização do Projeto

Primeiro, vamos criar a estrutura completa de pastas que seguirá as melhores práticas de organização para projetos Node.js/Express.

Passo 1: Criar a estrutura de diretórios

No terminal do VSCode (dentro da pasta `casino-backend`), execute os seguintes comandos para criar a estrutura de pastas:

```
mkdir src\config src\controllers src\middleware src\models src\routes src\services src\utils src\tests

mkdir database\migrations database\seeds

mkdir docs\api docs\deployment
```

Passo 2: Atualizar o package.json com dependências necessárias

Substitua o conteúdo do arquivo `package.json` pelo seguinte:

```
{

  "name": "casino-backend",

  "version": "1.0.0",
```

```
"description": "Backend for an online casino application",
"main": "src/server.js",
"scripts": {
  "start": "node src/server.js",
  "dev": "nodemon src/server.js",
  "test": "jest",
  "test:watch": "jest --watch",
  "migrate": "node database/migrate.js",
  "seed": "node database/seed.js"
},
"keywords": ["casino", "backend", "nodejs", "express", "postgresql"],
"author": "Manus AI",
"license": "ISC",
"dependencies": {
  "express": "^4.18.2",
  "pg": "^8.11.3",
  "bcryptjs": "^2.4.3",
  "jsonwebtoken": "^9.0.2",
  "joi": "^17.9.2",
  "helmet": "^7.0.0",
  "cors": "^2.8.5",
  "express-rate-limit": "^6.10.0",
```

```
    "winston": "^3.10.0",
    "dotenv": "^16.3.1",
    "uuid": "^9.0.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.1",
    "jest": "^29.6.2",
    "supertest": "^6.3.3",
    "eslint": "^8.46.0",
    "prettier": "^3.0.1"
  }
}
```

2.2. Configuração do Banco de Dados e Conexão

Passo 1: Criar o arquivo de configuração do banco de dados

Crie o arquivo `src/config/database.js`:

```
const { Pool } = require('pg');

require('dotenv').config();

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: process.env.NODE_ENV === 'production' ? { rejectUnauthorized: false } : false,
});
```

```
// Teste de conexão
pool.on('connect', () => {
  console.log('Conectado ao banco de dados PostgreSQL');
});
pool.on('error', (err) => {
  console.error('Erro na conexão com o banco de dados:', err);
  process.exit(-1);
});
module.exports = {
  query: (text, params) => pool.query(text, params),
  pool,
};
```

Passo 2: Criar arquivo de configuração JWT

Crie o arquivo `src/config/jwt.js`:

```
require('dotenv').config();
module.exports = {
  secret: process.env.JWT_SECRET || 'supersecretjwtkey',
  expiresIn: '24h',
  refreshExpiresIn: '7d',
};
```

Passo 3: Criar arquivo de configuração geral

Crie o arquivo `src/config/index.js`:

```
require('dotenv').config();

module.exports = {
  port: process.env.PORT || 3000,
  nodeEnv: process.env.NODE_ENV || 'development',
  databaseUrl: process.env.DATABASE_URL,
  jwt: require('./jwt'),
  cors: {
    origin: process.env.CORS_ORIGIN || '*',
    credentials: true,
  },
  rateLimit: {
    windowMs: 15 * 60 * 1000, // 15 minutos
    max: 100, // máximo 100 requisições por IP por janela de tempo
  },
};
```

2.3. Criação dos Modelos de Dados

Passo 1: Modelo de Usuário

Crie o arquivo `src/models/User.js`:

```
const db = require('../config/database');
const bcrypt = require('bcryptjs');
const { v4: uuidv4 } = require('uuid');
```

```
class User {
  constructor(data) {
    this.id = data.id;
    this.email = data.email;
    this.username = data.username;
    this.password = data.password;
    this.firstName = data.first_name;
    this.lastName = data.last_name;
    this.balance = data.balance || 0;
    this.role = data.role || 'player';
    this.isActive = data.is_active !== false;
    this.isVerified = data.is_verified || false;
    this.createdAt = data.created_at;
    this.updatedAt = data.updated_at;
  }
  static async create(userData) {
    const { email, username, password, firstName, lastName } = userData;

    // Hash da senha
    const saltRounds = 12;
    const hashedPassword = await bcrypt.hash(password, saltRounds);
```

```
    const id = uuidv4();

    const query = `
      INSERT INTO users (id, email, username, password, first_name, last_name, created_at,
updated_at)
      VALUES ($1, $2, $3, $4, $5, $6, NOW(), NOW())
      RETURNING *
    `;

    const values = [id, email, username, hashedPassword, firstName, lastName];

    try {
      const result = await db.query(query, values);
      return new User(result.rows[0]);
    } catch (error) {
      throw error;
    }
  }
  static async findById(id) {
    const query = 'SELECT * FROM users WHERE id = $1 AND is_active = true';

    try {
```

```
    const result = await db.query(query, [id]);
    return result.rows.length > 0 ? new User(result.rows[0]) : null;
  } catch (error) {
    throw error;
  }
}
static async findByEmail(email) {
  const query = 'SELECT * FROM users WHERE email = $1 AND is_active = true';

  try {
    const result = await db.query(query, [email]);
    return result.rows.length > 0 ? new User(result.rows[0]) : null;
  } catch (error) {
    throw error;
  }
}
static async findByUsername(username) {
  const query = 'SELECT * FROM users WHERE username = $1 AND is_active = true';

  try {
    const result = await db.query(query, [username]);
```

```
    return result.rows.length > 0 ? new User(result.rows[0]) : null;
  } catch (error) {
    throw error;
  }
}
async validatePassword(password) {
  return await bcrypt.compare(password, this.password);
}
async updateBalance(amount, operation = 'add') {
  const newBalance = operation === 'add'
    ? parseFloat(this.balance) + parseFloat(amount)
    : parseFloat(this.balance) - parseFloat(amount);

  if (newBalance < 0) {
    throw new Error('Saldo insuficiente');
  }

  const query = `
    UPDATE users
    SET balance = $1, updated_at = NOW()
    WHERE id = $2
    RETURNING balance
```

```
    `;

    try {
      const result = await db.query(query, [newBalance, this.id]);
      this.balance = result.rows[0].balance;
      return this.balance;
    } catch (error) {
      throw error;
    }
  }
  toJSON() {
    const { password, ...userWithoutPassword } = this;
    return userWithoutPassword;
  }
}
module.exports = User;
```

Passo 2: Modelo de Transação

Crie o arquivo `src/models/Transaction.js`:

```
const db = require('../config/database');
const { v4: uuidv4 } = require('uuid');
class Transaction {
```

```
  constructor(data) {
    this.id = data.id;
    this.userId = data.user_id;
    this.type = data.type; // 'deposit', 'withdrawal', 'bet', 'win'
    this.amount = data.amount;
    this.status = data.status; // 'pending', 'completed', 'failed', 'cancelled'
    this.description = data.description;
    this.gameId = data.game_id;
    this.createdAt = data.created_at;
    this.updatedAt = data.updated_at;
  }
  static async create(transactionData) {
    const { userId, type, amount, description, gameId } = transactionData;

    const id = uuidv4();

    const query = `
      INSERT INTO transactions (id, user_id, type, amount, status, description, game_id,
created_at, updated_at)
      VALUES ($1, $2, $3, $4, 'pending', $5, $6, NOW(), NOW())
      RETURNING *
    `;
```

```
  const values = [id, userId, type, amount, description, gameId];

  try {
    const result = await db.query(query, values);
    return new Transaction(result.rows[0]);
  } catch (error) {
    throw error;
  }
}
static async findByUserId(userId, limit = 50, offset = 0) {
  const query = `
    SELECT * FROM transactions
    WHERE user_id = $1
    ORDER BY created_at DESC
    LIMIT $2 OFFSET $3
  `;

  try {
    const result = await db.query(query, [userId, limit, offset]);
    return result.rows.map(row => new Transaction(row));
  } catch (error) {
```

```
      throw error;
    }
  }

  async updateStatus(status) {
    const query = `
      UPDATE transactions
      SET status = $1, updated_at = NOW()
      WHERE id = $2
      RETURNING *
    `;

    try {
      const result = await db.query(query, [status, this.id]);
      this.status = result.rows[0].status;
      this.updatedAt = result.rows[0].updated_at;
      return this;
    } catch (error) {
      throw error;
    }
  }
}
```

```
module.exports = Transaction;
```

Passo 3: Modelo de Jogo

Crie o arquivo `src/models/Game.js`:

```
const db = require('../config/database');

const { v4: uuidv4 } = require('uuid');

class Game {

  constructor(data) {

    this.id = data.id;

    this.name = data.name;

    this.type = data.type; // 'slot', 'blackjack', 'roulette', 'poker'

    this.minBet = data.min_bet;

    this.maxBet = data.max_bet;

    this.rtp = data.rtp; // Return to Player percentage

    this.isActive = data.is_active !== false;

    this.description = data.description;

    this.rules = data.rules;

    this.createdAt = data.created_at;

    this.updatedAt = data.updated_at;

  }

  static async findAll() {

    const query = 'SELECT * FROM games WHERE is_active = true ORDER BY name';
```

```
    try {
      const result = await db.query(query);
      return result.rows.map(row => new Game(row));
    } catch (error) {
      throw error;
    }
  }

  static async findById(id) {
    const query = 'SELECT * FROM games WHERE id = $1 AND is_active = true';

    try {
      const result = await db.query(query, [id]);
      return result.rows.length > 0 ? new Game(result.rows[0]) : null;
    } catch (error) {
      throw error;
    }
  }

  static async findByType(type) {
    const query = 'SELECT * FROM games WHERE type = $1 AND is_active = true ORDER BY
name';

    try {
```

```
      const result = await db.query(query, [type]);
      return result.rows.map(row => new Game(row));
    } catch (error) {
      throw error;
    }
  }
}
module.exports = Game;
```

2.4. Criação dos Middleware

Passo 1: Middleware de Autenticação

Crie o arquivo `src/middleware/auth.js`:

```
const jwt = require('jsonwebtoken');
const config = require('../config');
const User = require('../models/User');
const authenticateToken = async (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // Bearer TOKEN
  if (!token) {
    return res.status(401).json({
      error: 'Token de acesso requerido',
      code: 'MISSING_TOKEN'
```

```
  });
}
try {
  const decoded = jwt.verify(token, config.jwt.secret);
  const user = await User.findById(decoded.userId);

  if (!user) {
    return res.status(401).json({
      error: 'Usuário não encontrado',
      code: 'USER_NOT_FOUND'
    });
  }
  req.user = user;
  next();
} catch (error) {
  if (error.name === 'TokenExpiredError') {
    return res.status(401).json({
      error: 'Token expirado',
      code: 'TOKEN_EXPIRED'
    });
  }
```

```
    return res.status(403).json({
      error: 'Token inválido',
      code: 'INVALID_TOKEN'
    });
  }
};
const requireRole = (roles) => {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        error: 'Usuário não autenticado',
        code: 'NOT_AUTHENTICATED'
      });
    }
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({
        error: 'Acesso negado. Permissões insuficientes.',
        code: 'INSUFFICIENT_PERMISSIONS'
      });
    }
    next();
```

```
  };
};
module.exports = {
  authenticateToken,
  requireRole,
};
```

Passo 2: Middleware de Validação

Crie o arquivo `src/middleware/validation.js`:

```
const Joi = require('joi');
const validate = (schema) => {
  return (req, res, next) => {
    const { error } = schema.validate(req.body);

    if (error) {
      const errorDetails = error.details.map(detail => ({
        field: detail.path.join('.'),
        message: detail.message,
      }));

      return res.status(400).json({
        error: 'Dados de entrada inválidos',
        code: 'VALIDATION_ERROR',
```

```
        details: errorDetails,
      });
    }

    next();
  };
};
// Esquemas de validação
const schemas = {
  register: Joi.object({
    email: Joi.string().email().required().messages({
      'string.email': 'Email deve ter um formato válido',
      'any.required': 'Email é obrigatório',
    }),
    username: Joi.string().alphanum().min(3).max(30).required().messages({
      'string.alphanum': 'Username deve conter apenas letras e números',
      'string.min': 'Username deve ter pelo menos 3 caracteres',
      'string.max': 'Username deve ter no máximo 30 caracteres',
      'any.required': 'Username é obrigatório',
    }),
    password: Joi.string().min(8).pattern(new
RegExp('^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#\$%\^&\*])')).required().messages({
```

```
      'string.min': 'Senha deve ter pelo menos 8 caracteres',
      'string.pattern.base': 'Senha deve conter pelo menos: 1 letra minúscula, 1 maiúscula, 1 número e 1 caractere especial',
      'any.required': 'Senha é obrigatória',
    }),
    firstName: Joi.string().min(2).max(50).required().messages({
      'string.min': 'Nome deve ter pelo menos 2 caracteres',
      'string.max': 'Nome deve ter no máximo 50 caracteres',
      'any.required': 'Nome é obrigatório',
    }),
    lastName: Joi.string().min(2).max(50).required().messages({
      'string.min': 'Sobrenome deve ter pelo menos 2 caracteres',
      'string.max': 'Sobrenome deve ter no máximo 50 caracteres',
      'any.required': 'Sobrenome é obrigatório',
    }),
  }),
  login: Joi.object({
    email: Joi.string().email().required().messages({
      'string.email': 'Email deve ter um formato válido',
      'any.required': 'Email é obrigatório',
    }),
    password: Joi.string().required().messages({
```

```
      'any.required': 'Senha é obrigatória',
    }),
  }),
  bet: Joi.object({
    gameId: Joi.string().uuid().required().messages({
      'string.uuid': 'ID do jogo deve ser um UUID válido',
      'any.required': 'ID do jogo é obrigatório',
    }),
    amount: Joi.number().positive().precision(2).required().messages({
      'number.positive': 'Valor da aposta deve ser positivo',
      'any.required': 'Valor da aposta é obrigatório',
    }),
  }),
};

module.exports = {
  validate,
  schemas,
};
```

Passo 3: Middleware de Tratamento de Erros

Crie o arquivo `src/middleware/errorHandler.js`:

```
const logger = require('../utils/logger');
```

```
const errorHandler = (err, req, res, next) => {
  logger.error('Erro capturado pelo middleware:', {
    error: err.message,
    stack: err.stack,
    url: req.url,
    method: req.method,
    ip: req.ip,
    userAgent: req.get('User-Agent'),
  });
  // Erro de validação do Joi
  if (err.isJoi) {
    return res.status(400).json({
      error: 'Dados de entrada inválidos',
      code: 'VALIDATION_ERROR',
      details: err.details,
    });
  }
  // Erro de banco de dados PostgreSQL
  if (err.code) {
    switch (err.code) {
      case '23505': // Violação de constraint única
        return res.status(409).json({
```

```
        error: 'Dados já existem no sistema',
        code: 'DUPLICATE_ENTRY',
      });
    case '23503': // Violação de foreign key
      return res.status(400).json({
        error: 'Referência inválida',
        code: 'INVALID_REFERENCE',
      });
    case '23514': // Violação de check constraint
      return res.status(400).json({
        error: 'Dados não atendem aos critérios',
        code: 'CONSTRAINT_VIOLATION',
      });
  }
}
// Erro padrão
const statusCode = err.statusCode || 500;
const message = err.message || 'Erro interno do servidor';
res.status(statusCode).json({
  error: message,
  code: err.code || 'INTERNAL_ERROR',
```

```
    ...(process.env.NODE_ENV === 'development' && { stack: err.stack }),
  });
};
const notFound = (req, res, next) => {
  const error = new Error(`Rota não encontrada - ${req.originalUrl}`);
  error.statusCode = 404;
  error.code = 'ROUTE_NOT_FOUND';
  next(error);
};
module.exports = {
  errorHandler,
  notFound,
};
```

2.5. Utilitários e Serviços

Passo 1: Logger

Crie o arquivo `src/utils/logger.js`:

```
const winston = require('winston');
const path = require('path');
const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
```

```
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  defaultMeta: { service: 'casino-backend' },
  transports: [
    new winston.transports.File({
      filename: path.join(__dirname, '../../logs/error.log'),
      level: 'error'
    }),
    new winston.transports.File({
      filename: path.join(__dirname, '../../logs/combined.log')
    }),
  ],
});
if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: winston.format.combine(
      winston.format.colorize(),
      winston.format.simple()
    )
  }));
```

```
}

module.exports = logger;
```

Passo 2: Serviço de Autenticação

Crie o arquivo `src/services/authService.js`:

```
const jwt = require('jsonwebtoken');

const config = require('../config');

const User = require('../models/User');

class AuthService {

  static generateTokens(user) {

    const payload = {

      userId: user.id,

      email: user.email,

      role: user.role,

    };

    const accessToken = jwt.sign(payload, config.jwt.secret, {

      expiresIn: config.jwt.expiresIn,

    });

    const refreshToken = jwt.sign(payload, config.jwt.secret, {

      expiresIn: config.jwt.refreshExpiresIn,

    });

    return { accessToken, refreshToken };

  }
```

```
static async register(userData) {
  const { email, username } = userData;
  // Verificar se email já existe
  const existingUserByEmail = await User.findByEmail(email);
  if (existingUserByEmail) {
    const error = new Error('Email já está em uso');
    error.code = 'EMAIL_ALREADY_EXISTS';
    error.statusCode = 409;
    throw error;
  }
  // Verificar se username já existe
  const existingUserByUsername = await User.findByUsername(username);
  if (existingUserByUsername) {
    const error = new Error('Username já está em uso');
    error.code = 'USERNAME_ALREADY_EXISTS';
    error.statusCode = 409;
    throw error;
  }
  // Criar usuário
  const user = await User.create(userData);
  const tokens = this.generateTokens(user);
```

```
    return {
      user: user.toJSON(),
      tokens,
    };
  }
  static async login(email, password) {
    const user = await User.findByEmail(email);

    if (!user) {
      const error = new Error('Credenciais inválidas');
      error.code = 'INVALID_CREDENTIALS';
      error.statusCode = 401;
      throw error;
    }
    const isPasswordValid = await user.validatePassword(password);

    if (!isPasswordValid) {
      const error = new Error('Credenciais inválidas');
      error.code = 'INVALID_CREDENTIALS';
      error.statusCode = 401;
      throw error;
    }
```

```
    const tokens = this.generateTokens(user);
    return {
      user: user.toJSON(),
      tokens,
    };
  }
}
module.exports = AuthService;
```

Passo 3: Serviço RNG (Random Number Generator)

Crie o arquivo `src/services/rngService.js`:

```
const crypto = require('crypto');
class RNGService {
  /**
   * Gera um número aleatório entre min e max (inclusive)
   */
  static randomInt(min, max) {
    const range = max - min + 1;
    const bytesNeeded = Math.ceil(Math.log2(range) / 8);
    const maxValue = Math.pow(256, bytesNeeded);
    const threshold = maxValue - (maxValue % range);
```

```
  let randomBytes;
  let randomValue;

  do {
    randomBytes = crypto.randomBytes(bytesNeeded);
    randomValue = 0;

    for (let i = 0; i < bytesNeeded; i++) {
      randomValue = (randomValue << 8) + randomBytes[i];
    }
  } while (randomValue >= threshold);

  return min + (randomValue % range);
}
/**
 * Gera um número decimal aleatório entre 0 e 1
 */
static randomFloat() {
  const randomBytes = crypto.randomBytes(4);
  const randomInt = randomBytes.readUInt32BE(0);
  return randomInt / 0xFFFFFFFF;
}
```

```
/**
 * Simula um jogo de slot machine
 */
static slotMachine() {
  const symbols = ['🍒', '🍋', '🍊', '🍇', '⭐', '💎', '7️⃣'];
  const reels = [];

  for (let i = 0; i < 3; i++) {
    const symbolIndex = this.randomInt(0, symbols.length - 1);
    reels.push(symbols[symbolIndex]);
  }

  return {
    reels,
    isWin: this.checkSlotWin(reels),
    multiplier: this.getSlotMultiplier(reels),
  };
}
/**
 * Verifica se houve vitória no slot
 */
```

```
static checkSlotWin(reels) {
  // Três símbolos iguais
  if (reels[0] === reels[1] && reels[1] === reels[2]) {
    return true;
  }

  // Dois símbolos iguais (vitória menor)
  if (reels[0] === reels[1] || reels[1] === reels[2] || reels[0] === reels[2]) {
    return true;
  }

  return false;
}
/**
 * Calcula o multiplicador baseado nos símbolos
 */
static getSlotMultiplier(reels) {
  const symbolValues = {
    '🍒': 1,
    '🍋': 1.5,
    '🍊': 2,
    '🍇': 2.5,
```

```
    '⭐': 5,
    '💎': 10,
    '7️⃣': 20,
  };
  // Três símbolos iguais
  if (reels[0] === reels[1] && reels[1] === reels[2]) {
    return symbolValues[reels[0]] * 3;
  }

  // Dois símbolos iguais
  if (reels[0] === reels[1] || reels[1] === reels[2] || reels[0] === reels[2]) {
    const symbol = reels[0] === reels[1] ? reels[0] :
                   reels[1] === reels[2] ? reels[1] : reels[0];
    return symbolValues[symbol] * 0.5;
  }

  return 0;
}
/**
 * Simula um jogo de roleta
 */
```

```
  static roulette() {

    const number = this.randomInt(0, 36);

    const isRed = [1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34,
36].includes(number);

    const isBlack = number !== 0 && !isRed;

    const isEven = number !== 0 && number % 2 === 0;

    const isOdd = number !== 0 && number % 2 === 1;



    return {

      number,

      color: number === 0 ? 'green' : isRed ? 'red' : 'black',

      isRed,

      isBlack,

      isEven,

      isOdd,

      isLow: number >= 1 && number <= 18,

      isHigh: number >= 19 && number <= 36,

    };

  }

}

module.exports = RNGService;
```

2.6. Criação dos Controladores

Passo 1: Controlador de Autenticação

Crie o arquivo `src/controllers/authController.js`:

```
const AuthService = require('../services/authService');

const logger = require('../utils/logger');

class AuthController {

  static async register(req, res, next) {

    try {

      const result = await AuthService.register(req.body);



      logger.info('Novo usuário registrado', {

        userId: result.user.id,

        email: result.user.email,

        username: result.user.username,

      });

      res.status(201).json({

        success: true,

        message: 'Usuário registrado com sucesso',

        data: result,

      });

    } catch (error) {
```

```
    next(error);
  }
}
static async login(req, res, next) {
  try {
    const { email, password } = req.body;
    const result = await AuthService.login(email, password);

    logger.info('Usuário logado', {
      userId: result.user.id,
      email: result.user.email,
    });
    res.json({
      success: true,
      message: 'Login realizado com sucesso',
      data: result,
    });
  } catch (error) {
    next(error);
  }
}
static async profile(req, res, next) {
```

```
  try {
    res.json({
      success: true,
      data: {
        user: req.user.toJSON(),
      },
    });
  } catch (error) {
    next(error);
  }
}
static async logout(req, res, next) {
  try {
    // Em uma implementação mais robusta, você invalidaria o token aqui
    // Por exemplo, adicionando-o a uma blacklist no Redis

    logger.info('Usuário deslogado', {
      userId: req.user.id,
      email: req.user.email,
    });
    res.json({
```

```
        success: true,
        message: 'Logout realizado com sucesso',
      });
    } catch (error) {
      next(error);
    }
  }
}
module.exports = AuthController;
```

Passo 2: Controlador de Jogos

Crie o arquivo `src/controllers/gameController.js`:

```
const Game = require('../models/Game');
const Transaction = require('../models/Transaction');
const RNGService = require('../services/rngService');
const logger = require('../utils/logger');
class GameController {
  static async getGames(req, res, next) {
    try {
      const { type } = req.query;

      let games;
      if (type) {
```

```
      games = await Game.findByType(type);
    } else {
      games = await Game.findAll();
    }
    res.json({
      success: true,
      data: {
        games,
        count: games.length,
      },
    });
  } catch (error) {
    next(error);
  }
}
static async getGame(req, res, next) {
  try {
    const { id } = req.params;
    const game = await Game.findById(id);
    if (!game) {
      return res.status(404).json({
```

```
        success: false,
        error: 'Jogo não encontrado',
        code: 'GAME_NOT_FOUND',
      });
    }
    res.json({
      success: true,
      data: { game },
    });
  } catch (error) {
    next(error);
  }
}
static async playSlot(req, res, next) {
  try {
    const { gameId, amount } = req.body;
    const user = req.user;
    // Verificar se o jogo existe
    const game = await Game.findById(gameId);
    if (!game || game.type !== 'slot') {
      return res.status(404).json({
        success: false,
```

```
    error: 'Jogo de slot não encontrado',
    code: 'SLOT_GAME_NOT_FOUND',
  });
}
// Verificar limites de aposta
if (amount < game.minBet || amount > game.maxBet) {
  return res.status(400).json({
    success: false,
    error: `Aposta deve estar entre ${game.minBet} e ${game.maxBet}`,
    code: 'INVALID_BET_AMOUNT',
  });
}
// Verificar saldo
if (user.balance < amount) {
  return res.status(400).json({
    success: false,
    error: 'Saldo insuficiente',
    code: 'INSUFFICIENT_BALANCE',
  });
}
// Criar transação de aposta
```

```
const betTransaction = await Transaction.create({
  userId: user.id,
  type: 'bet',
  amount: amount,
  description: `Aposta no jogo ${game.name}`,
  gameId: gameId,
});
// Debitar valor da aposta
await user.updateBalance(amount, 'subtract');
// Jogar slot
const result = RNGService.slotMachine();
const winAmount = result.isWin ? amount * result.multiplier : 0;
// Se ganhou, criar transação de vitória e creditar
if (winAmount > 0) {
  const winTransaction = await Transaction.create({
    userId: user.id,
    type: 'win',
    amount: winAmount,
    description: `Vitória no jogo ${game.name}`,
    gameId: gameId,
  });
  await user.updateBalance(winAmount, 'add');
```

```
  await winTransaction.updateStatus('completed');
}
// Completar transação de aposta
await betTransaction.updateStatus('completed');
logger.info('Jogo de slot jogado', {
  userId: user.id,
  gameId: gameId,
  betAmount: amount,
  winAmount: winAmount,
  result: result.reels,
});
res.json({
  success: true,
  data: {
    result: result.reels,
    isWin: result.isWin,
    winAmount: winAmount,
    multiplier: result.multiplier,
    newBalance: user.balance,
    betTransaction: betTransaction.id,
  },
```

```
    });
  } catch (error) {
    next(error);
  }
}
static async playRoulette(req, res, next) {
  try {
    const { gameId, amount, bet } = req.body;
    const user = req.user;
    // Verificar se o jogo existe
    const game = await Game.findById(gameId);
    if (!game || game.type !== 'roulette') {
      return res.status(404).json({
        success: false,
        error: 'Jogo de roleta não encontrado',
        code: 'ROULETTE_GAME_NOT_FOUND',
      });
    }
    // Verificar limites de aposta
    if (amount < game.minBet || amount > game.maxBet) {
      return res.status(400).json({
        success: false,
```

```
    error: `Aposta deve estar entre ${game.minBet} e ${game.maxBet}`,
    code: 'INVALID_BET_AMOUNT',
  });
}
// Verificar saldo
if (user.balance < amount) {
  return res.status(400).json({
    success: false,
    error: 'Saldo insuficiente',
    code: 'INSUFFICIENT_BALANCE',
  });
}
// Validar tipo de aposta
const validBets = ['red', 'black', 'even', 'odd', 'low', 'high'];
const numberBets = Array.from({length: 37}, (_, i) => i.toString());

if (!validBets.includes(bet) && !numberBets.includes(bet)) {
  return res.status(400).json({
    success: false,
    error: 'Tipo de aposta inválido',
    code: 'INVALID_BET_TYPE',
```

```
  });
}
// Criar transação de aposta
const betTransaction = await Transaction.create({
  userId: user.id,
  type: 'bet',
  amount: amount,
  description: `Aposta na roleta: ${bet}`,
  gameId: gameId,
});
// Debitar valor da aposta
await user.updateBalance(amount, 'subtract');
// Jogar roleta
const result = RNGService.roulette();
let isWin = false;
let multiplier = 0;
// Verificar se ganhou
if (bet === 'red' && result.isRed) {
  isWin = true;
  multiplier = 2;
} else if (bet === 'black' && result.isBlack) {
  isWin = true;
```

```
  multiplier = 2;
} else if (bet === 'even' && result.isEven) {
  isWin = true;
  multiplier = 2;
} else if (bet === 'odd' && result.isOdd) {
  isWin = true;
  multiplier = 2;
} else if (bet === 'low' && result.isLow) {
  isWin = true;
  multiplier = 2;
} else if (bet === 'high' && result.isHigh) {
  isWin = true;
  multiplier = 2;
} else if (bet === result.number.toString()) {
  isWin = true;
  multiplier = 36; // Aposta em número específico
}
const winAmount = isWin ? amount * multiplier : 0;
// Se ganhou, criar transação de vitória e creditar
if (winAmount > 0) {
  const winTransaction = await Transaction.create({
```

```
    userId: user.id,
    type: 'win',
    amount: winAmount,
    description: `Vitória na roleta: ${bet}`,
    gameId: gameId,
  });
  await user.updateBalance(winAmount, 'add');
  await winTransaction.updateStatus('completed');
}
// Completar transação de aposta
await betTransaction.updateStatus('completed');
logger.info('Jogo de roleta jogado', {
  userId: user.id,
  gameId: gameId,
  betAmount: amount,
  betType: bet,
  result: result.number,
  winAmount: winAmount,
});
res.json({
  success: true,
  data: {
```

```
          result: result,
          bet: bet,
          isWin: isWin,
          winAmount: winAmount,
          multiplier: multiplier,
          newBalance: user.balance,
          betTransaction: betTransaction.id,
        },
      });
    } catch (error) {
      next(error);
    }
  }
}
module.exports = GameController;
```

Passo 3: Controlador de Carteira

Crie o arquivo `src/controllers/walletController.js`:

```
const Transaction = require('../models/Transaction');
const logger = require('../utils/logger');
class WalletController {
  static async getBalance(req, res, next) {
```

```
  try {
    const user = req.user;
    res.json({
      success: true,
      data: {
        balance: user.balance,
        userId: user.id,
      },
    });
  } catch (error) {
    next(error);
  }
}
static async getTransactions(req, res, next) {
  try {
    const user = req.user;
    const { page = 1, limit = 20 } = req.query;

    const offset = (page - 1) * limit;
    const transactions = await Transaction.findByUserId(user.id, parseInt(limit), offset);
    res.json({
      success: true,
```

```
      data: {
        transactions,
        pagination: {
          page: parseInt(page),
          limit: parseInt(limit),
          total: transactions.length,
        },
      },
    });
  } catch (error) {
    next(error);
  }
}
static async deposit(req, res, next) {
  try {
    const { amount } = req.body;
    const user = req.user;
    if (amount <= 0) {
      return res.status(400).json({
        success: false,
        error: 'Valor do depósito deve ser positivo',
```

```
    code: 'INVALID_DEPOSIT_AMOUNT',
  });
}
// Criar transação de depósito
const transaction = await Transaction.create({
  userId: user.id,
  type: 'deposit',
  amount: amount,
  description: 'Depósito na carteira',
});
// Creditar valor na carteira
await user.updateBalance(amount, 'add');
await transaction.updateStatus('completed');
logger.info('Depósito realizado', {
  userId: user.id,
  amount: amount,
  transactionId: transaction.id,
});
res.json({
  success: true,
  message: 'Depósito realizado com sucesso',
  data: {
```

```
        transaction: transaction,
        newBalance: user.balance,
      },
    });
  } catch (error) {
    next(error);
  }
}
static async withdraw(req, res, next) {
  try {
    const { amount } = req.body;
    const user = req.user;
    if (amount <= 0) {
      return res.status(400).json({
        success: false,
        error: 'Valor do saque deve ser positivo',
        code: 'INVALID_WITHDRAWAL_AMOUNT',
      });
    }
    if (user.balance < amount) {
      return res.status(400).json({
```

```
    success: false,
    error: 'Saldo insuficiente',
    code: 'INSUFFICIENT_BALANCE',
  });
}
// Criar transação de saque
const transaction = await Transaction.create({
  userId: user.id,
  type: 'withdrawal',
  amount: amount,
  description: 'Saque da carteira',
});
// Debitar valor da carteira
await user.updateBalance(amount, 'subtract');
await transaction.updateStatus('completed');
logger.info('Saque realizado', {
  userId: user.id,
  amount: amount,
  transactionId: transaction.id,
});
res.json({
  success: true,
```

```
        message: 'Saque realizado com sucesso',
        data: {
          transaction: transaction,
          newBalance: user.balance,
        },
      });
    } catch (error) {
      next(error);
    }
  }
}

module.exports = WalletController;
```

2.7. Criação das Rotas

Passo 1: Rotas de Autenticação

Crie o arquivo `src/routes/auth.js`:

```
const express = require('express');
const router = express.Router();
const AuthController = require('../controllers/authController');
const { validate, schemas } = require('../middleware/validation');
const { authenticateToken } = require('../middleware/auth');

// Registro de usuário
```

```
router.post('/register', validate(schemas.register), AuthController.register);

// Login

router.post('/login', validate(schemas.login), AuthController.login);

// Perfil do usuário (requer autenticação)

router.get('/profile', authenticateToken, AuthController.profile);

// Logout (requer autenticação)

router.post('/logout', authenticateToken, AuthController.logout);

module.exports = router;
```

Passo 2: Rotas de Jogos

Crie o arquivo `src/routes/games.js`:

```
const express = require('express');

const router = express.Router();

const GameController = require('../controllers/gameController');

const { authenticateToken } = require('../middleware/auth');

const { validate, schemas } = require('../middleware/validation');

// Listar jogos disponíveis

router.get('/', GameController.getGames);

// Obter detalhes de um jogo específico

router.get('/:id', GameController.getGame);

// Jogar slot (requer autenticação)

router.post('/slot/play', authenticateToken, validate(schemas.bet), GameController.playSlot);

// Jogar roleta (requer autenticação)
```

```
router.post('/roulette/play', authenticateToken, GameController.playRoulette);

module.exports = router;
```

Passo 3: Rotas de Carteira

Crie o arquivo `src/routes/wallet.js`:

```
const express = require('express');

const router = express.Router();

const WalletController = require('../controllers/walletController');

const { authenticateToken } = require('../middleware/auth');

const Joi = require('joi');

const { validate } = require('../middleware/validation');

// Schema para transações financeiras

const transactionSchema = Joi.object({

  amount: Joi.number().positive().precision(2).required().messages({

    'number.positive': 'Valor deve ser positivo',

    'any.required': 'Valor é obrigatório',

  }),

});

// Obter saldo (requer autenticação)

router.get('/balance', authenticateToken, WalletController.getBalance);

// Obter histórico de transações (requer autenticação)

router.get('/transactions', authenticateToken, WalletController.getTransactions);
```

```
// Fazer depósito (requer autenticação)

router.post('/deposit', authenticateToken, validate(transactionSchema), WalletController.deposit);

// Fazer saque (requer autenticação)

router.post('/withdraw', authenticateToken, validate(transactionSchema),
WalletController.withdraw);

module.exports = router;
```

Passo 4: Arquivo principal de rotas

Crie o arquivo `src/routes/index.js`:

```
const express = require('express');

const router = express.Router();

// Importar rotas específicas

const authRoutes = require('./auth');

const gameRoutes = require('./games');

const walletRoutes = require('./wallet');

// Rota de health check

router.get('/health', (req, res) => {

  res.json({

    success: true,

    message: 'API do Cassino Online funcionando',

    timestamp: new Date().toISOString(),

    version: '1.0.0',

  });
```

```
});

// Registrar rotas

router.use('/auth', authRoutes);

router.use('/games', gameRoutes);

router.use('/wallet', walletRoutes);

module.exports = router;
```

2.8. Arquivo Principal da Aplicação

Passo 1: Atualizar o arquivo `src/app.js`

Crie o arquivo `src/app.js`:

```
const express = require('express');

const helmet = require('helmet');

const cors = require('cors');

const rateLimit = require('express-rate-limit');

const config = require('./config');

const routes = require('./routes');

const { errorHandler, notFound } = require('./middleware/errorHandler');

const logger = require('./utils/logger');

const app = express();

// Middleware de segurança

app.use(helmet());

// CORS

app.use(cors(config.cors));
```

```
// Rate limiting
const limiter = rateLimit(config.rateLimit);
app.use('/api/', limiter);
// Parsing de JSON
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true, limit: '10mb' }));
// Logging de requisições
app.use((req, res, next) => {
  logger.info('Requisição recebida', {
    method: req.method,
    url: req.url,
    ip: req.ip,
    userAgent: req.get('User-Agent'),
  });
  next();
});
// Rotas principais
app.use('/api', routes);
// Rota raiz
app.get('/', (req, res) => {
  res.json({
```

```
    success: true,
    message: 'Bem-vindo ao Backend do Cassino Online!',
    version: '1.0.0',
    documentation: '/api/health',
  });
});
// Middleware para rotas não encontradas
app.use(notFound);
// Middleware de tratamento de erros
app.use(errorHandler);
module.exports = app;
```

Passo 2: Atualizar o arquivo `src/server.js`

Substitua o conteúdo do arquivo `src/server.js`:

```
const app = require('./app');
const config = require('./config');
const logger = require('./utils/logger');
const fs = require('fs');
const path = require('path');
// Criar diretório de logs se não existir
const logsDir = path.join(__dirname, '../logs');
if (!fs.existsSync(logsDir)) {
  fs.mkdirSync(logsDir, { recursive: true });
```

```
}

// Iniciar servidor
const server = app.listen(config.port, '0.0.0.0', () => {
  logger.info(`Servidor rodando na porta ${config.port}`, {
    port: config.port,
    environment: config.nodeEnv,
  });
});

// Graceful shutdown
process.on('SIGTERM', () => {
  logger.info('SIGTERM recebido, encerrando servidor graciosamente');
  server.close(() => {
    logger.info('Servidor encerrado');
    process.exit(0);
  });
});

process.on('SIGINT', () => {
  logger.info('SIGINT recebido, encerrando servidor graciosamente');
  server.close(() => {
    logger.info('Servidor encerrado');
    process.exit(0);
```

```
  });
});

module.exports = server;
```

2.9. Scripts de Banco de Dados

Passo 1: Script de Migração

Crie o arquivo `database/migrations/001_initial_schema.sql`:

```
-- Criação da tabela de usuários
CREATE TABLE IF NOT EXISTS users (
    id UUID PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    balance DECIMAL(10,2) DEFAULT 0.00,
    role VARCHAR(20) DEFAULT 'player',
    is_active BOOLEAN DEFAULT true,
    is_verified BOOLEAN DEFAULT false,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Criação da tabela de jogos
```

```
CREATE TABLE IF NOT EXISTS games (
    id UUID PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    type VARCHAR(50) NOT NULL,
    min_bet DECIMAL(10,2) DEFAULT 1.00,
    max_bet DECIMAL(10,2) DEFAULT 1000.00,
    rtp DECIMAL(5,2) DEFAULT 95.00,
    is_active BOOLEAN DEFAULT true,
    description TEXT,
    rules JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
-- Criação da tabela de transações
CREATE TABLE IF NOT EXISTS transactions (
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL REFERENCES users(id),
    type VARCHAR(20) NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    status VARCHAR(20) DEFAULT 'pending',
    description TEXT,
```

```
    game_id UUID REFERENCES games(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
-- Criação da tabela de sessões de jogo
CREATE TABLE IF NOT EXISTS game_sessions (
    id UUID PRIMARY KEY,
    user_id UUID NOT NULL REFERENCES users(id),
    game_id UUID NOT NULL REFERENCES games(id),
    start_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    end_time TIMESTAMP,
    total_bet DECIMAL(10,2) DEFAULT 0.00,
    total_win DECIMAL(10,2) DEFAULT 0.00,
    is_active BOOLEAN DEFAULT true
);
-- Índices para performance
CREATE INDEX IF NOT EXISTS idx_users_email ON users(email);
CREATE INDEX IF NOT EXISTS idx_users_username ON users(username);
CREATE INDEX IF NOT EXISTS idx_transactions_user_id ON transactions(user_id);
CREATE INDEX IF NOT EXISTS idx_transactions_created_at ON transactions(created_at);
CREATE INDEX IF NOT EXISTS idx_game_sessions_user_id ON game_sessions(user_id);
CREATE INDEX IF NOT EXISTS idx_games_type ON games(type);
```

```
-- Trigger para atualizar updated_at automaticamente

CREATE OR REPLACE FUNCTION update_updated_at_column()

RETURNS TRIGGER AS $$

BEGIN

    NEW.updated_at = CURRENT_TIMESTAMP;

    RETURN NEW;

END;

$$ language 'plpgsql';

CREATE TRIGGER update_users_updated_at BEFORE UPDATE ON users

    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_games_updated_at BEFORE UPDATE ON games

    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_transactions_updated_at BEFORE UPDATE ON transactions

    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();
```

Passo 2: Script de Seeds

Crie o arquivo `database/seeds/001_initial_data.sql`:

```
-- Inserir jogos iniciais

INSERT INTO games (id, name, type, min_bet, max_bet, rtp, description, rules) VALUES

(

    'a1b2c3d4-e5f6-7890-abcd-ef1234567890',

    'Slot Clássico',
```

```
    'slot',
    1.00,
    100.00,
    95.50,
    'Um slot machine clássico com 3 rolos e símbolos tradicionais',
    '{"paylines": 1, "reels": 3, "symbols": ["🍒", "🍋", "🍊", "🍇", "⭐", "💎", "7️⃣"]}'
),
(
    'b2c3d4e5-f6g7-8901-bcde-f23456789012',
    'Roleta Europeia',
    'roulette',
    5.00,
    500.00,
    97.30,
    'Roleta europeia com um zero, oferecendo melhores odds para o jogador',
    '{"type": "european", "numbers": 37, "zero_count": 1}'
),
(
    'c3d4e5f6-g7h8-9012-cdef-345678901234',
    'Blackjack Clássico',
    'blackjack',
    10.00,
```

```
    1000.00,
    99.50,
    'Blackjack tradicional com regras padrão',
    '{"decks": 6, "dealer_stands_on": "soft_17", "blackjack_pays": "3:2"}'
);
-- Inserir usuário administrador padrão (senha: Admin123!)
INSERT INTO users (id, email, username, password, first_name, last_name, balance, role,
is_verified) VALUES
(
    'd4e5f6g7-h8i9-0123-defg-456789012345',
    'admin@casino.com',
    'admin',
    '$2a$12$LQv3c1yqBw2jo6H1PAmi/.jigocJ9cxHljs8J/RCBbkUbdwQr19hS',
    'Administrador',
    'Sistema',
    10000.00,
    'admin',
    true
);
```

Passo 3: Script de Migração Node.js

Crie o arquivo `database/migrate.js`:

```
const fs = require('fs');
```

```
const path = require('path');

const db = require('../src/config/database');

async function runMigrations() {

  try {

    console.log('Iniciando migrações...');



    const migrationsDir = path.join(__dirname, 'migrations');

    const migrationFiles = fs.readdirSync(migrationsDir).sort();



    for (const file of migrationFiles) {

      if (file.endsWith('.sql')) {

        console.log(`Executando migração: ${file}`);

        const sql = fs.readFileSync(path.join(migrationsDir, file), 'utf8');

        await db.query(sql);

        console.log(`Migração ${file} executada com sucesso`);

      }

    }



    console.log('Todas as migrações foram executadas com sucesso!');

  } catch (error) {

    console.error('Erro ao executar migrações:', error);
```

```
    process.exit(1);
  } finally {
    process.exit(0);
  }
}
runMigrations();
```

Passo 4: Script de Seeds Node.js

Crie o arquivo `database/seed.js`:

```
const fs = require('fs');
const path = require('path');
const db = require('../src/config/database');
async function runSeeds() {
  try {
    console.log('Iniciando seeds...');

    const seedsDir = path.join(__dirname, 'seeds');
    const seedFiles = fs.readdirSync(seedsDir).sort();

    for (const file of seedFiles) {
      if (file.endsWith('.sql')) {
        console.log(`Executando seed: ${file}`);
        const sql = fs.readFileSync(path.join(seedsDir, file), 'utf8');
```

```
      await db.query(sql);
      console.log(`Seed ${file} executado com sucesso`);
    }
  }

  console.log('Todos os seeds foram executados com sucesso!');
 } catch (error) {
  console.error('Erro ao executar seeds:', error);
  process.exit(1);
 } finally {
  process.exit(0);
 }
}
runSeeds();
```

2.10. Arquivo de Ambiente

Passo 1: Criar arquivo `.env.example`

Crie o arquivo `.env.example` na raiz do projeto:

```
# Configurações do servidor
NODE_ENV=development
PORT=3000
# Configurações do banco de dados
```

DATABASE_URL=postgres://user:password@db:5432/casino_db

Configurações JWT

JWT_SECRET=supersecretjwtkey

Configurações CORS

CORS_ORIGIN=*

Configurações de log

LOG_LEVEL=info

Com essa estrutura completa, você terá um backend funcional para seu cassino online. Na próxima seção, abordaremos como testar todas essas funcionalidades e criar a documentação das APIs.

3. Testando e Documentando as APIs

Agora que temos toda a estrutura do backend implementada, é fundamental testar todas as funcionalidades e criar uma documentação clara das APIs. Esta seção cobrirá como executar o projeto, testar cada endpoint e criar uma documentação completa.

3.1. Executando o Projeto Completo

Passo 1: Instalar dependências

Primeiro, certifique-se de que todas as dependências estão instaladas. No terminal do VSCode, execute:

npm install

Passo 2: Executar migrações do banco de dados

Antes de iniciar o servidor, precisamos criar as tabelas no banco de dados:

docker compose up -d db

npm run migrate

npm run seed

Passo 3: Iniciar a aplicação

Agora você pode iniciar toda a aplicação:

docker compose up --build

Ou, se preferir rodar apenas o banco em Docker e o Node.js localmente:

docker compose up -d db

npm run dev

Passo 4: Verificar se tudo está funcionando

Acesse http://localhost:3000 no seu navegador. Você deve ver a mensagem de boas-vindas.

Acesse http://localhost:3000/api/health para verificar o health check da API.

3.2. Documentação Completa das APIs

3.2.1. Endpoints de Autenticação

Base URL: `http://localhost:3000/api/auth`

POST /api/auth/register

Registra um novo usuário no sistema.

Corpo da Requisição:

```
{
  "email": "usuario@exemplo.com",
  "username": "usuario123",
  "password": "MinhaSenh@123",
  "firstName": "João",
  "lastName": "Silva"
}
```

Resposta de Sucesso (201):


```
{
  "success": true,
  "message": "Usuário registrado com sucesso",
  "data": {
    "user": {
      "id": "uuid-do-usuario",
      "email": "usuario@exemplo.com",
      "username": "usuario123",
      "firstName": "João",
      "lastName": "Silva",
      "balance": 0,
      "role": "player",
      "isActive": true,
      "isVerified": false,
      "createdAt": "2024-01-01T12:00:00.000Z",
      "updatedAt": "2024-01-01T12:00:00.000Z"
    },
    "tokens": {
      "accessToken": "jwt-access-token",
      "refreshToken": "jwt-refresh-token"
    }
```

```
  }
}
```

Possíveis Erros:

- `409 Conflict`: Email ou username já existem
- `400 Bad Request`: Dados de entrada inválidos

POST /api/auth/login

Autentica um usuário existente.

Corpo da Requisição:

```
{
  "email": "usuario@exemplo.com",
  "password": "MinhaSenh@123"
}
```

Resposta de Sucesso (200):

```
{
  "success": true,
  "message": "Login realizado com sucesso",
  "data": {
    "user": {
      "id": "uuid-do-usuario",
      "email": "usuario@exemplo.com",
      "username": "usuario123",
      "firstName": "João",
```

```
    "lastName": "Silva",
    "balance": 100.50,
    "role": "player",
    "isActive": true,
    "isVerified": false,
    "createdAt": "2024-01-01T12:00:00.000Z",
    "updatedAt": "2024-01-01T12:00:00.000Z"
  },
  "tokens": {
    "accessToken": "jwt-access-token",
    "refreshToken": "jwt-refresh-token"
  }
 }
}
```


Possíveis Erros:

- `401 Unauthorized`: Credenciais inválidas

GET /api/auth/profile

Obtém o perfil do usuário autenticado.

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Resposta de Sucesso (200):


```
{
```

```
  "success": true,
  "data": {
    "user": {
      "id": "uuid-do-usuario",
      "email": "usuario@exemplo.com",
      "username": "usuario123",
      "firstName": "João",
      "lastName": "Silva",
      "balance": 100.50,
      "role": "player",
      "isActive": true,
      "isVerified": false,
      "createdAt": "2024-01-01T12:00:00.000Z",
      "updatedAt": "2024-01-01T12:00:00.000Z"
    }
  }
}
```


POST /api/auth/logout

Realiza logout do usuário (invalida o token).

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Resposta de Sucesso (200):

```
{
  "success": true,
  "message": "Logout realizado com sucesso"
}
```


3.2.2. Endpoints de Jogos

Base URL: `http://localhost:3000/api/games`

GET /api/games

Lista todos os jogos disponíveis.

Parâmetros de Query (opcionais):

- `type`: Filtrar por tipo de jogo (slot, roulette, blackjack)

Exemplo de Requisição:

GET /api/games?type=slot

Resposta de Sucesso (200):


```
{
  "success": true,
  "data": {
    "games": [
      {
        "id": "uuid-do-jogo",
        "name": "Slot Clássico",
        "type": "slot",
        "minBet": 1.00,
```

```
        "maxBet": 100.00,
        "rtp": 95.50,
        "isActive": true,
        "description": "Um slot machine clássico com 3 rolos",
        "rules": {
          "paylines": 1,
          "reels": 3,
          "symbols": ["🍒", "🍋", "🍊", "🍇", "⭐", "💎", "7️⃣"]
        },
        "createdAt": "2024-01-01T12:00:00.000Z",
        "updatedAt": "2024-01-01T12:00:00.000Z"
      }
    ],
    "count": 1
  }
}
```


GET /api/games/:id

Obtém detalhes de um jogo específico.

Resposta de Sucesso (200):


```
{
  "success": true,
  "data": {
```

```
    "game": {
      "id": "uuid-do-jogo",
      "name": "Slot Clássico",
      "type": "slot",
      "minBet": 1.00,
      "maxBet": 100.00,
      "rtp": 95.50,
      "isActive": true,
      "description": "Um slot machine clássico com 3 rolos",
      "rules": {
        "paylines": 1,
        "reels": 3,
        "symbols": ["🍒", "🍋", "🍊", "🍇", "⭐", "💎", "7️⃣"]
      },
      "createdAt": "2024-01-01T12:00:00.000Z",
      "updatedAt": "2024-01-01T12:00:00.000Z"
    }
  }
}
```


POST /api/games/slot/play

Joga uma rodada no slot machine.

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Corpo da Requisição:

```
{
  "gameId": "uuid-do-jogo-slot",
  "amount": 10.00
}
```

Resposta de Sucesso (200):

```
{
  "success": true,
  "data": {
    "result": ["🍒", "🍒", "🍋"],
    "isWin": true,
    "winAmount": 5.00,
    "multiplier": 0.5,
    "newBalance": 95.00,
    "betTransaction": "uuid-da-transacao"
  }
}
```

Possíveis Erros:

- `404 Not Found`: Jogo não encontrado
- `400 Bad Request`: Valor de aposta inválido ou saldo insuficiente

POST /api/games/roulette/play

Joga uma rodada na roleta.

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Corpo da Requisição:


```
{
  "gameId": "uuid-do-jogo-roleta",
  "amount": 20.00,
  "bet": "red"
}
```


Tipos de aposta válidos:

- Cores: `red`, `black`
- Paridade: `even`, `odd`
- Posição: `low` (1-18), `high` (19-36)
- Números específicos: `"0"`, `"1"`, `"2"`, ..., `"36"`

Resposta de Sucesso (200):


```
{
  "success": true,
  "data": {
    "result": {
      "number": 7,
      "color": "red",
      "isRed": true,
      "isBlack": false,
      "isEven": false,
```

```
      "isOdd": true,
      "isLow": true,
      "isHigh": false
    },
    "bet": "red",
    "isWin": true,
    "winAmount": 40.00,
    "multiplier": 2,
    "newBalance": 120.00,
    "betTransaction": "uuid-da-transacao"
  }
}
```


3.2.3. Endpoints de Carteira

Base URL: `http://localhost:3000/api/wallet`

GET /api/wallet/balance

Obtém o saldo atual do usuário.

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Resposta de Sucesso (200):


```
{
  "success": true,
  "data": {
```

```
    "balance": 150.75,
    "userId": "uuid-do-usuario"
  }
}
```


GET /api/wallet/transactions

Obtém o histórico de transações do usuário.

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Parâmetros de Query (opcionais):

- `page`: Número da página (padrão: 1)
- `limit`: Itens por página (padrão: 20)

Exemplo de Requisição:

GET /api/wallet/transactions?page=1&limit=10

Resposta de Sucesso (200):


```
{
  "success": true,
  "data": {
    "transactions": [
      {
        "id": "uuid-da-transacao",
        "userId": "uuid-do-usuario",
        "type": "bet",
```

```
      "amount": 10.00,
      "status": "completed",
      "description": "Aposta no jogo Slot Clássico",
      "gameId": "uuid-do-jogo",
      "createdAt": "2024-01-01T12:00:00.000Z",
      "updatedAt": "2024-01-01T12:00:00.000Z"
    }
  ],
  "pagination": {
    "page": 1,
    "limit": 10,
    "total": 1
  }
 }
}
```


POST /api/wallet/deposit

Realiza um depósito na carteira do usuário.

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Corpo da Requisição:


```
{
  "amount": 100.00
```

```
}
```

Resposta de Sucesso (200):

```
{
  "success": true,
  "message": "Depósito realizado com sucesso",
  "data": {
    "transaction": {
      "id": "uuid-da-transacao",
      "userId": "uuid-do-usuario",
      "type": "deposit",
      "amount": 100.00,
      "status": "completed",
      "description": "Depósito na carteira",
      "gameId": null,
      "createdAt": "2024-01-01T12:00:00.000Z",
      "updatedAt": "2024-01-01T12:00:00.000Z"
    },
    "newBalance": 250.75
  }
}
```

POST /api/wallet/withdraw

Realiza um saque da carteira do usuário.

Headers Obrigatórios:

Authorization: Bearer jwt-access-token

Corpo da Requisição:


```
{
  "amount": 50.00
}
```


Resposta de Sucesso (200):


```
{
  "success": true,
  "message": "Saque realizado com sucesso",
  "data": {
    "transaction": {
      "id": "uuid-da-transacao",
      "userId": "uuid-do-usuario",
      "type": "withdrawal",
      "amount": 50.00,
      "status": "completed",
      "description": "Saque da carteira",
      "gameId": null,
      "createdAt": "2024-01-01T12:00:00.000Z",
      "updatedAt": "2024-01-01T12:00:00.000Z"
    },
```

```
    "newBalance": 200.75
  }
}
```

Possíveis Erros:

- `400 Bad Request`: Saldo insuficiente ou valor inválido

3.3. Testando com Postman

Agora vamos criar uma coleção completa no Postman para testar todas as APIs.

Passo 1: Criar uma nova coleção

1. Abra o Postman
2. Clique em "New" → "Collection"
3. Nomeie a coleção como "Casino Backend API"
4. Adicione uma descrição: "Coleção para testar todas as APIs do backend do cassino"

Passo 2: Configurar variáveis de ambiente

1. Clique no ícone de engrenagem (Settings) no canto superior direito
2. Clique em "Add" para criar um novo ambiente
3. Nomeie como "Casino Local"
4. Adicione as seguintes variáveis:
    - `base_url`: `http://localhost:3000/api`
    - `access_token`: (deixe vazio por enquanto)

Passo 3: Criar requisições de teste

Teste 1: Health Check

GET {{base_url}}/health

Teste 2: Registrar Usuário

POST {{base_url}}/auth/register

Content-Type: application/json

```
{
```

```
  "email": "teste@casino.com",
  "username": "testuser",
  "password": "Teste123!",
  "firstName": "Usuário",
  "lastName": "Teste"
}
```

Script de Teste (aba Tests):

```
pm.test("Status code is 201", function () {
    pm.response.to.have.status(201);
});
pm.test("Response has access token", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.data.tokens.accessToken).to.exist;
    pm.environment.set("access_token", jsonData.data.tokens.accessToken);
});
```

Teste 3: Login

```
POST {{base_url}}/auth/login
Content-Type: application/json
{
  "email": "teste@casino.com",
  "password": "Teste123!"
}
```

Script de Teste:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Response has access token", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.data.tokens.accessToken).to.exist;
    pm.environment.set("access_token", jsonData.data.tokens.accessToken);
});
```

Teste 4: Obter Perfil

```
GET {{base_url}}/auth/profile
Authorization: Bearer {{access_token}}
```

Teste 5: Listar Jogos

```
GET {{base_url}}/games
```

Teste 6: Fazer Depósito

```
POST {{base_url}}/wallet/deposit
Authorization: Bearer {{access_token}}
Content-Type: application/json
{
  "amount": 100.00
}
```

Teste 7: Verificar Saldo

```
GET {{base_url}}/wallet/balance
```

Authorization: Bearer {{access_token}}

Teste 8: Jogar Slot

POST {{base_url}}/games/slot/play

Authorization: Bearer {{access_token}}

Content-Type: application/json

{

"gameId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",

"amount": 10.00

}

Teste 9: Jogar Roleta

POST {{base_url}}/games/roulette/play

Authorization: Bearer {{access_token}}

Content-Type: application/json

{

"gameId": "b2c3d4e5-f6g7-8901-bcde-f23456789012",

"amount": 20.00,

"bet": "red"

}

Teste 10: Histórico de Transações

GET {{base_url}}/wallet/transactions

Authorization: Bearer {{access_token}}

3.4. Testando com REST Client no VSCode

Como alternativa ao Postman, você pode usar a extensão REST Client do VSCode. Crie um arquivo `requests.http` na raiz do projeto:

```
### Variáveis

@baseUrl = http://localhost:3000/api

@accessToken =

### Health Check

GET {{baseUrl}}/health

### Registrar usuário

POST {{baseUrl}}/auth/register

Content-Type: application/json

{

  "email": "teste@casino.com",

  "username": "testuser",

  "password": "Teste123!",

  "firstName": "Usuário",

  "lastName": "Teste"

}

### Login

POST {{baseUrl}}/auth/login

Content-Type: application/json

{

  "email": "teste@casino.com",

  "password": "Teste123!"
```

```
}

### Obter perfil (substitua o token)

GET {{baseUrl}}/auth/profile

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

### Listar jogos

GET {{baseUrl}}/games

### Fazer depósito

POST {{baseUrl}}/wallet/deposit

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Content-Type: application/json

{

  "amount": 100.00

}

### Verificar saldo

GET {{baseUrl}}/wallet/balance

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

### Jogar slot

POST {{baseUrl}}/games/slot/play

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Content-Type: application/json

{

  "gameId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
```

```
  "amount": 10.00
}

### Jogar roleta

POST {{baseUrl}}/games/roulette/play

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Content-Type: application/json

{
  "gameId": "b2c3d4e5-f6g7-8901-bcde-f23456789012",
  "amount": 20.00,
  "bet": "red"
}

### Histórico de transações

GET {{baseUrl}}/wallet/transactions

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

3.5. Monitoramento e Logs

Passo 1: Visualizar logs da aplicação

Os logs são salvos na pasta `logs/` na raiz do projeto. Você pode visualizá-los em tempo real:

```
# Ver logs de erro

tail -f logs/error.log

# Ver todos os logs

tail -f logs/combined.log
```

Passo 2: Monitorar contêineres Docker

```
# Ver status dos contêineres

docker compose ps

# Ver logs do backend

docker compose logs -f backend

# Ver logs do banco de dados

docker compose logs -f db
```

Passo 3: Conectar ao banco de dados com DBeaver

1. Abra o DBeaver
2. Conecte-se ao banco usando as credenciais do `docker-compose.yml`
3. Explore as tabelas criadas pelas migrações
4. Execute queries para verificar os dados:

```
-- Ver usuários registrados

SELECT id, email, username, balance, created_at FROM users;

-- Ver transações

SELECT t.*, u.username

FROM transactions t

JOIN users u ON t.user_id = u.id

ORDER BY t.created_at DESC;

-- Ver jogos disponíveis

SELECT * FROM games WHERE is_active = true;
```

3.6. Testes Automatizados

Para uma aplicação robusta, é importante ter testes automatizados. Vamos criar alguns testes básicos usando Jest.

Passo 1: Configurar Jest

Crie o arquivo `jest.config.js` na raiz do projeto:

```
module.exports = {
  testEnvironment: 'node',
  testMatch: ['**/tests/**/*.test.js'],
  collectCoverageFrom: [
    'src/**/*.js',
    '!src/server.js',
    '!src/config/**',
  ],
  coverageDirectory: 'coverage',
  coverageReporters: ['text', 'lcov', 'html'],
};
```

Passo 2: Criar testes unitários

Crie o arquivo `src/tests/unit/authService.test.js`:

```
const AuthService = require('../../services/authService');
const User = require('../../models/User');
// Mock do modelo User
jest.mock('../../models/User');
describe('AuthService', () => {
  beforeEach(() => {
```

```
  jest.clearAllMocks();
});

describe('register', () => {
  it('deve registrar um novo usuário com sucesso', async () => {
    const userData = {
      email: 'test@example.com',
      username: 'testuser',
      password: 'Test123!',
      firstName: 'Test',
      lastName: 'User'
    };
    const mockUser = {
      id: 'test-id',
      ...userData,
      toJSON: () => ({ id: 'test-id', ...userData })
    };
    User.findByEmail.mockResolvedValue(null);
    User.findByUsername.mockResolvedValue(null);
    User.create.mockResolvedValue(mockUser);
    const result = await AuthService.register(userData);
    expect(result).toHaveProperty('user');
    expect(result).toHaveProperty('tokens');
```

```
      expect(result.tokens).toHaveProperty('accessToken');
      expect(result.tokens).toHaveProperty('refreshToken');
    });
    it('deve lançar erro se email já existir', async () => {
      const userData = {
        email: 'existing@example.com',
        username: 'testuser',
        password: 'Test123!',
        firstName: 'Test',
        lastName: 'User'
      };
      User.findByEmail.mockResolvedValue({ id: 'existing-user' });
      await expect(AuthService.register(userData)).rejects.toThrow('Email já está em uso');
    });
  });
});
```

Passo 3: Executar testes

```
# Executar todos os testes
npm test
# Executar testes em modo watch
npm run test:watch
```

Executar testes com coverage

npm test -- --coverage

3.7. Validação de Segurança

Passo 1: Testar rate limiting

Faça múltiplas requisições rapidamente para verificar se o rate limiting está funcionando:

Fazer 10 requisições rapidamente

for i in {1..10}; do curl http://localhost:3000/api/health; done

Passo 2: Testar validação de entrada

Tente enviar dados inválidos para verificar se a validação está funcionando:

POST http://localhost:3000/api/auth/register

Content-Type: application/json

```
{
  "email": "email-invalido",
  "username": "ab",
  "password": "123",
  "firstName": "",
  "lastName": ""
}
```

Passo 3: Testar autenticação

Tente acessar endpoints protegidos sem token ou com token inválido:

GET http://localhost:3000/api/auth/profile

Authorization: Bearer token-invalido

Com essa documentação e conjunto de testes, você terá uma base sólida para desenvolver e manter seu backend de cassino. Na próxima seção, abordaremos as considerações finais e próximos passos para expandir o sistema.

4. Considerações Finais e Próximos Passos

Parabéns! Você acabou de construir um backend completo e funcional para um cassino online. Este sistema implementa as funcionalidades essenciais de autenticação, gerenciamento de carteira, jogos básicos e um sistema robusto de transações. Vamos agora abordar as considerações finais e os próximos passos para expandir e aprimorar seu sistema.

4.1. Resumo do que foi Implementado

Ao longo deste tutorial, você criou:

Infraestrutura e Configuração:

- Ambiente Docker completo com PostgreSQL e Node.js
- Configuração do VSCode com extensões essenciais
- Sistema de logs estruturado com Winston
- Middleware de segurança com Helmet, CORS e Rate Limiting
- Validação robusta de dados com Joi

Sistema de Autenticação:

- Registro e login de usuários
- Autenticação JWT com tokens de acesso e refresh
- Middleware de autorização baseado em roles
- Hash seguro de senhas com bcrypt
- Validação de força de senha

Gerenciamento de Carteira:

- Sistema de saldo virtual
- Transações de depósito e saque
- Histórico completo de transações
- Controle de saldo em tempo real

Sistema de Jogos:

- Implementação de Slot Machine com RNG criptográfico
- Jogo de Roleta Europeia

- Sistema de apostas com validação de limites
- Cálculo automático de ganhos e multiplicadores

Banco de Dados:

- Esquema relacional bem estruturado
- Migrações e seeds automatizados
- Índices para otimização de performance
- Triggers para atualização automática de timestamps

APIs RESTful:

- Endpoints bem documentados
- Respostas padronizadas
- Tratamento de erros centralizado
- Códigos de status HTTP apropriados

4.2. Aspectos de Segurança Implementados

Seu sistema já inclui várias medidas de segurança importantes:

Autenticação e Autorização:

- Tokens JWT com expiração
- Middleware de autenticação robusto
- Controle de acesso baseado em roles
- Validação de força de senha

Proteção de Dados:

- Hash seguro de senhas com bcrypt (12 rounds)
- Validação rigorosa de entrada de dados
- Sanitização automática via Joi
- Headers de segurança com Helmet

Proteção contra Ataques:

- Rate limiting para prevenir ataques DDoS
- Proteção contra SQL injection via queries parametrizadas
- CORS configurado adequadamente
- Logs detalhados para auditoria

Geração de Números Aleatórios:

- RNG criptográfico usando crypto.randomBytes
- Algoritmos justos para jogos
- Prevenção de manipulação de resultados

4.3. Próximos Passos para Expansão

Agora que você tem uma base sólida, aqui estão as próximas funcionalidades que você pode implementar:

4.3.1. Funcionalidades de Negócio

Novos Jogos:

- Blackjack com lógica completa de cartas
- Poker (Texas Hold'em)
- Baccarat
- Crash games
- Jogos ao vivo com dealers

Sistema de Bônus:

- Bônus de boas-vindas
- Cashback
- Programa de fidelidade
- Promoções temporárias
- Rodadas grátis

Gestão de Usuários Avançada:

- Verificação KYC (Know Your Customer)
- Limites de depósito e aposta
- Auto-exclusão
- Histórico de sessões
- Preferências de usuário

4.3.2. Integrações Externas

Gateways de Pagamento:

- Stripe para cartões de crédito
- PayPal
- Criptomoedas (Bitcoin, Ethereum)
- PIX (para o mercado brasileiro)
- Boleto bancário

Provedores de Jogos:

- Integração com provedores como Pragmatic Play
- NetEnt
- Microgaming
- Evolution Gaming (jogos ao vivo)

Serviços de Terceiros:

- Verificação de identidade (Jumio, Onfido)
- Detecção de fraude
- Análise de comportamento
- Geolocalização

4.3.3. Melhorias Técnicas

Performance e Escalabilidade:

- Cache com Redis para sessões e dados frequentes
- CDN para assets estáticos
- Load balancing com múltiplas instâncias
- Database sharding para grandes volumes
- Otimização de queries com índices compostos

Monitoramento e Observabilidade:

- Métricas com Prometheus
- Dashboards com Grafana
- APM (Application Performance Monitoring)
- Health checks avançados
- Alertas automatizados

DevOps e Deploy:

- CI/CD com GitHub Actions
- Deploy automatizado
- Ambientes de staging
- Backup automatizado do banco
- Disaster recovery

4.3.4. Compliance e Regulamentação

Licenciamento:

- Adequação às regulamentações locais
- Certificação de jogos justos
- Auditoria de RNG
- Relatórios regulatórios

Proteção de Dados:

- Compliance com LGPD/GDPR
- Criptografia de dados sensíveis
- Anonização de dados
- Políticas de retenção

4.4. Melhores Práticas para Produção

Antes de colocar seu sistema em produção, considere estas práticas essenciais:

4.4.1. Segurança em Produção

Configurações de Ambiente:

Variáveis de ambiente para produção

NODE_ENV=production

JWT_SECRET=chave-super-secreta-de-256-bits

DATABASE_URL=postgresql://user:password@prod-db:5432/casino_prod

REDIS_URL=redis://prod-redis:6379

HTTPS Obrigatório:

- Certificados SSL/TLS válidos
- Redirecionamento automático HTTP → HTTPS
- HSTS (HTTP Strict Transport Security)
- Certificate pinning

Backup e Recovery:

- Backup automático diário do banco
- Backup incremental a cada hora
- Teste de restore regular
- Replicação geográfica

4.4.2. Monitoramento em Produção

Métricas Essenciais:

- Tempo de resposta das APIs
- Taxa de erro por endpoint
- Número de usuários ativos
- Volume de transações
- Performance do banco de dados

Alertas Críticos:

- Falhas de autenticação em massa
- Transações suspeitas
- Indisponibilidade de serviços
- Uso excessivo de recursos
- Tentativas de fraude

4.4.3. Otimização de Performance

Cache Strategy:

```
// Exemplo de cache para jogos

const redis = require('redis');

const client = redis.createClient(process.env.REDIS_URL);

const getGamesWithCache = async () => {

  const cacheKey = 'games:active';

  const cached = await client.get(cacheKey);



  if (cached) {

    return JSON.parse(cached);

  }
```

```
  const games = await Game.findAll();
  await client.setex(cacheKey, 300, JSON.stringify(games)); // 5 minutos

  return games;
};
```

Connection Pooling:

```
// Configuração otimizada do pool de conexões
const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  max: 20, // máximo de conexões
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000,
});
```

4.5. Estrutura de Arquivos Final

Aqui está a estrutura completa do seu projeto:

```
casino-backend/
├── docker-compose.yml
├── Dockerfile
├── package.json
├── jest.config.js
├── .env.example
```

```
├── .gitignore
├── README.md
├── requests.http
├── src/
│   ├── app.js
│   ├── server.js
│   ├── config/
│   │   ├── database.js
│   │   ├── jwt.js
│   │   └── index.js
│   ├── middleware/
│   │   ├── auth.js
│   │   ├── validation.js
│   │   └── errorHandler.js
│   ├── routes/
│   │   ├── index.js
│   │   ├── auth.js
│   │   ├── games.js
│   │   └── wallet.js
│   ├── controllers/
│   │   ├── authController.js
│   │   ├── gameController.js
```

```
│   │   └── walletController.js
│   ├── models/
│   │   ├── User.js
│   │   ├── Game.js
│   │   └── Transaction.js
│   ├── services/
│   │   ├── authService.js
│   │   └── rngService.js
│   ├── utils/
│   │   └── logger.js
│   └── tests/
│       └── unit/
│           └── authService.test.js
├── database/
│   ├── migrate.js
│   ├── seed.js
│   ├── migrations/
│   │   └── 001_initial_schema.sql
│   └── seeds/
│       └── 001_initial_data.sql
├── logs/
```

```
│   ├── error.log
│   └── combined.log
└── docs/
    └── api/
        └── README.md
```

4.6. Comandos Úteis para o Dia a Dia

Desenvolvimento:

```
# Iniciar ambiente de desenvolvimento
docker compose up -d db
npm run dev

# Executar migrações
npm run migrate

# Executar seeds
npm run seed

# Executar testes
npm test

# Ver logs em tempo real
tail -f logs/combined.log
```

Produção:

```
# Build e deploy
docker compose -f docker-compose.prod.yml up -d --build

# Backup do banco
```

```
docker exec casino-db pg_dump -U user casino_db > backup_$(date +%Y%m%d).sql

# Monitorar recursos

docker stats

# Ver logs de produção

docker compose logs -f --tail=100
```

4.7. Recursos Adicionais e Referências

Para continuar aprendendo e aprimorando seu sistema, recomendo os seguintes recursos:

Documentação Oficial:

- Node.js Documentation [1]
- Express.js Guide [2]
- PostgreSQL Documentation [3]
- Docker Documentation [4]

Segurança:

- OWASP Top 10 [5]
- JWT Best Practices [6]
- Node.js Security Checklist [7]

Performance:

- Node.js Performance Best Practices [8]
- PostgreSQL Performance Tuning [9]

Testes:

- Jest Documentation [10]
- Supertest for API Testing [11]

4.8. Conclusão

Você agora possui um backend de cassino online completo, seguro e escalável. Este sistema implementa as melhores práticas de desenvolvimento, segurança e arquitetura, fornecendo uma base sólida para um negócio real.

O que você aprendeu:

- Configuração de ambiente de desenvolvimento profissional
- Arquitetura de microsserviços com Docker
- Implementação de APIs RESTful robustas
- Sistema de autenticação e autorização seguro
- Gerenciamento de transações financeiras
- Implementação de jogos com RNG justo
- Testes automatizados e documentação de APIs
- Práticas de segurança e compliance

Próximos passos recomendados:

1. Implemente testes de integração mais abrangentes
2. Adicione novos jogos seguindo os padrões estabelecidos
3. Integre um gateway de pagamento real
4. Implemente cache com Redis para melhor performance
5. Configure monitoramento e alertas
6. Estude regulamentações locais para compliance

Lembre-se de que o desenvolvimento de software é um processo iterativo. Continue aprimorando seu sistema, adicionando funcionalidades e melhorando a segurança conforme necessário. Com a base sólida que você construiu, você está bem preparado para expandir e escalar seu cassino online.

Boa sorte com seu projeto e continue codificando!

Referências

[1] Node.js Documentation - https://nodejs.org/docs/ [2] Express.js Guide - https://expressjs.com/guide/ [3] PostgreSQL Documentation - https://www.postgresql.org/docs/ [4] Docker Documentation - https://docs.docker.com/ [5] OWASP Top 10 - https://owasp.org/www-project-top-ten/ [6] JWT Best Practices - https://auth0.com/blog/a-look-at-the-latest-draft-for-jwt-bcp/ [7] Node.js Security Checklist - https://blog.risingstack.com/node-js-security-checklist/ [8] Node.js Performance Best Practices - https://nodejs.org/en/docs/guides/simple-profiling/ [9] PostgreSQL Performance Tuning - https://wiki.postgresql.org/wiki/Performance_Optimization [10] Jest Documentation - https://jestjs.io/docs/getting-started [11] Supertest for API Testing - https://github.com/visionmedia/supertest

---

Autor: Manus AI
Data: Janeiro 2024
Versão: 1.0.0