Documentação dos Padrões de Projeto Utilizados

Scripts SQL

create_tables.sql

```sql
-- Tabela principal de usuários (herança para vendedor e comprador)
CREATE TABLE users (
 id CHAR(36) PRIMARY KEY,
 nome VARCHAR(100) NOT NULL,
 email VARCHAR(100) NOT NULL UNIQUE,
 senha VARCHAR(255) NOT NULL,
 celular VARCHAR(20),
 endereco TEXT,
 type VARCHAR(255),
 createdAt DATETIME NOT NULL,
 updatedAt DATETIME NOT NULL
);

-- Tabela de vendedor (herda de users)
CREATE TABLE vendedor (
 id CHAR(36) PRIMARY KEY,
 nota FLOAT DEFAULT 0.0,
 FOREIGN KEY (id) REFERENCES users(id)
);

-- Tabela de comprador (herda de users)
CREATE TABLE comprador (
 id CHAR(36) PRIMARY KEY,
 FOREIGN KEY (id) REFERENCES users(id)
);

-- Tabela de cavalos
CREATE TABLE cavalos (
 id CHAR(36) PRIMARY KEY,
 nome VARCHAR(100) NOT NULL,
 idade INT NOT NULL,
 raca VARCHAR(100) NOT NULL,
 preco DECIMAL(10,2) NOT NULL,
 descricao TEXT,
 disponivel BOOLEAN DEFAULT TRUE,
 createdAt DATETIME NOT NULL,
 updatedAt DATETIME NOT NULL,
 dono CHAR(36) NOT NULL,
 FOREIGN KEY (dono) REFERENCES users(id)
);
```

```sql
-- Tabela de anúncios
CREATE TABLE anuncios (
 id CHAR(36) PRIMARY KEY,
 titulo VARCHAR(150) NOT NULL,
 descricao TEXT,
 preco DECIMAL(10,2) NOT NULL,
 ativo BOOLEAN DEFAULT TRUE,
 createdAt DATETIME NOT NULL,
 updatedAt DATETIME NOT NULL,
 vendedorId CHAR(36) NOT NULL,
 cavaloId CHAR(36) NOT NULL,
 FOREIGN KEY (vendedorId) REFERENCES users(id),
 FOREIGN KEY (cavaloId) REFERENCES cavalos(id)
);

-- Tabela de mensagens
CREATE TABLE mensagens (
 id CHAR(36) PRIMARY KEY,
 conteudo TEXT NOT NULL,
 createAt DATETIME NOT NULL,
 remetente_id CHAR(36) NOT NULL,
 destinatario_id CHAR(36) NOT NULL,
 FOREIGN KEY (remetente_id) REFERENCES users(id),
 FOREIGN KEY (destinatario_id) REFERENCES users(id)
);
```

insert_data.sql (dados iniciais)

```sql
-- Exemplos de INSERT para cada tabela

-- Usuários
INSERT INTO users (id, nome, email, senha, celular, endereco, type,
createdAt, updatedAt)
VALUES ('uuid-1', 'João Vendedor', 'joao@exemplo.com', 'senha123',
'11999999999', 'Rua A, 123', 'vendedor', NOW(), NOW()),
       ('uuid-2', 'Maria Compradora', 'maria@exemplo.com', 'senha456',
'11988888888', 'Rua B, 456', 'comprador', NOW(), NOW());

-- Vendedor
INSERT INTO vendedor (id, nota) VALUES ('uuid-1', 4.5);
```

```sql
-- Comprador
INSERT INTO comprador (id) VALUES ('uuid-2');

-- Cavalos
INSERT INTO cavalos (id, nome, idade, raca, preco, descricao,
disponivel, createdAt, updatedAt, dono)
VALUES ('uuid-3', 'Trovão', 5, 'Mangalarga', 15000.00, 'Cavalo
premiado', TRUE, NOW(), NOW(), 'uuid-1');

-- Anúncios
INSERT INTO anuncios (id, titulo, descricao, preco, ativo, createdAt,
updatedAt, vendedorId, cavaloId)
VALUES ('uuid-4', 'Venda de Trovão', 'Cavalo saudável e premiado',
15000.00, TRUE, NOW(), NOW(), 'uuid-1', 'uuid-3');

-- Mensagens
INSERT INTO mensagens (id, conteudo, createAt, remetente_id,
destinatario_id)
VALUES ('uuid-5', 'Olá, tenho interesse no cavalo!', NOW(), 'uuid-2',
'uuid-1');

-- Exemplos de INSERT para cada tabela

-- Usuários
INSERT INTO users (id, nome, email, senha, celular, endereco, type,
createdAt, updatedAt)
VALUES ('uuid-1', 'João Vendedor', 'joao@exemplo.com', 'senha123',
'11999999999', 'Rua A, 123', 'vendedor', NOW(), NOW()),
       ('uuid-2', 'Maria Compradora', 'maria@exemplo.com', 'senha456',
'11988888888', 'Rua B, 456', 'comprador', NOW(), NOW());

-- Vendedor
INSERT INTO vendedor (id, nota) VALUES ('uuid-1', 4.5);

-- Comprador
INSERT INTO comprador (id) VALUES ('uuid-2');

-- Cavalos
INSERT INTO cavalos (id, nome, idade, raca, preco, descricao,
disponivel, createdAt, updatedAt, dono)
VALUES ('uuid-3', 'Trovão', 5, 'Mangalarga', 15000.00, 'Cavalo
premiado', TRUE, NOW(), NOW(), 'uuid-1');
```

```
-- Anúncios
INSERT INTO anuncios (id, titulo, descricao, preco, ativo, createdAt,
updatedAt, vendedorId, cavaloId)
VALUES ('uuid-4', 'Venda de Trovão', 'Cavalo saudável e premiado',
15000.00, TRUE, NOW(), NOW(), 'uuid-1', 'uuid-3');

-- Mensagens
INSERT INTO mensagens (id, conteudo, createAt, remetente_id,
destinatario_id)
VALUES ('uuid-5', 'Olá, tenho interesse no cavalo!', NOW(), 'uuid-2',
'uuid-1');
```

Conexão com DB — Singleton (TypeScript)
Implementação Singleton para a pool de conexões.

Código(database.ts)

```
import { DataSource } from 'typeorm';
import dotenv from 'dotenv';
import { fileURLToPath } from 'url';
import { dirname, join } from 'path';

// Load environment variables
dotenv.config();

// Get directory path for ES modules
const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

export const AppDataSource = new DataSource({
 type: 'postgres',
 host: process.env.DB_HOST || 'localhost',
 port: parseInt(process.env.DB_PORT as string) || 5432,
 username: process.env.DB_USERNAME || 'postgres',
 password: process.env.DB_PASSWORD || 'password',
 database: process.env.DB_NAME || 'equitrade',
  synchronize: process.env.NODE_ENV === 'development', // Auto-create
schema in development
 logging: process.env.NODE_ENV === 'development',
 entities: [
   join(__dirname, '..', 'db', 'entities', '*.{ts,js}')
 ],
```

```
    subscribers:    [join(__dirname,    '..',    'db',    'subscribers',
'*.{ts,js}')],
});

export default AppDataSource;
```

Repository Pattern

O padrão Repository foi utilizado para isolar a lógica de acesso a dados das entidades, como User (podendo ser estendido também para Comprador, Vendedor e outras). Dessa forma, as operações de persistência ficam centralizadas em classes específicas. Isso promove o desacoplamento entre a camada de negócio e a camada de dados, facilita a realização de testes e torna o sistema independente do SGBD escolhido.

Código (UserRepository.ts):

```
import { Repository } from 'typeorm';
import { AppDataSource } from '../../config/database';
import { User } from '../entities/User';
import { Comprador } from '../entities/Comprador';
import { Vendedor } from '../entities/Vendedor';

export class UserRepository {
 private compradorRepository: Repository<Comprador>;
 private vendedorRepository: Repository<Vendedor>;

 constructor() {
   this.compradorRepository = AppDataSource.getRepository(Comprador);
   this.vendedorRepository = AppDataSource.getRepository(Vendedor);
 }

 async findAll(): Promise<User[]> {
   const compradores = await this.compradorRepository.find();
   const vendedores = await this.vendedorRepository.find();
   return [...compradores, ...vendedores];
 }

 async findById(id: string): Promise<User | null> {
   // Try to find in Comprador first
   let user: User | null = await this.compradorRepository.findOne({
where: { id } });
   if (user) return user;
```

```
    // Then try Vendedor
    user = await this.vendedorRepository.findOne({ where: { id } });
    return user || null;
  }

  async findByEmail(email: string): Promise<User | null> {
    // Try to find in Comprador first
    let user: User | null = await this.compradorRepository.findOne({
where: { email } });
    if (user) return user;

    // Then try Vendedor
    user = await this.vendedorRepository.findOne({ where: { email } });
    return user || null;
  }

  async createComprador(userData: Partial<Comprador>):
Promise<Comprador> {
    const comprador = this.compradorRepository.create(userData);
    return await this.compradorRepository.save(comprador);
  }

  async createVendedor(userData: Partial<Vendedor>): Promise<Vendedor> {
    const vendedor = this.vendedorRepository.create(userData);
    return await this.vendedorRepository.save(vendedor);
  }

  async updateComprador(id: string, userData: Partial<Comprador>):
Promise<Comprador | null> {
    await this.compradorRepository.update(id, userData);
    return await this.compradorRepository.findOne({ where: { id } });
  }

  async updateVendedor(id: string, userData: Partial<Vendedor>):
Promise<Vendedor | null> {
    await this.vendedorRepository.update(id, userData);
    return await this.vendedorRepository.findOne({ where: { id } });
  }

  async delete(id: string): Promise<boolean> {
    // Try to delete from Comprador first
    let result = await this.compradorRepository.delete(id);
```

```
    if (result.affected && result.affected > 0) {
      return true;
    }

    // Then try Vendedor
    result = await this.vendedorRepository.delete(id);
    return (result.affected ?? 0) > 0;
  }

  async count(): Promise<number> {
    const compradorCount = await this.compradorRepository.count();
    const vendedorCount = await this.vendedorRepository.count();
    return compradorCount + vendedorCount;
  }

  async countCompradores(): Promise<number> {
    return await this.compradorRepository.count();
  }

  async countVendedores(): Promise<number> {
    return await this.vendedorRepository.count();
  }

  async existsByEmail(email: string): Promise<boolean> {
    const compradorExists = await this.compradorRepository.count({
where: { email } }) > 0;
    if (compradorExists) return true;

    const vendedorExists = await this.vendedorRepository.count({ where:
{ email } }) > 0;
    return vendedorExists;
  }

  // Type-specific finder methods
  async findCompradorById(id: string): Promise<Comprador | null> {
    return await this.compradorRepository.findOne({ where: { id } });
  }

  async findVendedorById(id: string): Promise<Vendedor | null> {
    return await this.vendedorRepository.findOne({ where: { id } });
  }

  async getAllCompradores(): Promise<Comprador[]> {
```

```
    return await this.compradorRepository.find();
  }

  async getAllVendedores(): Promise<Vendedor[]> {
    return await this.vendedorRepository.find();
  }
}
```

DAO Pattern

O padrão DAO é utilizado nas entidades para representar e mapear objetos do banco de dados. Cada classe encapsula os dados e fornece métodos para manipulação dos mesmos, trazendo abstração e organização.

Código (User.ts):

```
import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn,
UpdateDateColumn, TableInheritance, OneToMany } from 'typeorm';
import { Mensagem } from './Mensagem';

@Entity('users')
@TableInheritance({ column: { type: 'varchar', name: 'type' } })
export abstract class User {
  @PrimaryGeneratedColumn('uuid')
  id!: string;

  @Column({ type: 'varchar', length: 100 })
  nome!: string;

  @Column({ type: 'varchar', length: 100, unique: true })
  email!: string;

  @Column({ type: 'varchar', length: 255 })
  senha!: string;

  @Column({ type: 'varchar', length: 20, nullable: true })
  celular?: string;

  @Column({ type: 'text', nullable: true })
  endereco?: string;
```

```
 @CreateDateColumn()
 createdAt!: Date;

 @UpdateDateColumn()
 updatedAt!: Date;

 // Relacionamentos
 @OneToMany(() => Mensagem, mensagem => mensagem.remetente)
 mensagensEnviadas!: Mensagem[];

 @OneToMany(() => Mensagem, mensagem => mensagem.destinatario)
 mensagensRecebidas!: Mensagem[];

 // Métodos abstratos para implementação nas classes filhas
 abstract signUp(): Promise<void>;
 abstract login(email: string, senha: string): Promise<boolean>;
 abstract logout(): Promise<void>;
 abstract editarPerfil(dados: Partial<User>): Promise<void>;
 abstract enviarMensagem(destinatarioId: string, conteudo: string):
Promise<Mensagem>;
}
```

Controller Pattern

O padrão Controller foi aplicado para gerenciar as requisições HTTP relacionadas ao usuário. Ele recebe as requisições, valida os dados, chama os serviços necessários e retorna respostas, trazendo separação entre lógica de negócio e apresentação.

Código completo (UserController.ts):

```
import { Request, Response } from "express";
import { UserService } from "../services/UserService";
import {
 CreateUserDto,
 UpdateUserDto,
 CreateCompradorDto,
 CreateVendedorDto,
 USER_TYPE,
} from "../dto/user.dto";

export class UserController {
 private userService: UserService;
```

```
constructor() {
  this.userService = new UserService();
}

// GET /api/users - Get all users (both Compradores and Vendedores)
async getAllUsers(req: Request, res: Response): Promise<void> {
  try {
    const users = await this.userService.getAllUsers();
    res.json({
      success: true,
      count: users.length,
      data: users,
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      message: "Failed to fetch users",
      error: error instanceof Error ? error.message : "Unknown error",
    });
  }
     // Obs: Demais métodos. Iria ficar muito grande o documento
}
```

Service Pattern

O padrão Service centraliza regras de negócio e operações complexas relacionadas ao usuário, sendo chamado pelos controllers. Isso garante o desacoplamento entre controller e lógica de negócio, além de facilitar o reuso e a testabilidade.

Código (UserService.ts):

```
import { UserRepository } from '../db/repositories/UserRepository';
import { User } from '../db/entities/User';
import { Comprador } from '../db/entities/Comprador';
import { Vendedor } from '../db/entities/Vendedor';
import {
 CreateUserDto,
 UpdateUserDto,
 UserDto,
 CompradorDto,
 VendedorDto,
```

```
 CreateCompradorDto,
 CreateVendedorDto,
 UpdateCompradorDto,
 UpdateVendedorDto,
 USER_TYPE
} from '../dto/user.dto';

export class UserService {
 private userRepository: UserRepository;

 constructor() {
   this.userRepository = new UserRepository();
 }

 async getAllUsers(): Promise<UserDto[]> {
   const users = await this.userRepository.findAll();
   return users.map(user => this.toUserDto(user));
 }

 async getUserById(id: string): Promise<UserDto | null> {
   const user = await this.userRepository.findById(id);
   return user ? this.toUserDto(user) : null;
 }
     // Obs: Demais métodos. Iria ficar muito grande o documento
}
```