MedClinic MVP - Code Style Guide

📋 MedClinic MVP - Code Style Guide

Versão: 1.0 | **Data:** 24 Jan 2026 | **Linguagem:** TypeScript + Node.js

---

🎯 Princípios Fundamentais

1. **Simplicidade** - Código que um iniciante entenda em primeira leitura
2. **Testabilidade** - Sem reescrever para testar; injeção de dependências built-in
3. **Consistência** - Padrões únicos em toda a codebase
4. **Inglês** - Nomes de funções, classes, variáveis, comments, commits

---

📝 1. NOMEAÇÃO

Variáveis & Funções: `camelCase`

```
// ✅ BOM
const maxRetryAttempts = 3;
const userEmail = 'john@example.com';
const isValidEmail = (email: string): boolean => {};
const getUserById = (id: number) => {};

// ❌ RUIM
const max_retry_attempts = 3;            // snake_case
const MaxRetryAttempts = 3;              // PascalCase
const getUSERbyID = (id: number) => {}; // inconsistente
```

Classes: `PascalCase`

```
// ✅ BOM
class UserRepository {}
class AuthService {}
class ValidationError extends Error {}
class ProfessionalController {}
```

```
// ❌ RUIM
class userRepository {}
class auth_service {}
class validation_error {}
```

Constantes & Enums: `SCREAMING_SNAKE_CASE`

```
// ✅ BOM
const MAX_RETRY_ATTEMPTS = 3;
const DEFAULT_PAGE_SIZE = 20;
const JWT_EXPIRATION_HOURS = 24;

enum USER_ROLES {
  SYSTEM_ADMIN = 'system_admin',
  CLINIC_ADMIN = 'clinic_admin',
  HEALTH_PROFESSIONAL = 'health_professional',
  PATIENT = 'patient',
  RECEPTIONIST = 'receptionist',
}

// ❌ RUIM
const maxRetryAttempts = 3;     // constante deve ser
SCREAMING_SNAKE_CASE
const MAX_RETRY = 3;            // não é descritivo
```

Booleans: Prefixar com `is`, `has`, `can`, `should`

```
// ✅ BOM
const isValidEmail = true;
const hasPermission = false;
const canCreateUser = false;
const shouldRetry = true;

// ❌ RUIM
const validEmail = true;
const permission = false;
const create = false;
```

Métodos: Iniciar com verbo

```
// ✅ BOM
getUserById()
createAppointment()
updateUserEmail()
deleteTransaction()
validateEmail()
isEmailValid()
hasUserPermission()

// ❌ RUIM
getUser()               // qual user?
appointment()           // é ação ou dado?
userEmail()             // é getter ou setter?
emailValidator()        // não é verbo
```

🏗 2. ESTRUTURA DE PASTAS

Padrão por **responsabilidade**:

```
src/
├── config/
│   └── database.ts              # Singleton DB + env vars
├── controllers/
│   ├── AuthController.ts
│   ├── UserController.ts
│   ├── ProfessionalController.ts
│   └── AppointmentController.ts
├── services/
│   ├── AuthService.ts
│   ├── UserService.ts
│   ├── ProfessionalService.ts
│   ├── AppointmentService.ts
│   └── PaymentMockService.ts
├── repositories/
│   ├── UserRepository.ts
│   ├── ProfessionalRepository.ts
│   ├── AvailabilityRepository.ts
│   ├── AppointmentRepository.ts
│   ├── TransactionRepository.ts
│   └── CommissionSplitRepository.ts
│
```

```
├── models/
│   ├── User.ts
│   ├── Appointment.ts
│   ├── Professional.ts
│   └── types.ts                # Interfaces globais
├── middlewares/
│   ├── authMiddleware.ts
│   └── errorHandler.ts
├── utils/
│   ├── validators.ts
│   ├── logger.ts
│   └── helpers.ts
├── routes/
│   ├── auth.routes.ts
│   ├── users.routes.ts
│   ├── professionals.routes.ts
│   └── appointments.routes.ts
├── __tests__/
│   ├── auth.test.ts
│   ├── users.test.ts
│   ├── professionals.test.ts
│   └── appointments.test.ts
├── app.ts                      # Express setup
└── server.ts                   # Entry point
```

Regra: 1 arquivo = 1 classe/serviço principal

📦 3. ESTRUTURA DE CLASSES

Ordem de Elementos

```
class UserRepository {
  // 1️⃣ Propriedades privadas (fields)
  private database: Database;
  private logger: Logger;

  // 2️⃣ Constructor (injeção de dependências)
  constructor(database: Database, logger: Logger) {
    this.database = database;
    this.logger = logger;
  }

  // 3️⃣ Métodos públicos
  public getUserById(id: number): Promise<User | null> {
    return this.database.query('SELECT * FROM users WHERE id = ?',
[id]);
  }

  public listUsers(page: number, pageSize: number): Promise<User[]> {
    const offset = (page - 1) * pageSize;
    return this.database.query(
      'SELECT * FROM users LIMIT ? OFFSET ?',
      [pageSize, offset]
    );
  }

  // 4️⃣ Métodos privados (helpers)
  private mapRowToUser(row: any): User {
    return new User(row.id, row.name, row.email);
  }

  // 5️⃣ Getters/Setters (raros, apenas se necessário)
  public getDatabase(): Database {
    return this.database;
  }
}
```

Injeção de Dependências (Constructor Injection)

```
// ✅ TESTÁVEL - dependências são injetadas
class AppointmentService {
  constructor(
    private appointmentRepository: AppointmentRepository,
    private paymentService: PaymentMockService,
    private logger: Logger
  ) {}

  async createAppointment(data: CreateAppointmentDTO):
Promise<Appointment> {
    // Pode mockar appointmentRepository nos testes
    const appointment = await this.appointmentRepository.create(data);
    await this.paymentService.processPayment(appointment);
    return appointment;
  }
}

// ❌ DIFÍCIL DE TESTAR - dependências hardcoded
class AppointmentService {
  async createAppointment(data: CreateAppointmentDTO):
Promise<Appointment> {
    const repository = new AppointmentRepository(); // ← Não pode
mockar!
    const appointment = await repository.create(data);
    return appointment;
  }
}
```

Padrão Repository (Abstração de Dados)

```
// Interface (contrato)
interface IUserRepository {
  getUserById(id: number): Promise<User | null>;
  listUsers(page: number, pageSize: number): Promise<User[]>;
  createUser(data: CreateUserDTO): Promise<User>;
  updateUser(id: number, data: Partial<User>): Promise<User>;
  deleteUser(id: number): Promise<void>;
}

// Implementação
class UserRepository implements IUserRepository {
  constructor(private database: Database) {}

  async getUserById(id: number): Promise<User | null> {
    const row = await this.database.queryOne(
      'SELECT * FROM users WHERE id = ?',
      [id]
    );
    return row ? this.mapRowToUser(row) : null;
  }

  async listUsers(page: number, pageSize: number): Promise<User[]> {
    const offset = (page - 1) * pageSize;
    const rows = await this.database.query(
      'SELECT * FROM users LIMIT ? OFFSET ?',
      [pageSize, offset]
    );
    return rows.map((row) => this.mapRowToUser(row));
  }

  async createUser(data: CreateUserDTO): Promise<User> {
    const result = await this.database.run(
      'INSERT INTO users (name, email, password_hash) VALUES (?, ?,
?)',
      [data.name, data.email, data.passwordHash]
    );
    return new User(result.lastID, data.name, data.email);
  }

  private mapRowToUser(row: any): User {
    return new User(row.id, row.name, row.email);
  }
}

// Mock para testes
class MockUserRepository implements IUserRepository {
  private users: User[] = [];

  async getUserById(id: number): Promise<User | null> {
    return this.users.find((u) => u.id === id) || null;
  }

  async listUsers(): Promise<User[]> {
    return this.users;
  }

  async createUser(data: CreateUserDTO): Promise<User> {
    const user = new User(this.users.length + 1, data.name,
data.email);
    this.users.push(user);
    return user;
  }
}
```

🔄 4. ASYNC/AWAIT

Sempre use `async/await` para legibilidade:

```typescript
// ✅ BOM - async/await (legível para iniciantes)
async function createAppointment(data: CreateAppointmentDTO):
Promise<Appointment> {
  const professional = await this.professionalRepository.findById(
    data.professionalId
  );

  if (!professional) {
    throw new ProfessionalNotFoundError('Professional not found');
  }

  const appointment = await this.appointmentRepository.create(data);
  return appointment;
}

// ❌ EVITAR - promises encadeadas (difícil de ler)
function createAppointment(data: CreateAppointmentDTO):
Promise<Appointment> {
  return this.professionalRepository
    .findById(data.professionalId)
    .then((professional) => {
      if (!professional) throw new ProfessionalNotFoundError();
      return this.appointmentRepository.create(data);
    })
    .catch((error) => {
      throw error;
    });
}
```

⚠️ 5. TRATAMENTO DE ERROS

Ambos conforme contexto:

A. Exceções Customizadas (Para erros graves)

```typescript
// Definir erros customizados
class ValidationError extends Error {
  constructor(message: string, public field?: string) {
    super(message);
    this.name = 'ValidationError';
  }
}

class UserAlreadyExistsError extends Error {
  constructor(email: string) {
    super(`User with email ${email} already exists`);
    this.name = 'UserAlreadyExistsError';
  }
}

class UnauthorizedError extends Error {
  constructor(message: string = 'Unauthorized') {
    super(message);
    this.name = 'UnauthorizedError';
  }
}

// Usar exceptions para casos graves
async function registerUser(data: CreateUserDTO): Promise<User> {
  const exists = await this.userRepository.findByEmail(data.email);
  if (exists) {
    throw new UserAlreadyExistsError(data.email); // ← Grave, lança
exceção
  }

  const user = await this.userRepository.create(data);
  return user;
}

// Error handler global (middleware)
app.use((error: any, req: Request, res: Response, next: NextFunction)
=> {
  console.error(error);

  if (error instanceof ValidationError) {
    return res.status(400).json({
      success: false,
      error: {
        code: 'VALIDATION_ERROR',
        message: error.message,
        field: error.field,
      },
    });
  }

  if (error instanceof UserAlreadyExistsError) {
    return res.status(409).json({
      success: false,
      error: {
        code: 'USER_ALREADY_EXISTS',
        message: error.message,
      },
    });
  }

  // Erro genérico
  res.status(500).json({
    success: false,
    error: {
      code: 'INTERNAL_SERVER_ERROR',
      message: 'Something went wrong',
    },
  });
});
```

B. Result Objects (Para validações)

```
// Type de resultado (sem exceção)
type Result<T> = { success: true; data: T } | { success: false; error:
string };

// Usar para validações (não é erro grave)
async function validateEmail(email: string): Promise<Result<boolean>>
{
  const isValid = /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);

  if (!isValid) {
    return { success: false, error: 'Invalid email format' }; // ←
Valida, não grave
  }

  const exists = await this.userRepository.findByEmail(email);
  if (exists) {
    return { success: false, error: 'Email already exists' };
  }

  return { success: true, data: true };
}

// Usar no controller
async register(req: Request, res: Response) {
  const emailValidation = await
this.authService.validateEmail(req.body.email);

  if (!emailValidation.success) {
    return res.status(400).json({
      success: false,
      error: { code: 'INVALID_EMAIL', message: emailValidation.error
},
    });
  }

  // Proceder com sucesso
}
```

🔒 6. VALIDAÇÃO DE INPUTS

Use **guards no início** (early return) + exceptions para casos graves:

```
async function createAppointment(
  data: CreateAppointmentDTO
): Promise<Appointment> {
  // 1️⃣ Guards - validações rápidas
  if (!data.patientId) {
    throw new ValidationError('Patient ID is required', 'patientId');
  }

  if (!data.professionalId) {
    throw new ValidationError('Professional ID is required',
'professionalId');
  }

  if (data.date < new Date()) {
    throw new ValidationError('Date cannot be in the past', 'date');
  }

  // 2️⃣ Lógica principal (após guards)
  const professional = await this.professionalRepository.findById(
    data.professionalId
  );

  if (!professional) {
    throw new ProfessionalNotFoundError();
  }

  const appointment = await this.appointmentRepository.create(data);
  return appointment;
}
```

🧪 7. SUPORTE A TESTES

Interfaces para fácil mocking

```
// ✅ BOM - Interface bem definida
interface IAppointmentRepository {
  create(data: CreateAppointmentDTO): Promise<Appointment>;
  findById(id: number): Promise<Appointment | null>;
  listByPatient(patientId: number): Promise<Appointment[]>;
  update(id: number, data: Partial<Appointment>):
Promise<Appointment>;
  delete(id: number): Promise<void>;
}

// Implementação real
class AppointmentRepository implements IAppointmentRepository {
  constructor(private database: Database) {}

  async create(data: CreateAppointmentDTO): Promise<Appointment> {
    // SQL real
  }
}

// Mock para testes
```

```
class MockAppointmentRepository implements IAppointmentRepository {
  private appointments: Appointment[] = [];

  async create(data: CreateAppointmentDTO): Promise<Appointment> {
    const appointment = new Appointment(
      this.appointments.length + 1,
      data.patientId,
      data.professionalId,
      data.date,
      data.time
    );
    this.appointments.push(appointment);
    return appointment;
  }

  async findById(id: number): Promise<Appointment | null> {
    return this.appointments.find((a) => a.id === id) || null;
  }
}

// Teste
describe('AppointmentService', () => {
  let service: AppointmentService;
  let mockRepository: MockAppointmentRepository;

  beforeEach(() => {
    mockRepository = new MockAppointmentRepository();
    service = new AppointmentService(mockRepository,
mockPaymentService);
  });

  it('should create appointment with valid data', async () => {
    const result = await service.createAppointment({
      patientId: 1,
      professionalId: 2,
      date: new Date('2026-01-25'),
      time: '09:00',
    });

    expect(result.id).toBeDefined();
    expect(result.patientId).toBe(1);
  });
});
```

📝 8. COMENTÁRIOS & DOCUMENTAÇÃO

Código auto-explicativo - sem JSDoc exceto casos complexos:

```typescript
// ✅ BOM - nome e lógica clara
class UserRepository {
  async getUserById(id: number): Promise<User | null> {
    const row = await this.database.queryOne(
      'SELECT * FROM users WHERE id = ?',
      [id]
    );
    return row ? this.mapRowToUser(row) : null;
  }

  private mapRowToUser(row: any): User {
    return new User(row.id, row.name, row.email);
  }
}

// ❌ RUIM - precisa de comentário
class UserRepository {
  async gUBId(id: number) {
    // Gets user by id
    const r = await db.q('SELECT * FROM users WHERE id = ?', [id]);
    return r ? new User(r.id, r.name, r.email) : null;
  }
}
```

JSDoc para casos complexos

```typescript
/**
 * Calculates professional commission based on transaction amount.
 * Commission = net_amount * 60% (professional share)
 *
 * @param transactionAmount - Gross transaction amount (before MDR)
 * @param mdrPercentage - Merchant discount rate (default: 3.79%)
 * @returns Commission amount in cents
 */
function calculateProfessionalCommission(
  transactionAmount: number,
  mdrPercentage: number = 3.79
): number {
  const netAmount = transactionAmount * (1 - mdrPercentage / 100);
  return Math.floor(netAmount * 0.6 * 100); // Convert to cents
}
```

🔍 9. LOGGING

Use `console` simples - sem bibliotecas por enquanto:

```
// ✅ BOM
class UserService {
  async createUser(data: CreateUserDTO): Promise<User> {
    console.log('Creating user:', data.email);

    try {
      const user = await this.userRepository.create(data);
      console.log('User created successfully:', user.id);
      return user;
    } catch (error) {
      console.error('Error creating user:', error);
      throw error;
    }
  }
}

// Estruturado
console.log('[INFO]', 'User created', { userId: user.id, email:
user.email });
console.error('[ERROR]', 'Failed to create user', { email, reason:
error.message });
```

🔗 10. TIPOS & INTERFACES

Use interfaces para **contratos públicos**, tipos para dados internos:

```typescript
// types.ts - Interfaces públicas
export interface User {
  id: number;
  name: string;
  email: string;
  passwordHash: string;
  role: USER_ROLES;
  cpf: string;
  phone: string;
  createdAt: Date;
  deletedAt: Date | null;
}

export interface CreateUserDTO {
  name: string;
  email: string;
  password: string;
  cpf: string;
  phone: string;
}

// Internos - tipo anônimo é ok
type UserRow = {
  id: number;
  name: string;
  email: string;
  password_hash: string;
  role: string;
};

// Usar em classes
class User implements User {
  id: number;
  name: string;
  email: string;

  constructor(id: number, name: string, email: string) {
    this.id = id;
    this.name = name;
    this.email = email;
  }
}
```

🚀 11. EXEMPLO COMPLETO

Estrutura recomendada

```
// src/models/types.ts
export interface User {
  id: number;
  email: string;
  name: string;
}

export interface CreateUserDTO {
  email: string;
  name: string;
  password: string;
}

// src/repositories/UserRepository.ts
import { Database } from '../config/database';
import { User, CreateUserDTO } from '../models/types';

interface IUserRepository {
  getUserById(id: number): Promise<User | null>;
  createUser(data: CreateUserDTO): Promise<User>;
}

class UserRepository implements IUserRepository {
  constructor(private database: Database) {}

  async getUserById(id: number): Promise<User | null> {
    const row = await this.database.queryOne(
      'SELECT id, email, name FROM users WHERE id = ?',
      [id]
    );
    return row || null;
  }

  async createUser(data: CreateUserDTO): Promise<User> {
    const result = await this.database.run(
      'INSERT INTO users (email, name, password_hash) VALUES (?, ?,
?)',
      [data.email, data.name, hashPassword(data.password)]
    );
    return {
      id: result.lastID,
      email: data.email,
      name: data.name,
    };
  }
}

// src/services/UserService.ts
import { UserRepository } from '../repositories/UserRepository';
import { CreateUserDTO } from '../models/types';

class UserService {
  constructor(private userRepository: UserRepository) {}

  async registerUser(data: CreateUserDTO) {
    const existingUser = await
this.userRepository.getUserByEmail(data.email);

    if (existingUser) {
      throw new UserAlreadyExistsError(data.email);
    }

    return await this.userRepository.createUser(data);
  }
}

// src/controllers/UserController.ts
import { Request, Response } from 'express';
import { UserService } from '../services/UserService';

class UserController {
  constructor(private userService: UserService) {}

  async register(req: Request, res: Response) {
    try {
      const user = await this.userService.registerUser(req.body);

      res.status(201).json({
        success: true,
        data: user,
      });
    } catch (error) {
      if (error instanceof UserAlreadyExistsError) {
        return res.status(409).json({
          success: false,
          error: {
            code: 'USER_ALREADY_EXISTS',
            message: error.message,
          },
        });
      }

      throw error;
    }
  }
}

// src/__tests__/UserService.test.ts
import { expect } from 'chai';
import { UserService } from '../services/UserService';

class MockUserRepository {
  private users = [];

  async getUserById(id: number) {
    return this.users.find((u) => u.id === id) || null;
  }

  async createUser(data) {
    const user = { id: this.users.length + 1, ...data };
    this.users.push(user);
    return user;
  }
}

describe('UserService', () => {
  let service: UserService;
  let mockRepository: MockUserRepository;

  beforeEach(() => {
    mockRepository = new MockUserRepository();
    service = new UserService(mockRepository);
  });
  it('should create user with valid data', async () => {
    const result = await service.registerUser({
      email: 'john@example.com',
      name: 'John Doe',
      password: 'Password123',
    });

    expect(result.email).to.equal('john@example.com');
    expect(result.id).to.be.defined;
  });
});
```

📋 CHECKLIST - Antes de Fazer Commit

- ◯ Nomes em **camelCase** (funções/vars), **PascalCase** (classes), **SCREAMING_SNAKE_CASE** (constantes)
- ◯ Classes têm **Constructor** → **Public** → **Private** em ordem
- ◯ Dependências são **injetadas no constructor**
- ◯ Usa **async/await**, não promises
- ◯ Validações com **guards** + **exceptions** para casos graves
- ◯ Interfaces definidas em `models/types.ts`
- ◯ **Repositório** abstrai dados (interface + mock implementado)
- ◯ **Serviço** tem lógica de negócio
- ◯ **Controller** chama serviço e formata response
- ◯ Sem `console.log` de debug (apenas logs estruturados)
- ◯ Testes implementados com **mocks**

Próximo passo: Responda as 3 perguntas iniciais para detalharmos mais o guia! 🚀