Versão: 1.0 **Data:** Janeiro 2026

Sistema de Gestão de Clínicas Médicas

Janeiro 2026
Versão 1.0

Resumo

Este documento apresenta a especificação completa da implementação do banco de dados MedClinic, um sistema de gestão de clínicas médicas desenvolvido com SQLite e TypeScript. A arquitetura foi projetada com foco em integridade de dados, rastreabilidade, segurança e performance, respeitando todas as 28 regras de negócio especificadas.

Sumário

1 Visão Geral da Arquitetura

O banco de dados MedClinic foi modelado em SQLite com foco em:

- **Integridade de dados:** Uso extensivo de chaves estrangeiras e enumerações
- **Rastreabilidade:** Histórico completo de transações e operações
- **Segurança:** Separação de responsabilidades entre entidades
- **Performance:** Índices estratégicos e estrutura desnormalizada apenas quando necessário

1.1 Tecnologias

- **Banco de Dados:** SQLite (arquivo único em `/database/medclinic.db`)
- **Backend:** Node.js + Express.js
- **ORM/Driver:** better-sqlite3 (síncrono) ou sqlite3 (assíncrono)
- **Linguagem:** TypeScript com tipagem forte
- **Validação:** Implementada manualmente (sem bibliotecas externas)

1.2 Mudanças em Relação ao Modelo Original

O modelo descrito nas regras de negócio passou por evoluções estratégicas:

Aspecto	Original	Modelagem Final	Justificativa
Horários Profissional	String/texto	Tabela `availabilities`	Queries complexas sem processamento
Catálogo Exames	Manual	Tabela `exam_catalog`	Padroniza e facilita auditoria
Status Consulta	9 valores listados	Enum `appointment_status`	Consistência e previne estados inválidos
Rastreamento Pagamento	Monolítico	`transactions` + `commission_splits`	Separa o quê do como
Preço Congelado	Mencionado	Campos em `appointments` e `exam_requests`	Preserva histórico de preços

Tabela 1: Evolução do Modelo de Dados

2 Diagrama Entidade-Relacionamento (DER)

2.1 Blocos Principais

O modelo está organizado em 4 blocos principais:

1. **Usuários & Permissões:** Gestão de acesso e dados profissionais
2. **Consultas (CORE):** Núcleo do sistema de agendamentos
3. **Exames e Prescrições:** Pedidos médicos e resultados
4. **Financeiro:** Pagamentos, comissões e relatórios

3 Especificação das Tabelas com Exemplos TypeScript

3.1 Bloco 1: Usuários e Profissionais

3.1.1 Tabela: users

Propósito: Armazena todos os usuários do sistema com seus dados básicos e controle de acesso (6 roles diferentes).

Comentários Importantes:

- **password:** SEMPRE armazenar hash bcrypt. Nunca texto plano.
- **email:** UNIQUE garante que não há duplicatas
- **role:** Enum CHECK garante apenas 6 valores válidos
- **created_at/updated_at:** Rastreabilidade completa

3.1.2 Exemplo de Inserção em TypeScript

```typescript
interface User {
  id?: number;
  name: string;
  email: string;
  password: string; // Hash bcrypt
  role: 'patient' | 'receptionist' | 'lab_tech' |
        'health_professional' | 'clinic_admin' | 'system_admin';
  cpf?: string;
  phone?: string;
  created_at?: string;
  updated_at?: string;
}
```

Listing 1: Interface de Usuário

```typescript
class UserRepository {
  private db: Database;

  // Inserir novo paciente
  createPatient(userData: User): number {
    const stmt = this.db.prepare(`
      INSERT INTO users (name, email, password, role, cpf, phone,
         created_at)
      VALUES (?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP)
    `);

    const result = stmt.run(
      userData.name,
      userData.email,
      userData.password, // Deve ser hash bcrypt
      'patient',
      userData.cpf,
      userData.phone
    );
```

```
    return result.lastInsertRowid as number;
  }

  // Inserir novo médico
  createHealthProfessional(userData: User): number {
    const stmt = this.db.prepare(`
      INSERT INTO users (name, email, password, role, cpf, phone)
      VALUES (?, ?, ?, ?, ?, ?)
    `);

    const result = stmt.run(
      userData.name,
      userData.email,
      userData.password, // Hash bcrypt
      'health_professional',
      userData.cpf,
      userData.phone
    );
    return result.lastInsertRowid as number;
  }

  // Buscar por email (para login)
  findByEmail(email: string): User | null {
    const stmt = this.db.prepare(`SELECT * FROM users WHERE email =
       ?`);
    return stmt.get(email) as User | null;
  }

  // Buscar por ID
  findById(id: number): User | null {
    const stmt = this.db.prepare(`SELECT * FROM users WHERE id = ?`);
    return stmt.get(id) as User | null;
  }
}

// Uso
const userRepo = new UserRepository(db);

// Inserir paciente
const patientId = userRepo.createPatient({
  name: 'Maria Silva',
  email: 'maria@email.com',
  password: '$2b10$hash_bcrypt_aqui', // Nunca texto plano!
  role: 'patient',
  cpf: '12345678901',
  phone: '11987654321'
});

// Inserir médico
const docId = userRepo.createHealthProfessional({
  name: 'Dr. João Cardiologista',
  email: 'joao@clinica.com',
  password: '$2b10$hash_bcrypt_aqui...',
```

```
  role: 'health_professional',
  cpf: '98765432100',
  phone: '1133334444'
});
```

Listing 2: Repositório de Usuários

3.1.3 Tabela: professional_details

Propósito: Estende a tabela users com informações específicas de profissionais de saúde (especialidade, CRM, precificação).

```
interface ProfessionalDetails {
  id?: number;
  user_id: number;
  specialty: 'psicologia' | 'nutricao' | 'fonoaudiologia' |
     'fisioterapia' |
             'clinica_medica' | 'cardiologia' | 'oftalmologia' |
                'urologia' |
             'cirurgia_geral' | 'ortopedia' | 'neurologia';
  registration_number: string;
  council?: string;
  consultation_price: number;
  commission_percentage?: number;
}
```

Listing 3: Interface de Detalhes Profissionais

```
class ProfessionalDetailsRepository {
  private db: Database;

  // Inserir especialidade de um médico
  create(details: ProfessionalDetails): number {
    const stmt = this.db.prepare(`
      INSERT INTO professional_details (
        user_id, specialty, registration_number, council,
        consultation_price, commission_percentage
      ) VALUES (?, ?, ?, ?, ?, ?)
    `);

    const result = stmt.run(
      details.user_id,
      details.specialty,
      details.registration_number,
      details.council || null,
      details.consultation_price,
      details.commission_percentage || 60.00
    );
    return result.lastInsertRowid as number;
  }

  // Buscar por user_id
  findByUserId(userId: number): ProfessionalDetails | null {
    const stmt = this.db.prepare(
```

```
      `SELECT * FROM professional_details WHERE user_id = ?`
    );
    return stmt.get(userId) as ProfessionalDetails | null;
  }

  // Buscar por especialidade
  findBySpecialty(specialty: string): ProfessionalDetails[] {
    const stmt = this.db.prepare(
      `SELECT * FROM professional_details WHERE specialty = ?`
    );
    return stmt.all(specialty) as ProfessionalDetails[];
  }
}

// Uso
const profDetailsRepo = new ProfessionalDetailsRepository(db);

// Inserir especialidade do Dr. João
profDetailsRepo.create({
  user_id: 2,
  specialty: 'cardiologia',
  registration_number: 'CRM123456/SP',
  council: 'Conselho Regional de Medicina de São Paulo',
  consultation_price: 350.00,
  commission_percentage: 60.00
});

// Inserir especialidade da Psicóloga
profDetailsRepo.create({
  user_id: 3,
  specialty: 'psicologia',
  registration_number: 'CRP123456/SP',
  council: 'Conselho Regional de Psicologia',
  consultation_price: 120.00,
  commission_percentage: 60.00
});
```

Listing 4: Repositório de Detalhes Profissionais

Comentários Importantes:

- **consultation_price:** Preço CONGELADO quando agendamento é criado (não se altera com mudanças futuras)
- **commission_percentage:** Pode ser customizado por profissional, padrão é 60%

3.1.4 Tabela: availabilities

Propósito: Define a agenda semanal de cada profissional (RN-01: Disponibilidade de Horários).

```
interface Availability {
  id?: number;
  professional_id: number;
  day_of_week: number; // 0-6
  start_time: string; // "HH:MM"
```

```
  end_time: string; // "HH:MM"
  is_active?: boolean;
}
```

Listing 5: Interface de Disponibilidade

```
class AvailabilityRepository {
  private db: Database;

  // Inserir disponibilidade
  create(availability: Availability): number {
    const stmt = this.db.prepare(`
      INSERT INTO availabilities (
        professional_id, day_of_week, start_time, end_time, is_active
      ) VALUES (?, ?, ?, ?, ?)
    `);

    const result = stmt.run(
      availability.professional_id,
      availability.day_of_week,
      availability.start_time,
      availability.end_time,
      availability.is_active !== false ? 1 : 0
    );
    return result.lastInsertRowid as number;
  }

  // Buscar disponibilidades de um profissional
  findByProfessional(professionalId: number): Availability[] {
    const stmt = this.db.prepare(
      `SELECT * FROM availabilities
       WHERE professional_id = ?
       ORDER BY day_of_week, start_time`
    );
    return stmt.all(professionalId) as Availability[];
  }

  // Buscar disponibilidade específica
  findByDayAndTime(professionalId: number, dayOfWeek: number, time:
     string): Availability | null {
    const stmt = this.db.prepare(`
      SELECT * FROM availabilities
      WHERE professional_id = ?
      AND day_of_week = ?
      AND start_time <= ?
      AND end_time > ?
      AND is_active = 1
    `);
    return stmt.get(professionalId, dayOfWeek, time, time) as
       Availability | null;
  }
}

// Uso
```

```
47 const availabilityRepo = new AvailabilityRepository(db);
48
49 // Configurar agenda do Dr. João (ID = 2)
50 // Turno matutino terça-feira
51 availabilityRepo.create({
52   professional_id: 2,
53   day_of_week: 2, // Terça
54   start_time: '09:00',
55   end_time: '12:00',
56   is_active: true
57 });
58
59 // Turno vespertino terça-feira
60 availabilityRepo.create({
61   professional_id: 2,
62   day_of_week: 2, // Terça
63   start_time: '14:00',
64   end_time: '18:00',
65   is_active: true
66 });
```

Listing 6: Repositório de Disponibilidades

Comentários Importantes:

- **day_of_week:** 0=Dom, 1=Seg, 2=Ter, 3=Qua, 4=Qui, 5=Sex, 6=Sab
- **is_active:** Permite bloquear agendamentos sem deletar histórico

3.2 Bloco 2: Consultas (Core do Sistema)

3.2.1 Tabela: appointments

Propósito: Registra todas as consultas agendadas, seus status e pagamentos com preço CONGELADO.

```typescript
interface Appointment {
  id?: number;
  patient_id: number;
  professional_id: number;
  date: string; // YYYY-MM-DD
  time: string; // HH:MM
  duration_minutes?: number;
  type: 'presencial' | 'online';
  status?: 'scheduled' | 'confirmed' | 'waiting' | 'in_progress' |
           'completed' | 'no_show' | 'cancelled_by_patient' |
           'cancelled_by_clinic' | 'rescheduled';
  price: number;
  payment_status?: 'pending' | 'processing' | 'paid' | 'failed' |
                   'refunded' | 'partially_refunded';
  video_link?: string;
  room_number?: string;
  notes?: string;
  created_at?: string;
  updated_at?: string;
}
```

Listing 7: Interface de Agendamento

```typescript
class AppointmentRepository {
  private db: Database;

  // Agendar consulta
  create(appointment: Appointment): number {
    const stmt = this.db.prepare(`
      INSERT INTO appointments (
        patient_id, professional_id, date, time, duration_minutes,
        type, status, price, payment_status, video_link, room_number,
           created_at
      ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP)
    `);

    const result = stmt.run(
      appointment.patient_id,
      appointment.professional_id,
      appointment.date,
      appointment.time,
      appointment.duration_minutes || 30,
      appointment.type,
      appointment.status || 'scheduled',
      appointment.price,
      appointment.payment_status || 'pending',
      appointment.video_link || null,
      appointment.room_number || null
```

```
    );
    return result.lastInsertRowid as number;
  }

  // Buscar consulta por ID
  findById(id: number): Appointment | null {
    const stmt = this.db.prepare(`SELECT * FROM appointments WHERE id = ?`);
    return stmt.get(id) as Appointment | null;
  }

  // Buscar consultas do paciente
  findByPatientId(patientId: number): Appointment[] {
    const stmt = this.db.prepare(
      `SELECT * FROM appointments WHERE patient_id = ? ORDER BY date DESC, time DESC`
    );
    return stmt.all(patientId) as Appointment[];
  }

  // Buscar consultas do profissional (agenda)
  findByProfessionalId(professionalId: number, date?: string): Appointment[] {
    let query = `SELECT * FROM appointments WHERE professional_id = ?`;
    const params: any[] = [professionalId];

    if (date) {
      query += ` AND date = ?`;
      params.push(date);
    }
    query += ` ORDER BY date, time`;

    const stmt = this.db.prepare(query);
    return stmt.all(...params) as Appointment[];
  }

  // Validar RN-04 (Sem duplicação)
  checkConflict(patientId: number, professionalId: number, date: string): boolean {
    const stmt = this.db.prepare(`
      SELECT COUNT(*) as count FROM appointments
      WHERE patient_id = ? AND professional_id = ? AND DATE(date) = ?
      AND status NOT IN ('cancelled_by_patient', 'cancelled_by_clinic', 'no_show')
    `);
    const result = stmt.get(patientId, professionalId, date) as { count: number };
    return result.count > 0; // true = conflito
  }

  // Atualizar status
  updateStatus(id: number, status: string): void {
```

```
    const stmt = this.db.prepare(
      `UPDATE appointments SET status = ?, updated_at =
         CURRENT_TIMESTAMP WHERE id = ?`
    );
    stmt.run(status, id);
  }

  // Atualizar status de pagamento
  updatePaymentStatus(id: number, paymentStatus: string): void {
    const stmt = this.db.prepare(
      `UPDATE appointments SET payment_status = ?, updated_at =
         CURRENT_TIMESTAMP WHERE id = ?`
    );
    stmt.run(paymentStatus, id);
  }

  // Cancelar consulta
  cancel(id: number, reason: string, cancelledById: number): void {
    const stmt = this.db.prepare(`
      UPDATE appointments
      SET status = 'cancelled_by_patient', cancellation_reason = ?,
          cancelled_by = ?, updated_at = CURRENT_TIMESTAMP
      WHERE id = ?
    `);
    stmt.run(reason, cancelledById, id);
  }
}

// Uso
const appointmentRepo = new AppointmentRepository(db);

// Agendar consulta de cardiologia (online)
const appointmentId = appointmentRepo.create({
  patient_id: 1, // Maria Silva
  professional_id: 2, // Dr. João
  date: '2026-02-15',
  time: '14:00',
  duration_minutes: 30,
  type: 'online',
  status: 'scheduled',
  price: 350.00, // Congelado do consultation_price
  payment_status: 'pending',
  video_link: 'https://meet.zoom.com/medclinic/12345'
});

// Validar RN-04 (sem duplicação)
const hasConflict = appointmentRepo.checkConflict(1, 2, '2026-02-15');
if (hasConflict) {
  console.log('ERRO: Paciente já tem consulta com este
     profissional!');
}

// Atualizar status para confirmada
```

```
appointmentRepo.updateStatus(appointmentId, 'confirmed');
appointmentRepo.updatePaymentStatus(appointmentId, 'paid');

// Cancelar consulta
appointmentRepo.cancel(appointmentId, 'Paciente solicitou
   cancelamento', 1);
```

Listing 8: Repositório de Agendamentos

Comentários Importantes:

- **price:** MUITO IMPORTANTE armazenar preço congelado. Consultas antigas mantêm preço original.
- **status:** Estado máquina finito (9 valores possíveis). Sistema não deve aceitar valores fora deste conjunto.
- **date:** CHECK constraint impede agendamentos no passado.

3.3 Bloco 3: Exames e Prescrições

3.3.1 Tabela: exam_catalog

Propósito: Catálogo padronizado de exames (mudança importante em relação ao modelo original).

```
interface ExamCatalogItem {
  id?: number;
  name: string;
  type: 'blood' | 'image';
  base_price: number;
  description?: string;
}
```

Listing 9: Interface do Catálogo de Exames

```
class ExamCatalogRepository {
  private db: Database;

  // Inserir exame no catálogo
  create(exam: ExamCatalogItem): number {
    const stmt = this.db.prepare(
      `INSERT INTO exam_catalog (name, type, base_price, description)
       VALUES (?, ?, ?, ?)`
    );
    const result = stmt.run(
      exam.name,
      exam.type,
      exam.base_price,
      exam.description || null
    );
    return result.lastInsertRowid as number;
  }

  // Buscar todos exames
  findAll(): ExamCatalogItem[] {
    const stmt = this.db.prepare(
      `SELECT * FROM exam_catalog ORDER BY type, name`
    );
    return stmt.all() as ExamCatalogItem[];
  }

  // Buscar por tipo
  findByType(type: 'blood' | 'image'): ExamCatalogItem[] {
    const stmt = this.db.prepare(
      `SELECT * FROM exam_catalog WHERE type = ? ORDER BY name`
    );
    return stmt.all(type) as ExamCatalogItem[];
  }

  // Buscar por ID
  findById(id: number): ExamCatalogItem | null {
    const stmt = this.db.prepare(`SELECT * FROM exam_catalog WHERE id
       = ?`);
```

```
    return stmt.get(id) as ExamCatalogItem | null;
  }
}

// Uso
const examCatalogRepo = new ExamCatalogRepository(db);

// Inserir exames de sangue
examCatalogRepo.create({
  name: 'Hemograma Completo',
  type: 'blood',
  base_price: 80.00,
  description: 'Análise completa de células sanguíneas'
});

examCatalogRepo.create({
  name: 'Glicemia em Jejum',
  type: 'blood',
  base_price: 35.00,
  description: 'Medição de glicose (requer 8h jejum)'
});

// Inserir exames de imagem
examCatalogRepo.create({
  name: 'Raio-X Tórax',
  type: 'image',
  base_price: 120.00,
  description: 'Radiografia de tórax em PA e perfil'
});
```

Listing 10: Repositório do Catálogo de Exames

Comentários Importantes:

- **name UNIQUE:** Evita duplicação de exames no catálogo
- **base_price:** Preço padrão que pode ser ajustado por clínica sem afetar histórico

3.3.2 Tabela: exam_requests

Propósito: Requisições de exame solicitadas por profissionais (RN-09: Pedido médico obrigatório).

```
interface ExamRequest {
  id?: number;
  appointment_id: number;
  patient_id: number;
  requesting_professional_id: number;
  exam_catalog_id: number;
  clinical_indication: string;
  price: number;
  status?: string;
  payment_status?: string;
  scheduled_date?: string;
  result_file_url?: string;
```

```
  result_text?: string;
  lab_tech_id?: number;
  created_at?: string;
  updated_at?: string;
}
```

Listing 11: Interface de Requisição de Exame

```
class ExamRequestRepository {
  private db: Database;
  private examCatalogRepo: ExamCatalogRepository;

  constructor(db: Database, examCatalogRepo: ExamCatalogRepository) {
    this.db = db;
    this.examCatalogRepo = examCatalogRepo;
  }

  // Solicitar exame (RN-09: justificativa obrigatória)
  create(examRequest: ExamRequest): number {
    // Validar que clinical_indication não é vazio
    if (!examRequest.clinical_indication ||
        examRequest.clinical_indication.trim() === '') {
      throw new Error('Justificativa clínica é obrigatória (RN-09)');
    }

    // Buscar preço do catálogo (congelar)
    const catalogItem =
       this.examCatalogRepo.findById(examRequest.exam_catalog_id);
    if (!catalogItem) {
      throw new Error('Exame não encontrado no catálogo');
    }

    const stmt = this.db.prepare(`
      INSERT INTO exam_requests (
        appointment_id, patient_id, requesting_professional_id,
        exam_catalog_id, clinical_indication, price,
        status, payment_status, created_at
      ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP)
    `);

    const result = stmt.run(
      examRequest.appointment_id,
      examRequest.patient_id,
      examRequest.requesting_professional_id,
      examRequest.exam_catalog_id,
      examRequest.clinical_indication,
      catalogItem.base_price, // Congelado
      examRequest.status || 'pending_payment',
      examRequest.payment_status || 'pending'
    );
    return result.lastInsertRowid as number;
  }

  // Buscar exame por ID
```

```
46  findById(id: number): ExamRequest | null {
47    const stmt = this.db.prepare(`SELECT * FROM exam_requests WHERE
        id = ?`);
48    return stmt.get(id) as ExamRequest | null;
49  }
50
51  // Buscar exames do paciente
52  findByPatientId(patientId: number): ExamRequest[] {
53    const stmt = this.db.prepare(
54      `SELECT * FROM exam_requests WHERE patient_id = ? ORDER BY
          created_at DESC`
55    );
56    return stmt.all(patientId) as ExamRequest[];
57  }
58
59  // Atualizar status
60  updateStatus(id: number, status: string): void {
61    const stmt = this.db.prepare(
62      `UPDATE exam_requests SET status = ?, updated_at =
          CURRENT_TIMESTAMP WHERE id = ?`
63    );
64    stmt.run(status, id);
65  }
66
67  // Lançar resultado (lab_tech)
68  uploadResult(id: number, resultText: string, resultFileUrl?: string,
69               labTechId?: number): void {
70    const stmt = this.db.prepare(`
71      UPDATE exam_requests
72      SET result_text = ?, result_file_url = ?, lab_tech_id = ?,
73          status = 'ready', updated_at = CURRENT_TIMESTAMP
74      WHERE id = ?
75    `);
76
77    stmt.run(resultText, resultFileUrl || null, labTechId || null,
        id);
78  }
79
80  // RN-13: Validar acesso ao resultado
81  canViewResult(examId: number, userId: number, userRole: string):
      boolean {
82    const exam = this.findById(examId);
83    if (!exam) return false;
84
85    // Paciente pode ver seus próprios resultados
86    if (userRole === 'patient' && exam.patient_id === userId) return
        true;
87    // Médico que solicitou pode ver
88    if (userRole === 'health_professional' &&
89        exam.requesting_professional_id === userId) return true;
90    // Admin pode ver tudo
91    if (userRole === 'clinic_admin' || userRole === 'system_admin')
        return true;
```

```
92
93      return false;
94    }
95  }
96
97  // Uso
98  const examRequestRepo = new ExamRequestRepository(db,
        examCatalogRepo);
99
100 // Solicitar exame após consulta
101 const examRequestId = examRequestRepo.create({
102   appointment_id: 1,
103   patient_id: 1, // Maria Silva
104   requesting_professional_id: 2, // Dr. João
105   exam_catalog_id: 1, // Hemograma Completo
106   clinical_indication: 'Paciente relata fadiga e fraqueza. Necessário
        avaliar série vermelha',
107   price: 80.00
108 });
109
110 // Lab tech lança resultado
111 examRequestRepo.uploadResult(
112   examRequestId,
113   'Hemoglobina: 13.5 g/dL (Normal)\nGlóbulos vermelhos: 4.8
        milhões/µL (Normal)',
114   'https://storage/results/exam_1_2026-01-24.pdf',
115   4 // lab_tech_id
116 );
117
118 // Validar RN-13 (controle de acesso)
119 const canView = examRequestRepo.canViewResult(examRequestId, 1,
        'patient');
120 if (canView) {
121   console.log('Paciente pode visualizar resultado');
122 }
```

Listing 12: Repositório de Requisições de Exame

Comentários Importantes:

- **clinical_indication:** NÃO pode ser nulo. RN-09 exige justificativa.
- **price:** Congelado do catálogo. Pedidos antigos mantêm preço original.
- **RN-13:** Acesso restrito. Implementar no backend antes de retornar dados.

3.3.3 Tabela: prescriptions

Propósito: Armazena prescrições digitais emitidas durante consultas.

```
1  interface Prescription {
2    id?: number;
3    appointment_id: number;
4    patient_id: number;
5    professional_id: number;
```

```
  medication_name: string;
  dosage?: string;
  instructions?: string;
  is_controlled?: boolean;
  pdf_url?: string;
  created_at?: string;
}
```

Listing 13: Interface de Prescrição

```
class PrescriptionRepository {
  private db: Database;

  // Criar prescrição
  create(prescription: Prescription): number {
    const stmt = this.db.prepare(`
      INSERT INTO prescriptions (
        appointment_id, patient_id, professional_id, medication_name,
        dosage, instructions, is_controlled, pdf_url, created_at
      ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP)
    `);

    const result = stmt.run(
      prescription.appointment_id,
      prescription.patient_id,
      prescription.professional_id,
      prescription.medication_name,
      prescription.dosage || null,
      prescription.instructions || null,
      prescription.is_controlled ? 1 : 0,
      prescription.pdf_url || null
    );
    return result.lastInsertRowid as number;
  }

  // Buscar prescrições do paciente
  findByPatientId(patientId: number): Prescription[] {
    const stmt = this.db.prepare(
      `SELECT * FROM prescriptions WHERE patient_id = ? ORDER BY
         created_at DESC`
    );
    return stmt.all(patientId) as Prescription[];
  }

  // Buscar prescrições de uma consulta
  findByAppointmentId(appointmentId: number): Prescription[] {
    const stmt = this.db.prepare(
      `SELECT * FROM prescriptions WHERE appointment_id = ? ORDER BY
         created_at DESC`
    );
    return stmt.all(appointmentId) as Prescription[];
  }
}

```

```
// Uso
const prescriptionRepo = new PrescriptionRepository(db);

// Prescrever medicamento após consulta
prescriptionRepo.create({
  appointment_id: 1,
  patient_id: 1, // Maria Silva
  professional_id: 2, // Dr. João
  medication_name: 'Amoxicilina 500mg',
  dosage: '1 comprimido',
  instructions: 'A cada 8 horas durante 7 dias. Tomar com água',
  is_controlled: false,
  pdf_url: 'https://storage/prescriptions/123456.pdf'
});

// Prescrever medicamento controlado
prescriptionRepo.create({
  appointment_id: 1,
  patient_id: 1,
  professional_id: 2,
  medication_name: 'Alprazolam 0.5mg',
  dosage: '1 comprimido',
  instructions: 'Uma vez ao dia, antes de dormir. Usar por máximo 15
     dias',
  is_controlled: true, // Requer assinatura digital
  pdf_url: 'https://storage/prescriptions/123457.pdf'
});
```

Listing 14: Repositório de Prescrições

3.4 Bloco 4: Financeiro (Pagamentos e Comissionamento)

3.4.1 Tabela: transactions

Propósito: Registra TODA transação financeira com valores brutos, MDR e líquidos.

```typescript
interface Transaction {
  id?: number;
  type: 'appointment_payment' | 'exam_payment' | 'refund';
  reference_id: number;
  reference_type: 'appointment' | 'exam';
  payer_id: number;
  amount_gross: number;
  mdr_fee: number;
  amount_net: number;
  installments?: number;
  payment_method?: string;
  gateway_transaction_id?: string;
  card_brand?: string;
  card_last4?: string;
  status?: string;
  processed_at?: string;
  created_at?: string;
  updated_at?: string;
}
```

Listing 15: Interface de Transação

```typescript
class TransactionRepository {
  private db: Database;

  // Criar transação
  create(transaction: Transaction): number {
    // Validar MDR (3.79%)
    const expectedMdr = Math.round(transaction.amount_gross * 0.0379
       * 100) / 100;
    if (Math.abs(transaction.mdr_fee - expectedMdr) > 0.01) {
      throw new Error(
        `MDR inválido. Esperado: ${expectedMdr}, Recebido:
           ${transaction.mdr_fee}`
      );
    }

    // Validar amount_net = amount_gross - mdr_fee
    const expectedNet = transaction.amount_gross -
       transaction.mdr_fee;
    if (Math.abs(transaction.amount_net - expectedNet) > 0.01) {
      throw new Error(
        `Valor líquido inválido. Esperado: ${expectedNet}, Recebido:
           ${transaction.amount_net}`
      );
    }

    const stmt = this.db.prepare(`
      INSERT INTO transactions (
```

```
        type, reference_id, reference_type, payer_id, amount_gross,
        mdr_fee, amount_net, installments, payment_method,
        gateway_transaction_id, card_brand, card_last4,
        status, processed_at, created_at
      ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
         CURRENT_TIMESTAMP)
    `);

    const result = stmt.run(
      transaction.type,
      transaction.reference_id,
      transaction.reference_type,
      transaction.payer_id,
      transaction.amount_gross,
      transaction.mdr_fee,
      transaction.amount_net,
      transaction.installments || 1,
      transaction.payment_method || 'credit_card',
      transaction.gateway_transaction_id || null,
      transaction.card_brand || null,
      transaction.card_last4 || null,
      transaction.status || 'processing',
      transaction.processed_at || null
    );
    return result.lastInsertRowid as number;
  }

  // Buscar transação por ID
  findById(id: number): Transaction | null {
    const stmt = this.db.prepare(`SELECT * FROM transactions WHERE id
       = ?`);
    return stmt.get(id) as Transaction | null;
  }

  // Buscar transações do pagador
  findByPayerId(payerId: number): Transaction[] {
    const stmt = this.db.prepare(
      `SELECT * FROM transactions WHERE payer_id = ? ORDER BY
         created_at DESC`
    );
    return stmt.all(payerId) as Transaction[];
  }

  // Atualizar status
  updateStatus(id: number, status: string, processedAt?: string):
     void {
    const stmt = this.db.prepare(
      `UPDATE transactions SET status = ?, processed_at = ?,
       updated_at = CURRENT_TIMESTAMP WHERE id = ?`
    );
    stmt.run(status, processedAt || new Date().toISOString(), id);
  }
}
```

```
73
74 // Uso
75 const transactionRepo = new TransactionRepository(db);
76
77 // Processar pagamento de consulta
78 const transactionId = transactionRepo.create({
79   type: 'appointment_payment',
80   reference_id: 1, // ID do appointment
81   reference_type: 'appointment',
82   payer_id: 1, // Patient ID (Maria Silva)
83   amount_gross: 350.00,
84   mdr_fee: 13.27, // 3.79% de MDR
85   amount_net: 336.73, // 350.00 - 13.27
86   installments: 3, // 3x sem juros
87   payment_method: 'credit_card',
88   gateway_transaction_id: 'MOCK-' + Date.now(),
89   card_brand: 'visa',
90   card_last4: '1234', // Apenas últimos 4 dígitos!
91   status: 'paid',
92   processed_at: new Date().toISOString()
93 });
```

Listing 16: Repositório de Transações

Comentários Importantes:

- **card_last4:** NUNCA armazenar número completo. RN-17: apenas últimos 4 dígitos.
- **mdr_fee:** SEMPRE 3.79% (CloudWalk). Validar com verificação.
- **amount_net:** Deve ser amount_gross - mdr_fee.

3.4.2 Tabela: commission_splits

Propósito: Rastreia divisão de cada transação: profissional (60%), clínica (35%), sistema (5%).

```
1 interface CommissionSplit {
2   id?: number;
3   transaction_id: number;
4   recipient_id?: number;
5   recipient_type: 'professional' | 'clinic' | 'system';
6   percentage: number;
7   amount: number;
8   status?: string;
9   created_at?: string;
10 }
```

Listing 17: Interface de Divisão de Comissão

```
1 class CommissionSplitRepository {
2   private db: Database;
3
4   // Criar splits para uma transação
5   createSplits(transactionId: number, amountNet: number): void {
6     // 60% para profissional
```

```
    this.create({
      transaction_id: transactionId,
      recipient_type: 'professional',
      percentage: 60.00,
      amount: Math.round(amountNet * 0.60 * 100) / 100,
      status: 'pending'
    });

    // 35% para clínica
    this.create({
      transaction_id: transactionId,
      recipient_type: 'clinic',
      percentage: 35.00,
      amount: Math.round(amountNet * 0.35 * 100) / 100,
      status: 'pending'
    });

    // 5% para sistema
    this.create({
      transaction_id: transactionId,
      recipient_type: 'system',
      percentage: 5.00,
      amount: Math.round(amountNet * 0.05 * 100) / 100,
      status: 'pending'
    });
  }

  // Inserir split individual
  private create(split: CommissionSplit): number {
    const stmt = this.db.prepare(`
      INSERT INTO commission_splits (
        transaction_id, recipient_id, recipient_type,
        percentage, amount, status, created_at
      ) VALUES (?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP)
    `);

    const result = stmt.run(
      split.transaction_id,
      split.recipient_id || null,
      split.recipient_type,
      split.percentage,
      split.amount,
      split.status || 'pending'
    );
    return result.lastInsertRowid as number;
  }

  // Buscar comissões pendentes do profissional
  findPendingByProfessional(professionalId: number):
     CommissionSplit[] {
    const stmt = this.db.prepare(`
      SELECT * FROM commission_splits
      WHERE recipient_id = ? AND recipient_type = 'professional'
```

```
      AND status = 'pending'
      ORDER BY created_at DESC
    `);
    return stmt.all(professionalId) as CommissionSplit[];
  }

  // Relatório mensal de comissão (RN-28)
  getMonthlyCommission(professionalId: number, month: number, year:
     number): {
    totalAppointments: number;
    totalCommission: number;
    splits: CommissionSplit[];
  } {
    const startDate = `${year}-${String(month).padStart(2, '0')}-01`;
    const endDate = new Date(year, month,
       0).toISOString().split('T')[0];

    const commissionsStmt = this.db.prepare(`
      SELECT cs.* FROM commission_splits cs
      INNER JOIN transactions t ON cs.transaction_id = t.id
      WHERE cs.recipient_id = ?
      AND cs.recipient_type = 'professional'
      AND DATE(t.created_at) BETWEEN ? AND ?
      AND t.status = 'paid'
    `);

    const splits = commissionsStmt.all(
      professionalId, startDate, endDate
    ) as CommissionSplit[];
    const totalCommission = splits.reduce((sum, split) => sum +
       split.amount, 0);
    const totalAppointments = new Set(splits.map(s =>
       s.transaction_id)).size;

    return {
      totalAppointments,
      totalCommission,
      splits
    };
  }
}

// Uso
const commissionSplitRepo = new CommissionSplitRepository(db);

// Após processar pagamento
commissionSplitRepo.createSplits(1, 336.73);
// Resultado:
// |- Profissional (60%): R$ 202.04
// |- Clinica (35%): R$ 117.86
// L- Sistema (5%): R$ 16.83

// Buscar comissões pendentes do Dr. João
```

```
const pendingCommissions =
   commissionSplitRepo.findPendingByProfessional(2);
console.log(
  `Comissões pendentes: R$ ${
    pendingCommissions.reduce((sum, c) => sum + c.amount,
       0).toFixed(2)
  }`
);

// Relatório mensal (janeiro 2026, Dr. João ID = 2)
const monthlyReport = commissionSplitRepo.getMonthlyCommission(2, 1,
   2026);
console.log(`Total consultas: ${monthlyReport.totalAppointments}`);
console.log(`Total comissão: R$
   ${monthlyReport.totalCommission.toFixed(2)}`);
```

Listing 18: Repositório de Divisão de Comissões

3.4.3 Tabela: refunds

Propósito: Registra reembolsos (RN-21:
¿24h = 100%, RN-22:
¡24h = 70%).

```
class RefundRepository {
  private db: Database;

  // Calcular reembolso (RN-21 ou RN-22)
  calculateRefundAmount(amountNet: number,
                        hoursBeforeAppointment: number): number {
    if (hoursBeforeAppointment > 24) {
      // RN-21: >24h = 100% reembolso
      return amountNet;
    } else {
      // RN-22: <24h = 70% reembolso
      return Math.round(amountNet * 0.70 * 100) / 100;
    }
  }

  // Processar reembolso
  create(refund: {
    transaction_id: number;
    amount_refunded: number;
    reason: string;
    requested_by: number;
  }): number {
    const stmt = this.db.prepare(`
      INSERT INTO refunds (
        transaction_id, amount_refunded, reason, requested_by,
           created_at
      ) VALUES (?, ?, ?, ?, CURRENT_TIMESTAMP)
    `);

    const result = stmt.run(
```

```
      refund.transaction_id,
      refund.amount_refunded,
      refund.reason,
      refund.requested_by
    );
    return result.lastInsertRowid as number;
  }
}

// Uso
const refundRepo = new RefundRepository(db);

// Cenário 1: Cancelamento >24h (100% reembolso)
const refundAmount100 = refundRepo.calculateRefundAmount(336.73, 25);
// Resultado: 336.73 (100%)

// Cenário 2: Cancelamento <24h (70% reembolso)
const refundAmount70 = refundRepo.calculateRefundAmount(336.73, 12);
// Resultado: 235.71 (70%)

// Processar reembolso
refundRepo.create({
  transaction_id: 1,
  amount_refunded: refundAmount70,
  reason: 'cancelled_by_patient_less_24h',
  requested_by: 1 // Patient ID
});
```

Listing 19: Repositório de Reembolsos

3.4.4 Tabela: monthly_reports

Propósito: Relatórios mensais de comissão (RN-28: Repasse até dia 10 do mês).

```
class MonthlyReportRepository {
  private db: Database;
  private commissionSplitRepo: CommissionSplitRepository;

  constructor(db: Database, commissionSplitRepo:
     CommissionSplitRepository) {
    this.db = db;
    this.commissionSplitRepo = commissionSplitRepo;
  }

  // Gerar relatório mensal (cron job do dia 1°)
  generateMonthlyReport(professionalId: number, month: number, year:
     number): number {
    const prevMonth = month === 1 ? 12 : month - 1;
    const prevYear = month === 1 ? year - 1 : year;

    const startDate = `${prevYear}-${String(prevMonth).padStart(2,
       '0')}-01`;
    const endDate = new Date(prevYear, prevMonth, 0)
      .toISOString().split('T')[0];
```

```
18
19    const stmt = this.db.prepare(`
20      SELECT
21        COUNT(DISTINCT a.id) as total_appointments,
22        SUM(t.amount_gross) as total_gross_amount,
23        SUM(t.amount_net) as total_net_amount,
24        SUM(cs.amount) as total_commission
25      FROM commission_splits cs
26      INNER JOIN transactions t ON cs.transaction_id = t.id
27      INNER JOIN appointments a ON t.reference_id = a.id
28        AND t.reference_type = 'appointment'
29      WHERE cs.recipient_id = ?
30      AND cs.recipient_type = 'professional'
31      AND a.status = 'completed'
32      AND DATE(t.created_at) BETWEEN ? AND ?
33      AND t.status = 'paid'
34    `);
35
36    const result = stmt.get(professionalId, startDate, endDate) as {
37      total_appointments: number;
38      total_gross_amount: number;
39      total_net_amount: number;
40      total_commission: number;
41    };
42
43    const amountToReceive = (result.total_commission || 0) > 0
44      ? result.total_commission
45      : 0;
46
47    const reportStmt = this.db.prepare(`
48      INSERT INTO monthly_reports (
49        professional_id, month, year, total_appointments,
             total_gross_amount,
50        total_net_amount, total_commission, total_deductions,
             amount_to_receive,
51        payment_status, generated_at
52      ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP)
53    `);
54
55    const reportResult = reportStmt.run(
56      professionalId,
57      month,
58      year,
59      result.total_appointments || 0,
60      result.total_gross_amount || 0,
61      result.total_net_amount || 0,
62      result.total_commission || 0,
63      0, // Sem deduções por enquanto
64      amountToReceive,
65      'generated'
66    );
67    return reportResult.lastInsertRowid as number;
68  }
```

```
69
70    // Marcar relatório como pago
71    markAsPaid(reportId: number): void {
72      const stmt = this.db.prepare(
73        `UPDATE monthly_reports SET payment_status = 'paid',
74         payment_date = CURRENT_TIMESTAMP WHERE id = ?`
75      );
76      stmt.run(reportId);
77    }
78  }
79
80  // Uso (cron job executado todo dia 1° às 00:00)
81  const monthlyReportRepo = new MonthlyReportRepository(
82    db,
83    commissionSplitRepo
84  );
85
86  // Gerar relatórios para todos os profissionais
87  const allProfessionals = userRepo.findByRole('health_professional');
88
89  allProfessionals.forEach(prof => {
90    const reportId = monthlyReportRepo.generateMonthlyReport(prof.id,
        2, 2026);
91    console.log(`Relatório ${reportId} gerado para profissional
        ${prof.id}`);
92  });
```

Listing 20: Repositório de Relatórios Mensais

4 Setup de TypeScript com SQLite

4.1 Configuração Recomendada

```typescript
import Database from 'better-sqlite3';
import path from 'path';

export class MedClinicDatabase {
  private db: Database.Database;

  constructor() {
    const dbPath = path.join(process.cwd(), 'database',
       'medclinic.db');
    this.db = new Database(dbPath);

    // Habilitar foreign keys
    this.db.pragma('foreign_keys = ON');
    // Modo journal mais seguro
    this.db.pragma('journal_mode = WAL');
  }

  initialize(): void {
    const schema = require('fs').readFileSync(
      path.join(process.cwd(), 'src', 'database', 'schema.sql'),
      'utf-8'
    );

    this.db.exec(schema);
    console.log(' Banco de dados inicializado');
  }

  getConnection(): Database.Database {
    return this.db;
  }

  close(): void {
    this.db.close();
  }
}

export const database = new MedClinicDatabase();
```

Listing 21: Inicialização do Banco de Dados

4.2 Package.json Recomendado

Listing 22: Dependências do Projeto

```
{
  "dependencies": {
    "express": "^4.18.0",
    "better-sqlite3": "^9.0.0",
    "bcrypt": "^5.1.0",
```

```
    "jsonwebtoken": "^9.0.0",
    "dotenv": "^16.0.0"
  },
  "devDependencies": {
    "typescript": "^5.0.0",
    "@types/node": "^20.0.0",
    "@types/express": "^4.17.0",
    "@types/better-sqlite3": "^7.6.0",
    "@types/bcrypt": "^5.0.0"
  }
}
```

5 Conclusão

Este modelo de banco de dados foi projetado para atender aos mais altos padrões de qualidade profissional:

- ✓**Respeita TODAS as 28 regras de negócio** especificadas
- ✓**Implementa separação clara de responsabilidades** entre entidades
- ✓**Facilita auditoria completa** de transações, comissões e cancelamentos
- ✓**Previne inconsistências** com constraints e validações robustas
- ✓**Otimiza queries** com índices estratégicos
- ✓**Mantém histórico de preços** que não se altera com atualizações futuras
- ✓**Implementa segurança** nunca armazenando dados sensíveis completos
- ✓**Tipagem forte em TypeScript** para desenvolvimento seguro

O backend deve complementar este modelo implementando validações adicionais em camada aplicação (regras de negócio complexas), enquanto o banco de dados garante integridade estrutural de todos os dados.

Versão: 1.0 **Data:** Janeiro 2026