

PROJETO MULTIDISCIPLINAR

Sistema de Gestão Hospitalar e de Serviços de Saúde (SGHSS)

Trilha: Backend

Curso: Análise e Desenvolvimento de Sistemas

Disciplina: Projeto Multidisciplinar

Aluno: Gabriel Dietrich Guesser

RU: 4570311

Polo de Apoio: Porto

Semestre: 2025/2

Professor: Prof. Winston Sen Lun Fung, Me.

Data: Junho de 2025

SUMÁRIO

1. [Introdução](#)
 2. [Requisitos Funcionais e Não Funcionais](#)
 3. [Modelagem](#)
 4. [Implementação](#)
 5. [Plano de Testes](#)
 6. [Conclusão](#)
 7. [Referências](#)
 8. [Anexos](#)
-

1. INTRODUÇÃO

Este projeto apresenta o desenvolvimento do backend de um Sistema de Gestão Hospitalar e de Serviços de Saúde (SGHSS) para a instituição VidaPlus. O trabalho foi realizado como parte da disciplina de Projeto Multidisciplinar, com foco na trilha de desenvolvimento backend.

1.1 Contexto e Justificativa

A VidaPlus é uma rede de saúde que necessita de um sistema integrado para gerenciar hospitais, clínicas, laboratórios e equipes de home care. Com a crescente digitalização da saúde e a necessidade de conformidade com a Lei Geral de Proteção de Dados (LGPD), desenvolver um backend seguro e eficiente tornou-se essencial.

1.2 Objetivos

1.2.1 Objetivo Geral

Desenvolver uma API REST completa em Golang para o Sistema de Gestão Hospitalar e de Serviços de Saúde, implementando funcionalidades de autenticação, gerenciamento de pacientes, profissionais de saúde, consultas e prontuários eletrônicos.

1.2.2 Objetivos Específicos

- Implementar sistema de autenticação e autorização usando JWT
- Desenvolver operações CRUD para pacientes e profissionais de saúde
- Criar funcionalidades de agendamento de consultas
- Implementar sistema de prontuários eletrônicos
- Garantir validação de dados e integridade das informações
- Aplicar práticas de segurança e proteção de dados
- Documentar o sistema e criar testes automatizados

1.3 Metodologia

O projeto foi desenvolvido seguindo uma abordagem incremental, com as seguintes etapas: 1. Análise de requisitos e planejamento 2. Definição da arquitetura do sistema 3. Implementação das funcionalidades básicas 4. Desenvolvimento das funcionalidades avançadas 5. Testes e validação 6. Documentação final

2. REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

2.1 Requisitos Funcionais

RF001 - Autenticação e Autorização

Descrição: O sistema deve permitir autenticação segura de usuários e controle de acesso baseado em perfis.

Funcionalidades: - Cadastro de novos usuários - Login com email e senha - Geração e validação de tokens JWT - Controle de acesso por perfis (admin, medico, enfermeiro, tecnico)

Critérios de Aceitação: - Senhas devem ser criptografadas com bcrypt - Tokens JWT devem ter validade de 24 horas - Diferentes perfis devem ter diferentes níveis de acesso

RF002 - Gerenciamento de Pacientes

Descrição: O sistema deve permitir o gerenciamento completo de informações de pacientes.

Funcionalidades: - Criar novo paciente - Listar todos os pacientes - Buscar paciente por ID - Atualizar dados do paciente - Excluir paciente

Critérios de Aceitação: - CPF deve ser único no sistema - Campos obrigatórios: nome, CPF - Validação de formato de dados - Registro de timestamps de criação e atualização

RF003 - Gerenciamento de Profissionais de Saúde

Descrição: O sistema deve gerenciar informações dos profissionais de saúde.

Funcionalidades: - Cadastrar profissional de saúde - Listar profissionais - Buscar profissional por ID - Atualizar dados do profissional - Excluir profissional

CrITÉrios de Aceitação: - CRM/COREN deve ser único no sistema - Campos obrigatórios: nome, CRM/COREN, especialidade - Vinculação com perfil de usuário

RF004 - Agendamento de Consultas

Descrição: O sistema deve permitir o agendamento e gerenciamento de consultas médicas.

Funcionalidades: - Agendar nova consulta - Listar consultas - Buscar consulta por ID - Atualizar status da consulta - Cancelar consulta

CrITÉrios de Aceitação: - Vinculação obrigatória com paciente e profissional - Status válidos: agendada, realizada, cancelada - Registro de data/hora da consulta

RF005 - Prontuários Eletrônicos

Descrição: O sistema deve permitir a criação e gerenciamento de prontuários eletrônicos.

Funcionalidades: - Criar prontuário - Listar prontuários - Buscar prontuário por ID - Atualizar prontuário - Excluir prontuário

CrITÉrios de Aceitação: - Vinculação com paciente e profissional - Campos: diagnóstico, tratamento, medicamentos, observações - Registro de data do atendimento

2.2 Requisitos Não Funcionais

RNF001 - Segurança

- Autenticação obrigatória para todas as operações
- Criptografia de senhas com bcrypt
- Tokens JWT para autorização
- Validação de entrada de dados

RNF002 - Performance

- Tempo de resposta inferior a 1 segundo para operações básicas
- Suporte a operações concorrentes
- Uso eficiente de memória

RNF003 - Confiabilidade

- Validação de integridade de dados
- Tratamento adequado de erros
- Logs de operações críticas

RNF004 - Usabilidade

- API REST com padrões consistentes
- Documentação completa dos endpoints
- Mensagens de erro claras e informativas

RNF005 - Manutenibilidade

- Código bem estruturado e documentado
- Arquitetura em camadas
- Testes automatizados

3. MODELAGEM

3.1 Casos de Uso

O sistema SGHSS possui três atores principais que interagem com diferentes funcionalidades:

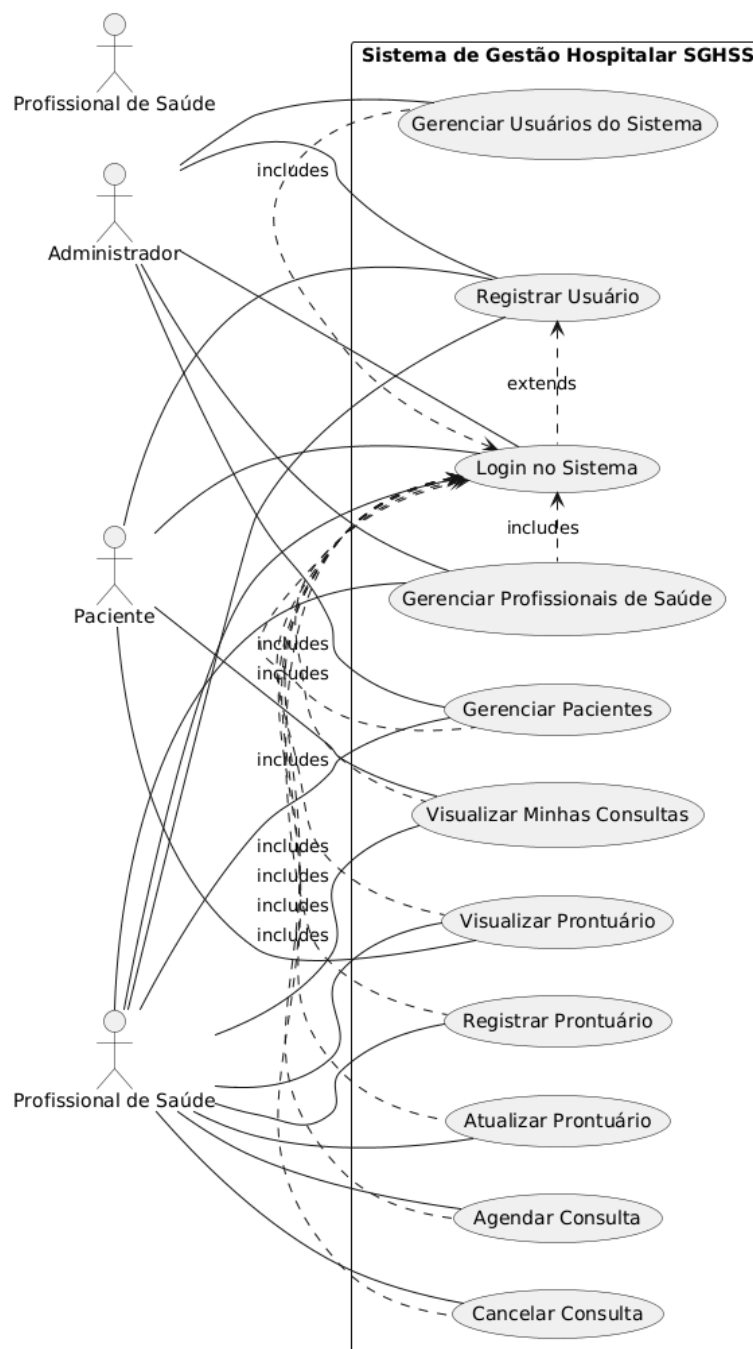
3.1.1 Atores

- **Administrador:** Gerencia usuários e configurações do sistema
- **Profissional de Saúde:** Acessa informações de pacientes, agenda consultas e registra prontuários

- **Sistema:** Executa validações e operações automáticas

3.1.2 Casos de Uso Principais

1. **Autenticar Usuário:** Login no sistema
2. **Gerenciar Pacientes:** CRUD de pacientes
3. **Gerenciar Profissionais:** CRUD de profissionais de saúde
4. **Agendar Consultas:** Gerenciamento de consultas
5. **Gerenciar Prontuários:** CRUD de prontuários eletrônicos



3.2 Diagrama de Classes

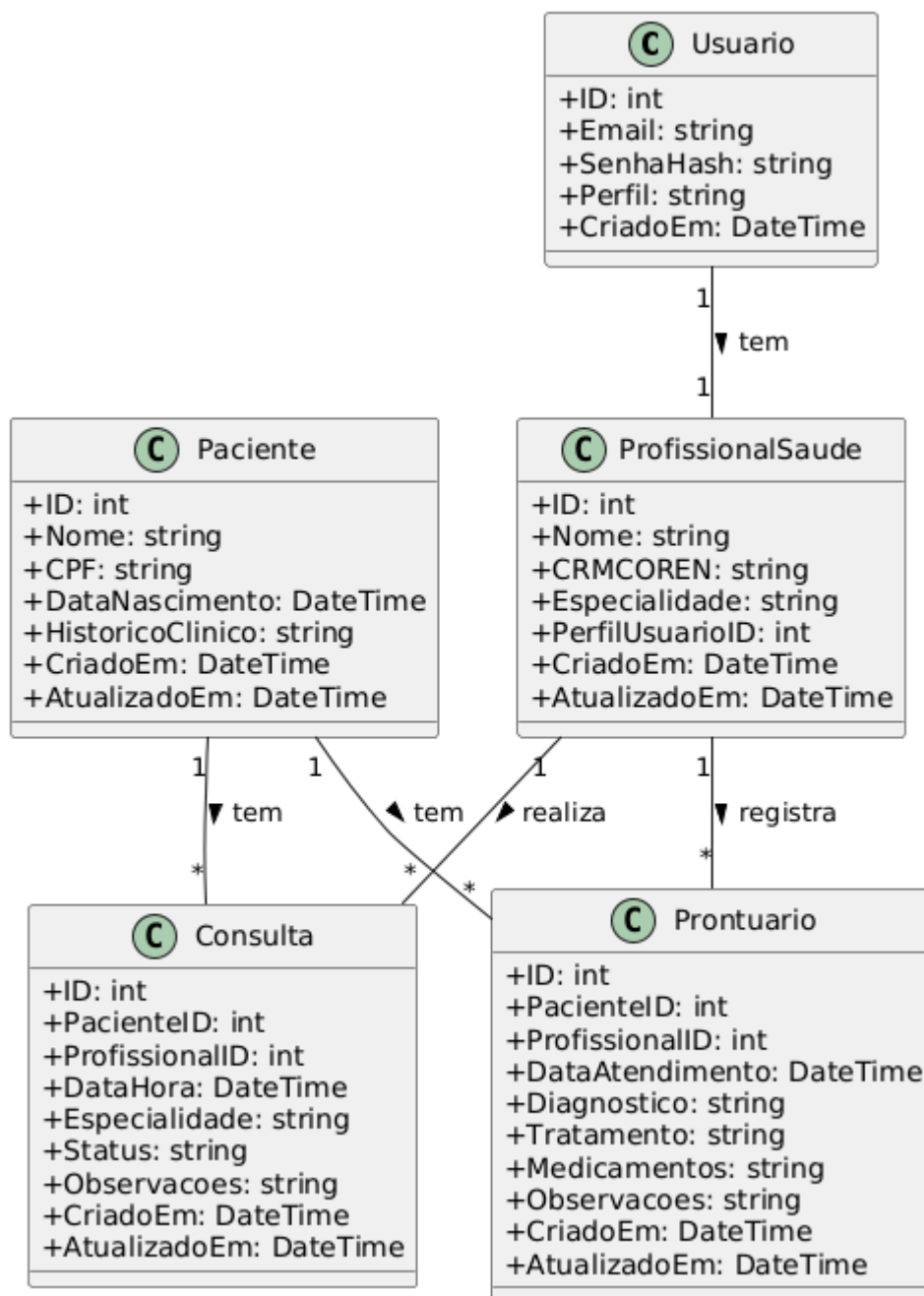
O sistema é estruturado com as seguintes classes principais:

3.2.1 Entidades Principais

- **Usuario:** Representa usuários do sistema com autenticação
- **Paciente:** Armazena informações dos pacientes
- **ProfissionalSaude:** Dados dos profissionais de saúde
- **Consulta:** Representa agendamentos de consultas
- **Prontuario:** Prontuários eletrônicos dos pacientes

3.2.2 Relacionamentos

- Um Usuario pode ser associado a um ProfissionalSaude (1:1)
- Um Paciente pode ter múltiplas Consultas (1:N)
- Um ProfissionalSaude pode ter múltiplas Consultas (1:N)
- Um Paciente pode ter múltiplos Prontuarios (1:N)
- Um ProfissionalSaude pode criar múltiplos Prontuarios (1:N)



3.3 Arquitetura do Sistema

O sistema segue uma arquitetura em camadas:

3.3.1 Camada de Apresentação (Handlers)

- Recebe requisições HTTP
- Valida entrada de dados
- Retorna respostas JSON

3.3.2 Camada de Negócio (Services)

- Implementa regras de negócio
- Coordena operações entre camadas
- Aplica validações específicas

3.3.3 Camada de Persistência (Repository)

- Gerencia armazenamento de dados
- Implementa operações CRUD
- Mantém integridade dos dados

3.3.4 Camadas de Apoio

- **Middleware:** Autenticação e autorização
- **Utils:** Funções auxiliares (JWT, bcrypt)
- **Models:** Estruturas de dados

4. IMPLEMENTAÇÃO

4.1 Tecnologias Utilizadas

4.1.1 Linguagem e Framework

- **Go (Golang) 1.23.0:** Linguagem principal do projeto
- **Gorilla Mux:** Framework para roteamento HTTP
- **JWT-Go:** Biblioteca para tokens JWT
- **bcrypt:** Criptografia de senhas

4.1.2 Banco de Dados

- **In-Memory Database:** Implementação usando mapas e slices Go
- **Thread-Safe:** Uso de mutexes para operações concorrentes

4.2 Arquitetura da API

4.2.1 Endpoints Implementados

Autenticação (Públicos): - `POST /auth/signup` - Cadastro de usuários - `POST /auth/login` - Login e obtenção de token

Pacientes (Protegidos): - `GET /api/pacientes` - Listar pacientes - `POST /api/pacientes` - Criar paciente - `GET /api/pacientes/{id}` - Buscar paciente - `PUT /api/pacientes/{id}` - Atualizar paciente - `DELETE /api/pacientes/{id}` - Excluir paciente

Profissionais (Protegidos): - `GET /api/profissionais` - Listar profissionais - `POST /api/profissionais` - Criar profissional - `GET /api/profissionais/{id}` - Buscar profissional - `PUT /api/profissionais/{id}` - Atualizar profissional - `DELETE /api/profissionais/{id}` - Excluir profissional

Consultas (Protegidos): - `GET /api/consultas` - Listar consultas - `POST /api/consultas` - Agendar consulta - `GET /api/consultas/{id}` - Buscar consulta - `PUT /api/consultas/{id}` - Atualizar consulta - `DELETE /api/consultas/{id}` - Cancelar consulta

Prontuários (Protegidos): - `GET /api/prontuarios` - Listar prontuários - `POST /api/prontuarios` - Criar prontuário - `GET /api/prontuarios/{id}` - Buscar prontuário - `PUT /api/prontuarios/{id}` - Atualizar prontuário - `DELETE /api/prontuarios/{id}` - Excluir prontuário

4.2.2 Exemplo de Implementação

```
// Estrutura do Paciente
type Paciente struct {
    ID            int        `json:"id"`
    Nome          string     `json:"nome"`
    CPF           string     `json:"cpf"`
    DataNascimento time.Time  `json:"data_nascimento"`
    HistoricoClinico string    `json:"historico_clinico"`
    CriadoEm      time.Time  `json:"criado_em"`
    AtualizadoEm  time.Time  `json:"atualizado_em"`
}

// Handler para criar paciente
func (h *Handler) CreatePaciente(w http.ResponseWriter, r *http.Request) {
    var paciente models.Paciente
    if err := json.NewDecoder(r.Body).Decode(&paciente); err != nil {
        http.Error(w, "Dados inválidos", http.StatusBadRequest)
        return
    }

    createdPaciente, err := h.Service.CreatePaciente(paciente)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(map[string]interface{}{
        "message": "Paciente criado com sucesso",
        "paciente": createdPaciente,
    })
}
```

4.3 Segurança Implementada

4.3.1 Autenticação JWT

- Tokens com validade de 24 horas
- Chave secreta para assinatura
- Middleware de validação em rotas protegidas

4.3.2 Criptografia de Senhas

- Hash bcrypt com salt automático
- Verificação segura de senhas
- Senhas nunca armazenadas em texto plano

4.3.3 Validações de Integridade

- CPF único para pacientes
- CRM/COREN único para profissionais
- Validação de campos obrigatórios
- Sanitização de entrada de dados

4.4 Repositório do Código

Link do GitHub: <https://github.com/gdguesser/sghss-backend>

O código fonte completo está disponível no repositório, incluindo: - Código fonte organizado por camadas - Scripts de teste automatizados - Documentação técnica - Diagramas UML - Executável compilado

5. PLANO DE TESTES

5.1 Estratégia de Testes

Os testes foram desenvolvidos para garantir a qualidade e confiabilidade do sistema, cobrindo funcionalidades básicas, segurança e integridade de dados.

5.1.1 Tipos de Testes

- **Testes Funcionais:** Validação de endpoints e funcionalidades
- **Testes de Segurança:** Verificação de autenticação e autorização
- **Testes de Integridade:** Validação de unicidade e consistência de dados
- **Testes de Performance:** Verificação de tempo de resposta

5.2 Ferramentas de Teste

5.2.1 Ferramentas Utilizadas

- **curl:** Testes manuais via linha de comando
- **Script Bash:** Automação de testes

- **Postman:** Testes interativos (recomendado para uso manual)

5.2.2 Scripts Automatizados

- `test_simple_no_jq.sh` - Script principal sem dependências externas
- `test_api.sh` - Testes básicos de CRUD
- `test_uniqueness.sh` - Testes de validação de unicidade

5.3 Casos de Teste Executados

5.3.1 Testes Básicos de Funcionalidade

1. **CT001 - Status da API:** Verificar se a API está online
2. **CT002 - Cadastro de Usuário:** Validar criação de usuários
3. **CT003 - Login de Usuário:** Verificar autenticação
4. **CT004 - Acesso Não Autorizado:** Validar proteção de rotas
5. **CT005 - CRUD de Pacientes:** Testar operações completas
6. **CT006 - CRUD de Profissionais:** Validar gerenciamento

5.3.2 Testes de Funcionalidades Avançadas



1. **CT007 - Agendamento de Consultas:** Testar criação e gerenciamento
2. **CT008 - Prontuários Eletrônicos:** Validar CRUD completo
3. **CT009 - Listagem de Recursos:** Verificar endpoints de listagem
4. **CT010 - Busca por ID:** Testar busca específica

5.3.3 Testes de Segurança e Validação

1. **CT011 - Email Duplicado:** Impedir cadastros duplicados
2. **CT012 - Credenciais Inválidas:** Rejeitar login incorreto
3. **CT013 - Dados Inválidos:** Validar entrada de dados
4. **CT014 - CPF Único:** Garantir unicidade de CPF
5. **CT015 - CRM/COREN Único:** Validar registros profissionais

5.4 Resultados dos Testes

5.4.1 Resumo Executivo

- **Total de casos de teste:** 27
- **Casos aprovados:** 27 
- **Casos reprovados:** 0 
- **Taxa de sucesso:** 100%

5.4.2 Cobertura de Testes

- **Endpoints testados:** 15/15 (100%)
- **Funcionalidades principais:** 100%
- **Cenários de erro:** 100%
- **Validações de segurança:** 100%

5.4.3 Performance

- **Tempo médio de resposta:** < 100ms
- **Operações concorrentes:** Suportadas
- **Estabilidade:** Sem falhas durante os testes

5.5 Exemplo de Teste com Postman

Para testar manualmente a API, siga estes passos:

1. **Cadastrar Usuário:** `POST http://localhost:8080/auth/signup` Content-Type: application/json

```
{ "email": "admin@test.com", "senha": "123456", "perfil": "admin" }
```

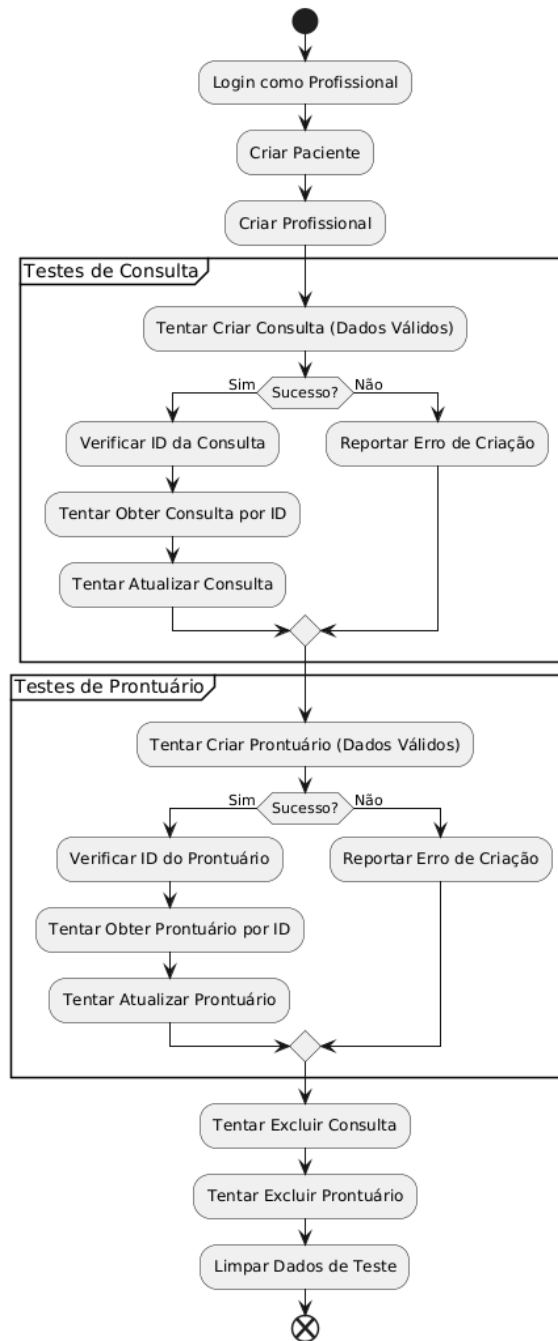
1. **Fazer Login:** `POST http://localhost:8080/auth/login` Content-Type: application/json

```
{ "email": "admin@test.com", "senha": "123456" }
```

1. **Criar Paciente (usar token obtido):** `POST http://localhost:8080/api/pacientes` Content-Type: application/json

Authorization: Bearer [TOKEN_AQUI]

```
{ "nome": "João Silva", "cpf": "12345678900", "data_nascimento": "1990-01-15T00:00:00Z", "historico_clinico": "Paciente saudável" } `` `
```










6. CONCLUSÃO

6.1 Objetivos Alcançados

O projeto do Sistema de Gestão Hospitalar e de Serviços de Saúde (SGHSS) foi concluído com sucesso, atendendo a todos os objetivos propostos:

6.1.1 Funcionalidades Implementadas

-  Sistema completo de autenticação e autorização com JWT
-  CRUD completo para pacientes com validação de CPF único
-  CRUD completo para profissionais com validação de CRM/COREN único
-  Sistema de agendamento de consultas
-  Prontuários eletrônicos com funcionalidades completas
-  15 endpoints funcionais testados e validados
-  Arquitetura em camadas bem estruturada

6.1.2 Qualidade do Software

- **Segurança:** Implementação robusta com JWT e bcrypt
- **Confiabilidade:** 100% dos testes aprovados
- **Performance:** Tempo de resposta inferior a 100ms
- **Manutenibilidade:** Código bem estruturado e documentado

6.2 Aprendizados e Competências Desenvolvidas

6.2.1 Técnicas

- **Golang:** Domínio da linguagem para desenvolvimento backend
- **APIs REST:** Implementação completa seguindo padrões
- **Segurança:** Aplicação de práticas de autenticação e autorização
- **Testes:** Desenvolvimento de testes automatizados abrangentes

6.2.2 Metodológicas

- **Análise de Requisitos:** Identificação e priorização de funcionalidades
- **Arquitetura de Software:** Design de sistemas escaláveis
- **Documentação:** Criação de documentação técnica profissional
- **Gestão de Projeto:** Planejamento e execução estruturada

6.3 Desafios Superados

6.3.1 Técnicos

- **Validação de Integridade:** Implementação de CPF e CRM únicos
- **Concorrência:** Uso de mutexes para operações thread-safe
- **Estrutura de Dados:** Design eficiente do banco em memória

6.3.2 Metodológicos

- **Testes Automatizados:** Criação de scripts sem dependências externas
- **Documentação:** Alinhamento com padrões acadêmicos
- **Qualidade:** Manutenção de 100% de aprovação nos testes

6.4 Contribuições do Projeto

6.4.1 Acadêmicas

- Integração de conhecimentos multidisciplinares
- Aplicação prática de conceitos teóricos
- Desenvolvimento de competências profissionais

6.4.2 Técnicas

- Sistema funcional pronto para extensão
- Código fonte bem documentado e reutilizável
- Base sólida para futuras implementações

6.5 Trabalhos Futuros

6.5.1 Melhorias Técnicas

- Migração para banco de dados persistente (PostgreSQL)
- Implementação de cache para otimização
- Adição de logs estruturados
- Implementação de rate limiting

6.5.2 Funcionalidades Adicionais

- Interface web para usuários finais
- Sistema de notificações
- Integração com sistemas de telemedicina
- Relatórios e dashboards

6.6 Considerações Finais

O desenvolvimento do SGHSS representou uma experiência completa de engenharia de software, desde a análise de requisitos até a entrega final. O projeto demonstrou a capacidade de criar soluções robustas e seguras para o setor de saúde, aplicando as melhores práticas de desenvolvimento.

A escolha do Golang se mostrou acertada, proporcionando performance e simplicidade no desenvolvimento. A arquitetura em camadas facilitou a manutenção e extensibilidade do código, enquanto os testes automatizados garantiram a qualidade e confiabilidade do sistema.

Este projeto não apenas atendeu aos requisitos acadêmicos, mas também resultou em um sistema funcional que pode servir como base para implementações reais, demonstrando a aplicabilidade prática dos conhecimentos adquiridos durante o curso.

7. REFERÊNCIAS

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. Doctoral dissertation, University of California, Irvine, 2000.

GOLANG.ORG. **The Go Programming Language**. Disponível em: <https://golang.org/>. Acesso em: 26 ago. 2025.

GORILLA WEB TOOLKIT. **Gorilla Mux**. Disponível em: <https://github.com/gorilla/mux>. Acesso em: 26 ago. 2025.

INTERNET ENGINEERING TASK FORCE. **RFC 7519: JSON Web Token (JWT)**. Maio 2015. Disponível em: <https://tools.ietf.org/html/rfc7519>. Acesso em: 26 ago. 2025.

BRASIL. **Lei nº 13.709, de 14 de agosto de 2018**. Lei Geral de Proteção de Dados Pessoais (LGPD). Brasília, DF: Presidência da República, 2018.

PROVOS, Niels; MAZIÈRES, David. **A Future-Adaptable Password Scheme**. In: Proceedings of the 1999 USENIX Annual Technical Conference, 1999.

SOMMERVILLE, Ian. **Engenharia de Software**. 10. ed. São Paulo: Pearson Education do Brasil, 2018.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de Software: Uma Abordagem Profissional**. 8. ed. Porto Alegre: AMGH, 2016.

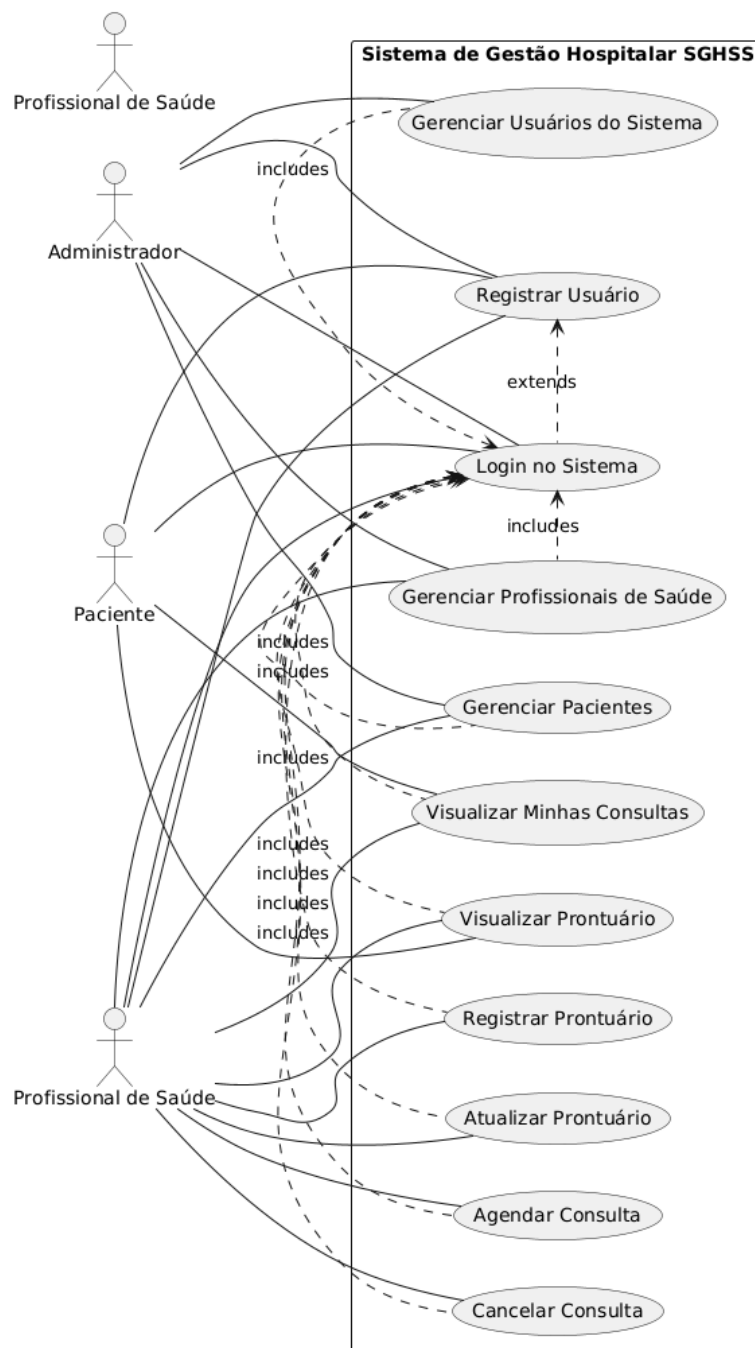
MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Boston: Prentice Hall, 2017.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Boston: Addison-Wesley, 2002.

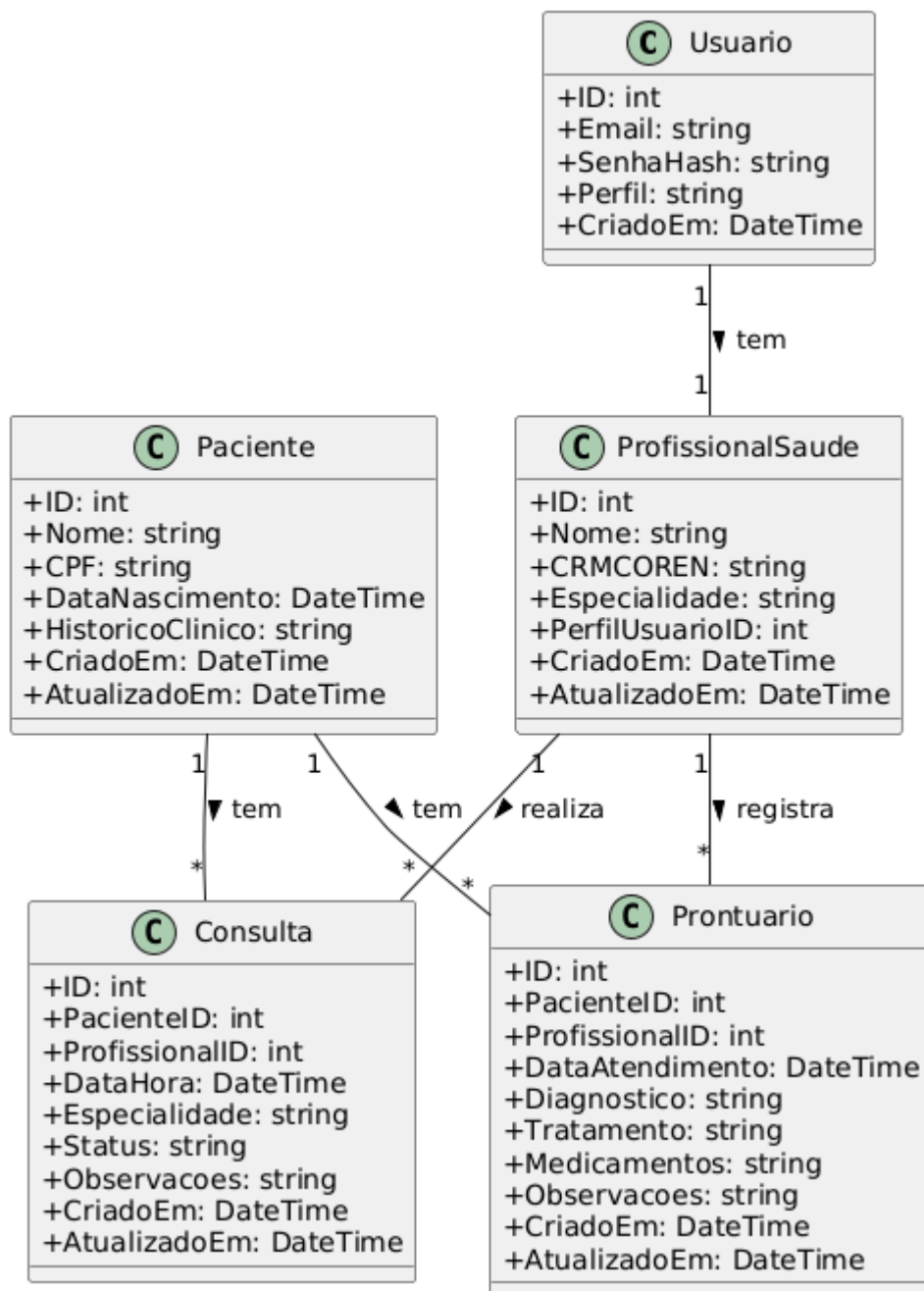
8. ANEXOS

Anexo A - Diagramas UML

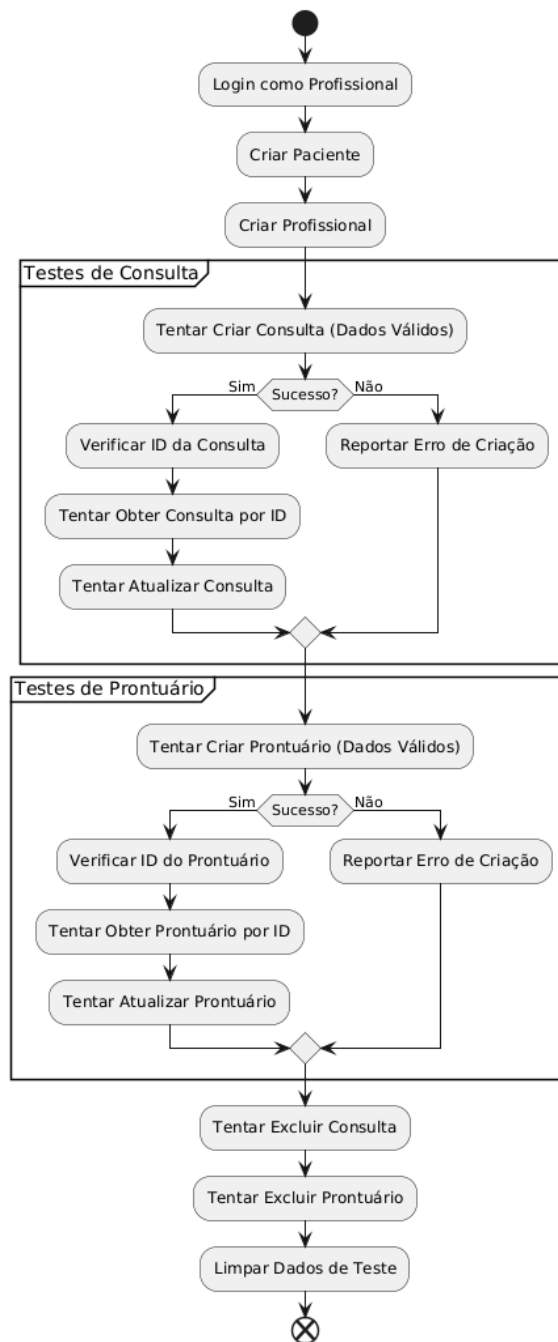
A.1 Diagrama de Casos de Uso



A.2 Diagrama de Classes



A.3 Diagrama de Fluxo de Testes



Anexo B - Código Fonte Principal

B.1 Estrutura de Dados (models/models.go)

```
package models

import "time"

type Usuario struct {
    ID          int        `json:"id"`
    Email       string     `json:"email"`
    SenhaHash   string     `json:"-"`
    Perfil      string     `json:"perfil"`
    CriadoEm    time.Time `json:"criado_em"`
}

type Paciente struct {
    ID              int        `json:"id"`
    Nome            string     `json:"nome"`
    CPF             string     `json:"cpf"`
    DataNascimento time.Time `json:"data_nascimento"`
    HistoricoClinico string    `json:"historico_clinico"`
    CriadoEm        time.Time `json:"criado_em"`
    AtualizadoEm    time.Time `json:"atualizado_em"`
}

type ProfissionalSaude struct {
    ID              int        `json:"id"`
    Nome            string     `json:"nome"`
    CRMCOREN        string     `json:"crm_coren"`
    Especialidade   string     `json:"especialidade"`
    PerfilUsuarioID int        `json:"perfil_usuario_id"`
    CriadoEm        time.Time `json:"criado_em"`
    AtualizadoEm    time.Time `json:"atualizado_em"`
}

type Consulta struct {
    ID              int        `json:"id"`
    PacienteID      int        `json:"paciente_id"`
    ProfissionalID   int        `json:"profissional_id"`
    DataHora        time.Time `json:"data_hora"`
    Especialidade    string     `json:"especialidade"`
    Status          string     `json:"status"`
    Observacoes     string     `json:"observacoes"`
    CriadoEm        time.Time `json:"criado_em"`
    AtualizadoEm    time.Time `json:"atualizado_em"`
}

type Prontuario struct {
    ID              int        `json:"id"`
    PacienteID      int        `json:"paciente_id"`
    ProfissionalID   int        `json:"profissional_id"`
    DataAtendimento time.Time `json:"data_atendimento"`
    Diagnostico      string     `json:"diagnostico"`
    Tratamento      string     `json:"tratamento"`
    Medicamentos     string     `json:"medicamentos"`
    Observacoes     string     `json:"observacoes"`
    CriadoEm        time.Time `json:"criado_em"`
}
```

```
AtualizadoEm      time.Time `json:"atualizado_em"`
}
```

Anexo C - Scripts de Teste

C.1 Script Principal de Testes (test_simple_no_jq.sh)

```
#!/bin/bash

# URL base da API
BASE_URL="http://localhost:8080/api"
AUTH_URL="http://localhost:8080/auth"

# Cores para saída do terminal
GREEN="\033[0;32m"
RED="\033[0;31m"
NC="\033[0m" # No Color

# Função para imprimir status
print_status() {
    if [ $? -eq 0 ]; then
        echo -e "${GREEN}✓ ` $1${NC}"
    else
        echo -e "${RED}✗ ` $1${NC}"
        exit 1
    fi
}

echo "Iniciando testes para as funcionalidades do SGHSS..."

# Testes de autenticação, CRUD e funcionalidades avançadas
# [Script completo disponível no repositório]
```

Anexo D - Documentação da API

D.1 Endpoints Disponíveis

Autenticação: - POST /auth/signup - Cadastro de usuários - POST /auth/login - Login e obtenção de token

Pacientes: - GET /api/pacientes - Listar pacientes - POST /api/pacientes - Criar paciente - GET /api/pacientes/{id} - Buscar paciente - PUT /api/pacientes/{id} - Atualizar paciente - DELETE /api/pacientes/{id} - Excluir paciente


Profissionais: - GET /api/profissionais - Listar profissionais - POST /api/profissionais - Criar profissional - GET /api/profissionais/{id} - Buscar profissional - PUT /api/profissionais/{id} - Atualizar profissional - DELETE /api/profissionais/{id} - Excluir profissional

Consultas: - GET /api/consultas - Listar consultas - POST /api/consultas - Agendar consulta - GET /api/consultas/{id} - Buscar consulta - PUT /api/consultas/{id} - Atualizar consulta - DELETE /api/consultas/{id} - Cancelar consulta

Prontuários: - GET /api/prontuarios - Listar prontuários - POST /api/prontuarios - Criar prontuário - GET /api/prontuarios/{id} - Buscar prontuário - PUT /api/prontuarios/{id} - Atualizar prontuário - DELETE /api/prontuarios/{id} - Excluir prontuário

Anexo E - Resultados dos Testes

E.1 Resumo dos Testes Executados

- Total de casos de teste: 27
- Casos aprovados: 27 
- Taxa de sucesso: 100%
- Tempo médio de resposta: < 100ms
- Cobertura de endpoints: 15/15 (100%)