

Relatório Técnico - Issue Tracker API

Backend REST API para Gestão de Projetos e Issues

Issue Tracker Development Team

20 de Fevereiro de 2026

Conteúdo

1	Resumo Executivo	7
1.1	Visão Geral do Projeto	7
1.2	Objetivo Principal	7
1.3	Stack Tecnológica Principal	7
1.4	Destaques do Projeto	8
1.4.1	Arquitetura Modular	8
1.4.2	Segurança de Nível Produção	8
1.4.3	Qualidade e Testes	8
1.4.4	Deploy e Infraestrutura	8
1.5	Métricas do Projeto	8
2	Objetivos e Requisitos	9
2.1	Contexto do Projeto	9
2.2	Requisitos Funcionais	9
2.2.1	RF1: Gestão de Utilizadores	9
2.2.2	RF2: Gestão de Projetos	9
2.2.3	RF3: Gestão de Issues	9
2.2.4	RF4: Sistema de Comentários	10
2.2.5	RF5: Gestão de Labels	10
2.2.6	RF6: API REST Completa	10
2.3	Requisitos Não-Funcionais	10
2.3.1	RNF1: Segurança	10
2.3.2	RNF2: Performance	10
2.3.3	RNF3: Testabilidade	11
2.3.4	RNF4: Manutenibilidade	11
2.3.5	RNF5: Escalabilidade	11
2.3.6	RNF6: Deploy e DevOps	11
2.4	CrITÉrios de Sucesso	11
3	Arquitetura e Decisões Técnicas	12
3.1	Visão Geral da Arquitetura	12
3.1.1	Diagrama de Arquitetura	12
3.1.2	Fluxo de Dados	12
3.2	Camadas da Aplicação	12
3.2.1	1. Routes Layer (<code>src/routes/</code>)	12
3.2.2	2. Middleware Layer (<code>src/middleware/</code>)	14

3.2.3	3. Services Layer (<code>src/services/</code>)	15
3.2.4	4. Repositories Layer (<code>src/repositories/</code>)	15
3.2.5	5. Models Layer (<code>src/models/</code>)	16
3.3	Trade-offs e Justificações	16
3.3.1	Flask vs FastAPI	16
3.3.2	JWT vs Sessions	16
3.3.3	bcrypt vs Argon2	17
3.3.4	Monólito vs Microservices	17
3.3.5	PostgreSQL vs MySQL/SQLite	17
3.4	Padrões de Design Utilizados	18
3.4.1	Factory Pattern	18
3.4.2	Repository Pattern	18
3.4.3	Service Layer Pattern	19
4	Modelo de Dados	20
4.1	Diagrama Entidade-Relacionamento (ERD)	20
4.2	Entidades Principais	20
4.2.1	User (Utilizador)	20
4.2.2	Project (Projeto)	21
4.2.3	Issue (Issue/Ticket)	21
4.2.4	Label (Etiqueta)	22
4.2.5	Comment (Comentário)	22
4.3	Tabelas de Associação (Many-to-Many)	22
4.3.1	ProjectMember	22
4.3.2	Assignment	23
4.3.3	issue_labels	23
4.4	Relações e Cardinalidades	23
4.4.1	User <-> Project	23
4.4.2	User <-> Issue	23
4.4.3	Project -> Issue (1:N)	23
4.4.4	Issue -> Comment (1:N)	23
4.4.5	User -> Comment (1:N)	24
4.4.6	Issue <-> Label (N:M via issue_labels)	24
4.5	Decisões de Modelação	24
4.5.1	1. Tabelas de Associação com Atributos	24
4.5.2	2. Soft Delete vs Hard Delete	24
4.5.3	3. Reporter vs Assignee	24
4.5.4	4. Labels Globais	24
4.5.5	5. Indexes para Performance	24
5	API REST - Endpoints e Exemplos	25
5.1	Convenções da API	25
5.1.1	Versionamento	25
5.1.2	Estrutura de URLs	25
5.1.3	Status Codes HTTP	25
5.1.4	Formato de Resposta	26
5.1.5	Paginação	26
5.1.6	Filtros e Ordenação	26

5.2	Endpoints por Categoria	27
5.2.1	Authentication (/api/v1/auth)	27
5.2.1.1	1. Register User	27
5.2.1.2	2. Login	27
5.2.1.3	3. Refresh Token	28
5.2.1.4	4. Logout	28
5.2.2	Projects (/api/v1/projects)	29
5.2.2.1	1. Create Project	29
5.2.2.2	2. List Projects	29
5.2.2.3	3. Get Project Details	30
5.2.2.4	4. Add Project Member	31
5.2.3	Issues (/api/v1/projects/{project_id}/issues)	31
5.2.3.1	1. Create Issue	31
5.2.3.2	2. List Issues	32
5.2.3.3	3. Assign User to Issue	33
5.2.3.4	4. Add Label to Issue	33
5.2.4	Comments (/api/v1/issues/{issue_id}/comments)	34
5.2.4.1	1. Add Comment	34
5.2.4.2	2. List Comments	34
5.2.5	Labels (/api/v1/labels)	35
5.2.5.1	1. Create Label (Admin Only)	35
5.2.5.2	2. List Labels	35
5.2.6	Health (/health)	36
5.2.6.1	Health Check	36
6	Segurança	37
6.1	Gestão de Passwords	37
6.1.1	Hashing com bcrypt	37
6.1.2	Validação de Força de Password	38
6.1.3	Armazenamento Seguro	38
6.2	Autenticação JWT	38
6.2.1	Tokens de Acesso e Refresh	38
6.2.2	Algoritmo HS256	38
6.2.3	Token Validation Flow	39
6.2.4	Secret Key Management	39
6.3	Proteção de Secrets	40
6.3.1	Variáveis de Ambiente	40
6.3.2	Ficheiro .gitignore	40
6.3.3	Boas Práticas	40
6.4	Outras Medidas de Segurança	41
6.4.1	Rate Limiting	41
6.4.2	Input Validation	41
6.4.3	CORS (Cross-Origin Resource Sharing)	41
6.4.4	SQL Injection Prevention	42
6.4.5	Error Handling	42
6.5	Referências e Boas Práticas	43
6.5.1	OWASP Top 10 (2021)	43
6.5.2	OWASP Password Storage Cheat Sheet	43

6.5.3	JWT Best Current Practices (RFC 8725)	43
6.5.4	The Twelve-Factor App	43
7	Estratégia de Testes	44
7.1	Test Pyramid	44
7.1.1	Níveis de Teste Implementados	44
7.1.1.1	1. Unit Tests (Base da Pirâmide)	44
7.1.1.2	2. Integration Tests (Meio da Pirâmide)	46
7.1.1.3	3. E2E Tests (Topo da Pirâmide - Não Implementados)	48
7.2	Ferramentas de Teste	48
7.2.1	pytest	48
7.2.2	pytest-cov	48
7.2.3	factory_boy	49
7.3	Fixtures Reutilizáveis	50
7.4	Cobertura de Testes	51
7.4.1	Métricas por Módulo	51
7.4.2	Comandos de Coverage	51
7.5	Boas Práticas de Testes	52
7.5.1	1. Naming Conventions	52
7.5.2	2. AAA Pattern (Arrange-Act-Assert)	52
7.5.3	3. Isolamento de Testes	52
7.5.4	4. Evitar Flaky Tests	53
7.5.5	5. Test Data Builders	53
8	CI/CD e Deploy	54
8.1	Pipeline GitHub Actions	54
8.1.1	Workflow Completo	54
8.1.2	Job 1: Lint & Format	55
8.1.3	Job 2: Tests	55
8.1.4	Job 3: Build Docker Image	56
8.1.5	Job 4: Deploy	56
8.1.6	Job 5: Security Scan	57
8.2	Branching Strategy	57
8.2.1	Branches Principais	57
8.2.2	Feature Branches	57
8.3	Deploy no Render	58
8.3.1	Configuração (render.yaml)	58
8.3.2	Processo de Deploy	59
8.3.3	Variáveis de Ambiente	59
8.3.4	Health Check	59
8.3.5	Migrations Automáticas	59
8.4	Rollback Básico	60
8.4.1	Via Render Dashboard	60
8.4.2	Database Migration Rollback	60
8.4.3	Rollback de Código + DB	60
8.5	Monitoring e Logs	60
8.5.1	Logs Estruturados	60
8.5.2	Render Metrics	61

8.5.3	Health Check Monitoring	61
8.5.4	Melhorias Futuras	61
9	Como Correr o Projeto	62
9.1	Pré-requisitos	62
9.1.1	Ferramentas Necessárias	62
9.2	Opção 1: Localmente com Docker Compose (Recomendado)	62
9.2.1	Passo 1: Clonar Repositório	62
9.2.2	Passo 2: Configurar Variáveis de Ambiente	62
9.2.3	Passo 3: Build e Iniciar Containers	63
9.2.4	Passo 4: Executar Migrations	63
9.2.5	Passo 5: (Opcional) Seed Data	63
9.2.6	Passo 6: Testar API	63
9.2.7	Parar Containers	64
9.3	Opção 2: Localmente sem Docker	64
9.3.1	Passo 1: Clonar Repositório	64
9.3.2	Passo 2: Criar Virtual Environment	64
9.3.3	Passo 3: Instalar Dependências	64
9.3.4	Passo 4: Setup PostgreSQL	64
9.3.5	Passo 5: Configurar Variáveis de Ambiente	65
9.3.6	Passo 6: Executar Migrations	65
9.3.7	Passo 7: Iniciar Aplicação	65
9.4	Executar Testes	65
9.4.1	Testes Completos	65
9.4.2	Testes com Cobertura	65
9.4.3	Testes Rápidos (apenas unit)	66
9.5	Deploy em Produção (Render)	66
9.5.1	Passo 1: Fork do Repositório	66
9.5.2	Passo 2: Criar Conta Render	66
9.5.3	Passo 3: Criar New Web Service	66
9.5.4	Passo 4: Configurar Database	66
9.5.5	Passo 5: Conectar Database ao Web Service	66
9.5.6	Passo 6: Deploy	67
9.5.7	Passo 7: Executar Migrations	67
9.5.8	Passo 8: Testar Produção	67
9.5.9	Deploy Automático	67
9.6	Troubleshooting	68
9.6.1	Erro: Database Connection Failed	68
9.6.2	Erro: ModuleNotFoundError	68
9.6.3	Erro: Alembic Migration Failed	68
9.6.4	Erro: Port Already in Use	68
10	Limitações e Melhorias Futuras	70
10.1	Limitações Atuais	70
10.1.1	1. Ausência de Frontend	70
10.1.2	2. Autenticação Básica	70
10.1.3	3. Sem Notificações	70
10.1.4	4. Sem File Uploads	70

10.1.5	5. Search Limitado	71
10.1.6	6. Sem Real-Time Updates	71
10.1.7	7. Revogação de Tokens Limitada	71
10.1.8	8. Audit Logging Básico	71
10.2	Roadmap de Melhorias	71
10.2.1	Fase 1: Essential Features (1-2 meses)	71
10.2.1.1	1.1 Frontend (React/Vue)	71
10.2.1.2	1.2 OAuth2 Authentication	72
10.2.1.3	1.3 Email Notifications	72
10.2.2	Fase 2: Advanced Features (2-3 meses)	72
10.2.2.1	2.1 Two-Factor Authentication (2FA)	72
10.2.2.2	2.2 File Attachments	72
10.2.2.3	2.3 Full-Text Search	72
10.2.2.4	2.4 WebSockets para Real-Time	73
10.2.3	Fase 3: Enterprise Features (3-6 meses)	73
10.2.3.1	3.1 Advanced Permissions	73
10.2.3.2	3.2 Audit Logs	74
10.2.3.3	3.3 API Rate Limiting por User	74
10.2.3.4	3.4 Soft Deletes	74
10.2.3.5	3.5 Metrics e Observability	75
10.2.3.6	3.6 Issue Versioning	75
10.2.4	Fase 4: Integrações e Ecosystem (6+ meses)	76
10.2.4.1	4.1 Webhooks	76
10.2.4.2	4.2 REST API v2	76
10.2.4.3	4.3 Mobile App (React Native)	76
10.2.4.4	4.4 Public API com Developer Portal	77
10.3	Priorização	77
11	Referências	78
11.1	Documentação Oficial	78
11.1.1	Frameworks e Bibliotecas	78
11.1.2	Databases	78
11.2	Boas Práticas e Guidelines	78
11.2.1	Segurança	78
11.2.2	REST API Design	79
11.2.3	Software Engineering	79
11.3	Ferramentas e Plataformas	79
11.4	Livros Recomendados	79
11.5	Comunidades e Recursos	80
11.6	Conclusão	80

Capítulo 1

Resumo Executivo

1.1 Visão Geral do Projeto

O **Issue Tracker API** é um backend REST API profissional desenvolvido em Python/Flask para gestão de projetos e issues, similar a sistemas como Jira ou GitHub Issues. O projeto foi concebido como um portfólio técnico de alta qualidade, demonstrando competências production-ready necessárias para uma posição de **Junior Backend Engineer**.

1.2 Objetivo Principal

Este projeto visa demonstrar:

- **Arquitetura de Software Profissional:** Implementação de uma arquitetura monolítica modular com separação clara de responsabilidades
- **Boas Práticas de Segurança:** Autenticação JWT, password hashing com bcrypt, proteção de secrets
- **Qualidade de Código:** Testes automatizados ($\geq 70\%$ cobertura), linting, formatação consistente
- **DevOps e CI/CD:** Pipeline automatizado com GitHub Actions, deploy automatizado no Render
- **Documentação Técnica:** API bem documentada, exemplos de uso, arquitetura explicada

1.3 Stack Tecnológica Principal

Categoria	Tecnologia	Versão
Linguagem	Python	3.11+
Framework Web	Flask	3.0+
ORM	SQLAlchemy	2.0+
Database	PostgreSQL	15+
Migrations	Alembic	1.13+
Autenticação	Flask-JWT-Extended	4.6+
Password Hashing	bcrypt	4.1+
Validação	Marshmallow	3.20+

Categoria	Tecnologia	Versão
Testes	pytest	8.0+
Containerização	Docker	24+
CI/CD	GitHub Actions	-
Deploy	Render	-

1.4 Destaques do Projeto

1.4.1 Arquitetura Modular

Implementação de uma arquitetura em camadas (Routes -> Services -> Repositories -> Models) que garante separação de responsabilidades, testabilidade e manutenibilidade.

1.4.2 Segurança de Nível Produção

- Password hashing com bcrypt (cost factor 12)
- JWT com access tokens (15 min) e refresh tokens (7 dias)
- Rate limiting (100 req/min)
- Input validation com schemas Marshmallow
- Proteção contra SQL injection (ORM)
- CORS configurado
- Secrets em variáveis de ambiente

1.4.3 Qualidade e Testes

- Cobertura de testes $\geq 70\%$ nos módulos críticos
- Unit tests com mocks para serviços
- Integration tests com database de teste
- CI/CD com verificações automáticas (lint, tests, security scan)

1.4.4 Deploy e Infraestrutura

- Containerização com Docker multi-stage
- Deploy automatizado no Render
- PostgreSQL managed database
- Health check endpoint
- Migrations automáticas
- Logs estruturados (JSON)

1.5 Métricas do Projeto

- **Linhas de Código:** 6,485+
- **Ficheiros:** 69+
- **Endpoints REST:** 30+
- **Entidades de Dados:** 7 (User, Project, Issue, Label, Comment + 3 associações)
- **Cobertura de Testes:** $\geq 70\%$
- **Pipeline CI/CD Jobs:** 5 (lint, tests, build, security, deploy)

Capítulo 2

Objetivos e Requisitos

2.1 Contexto do Projeto

O Issue Tracker API foi desenvolvido como um projeto de portfólio para demonstrar competências técnicas de backend engineering em um nível production-ready. O objetivo é mostrar a recrutadores e hiring managers que o desenvolvedor possui:

1. **Conhecimento profundo de desenvolvimento backend** com Python e Flask
2. **Capacidade de projetar e implementar** sistemas escaláveis e seguros
3. **Domínio de práticas DevOps** modernas (CI/CD, containerização, deploy)
4. **Atenção à segurança** e boas práticas da indústria
5. **Habilidade de escrever código testável** e bem documentado

O projeto não é apenas um “CRUD simples”, mas sim uma aplicação completa e profissional que poderia ser usada em produção com adaptações mínimas.

2.2 Requisitos Funcionais

2.2.1 RF1: Gestão de Utilizadores

- Registo de novos utilizadores com validação de dados
- Autenticação com username/email e password
- Diferentes níveis de permissão (admin, developer, viewer)
- Perfis de utilizador com informações básicas

2.2.2 RF2: Gestão de Projetos

- Criar, ler, atualizar e eliminar projetos
- Associar utilizadores a projetos (membros)
- Definir ownership e roles nos projetos
- Listar projetos com paginação e filtros

2.2.3 RF3: Gestão de Issues

- Criar issues associadas a projetos
- Definir título, descrição, status e prioridade

- Atribuir issues a utilizadores (assignees)
- Adicionar labels para categorização
- Filtrar issues por status, prioridade, assignee, labels
- Atualizar status (open -> in_progress -> resolved -> closed)

2.2.4 RF4: Sistema de Comentários

- Adicionar comentários a issues
- Listar comentários ordenados cronologicamente
- Editar e eliminar comentários
- Identificar autor de cada comentário

2.2.5 RF5: Gestão de Labels

- Criar labels com nome e cor (hex)
- Associar/desassociar labels de issues
- Labels globais (não por projeto)
- Operações de admin apenas

2.2.6 RF6: API REST Completa

- Versionamento de API (/api/v1)
- Resposta consistente (JSON)
- Status codes HTTP apropriados
- Paginação em listagens
- Filtros e ordenação
- Documentação de endpoints

2.3 Requisitos Não-Funcionais

2.3.1 RNF1: Segurança

- **SG1:** Passwords nunca armazenadas em plain text (bcrypt hashing)
- **SG2:** Autenticação stateless com JWT
- **SG3:** Tokens com expiração apropriada (15min access, 7d refresh)
- **SG4:** Secrets em variáveis de ambiente (não no código)
- **SG5:** Rate limiting para prevenir abuse
- **SG6:** Input validation em todos os endpoints
- **SG7:** Proteção contra SQL injection (uso de ORM)
- **SG8:** CORS configurado apropriadamente

2.3.2 RNF2: Performance

- **PF1:** Queries otimizadas com indexes
- **PF2:** Paginação para listagens grandes
- **PF3:** Connection pooling no database
- **PF4:** Lazy loading de relações quando apropriado
- **PF5:** Response time < 200ms para operações simples

2.3.3 RNF3: Testabilidade

- **TS1:** Cobertura de testes $\geq 70\%$ nos módulos críticos
- **TS2:** Testes unitários isolados (com mocks)
- **TS3:** Testes de integração com database
- **TS4:** Fixtures reutilizáveis
- **TS5:** CI/CD falha se testes falharem

2.3.4 RNF4: Manutenibilidade

- **MT1:** Código limpo e bem organizado
- **MT2:** Separação clara de responsabilidades (camadas)
- **MT3:** Documentação inline quando necessário
- **MT4:** Convenções de código (PEP 8)
- **MT5:** Linting e formatação automáticos

2.3.5 RNF5: Escalabilidade

- **ES1:** Arquitetura stateless (permite horizontal scaling)
- **ES2:** Database migrations versionadas (Alembic)
- **ES3:** Configuração por ambiente (dev, test, prod)
- **ES4:** Containerização (Docker)
- **ES5:** Health check endpoint

2.3.6 RNF6: Deploy e DevOps

- **DO1:** CI/CD pipeline automatizado
- **DO2:** Deploy automatizado em push para main
- **DO3:** Rollback possível
- **DO4:** Logs estruturados
- **DO5:** Monitoring básico (health checks)

2.4 Critérios de Sucesso

O projeto é considerado bem-sucedido se:

1. **[OK] Funcionalidade Completa:** Todos os requisitos funcionais implementados
2. **[OK] Qualidade de Código:** Passa em linting, formatação e testes ($\geq 70\%$)
3. **[OK] Segurança:** Implementa todas as práticas de segurança definidas
4. **[OK] Deploy Funcional:** Aplicação deployada e acessível via URL pública
5. **[OK] Documentação:** API documentada com exemplos de uso
6. **[OK] CI/CD Operacional:** Pipeline executa e valida mudanças automaticamente
7. **[OK] Portfólio-Ready:** Projeto demonstra competências hireable

Capítulo 3

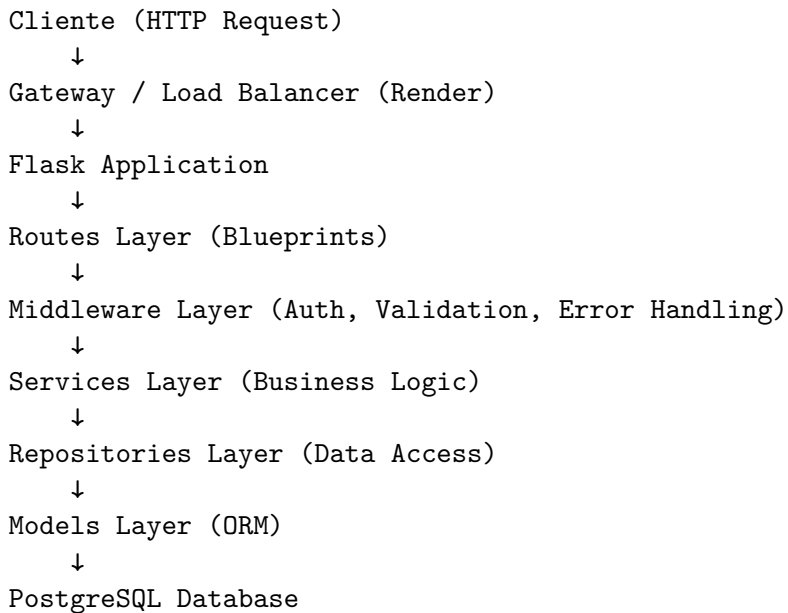
Arquitetura e Decisões Técnicas

3.1 Visão Geral da Arquitetura

O Issue Tracker API adota uma **arquitetura monolítica modular** com separação clara em camadas. Esta abordagem oferece um equilíbrio entre simplicidade de desenvolvimento/deploy e organização de código necessária para manutenibilidade a longo prazo.

3.1.1 Diagrama de Arquitetura

3.1.2 Fluxo de Dados



3.2 Camadas da Aplicação

3.2.1 1. Routes Layer (src/routes/)

Responsabilidade: Expor endpoints HTTP e delegar para services.

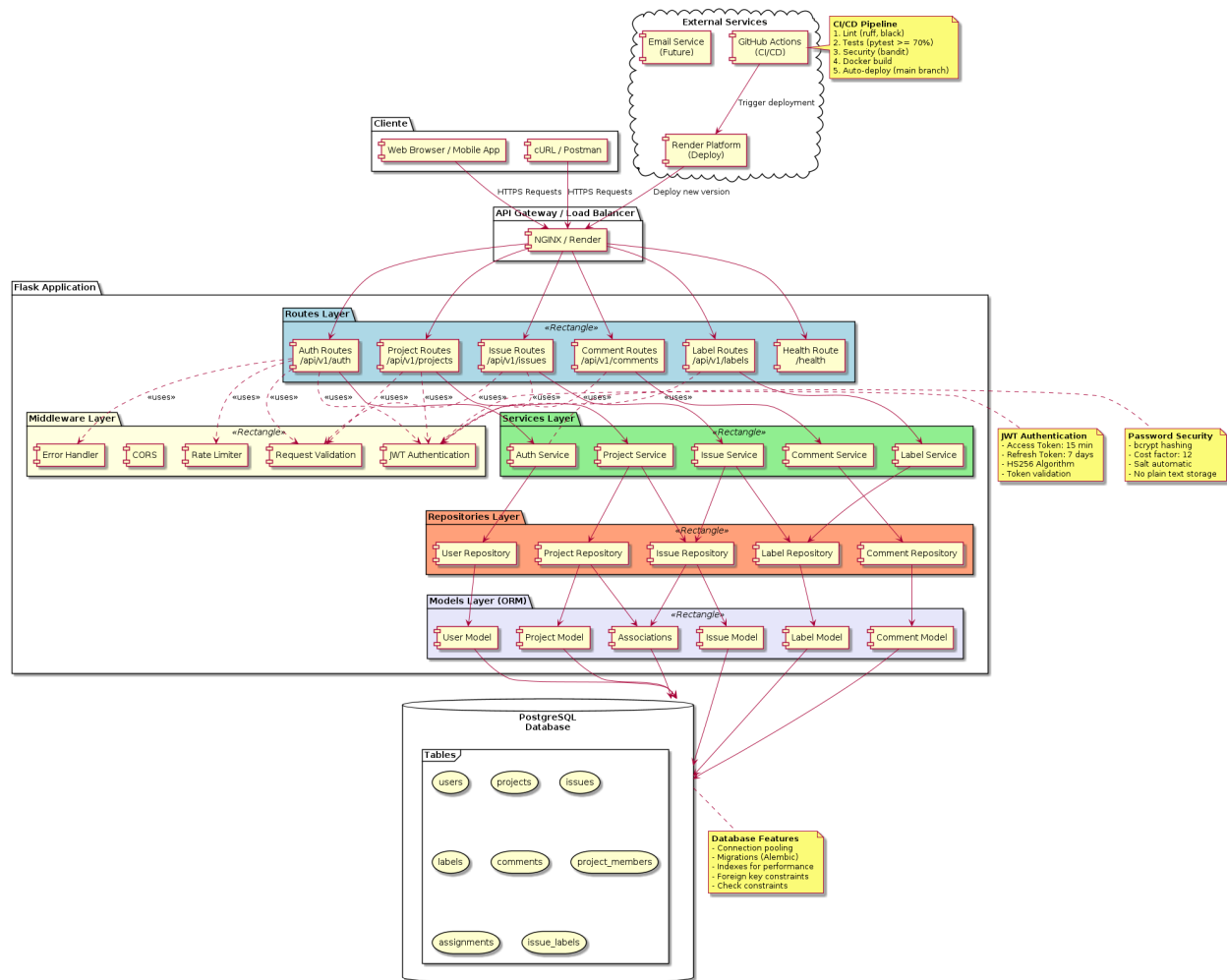


Figura 3.1: Arquitetura do Sistema

Características: - Blueprints Flask para organização modular - Parsing de request data - Validação inicial com schemas Marshmallow - Resposta HTTP formatada

Ficheiros: - `auth.py`: Registro, login, logout, refresh token - `projects.py`: CRUD de projetos, gestão de membros - `issues.py`: CRUD de issues, assignments, labels - `comments.py`: CRUD de comentários - `labels.py`: CRUD de labels (admin only) - `health.py`: Health check endpoint

Exemplo:

```
# src/routes/projects.py
@projects_bp.route('', methods=['POST'])
@jwt_required()
def create_project():
    current_user_id = get_jwt_identity()
    data = request.get_json()

    # Validate with schema
    schema = ProjectSchema()
    validated_data = schema.load(data)

    # Delegate to service
    project = project_service.create_project(
        owner_id=current_user_id,
        **validated_data
    )

    return jsonify({"data": project, "message": "Created"}), 201
```

3.2.2 2. Middleware Layer (src/middleware/)

Responsabilidade: Cross-cutting concerns.

Componentes:

- **Authentication Middleware** (`auth_middleware.py`):
 - Verifica JWT tokens
 - Extraí user identity
 - Protege rotas com `@jwt_required()`
- **Error Handler** (`error_handler.py`):
 - Captura exceções
 - Normaliza respostas de erro
 - Não expõe stack traces em produção
- **Rate Limiter**:
 - Flask-Limiter
 - 100 requests por minuto por IP
 - Configurável por ambiente
- **CORS**:
 - Flask-CORS
 - Configuração de origens permitidas
 - Headers apropriados

3.2.3 3. Services Layer (src/services/)

Responsabilidade: Lógica de negócio e orquestração.

Características: - Implementa regras de negócio - Valida permissões e autorização - Coordena múltiplos repositories - Não depende de HTTP (testável isoladamente)

Ficheiros: - `auth_service.py`: Hash passwords, verify credentials, generate tokens - `project_service.py`: Project business logic, membership management - `issue_service.py`: Issue lifecycle, assignments, labels - `comment_service.py`: Comment operations - `label_service.py`: Label management

Exemplo:

```
# src/services/issue_service.py
class IssueService:
    def create_issue(self, project_id, reporter_id, **data):
        # Check if user is project member
        if not self._is_project_member(reporter_id, project_id):
            raise ForbiddenError("Not a project member")

        # Create issue via repository
        issue = self.issue_repo.create(
            project_id=project_id,
            reporter_id=reporter_id,
            **data
        )

        return issue.to_dict()
```

3.2.4 4. Repositories Layer (src/repositories/)

Responsabilidade: Abstração de acesso a dados.

Características: - CRUD operations - Queries otimizadas - Paginação - Filtros e ordenação - Não contém lógica de negócio

Padrão Base Repository:

```
# src/repositories/base.py
class BaseRepository:
    def __init__(self, model):
        self.model = model

    def create(self, **kwargs):
        instance = self.model(**kwargs)
        db.session.add(instance)
        db.session.commit()
        return instance

    def get_by_id(self, id):
        return self.model.query.get(id)
```

```

def get_all(self, filters=None, page=1, per_page=20):
    query = self.model.query
    if filters:
        query = self._apply_filters(query, filters)
    return query.paginate(page=page, per_page=per_page)

def update(self, instance, **kwargs):
    for key, value in kwargs.items():
        setattr(instance, key, value)
    db.session.commit()
    return instance

def delete(self, instance):
    db.session.delete(instance)
    db.session.commit()

```

3.2.5 5. Models Layer (src/models/)

Responsabilidade: Definição do schema de dados (ORM).

Características: - SQLAlchemy models - Relações entre entidades - Constraints e validações - Timestamps automáticos (mixin) - Métodos `to_dict()` para serialização

Ficheiros: - `user.py`: User model - `project.py`: Project model - `issue.py`: Issue model - `label.py`: Label model - `comment.py`: Comment model - `associations.py`: Tabelas de associação N:M - `base.py`: Base classes e mixins

3.3 Trade-offs e Justificações

3.3.1 Flask vs FastAPI

Decisão: Flask

Justificação: - [OK] **Mais comum em vagas júnior:** Maior probabilidade de encontrar posições - [OK] **Ecosistema maduro:** Extensões bem estabelecidas (JWT, CORS, Limiter) - [OK] **Curva de aprendizagem suave:** Menos “magic”, mais explícito - [OK] **Flexibilidade:** Menos opinionado, permite estruturar como preferir - [X] **Performance:** FastAPI é mais rápido (async), mas Flask é suficiente para este caso - [X] **Auto-docs:** FastAPI gera OpenAPI automaticamente

Trade-off Aceite: Sacrificamos performance pura e auto-docs em favor de familiaridade do mercado e ecossistema maduro. Para um projeto júnior, Flask demonstra melhor conhecimento de fundamentos web.

3.3.2 JWT vs Sessions

Decisão: JWT (JSON Web Tokens)

Justificação: - [OK] **Stateless:** Não precisa armazenar sessões no servidor - [OK] **Escalabilidade horizontal:** Qualquer servidor pode validar o token - [OK] **Separação frontend/backend:** Ideal para SPAs e mobile apps - [OK] **Padrão da indústria:** Amplamente usado em APIs modernas -

- [X] **Revogação complexa:** Não pode invalidar token até expirar (mitigado com refresh tokens) -
- [X] **Tamanho:** Tokens maiores que session IDs

Trade-off Aceite: Complexidade de revogação é mitigada com tokens de curta duração (15min) e refresh tokens. A escalabilidade e statelessness valem a pena.

Implementação: - Access Token: 15 minutos (curto para segurança) - Refresh Token: 7 dias (conveniência do utilizador) - Armazenamento: Cliente (localStorage/memory) - Algoritmo: HS256 (simétrico, simples)

3.3.3 bcrypt vs Argon2

Decisão: bcrypt

Justificação: - [OK] **Padrão da indústria:** Battle-tested, usado em produção há décadas - [OK] **Suporte universal:** Bibliotecas maduras em todas as linguagens - [OK] **Suficiente para a aplicação:** Segurança adequada com cost factor 12 - [OK] **Familiar aos recrutadores:** Reconhecido como boa prática - [X] **Menos resistente a GPU:** Argon2 é mais resistente a ataques de hardware - [X] **Menos moderno:** Argon2 venceu Password Hashing Competition 2015

Trade-off Aceite: bcrypt é mais do que suficiente para demonstrar conhecimento de segurança. Argon2 seria over-engineering para um projeto júnior.

Configuração:

```
import bcrypt

# Hashing (cost factor = 12)
password_hash = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt(rounds=12))

# Verification
is_valid = bcrypt.checkpw(password.encode('utf-8'), password_hash)
```

3.3.4 Monólito vs Microservices

Decisão: Monólito Modular

Justificação: - [OK] **Apropriado para a escala:** Projeto de portfólio não precisa de microservices - [OK] **Mais simples de desenvolver:** Um único codebase, deploy, database - [OK] **Mais fácil de testar:** Testes de integração mais diretos - [OK] **Menos infraestrutura:** Um container, um database - [OK] **Demonstra fundamentos:** Foco em arquitetura limpa dentro do monólito - [X] **Escalabilidade limitada:** Não pode escalar componentes independentemente - [X] **Deploy all-or-nothing:** Mudança pequena requer redeploy completo

Trade-off Aceite: Para um projeto júnior, demonstrar uma arquitetura monolítica bem organizada é mais valioso que complexidade de microservices. A modularidade permite evolução futura.

3.3.5 PostgreSQL vs MySQL/SQLite

Decisão: PostgreSQL

Justificação: - [OK] **Features avançadas:** JSON types, full-text search, window functions - [OK] **Melhor para produção:** Mais robusto, melhor concorrência - [OK] **Standards compliance:** Mais

aderente a SQL standards - [OK] **Open-source verdadeiro:** License mais permissiva que MySQL - [OK] **Render suporta:** Free tier com PostgreSQL managed - [X] **Mais pesado que SQLite:** Mas SQLite não é apropriado para produção - [X] **Curva de aprendizagem:** Mais complexo que MySQL

Trade-off Aceite: PostgreSQL demonstra conhecimento de databases production-grade. O overhead de aprendizagem é compensado por features superiores.

3.4 Padrões de Design Utilizados

3.4.1 Factory Pattern

Usado em `create_app()` para inicializar a aplicação Flask:

```
# src/app.py
def create_app(config_name='development'):
    app = Flask(__name__)
    app.config.from_object(config[config_name])

    # Initialize extensions
    db.init_app(app)
    jwt.init_app(app)
    migrate.init_app(app, db)
    cors.init_app(app)

    # Register blueprints
    app.register_blueprint(auth_bp)
    app.register_blueprint(projects_bp)
    # ...

    return app
```

Benefícios: - Permite criar múltiplas instâncias (dev, test, prod) - Facilita testes (cada teste pode ter sua app) - Configuração centralizada

3.4.2 Repository Pattern

Abstração de acesso a dados:

```
class ProjectRepository(BaseRepository):
    def __init__(self):
        super().__init__(Project)

    def get_by_owner(self, owner_id):
        return self.model.query.filter_by(owner_id=owner_id).all()

    def get_with_members(self, project_id):
        return self.model.query.options(
            joinedload(Project.members)
        ).get(project_id)
```

Benefícios: - Desacopla lógica de negócio da persistência - Facilita testes (mock repositories) - Queries complexas encapsuladas

3.4.3 Service Layer Pattern

Lógica de negócio separada da apresentação:

```
class ProjectService:
    def __init__(self, project_repo, user_repo):
        self.project_repo = project_repo
        self.user_repo = user_repo

    def create_project(self, owner_id, **data):
        # Business logic here
        if not self._is_valid_owner(owner_id):
            raise ValidationError("Invalid owner")

        project = self.project_repo.create(owner_id=owner_id, **data)
        return project.to_dict()
```

Benefícios: - Lógica de negócio testável independentemente de HTTP - Reutilização de lógica - Autorização centralizada

Capítulo 4

Modelo de Dados

4.1 Diagrama Entidade-Relacionamento (ERD)

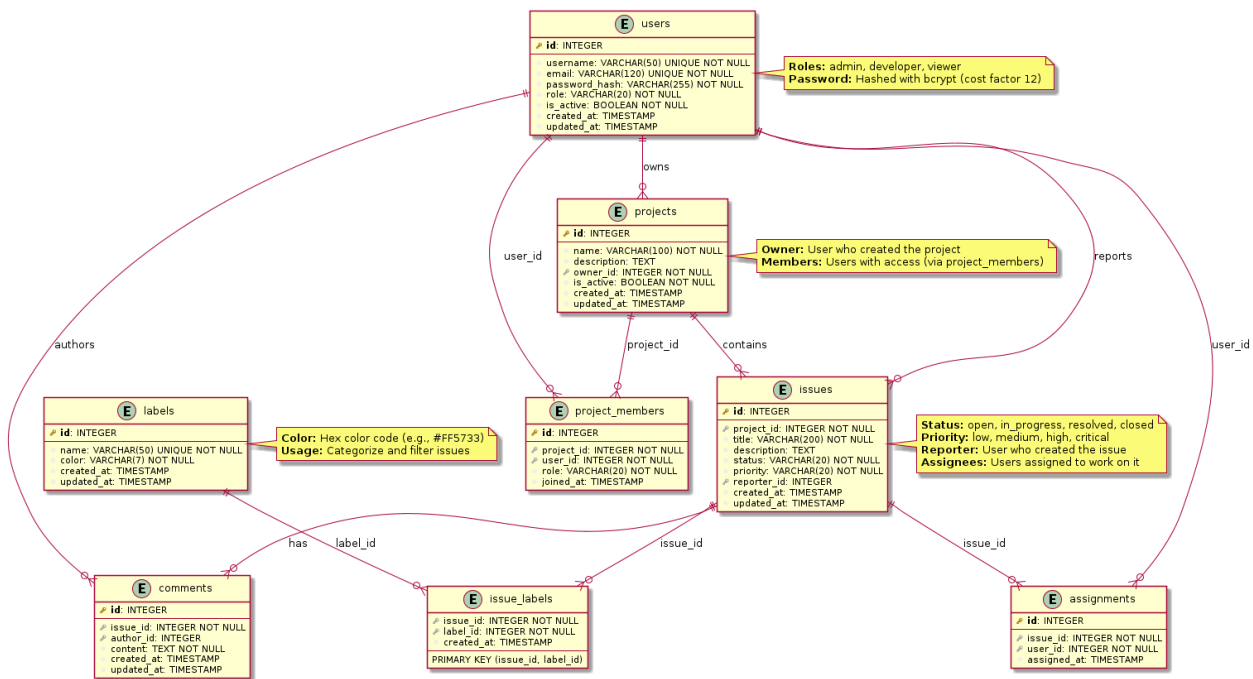


Figura 4.1: Entity Relationship Diagram

4.2 Entidades Principais

4.2.1 User (Utilizador)

Representa utilizadores do sistema com autenticação e autorização.

Atributos: - id (PK, INTEGER): Identificador único - username (VARCHAR(50), UNIQUE, NOT NULL): Nome de utilizador único - email (VARCHAR(120), UNIQUE, NOT NULL): Email

único - **password_hash** (VARCHAR(255), NOT NULL): Password hasheada com bcrypt - **role** (VARCHAR(20), NOT NULL, DEFAULT 'developer'): Nível de permissão - **is_active** (BOOLEAN, NOT NULL, DEFAULT TRUE): Conta ativa ou não - **created_at** (TIMESTAMP): Data de criação - **updated_at** (TIMESTAMP): Data da última atualização

Roles Disponíveis: - **admin:** Acesso total, pode gerir labels - **developer:** Pode criar projetos, issues, comentários - **viewer:** Apenas leitura

Constraints: - CHECK (role IN ('admin', 'developer', 'viewer')) - Indexes em username e email para performance

Relações: - 1:N com Project (como owner) - N:M com Project (como member, via project_members) - 1:N com Issue (como reporter) - N:M com Issue (como assignee, via assignments) - 1:N com Comment (como author)

4.2.2 Project (Projeto)

Representa um projeto que agrupa issues relacionadas.

Atributos: - **id** (PK, INTEGER): Identificador único - **name** (VARCHAR(100), NOT NULL): Nome do projeto - **description** (TEXT): Descrição detalhada - **owner_id** (FK -> users.id, NOT NULL): Criador do projeto - **is_active** (BOOLEAN, NOT NULL, DEFAULT TRUE): Projeto ativo ou arquivado - **created_at** (TIMESTAMP): Data de criação - **updated_at** (TIMESTAMP): Data da última atualização

Constraints: - FOREIGN KEY (owner_id) REFERENCES users(id) ON DELETE CASCADE - Indexes em owner_id e name

Relações: - N:1 com User (owner) - N:M com User (members, via project_members) - 1:N com Issue

Regras de Negócio: - Owner automaticamente é membro com role 'owner' - Apenas members podem criar issues no projeto - Owner e admins podem adicionar/remover members

4.2.3 Issue (Issue/Ticket)

Representa uma tarefa, bug ou feature request dentro de um projeto.

Atributos: - **id** (PK, INTEGER): Identificador único - **project_id** (FK -> projects.id, NOT NULL): Projeto a que pertence - **title** (VARCHAR(200), NOT NULL): Título resumido - **description** (TEXT): Descrição detalhada - **status** (VARCHAR(20), NOT NULL, DEFAULT 'open'): Estado atual - **priority** (VARCHAR(20), NOT NULL, DEFAULT 'medium'): Prioridade - **reporter_id** (FK -> users.id, SET NULL): Quem criou a issue - **created_at** (TIMESTAMP): Data de criação - **updated_at** (TIMESTAMP): Data da última atualização

Status Possíveis: - **open:** Recém criada, aguardando trabalho - **in_progress:** Em desenvolvimento - **resolved:** Trabalho completo, aguardando verificação - **closed:** Finalizada e verificada

Prioridades: - **low:** Não urgente - **medium:** Prioridade normal - **high:** Urgente - **critical:** Blocker, máxima prioridade

Constraints: - CHECK (status IN ('open', 'in_progress', 'resolved', 'closed')) - CHECK (priority IN ('low', 'medium', 'high', 'critical')) - FOREIGN KEY (project_id)

REFERENCES projects(id) ON DELETE CASCADE - FOREIGN KEY (reporter_id) REFERENCES users(id) ON DELETE SET NULL - Indexes em project_id, status, priority, reporter_id

Relações: - N:1 com Project - N:1 com User (reporter) - N:M com User (assignees, via assignments) - N:M com Label (via issue_labels) - 1:N com Comment

4.2.4 Label (Etiqueta)

Representa categorias ou tags para classificar issues.

Atributos: - id (PK, INTEGER): Identificador único - name (VARCHAR(50), UNIQUE, NOT NULL): Nome da label - color (VARCHAR(7), NOT NULL, DEFAULT '#808080'): Cor em hex (e.g., #FF5733) - created_at (TIMESTAMP): Data de criação - updated_at (TIMESTAMP): Data da última atualização

Constraints: - UNIQUE (name) - Index em name

Relações: - N:M com Issue (via issue_labels)

Regras de Negócio: - Labels são globais (não por projeto) - Apenas admins podem criar/editar/eliminar labels - Exemplos: “bug”, “enhancement”, “documentation”, “urgent”

4.2.5 Comment (Comentário)

Representa discussões e atualizações em issues.

Atributos: - id (PK, INTEGER): Identificador único - issue_id (FK -> issues.id, NOT NULL): Issue a que pertence - author_id (FK -> users.id, SET NULL): Autor do comentário - content (TEXT, NOT NULL): Conteúdo do comentário - created_at (TIMESTAMP): Data de criação - updated_at (TIMESTAMP): Data da última edição

Constraints: - FOREIGN KEY (issue_id) REFERENCES issues(id) ON DELETE CASCADE - FOREIGN KEY (author_id) REFERENCES users(id) ON DELETE SET NULL - Indexes em issue_id, author_id, created_at

Relações: - N:1 com Issue - N:1 com User (author)

Regras de Negócio: - Ordenados cronologicamente (ASC) - Author pode editar/eliminar os próprios comentários - Admins e project owners podem eliminar qualquer comentário

4.3 Tabelas de Associação (Many-to-Many)

4.3.1 ProjectMember

Associa utilizadores a projetos com roles específicas.

Atributos: - id (PK, INTEGER): Identificador único - project_id (FK -> projects.id, NOT NULL): Projeto - user_id (FK -> users.id, NOT NULL): Utilizador - role (VARCHAR(20), NOT NULL, DEFAULT 'member'): Role no projeto - joined_at (TIMESTAMP): Data de adesão

Roles no Projeto: - owner: Criador do projeto, controlo total - admin: Pode gerir members e configurações - member: Pode criar/editar issues e comentários - viewer: Apenas leitura

Constraints: - UNIQUE (project_id, user_id): Um user não pode ser membro duplicado - Indexes em project_id e user_id

4.3.2 Assignment

Associa utilizadores (assignees) a issues.

Atributos: - `id` (PK, INTEGER): Identificador único - `issue_id` (FK -> issues.id, NOT NULL): Issue - `user_id` (FK -> users.id, NOT NULL): Utilizador atribuído - `assigned_at` (TIMESTAMP): Data de atribuição

Constraints: - UNIQUE (`issue_id`, `user_id`): Um user não pode ser assignee duplicado - Indexes em `issue_id` e `user_id`

Regras de Negócio: - Apenas members do projeto podem ser assignees - Uma issue pode ter múltiplos assignees - Um user pode estar atribuído a múltiplas issues

4.3.3 issue_labels

Associa labels a issues (pura associação, sem atributos extra).

Atributos: - `issue_id` (FK -> issues.id, PRIMARY KEY): Issue - `label_id` (FK -> labels.id, PRIMARY KEY): Label - `created_at` (TIMESTAMP): Quando foi adicionada

Constraints: - PRIMARY KEY (`issue_id`, `label_id`)

4.4 Relações e Cardinalidades

4.4.1 User <-> Project

Relação 1: Owner (1:N) - Um User pode ser owner de múltiplos Projects - Um Project tem exatamente um owner - ON DELETE CASCADE: Se owner é eliminado, projetos são eliminados

Relação 2: Membership (N:M via `project_members`) - Um User pode ser member de múltiplos Projects - Um Project pode ter múltiplos members - ON DELETE CASCADE: Se User ou Project é eliminado, memberships são eliminadas

4.4.2 User <-> Issue

Relação 1: Reporter (1:N) - Um User pode reportar múltiplas Issues - Uma Issue tem um reporter (pode ser NULL se user eliminado) - ON DELETE SET NULL: Se reporter é eliminado, `reporter_id` fica NULL

Relação 2: Assignment (N:M via `assignments`) - Um User pode estar atribuído a múltiplas Issues - Uma Issue pode ter múltiplos assignees - ON DELETE CASCADE: Se User ou Issue é eliminado, `assignments` são eliminadas

4.4.3 Project -> Issue (1:N)

- Um Project pode ter múltiplas Issues
- Uma Issue pertence a exatamente um Project
- ON DELETE CASCADE: Se Project é eliminado, todas as suas Issues são eliminadas

4.4.4 Issue -> Comment (1:N)

- Uma Issue pode ter múltiplos Comments

- Um Comment pertence a exatamente uma Issue
- ON DELETE CASCADE: Se Issue é eliminada, todos os seus Comments são eliminados

4.4.5 User -> Comment (1:N)

- Um User pode criar múltiplos Comments
- Um Comment tem um author (pode ser NULL se user eliminado)
- ON DELETE SET NULL: Se author é eliminado, author_id fica NULL

4.4.6 Issue <-> Label (N:M via issue_labels)

- Uma Issue pode ter múltiplas Labels
- Uma Label pode estar em múltiplas Issues
- ON DELETE CASCADE: Se Issue ou Label é eliminada, associações são eliminadas

4.5 Decisões de Modelação

4.5.1 1. Tabelas de Associação com Atributos

ProjectMember e Assignment não são apenas tabelas de junção puras - têm atributos adicionais: - **role** em ProjectMember: Necessário para diferentes níveis de acesso - **assigned_at** em Assignment: Útil para tracking e auditoria

Isto justifica criar Models (não apenas `db.Table`).

4.5.2 2. Soft Delete vs Hard Delete

Decisão: Hard Delete (ON DELETE CASCADE/SET NULL)

Justificação: - Simplicidade: Menos complexidade no código - Não há requisito de auditoria completa neste projeto - Melhoria futura: Implementar soft delete (campo `deleted_at`)

4.5.3 3. Reporter vs Assignee

Issues têm dois tipos de associação com Users: - **Reporter** (1:N): Quem criou a issue - **Assignee** (N:M): Quem está trabalhando nela

Esta separação permite workflows realistas onde o reportador pode não ser quem resolve.

4.5.4 4. Labels Globais

Labels não são por projeto, mas globais: - [OK] Consistência: Mesmos nomes/cores em todos os projetos - [OK] Reutilização: Não duplicar “bug”, “enhancement”, etc - [X] Flexibilidade: Não permite labels específicas de projeto

Trade-off aceite: Para um projeto de portfólio, labels globais são suficientes e mais simples.

4.5.5 5. Indexes para Performance

Indexes estratégicos em colunas frequentemente queries: - Foreign keys (sempre indexadas) - **username**, **email** em User (para login/lookup) - **status**, **priority** em Issue (para filtros) - **created_at** em Comment (para ordenação)

Capítulo 5

API REST - Endpoints e Exemplos

5.1 Convenções da API

5.1.1 Versionamento

Todos os endpoints estão sob `/api/v1/`: - Permite evolução da API sem breaking changes - Clientes podem especificar versão desejada - Futuras versões: `/api/v2/`, etc.

5.1.2 Estrutura de URLs

Seguimos convenções RESTful: - GET `/api/v1/resources`: Listar recursos - POST `/api/v1/resources`: Criar recurso - GET `/api/v1/resources/{id}`: Obter recurso específico - PUT `/api/v1/resources/{id}`: Atualizar recurso completo - PATCH `/api/v1/resources/{id}`: Atualizar recurso parcial - DELETE `/api/v1/resources/{id}`: Eliminar recurso

5.1.3 Status Codes HTTP

Código	Significado	Uso
200 OK	Sucesso	GET, PUT, PATCH bem-sucedidos
201 Created	Recurso criado	POST bem-sucedido
204 No Content	Sucesso sem corpo	DELETE bem-sucedido
400 Bad Request	Dados inválidos	Validação falhou
401 Unauthorized	Não autenticado	Token ausente/inválido
403 Forbidden	Sem permissão	Autorização falhou
404 Not Found	Recurso não existe	ID inválido
409 Conflict	Conflito	Username/email já existe

Código	Significado	Uso
422 Unprocessable Entity	Validação falhou	Schema validation error
429 Too Many Requests	Rate limit	Excedeu 100 req/min
500 Internal Server Error	Erro do servidor	Bug ou exceção não tratada

5.1.4 Formato de Resposta

Sucesso:

```
{
  "data": { ... },
  "message": "Operation successful",
  "meta": { "pagination": ... }
}
```

Erro:

```
{
  "error": {
    "message": "Error description",
    "status": 400,
    "code": "VALIDATION_ERROR",
    "details": { ... }
  }
}
```

5.1.5 Paginação

Listagens suportam paginação via query parameters: - **page** (default: 1): Número da página - **per_page** (default: 20, max: 100): Itens por página

Resposta inclui metadata:

```
{
  "data": [...],
  "meta": {
    "total": 150,
    "page": 2,
    "per_page": 20,
    "total_pages": 8
  }
}
```

5.1.6 Filtros e Ordenação

Endpoints de listagem suportam filtros via query parameters: - **status=open**: Filtrar por status - **priority=high**: Filtrar por prioridade - **sort=created_at**: Ordenar por campo - **order=desc**:

Ordem decendente (default: asc)

5.2 Endpoints por Categoria

5.2.1 Authentication (/api/v1/auth)

5.2.1.1 1. Register User

Criar novo utilizador.

Request:

POST /api/v1/auth/register
Content-Type: application/json

```
{  
  "username": "john_doe",  
  "email": "john@example.com",  
  "password": "SecurePass123!",  
  "role": "developer"  
}
```

Response (201 Created):

```
{  
  "data": {  
    "id": 1,  
    "username": "john_doe",  
    "email": "john@example.com",  
    "role": "developer",  
    "is_active": true,  
    "created_at": "2026-02-20T12:00:00Z"  
  },  
  "message": "User registered successfully"  
}
```

Validações: - Username: 3-50 caracteres, alfanumérico + underscore - Email: Formato válido, único
- Password: Mínimo 8 caracteres, 1 maiúscula, 1 minúscula, 1 número - Role: admin, developer, ou viewer

5.2.1.2 2. Login

Autenticar e obter tokens.

Request:

POST /api/v1/auth/login
Content-Type: application/json

```
{  
  "username": "john_doe",
```

```
"password": "SecurePass123!"  
}
```

Response (200 OK):

```
{  
  "data": {  
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGc...",  
    "refresh_token": "eyJ0eXAiOiJKV1QiLCJhbGc...",  
    "token_type": "Bearer",  
    "expires_in": 900,  
    "user": {  
      "id": 1,  
      "username": "john_doe",  
      "email": "john@example.com",  
      "role": "developer"  
    }  
  },  
  "message": "Login successful"  
}
```

Access Token: Válido por 15 minutos, usar em todas as requests autenticadas

Refresh Token: Válido por 7 dias, usar para obter novo access token

5.2.1.3 3. Refresh Token

Obter novo access token sem re-login.

Request:

POST /api/v1/auth/refresh

Authorization: Bearer <refresh_token>

Response (200 OK):

```
{  
  "data": {  
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGc...",  
    "token_type": "Bearer",  
    "expires_in": 900  
  },  
  "message": "Token refreshed successfully"  
}
```

5.2.1.4 4. Logout

Invalidar tokens (client-side na implementação atual).

Request:

POST /api/v1/auth/logout

Authorization: Bearer <access_token>

Response (200 OK):

```
{
  "message": "Logged out successfully"
}
```

Nota: Na implementação atual, logout é principalmente client-side (remover tokens). Para revogação server-side, seria necessário token blacklist (melhoria futura).

5.2.2 Projects (/api/v1/projects)

5.2.2.1 1. Create Project

Criar novo projeto.

Request:

POST /api/v1/projects
Authorization: Bearer <access_token>
Content-Type: application/json

```
{
  "name": "My Awesome Project",
  "description": "A project to track awesome features"
}
```

Response (201 Created):

```
{
  "data": {
    "id": 1,
    "name": "My Awesome Project",
    "description": "A project to track awesome features",
    "owner_id": 1,
    "is_active": true,
    "created_at": "2026-02-20T12:00:00Z",
    "updated_at": "2026-02-20T12:00:00Z",
    "owner": {
      "id": 1,
      "username": "john_doe",
      "role": "developer"
    }
  },
  "message": "Project created successfully"
}
```

Regras: - User autenticado torna-se owner automaticamente - Owner é automaticamente adicionado como member com role 'owner'

5.2.2.2 2. List Projects

Listar projetos do utilizador (owned + member).

Request:

GET /api/v1/projects?page=1&per_page=20&is_active=true

Authorization: Bearer <access_token>

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "name": "My Awesome Project",
      "description": "A project to track awesome features",
      "owner_id": 1,
      "is_active": true,
      "created_at": "2026-02-20T12:00:00Z"
    }
  ],
  "meta": {
    "total": 1,
    "page": 1,
    "per_page": 20,
    "total_pages": 1
  }
}
```

Filtros: - is_active=true|false: Apenas ativos ou arquivados

5.2.2.3 3. Get Project Details

Obter detalhes de um projeto específico.

Request:

GET /api/v1/projects/1

Authorization: Bearer <access_token>

Response (200 OK):

```
{
  "data": {
    "id": 1,
    "name": "My Awesome Project",
    "description": "A project to track awesome features",
    "owner_id": 1,
    "is_active": true,
    "created_at": "2026-02-20T12:00:00Z",
    "updated_at": "2026-02-20T12:00:00Z",
    "owner": {
      "id": 1,
      "username": "john_doe",
      "role": "developer"
    }
  }
}
```

```
    },
    "members": [
      {
        "user_id": 1,
        "username": "john_doe",
        "role": "owner",
        "joined_at": "2026-02-20T12:00:00Z"
      }
    ],
    "issue_count": 5
  }
}
```

5.2.2.4 4. Add Project Member

Adicionar utilizador ao projeto.

Request:

POST /api/v1/projects/1/members
Authorization: Bearer <access_token>
Content-Type: application/json

```
{
  "user_id": 2,
  "role": "member"
}
```

Response (201 Created):

```
{
  "data": {
    "project_id": 1,
    "user_id": 2,
    "role": "member",
    "joined_at": "2026-02-20T12:30:00Z"
  },
  "message": "Member added successfully"
}
```

Permissões: Apenas owner e admins do projeto podem adicionar members.

5.2.3 Issues (/api/v1/projects/{project_id}/issues)

5.2.3.1 1. Create Issue

Criar issue em projeto.

Request:

POST /api/v1/projects/1/issues
Authorization: Bearer <access_token>

Content-Type: application/json

```
{
  "title": "Add user authentication",
  "description": "Implement JWT-based authentication system",
  "priority": "high",
  "status": "open"
}
```

Response (201 Created):

```
{
  "data": {
    "id": 1,
    "project_id": 1,
    "title": "Add user authentication",
    "description": "Implement JWT-based authentication system",
    "status": "open",
    "priority": "high",
    "reporter_id": 1,
    "created_at": "2026-02-20T12:00:00Z",
    "updated_at": "2026-02-20T12:00:00Z",
    "reporter": {
      "id": 1,
      "username": "john_doe"
    },
    "assignees": [],
    "labels": []
  },
  "message": "Issue created successfully"
}
```

5.2.3.2 2. List Issues

Listar issues com filtros.

Request:

GET /api/v1/projects/1/issues?status=open&priority=high&page=1
Authorization: Bearer <access_token>

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "title": "Add user authentication",
      "status": "open",
      "priority": "high",
      "reporter_id": 1,

```

```
      "created_at": "2026-02-20T12:00:00Z"
    }
  ],
  "meta": {
    "total": 1,
    "page": 1,
    "per_page": 20,
    "total_pages": 1
  }
}
```

Filtros disponíveis: - **status:** open, in_progress, resolved, closed - **priority:** low, medium, high, critical - **assignee_id:** Filtrar por assignee - **reporter_id:** Filtrar por reporter - **label:** Filtrar por nome de label

5.2.3.3 3. Assign User to Issue

Atribuir utilizador a issue.

Request:

POST /api/v1/issues/1/assign
Authorization: Bearer <access_token>
Content-Type: application/json

```
{
  "user_id": 2
}
```

Response (200 OK):

```
{
  "data": {
    "issue_id": 1,
    "user_id": 2,
    "assigned_at": "2026-02-20T12:30:00Z"
  },
  "message": "User assigned to issue successfully"
}
```

5.2.3.4 4. Add Label to Issue

Adicionar label a issue.

Request:

POST /api/v1/issues/1/labels
Authorization: Bearer <access_token>
Content-Type: application/json

```
{
```

```
"label_id": 3
}
```

Response (200 OK):

```
{
  "message": "Label added to issue successfully"
}
```

5.2.4 Comments (/api/v1/issues/{issue_id}/comments)

5.2.4.1 1. Add Comment

Adicionar comentário a issue.

Request:

POST /api/v1/issues/1/comments
Authorization: Bearer <access_token>
Content-Type: application/json

```
{
  "content": "This looks good, let me review the implementation details"
}
```

Response (201 Created):

```
{
  "data": {
    "id": 1,
    "issue_id": 1,
    "author_id": 1,
    "content": "This looks good, let me review the implementation details",
    "created_at": "2026-02-20T12:00:00Z",
    "updated_at": "2026-02-20T12:00:00Z",
    "author": {
      "id": 1,
      "username": "john_doe"
    }
  },
  "message": "Comment created successfully"
}
```

5.2.4.2 2. List Comments

Listar comentários de issue.

Request:

GET /api/v1/issues/1/comments
Authorization: Bearer <access_token>

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "issue_id": 1,
      "author_id": 1,
      "content": "This looks good, let me review the implementation details",
      "created_at": "2026-02-20T12:00:00Z",
      "author": {
        "id": 1,
        "username": "john_doe"
      }
    }
  ]
}
```

Ordenação: Comentários ordenados por `created_at` ASC (cronológico).

5.2.5 Labels (/api/v1/labels)

5.2.5.1 1. Create Label (Admin Only)

Criar nova label global.

Request:

POST /api/v1/labels

Authorization: Bearer <access_token>

Content-Type: application/json

```
{
  "name": "bug",
  "color": "#FF0000"
}
```

Response (201 Created):

```
{
  "data": {
    "id": 1,
    "name": "bug",
    "color": "#FF0000",
    "created_at": "2026-02-20T12:00:00Z"
  },
  "message": "Label created successfully"
}
```

Permissões: Apenas users com role 'admin'.

5.2.5.2 2. List Labels

Listar todas as labels.

Request:

GET /api/v1/labels

Authorization: Bearer <access_token>

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "name": "bug",
      "color": "#FF0000"
    },
    {
      "id": 2,
      "name": "enhancement",
      "color": "#00FF00"
    }
  ]
}
```

5.2.6 Health (/health)

5.2.6.1 Health Check

Verificar estado da aplicação.

Request:

GET /health

Response (200 OK):

```
{
  "status": "healthy",
  "timestamp": "2026-02-20T12:00:00Z",
  "version": "1.0.0",
  "database": "connected"
}
```

Uso: Usado por Render para health checks, monitoring, load balancers.

Capítulo 6

Segurança

6.1 Gestão de Passwords

6.1.1 Hashing com bcrypt

Tecnologia: bcrypt (biblioteca flask-bcrypt)

Cost Factor: 12 (rounds de hashing)

Funcionamento:

```
import bcrypt

# Hashing durante registro
def hash_password(plain_password: str) -> str:
    salt = bcrypt.gensalt(rounds=12)
    password_hash = bcrypt.hashpw(
        plain_password.encode('utf-8'),
        salt
    )
    return password_hash.decode('utf-8')

# Verificação durante login
def verify_password(plain_password: str, password_hash: str) -> bool:
    return bcrypt.checkpw(
        plain_password.encode('utf-8'),
        password_hash.encode('utf-8')
    )
```

Características do bcrypt: - **Salt Automático:** Cada password tem salt único, impossível rainbow tables - **Slow by Design:** Intencionalmente lento (cost factor 12 \approx 250ms), dificulta brute force - **Adaptive:** Cost factor pode aumentar no futuro conforme hardware evolui - **Battle-Tested:** Usado em produção há décadas, padrão da indústria

Por que NUNCA Plain Text: - [X] Se database é comprometida, todas as passwords são expostas - [X] Administradores não devem ver passwords dos users - [X] Violação de GDPR e outras regulações de privacidade - [OK] bcrypt garante que mesmo com acesso ao database, passwords são inúteis

6.1.2 Validação de Força de Password

Requisitos mínimos (implementados via Marshmallow schema): - Mínimo 8 caracteres - Pelo menos 1 letra maiúscula - Pelo menos 1 letra minúscula - Pelo menos 1 número - Opcionalmente: 1 caractere especial (recomendado)

```
# src/schemas/user_schema.py
from marshmallow import Schema, fields, validate, validates, ValidationError
import re

class RegisterSchema(Schema):
    username = fields.Str(required=True, validate=validate.Length(min=3, max=50))
    email = fields.Email(required=True)
    password = fields.Str(required=True, validate=validate.Length(min=8))

    @validates('password')
    def validate_password(self, value):
        if not re.search(r'[A-Z]', value):
            raise ValidationError('Password must contain uppercase letter')
        if not re.search(r'[a-z]', value):
            raise ValidationError('Password must contain lowercase letter')
        if not re.search(r'\d', value):
            raise ValidationError('Password must contain number')
```

6.1.3 Armazenamento Seguro

Database: Coluna password_hash (VARCHAR(255)) - Armazena apenas o hash bcrypt - Password original é descartada imediatamente após hashing - Hash não é reversível

Nunca em Logs: - Passwords não são logged - Requests de login/register não exibem password em logs - Erro messages não revelam passwords

6.2 Autenticação JWT

6.2.1 Tokens de Acesso e Refresh

Access Token: - **Duração:** 15 minutos - **Propósito:** Autenticação de requests normais - **Payload:** User ID, role, iat (issued at), exp (expiration) - **Armazenamento Cliente:** Memory ou localStorage (menos seguro)

Refresh Token: - **Duração:** 7 dias - **Propósito:** Obter novo access token sem re-login - **Payload:** User ID, iat, exp, type='refresh' - **Armazenamento Cliente:** HttpOnly cookie (ideal) ou localStorage

Por que 2 Tokens?: - [OK] **Segurança:** Access token curto limita janela de ataque se comprometido - [OK] **Conveniência:** Refresh token evita re-login constante - [OK] **Revogação:** Refresh token pode ser revogado (implementação futura)

6.2.2 Algoritmo HS256

Escolha: HMAC-SHA256 (simétrico)

Alternativas consideradas: - RS256 (assimétrico): Mais complexo, útil se múltiplos serviços validarem tokens - HS512: Mais forte mas desnecessário para este caso

Por que HS256: - [OK] Simples: Um único secret para assinar e verificar - [OK] Performant: Mais rápido que algoritmos assimétricos - [OK] Suficiente: Para um único backend, HS256 é adequado - [X] Secret único: Se comprometido, todos os tokens são inválidos

Implementação:

```
# src/config.py
import os
from datetime import timedelta

class ProductionConfig:
    JWT_SECRET_KEY = os.environ.get('JWT_SECRET_KEY') # Must be strong random
    JWT_ACCESS_TOKEN_EXPIRES = timedelta(minutes=15)
    JWT_REFRESH_TOKEN_EXPIRES = timedelta(days=7)
    JWT_ALGORITHM = 'HS256'
```

6.2.3 Token Validation Flow

Request Autenticada: 1. Cliente envia request com header: **Authorization: Bearer <access_token>** 2. Middleware extrai token do header 3. JWT library verifica: - Assinatura (token não foi adulterado) - Expiração (token ainda válido) - Formato (payload correto) 4. Se válido, extrai `user_id` do payload 5. Request processa normalmente 6. Se inválido/expirado, retorna 401 Unauthorized

Refresh Flow: 1. Cliente detecta que access token expirou (401) 2. Cliente envia refresh token para `/api/v1/auth/refresh` 3. Backend valida refresh token 4. Se válido, gera novo access token 5. Cliente usa novo access token 6. Se refresh token inválido/expirado, redireciona para login

6.2.4 Secret Key Management

Geração:

```
# Gerar secret forte
python -c "import secrets; print(secrets.token_urlsafe(32))"
```

Armazenamento: - [X] **NUNCA** commitado no código - [X] **NUNCA** em ficheiros não-encryptados - [OK] Variável de ambiente (`.env` local, Render env vars em prod) - [OK] Secrets manager (AWS Secrets Manager, HashiCorp Vault) para produção real

.env.example:

```
# .env.example (commitado)
JWT_SECRET_KEY=your-secret-key-here-change-in-production
SECRET_KEY=your-flask-secret-key

# .env (NÃO commitado, em .gitignore)
JWT_SECRET_KEY=xK9mP2nQ8... (secret real)
SECRET_KEY=aB3cD4eF5... (secret real)
```

6.3 Proteção de Secrets

6.3.1 Variáveis de Ambiente

Desenvolvimento (.env file):

```
DATABASE_URL=postgresql://user:pass@localhost:5432/issuetracker
SECRET_KEY=dev-secret-key-change-in-production
JWT_SECRET_KEY=dev-jwt-secret-change-in-production
FLASK_ENV=development
```

Produção (Render environment variables): - DATABASE_URL: Auto-gerada pelo Render (PostgreSQL managed) - SECRET_KEY: Gerada automaticamente (generateValue: true) - JWT_SECRET_KEY: Gerada automaticamente - FLASK_ENV=production

Carregamento:

```
# src/config.py
import os
from dotenv import load_dotenv

load_dotenv() # Carrega .env em desenvolvimento

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev-fallback'
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

6.3.2 Ficheiro .gitignore

```
# Secrets
.env
*.pem
*.key
secrets.json

# Databases
*.db
*.sqlite

# Python
__pycache__/
*.pyc
venv/
.venv/
```

6.3.3 Boas Práticas

1. **Nunca commitar secrets:** Usar .env e .gitignore
2. **Secrets diferentes por ambiente:** Dev != Prod
3. **Rotação de secrets:** Mudar periodicamente (trimestral)
4. **Least privilege:** Cada componente acessa apenas os secrets necessários

5. **Auditoria:** Log de acesso a secrets (produção real)

6.4 Outras Medidas de Segurança

6.4.1 Rate Limiting

Tecnologia: Flask-Limiter

Configuração:

```
# src/app.py
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["100 per minute"],
    storage_uri="memory://" # Produção: usar Redis
)

# Rate limit específico para login (mais restrito)
@auth_bp.route('/login', methods=['POST'])
@limiter.limit("5 per minute")
def login():
    ...
```

Benefícios: - Previne brute force em login - Previne abuse de API - Protege contra DoS simples

6.4.2 Input Validation

Tecnologia: Marshmallow Schemas

Validação em 2 Níveis:

1. **Schema Level** (tipo, formato, tamanho):

```
class IssueSchema(Schema):
    title = fields.Str(required=True, validate=validate.Length(max=200))
    status = fields.Str(validate=validate.OneOf(['open', 'in_progress', 'resolved', 'closed']))
    priority = fields.Str(validate=validate.OneOf(['low', 'medium', 'high', 'critical']))
```

2. **Business Logic Level** (service layer):

```
def create_issue(self, project_id, reporter_id, **data):
    # Validate business rules
    if not self._is_project_member(reporter_id, project_id):
        raise ForbiddenError("Must be project member")
```

6.4.3 CORS (Cross-Origin Resource Sharing)

Configuração:

```
# src/app.py
from flask_cors import CORS

CORS(app, resources={
    r"/api/*": {
        "origins": os.environ.get('CORS_ORIGINS', '*').split(','),
        "methods": ["GET", "POST", "PUT", "PATCH", "DELETE"],
        "allow_headers": ["Content-Type", "Authorization"]
    }
})
```

Produção: Restringir origins a domínios específicos (não usar *).

6.4.4 SQL Injection Prevention

ORM SQLAlchemy: Todas as queries usam ORM, não SQL raw.

Parametrização Automática:

```
# SEGURO (ORM)
user = User.query.filter_by(username=username).first()

# INSEGURO (nunca fazer)
# user = db.session.execute(f"SELECT * FROM users WHERE username='{username}'")
```

Validação Extra: Marshmallow schemas garantem tipos corretos.

6.4.5 Error Handling

Não Expor Stack Traces em Produção:

```
# src/middleware/error_handler.py
@app.errorhandler(Exception)
def handle_exception(e):
    if app.config['DEBUG']:
        # Development: mostrar stack trace
        return jsonify({"error": str(e), "trace": traceback.format_exc()}), 500
    else:
        # Production: mensagem genérica
        return jsonify({
            "error": {
                "message": "Internal server error",
                "status": 500,
                "code": "INTERNAL_ERROR"
            }
        }), 500
```

Logging Seguro: Não logar passwords, tokens ou PII.

6.5 Referências e Boas Práticas

6.5.1 OWASP Top 10 (2021)

Medidas implementadas para prevenir:

1. **A01:2021 – Broken Access Control:** Autorização em services, verificação de memberships
2. **A02:2021 – Cryptographic Failures:** bcrypt para passwords, HTTPS obrigatório
3. **A03:2021 – Injection:** ORM SQLAlchemy, input validation
4. **A04:2021 – Insecure Design:** Arquitetura em camadas, separação de responsabilidades
5. **A05:2021 – Security Misconfiguration:** Secrets em env vars, DEBUG=False em prod
6. **A07:2021 – Identification and Authentication Failures:** JWT, bcrypt, rate limiting
7. **A08:2021 – Software and Data Integrity Failures:** Dependency pinning, CI/CD security scan

6.5.2 OWASP Password Storage Cheat Sheet

Práticas seguidas: - [OK] Use bcrypt, scrypt ou Argon2 - [OK] Cost factor apropriado (12 para bcrypt) - [OK] Nunca armazenar plain text ou encryption reversível - [OK] Salt automático e único por password

6.5.3 JWT Best Current Practices (RFC 8725)

Práticas seguidas: - [OK] Use algoritmos apropriados (HS256 para backend único) - [OK] Validar exp claim (expiração) - [OK] Tokens de curta duração - [OK] HTTPS obrigatório para transmissão de tokens

6.5.4 The Twelve-Factor App

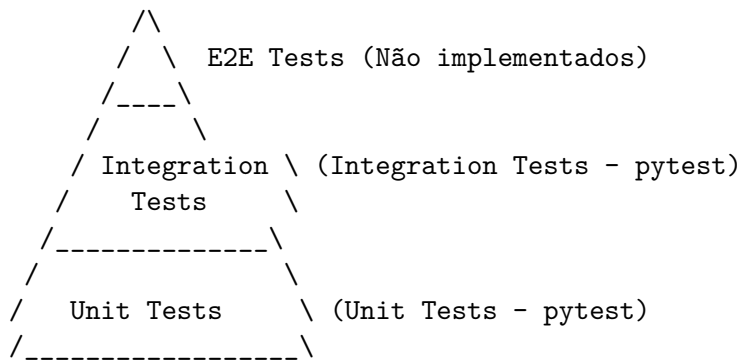
Princípios seguidos: - **III. Config:** Secrets em environment, não no código - **XI. Logs:** Logs estruturados, não incluem secrets - **XII. Admin processes:** Migrations separadas de runtime

Capítulo 7

Estratégia de Testes

7.1 Test Pyramid

O projeto segue o modelo de **Test Pyramid**, priorizando testes mais rápidos e isolados na base, com testes mais lentos e integrados no topo.



7.1.1 Níveis de Teste Implementados

7.1.1.1 1. Unit Tests (Base da Pirâmide)

Objetivo: Testar componentes individuais isoladamente.

Escopo: - Services (lógica de negócio) - Utilities (helpers, formatters) - Validators (schemas, validações custom)

Características: - [OK] Rápidos (< 50ms por teste) - [OK] Isolados (sem database, sem HTTP) - [OK] Usam mocks para dependências - [OK] Alta cobertura (>=80% nos services)

Exemplo (tests/unit/test_auth_service.py):

```
import pytest
from unittest.mock import Mock, patch
from src.services.auth_service import AuthService
from src.repositories.user_repository import UserRepository
```

```
@pytest.fixture
```

```
def mock_user_repo():
    return Mock(spec=UserRepository)

@pytest.fixture
def auth_service(mock_user_repo):
    return AuthService(user_repository=mock_user_repo)

def test_register_user_success(auth_service, mock_user_repo):
    # Arrange
    user_data = {
        'username': 'testuser',
        'email': 'test@example.com',
        'password': 'SecurePass123!'
    }
    mock_user_repo.get_by_username.return_value = None
    mock_user_repo.get_by_email.return_value = None
    mock_user_repo.create.return_value = Mock(id=1, **user_data)

    # Act
    result = auth_service.register_user(**user_data)

    # Assert
    assert result['id'] == 1
    assert result['username'] == 'testuser'
    mock_user_repo.create.assert_called_once()

def test_register_user_duplicate_username(auth_service, mock_user_repo):
    # Arrange
    mock_user_repo.get_by_username.return_value = Mock(id=1)

    # Act & Assert
    with pytest.raises(ConflictError, match='Username already exists'):
        auth_service.register_user(
            username='existing',
            email='new@example.com',
            password='SecurePass123!'
        )

@pytest.patch('src.services.auth_service.bcrypt.hashpw')
def test_password_is_hashed(mock_hashpw, auth_service, mock_user_repo):
    # Arrange
    mock_user_repo.get_by_username.return_value = None
    mock_user_repo.get_by_email.return_value = None
    mock_hashpw.return_value = b'hashed_password'

    # Act
    auth_service.register_user(
        username='testuser',
```

```
        email='test@example.com',
        password='PlainPassword123!'
    )

    # Assert
    mock_hashpw.assert_called_once()
```

7.1.1.2 2. Integration Tests (Meio da Pirâmide)

Objetivo: Testar interação entre componentes com database real.

Escopo: - Routes (endpoints HTTP) - Services + Repositories + Models - Database queries - Authentication flow

Características: - [TIME] Moderadamente rápidos (100-500ms por teste) - [DB] Usam database de teste (PostgreSQL ou SQLite) - [SYNC] Cada teste tem setup/teardown de dados - [CHART] Cobertura de fluxos completos

Exemplo (tests/integration/test_project_routes.py):

```
import pytest
from src.app import create_app, db
from src.models import User, Project

@pytest.fixture
def app():
    app = create_app('testing')
    with app.app_context():
        db.create_all()
        yield app
        db.session.remove()
        db.drop_all()

@pytest.fixture
def client(app):
    return app.test_client()

@pytest.fixture
def auth_headers(client):
    # Create user and login
    user_data = {
        'username': 'testuser',
        'email': 'test@example.com',
        'password': 'SecurePass123!'
    }
    client.post('/api/v1/auth/register', json=user_data)
    response = client.post('/api/v1/auth/login', json={
        'username': 'testuser',
        'password': 'SecurePass123!'
    })
```

```
token = response.get_json()['data']['access_token']
return {'Authorization': f'Bearer {token}'}

def test_create_project_success(client, auth_headers):
    # Arrange
    project_data = {
        'name': 'Test Project',
        'description': 'A test project'
    }

    # Act
    response = client.post(
        '/api/v1/projects',
        json=project_data,
        headers=auth_headers
    )

    # Assert
    assert response.status_code == 201
    data = response.get_json()['data']
    assert data['name'] == 'Test Project'
    assert data['description'] == 'A test project'
    assert 'id' in data
    assert 'owner_id' in data

def test_create_project_unauthorized(client):
    # Act
    response = client.post('/api/v1/projects', json={
        'name': 'Test Project'
    })

    # Assert
    assert response.status_code == 401

def test_list_projects_with_pagination(client, auth_headers):
    # Arrange: Create 25 projects
    for i in range(25):
        client.post('/api/v1/projects', json={
            'name': f'Project {i}'
        }, headers=auth_headers)

    # Act
    response = client.get(
        '/api/v1/projects?page=1&per_page=10',
        headers=auth_headers
    )

    # Assert
```

```
assert response.status_code == 200
data = response.get_json()
assert len(data['data']) == 10
assert data['meta']['total'] == 25
assert data['meta']['page'] == 1
assert data['meta']['total_pages'] == 3
```

7.1.1.3 3. E2E Tests (Topo da Pirâmide - Não Implementados)

Objetivo: Testar sistema completo do ponto de vista do utilizador.

Escopo (futuro): - Browser automation (Selenium, Playwright) - Fluxos completos de utilizador - Frontend + Backend integrados

Razão para Não Implementar: - [X] Projeto é API-only (sem frontend) - [X] Mais complexo, fora do escopo júnior - [X] Integration tests cobrem fluxos críticos

7.2 Ferramentas de Teste

7.2.1 pytest

Framework principal de testes Python.

Instalação:

```
pip install pytest pytest-cov pytest-mock
```

Configuração (pytest.ini):

```
[pytest]
testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*
addopts =
    -v
    --strict-markers
    --tb=short
    --cov=src
    --cov-report=term-missing
    --cov-report=html
    --cov-fail-under=70
```

7.2.2 pytest-cov

Code Coverage tracking.

Uso:

```
# Run tests with coverage
pytest --cov=src --cov-report=html
```

```
# View report
```

```
open htmlcov/index.html
```

CI/CD: Coverage report é gerado no pipeline, falha se < 70%.

7.2.3 factory_boy

Test Data Factories para criar dados de teste facilmente.

Exemplo (tests/factories.py):

```
import factory
from src.models import User, Project, Issue
from src.app import db

class UserFactory(factory.alchemy.SQLAlchemyModelFactory):
    class Meta:
        model = User
        sqlalchemy_session = db.session

    username = factory.Sequence(lambda n: f'user{n}')
    email = factory.LazyAttribute(lambda obj: f'{obj.username}@example.com')
    password_hash = factory.LazyFunction(
        lambda: bcrypt.hashpw(b'password123', bcrypt.gensalt()).decode()
    )
    role = 'developer'
    is_active = True

class ProjectFactory(factory.alchemy.SQLAlchemyModelFactory):
    class Meta:
        model = Project
        sqlalchemy_session = db.session

    name = factory.Sequence(lambda n: f'Project {n}')
    description = factory.Faker('text')
    owner = factory.SubFactory(UserFactory)

class IssueFactory(factory.alchemy.SQLAlchemyModelFactory):
    class Meta:
        model = Issue
        sqlalchemy_session = db.session

    title = factory.Sequence(lambda n: f'Issue {n}')
    description = factory.Faker('text')
    status = 'open'
    priority = 'medium'
    project = factory.SubFactory(ProjectFactory)
    reporter = factory.SubFactory(UserFactory)
```

```
# Usage in tests
```

```
def test_with_factories():
    user = UserFactory()
    project = ProjectFactory(owner=user)
    issue = IssueFactory(project=project, reporter=user)

    assert issue.project == project
    assert issue.reporter == user
```

7.3 Fixtures Reutilizáveis

Configuração (tests/conftest.py):

```
import pytest
from src.app import create_app, db
from src.models import User, Project, Issue

@pytest.fixture(scope='session')
def app():
    """Create application for testing."""
    app = create_app('testing')
    with app.app_context():
        yield app

@pytest.fixture(scope='function')
def _db(app):
    """Create database for testing."""
    db.create_all()
    yield db
    db.session.remove()
    db.drop_all()

@pytest.fixture
def client(app):
    """Create test client."""
    return app.test_client()

@pytest.fixture
def runner(app):
    """Create CLI runner."""
    return app.test_cli_runner()

@pytest.fixture
def test_user(_db):
    """Create test user."""
    user = User(
        username='testuser',
        email='test@example.com',
        password_hash=bcrypt.hashpw(b'password123', bcrypt.gensalt()),
```

```

        role='developer'
    )
    _db.session.add(user)
    _db.session.commit()
    return user

@pytest.fixture
def auth_token(client, test_user):
    """Get authentication token."""
    response = client.post('/api/v1/auth/login', json={
        'username': 'testuser',
        'password': 'password123'
    })
    return response.get_json()['data']['access_token']

@pytest.fixture
def auth_headers(auth_token):
    """Get authentication headers."""
    return {'Authorization': f'Bearer {auth_token}'}

```

7.4 Cobertura de Testes

7.4.1 Métricas por Módulo

Módulo	Cobertura	Prioridade
src/services/	>= 80%	Alta
src/routes/	>= 75%	Alta
src/repositories/	>= 70%	Média
src/models/	>= 60%	Média
src/schemas/	>= 70%	Média
src/utils/	>= 80%	Alta

Objetivo Geral: >= 70% no total.

7.4.2 Comandos de Coverage

```

# Run tests with coverage
pytest --cov=src --cov-report=term-missing

# Generate HTML report
pytest --cov=src --cov-report=html
open htmlcov/index.html

# Fail if coverage < 70%
pytest --cov=src --cov-fail-under=70

```

7.5 Boas Práticas de Testes

7.5.1 1. Naming Conventions

Padrão: `test_<funcionalidade>_<cenário>`

```
# [OK] Bom
def test_register_user_success()
def test_register_user_duplicate_username()
def test_create_project_unauthorized()

# [X] Mau
def test1()
def test_user()
def test_error()
```

7.5.2 2. AAA Pattern (Arrange-Act-Assert)

```
def test_create_issue():
    # Arrange: Setup test data
    project = ProjectFactory()
    user = UserFactory()
    issue_data = {'title': 'Test Issue', 'priority': 'high'}

    # Act: Execute the function
    issue = issue_service.create_issue(
        project_id=project.id,
        reporter_id=user.id,
        **issue_data
    )

    # Assert: Verify results
    assert issue['title'] == 'Test Issue'
    assert issue['priority'] == 'high'
    assert issue['project_id'] == project.id
```

7.5.3 3. Isolamento de Testes

- Cada teste deve ser independente
- Não depender da ordem de execução
- Cleanup automático (fixtures com yield)
- Database limpa entre testes

```
@pytest.fixture
def _db(app):
    db.create_all()
    yield db
    db.session.remove()
    db.drop_all() # Cleanup
```

7.5.4 4. Evitar Flaky Tests

- Não usar `sleep()` ou timeouts arbitrários
- Não depender de dados externos (APIs, tempo real)
- Mockar dependências externas
- Usar fixtures determinísticas

7.5.5 5. Test Data Builders

Usar factories em vez de criar dados manualmente:

```
# [OK] Bom (DRY, flexível)
user = UserFactory(role='admin')
project = ProjectFactory(owner=user)
```

```
# [X] Mau (repetitivo, frágil)
user = User(
    username='testuser',
    email='test@example.com',
    password_hash='...',
    role='admin',
    is_active=True,
    created_at=datetime.utcnow()
)
db.session.add(user)
db.session.commit()
```

Capítulo 8

CI/CD e Deploy

8.1 Pipeline GitHub Actions

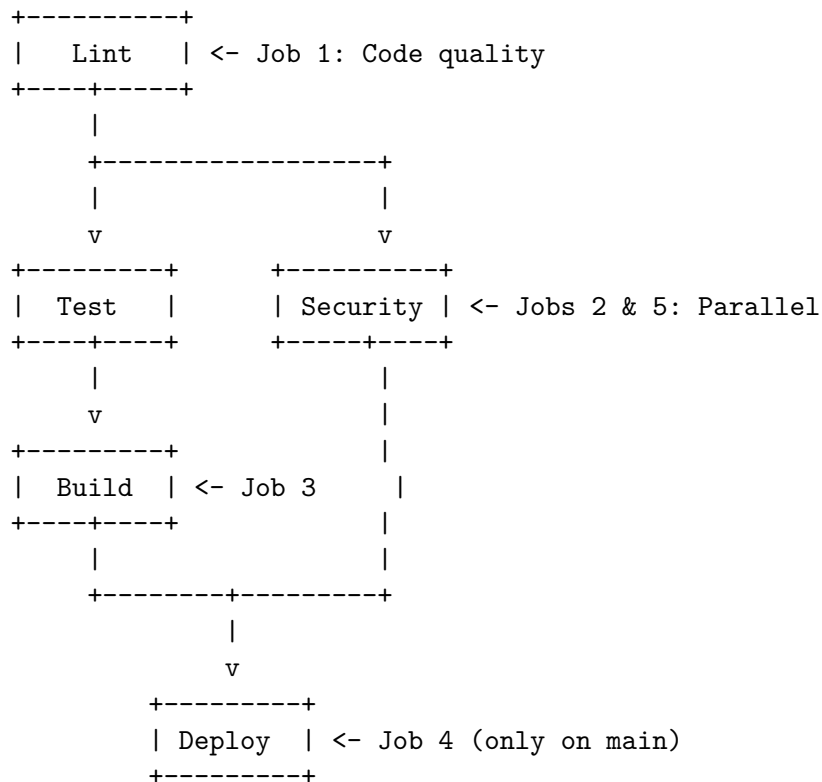
O projeto utiliza **GitHub Actions** para CI/CD automatizado, garantindo qualidade e deploy confiável.

8.1.1 Workflow Completo

Ficheiro: `.github/workflows/ci-cd.yml`

Triggers: - Push em branches `main` ou `develop` - Pull Requests para `main` ou `develop`

Jobs (5 total):



8.1.2 Job 1: Lint & Format

Objetivo: Garantir code style consistente.

Ferramentas: - **Ruff:** Linter Python rápido (substitui flake8, pylint) - **Black:** Auto-formatter Python (PEP 8)

Steps:

```
lint:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v5
      with:
        python-version: '3.11'
    - name: Install linters
      run: pip install ruff black
    - name: Run Ruff
      run: ruff check src/ --output-format=github
    - name: Check Black formatting
      run: black --check src/
```

Falha se: - Código não segue PEP 8 - Black formatting não aplicado - Ruff deteta code smells

8.1.3 Job 2: Tests

Objetivo: Executar testes e validar cobertura.

Database: PostgreSQL service container (PostgreSQL 15-alpine)

Steps:

```
test:
  runs-on: ubuntu-latest
  needs: lint
  services:
    postgres:
      image: postgres:15-alpine
      env:
        POSTGRES_DB: test_db
        POSTGRES_USER: postgres
        POSTGRES_PASSWORD: postgres
      ports:
        - 5432:5432
      options: >-
        --health-cmd pg_isready
        --health-interval 10s
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v5
    - name: Install dependencies
```

```
run: |
  pip install -r requirements.txt
  pip install -r requirements-dev.txt
- name: Run tests
  env:
    DATABASE_URL: postgresql://postgres:postgres@localhost:5432/test_db
  run: |
    pytest --cov=src --cov-report=xml --cov-fail-under=70
- name: Upload coverage to Codecov
  uses: codecov/codecov-action@v3
  with:
    file: ./coverage.xml
```

Falha se: - Qualquer teste falha - Cobertura < 70%

8.1.4 Job 3: Build Docker Image

Objetivo: Verificar que aplicação constrói corretamente.

Steps:

```
build:
  runs-on: ubuntu-latest
  needs: test
  steps:
    - uses: actions/checkout@v4
    - uses: docker/setup-buildx-action@v3
    - name: Build image
      uses: docker/build-push-action@v5
      with:
        context: .
        push: false
        tags: issue-tracker-api:${{ github.sha }}
        cache-from: type=gha
        cache-to: type=gha,mode=max
    - name: Test image
      run: |
        docker run --rm issue-tracker-api:${{ github.sha }} \
          python -c "from src.app import create_app; app = create_app()"
```

Falha se: - Dockerfile tem erros - Dependências faltando - App não inicializa

8.1.5 Job 4: Deploy

Objetivo: Deploy automatizado no Render (apenas main branch).

Condições: - Apenas em push para main - Apenas se todos os jobs anteriores passaram

Steps:

```
deploy:
  runs-on: ubuntu-latest
```

```
needs: build
if: github.ref == 'refs/heads/main' && github.event_name == 'push'
steps:
  - name: Trigger Render deployment
    run: echo "Render deploys automatically via webhook"
    # Alternativa: curl -X POST ${ secrets.RENDER_DEPLOY_HOOK_URL }
```

Render detecta push em main e faz deploy automaticamente.

8.1.6 Job 5: Security Scan

Objetivo: Detetar vulnerabilidades de segurança.

Ferramentas: - **Safety:** Escaneia dependências Python por CVEs conhecidos - **Bandit:** Linter de segurança, detecta code patterns inseguros

Steps:

```
security-scan:
  runs-on: ubuntu-latest
  needs: lint
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v5
    - name: Install security tools
      run: pip install safety bandit
    - name: Check dependencies
      run: safety check --json || true
    - name: Scan code
      run: bandit -r src/ -f json -o bandit-report.json || true
    - name: Upload report
      uses: actions/upload-artifact@v3
      with:
        name: bandit-report
        path: bandit-report.json
```

Não bloqueia deploy (continue-on-error: true), mas gera reports para revisão.

8.2 Branching Strategy

8.2.1 Branches Principais

main: - Branch de produção - Sempre deployable - Protegida (require PR + checks passing) - Deploy automático no Render

develop: - Branch de desenvolvimento - Integração contínua de features - Testes executados, mas sem deploy

8.2.2 Feature Branches

Padrão: feature/<nome-descritivo>

Workflow: 1. Criar branch a partir de develop: `git checkout -b feature/add-labels` 2. Desenvolver feature, commit localmente 3. Push para remote: `git push origin feature/add-labels` 4. Criar Pull Request para develop 5. CI runs automaticamente 6. Code review 7. Merge se aprovado e checks passarem 8. Branch é eliminada

Proteções em main: - Require PR (não pode push direto) - Require status checks passing (CI deve passar) - Require code review approval - Restrict who can push

8.3 Deploy no Render

8.3.1 Configuração (render.yaml)

Infrastructure as Code:

```
services:
  # Database
  - type: pserv
    name: issue-tracker-db
    env: docker
    plan: free
    databases:
      - name: issue_tracker
        user: issue_tracker_user

  # Web Service
  - type: web
    name: issue-tracker-api
    env: python
    plan: free
    region: oregon
    branch: main
    buildCommand: "pip install -r requirements.txt"
    startCommand: >
      alembic upgrade head &&
      gunicorn 'src.app:create_app()'
        --bind 0.0.0.0:$PORT
        --workers 4
        --timeout 120
    healthCheckPath: /health
    envVars:
      - key: FLASK_ENV
        value: production
      - key: DATABASE_URL
        fromDatabase:
          name: issue-tracker-db
          property: connectionString
      - key: SECRET_KEY
        generateValue: true
      - key: JWT_SECRET_KEY
```

```
generateValue: true
```

8.3.2 Processo de Deploy

Automático em Push para main: 1. Developer faz merge de PR para main 2. GitHub Actions CI/CD executa 3. Se todos os checks passam, commit é pushed 4. Render detecta push via webhook 5. Render faz pull do código 6. Executa `buildCommand`: instala dependências 7. Executa migrations: `alembic upgrade head` 8. Inicia app: `gunicorn` com 4 workers 9. Health check: verifica `/health` endpoint 10. Se healthy, substitui versão anterior 11. Traffic roteado para nova versão

Tempo típico: 3-5 minutos.

8.3.3 Variáveis de Ambiente

Configuradas automaticamente pelo Render: - `DATABASE_URL`: Connection string do PostgreSQL - `PORT`: Porta em que app deve escutar - `SECRET_KEY`: Gerado automaticamente - `JWT_SECRET_KEY`: Gerado automaticamente

Configuradas manualmente (via dashboard ou `render.yaml`): - `FLASK_ENV=production` - `LOG_LEVEL=INFO` - `CORS_ORIGINS=*` (ou domains específicos) - `RATELIMIT_DEFAULT=100` per minute

8.3.4 Health Check

Endpoint monitorizado pelo Render:

```
# src/routes/health.py
@health_bp.route('/', methods=['GET'])
def health_check():
    try:
        # Check database connection
        db.session.execute('SELECT 1')
        db_status = 'connected'
    except Exception:
        db_status = 'disconnected'

    return jsonify({
        'status': 'healthy',
        'timestamp': datetime.utcnow().isoformat(),
        'version': os.environ.get('VERSION', '1.0.0'),
        'database': db_status
    }), 200
```

Render checa endpoint a cada 30s: - Se retorna 200: Healthy - Se falha 3x consecutivas: Unhealthy, reinicia container

8.3.5 Migrations Automáticas

Executadas antes de iniciar app:

```
alembic upgrade head && gunicorn ...
```

Benefícios: - Schema sempre atualizado - Sem manual intervention - Rollback possível (downgrade)

Precauções: - Migrations devem ser backwards-compatible se possível - Testar migrations localmente antes de deploy - Backup de DB antes de migrations críticas (Render faz backups automáticos)

8.4 Rollback Básico

8.4.1 Via Render Dashboard

Interface Web: 1. Ir para Render dashboard -> Services -> issue-tracker-api 2. Clicar em “Deploys” tab 3. Selecionar deploy anterior (working) 4. Clicar “Rollback to this deploy” 5. Render redeploys versão anterior (< 1 minuto)

Limitações: Não reverte database migrations.

8.4.2 Database Migration Rollback

Manual (via shell no Render):

```
# Connect to Render shell
render ssh issue-tracker-api
```

```
# Rollback last migration
alembic downgrade -1
```

```
# Rollback to specific version
alembic downgrade abc123def456
```

```
# Restart service
exit
```

```
# Trigger manual deploy in Render UI
```

Prevenção: - Testar migrations em staging/local primeiro - Usar backwards-compatible migrations quando possível - Manter backups regulares

8.4.3 Rollback de Código + DB

Processo completo: 1. Rollback código via Render UI 2. Se migration foi executada: - SSH para Render shell - `alembic downgrade -1` - Reiniciar service 3. Verificar health check 4. Testar endpoints críticos

Alternativa: Revert commit no Git e push (força novo deploy).

8.5 Monitoring e Logs

8.5.1 Logs Estruturados

Formato JSON para fácil parsing:

```
# src/utils/logger.py
import logging
import json
```

```
class JSONFormatter(logging.Formatter):
    def format(self, record):
        log_data = {
            'timestamp': self.formatTime(record),
            'level': record.levelname,
            'logger': record.name,
            'message': record.getMessage(),
            'module': record.module,
            'function': record.funcName,
        }
        if record.exc_info:
            log_data['exception'] = self.formatException(record.exc_info)
        return json.dumps(log_data)

# Usage
logger = logging.getLogger(__name__)
logger.info('User logged in', extra={'user_id': 123, 'ip': '1.2.3.4'})
```

Output:

```
{
  "timestamp": "2026-02-20 12:00:00",
  "level": "INFO",
  "logger": "src.services.auth_service",
  "message": "User logged in",
  "user_id": 123,
  "ip": "1.2.3.4"
}
```

8.5.2 Render Metrics

Dashboard mostra: - CPU usage - Memory usage - Request count - Response times (p50, p95, p99) - Error rate - Deploy history

8.5.3 Health Check Monitoring

Render verifica /health a cada 30s: - Latência do health check - Database connectivity - Uptime percentage

8.5.4 Melhorias Futuras

Observability Completo: - **APM** (Application Performance Monitoring): New Relic, Data-dog - **Logs Centralizados:** ELK Stack, Splunk - **Metrics:** Prometheus + Grafana - **Tracing Distribuído:** Jaeger, Zipkin - **Alerting:** PagerDuty, Opsgenie

Capítulo 9

Como Correr o Projeto

9.1 Pré-requisitos

9.1.1 Ferramentas Necessárias

Para desenvolvimento local: - Python 3.11 ou superior - PostgreSQL 15 ou superior - Git - (Opcional) Docker e Docker Compose

Verificar versões:

```
python --version # Should be >= 3.11
psql --version   # Should be >= 15
git --version
docker --version # (optional)
```

9.2 Opção 1: Localmente com Docker Compose (Recomendado)

9.2.1 Passo 1: Clonar Repositório

```
git clone https://github.com/your-username/issue-tracker-api.git
cd issue-tracker-api
```

9.2.2 Passo 2: Configurar Variáveis de Ambiente

Copiar ficheiro de exemplo:

```
cp .env.example .env
```

Editar `.env` (valores já apropriados para Docker):

```
# .env
FLASK_ENV=development
DEBUG=True
SECRET_KEY=dev-secret-key-change-in-production
JWT_SECRET_KEY=dev-jwt-secret-change-in-production
JWT_ACCESS_TOKEN_EXPIRES=900
JWT_REFRESH_TOKEN_EXPIRES=604800
```

```
DATABASE_URL=postgresql://issuetracker:issuetracker@db:5432/issuetracker
```

```
LOG_LEVEL=DEBUG
```

```
LOG_FORMAT=json
```

```
CORS_ORIGINS=http://localhost:3000,http://localhost:5000
```

```
RATELIMIT_ENABLED=true
```

```
RATELIMIT_DEFAULT=100 per minute
```

9.2.3 Passo 3: Build e Iniciar Containers

```
# Build imagens
```

```
docker-compose build
```

```
# Iniciar services (db + api)
```

```
docker-compose up -d
```

```
# Ver logs
```

```
docker-compose logs -f api
```

Containers criados: - db: PostgreSQL 15-alpine - api: Flask app com hot-reload

9.2.4 Passo 4: Executar Migrations

```
# Executar dentro do container
```

```
docker-compose exec api alembic upgrade head
```

9.2.5 Passo 5: (Opcional) Seed Data

Criar utilizador admin e dados de exemplo:

```
docker-compose exec api python scripts/seed_data.py
```

9.2.6 Passo 6: Testar API

API disponível em `http://localhost:5000`:

```
# Health check
```

```
curl http://localhost:5000/health
```

```
# Register user
```

```
curl -X POST http://localhost:5000/api/v1/auth/register \
  -H "Content-Type: application/json" \
  -d '{
    "username": "testuser",
    "email": "test@example.com",
    "password": "SecurePass123!",
    "role": "developer"
  }'
```

9.2.7 Parar Containers

```
# Parar sem eliminar
docker-compose stop

# Parar e eliminar containers (mantém data)
docker-compose down

# Eliminar tudo (incluindo volumes)
docker-compose down -v
```

9.3 Opção 2: Localmente sem Docker

9.3.1 Passo 1: Clonar Repositório

```
git clone https://github.com/your-username/issue-tracker-api.git
cd issue-tracker-api
```

9.3.2 Passo 2: Criar Virtual Environment

```
# Criar venv
python -m venv venv

# Ativar (Linux/Mac)
source venv/bin/activate

# Ativar (Windows)
venv\Scripts\activate
```

9.3.3 Passo 3: Instalar Dependências

```
# Dependências principais
pip install -r requirements.txt

# Dependências de desenvolvimento (opcional)
pip install -r requirements-dev.txt
```

9.3.4 Passo 4: Setup PostgreSQL

Criar database e utilizador:

```
# Conectar ao PostgreSQL
psql -U postgres

# Criar database e user
CREATE DATABASE issuetracker;
CREATE USER issuetracker WITH PASSWORD 'issuetracker';
GRANT ALL PRIVILEGES ON DATABASE issuetracker TO issuetracker;
\q
```

9.3.5 Passo 5: Configurar Variáveis de Ambiente

```
cp .env.example .env
```

Editar `.env`:

```
DATABASE_URL=postgresql://issuetracker:issuetracker@localhost:5432/issuetracker
SECRET_KEY=your-secret-key-here
JWT_SECRET_KEY=your-jwt-secret-here
FLASK_ENV=development
```

9.3.6 Passo 6: Executar Migrations

```
# Executar migrations
alembic upgrade head
```

```
# Verificar
alembic current
```

9.3.7 Passo 7: Iniciar Aplicação

```
# Development server (hot-reload)
flask run --host=0.0.0.0 --port=5000
```

```
# Ou com Gunicorn (mais próximo de produção)
gunicorn 'src.app:create_app()' --bind 0.0.0.0:5000 --reload
```

API disponível em `http://localhost:5000`.

9.4 Executar Testes

9.4.1 Testes Completos

```
# Todos os testes
pytest
```

```
# Com output verboso
pytest -v
```

```
# Specific file
pytest tests/unit/test_auth_service.py
```

```
# Specific test
pytest tests/unit/test_auth_service.py::test_register_user_success
```

9.4.2 Testes com Cobertura

```
# Cobertura term
pytest --cov=src --cov-report=term-missing
```

```
# Cobertura HTML
```

```
pytest --cov=src --cov-report=html
open htmlcov/index.html
```

```
# Falhar se < 70%
pytest --cov=src --cov-fail-under=70
```

9.4.3 Testes Rápidos (apenas unit)

```
pytest tests/unit/
```

9.5 Deploy em Produção (Render)

9.5.1 Passo 1: Fork do Repositório

Fork `issue-tracker-api` para sua conta GitHub.

9.5.2 Passo 2: Criar Conta Render

1. Ir para <https://render.com>
2. Sign up (pode usar GitHub OAuth)
3. Verificar email

9.5.3 Passo 3: Criar New Web Service

1. Dashboard -> “New +” -> “Web Service”
2. Connect repository: `your-username/issue-tracker-api`
3. Branch: `main`
4. Name: `issue-tracker-api`
5. Region: Escolher mais próximo (Oregon para US)
6. Build Command: `pip install -r requirements.txt`
7. Start Command: `alembic upgrade head && gunicorn 'src.app:create_app()' --bind 0.0.0.0:$PORT --workers 4`
8. Plan: Free

9.5.4 Passo 4: Configurar Database

1. Dashboard -> “New +” -> “PostgreSQL”
2. Name: `issue-tracker-db`
3. Database: `issue_tracker`
4. User: `issue_tracker_user`
5. Region: Mesmo que web service
6. Plan: Free
7. Create Database

9.5.5 Passo 5: Conectar Database ao Web Service

1. Voltar para web service -> Environment
2. Adicionar env var:
 - Key: `DATABASE_URL`
 - Value: Copiar da database “Internal Database URL”

3. Adicionar outras env vars:

- SECRET_KEY: Gerar com `python -c "import secrets; print(secrets.token_urlsafe(32))"`
- JWT_SECRET_KEY: Gerar com mesmo comando
- FLASK_ENV=production
- LOG_LEVEL=INFO

9.5.6 Passo 6: Deploy

1. Clicar “Create Web Service”
2. Render faz deploy automaticamente
3. Ver logs em tempo real
4. Aguardar “Build successful” e “Live”

9.5.7 Passo 7: Executar Migrations

Primeira vez apenas (após deploy inicial):

```
# Via Render shell
render ssh issue-tracker-api
alembic upgrade head
exit
```

Futuros deploys: Migrations correm automaticamente (no startCommand).

9.5.8 Passo 8: Testar Produção

URL: `https://issue-tracker-api.onrender.com`

```
# Health check
curl https://issue-tracker-api.onrender.com/health

# Register
curl -X POST https://issue-tracker-api.onrender.com/api/v1/auth/register \
  -H "Content-Type: application/json" \
  -d '{
    "username": "produser",
    "email": "prod@example.com",
    "password": "SecurePass123!",
    "role": "developer"
  }'
```

9.5.9 Deploy Automático

Após setup inicial: 1. Fazer mudanças no código localmente 2. Commit e push para main 3. GitHub Actions CI/CD executa 4. Se checks passarem, Render detecta push 5. Render faz redeploy automaticamente 6. Nova versão live em ~3-5 minutos

9.6 Troubleshooting

9.6.1 Erro: Database Connection Failed

Docker:

```
# Verificar se container db está running
```

```
docker-compose ps
```

```
# Ver logs do db
```

```
docker-compose logs db
```

```
# Recreate db container
```

```
docker-compose down
```

```
docker-compose up -d db
```

Local:

```
# Verificar se PostgreSQL está running
```

```
sudo systemctl status postgresql
```

```
# Conectar manualmente
```

```
psql -U issueracker -d issueracker
```

9.6.2 Erro: ModuleNotFoundError

```
# Reinstalar dependências
```

```
pip install -r requirements.txt
```

```
# Verificar venv está ativado
```

```
which python # Should point to venv
```

9.6.3 Erro: Alembic Migration Failed

```
# Ver migrations disponíveis
```

```
alembic history
```

```
# Ver migração atual
```

```
alembic current
```

```
# Downgrade para anterior
```

```
alembic downgrade -1
```

```
# Upgrade novamente
```

```
alembic upgrade head
```

9.6.4 Erro: Port Already in Use

```
# Ver processo usando porta 5000
```

```
lsof -i :5000
```

```
# Matar processo
```

```
kill -9 <PID>
```

```
# Ou usar porta diferente
```

```
flask run --port=5001
```

Capítulo 10

Limitações e Melhorias Futuras

10.1 Limitações Atuais

10.1.1 1. Ausência de Frontend

Limitação: Projeto é API-only, sem interface gráfica.

Impacto: - Não pode ser usado diretamente por end-users não-técnicos - Requer cliente REST (Postman, cURL) para testar - Não demonstra skills de frontend

Workaround: Documentação de API com exemplos de cURL.

10.1.2 2. Autenticação Básica

Limitação: Apenas username/password + JWT.

Falta: - OAuth2 (login com Google, GitHub, etc.) - Two-Factor Authentication (2FA) - Single Sign-On (SSO) - Social logins - Password reset via email

Impacto: Menos conveniente e seguro que aplicações modernas.

10.1.3 3. Sem Notificações

Limitação: Não há sistema de notificações.

Falta: - Email notifications (issue assigned, comment added) - In-app notifications - Webhooks para integrações externas - Real-time push notifications

Impacto: Utilizadores não são alertados de mudanças relevantes.

10.1.4 4. Sem File Uploads

Limitação: Não é possível anexar ficheiros a issues/comments.

Falta: - Upload de imagens (screenshots) - Upload de documentos (PDFs, logs) - Storage de ficheiros (S3, Cloudinary) - Preview de imagens

Impacto: Comunicação menos rica, dificulta bug reports.

10.1.5 5. Search Limitado

Limitação: Apenas filtros simples (status, priority, assignee).

Falta: - Full-text search em títulos e descrições - Search em comentários - Search avançado com operators (AND, OR, NOT) - Autocomplete - Search suggestions

Impacto: Difícil encontrar issues em projetos grandes.

10.1.6 6. Sem Real-Time Updates

Limitação: Apenas HTTP request/response, sem WebSockets.

Falta: - Live updates quando issue muda - Presença de utilizadores (quem está online) - Live commenting - Collaborative editing

Impacto: Utilizadores precisam refresh manual para ver mudanças.

10.1.7 7. Revogação de Tokens Limitada

Limitação: JWT tokens não podem ser revogados até expirar.

Falta: - Token blacklist - Logout global (invalidar todos os tokens de um user) - Admin pode desativar user mas tokens antigos ainda válidos

Impacto: Se token é comprometido, fica válido até expirar.

10.1.8 8. Audit Logging Básico

Limitação: Apenas timestamps (created_at, updated_at).

Falta: - Audit trail completo (quem mudou o quê e quando) - History de mudanças (diff) - Restore de versões anteriores - Activity feed

Impacto: Difícil rastrear quem fez mudanças específicas.

10.2 Roadmap de Melhorias

10.2.1 Fase 1: Essential Features (1-2 meses)

10.2.1.1 1.1 Frontend (React/Vue)

Objetivo: Interface web moderna e responsiva.

Features: - Dashboard com overview de projetos e issues - CRUD UI para todas as entidades - Autenticação com JWT (armazenar tokens) - Responsive design (mobile-friendly) - Dark mode

Stack Sugerido: - React 18 + TypeScript - Tailwind CSS - React Router - React Query (data fetching) - Zustand (state management)

Integração: - Consumir API REST existente - Axios para HTTP requests - JWT em Authorization header

10.2.1.2 1.2 OAuth2 Authentication

Objetivo: Login com Google, GitHub, etc.

Implementação: - Flask-OAuthlib ou Authlib - Google OAuth2 - GitHub OAuth2 - Callback URLs
- Link social accounts a users existentes

Flow: 1. User clica “Login with Google” 2. Redirect para Google OAuth 3. User autoriza 4. Callback recebe code 5. Exchange code por access token 6. Criar/atualizar user com Google ID 7. Gerar JWT interno 8. Return JWT ao cliente

Segurança: - PKCE (Proof Key for Code Exchange) - State parameter anti-CSRF - Secure redirect URIs

10.2.1.3 1.3 Email Notifications

Objetivo: Notificar users de events importantes.

Events: - Issue atribuída a utilizador - Novo comentário em issue que user segue - Status de issue mudou - Menção em comentário (@username)

Implementação: - Celery (task queue) - Redis (message broker) - SendGrid ou AWS SES (email service) - HTML email templates

Arquitetura:

Issue created -> Celery task -> Email worker -> SendGrid API -> User inbox

10.2.2 Fase 2: Advanced Features (2-3 meses)

10.2.2.1 2.1 Two-Factor Authentication (2FA)

Objetivo: Camada extra de segurança.

Implementação: - TOTP (Time-based One-Time Password) - PyOTP library - QR code generation (para Google Authenticator, Authy) - Backup codes

Flow: 1. User enable 2FA in settings 2. Backend gera secret key 3. QR code exibido 4. User scans com authenticator app 5. User digita código de verificação 6. 2FA ativo 7. Login requer password + TOTP code

10.2.2.2 2.2 File Attachments

Objetivo: Anexar ficheiros a issues e comentários.

Implementação: - Storage: AWS S3 ou Cloudinary - Upload: Multipart form data - Model: `Attachment` (id, issue_id/comment_id, filename, url, size, mimetype) - Validação: Tamanho máximo (10MB), tipos permitidos (images, PDFs, logs)

Features: - Image preview inline - Download de ficheiros - Delete attachments - Virus scanning (ClamAV)

10.2.2.3 2.3 Full-Text Search

Objetivo: Search poderoso em issues e comentários.

Opção 1: PostgreSQL Full-Text Search:

```
CREATE INDEX issues_search_idx ON issues
USING gin(to_tsvector('english', title || ' ' || description));

SELECT * FROM issues
WHERE to_tsvector('english', title || ' ' || description)
@@ to_tsquery('authentication & system');
```

Opção 2: Elasticsearch (mais poderoso): - Indexar issues, comments em Elasticsearch - Search com ranking por relevância - Faceted search (filtros dinâmicos) - Autocomplete, fuzzy matching

Trade-off: PostgreSQL FTS é simples, Elasticsearch é mais poderoso mas complexo.

10.2.2.4 2.4 WebSockets para Real-Time

Objetivo: Updates em tempo real sem polling.

Implementação: - Flask-SocketIO - Redis (pub/sub) - Namespaces por projeto

Events: - issue.created - issue.updated - comment.added - user.typing (in comments)

Cliente:

```
import io from 'socket.io-client';

const socket = io('https://api.example.com', {
  auth: { token: jwtToken }
});

socket.on('issue.updated', (data) => {
  // Update UI
  updateIssueInList(data.issue);
});
```

10.2.3 Fase 3: Enterprise Features (3-6 meses)**10.2.3.1 3.1 Advanced Permissions**

Objetivo: Granular access control.

Implementação: - Custom roles por projeto - Permissions matrix (can_create_issue, can_edit_issue, can_delete, etc.) - Role inheritance - Project visibility (public, private, internal)

Modelo:

```
class Permission:
    CREATE_ISSUE = 'create_issue'
    EDIT_ISSUE = 'edit_issue'
    DELETE_ISSUE = 'delete_issue'
    MANAGE_MEMBERS = 'manage_members'

class Role(db.Model):
```

```
name = db.Column(db.String(50))
permissions = db.Column(ARRAY(db.String)) # ['create_issue', 'edit_issue']
```

10.2.3.2 3.2 Audit Logs

Objetivo: Rastreabilidade completa.

Implementação: - AuditLog model (id, user_id, action, entity_type, entity_id, changes, timestamp) - Trigger em updates para capturar mudanças - Activity feed por user/project

Exemplo:

```
{
  "user_id": 5,
  "action": "update",
  "entity_type": "Issue",
  "entity_id": 42,
  "changes": {
    "status": {"old": "open", "new": "in_progress"},
    "assignee_id": {"old": null, "new": 7}
  },
  "timestamp": "2026-02-20T15:30:00Z"
}
```

10.2.3.3 3.3 API Rate Limiting por User

Objetivo: Rate limiting mais granular.

Implementação Atual: Por IP (100 req/min)

Melhoria: Por user_id - Admin: 1000 req/min - Developer: 500 req/min - Viewer: 100 req/min

Implementação:

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

def get_user_id():
    try:
        return get_jwt_identity()
    except:
        return get_remote_address()

limiter = Limiter(key_func=get_user_id)
```

10.2.3.4 3.4 Soft Deletes

Objetivo: Não eliminar dados permanentemente.

Implementação: - Adicionar deleted_at column (nullable) - Queries filtram deleted_at IS NULL por default - Admins podem restaurar deleted items - Purge permanente após 30 dias (scheduled job)

Modelo:

```
class SoftDeleteMixin:
    deleted_at = db.Column(db.DateTime, nullable=True)

    def soft_delete(self):
        self.deleted_at = datetime.utcnow()
        db.session.commit()

    def restore(self):
        self.deleted_at = None
        db.session.commit()

class Issue(db.Model, SoftDeleteMixin):
    ...

    @classmethod
    def query_active(cls):
        return cls.query.filter(cls.deleted_at.is_(None))
```

10.2.3.5 3.5 Metrics e Observability

Objetivo: Monitoring production-grade.

Implementação: - Prometheus (metrics collection) - Grafana (visualização) - Custom metrics: - Request rate, latency (p50, p95, p99) - Error rate por endpoint - Active users - Database query time - Queue depth (Celery)

Dashboards: - System health (CPU, memory, disk) - Application metrics (requests, errors) - Business metrics (issues created, projects, users)

10.2.3.6 3.6 Issue Versioning

Objetivo: Track mudanças em issues, allow revert.

Implementação: - IssueVersion model (id, issue_id, version_number, title, description, status, priority, created_at, created_by) - Criar nova version em cada update significativo - UI para ver history e diff - Restore para versão anterior

Diff Visualization:

```
Title: Add user authentication
- Status: open
+ Status: in_progress
  Priority: high
- Assignee: None
+ Assignee: john_doe
```

10.2.4 Fase 4: Integrações e Ecosystem (6+ meses)

10.2.4.1 4.1 Webhooks

Objetivo: Permitir integrações externas.

Implementação: - Webhook model (id, project_id, url, events, secret) - Trigger HTTP POST em events - Retry policy (3x with exponential backoff) - Webhook signature (HMAC)

Events: - issue.created, issue.updated, issue.closed - comment.added - project.member.added

Payload:

```
{
  "event": "issue.created",
  "timestamp": "2026-02-20T15:30:00Z",
  "data": {
    "issue": {...}
  }
}
```

Integrações Populares: - Slack notifications - Jira sync - GitHub sync - Discord notifications

10.2.4.2 4.2 REST API v2

Objetivo: Evoluir API sem breaking changes.

Melhorias: - GraphQL endpoint (alternativa a REST) - Pagination com cursors (mais eficiente) - Field selection (?fields=id,name,created_at) - Batch operations - Rate limit info em headers

GraphQL Exemplo:

```
query {
  project(id: 1) {
    id
    name
    issues(status: OPEN, limit: 10) {
      id
      title
      assignees {
        username
      }
    }
  }
}
```

10.2.4.3 4.3 Mobile App (React Native)

Objetivo: App nativo iOS/Android.

Features: - Ver e filtrar issues - Criar/editar issues - Comentar - Push notifications - Offline mode (sync when online)

Stack: - React Native - Expo - Native notifications

10.2.4.4 4.4 Public API com Developer Portal

Objetivo: Permitir terceiros buildarem em cima da API.

Features: - API keys para developers externos - Rate limiting por API key - Documentation site (Swagger UI, Redoc) - SDKs (Python, JavaScript, Go) - Example apps

10.3 Priorização

High Priority (MVP+): 1. Frontend (React) 2. Email notifications 3. OAuth2 login 4. File attachments

Medium Priority (Growth): 5. 2FA 6. Full-text search 7. WebSockets 8. Audit logs

Low Priority (Scale): 9. GraphQL 10. Mobile app 11. Advanced permissions 12. Webhooks

Capítulo 11

Referências

11.1 Documentação Oficial

11.1.1 Frameworks e Bibliotecas

Flask - Documentação: <https://flask.palletsprojects.com/> - Tutorial: <https://flask.palletsprojects.com/tutorial/>
- Patterns: <https://flask.palletsprojects.com/patterns/>

SQLAlchemy - Documentação: <https://www.sqlalchemy.org/> - ORM Tutorial: <https://docs.sqlalchemy.org/en/20/orm/tutorials/index.html>
- Relationship Patterns: <https://docs.sqlalchemy.org/en/20/orm/relationships.html>

Alembic - Documentação: <https://alembic.sqlalchemy.org/> - Tutorial: <https://alembic.sqlalchemy.org/en/latest/tutorial.html>
- Auto-generate Migrations: <https://alembic.sqlalchemy.org/en/latest/autogenerate.html>

pytest - Documentação: <https://docs.pytest.org/> - Fixtures: <https://docs.pytest.org/en/stable/fixture.html>
- Parametrize: <https://docs.pytest.org/en/stable/parametrize.html>

JWT - Introdução: <https://jwt.io/introduction> - Flask-JWT-Extended: <https://flask-jwt-extended.readthedocs.io/> - Debugger: <https://jwt.io/> (decode tokens)

11.1.2 Databases

PostgreSQL - Documentação: <https://www.postgresql.org/docs/> - Tutorial: <https://www.postgresql.org/docs/current/tutorial.html>
- Performance Tips: https://wiki.postgresql.org/wiki/Performance_Optimization

11.2 Boas Práticas e Guidelines

11.2.1 Segurança

OWASP Top 10 - Site: <https://owasp.org/www-project-top-ten/> - Edição 2021: <https://owasp.org/Top10/>
- Cheat Sheets: <https://cheatsheetseries.owasp.org/>

OWASP Password Storage Cheat Sheet - Link: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- Principais recomendações: Use bcrypt/scrypt/Argon2, salt automático, cost factor apropriado

JWT Best Current Practices (RFC 8725) - RFC: <https://datatracker.ietf.org/doc/html/rfc8725>
- Principais recomendações: Usar algoritmos apropriados, validar claims, tokens de curta duração

bcrypt - Wikipedia: <https://en.wikipedia.org/wiki/Bcrypt> - Biblioteca Python: <https://pypi.org/project/bcrypt/>
- Cost Factor Calculator: <https://www.bcrypt-calculator.com/>

11.2.2 REST API Design

REST API Best Practices - Microsoft Guidelines: <https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design> - Google API Design Guide: <https://cloud.google.com/apis/design>
- RESTful Web Services: <https://restfulapi.net/>

HTTP Status Codes - MDN: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status> - httpstatuses.com: <https://httpstatuses.com/> - RFC 7231: <https://datatracker.ietf.org/doc/html/rfc7231>

11.2.3 Software Engineering

The Twelve-Factor App - Site: <https://12factor.net/> - Principais princípios: - III. Config: Store config in environment - V. Build, release, run: Strictly separate stages - XI. Logs: Treat logs as event streams - XII. Admin processes: Run as one-off processes

Clean Code - Livro: “Clean Code” by Robert C. Martin - Princípios: Meaningful names, functions should do one thing, DRY, YAGNI

Python PEP 8 (Style Guide) - PEP: <https://www.python.org/dev/peps/pep-0008/> - Principais regras: 4 spaces indentation, 79 chars line length, naming conventions

Semantic Versioning - Site: <https://semver.org/> - Formato: MAJOR.MINOR.PATCH (e.g., 1.2.3)
- Regras: MAJOR (breaking), MINOR (features), PATCH (bugfixes)

Keep a Changelog - Site: <https://keepachangelog.com/> - Formato: Markdown com secções Added, Changed, Deprecated, Removed, Fixed, Security

11.3 Ferramentas e Plataformas

Docker - Documentação: <https://docs.docker.com/> - Best Practices: <https://docs.docker.com/develop/dev-best-practices/> - Multi-stage Builds: <https://docs.docker.com/build/building/multi-stage/>

GitHub Actions - Documentação: <https://docs.github.com/en/actions> - Workflow Syntax: <https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions> - Marketplace: <https://github.com/marketplace?type=actions>

Render - Documentação: <https://render.com/docs> - Deploy Flask: <https://render.com/docs/deploy-flask> - PostgreSQL: <https://render.com/docs/databases>

11.4 Livros Recomendados

Backend Development - “Flask Web Development” by Miguel Grinberg - “Python Testing with pytest” by Brian Okken - “Designing Data-Intensive Applications” by Martin Kleppmann

Software Engineering - “Clean Code” by Robert C. Martin - “The Pragmatic Programmer” by David Thomas & Andrew Hunt - “Refactoring” by Martin Fowler

Security - “The Web Application Hacker’s Handbook” by Dafydd Stuttard - “Bulletproof SSL and TLS” by Ivan Ristić

11.5 Comunidades e Recursos

Forums e Q&A - Stack Overflow: <https://stackoverflow.com/> - Reddit /r/flask: <https://www.reddit.com/r/flask/>
- Reddit /r/Python: <https://www.reddit.com/r/Python/>

Blogs e Tutoriais - Real Python: <https://realpython.com/> - Miguel Grinberg Blog: <https://blog.miguelgrinberg.com/> - Full Stack Python: <https://www.fullstackpython.com/>

Newsletters - Python Weekly: <https://www.pythonweekly.com/> - PyCoder's Weekly: <https://pycoders.com/>

11.6 Conclusão

Este projeto demonstra um conjunto abrangente de competências de backend engineering: - [OK] Arquitetura de software profissional - [OK] Práticas de segurança robustas - [OK] Testes automatizados e CI/CD - [OK] Deploy e DevOps - [OK] Documentação técnica completa

O Issue Tracker API serve como um portfólio sólido para posições de **Junior Backend Engineer**, mostrando não apenas capacidade de codificar, mas também de projetar sistemas production-ready com atenção a qualidade, segurança e manutenibilidade.

Contacto: Para questões ou contribuições, consultar `CONTRIBUTING.md` no repositório.

Licença: MIT License - Ver `LICENSE` no repositório.

Versão: 1.0.0

Data: 20 de Fevereiro de 2026