- \usepackage{fancyhdr}
- \pagestyle{fancy}
- \fancyhead[L]{Issue Tracker API}
- \fancyhead[R]{\thepage}
- \fancyfoot[C]{Relatório Técnico}
- \usepackage{graphicx}
- \usepackage{float}

\newpage

Resumo Executivo

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**.

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 (>=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

Stack Tecnológica Principal

Categoria	Tecnologia	Justificação
Framework Web	Flask 3.0	Simplicidade, ecossistema maduro, comum em vagas júnior
Base de Dados	PostgreSQL 15	Features avançadas, JSON support, production-ready
ORM	SQLAlchemy 2.0	ORM completo, prevenção SQL injection, migrations
Autenticação	JWT (PyJWT)	Stateless, escalável, separação frontend/backend
Password Hashing	bcrypt	Padrão da indústria, resistente a brute-force
Validação	Marshmallow	Schemas declarativos, validação robusta
Testes	pytest	Framework moderno, fixtures poderosas, plugins
CI/CD	GitHub Actions	Integração nativa, gratuito para projetos públicos
Deploy	Render	Free tier, PostgreSQL managed, fácil configuração
Containerização	Docker + Compose	Consistência entre ambientes, fácil deploy

Estatísticas do Projeto

- **Linhas de Código**: ~6,485 linhas (excluindo testes e dependencies)
- **Ficheiros Python**: 69+ ficheiros organizados
- **Endpoints API**: 30+ endpoints REST versionados
- **Entidades do Modelo**: 7 entidades principais + 3 tabelas de associação
- **Cobertura de Testes**: >= 70% nos módulos críticos
- **Tempo de Desenvolvimento**: ~4 semanas (estimativa)

Destaques Técnicos

Segurança

- Passwords hashed com **bcrypt** (cost factor 12)

- JWT com tokens de acesso (15min) e refresh (7 dias)
- Rate limiting (100 req/min)
- Input validation em todos os endpoints
- Secrets geridos através de variáveis de ambiente

Arquitetura

- Monólito modular com 5 camadas bem definidas
- Factory Pattern para criação da aplicação Flask
- Blueprints para organização de rotas
- Repository Pattern para abstração de dados

Testes

- Unit tests para lógica de negócio (services)
- Integration tests para endpoints (com DB de teste)
- Fixtures reutilizáveis
- Coverage reports automáticos no CI

DevOps

- Pipeline CI/CD com 5 jobs automatizados
- Lint, format checking, security scanning
- Deploy automático em push para main
- Migrations de BD automáticas

\newpage

Objetivos e Requisitos

Contexto do Projeto

O Issue Tracker API foi desenvolvido com o objetivo de criar um **portfólio técnico robusto** que demonstre competências essenciais para um Backend Engineer Junior. O projeto simula um cenário real de desenvolvimento de uma API REST para gestão de projetos, com foco em:

1. **Demonstração de Competências Técnicas**: Capacidade de implementar uma API completa seguindo best practices
2. **Preparação para Entrevistas**: Código limpo e bem estruturado que pode ser discutido em entrevistas técnicas
3. **Aprendizagem Prática**: Aplicação de conceitos teóricos em um projeto real e completo
4. **Diferenciação no Mercado**: Portfólio que vai além de tutoriais básicos, demonstrando production readiness

Requisitos Funcionais

RF01: Gestão de Utilizadores

- **RF01.1**: Registo de novos utilizadores com validação de email único
- **RF01.2**: Autenticação via username/password
- **RF01.3**: Gestão de roles (admin, developer, viewer)

- **RF01.4**: Perfis de utilizador com informações básicas

RF02: Gestão de Projetos

- **RF02.1**: Criar, editar, listar e eliminar projetos
- **RF02.2**: Associar utilizadores a projetos (membership)
- **RF02.3**: Definir roles de membros em projetos
- **RF02.4**: Filtrar projetos por owner e status

RF03: Gestão de Issues

- **RF03.1**: Criar, editar, listar e eliminar issues em projetos
- **RF03.2**: Definir status (open, in_progress, resolved, closed)
- **RF03.3**: Definir prioridades (low, medium, high, critical)
- **RF03.4**: Atribuir issues a utilizadores (assignments)
- **RF03.5**: Filtrar issues por status, prioridade, assignee

RF04: Sistema de Labels

- **RF04.1**: Criar e gerir labels (apenas admin)
- **RF04.2**: Associar múltiplas labels a issues
- **RF04.3**: Filtrar issues por labels

RF05: Sistema de Comentários

- **RF05.1**: Adicionar comentários a issues
- **RF05.2**: Editar e eliminar comentários próprios
- **RF05.3**: Listar comentários ordenados por data

RF06: API REST

- **RF06.1**: Endpoints RESTful seguindo convenções HTTP
- **RF06.2**: Versionamento de API (/api/v1)
- **RF06.3**: Paginação em listagens
- **RF06.4**: Respostas JSON normalizadas
- **RF06.5**: Códigos de status HTTP apropriados

Requisitos Não-Funcionais

RNF01: Segurança

- **RNF01.1**: Passwords nunca armazenadas em plain text
- **RNF01.2**: Autenticação JWT com tokens de curta duração
- **RNF01.3**: Proteção contra SQL injection (uso de ORM)
- **RNF01.4**: Validação rigorosa de inputs
- **RNF01.5**: Secrets não commitados no repositório
- **RNF01.6**: Rate limiting para prevenir abuse

RNF02: Performance

- **RNF02.1**: Queries otimizadas com indexes
- **RNF02.2**: Connection pooling para base de dados
- **RNF02.3**: Lazy loading de relacionamentos quando apropriado
- **RNF02.4**: Tempo de resposta < 500ms para queries simples

RNF03: Testabilidade

- **RNF03.1**: Cobertura de testes >= 70% em módulos críticos
- **RNF03.2**: Testes isolados com base de dados de teste
- **RNF03.3**: Fixtures reutilizáveis
- **RNF03.4**: Testes devem executar em < 60 segundos

RNF04: Manutenibilidade

- **RNF04.1**: Código seguindo PEP 8 (Python style guide)
- **RNF04.2**: Arquitetura modular com separação de responsabilidades
- **RNF04.3**: Documentação inline em código complexo
- **RNF04.4**: Type hints onde aplicável
- **RNF04.5**: Migrations de BD versionadas

RNF05: Deployabilidade

- **RNF05.1**: Containerização com Docker
- **RNF05.2**: Configuração por variáveis de ambiente
- **RNF05.3**: CI/CD automatizado
- **RNF05.4**: Migrations automáticas no deploy
- **RNF05.5**: Health check endpoint

RNF06: Escalabilidade

- **RNF06.1**: Stateless (JWT permite escalamento horizontal)
- **RNF06.2**: Database connection pooling
- **RNF06.3**: Preparado para cache (Redis - futuro)

Critérios de Sucesso

O projeto é considerado bem-sucedido se:

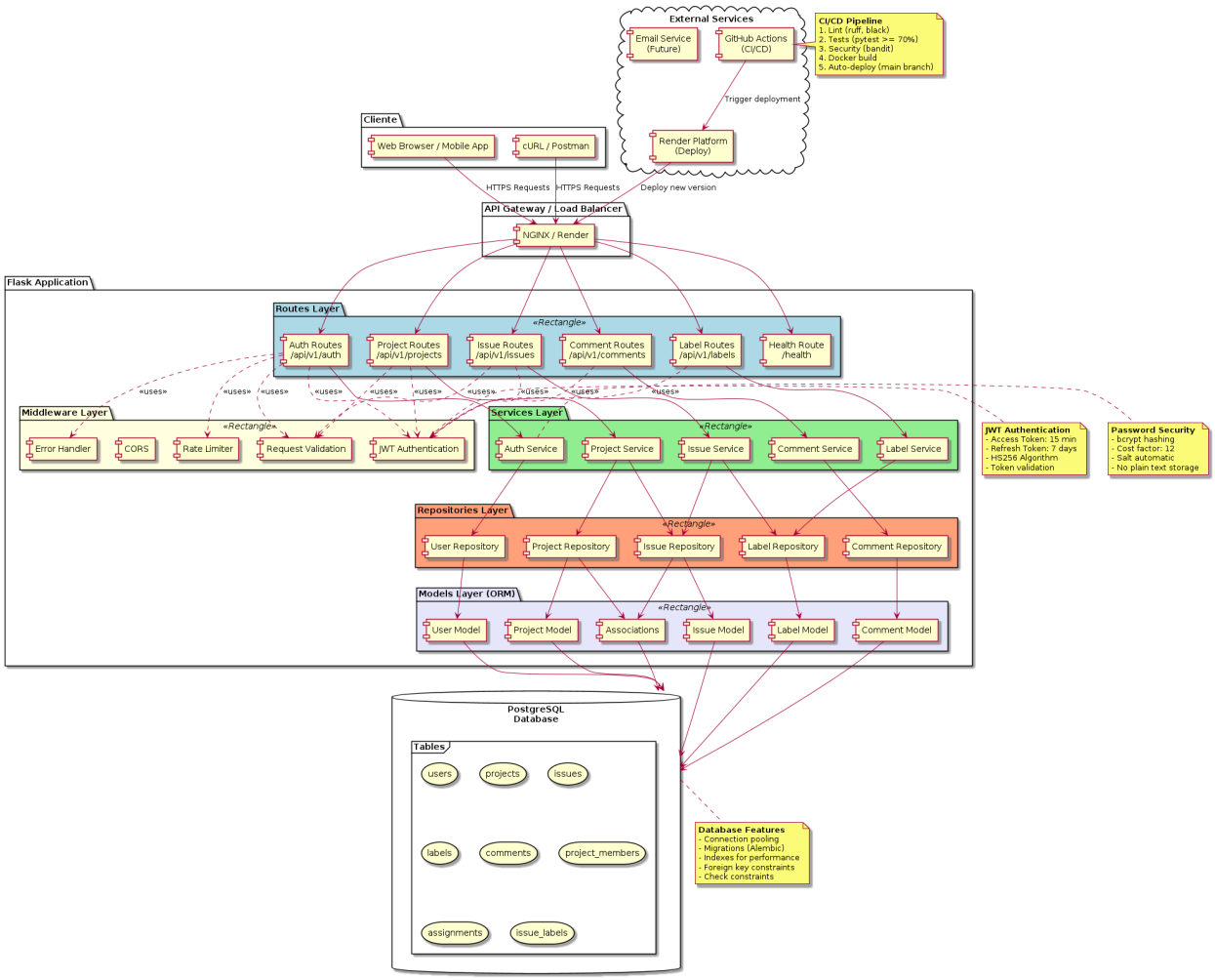
1. ✅ **Funcionalidade Completa**: Todos os requisitos funcionais implementados
2. ✅ **Qualidade de Código**: Linting passa, formatação consistente, sem code smells críticos
3. ✅ **Testes Passam**: Todos os testes passam, cobertura >= 70%
4. ✅ **CI/CD Funcional**: Pipeline executa sem erros em todos os jobs
5. ✅ **Deploy Bem-Sucedido**: API acessível em produção (Render)
6. ✅ **Documentação Completa**: README, API docs, architecture docs
7. ✅ **Segurança**: Nenhuma vulnerabilidade crítica detectada por scanners

\newpage

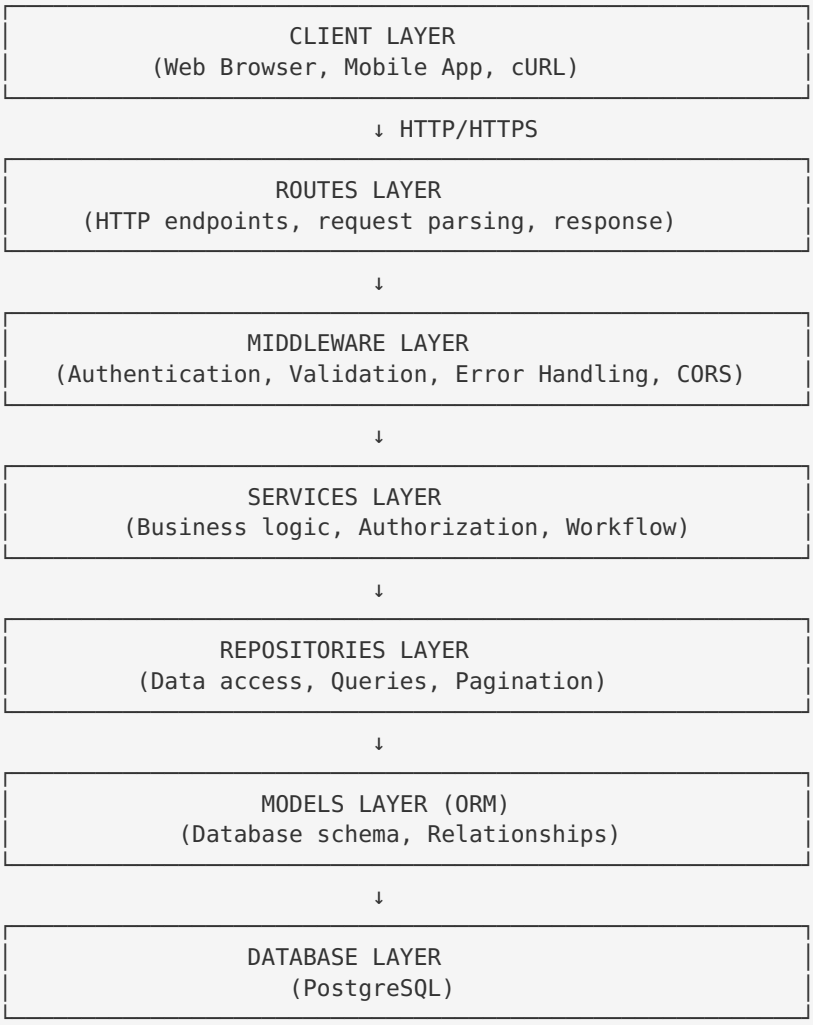
Arquitetura e Decisões Técnicas

Visão Geral da Arquitetura

O Issue Tracker API implementa uma **arquitetura monolítica modular** com separação clara de responsabilidades através de camadas. Esta escolha arquitetural é apropriada para a escala do projeto e facilita o desenvolvimento, teste e manutenção.


External Services
Email Service (Future)
GitHub Actions (CI/CD)
CI/CD Pipeline
1. Lint (ruff, black)
2. Tests (pytest >= 70%)
3. Security (bandit)
4. Docker build
5. Auto-deploy (main branch)
Trigger deployment
Render Platform (Deploy)
Cliente
Web Browser / Mobile App
cURL / Postman
HTTPS Requests
HTTPS Requests
Deploy new version
API Gateway / Load Balancer
NGINX / Render
Flask Application
Routes Layer
«Rectangle»
Auth Routes /api/v1/auth
Project Routes /api/v1/projects
Issue Routes /api/v1/issues
Comment Routes /api/v1/comments
Label Routes /api/v1/labels
Health Route /health
«uses»
Middleware Layer
«Rectangle»
Error Handler
CORS
Rate Limiter
Request Validation
JWT Authentication
Services Layer
«Rectangle»
Auth Service
Project Service
Issue Service
Comment Service
Label Service
JWT Authentication
- Access Token: 15 min
- Refresh Token: 7 days
- HS256 Algorithm
- Token validation
Password Security
- bcrypt hashing
- Cost factor: 12
- Salt automatic
- No plain text storage
Repositories Layer
«Rectangle»
User Repository
Project Repository
Issue Repository
Label Repository
Comment Repository
Models Layer (ORM)
«Rectangle»
User Model
Project Model
Associations
Issue Model
Label Model
Comment Model
PostgreSQL Database
Tables
users
projects
issues
labels
comments
project_members
assignments
issue_labels
Database Features
- Connection pooling
- Migrations (Alembic)
- Indexes for performance
- Foreign key constraints
- Check constraints

Camadas da Aplicação



Descrição das Camadas

1. Routes Layer (src/routes/)

Responsabilidades:
- Definir endpoints HTTP
- Parsing de request data (JSON, query params)
- Aplicar decorators (authentication, validation)
- Chamar services apropriados
- Retornar respostas HTTP formatadas

Ficheiros:
- `auth.py`: Autenticação (register, login, refresh, logout)
- `projects.py`: CRUD de projetos + membership
- `issues.py`: CRUD de issues + assignments + labels
- `comments.py`: CRUD de comentários
- `labels.py`: CRUD de labels (admin only)
- `health.py`: Health check endpoint

2. Middleware Layer (src/middleware/)

Responsabilidades:
- Autenticação JWT (verificação de tokens)
- Autorização role-based
- Error handling global
- Rate limiting
- CORS configuration
- Request validation

Ficheiros:
- `auth_middleware.py`: JWT token verification, user context
- `error_handler.py`: Exception catching, error formatting

3. Services Layer (src/services/)

Responsabilidades:
- Implementar lógica de negócio
- Validação de regras de negócio
- Autorização resource-based
- Coordenação entre múltiplos repositories
- Operações transacionais complexas

Ficheiros:
- `auth_service.py`: Password hashing, token generation
- `project_service.py`: Project business logic, membership
- `issue_service.py`: Issue lifecycle, assignments
- `comment_service.py`: Comment moderation
- `label_service.py`: Label management

4. Repositories Layer (src/repositories/)

Responsabilidades:
- Abstração de acesso a dados
- CRUD operations
- Queries complexas
- Paginação e filtering
- Sorting

Ficheiros:
- `base.py`: BaseRepository com métodos comuns
- `user_repository.py`, `project_repository.py`, `issue_repository.py`, etc.

5. Models Layer (src/models/)

Responsabilidades:
- Definir schema da base de dados
- Relações entre entidades
- Constraints e validações ORM-level
- Serialização (to_dict methods)

Ficheiros:
- `user.py`, `project.py`, `issue.py`, `label.py`, `comment.py`
- `associations.py`: Tabelas de associação many-to-many
- `base.py`: Base classes (TimestampMixin)

Decisões Técnicas e Trade-offs

Flask vs FastAPI

Decisão: Flask

Critério	Flask	FastAPI
Maturidade	Muito maduro (2010)	Relativamente novo (2018)
Ecossistema	Extenso, muitas extensões	Em crescimento
Curva de Aprendizagem	Suave, simples	Requer type hints, assíncrono
Performance	Boa (WSGI)	Superior (ASGI)
Async Support	Limitado	Nativo
Documentação Auto	Swagger via extensão	Automática (OpenAPI)
Vagas Júnior	Muito comum	Menos comum

Justificação:
- Flask é **mais prevalente em vagas júnior** (análise de job postings)
- Ecossistema maduro e bem estabelecido
- Curva de aprendizagem mais suave para júnior
- Performance suficiente para aplicação típica
- Melhor para demonstrar fundamentos sem complexidade de async

Trade-off Aceite: Menor performance comparado com FastAPI, sem async nativo.

JWT vs Sessions

Decisão: JWT (JSON Web Tokens)

Critério	JWT	Session Cookies
Stateless	Sim	Não (requer session store)
Escalabilidade	Fácil (horizontal)	Requer session replication
Performance	Baixa latência	Requer DB lookup
Segurança	Token pode ser roubado	Cookie can be stolen
Revocação	Complexa	Simples (delete session)
Mobile-Friendly	Sim	Limitado
Separação Frontend/Backend	Excelente	Requer same domain

Justificação:
- **Stateless**: Não requer session store (Redis/Memcached)
- **Escalabilidade horizontal**: Qualquer server pode validar token
- **Separação de concerns**: Frontend e backend podem estar em domínios diferentes
- **Mobile-friendly**: Tokens podem ser facilmente usados em apps mobile
- **Padrão moderno**: JWT é amplamente usado em APIs RESTful

Trade-off Aceite:
- Revocação de tokens é complexa (implementado token blacklist - futuro)
- Tokens maiores que session IDs (mas compressão ajuda)

Implementação:
- Access tokens: 15 minutos (curta duração para segurança)
- Refresh tokens: 7 dias (permite renovação sem re-login)
- HS256 algorithm (symmetric, suficiente para single-server)

bcrypt vs Argon2

Decisão: bcrypt

Critério	bcrypt	Argon2
Maturidade	Muito maduro (1999)	Novo (2015, vencedor PHC)
Adoção	Muito ampla	Crescente
Resistência	Excelente vs brute-force	Superior (memory-hard)
Performance	Configurável (cost factor)	Configurável
Simplicidade	Simples	Mais configurações
Suporte	Universal	Bom mas menos universal

Justificação:
- **Padrão da indústria**: bcrypt é amplamente aceite como seguro
- **Bem suportado**: Excelente suporte em todas as linguagens
- **Simplicidade**: Mais simples de configurar corretamente
- **Suficientemente seguro**: Cost factor 12 fornece excelente proteção
- **Reconhecido por recrutadores**: Mais provável de ser conhecido

Trade-off Aceite: Argon2 tem resistência teórica superior a ataques GPU/ASIC, mas bcrypt com cost factor adequado é mais que suficiente para aplicação típica.

Implementação:

```
# Cost factor 12 = 2^12 iterations (~300ms no hash)
bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt(rounds=12))
```

Monólito vs Microservices

Decisão: Monólito Modular

Critério	Monólito	Microservices
Complexidade	Baixa	Alta
Deploy	Simples	Complexo (orchestração)
Desenvolvimento	Rápido	Overhead inicial alto
Testes	Simples	Complexo (integração)
Debugging	Fácil	Difícil (distributed tracing)
Escalabilidade	Vertical (mais recursos)	Horizontal (mais instâncias)
Adequado para Júnior	Sim	Não

Justificação:
- **Escala apropriada**: Para um projeto de portfólio, monólito é suficiente
- **Simplicidade**: Mais fácil de desenvolver, testar e debugar
- **Deploy simples**: Single deployment unit
- **Demonstração de competências**: Foco em qualidade de código, não em DevOps complexo
- **Performance**: Sem overhead de network calls entre services

Modularização:
Apesar de monólito, o código é altamente modular:
- Camadas bem definidas e separadas
- Baixo acoplamento entre módulos
- Alta coesão dentro de módulos
- Fácil de extrair para microservices no futuro

Trade-off Aceite: Escalabilidade limitada (mas pode escalar verticalmente e com replicas), single point of failure.

PostgreSQL vs MySQL/SQLite

Decisão: PostgreSQL

Critério	PostgreSQL	MySQL	SQLite
Features	Muito completo	Bom	Básico
JSON Support	Excelente (JSONB)	Bom	Limitado
Compliance	ACID completo	ACID	ACID (com limites)
Performance	Excelente	Excelente	Bom
Concurrency	Excelente	Bom	Limitado
Full-text Search	Nativo	Via plugins	Básico
Production-Ready	Sim	Sim	Para desenvolvimento

Justificação:
- **Features avançadas**: Window functions, CTEs, array types
- **JSON support**: JSONB para dados semi-estruturados (futuro)
- **Full-text search**: Nativo (sem precisar Elasticsearch para search básico)
- **Open source completo**: Sem versões enterprise limitadas
- **Padrão de facto**: Amplamente usado em startups e empresas tech

Trade-off Aceite: Ligeiramente mais pesado que MySQL, mas features valem a pena.

Factory Pattern e Application Factory

O projeto usa o **Application Factory Pattern** para criar instâncias da aplicação Flask:

```python
def create_app(config_name: str = None) -> Flask:
    """Application factory pattern."""
    app = Flask(__name__)

    # Load configuration
    app.config.from_object(config[config_name or os.getenv('FLASK_ENV', 'development')
])

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

    # Register blueprints
    app.register_blueprint(auth_bp)
    app.register_blueprint(projects_bp)
    app.register_blueprint(issues_bp)
    app.register_blueprint(comments_bp)
    app.register_blueprint(labels_bp)
    app.register_blueprint(health_bp)

    # Register error handlers
    register_error_handlers(app)

    return app
```

Vantagens:
- Permite criar múltiplas instâncias da app (desenvolvimento, testes, produção)
- Facilita testes (cada teste pode ter app isolada)
- Configuração por ambiente
- Extensões inicializadas corretamente

Blueprints para Organização

Flask Blueprints são usados para modularizar a aplicação:

```python
# src/routes/projects.py
projects_bp = Blueprint('projects', __name__, url_prefix='/api/v1/projects')

@projects_bp.route('/', methods=['GET'])
@jwt_required()
def list_projects():
    ...
```

Vantagens:
- Código organizado por funcionalidade
- Prefixos de URL centralizados
- Fácil de adicionar/remover features
- Middleware pode ser aplicado por blueprint

Configuração por Ambiente

Configuração é gerida através de classes:

```python
class Config:
    """Base configuration."""
    SECRET_KEY = os.getenv('SECRET_KEY', 'dev-secret')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    JWT_SECRET_KEY = os.getenv('JWT_SECRET_KEY')

class DevelopmentConfig(Config):
    DEBUG = True
    TESTING = False

class ProductionConfig(Config):
    DEBUG = False
    TESTING = False
    # Production-specific settings

class TestingConfig(Config):
    TESTING = True
    # Faster password hashing for tests
```

\newpage

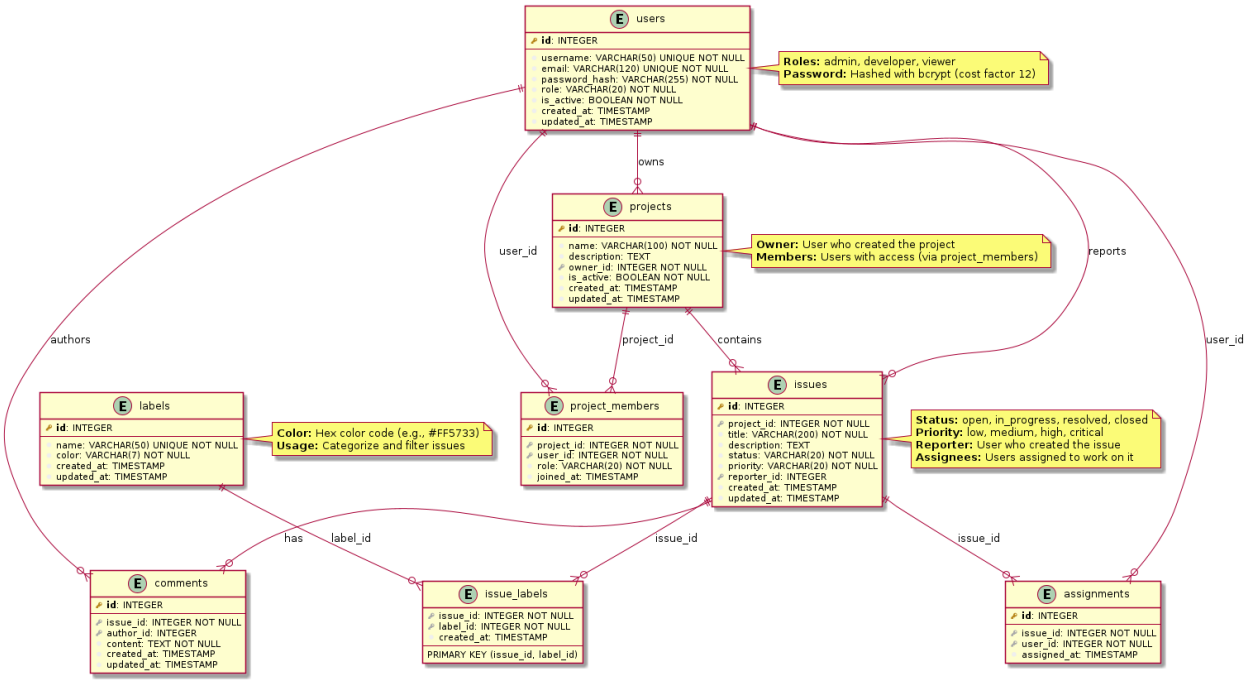
Modelo de Dados

Visão Geral

O modelo de dados do Issue Tracker API foi desenhado seguindo princípios de normalização de bases de dados relacionais, com foco em:

- **Integridade referencial**: Foreign keys e constraints
- **Normalização**: Evitar redundância de dados
- **Performance**: Indexes estratégicos
- **Flexibilidade**: Suporte a relações many-to-many

Entity Relationship Diagram (ERD)



Entidades Principais

1. User (users)

Representa utilizadores do sistema.

Atributos:
- `id` (INTEGER, PK): Identificador único
- `username` (VARCHAR(50), UNIQUE, NOT NULL): Nome de utilizador
- `email` (VARCHAR(120), UNIQUE, NOT NULL): Email único
- `password_hash` (VARCHAR(255), NOT NULL): Password hashed com bcrypt
- `role` (VARCHAR(20), NOT NULL): Role do utilizador
- `is_active` (BOOLEAN, NOT NULL): Flag de ativação
- `created_at` (TIMESTAMP): Data de criação
- `updated_at` (TIMESTAMP): Data de última atualização

Constraints:
- `CHECK (role IN ('admin', 'developer', 'viewer'))` : Roles válidos
- `UNIQUE INDEX` em username e email

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

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

2. Project (projects)

Representa projetos que agrupam issues.

Atributos:
- `id` (INTEGER, PK): Identificador único
- `name` (VARCHAR(100), NOT NULL): Nome do projeto
- `description` (TEXT): Descrição detalhada
- `owner_id` (INTEGER, FK → users.id, NOT NULL): Dono do projeto
- `is_active` (BOOLEAN, NOT NULL): Flag de ativação
- `created_at` (TIMESTAMP): Data de criação
- `updated_at` (TIMESTAMP): Data de última atualização

Indexes:
- `INDEX` em owner_id (queries frequentes por owner)
- `INDEX` em name (search)

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

Regras de Negócio:
- Owner tem acesso total ao projeto
- Membros podem ter roles diferentes (owner, admin, member, viewer)
- Apenas owner e admins podem adicionar membros

3. Issue (issues)

Representa tarefas/bugs/features a serem trabalhadas.

Atributos:
- `id` (INTEGER, PK): Identificador único
- `project_id` (INTEGER, FK → projects.id, NOT NULL): Projeto associado
- `title` (VARCHAR(200), NOT NULL): Título da issue
- `description` (TEXT): Descrição detalhada
- `status` (VARCHAR(20), NOT NULL): Estado atual
- `priority` (VARCHAR(20), NOT NULL): Prioridade
- `reporter_id` (INTEGER, FK → users.id): Quem reportou
- `created_at` (TIMESTAMP): Data de criação
- `updated_at` (TIMESTAMP): Data de última atualização

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

Indexes:
- `INDEX` em project_id (filtering por projeto)
- `INDEX` em status (filtering por status)
- `INDEX` em priority (filtering por prioridade)
- `INDEX` em reporter_id

Status Flow:

```
open → in_progress → resolved → closed
  ↓          ↓            ↓
  └──────────┴────────────┘ (pode voltar a open)
```

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

4. Label (labels)

Representa categorias/tags para issues.

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

Indexes:
- `UNIQUE INDEX` em name

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

Regras de Negócio:
- Apenas admins podem criar/editar/eliminar labels
- Labels são globais (não por projeto)
- Uma issue pode ter múltiplas labels

Exemplos de Labels:
- bug (vermelho #FF0000)
- feature (verde #00FF00)
- documentation (azul #0000FF)
- enhancement (amarelo #FFFF00)

5. Comment (comments)

Representa comentários em issues.

Atributos:
- `id` (INTEGER, PK): Identificador único
- `issue_id` (INTEGER, FK → issues.id, NOT NULL): Issue associada
- `author_id` (INTEGER, FK → users.id): Autor do comentário
- `content` (TEXT, NOT NULL): Conteúdo do comentário
- `created_at` (TIMESTAMP): Data de criação
- `updated_at` (TIMESTAMP): Data de última edição

Indexes:
- `INDEX` em issue_id (loading comments por issue)

- `INDEX` em author_id
- `INDEX` em created_at (ordenação temporal)

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

Regras de Negócio:
- Apenas author pode editar/eliminar comentário
- Comentários são ordenados por created_at
- Author pode ser NULL (soft delete de user)

Tabelas de Associação (Many-to-Many)

6. ProjectMember (project_members)

Relação N:M entre Project e User (membership).

Atributos:
- `id` (INTEGER, PK): Identificador único
- `project_id` (INTEGER, FK → projects.id, NOT NULL): Projeto
- `user_id` (INTEGER, FK → users.id, NOT NULL): Utilizador
- `role` (VARCHAR(20), NOT NULL): Role no projeto
- `joined_at` (TIMESTAMP): Data de entrada

Constraints:
- `UNIQUE (project_id, user_id)`: User não pode estar duplicado no mesmo projeto

Roles no Projeto:
- **owner**: Dono do projeto (geralmente quem criou)
- **admin**: Pode gerir membros e settings
- **member**: Pode criar/editar issues
- **viewer**: Apenas leitura

7. Assignment (assignments)

Relação N:M entre Issue e User (assignees).

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

Constraints:
- `UNIQUE (issue_id, user_id)`: User não pode estar atribuído multiplas vezes à mesma issue

Regras de Negócio:
- Uma issue pode ter múltiplos assignees
- Apenas membros do projeto podem ser atribuídos

8. issue_labels (issue_labels)

Relação N:M entre Issue e Label.

Atributos:
- `issue_id` (INTEGER, FK → issues.id, NOT NULL, PK): Issue
- `label_id` (INTEGER, FK → labels.id, NOT NULL, PK): Label
- `created_at` (TIMESTAMP): Data de associação

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

Decisões de Modelação

Soft Deletes vs Hard Deletes

Decisão atual: Hard Deletes com CASCADE

- Foreign keys com `ON DELETE CASCADE` para integridade referencial
- Comentários e assignments são eliminados quando issue é eliminada
- Issues são eliminadas quando projeto é eliminado

Futuro: Soft Deletes
- Adicionar coluna `deleted_at` (TIMESTAMP, nullable)
- Queries devem filtrar `WHERE deleted_at IS NULL`
- Permite auditoria e recovery

Timestamps Automáticos

Todas as entidades principais herdam de `TimestampMixin` :

```
class TimestampMixin:
    created_at = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utc
now)
```

Vantagens:
- Auditoria básica
- Útil para queries temporais
- Ordenação por data

Indexes Estratégicos

Indexes foram adicionados baseados em padrões de query esperados:

- **Foreign keys**: Todas as FKs têm indexes (queries JOIN frequentes)
- **Filtering columns**: status, priority (WHERE clauses)
- **Unique constraints**: username, email, label.name (lookups únicos)
- **Ordering columns**: created_at em comments

Trade-off: Indexes melhoram reads mas penalizam writes. Como a aplicação é read-heavy, o trade-off é positivo.

Normalização

O modelo está na **3ª Forma Normal (3NF)**:

- Sem redundância de dados
- Cada atributo depende apenas da primary key

- Sem dependências transitivas

Exemplo: User information não é duplicada em Comment, apenas `author_id` é armazenado.

\newpage

API REST - Endpoints e Exemplos

Convenções da API

Versionamento

Todos os endpoints estão sob o prefixo `/api/v1`, permitindo futuras versões sem breaking changes:

```
/api/v1/auth/*
/api/v1/projects/*
/api/v1/issues/*
...
```

Métodos HTTP

Método	Uso	Idempotente	Safe
GET	Obter recursos	Sim	Sim
POST	Criar recursos	Não	Não
PUT	Atualizar (completo)	Sim	Não
PATCH	Atualizar (parcial)	Não	Não
DELETE	Eliminar recursos	Sim	Não

Status Codes

Código	Significado	Uso
200 OK	Sucesso (GET, PUT, PATCH)	Recurso retornado/atualizado
201 Created	Sucesso (POST)	Recurso criado
204 No Content	Sucesso (DELETE)	Recurso eliminado
400 Bad Request	Erro do cliente	Dados inválidos
401 Unauthorized	Não autenticado	Token inválido/ausente
403 Forbidden	Não autorizado	Sem permissões
404 Not Found	Recurso não existe	ID inválido
409 Conflict	Conflito	Username/email já existe
422 Unprocessable Entity	Validação falhou	Schema validation error
429 Too Many Requests	Rate limit	Excedeu limite
500 Internal Server Error	Erro do servidor	Bug ou exceção

Formato de Respostas

Sucesso

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

Com paginação:

```
{
  "data": [ ... ],
  "meta": {
    "total": 100,
    "page": 1,
    "per_page": 20,
    "total_pages": 5
  }
}
```

Erro

```
{
  "error": {
    "message": "Validation failed",
    "status": 400,
    "code": "VALIDATION_ERROR",
    "details": {
      "validation_errors": {
        "email": ["Not a valid email address"]
      }
    }
  }
}
```

Paginação

Endpoints de listagem suportam paginação via query parameters:

- `page`: Número da página (default: 1)
- `per_page`: Items por página (default: 20, max: 100)

Exemplo:

```
GET /api/v1/projects?page=2&per_page=10
```

Filtering e Sorting

Alguns endpoints suportam filtering:

```
GET /api/v1/issues?status=open&priority=high
```

Autenticação (Auth)

POST /api/v1/auth/register

Regista novo utilizador.

Request:

```
curl -X POST http://localhost:5000/api/v1/auth/register \
  -H "Content-Type: application/json" \
  -d '{
    "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": "2026-02-20T12:00:00"
  },
  "message": "User registered successfully"
}
```

Validações:
- Username: 3-50 caracteres, alfanumérico
- Email: Formato válido, único
- Password: Mínimo 8 caracteres, complexidade
- Role: admin, developer, viewer

POST /api/v1/auth/login

Autentica utilizador e retorna tokens JWT.

Request:

```
curl -X POST http://localhost:5000/api/v1/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "username": "john_doe",
    "password": "SecurePass123!"
  }'
```

Response (200 OK):

```
{
  "data": {
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",
    "refresh_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",
    "token_type": "Bearer",
    "expires_in": 900,
    "user": {
      "id": 1,
      "username": "john_doe",
      "email": "john@example.com",
      "role": "developer"
    }
  },
  "message": "Login successful"
}
```

Tokens:
- `access_token`: Válido por 15 minutos
- `refresh_token`: Válido por 7 dias

POST /api/v1/auth/refresh

Renova access token usando refresh token.

Request:

```
curl -X POST http://localhost:5000/api/v1/auth/refresh \
  -H "Authorization: Bearer <refresh_token>"
```

Response (200 OK):

```
{
  "data": {
    "access_token": "eyJ0eXAiOiJKV1QiLCJhbGc..."
  },
  "message": "Token refreshed successfully"
}
```

POST /api/v1/auth/logout

Invalida tokens (futuro: blacklist).

Request:

```
curl -X POST http://localhost:5000/api/v1/auth/logout \
  -H "Authorization: Bearer <access_token>"
```

Response (200 OK):

```
{
  "message": "Logout successful"
}
```

Projetos (Projects)

POST /api/v1/projects

Cria novo projeto. Requer autenticação.

Request:

```
curl -X POST http://localhost:5000/api/v1/projects \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "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,
    "owner": {
      "id": 1,
      "username": "john_doe",
      "role": "developer"
    },
    "is_active": true,
    "created_at": "2026-02-20T12:00:00",
    "updated_at": "2026-02-20T12:00:00"
  },
  "message": "Project created successfully"
}
```

GET /api/v1/projects

Lista projetos do utilizador (owned + member). Suporta paginação e filtros.

Request:

```
curl -X GET "http://localhost:5000/api/v1/projects?page=1&per_page=20&is_active=true"
\
  -H "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": "2026-02-20T12:00:00"
    }
  ],
  "meta": {
    "total": 1,
    "page": 1,
    "per_page": 20,
    "total_pages": 1
  }
}
```

Query Parameters:

- `page` : Página (default: 1)
- `per_page` : Items por página (default: 20)
- `is_active` : Filtrar por status (true/false)

GET /api/v1/projects/{id}

Obtém detalhes de um projeto específico.

Request:

```
curl -X GET http://localhost:5000/api/v1/projects/1 \
  -H "Authorization: Bearer <access_token>"
```

Response (200 OK):


```
{
  "data": {
    "id": 1,
    "name": "My Awesome Project",
    "description": "A project to track awesome features",
    "owner_id": 1,
    "owner": {
      "id": 1,
      "username": "john_doe",
      "role": "developer"
    },
    "is_active": true,
    "created_at": "2026-02-20T12:00:00",
    "updated_at": "2026-02-20T12:00:00",
    "members": [
      {
        "user_id": 1,
        "username": "john_doe",
        "role": "owner",
        "joined_at": "2026-02-20T12:00:00"
      }
    ],
    "issue_count": 5
  }
}
```


PUT /api/v1/projects/{id}

Atualiza projeto. Requer ser owner ou admin.

Request:

```
curl -X PUT http://localhost:5000/api/v1/projects/1 \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Updated Project Name",
    "description": "Updated description"
  }'
```

DELETE /api/v1/projects/{id}

Elimina projeto. Requer ser owner.

Request:

```
curl -X DELETE http://localhost:5000/api/v1/projects/1 \
  -H "Authorization: Bearer <access_token>"
```

Response (204 No Content)

POST /api/v1/projects/{id}/members

Adiciona membro ao projeto. Requer ser owner ou admin.

Request:

```
curl -X POST http://localhost:5000/api/v1/projects/1/members \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": 2,
    "role": "member"
  }'
```

Issues

POST /api/v1/projects/{project_id}/issues

Cria issue em projeto. Requer ser membro do projeto.

Request:

```
curl -X POST http://localhost:5000/api/v1/projects/1/issues \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "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,
    "reporter": {
      "id": 1,
      "username": "john_doe"
    },
    "created_at": "2026-02-20T12:00:00",
    "updated_at": "2026-02-20T12:00:00",
    "assignees": [],
    "labels": []
  },
  "message": "Issue created successfully"
}
```

GET /api/v1/issues/{id}

Obtém detalhes de issue específica.

Request:

```
curl -X GET http://localhost:5000/api/v1/issues/1 \
  -H "Authorization: Bearer <access_token>"
```

Response (200 OK):

```
{
  "data": {
    "id": 1,
    "project_id": 1,
    "title": "Add user authentication",
    "description": "Implement JWT-based authentication system",
    "status": "open",
    "priority": "high",
    "reporter_id": 1,
    "reporter": {
      "id": 1,
      "username": "john_doe"
    },
    "assignees": [
      {
        "id": 2,
        "username": "jane_smith"
      }
    ],
    "labels": [
      {
        "id": 1,
        "name": "feature",
        "color": "#00FF00"
      }
    ],
    "comment_count": 3,
    "created_at": "2026-02-20T12:00:00",
    "updated_at": "2026-02-20T12:10:00"
  }
}
```

GET /api/v1/projects/{project_id}/issues

Lista issues de projeto com filtros.

Request:

```
curl -X GET "http://localhost:5000/api/v1/projects/1/issues?
status=open&priority=high&page=1" \
  -H "Authorization: Bearer <access_token>"
```

Query Parameters:
- `status` : Filtrar por status (open, in_progress, resolved, closed)
- `priority` : Filtrar por prioridade (low, medium, high, critical)
- `assignee_id` : Filtrar por assignee

- `label_id`: Filtrar por label
- `page`, `per_page`: Paginação

PUT /api/v1/issues/{id}

Atualiza issue.

Request:

```
curl -X PUT http://localhost:5000/api/v1/issues/1 \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "status": "in_progress",
    "priority": "critical"
  }'
```

POST /api/v1/issues/{id}/assign

Atribui utilizador a issue.

Request:

```
curl -X POST http://localhost:5000/api/v1/issues/1/assign \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": 2
  }'
```

POST /api/v1/issues/{id}/labels

Adiciona label a issue.

Request:

```
curl -X POST http://localhost:5000/api/v1/issues/1/labels \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "label_id": 1
  }'
```

Comentários (Comments)

POST /api/v1/issues/{issue_id}/comments

Adiciona comentário a issue.

Request:

```
curl -X POST http://localhost:5000/api/v1/issues/1/comments \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "content": "This looks good, let me review the implementation"
  }'
```

Response (201 Created):

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

GET /api/v1/issues/{issue_id}/comments

Lista comentários de issue, ordenados por data.

Request:

```
curl -X GET "http://localhost:5000/api/v1/issues/1/comments?page=1&per_page=20" \
  -H "Authorization: Bearer <access_token>"
```

Labels (Admin Only)

POST /api/v1/labels

Cria nova label. Requer role admin.

Request:

```
curl -X POST http://localhost:5000/api/v1/labels \
  -H "Authorization: Bearer <access_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "bug",
    "color": "#FF0000"
  }'
```

GET /api/v1/labels

Lista todas as labels.

Request:

```
curl -X GET http://localhost:5000/api/v1/labels \
  -H "Authorization: Bearer <access_token>"
```

Health Check

GET /api/v1/health

Endpoint de health check (não requer autenticação).

Request:

```
curl -X GET http://localhost:5000/api/v1/health
```

Response (200 OK):

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

\newpage

Segurança

A segurança é um pilar fundamental do Issue Tracker API. Foram implementadas múltiplas camadas de proteção seguindo as melhores práticas da indústria e recomendações da OWASP (Open Web Application Security Project).

Gestão de Passwords

Password Hashing com bcrypt

Nunca armazenar passwords em plain text é uma regra absoluta. O projeto usa bcrypt para hashing de passwords.

Características do bcrypt:
- **Salt automático**: Cada password tem um salt único gerado automaticamente
- **Cost factor configurável**: Controla o número de rounds (complexidade computacional)
- **Resistente a brute-force**: Cost factor aumenta o tempo necessário para testar passwords
- **Resistente a rainbow tables**: Salt único previne uso de tabelas pré-computadas

Implementação:

```python
import bcrypt

def hash_password(password: str) -> str:
    """Hash password com bcrypt usando cost factor 12."""
    salt = bcrypt.gensalt(rounds=12)  # 2^12 = 4096 iterations
    password_hash = bcrypt.hashpw(password.encode('utf-8'), salt)
    return password_hash.decode('utf-8')

def verify_password(password: str, password_hash: str) -> bool:
    """Verifica password contra hash."""
    return bcrypt.checkpw(
        password.encode('utf-8'),
        password_hash.encode('utf-8')
    )
```

Cost Factor: Porquê 12?

Cost Factor	Iterations	Tempo (~)	Adequado para
10	1,024	~100ms	Desenvolvimento
12	4,096	~300ms	**Produção (re-comendado)**
14	16,384	~1.2s	Alta segurança

Trade-off: Cost factor 12 equilibra segurança e performance:
- **Segurança**: Suficientemente lento para prevenir brute-force
- **Performance**: Não impacta negativamente a experiência do utilizador
- **Futuro-proof**: Pode ser aumentado conforme hardware evolui

Validação de Password Strength

Passwords devem cumprir requisitos mínimos:

- **Comprimento**: Mínimo 8 caracteres (recomendado: 12+)
- **Complexidade**: Pelo menos 3 de 4 categorias:
- Letras maiúsculas (A-Z)
- Letras minúsculas (a-z)
- Números (0-9)
- Caracteres especiais (!@#$%^&*)

Validação implementada no schema:

```python
from marshmallow import validates, ValidationError
import re

@validates('password')
def validate_password(self, value):
    if len(value) < 8:
        raise ValidationError('Password must be at least 8 characters')

    # Verificar complexidade
    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'[0-9]', value):
        raise ValidationError('Password must contain digit')
```

Password Reset (Futuro)

Implementação futura deve incluir:
1. Token temporário enviado por email
2. Token válido por curto período (15-30 minutos)
3. Token usado apenas uma vez
4. Password anterior não pode ser reutilizada (password history)

Autenticação JWT (JSON Web Tokens)

Arquitetura JWT

O sistema usa dois tipos de tokens:

1. **Access Token** (curta duração: 15 minutos)
   - Usado para autenticar requests à API
   - Contém: user_id, username, role
   - Expira rapidamente para limitar janela de comprometimento

2. **Refresh Token** (longa duração: 7 dias)
   - Usado apenas para renovar access tokens
   - Armazenado de forma segura pelo cliente
   - Permite renovação sem re-login

Flow de Autenticação:

```
1. Login com username/password
   ↓
2. API valida credenciais (bcrypt)
   ↓
3. API gera access_token + refresh_token
   ↓
4. Cliente armazena tokens (localStorage/secure storage)
   ↓
5. Cliente envia access_token em cada request (Authorization header)
   ↓
6. API valida access_token (assinatura + expiração)
   ↓
7. Quando access_token expira:
   - Cliente envia refresh_token para /auth/refresh
   - API valida refresh_token
   - API gera novo access_token
   ↓
8. Quando refresh_token expira: Cliente deve fazer login novamente
```

Estrutura do Token

Access Token Payload:

```
{
  "sub": "1",                    // User ID (subject)
  "username": "john_doe",
  "role": "developer",
  "iat": 1708437600,             // Issued at (timestamp)
  "exp": 1708438500,             // Expires at (15 min depois)
  "type": "access"
}
```

Refresh Token Payload:

```
{
  "sub": "1",
  "iat": 1708437600,
  "exp": 1709042400,             // Expires at (7 dias depois)
  "type": "refresh"
}
```

Algoritmo: HS256

HS256 (HMAC com SHA-256) foi escolhido:

Aspecto	HS256 (Symmetric)	RS256 (Asymmetric)
Velocidade	Rápido	Mais lento
Complexidade	Simples	Complexo (key pairs)
Uso ideal	Single server	Distributed systems
Rotação de chaves	Mais difícil	Mais fácil
Segurança	Seguro (secret protegido)	Seguro

Justificação: Para aplicação monolítica com single server, HS256 é suficiente e mais eficiente.

Secret Key Management:
- Secret jamais commitada no código
- Gerada com cryptographically secure random generator
- Armazenada em variável de ambiente
- Diferente entre ambientes (dev, test, prod)
- Mínimo 256 bits (32 bytes) de entropia

```
import secrets

# Gerar secret seguro
jwt_secret = secrets.token_urlsafe(32)  # 256 bits
```

Validação de Tokens

Verificações realizadas em cada request:

1. **Token presente**: Authorization header existe e formato é correto
2. **Assinatura válida**: Token não foi adulterado
3. **Não expirado**: Timestamp atual < exp claim
4. **Tipo correto**: Access token para operações, refresh para renovação
5. **Utilizador existe**: User ID no token corresponde a user ativo

Implementação (middleware):

```
from flask_jwt_extended import jwt_required, get_jwt_identity

@jwt_required()
def protected_endpoint():
    current_user_id = get_jwt_identity()
    # current_user_id é extraído do token e validado
    ...
```

Token Revocation (Futuro)

Atualmente, tokens não podem ser revogados antes de expirar. Implementação futura:

1. **Token Blacklist** (Redis):
   - Adicionar token à blacklist no logout

   - Verificar blacklist antes de aceitar token
   - Expiração automática (TTL igual à expiração do token)

2. **Token Versioning**:
   - Adicionar campo `token_version` em User
   - Incrementar versão no logout ou password change
   - Rejeitar tokens com versão desatualizada

Proteção de Secrets

Variáveis de Ambiente

Todos os secrets são geridos através de variáveis de ambiente:

Secrets principais:
- `SECRET_KEY`: Flask secret key (sessions, CSRF)
- `JWT_SECRET_KEY`: JWT signing key
- `DATABASE_URL`: Connection string da base de dados

Ficheiro .env (desenvolvimento):

```
# .env (NÃO COMMITAR)
SECRET_KEY=dev-secret-key-change-in-production
JWT_SECRET_KEY=dev-jwt-secret-key-change-in-production
DATABASE_URL=postgresql://user:password@localhost:5432/issue_tracker
```

Ficheiro .env.example (commitado):

```
# .env.example (template)
SECRET_KEY=your-secret-key-here
JWT_SECRET_KEY=your-jwt-secret-key-here
DATABASE_URL=postgresql://user:password@localhost:5432/dbname
```

Gitignore

`.gitignore` configurado para prevenir commit de secrets:

```
.env
*.env
.env.local
.env.*.local
```

Produção (Render)

Em produção, secrets são geridos pela plataforma:

- **Auto-generated**: Render gera automaticamente secrets seguros
- **Encrypted**: Armazenados encriptados
- **Access control**: Apenas aplicação tem acesso
- **Rotation**: Podem ser rotacionados via dashboard

Outras Medidas de Segurança

Rate Limiting

Proteção contra brute-force e DoS attacks:

```
from flask_limiter import Limiter

limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["100 per minute"]
)

@app.route("/api/v1/auth/login", methods=["POST"])
@limiter.limit("5 per minute")  # Max 5 login attempts por minuto
def login():
    ...
```

Limites configurados:
- **Default**: 100 requests/minuto (geral)
- **Login**: 5 attempts/minuto (prevenir brute-force)
- **Register**: 3 registrations/hora (prevenir spam)

Input Validation

Marshmallow schemas garantem validação rigorosa:

Validações implementadas:
- Type checking (string, integer, email, etc.)
- Length constraints (min/max)
- Format validation (email, URL, color hex)
- Required fields
- Custom validators

Exemplo:

```
from marshmallow import Schema, fields, validates

class ProjectSchema(Schema):
    name = fields.Str(required=True, validate=validate.Length(min=3, max=100))
    description = fields.Str(validate=validate.Length(max=1000))

    @validates('name')
    def validate_name(self, value):
        if not value.strip():
            raise ValidationError('Name cannot be empty')
```

CORS (Cross-Origin Resource Sharing)

Configuração de CORS para prevenir requests de origens não autorizadas:

```
from flask_cors import CORS

CORS(app, resources={
    r"/api/*": {
        "origins": ["https://myapp.com", "https://app.myapp.com"],
        "methods": ["GET", "POST", "PUT", "DELETE"],
        "allow_headers": ["Content-Type", "Authorization"]
    }
})
```

Em desenvolvimento: `origins: ["*"]` (aceitar todos)
Em produção: Lista restrita de domínios autorizados

SQL Injection Prevention

SQLAlchemy ORM previne SQL injection automaticamente:

❌ **Vulnerável (raw SQL):**

```
query = f"SELECT * FROM users WHERE username = '{username}'"
db.execute(query)  # VULNERÁVEL!
```

✅ **Seguro (ORM):**

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

SQLAlchemy usa **parameterized queries** internamente, prevenindo injection.

Error Handling

Erros são tratados de forma segura sem expor informações sensíveis:

❌ **Não fazer:**

```
# Expor stack trace para cliente
return jsonify({"error": str(exception)}), 500
```

✅ **Fazer:**

```
# Erro genérico para cliente, detalhe em logs
logger.error(f"Internal error: {exception}", exc_info=True)
return jsonify({
    "error": {
        "message": "Internal server error",
        "status": 500,
        "code": "INTERNAL_ERROR"
    }
}), 500
```

Security Headers (Futuro)

Headers HTTP de segurança devem ser adicionados:

```
@app.after_request
def set_security_headers(response):
    response.headers['X-Content-Type-Options'] = 'nosniff'
    response.headers['X-Frame-Options'] = 'DENY'
    response.headers['X-XSS-Protection'] = '1; mode=block'
    response.headers['Strict-Transport-Security'] = 'max-age=31536000; includeSubDo-
mains'
    return response
```

Referências de Segurança

OWASP Top 10 (2021)

Proteções implementadas contra o OWASP Top 10:

1. **A01:2021 – Broken Access Control**: Autorização em services, verificação de ownership
2. **A02:2021 – Cryptographic Failures**: bcrypt para passwords, HTTPS em produção
3. **A03:2021 – Injection**: SQLAlchemy ORM previne SQL injection
4. **A04:2021 – Insecure Design**: Arquitetura com segurança em mente
5. **A05:2021 – Security Misconfiguration**: Secrets em env vars, não em código
6. **A06:2021 – Vulnerable Components**: Dependencies atualizadas, security scanning
7. **A07:2021 – Identification and Authentication Failures**: JWT robusto, bcrypt
8. **A08:2021 – Software and Data Integrity Failures**: Validação de inputs
9. **A09:2021 – Security Logging Failures**: Logging estruturado (JSON)
10. **A10:2021 – Server-Side Request Forgery**: N/A (sem requests para external services)

JWT Best Practices (RFC 8725)

Seguindo recomendações do RFC 8725:

- ✅ Usar algoritmo apropriado (HS256)
- ✅ Validar “exp” claim (expiração)
- ✅ Validar “iat” claim (issued at)
- ✅ Tokens curtos para access (15min)
- ✅ Refresh tokens para renovação
- ⏳ Token revocation (futuro: blacklist)

OWASP Password Storage Cheat Sheet

- ✅ Usar algoritmo moderno (bcrypt)
- ✅ Cost factor adequado (12)
- ✅ Nunca armazenar plain text
- ✅ Validar strength na criação
- ⏳ Password history (futuro)
- ⏳ Rate limiting em password reset (futuro)

\newpage

Estratégia de Testes

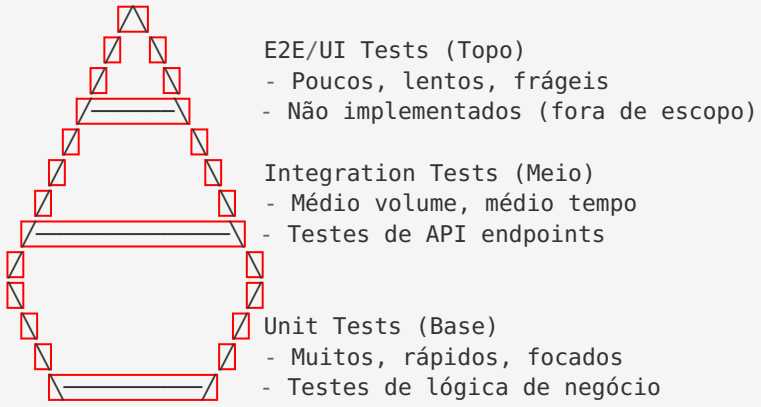
Filosofia de Testes

O Issue Tracker API adota uma abordagem pragmática de testes, focando em **qualidade sobre quantidade**. O objetivo não é 100% de cobertura, mas sim **testes significativos que garantem confiabilidade**.

Princípios:
- **Testar comportamento, não implementação**: Testes devem validar o que o código faz, não como faz
- **Testes rápidos**: Suite completa deve executar em < 60 segundos
- **Testes isolados**: Cada teste é independente e pode executar sozinho
- **Testes legíveis**: Nomenclatura clara, arrange-act-assert pattern

Test Pyramid

A estratégia segue a **Test Pyramid** de Mike Cohn:



Distribuição de testes (aproximada):
- 70% Unit tests (services, repositories, utils)
- 30% Integration tests (API endpoints)
- 0% E2E tests (futuro: com frontend)

Ferramentas e Frameworks

pytest

pytest é o framework de testes usado:

Vantagens:
- Sintaxe simples e pythônica
- Fixtures poderosas e reutilizáveis
- Descoberta automática de testes
- Output informativo
- Extensível via plugins

Instalação:

```
pip install pytest pytest-cov pytest-flask
```

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=html
    --cov-report=term-missing
```

pytest-cov

Plugin para medir cobertura de testes:

```
pytest --cov=src --cov-report=html --cov-report=term
```

Objetivos de cobertura:
- **Services**: >= 80% (lógica de negócio crítica)
- **Routes**: >= 70% (integration tests)
- **Repositories**: >= 70% (data access)
- **Overall**: >= 70%

Fixtures

Fixtures são funções reutilizáveis que preparam ambiente de teste.

Fixtures principais (conftest.py):

```python
import pytest
from src.app import create_app
from src.models.base import db as _db

@pytest.fixture(scope='session')
def app():
    """Criar app Flask para testes."""
    app = create_app('testing')

    with app.app_context():
        _db.create_all()
        yield app
        _db.drop_all()

@pytest.fixture(scope='function')
def client(app):
    """Test client Flask."""
    return app.test_client()

@pytest.fixture(scope='function')
def db(app):
    """Database limpa para cada teste."""
    with app.app_context():
        _db.session.begin_nested()
        yield _db
        _db.session.rollback()
        _db.session.remove()

@pytest.fixture
def auth_headers(client):
    """Headers de autenticação para testes."""
    # Criar user e obter token
    response = client.post('/api/v1/auth/register', json={
        'username': 'testuser',
        'email': 'test@example.com',
        'password': 'TestPass123!',
        'role': 'developer'
    })

    login_response = client.post('/api/v1/auth/login', json={
        'username': 'testuser',
        'password': 'TestPass123!'
    })

    token = login_response.json['data']['access_token']
    return {'Authorization': f'Bearer {token}'}
```

Scopes de fixtures:
- `session` : Uma vez por sessão de testes (DB setup)
- `module` : Uma vez por módulo de testes
- `function` : Uma vez por teste (default, máximo isolamento)

Unit Tests

Unit tests focam em **unidades isoladas de código** (funções, métodos, classes) sem dependências externas.

Estrutura de Unit Test

```python
# 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

class TestAuthService:
    """Testes do AuthService."""

    @pytest.fixture
    def mock_user_repo(self):
        """Mock do UserRepository."""
        return Mock(spec=UserRepository)

    @pytest.fixture
    def auth_service(self, mock_user_repo):
        """AuthService com repository mockado."""
        return AuthService(mock_user_repo)

    def test_register_user_success(self, auth_service, mock_user_repo):
        """
        GIVEN valid user data
        WHEN registering a new user
        THEN user is created with hashed password
        """
        # Arrange
        user_data = {
            'username': 'john_doe',
            'email': 'john@example.com',
            'password': 'SecurePass123!',
            'role': 'developer'
        }
        mock_user_repo.get_by_username.return_value = None
        mock_user_repo.get_by_email.return_value = None

        # Act
        result = auth_service.register(user_data)

        # Assert
        assert result is not None
        assert result['username'] == 'john_doe'
        mock_user_repo.create.assert_called_once()
        # Verificar que password foi hashed
        call_args = mock_user_repo.create.call_args[0][0]
        assert 'password' not in call_args  # Password não deve estar em plain text
        assert 'password_hash' in call_args

    def test_register_user_duplicate_username(self, auth_service, mock_user_repo):
        """
        GIVEN existing username
        WHEN registering with same username
        THEN raise ValueError
        """
        # Arrange
        user_data = {'username': 'existing_user', ...}
        mock_user_repo.get_by_username.return_value = Mock()  # User já existe

        # Act & Assert
        with pytest.raises(ValueError, match="Username already exists"):
            auth_service.register(user_data)

    def test_login_success(self, auth_service, mock_user_repo):
```

```
        """
        GIVEN valid credentials
        WHEN logging in
        THEN return tokens
        """
        # Arrange
        mock_user = Mock()
        mock_user.id = 1
        mock_user.username = 'john_doe'
        mock_user.password_hash = bcrypt.hashpw(b'password123', bcrypt.gensalt()).de-
code()
        mock_user.is_active = True

        mock_user_repo.get_by_username.return_value = mock_user

        # Act
        result = auth_service.login('john_doe', 'password123')

        # Assert
        assert 'access_token' in result
        assert 'refresh_token' in result
        assert result['user']['username'] == 'john_doe'

    def test_login_invalid_password(self, auth_service, mock_user_repo):
        """
        GIVEN wrong password
        WHEN logging in
        THEN raise ValueError
        """
        # Arrange
        mock_user = Mock()
        mock_user.password_hash = bcrypt.hashpw(b'correct_password',
bcrypt.gensalt()).decode()
        mock_user_repo.get_by_username.return_value = mock_user

        # Act & Assert
        with pytest.raises(ValueError, match="Invalid credentials"):
            auth_service.login('john_doe', 'wrong_password')
```

Padrão Given-When-Then (Gherkin):
- **GIVEN**: Pré-condições e setup (Arrange)
- **WHEN**: Ação a ser testada (Act)
- **THEN**: Resultado esperado (Assert)

Mocking:
- `unittest.mock.Mock`: Criar objetos fake
- `@patch`: Substituir dependencies temporariamente
- `return_value`: Definir o que mock retorna
- `assert_called_once()`: Verificar que método foi chamado

Integration Tests

Integration tests validam **interações entre componentes**, especialmente endpoints da API com base de dados real (de teste).

Estrutura de Integration Test

```python
# tests/integration/test_project_routes.py
import pytest

class TestProjectRoutes:
    """Testes de integração dos endpoints de projetos."""

    def test_create_project_success(self, client, auth_headers, db):
        """
        GIVEN authenticated user
        WHEN creating a new project
        THEN project is created and returned
        """
        # 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.json['data']
        assert data['name'] == 'Test Project'
        assert data['description'] == 'A test project'
        assert 'id' in data
        assert 'owner_id' in data
        assert 'created_at' in data

    def test_create_project_unauthenticated(self, client, db):
        """
        GIVEN no authentication
        WHEN creating a project
        THEN return 401 Unauthorized
        """
        # Act
        response = client.post(
            '/api/v1/projects',
            json={'name': 'Test'}
        )

        # Assert
        assert response.status_code == 401
        assert 'error' in response.json

    def test_list_projects_with_pagination(self, client, auth_headers, db):
        """
        GIVEN multiple projects
        WHEN listing with pagination
        THEN return paginated results
        """
        # Arrange - Criar 25 projetos
        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=2&per_page=10',
            headers=auth_headers
        )

        # Assert
        assert response.status_code == 200
        data = response.json
        assert len(data['data']) == 10
        assert data['meta']['page'] == 2
        assert data['meta']['per_page'] == 10
        assert data['meta']['total'] == 25
        assert data['meta']['total_pages'] == 3

    def test_update_project_as_owner(self, client, auth_headers, db):
        """
        GIVEN existing project
        WHEN owner updates it
        THEN project is updated
        """
        # Arrange - Criar projeto
        create_response = client.post(
            '/api/v1/projects',
            json={'name': 'Original Name'},
            headers=auth_headers
        )
        project_id = create_response.json['data']['id']

        # Act - Atualizar
        response = client.put(
            f'/api/v1/projects/{project_id}',
            json={'name': 'Updated Name'},
            headers=auth_headers
        )

        # Assert
        assert response.status_code == 200
        assert response.json['data']['name'] == 'Updated Name'

    def test_delete_project_as_non_owner(self, client, auth_headers, db):
        """
        GIVEN project owned by another user
        WHEN non-owner tries to delete
        THEN return 403 Forbidden
        """
        # Arrange - Criar projeto com user1
        create_response = client.post(
            '/api/v1/projects',
            json={'name': 'Project'},
            headers=auth_headers
        )
        project_id = create_response.json['data']['id']

        # Criar user2 e obter auth
        # [código para criar segundo user e auth_headers_user2]

        # Act - User2 tenta eliminar projeto de User1
        response = client.delete(
            f'/api/v1/projects/{project_id}',
            headers=auth_headers_user2
```

```
        )

        # Assert
        assert response.status_code == 403
```

Database de Teste

Testes de integração usam uma **base de dados de teste isolada**:

Configuração (config.py):

```
class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:///:memory:'  # DB in-memory
    # OU
    # SQLALCHEMY_DATABASE_URI = 'postgresql://localhost/test_db'
```

Vantagens de SQLite in-memory:
- Muito rápido (sem I/O de disco)
- Isolado (não afeta dev DB)
- Cleanup automático (desaparece ao terminar)

Desvantagem: SQLite tem algumas diferenças de PostgreSQL. Para testes mais rigorosos, usar PostgreSQL real.

Factory Data para Testes

Factory Boy pode ser usado para criar test data facilmente:

```python
# tests/factories.py
import factory
from src.models import User, Project, Issue
from src.models.base 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 = 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', max_nb_chars=200)
    owner = factory.SubFactory(UserFactory)
    is_active = True

# Uso em testes
def test_with_factory(db):
    user = UserFactory()
    project = ProjectFactory(owner=user)
    assert project.owner == user
```

Cobertura de Testes

Medição de Cobertura

```
# Executar testes com cobertura
pytest --cov=src --cov-report=html --cov-report=term-missing

# Output
---------- coverage: platform linux, python 3.11.0 ----------
Name                            Stmts   Miss  Cover   Missing
---------------------------------------------------------------
src/__init__.py                     0      0   100%
src/services/auth_service.py       85      8    91%   45-47, 102-105
src/services/project_service.py    112     15    87%   78-82, 156-160
src/routes/projects.py              95     12    87%   145-150, 203-206
---------------------------------------------------------------
TOTAL                             1542    185    88%
```

Objetivos de Cobertura

Módulo	Target	Justificação
Services	>= 80%	Lógica de negócio crítica
Routes	>= 70%	Integração e validação
Repositories	>= 70%	Queries e data access
Models	>= 60%	Principalmente declarativo
Utils	>= 80%	Funções auxiliares críticas

Nota: Cobertura de 100% não é objetivo. Alguns códigos (error handling excepcional, código defensivo) são difíceis de testar e têm ROI baixo.

Boas Práticas

Nomenclatura de Testes

```
def test_<action>_<scenario>_<expected_result>():
    ...

# Exemplos:
def test_register_user_with_valid_data_returns_user():
def test_login_with_wrong_password_raises_error():
def test_create_project_without_auth_returns_401():
```

Isolamento

Cada teste deve ser **completamente independente**:

```
# ❌ Ruim - testes dependentes
def test_create_user():
    user = create_user('john')
    assert user.id == 1

def test_get_user():
    user = get_user(1)  # Assume que test_create_user executou
    assert user.username == 'john'

# ✅ Bom - testes independentes
def test_create_user(db):
    user = UserFactory()
    assert user.id is not None

def test_get_user(db):
    user = UserFactory(username='john')
    found = User.query.get(user.id)
    assert found.username == 'john'
```

Cleanup Entre Testes

Fixture `db` garante rollback após cada teste:

```
@pytest.fixture(scope='function')
def db(app):
    with app.app_context():
        _db.session.begin_nested()  # Savepoint
        yield _db
        _db.session.rollback()  # Rollback para savepoint
        _db.session.remove()
```

Evitar Flaky Tests

Flaky tests são testes que falham intermitentemente.

Causas comuns:
- Dependência de timestamps atuais
- Dependência de ordem de execução
- Dependência de estado externo (rede, filesystem)
- Concorrência issues

Soluções:

```
# ❌ Flaky - timestamp
def test_timestamp():
    user = create_user()
    assert user.created_at == datetime.utcnow()  # Pode falhar por milissegundos

# ✅ Não flaky
def test_timestamp():
    before = datetime.utcnow()
    user = create_user()
    after = datetime.utcnow()
    assert before <= user.created_at <= after
```

Executar Testes

Comandos Básicos

```
# Todos os testes
pytest

# Testes de um módulo específico
pytest tests/unit/test_auth_service.py

# Um teste específico
pytest tests/unit/test_auth_service.py::TestAuthService::test_register_user_success

# Com cobertura
pytest --cov=src --cov-report=html

# Verbose (mais output)
pytest -v

# Stop na primeira falha
pytest -x

# Mostrar print statements
pytest -s

# Executar em paralelo (requer pytest-xdist)
pytest -n auto
```

CI/CD Integration

Testes executam automaticamente no CI (GitHub Actions):

```
- name: Run tests with coverage
  env:
    DATABASE_URL: postgresql://postgres:postgres@localhost:5432/test_db
  run: |
    pytest --cov=src --cov-report=xml --cov-fail-under=70
```

`--cov-fail-under=70` : Pipeline falha se cobertura < 70%

\newpage

CI/CD e Deploy

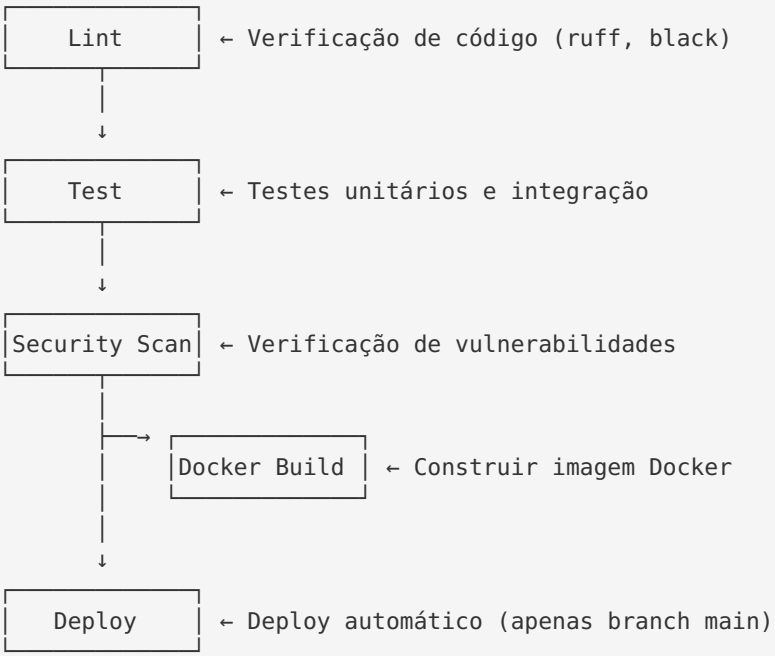
O Issue Tracker API implementa um pipeline CI/CD completo usando **GitHub Actions**, com deploy automatizado para **Render**. Este pipeline garante que apenas código de qualidade chega a produção.

GitHub Actions Pipeline

Visão Geral

O pipeline é ativado automaticamente em:
- **Push** para branches `main` ou `develop`
- **Pull Requests** para `main` ou `develop`

Jobs do Pipeline:



Job 1: Lint & Format

Garante consistência e qualidade do código.

Ferramentas:
- **ruff**: Linter moderno e rápido (substitui flake8, pylint)
- **black**: Code formatter automático (PEP 8)

```yaml
lint:
  name: Lint and Code Quality
  runs-on: ubuntu-latest

  steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Set up Python
    uses: actions/setup-python@v5
    with:
      python-version: '3.11'

  - name: Install dependencies
    run: |
      pip install ruff black

  - name: Run Ruff linter
    run: |
      ruff check src/ --output-format=github
    continue-on-error: true

  - name: Check code formatting with Black
    run: |
      black --check src/
    continue-on-error: true
```

Verificações do Ruff:
- Unused imports
- Undefined variables
- Syntax errors
- Code complexity
- PEP 8 violations
- Security issues (básico)

Black Formatting:
- Line length: 88 caracteres (default)
- Consistência absoluta
- Sem debates sobre style

Job 2: Tests

Executa todos os testes com cobertura.

```yaml
test:
  name: Run Tests
  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
        --health-timeout 5s
        --health-retries 5

  steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Set up Python
    uses: actions/setup-python@v5
    with:
      python-version: '3.11'

  - name: Install dependencies
    run: |
      pip install -r requirements.txt
      pip install -r requirements-dev.txt

  - name: Run tests with coverage
    env:
      DATABASE_URL: postgresql://postgres:postgres@localhost:5432/test_db
      FLASK_ENV: testing
      SECRET_KEY: test-secret-key
      JWT_SECRET_KEY: test-jwt-secret-key
    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
      flags: unittests
      fail_ci_if_error: false
```

PostgreSQL Service Container:
- Fornece DB real para integration tests
- Health checks garantem que DB está pronto
- Isolado (não afeta nada)

Falha do Pipeline:
- Se qualquer teste falhar
- Se cobertura < 70% (`--cov-fail-under=70`)

Job 3: Security Scan

Verifica vulnerabilidades de segurança.

```yaml
security-scan:
  name: Security Scan
  runs-on: ubuntu-latest
  needs: lint

  steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Set up Python
    uses: actions/setup-python@v5
    with:
      python-version: '3.11'

  - name: Install dependencies
    run: |
      pip install safety bandit

  - name: Run Safety check (dependency vulnerabilities)
    run: |
      safety check --json || true
    continue-on-error: true

  - name: Run Bandit (security linting)
    run: |
      bandit -r src/ -f json -o bandit-report.json || true
    continue-on-error: true

  - name: Upload Bandit report
    uses: actions/upload-artifact@v3
    with:
      name: bandit-report
      path: bandit-report.json
    if: always()
```

Safety:
- Verifica vulnerabilidades em dependencies (CVEs conhecidas)
- Compara contra base de dados de vulnerabilidades
- Exemplo: detecta se usando versão antiga do Flask com vulnerabilidade

Bandit:
- Static analysis de código Python para security issues
- Detecta: hardcoded passwords, SQL injection, uso inseguro de random, etc.

Job 4: Docker Build

Valida que imagem Docker constrói corretamente.

```yaml
build:
  name: Build Docker Image
  runs-on: ubuntu-latest
  needs: test

  steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Set up Docker Buildx
    uses: docker/setup-buildx-action@v3

  - name: Build Docker 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 Docker image
    run: |
      docker run --rm issue-tracker-api:${{ github.sha }} \
        python -c "from src.app import create_app; app = create_app(); print('App
created successfully')"
```

Cache:

- `cache-from/cache-to`: Usa GitHub Actions cache para acelerar builds
- Layers são reutilizadas entre builds

Job 5: Deploy

Deploy automático para Render (apenas branch `main`).

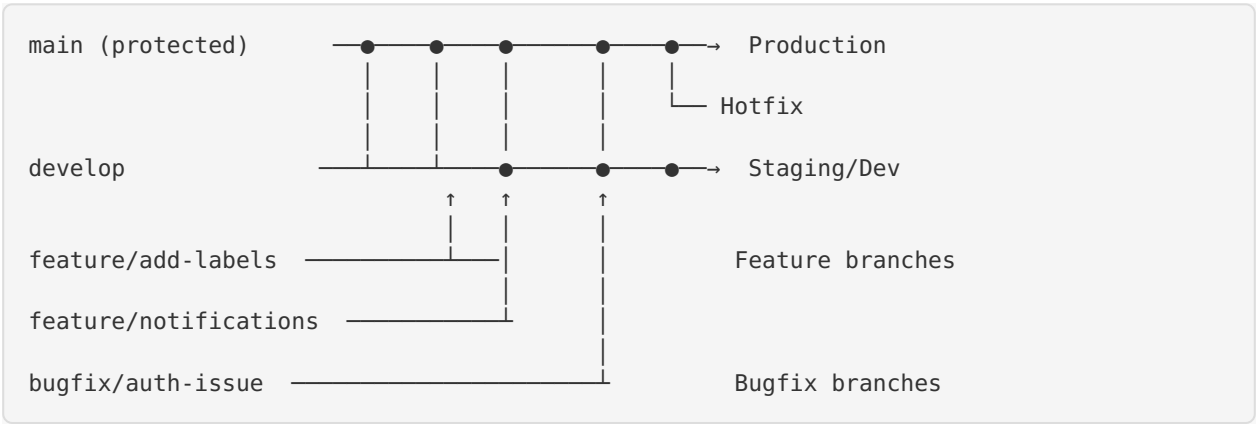
```yaml
deploy:
  name: Deploy to Render
  runs-on: ubuntu-latest
  needs: build
  if: github.ref == 'refs/heads/main' && github.event_name == 'push'

  steps:
  - name: Trigger Render deployment
    run: |
      echo "Deployment to Render is triggered automatically via webhook"
      # Alternativamente, pode usar deploy hook:
      # curl -X POST ${{ secrets.RENDER_DEPLOY_HOOK_URL }}
```

Render Auto-Deploy:

- Render detecta push para `main` automaticamente
- Puxa código mais recente
- Executa build
- Executa migrations
- Substitui instância antiga por nova (zero-downtime)

Branching Strategy

Modelo de Branches



Branches principais:

1. **main**: Código em produção
   - **Protegido**: Requer PR + checks passing
   - **Deploy automático**: Cada push vai para produção
   - **Sempre estável**: Não deve quebrar nunca

2. **develop**: Branch de desenvolvimento
   - Integração de features
   - Testing mais extensivo
   - Não protegido (mais flexível)

3. **feature/***: Features novas
   - Criadas a partir de `develop`
   - Merged de volta para `develop` via PR
   - Exemplo: `feature/add-notifications`

4. **bugfix/***: Correções de bugs
   - Criadas a partir de `develop`
   - Merged de volta para `develop` via PR
   - Exemplo: `bugfix/fix-auth-token-expiry`

5. **hotfix/***: Correções urgentes para produção
   - Criadas a partir de `main`
   - Merged para `main` E `develop`
   - Exemplo: `hotfix/security-vulnerability`

Pull Request Workflow

```
1. Developer cria feature branch
   git checkout -b feature/my-feature develop

2. Developer faz commits
   git commit -m "feat: implement my feature"

3. Developer push para remoto
   git push origin feature/my-feature

4. Developer abre Pull Request para develop
   - Preenche template de PR
   - Adiciona descrição e screenshots se aplicável

5. CI Pipeline executa automaticamente
   - Lint ✓
   - Tests ✓
   - Security Scan ✓
   - Docker Build ✓

6. Code Review
   - Outro developer review código
   - Comentários e sugestões
   - Request changes ou Approve

7. Developer atualiza código baseado em feedback
   git commit -m "fix: address PR feedback"
   git push origin feature/my-feature

8. PR é aprovado e merged
   - Squash and merge (limpar histórico)
   - Feature branch é eliminada automaticamente

9. Se PR para main: Deploy automático acontece
```

Branch Protection Rules (main)

Configuração no GitHub:

```
Protected Branch: main
✓ Require pull request reviews before merging
  - Required approvals: 1
✓ Require status checks to pass before merging
  - lint
  - test
  - security-scan
  - build
✓ Require branches to be up to date before merging
✓ Require linear history
✓ Include administrators
✗ Allow force pushes
✗ Allow deletions
```

Deploy no Render

Configuração (render.yaml)

Infrastructure as Code para Render:

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

  # Web Service (API)
  - type: web
    name: issue-tracker-api
    env: python
    region: oregon
    plan: free
    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"
    healthCheckPath: /api/v1/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
```

Componentes:

1. **PostgreSQL Managed Database**
   - Backups automáticos
   - Connection pooling
   - SSL connections
   - Monitoring

2. **Web Service**
   - Python runtime
   - Gunicorn WSGI server (4 workers)
   - Auto-scaling (dentro dos limites do plano)
   - Health checks (HTTP GET /api/v1/health)

Processo de Deploy

```
1. Developer push para main
   git push origin main

2. GitHub Actions pipeline executa
   Lint → Test → Security → Build → Deploy (triggered)

3. Render detecta novo commit
   - Webhook do GitHub notifica Render

4. Render inicia build
   - Clone do repositório
   - Instalar dependencies (pip install -r requirements.txt)
   - Build completo (~2-3 minutos)

5. Render executa migrations
   - alembic upgrade head
   - Aplicar alterações no schema

6. Render inicia nova instância
   - gunicorn 'src.app:create_app()'
   - Health check loop

7. Quando health check passa
   - Traffic redirecionado para nova instância
   - Instância antiga é terminada
   - Zero-downtime deploy ✓

8. Logs e monitoring disponíveis no dashboard
```

Variáveis de Ambiente (Produção)

Configuradas no Render Dashboard:

Variável	Origem	Descrição
`DATABASE_URL`	Auto (from DB)	PostgreSQL connection string
`SECRET_KEY`	Auto-generated	Flask secret key (sessions)
`JWT_SECRET_KEY`	Auto-generated	JWT signing key
`FLASK_ENV`	Manual	`production`
`LOG_LEVEL`	Manual	`INFO`
`LOG_FORMAT`	Manual	`json`
`CORS_ORIGINS`	Manual	Lista de domínios permitidos
`JWT_ACCESS_TOKEN_EXPIRES`	Manual	`900` (15 min)
`JWT_REFRESH_TOKEN_EXPIRES`	Manual	`604800` (7 dias)

Segurança:
- Secrets são encriptados
- Não visíveis em logs
- Apenas aplicação tem acesso

Rollback

Rollback via UI (Render Dashboard):
1. Aceder a Render Dashboard
2. Selecionar service “issue-tracker-api”
3. Tab “Events” → lista de deploys
4. Selecionar deploy anterior (working)
5. Click “Rollback” → restaura código e configuração

Rollback de Migrations:

Se migration causou problemas:

```
# Ligar ao shell da aplicação no Render
render shell issue-tracker-api

# Reverter última migration
alembic downgrade -1

# Ou reverter para versão específica
alembic downgrade <revision_id>
```

Boas Práticas:
- **Backup de DB antes de migrations críticas**
`bash`
`pg_dump $DATABASE_URL > backup_pre_migration.sql`
- **Migrations reversíveis**: Implementar `downgrade()` em cada migration
- **Testing de migrations**: Testar em staging antes de produção

Monitoring e Logs

Logs Estruturados (JSON):

```
import logging
import json

class JSONFormatter(logging.Formatter):
    def format(self, record):
        log_data = {
            'timestamp': self.formatTime(record),
            'level': record.levelname,
            '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)
```

Vantagens:
- Fácil parsing e filtering

- Integração com log aggregators (Datadog, Loggly)
- Searchable

Métricas no Render:
- **CPU usage**: % de utilização
- **Memory usage**: RAM consumida
- **Request rate**: Requests por segundo
- **Response time**: Latência média
- **Error rate**: % de erros (5xx)

Alertas (Futuro):
- Email quando CPU > 80%
- Email quando error rate > 5%
- Email quando health check falha

Health Check Endpoint

```python
@health_bp.route('/health', methods=['GET'])
def health_check():
    """Health check endpoint para monitoring."""
    try:
        # Verificar conexão à DB
        db.session.execute('SELECT 1')
        db_status = 'connected'
    except Exception:
        db_status = 'disconnected'
        return jsonify({
            'status': 'unhealthy',
            'database': db_status
        }), 503

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

Render Health Checks:
- GET request a /api/v1/health
- Intervalo: 30 segundos
- Timeout: 5 segundos
- Falhas toleradas: 3 consecutivas

Se health check falhar 3x: Render reinicia instância automaticamente.

\newpage

Como Correr o Projeto

Este capítulo fornece instruções detalhadas para correr o Issue Tracker API localmente (com e sem Docker) e em produção (Render).

Prerequisites

Software Necessário

Software	Versão Mínima	Notas
Python	3.11+	Usar pyenv para gerir versões
PostgreSQL	13+	Ou Docker para evitar instalação local
Git	2.30+	Para clonar repositório
Docker (opcional)	20.10+	Para desenvolvimento containerizado
Docker Compose (opcional)	2.0+	Incluído no Docker Desktop

Verificar Versões

```
python --version          # Python 3.11.x
psql --version           # PostgreSQL 13.x ou superior
git --version            # git version 2.x
docker --version         # Docker version 20.x (opcional)
docker-compose --version # Docker Compose version 2.x (opcional)
```

Setup Inicial (Comum a Todos)

1. Clonar Repositório

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

# Ou SSH
git clone git@github.com:your-username/issue-tracker-api.git
cd issue-tracker-api
```

2. Criar Ficheiro .env

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

Editar .env com suas configurações:

```
# .env (desenvolvimento local)

# Flask
FLASK_ENV=development
SECRET_KEY=your-super-secret-key-change-me
DEBUG=True

# Database
DATABASE_URL=postgresql://postgres:password@localhost:5432/issue_tracker

# JWT
JWT_SECRET_KEY=your-jwt-secret-key-change-me
JWT_ACCESS_TOKEN_EXPIRES=900      # 15 minutos
JWT_REFRESH_TOKEN_EXPIRES=604800  # 7 dias

# Logging
LOG_LEVEL=DEBUG
LOG_FORMAT=text

# CORS (desenvolvimento permite todos)
CORS_ORIGINS=*

# Rate Limiting
RATELIMIT_ENABLED=True
RATELIMIT_DEFAULT=100 per minute
```

Gerar secrets seguros:

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

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

Opção 1: Docker Compose (Recomendado para Desenvolvimento)

Vantagens

- ✅ Setup mais simples
- ✅ Sem necessidade de instalar PostgreSQL localmente
- ✅ Ambiente consistente
- ✅ Fácil cleanup

Passos

1. Iniciar Containers

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

Serviços iniciados:
- **db**: PostgreSQL 15
- **api**: Flask application

2. Verificar Containers

```
docker-compose ps

# Output esperado:
# NAME                  STATUS
# issue-tracker-db      Up (healthy)
# issue-tracker-api     Up
```

3. Ver Logs

```
# Logs de ambos os serviços
docker-compose logs -f

# Logs apenas da API
docker-compose logs -f api

# Logs apenas do DB
docker-compose logs -f db
```

4. Executar Migrations

Migrations devem executar automaticamente no startup, mas se necessário:

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

5. Criar Dados de Exemplo (Opcional)

```
# Entrar no container
docker-compose exec api bash

# Abrir Python shell
python

# Criar admin user
from src.app import create_app
from src.models.base import db
from src.models.user import User
import bcrypt

app = create_app()
with app.app_context():
    admin = User(
        username='admin',
        email='admin@example.com',
        password_hash=bcrypt.hashpw(b'admin123', bcrypt.gensalt(12)).decode(),
        role='admin',
        is_active=True
    )
    db.session.add(admin)
    db.session.commit()
    print(f"Admin user created: {admin.id}")
```

6. Aceder à API

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

Testar health check:

```
curl http://localhost:5000/api/v1/health

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

7. Parar Containers

```
# Parar (preserva dados)
docker-compose stop

# Parar e remover containers (preserva volumes/dados)
docker-compose down

# Parar e remover tudo incluindo volumes (CUIDADO: elimina dados)
docker-compose down -v
```

Opção 2: Setup Local (Sem Docker)

Vantagens

- ✅ Desenvolvimento mais rápido (sem rebuild de containers)
- ✅ Debugging mais fácil
- ✅ Acesso direto ao código

Passos

1. Instalar PostgreSQL

Ubuntu/Debian:

```
sudo apt update
sudo apt install postgresql postgresql-contrib
sudo systemctl start postgresql
sudo systemctl enable postgresql
```

macOS (Homebrew):

```
brew install postgresql@15
brew services start postgresql@15
```

Windows:
- Descarregar installer de https://www.postgresql.org/download/windows/

2. Criar Base de Dados

```
# Aceder ao PostgreSQL
sudo -u postgres psql

# No prompt do PostgreSQL:
CREATE DATABASE issue_tracker;
CREATE USER issue_tracker_user WITH PASSWORD 'your_password';
GRANT ALL PRIVILEGES ON DATABASE issue_tracker TO issue_tracker_user;
\q
```

Atualizar DATABASE_URL no .env:

```
DATABASE_URL=postgresql://issue_tracker_user:your_password@localhost:5432/is-
sue_tracker
```

3. Criar Virtual Environment

```
# Criar venv
python -m venv venv

# Ativar venv
# Linux/macOS:
source venv/bin/activate

# Windows:
venv\Scripts\activate
```

4. Instalar Dependencies

```
# Dependencies de produção
pip install -r requirements.txt

# Dependencies de desenvolvimento (para testes, linting)
pip install -r requirements-dev.txt
```

5. Executar Migrations

```
# Inicializar Alembic (se ainda não inicializado)
alembic upgrade head
```

6. Executar Aplicação

Desenvolvimento (Flask dev server):

```
# Com auto-reload
flask --app src.app:create_app run --debug --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 \
  --workers 4 \
  --reload \
  --log-level debug
```

7. Verificar

```
curl http://localhost:5000/api/v1/health
```

Executar Testes

Todos os Testes

```
# Ativar venv (se setup local)
source venv/bin/activate

# Executar testes
pytest

# Com verbose
pytest -v

# Com cobertura
pytest --cov=src --cov-report=html --cov-report=term
```

Testes Específicos

```
# Unit tests apenas
pytest tests/unit/

# Integration tests apenas
pytest tests/integration/

# Teste específico
pytest tests/unit/test_auth_service.py::TestAuthService::test_register_user_success
```

Ver Cobertura (HTML Report)

```
pytest --cov=src --cov-report=html

# Abrir no browser
# Linux/macOS:
open htmlcov/index.html

# Windows:
start htmlcov/index.html
```

Quickstart Script

Para facilitar, um script `quickstart.sh` está incluído:

```
#!/bin/bash

echo "Issue Tracker API - Quickstart"

# Verificar se .env existe
if [ ! -f .env ]; then
    echo "Criando .env a partir de .env.example..."
    cp .env.example .env
    echo "⚠️  Edite .env e configure suas variáveis!"
    exit 1
fi

# Opção: Docker ou Local
echo "Escolha método de setup:"
echo "1. Docker Compose (recomendado)"
echo "2. Setup Local"
read -p "Escolha (1 ou 2): " choice

if [ "$choice" == "1" ]; then
    echo "Iniciando com Docker Compose..."
    docker-compose up -d
    echo "✅ API disponível em http://localhost:5000"
    echo "📊 Ver logs: docker-compose logs -f"
else
    echo "Setup local..."

    # Criar venv
    if [ ! -d "venv" ]; then
        python -m venv venv
    fi

    source venv/bin/activate

    # Instalar deps
    pip install -r requirements.txt
    pip install -r requirements-dev.txt

    # Migrations
    alembic upgrade head

    # Executar
    echo "✅ Setup completo!"
    echo "Execute: source venv/bin/activate && flask --app src.app:create_app run --
debug"
fi
```

Uso:

```
chmod +x quickstart.sh
./quickstart.sh
```

Troubleshooting

Problema: Port 5000 já em uso

Solução:

```
# Encontrar processo usando porta 5000
# Linux/macOS:
lsof -i :5000

# Windows:
netstat -ano | findstr :5000

# Matar processo
kill -9 <PID>

# Ou usar porta diferente
flask --app src.app:create_app run --port 5001
```

Problema: PostgreSQL connection refused

Verificações:

```
# Verificar se PostgreSQL está a correr
# Linux:
sudo systemctl status postgresql

# macOS:
brew services list | grep postgresql

# Verificar se pode conectar
psql -U postgres -h localhost
```

Soluções:
- Iniciar PostgreSQL: `sudo systemctl start postgresql`
- Verificar credenciais em DATABASE_URL
- Verificar que DB foi criada: `psql -U postgres -l`

Problema: Alembic migrations fail

Erro comum: "Target database is not up to date"

Solução:

```
# Ver estado atual
alembic current

# Ver histórico
alembic history

# Forçar para latest
alembic stamp head
alembic upgrade head
```

Problema: Import errors

Erro: `ModuleNotFoundError: No module named 'src'`

Solução:

```
# Garantir que está no diretório raiz do projeto
pwd  # Deve mostrar .../issue-tracker-api

# Garantir que venv está ativado
which python  # Deve mostrar .../venv/bin/python

# Reinstalar dependencies
pip install -r requirements.txt
```

Deploy em Produção (Render)

Prerequisites

1. Conta no GitHub
2. Repositório pushed para GitHub
3. Conta no Render (gratuita): https://render.com

Passos

1. Push para GitHub

```
git add .
git commit -m "feat: prepare for deploy"
git push origin main
```

2. Criar Conta Render

Aceder https://render.com e criar conta (pode usar GitHub login).

3. Criar PostgreSQL Database

1. Dashboard → **New +** → **PostgreSQL**
2. **Name**: `issue-tracker-db`
3. **Database**: `issue_tracker`
4. **User**: `issue_tracker_user`
5. **Region**: Escolher região próxima
6. **Plan**: Free
7. **Create Database**

Esperar ~2 minutos até DB estar disponível.

4. Criar Web Service

1. Dashboard → **New +** → **Web Service**
2. **Connect Repository**: Selecionar seu repositório GitHub
3. **Name**: `issue-tracker-api`
4. **Environment**: Python
5. **Branch**: `main`
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`

5. Configurar Environment Variables

No dashboard do Web Service, **Environment** tab:

Key	Value
`FLASK_ENV`	`production`
`DATABASE_URL`	Selecionar: From Database → issue-tracker-db
`SECRET_KEY`	Click “Generate”
`JWT_SECRET_KEY`	Click “Generate”
`JWT_ACCESS_TOKEN_EXPIRES`	`900`
`JWT_REFRESH_TOKEN_EXPIRES`	`604800`
`LOG_LEVEL`	`INFO`
`LOG_FORMAT`	`json`
`CORS_ORIGINS`	`*` (ou seu domínio)

6. Deploy

Click **Create Web Service** → Deploy inicia automaticamente.

Acompanhar logs em tempo real. Deploy completo: ~3-5 minutos.

7. Testar

Quando deploy terminar, API estará disponível em:

```
https://issue-tracker-api-<random>.onrender.com
```

Testar:

```
curl https://issue-tracker-api-<random>.onrender.com/api/v1/health

# Resposta esperada:
{
  "status": "healthy",
  "version": "1.0.0",
  "database": "connected",
  "timestamp": "2026-02-20T..."
}
```

8. Configurar Auto-Deploy

Render deteta automaticamente pushes para `main` e faz deploy.

Para desativar: **Settings** → **Auto-Deploy** → Disable

9. Custom Domain (Opcional)

Settings → **Custom Domains** → Adicionar seu domínio.

Instruções para configurar DNS serão fornecidas.

Monitoring em Produção

Logs:
- Dashboard → Service → **Logs** tab
- Real-time streaming
- Searchable

Metrics:
- Dashboard → Service → **Metrics** tab
- CPU, Memory, Requests

Alertas (Email):
- Settings → **Alerts**
- Notificações quando serviço fica down

\newpage

Limitações e Melhorias Futuras

Limitações Atuais

O Issue Tracker API, sendo um projeto de portfólio para nível júnior, tem algumas limitações intencionais e outras que representam oportunidades de evolução.

1. Sem Frontend

Limitação:
- Apenas API backend
- Sem interface gráfica para utilizadores finais
- Testing manual via cURL/Postman

Impacto:
- Menos visível para recrutadores não-técnicos
- Requer conhecimento técnico para testar

Mitigação Atual:
- Documentação API extensa com exemplos
- Postman collection (futuro)

2. Autenticação Básica

Limitações:
- Apenas username/password authentication
- Sem OAuth2 / Social Login (Google, GitHub)
- Sem Two-Factor Authentication (2FA)
- Sem password reset via email
- Token revocation não implementado (sem blacklist)

Impacto:
- Menos seguro que sistemas enterprise
- Experiência de utilizador limitada

3. Sem Notificações

Limitações:
- Sem notificações por email
- Sem notificações in-app
- Sem webhooks para integrações externas
- Utilizadores não são notificados de mudanças relevantes

Impacto:
- Utilizadores devem verificar manualmente por updates
- Menos engagement

4. Sem File Uploads

Limitações:
- Não é possível anexar ficheiros a issues
- Sem suporte para imagens, documentos, screenshots
- Descrições apenas em text

Impacto:
- Menos útil para casos de uso reais (bugs reports sem screenshots)

5. Search Limitado

Limitações:
- Apenas filtering básico (status, priority, assignee)
- Sem full-text search em títulos/descrições
- Sem search avançado com operators

Impacto:
- Difícil encontrar issues específicas em projetos grandes

6. Sem Real-Time Updates

Limitações:
- Sem WebSockets
- Sem Server-Sent Events
- Updates requerem polling

Impacto:
- Experiência não é real-time
- Maior latência para ver mudanças

7. Sem Audit Logs

Limitações:
- Não rastreia quem fez o quê e quando
- Sem histórico de mudanças (changelog)
- Timestamps apenas (created_at, updated_at)

Impacto:
- Difícil auditoria
- Impossível ver evolução de uma issue

8. Soft Deletes Não Implementados

Limitação:
- Deletes são hard deletes (permanentes)
- Não é possível recuperar dados eliminados acidentalmente

Impacto:
- Risco de perda de dados
- Sem “undo” para deletes

Roadmap de Melhorias

Short-term (3-6 meses)

1. Frontend (React)

Objetivo: Interface gráfica profissional

Features:
- Dashboard com overview de projetos e issues
- Kanban board para issues (drag-and-drop)
- Forms para criar/editar recursos
- Search e filtering avançado
- Responsive design (mobile-friendly)

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

Esforço: ~40 horas

2. Password Reset via Email

Objetivo: Permitir recovery de passwords esquecidas

Features:
- Endpoint POST /api/v1/auth/forgot-password
- Gerar token temporário (15-30 min)
- Enviar email com link de reset
- Endpoint POST /api/v1/auth/reset-password

Stack Sugerida:
- SendGrid ou Mailgun (email service)
- Token armazenado em Redis (expiração automática)

Esforço: ~8 horas

3. Email Notifications

Objetivo: Notificar utilizadores de eventos relevantes

Eventos:
- Issue atribuída a utilizador
- Novo comentário em issue que sigo

- Issue movida para “resolved”
- Adicionado a projeto

Implementação:
- Background tasks com Celery + Redis
- Email templates (HTML)
- Preferências de notificação por utilizador

Esforço: ~16 horas

4. Token Blacklist (Logout Real)

Objetivo: Permitir revocação de tokens antes de expirar

Implementação:
- Redis para armazenar tokens revogados
- TTL igual à expiração do token
- Verificar blacklist em cada request

Esforço: ~4 horas

Medium-term (6-12 meses)

5. OAuth2 / Social Login

Objetivo: Login com Google, GitHub, Microsoft

Features:
- OAuth2 flow completo
- Link/unlink social accounts
- Criar conta automaticamente no primeiro login

Stack Sugerida:
- Authlib (OAuth library)
- Provider-specific SDKs

Esforço: ~20 horas

6. File Uploads (Attachments)

Objetivo: Anexar ficheiros a issues

Features:
- Upload de imagens, PDFs, documentos
- Preview de imagens inline
- Limite de tamanho (ex: 10MB por ficheiro)
- Validação de tipos de ficheiro

Implementação:
- Object storage (S3, Cloudinary)
- Tabela `attachments` no DB
- Pre-signed URLs para downloads seguros

Esforço: ~16 horas

7. Full-Text Search

Objetivo: Search rápido em títulos e descrições

Implementação Simples:
- PostgreSQL Full-Text Search
- Índices GIN em campos text
- Ranking de resultados

Implementação Avançada:
- Elasticsearch
- Autocomplete
- Fuzzy matching

Esforço: 8 horas (PostgreSQL) ou 24 horas (Elasticsearch)

8. Soft Deletes

Objetivo: Recuperar recursos eliminados acidentalmente

Implementação:
- Adicionar coluna `deleted_at` (nullable)
- Queries devem filtrar `WHERE deleted_at IS NULL`
- Endpoint para listar recursos eliminados
- Endpoint para restaurar: `POST /api/v1/projects/{id}/restore`

Esforço: ~12 horas (impacta todos os modelos e queries)

Long-term (12+ meses)

9. WebSockets para Real-Time

Objetivo: Updates em tempo real

Features:
- Ver quando outros utilizadores editam issue
- Notificações in-app instantâneas
- Indicador “X está a escrever comentário”

Stack Sugerida:
- Socket.IO (Python: python-socketio)
- Redis para pub/sub entre workers

Esforço: ~32 horas

10. Two-Factor Authentication (2FA)

Objetivo: Segurança adicional para contas

Features:
- TOTP (Time-based OTP) com Google Authenticator
- Backup codes
- SMS OTP (via Twilio)

Esforço: ~24 horas

11. Audit Logs

Objetivo: Rastrear todas as ações

Features:
- Tabela `audit_logs`

- Registar: user, action, resource, timestamp, changes
- Endpoint GET /api/v1/audit-logs (admin only)
- Filtrar por user, resource, date range

Esforço: ~16 horas

12. API Rate Limiting por User

Objetivo: Rate limiting mais granular

Implementação Atual:
- Rate limiting por IP (100 req/min)

Melhoramento:
- Rate limiting por user_id
- Limites diferentes por role (admin > developer > viewer)
- Armazenar contadores em Redis

Esforço: ~8 horas

13. Metrics e Observability

Objetivo: Monitoring profissional

Features:
- Prometheus metrics (request count, latency, errors)
- Grafana dashboards
- Distributed tracing (Jaeger)
- Error tracking (Sentry)

Stack Sugerida:
- prometheus-flask-exporter
- Sentry SDK
- Grafana Cloud (free tier)

Esforço: ~20 horas

14. Caching Layer

Objetivo: Performance para reads frequentes

Implementação:
- Redis para cache
- Cache em listagens (projetos, issues)
- Cache invalidation em updates
- TTL apropriado (5-10 min)

Esforço: ~12 horas

15. GraphQL API

Objetivo: API alternativa mais flexível

Vantagens sobre REST:
- Cliente pede apenas dados necessários
- Menos requests (resolve N+1 problem)
- Strongly typed schema

Stack Sugerida:
- Graphene-Python
- GraphQL Playground

Esforço: ~40 horas (reimplementar endpoints)

16. Microservices (Se Necessário)

Quando considerar:
- Sistema cresceu muito
- Equipas diferentes trabalham em features diferentes
- Necessidade de escalar componentes independentemente

Serviços potenciais:
- **Auth Service**: Autenticação, users
- **Project Service**: Projetos, membros
- **Issue Service**: Issues, comments, labels
- **Notification Service**: Emails, webhooks
- **Search Service**: Full-text search, indexing

Stack Sugerida:
- Kubernetes (orquestração)
- gRPC (comunicação inter-service)
- API Gateway (Kong, Nginx)
- Service mesh (Istio)

Esforço: 200+ horas (reestruturação completa)

Melhorias de Code Quality

Adicionar Type Hints Completos

```
# Atual (parcial)
def create_project(data):
    ...

# Futuro (completo)
def create_project(data: ProjectCreateSchema) -> ProjectModel:
    ...
```

Docstrings Completas (Google Style)

```python
def create_project(data: ProjectCreateSchema) -> ProjectModel:
    """Create a new project.

    Args:
        data: Project creation data including name and description.

    Returns:
        The created project model instance.

    Raises:
        ValueError: If project name already exists.
        PermissionError: If user doesn't have permission to create projects.

    Example:
        >>> data = ProjectCreateSchema(name="My Project", description="...")
        >>> project = create_project(data)
        >>> print(project.id)
        1
    """
    ...
```

Pre-commit Hooks

Garantir qualidade antes de commit:

```yaml
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/psf/black
    hooks:
      - id: black

  - repo: https://github.com/charliermarsh/ruff-pre-commit
    hooks:
      - id: ruff

  - repo: https://github.com/pre-commit/pre-commit-hooks
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-json
```

\newpage

Referências

Documentação Oficial

Python e Frameworks

1. **Python**
   - Site: https://www.python.org/

- Docs: https://docs.python.org/3/
- PEP 8 (Style Guide): https://peps.python.org/pep-0008/

2. **Flask**
   - Site: https://flask.palletsprojects.com/
   - Docs: https://flask.palletsprojects.com/en/3.0.x/
   - Tutorial: https://flask.palletsprojects.com/en/3.0.x/tutorial/

3. **SQLAlchemy**
   - Site: https://www.sqlalchemy.org/
   - Docs: https://docs.sqlalchemy.org/en/20/
   - ORM Tutorial: https://docs.sqlalchemy.org/en/20/tutorial/

4. **Alembic**
   - Docs: https://alembic.sqlalchemy.org/
   - Tutorial: https://alembic.sqlalchemy.org/en/latest/tutorial.html

5. **Marshmallow**
   - Site: https://marshmallow.readthedocs.io/
   - Quickstart: https://marshmallow.readthedocs.io/en/stable/quickstart.html

Testing

1. **pytest**
   - Site: https://docs.pytest.org/
   - Docs: https://docs.pytest.org/en/stable/
   - Fixtures: https://docs.pytest.org/en/stable/fixture.html

2. **pytest-cov**
   - Docs: https://pytest-cov.readthedocs.io/

Authentication & Security

1. **PyJWT**
   - Site: https://pyjwt.readthedocs.io/
   - Docs: https://pyjwt.readthedocs.io/en/stable/

2. **bcrypt**
   - Docs: https://github.com/pyca/bcrypt/

3. **Flask-JWT-Extended**

   ◦ Docs: https://flask-jwt-extended.readthedocs.io/

Database

1. **PostgreSQL**
   ◦ Site: https://www.postgresql.org/
   ◦ Docs: https://www.postgresql.org/docs/

Boas Práticas e Standards

Security

1. **OWASP Top 10 (2021)**
   - Site: https://owasp.org/www-project-top-ten/
   - Descrição: Lista dos 10 riscos de segurança mais críticos em aplicações web
2. **OWASP Password Storage Cheat Sheet**
   - URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
   - Guia: Como armazenar passwords de forma segura
3. **JWT Best Current Practices (RFC 8725)**
   - URL: https://datatracker.ietf.org/doc/html/rfc8725
   - Descrição: Recomendações oficiais para uso seguro de JWT

API Design

1. **REST API Design Best Practices**
   - Microsoft: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design
   - Google: https://cloud.google.com/apis/design
2. **HTTP Status Codes**
   - MDN: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
   - RFC 7231: https://datatracker.ietf.org/doc/html/rfc7231

Software Architecture

1. **The Twelve-Factor App**
   - Site: https://12factor.net/
   - Descrição: Metodologia para construir aplicações SaaS modernas
2. **Clean Architecture (Robert C. Martin)**
   - Livro: “Clean Architecture: A Craftsman’s Guide to Software Structure and Design”
3. **Domain-Driven Design (Eric Evans)**
   - Livro: “Domain-Driven Design: Tackling Complexity in the Heart of Software”

Testing

1. **Test Pyramid (Mike Cohn)**
   - Article: https://martinfowler.com/articles/practical-test-pyramid.html
   - Descrição: Estratégia de testes com unit, integration, e E2E tests
2. **Given-When-Then (Gherkin)**
   - Docs: https://cucumber.io/docs/gherkin/reference/
   - Descrição: Padrão para escrever testes legíveis

DevOps e Deploy

1. **Docker**
   - Site: https://www.docker.com/
   - Docs: https://docs.docker.com/
   - Best Practices: https://docs.docker.com/develop/dev-best-practices/
2. **Docker Compose**
   - Docs: https://docs.docker.com/compose/
3. **GitHub Actions**
   - Site: https://github.com/features/actions
   - Docs: https://docs.github.com/en/actions
4. **Render**
   - Site: https://render.com/
   - Docs: https://render.com/docs

Code Quality

1. **Ruff**
   - Site: https://github.com/astral-sh/ruff
   - Descrição: Linter Python extremamente rápido
2. **Black**
   - Site: https://black.readthedocs.io/
   - Descrição: Code formatter Python
3. **Bandit**
   - Site: https://bandit.readthedocs.io/
   - Descrição: Security linter para Python

Artigos e Blogs Relevantes

1. **Real Python**
   - Site: https://realpython.com/
   - Tutoriais: Flask, PostgreSQL, Testing, JWT, etc.
2. **Full Stack Python**
   - Site: https://www.fullstackpython.com/
   - Guias: Flask, Deployment, Security, Testing
3. **Martin Fowler’s Blog**
   - Site: https://martinfowler.com/
   - Tópicos: Architecture, Testing, Refactoring

4. **Auth0 Blog**
    - Site: https://auth0.com/blog/
    - Tópicos: JWT, OAuth, Authentication best practices

Ferramentas Utilizadas

1. **Visual Studio Code**
    - Site: https://code.visualstudio.com/
    - Extensions: Python, Docker, GitLens
2. **Postman**
    - Site: https://www.postman.com/
    - Uso: Testing de API REST
3. **DBeaver**
    - Site: https://dbeaver.io/
    - Uso: Cliente de base de dados universal
4. **Git**
    - Site: https://git-scm.com/
    - Docs: https://git-scm.com/doc

Versioning

1. **Semantic Versioning (SemVer)**
    - Site: https://semver.org/
    - Formato: MAJOR.MINOR.PATCH (ex: 1.2.3)
2. **Keep a Changelog**
    - Site: https://keepachangelog.com/
    - Descrição: Como manter um CHANGELOG.md

Learning Resources

1. **Flask Mega-Tutorial (Miguel Grinberg)**
    - URL: https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world
    - Descrição: Tutorial completo de Flask
2. **Test-Driven Development with Python (Harry Percival)**
    - Livro: “Test-Driven Development with Python”
    - Site: https://www.obeythetestinggoat.com/

Communities

1. **Stack Overflow**
    - Site: https://stackoverflow.com/

    - Tags: python, flask, postgresql, jwt

2. **Reddit**
    - r/Python: https://www.reddit.com/r/Python/
    - r/flask: https://www.reddit.com/r/flask/
    - r/webdev: https://www.reddit.com/r/webdev/

3. **Dev.to**
    - Site: https://dev.to/
    - Tags: #python #flask #api

Standards e RFCs

1. **HTTP/1.1 (RFC 7230-7235)**
    - URL: https://datatracker.ietf.org/doc/html/rfc7230

2. **JSON (RFC 8259)**
    - URL: https://datatracker.ietf.org/doc/html/rfc8259

3. **URI (RFC 3986)**
    - URL: https://datatracker.ietf.org/doc/html/rfc3986

Nota Final:

Este relatório técnico documenta o estado atual do Issue Tracker API em 20 de Fevereiro de 2026. O projeto continua em desenvolvimento ativo, e melhorias listadas no capítulo “Limitações e Melhorias Futuras” representam o roadmap planeado.

Para mais informações, consultar:
- **Repositório GitHub**: https://github.com/your-username/issue-tracker-api
- **Documentação API**: /docs/api_examples.md
- **Arquitetura**: /docs/architecture.md

Contacto:
- Email: your.email@example.com
- LinkedIn: https://linkedin.com/in/your-profile
- Portfolio: https://your-portfolio.com

Agradecimentos:

Este projeto foi desenvolvido como parte do portfólio técnico para demonstrar competências de Backend Engineering. Agradecimentos à comunidade open-source e aos recursos educativos listados nas referências.