Documentação Técnica - Sistema de Gerenciamento para Empresa de Instalação de Tile

Data: 06/06/2025
Versão: 1.0

Sumário

1. Visão Geral da Arquitetura
2. Tecnologias Utilizadas
3. Estrutura do Projeto
4. Backend
5. Frontend
6. Banco de Dados
7. API
8. Autenticação e Segurança
9. Implantação
10. Manutenção
11. Backup e Recuperação
12. Escalabilidade

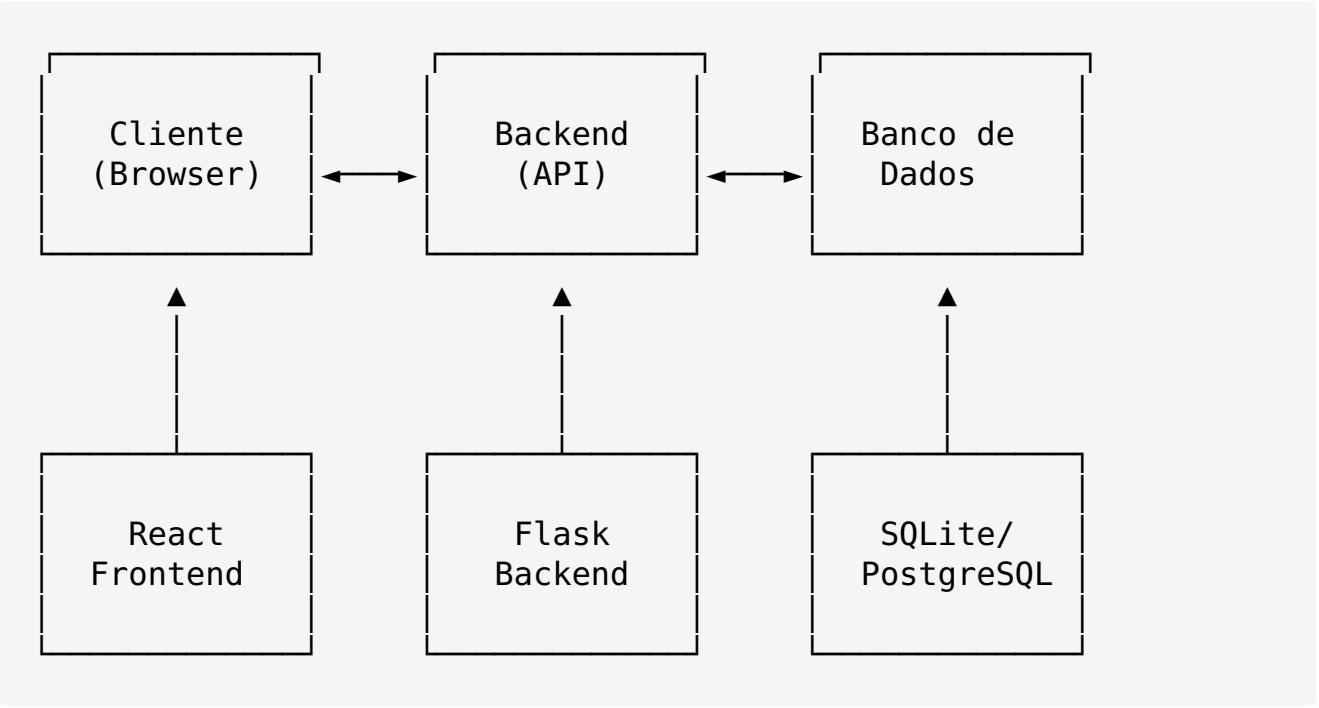
1. Visão Geral da Arquitetura

O Sistema de Gerenciamento para Empresa de Instalação de Tile foi desenvolvido utilizando uma arquitetura de três camadas:

1. **Frontend**: Interface de usuário desenvolvida em React
2. **Backend**: API RESTful desenvolvida em Python com Flask
3. **Banco de Dados**: SQLite para desenvolvimento e PostgreSQL para produção

A arquitetura foi projetada para ser modular, escalável e de fácil manutenção, permitindo que novos recursos sejam adicionados com o mínimo de impacto no sistema existente.

1.1 Diagrama de Arquitetura



2. Tecnologias Utilizadas

2.1 Frontend

- **Framework**: React 19.1.0
- **Gerenciador de Pacotes**: pnpm 10.4.1
- **Biblioteca de UI**: Tailwind CSS
- **Roteamento**: React Router 6
- **Gerenciamento de Estado**: React Context API
- **Requisições HTTP**: Axios
- **Validação de Formulários**: React Hook Form
- **Componentes de UI**: Shadcn UI

2.2 Backend

- **Linguagem**: Python 3.11
- **Framework**: Flask 2.3.3
- **ORM**: SQLAlchemy 2.0.0
- **Autenticação**: Flask-JWT-Extended 4.5.2
- **Validação**: Marshmallow 3.19.0
- **Migrações de Banco de Dados**: Flask-Migrate 4.0.4
- **CORS**: Flask-CORS 4.0.0

2.3 Banco de Dados

- **Desenvolvimento**: SQLite 3
- **Produção**: PostgreSQL 14

2.4 Ferramentas de Desenvolvimento

- **Controle de Versão**: Git
- **Ambiente Virtual Python**: venv
- **Servidor de Desenvolvimento Frontend**: Vite
- **Testes**: Pytest (backend), Puppeteer (frontend)
- **Linting**: ESLint (frontend), Flake8 (backend)
- **Formatação de Código**: Prettier (frontend), Black (backend)

3. Estrutura do Projeto

A estrutura do projeto é organizada da seguinte forma:

```
tile-system/
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── config.py
│   │   ├── models/
│   │   ├── resources/
│   │   ├── schemas/
│   │   └── utils/
│   ├── migrations/
│   ├── tests/
│   ├── venv/
│   ├── .env
│   ├── .flaskenv
│   ├── requirements.txt
│   └── run.py
├── frontend/
│   ├── public/
│   ├── src/
│   │   ├── components/
│   │   ├── services/
│   │   ├── pages/
│   │   ├── hooks/
│   │   ├── utils/
│   │   ├── App.jsx
│   │   └── main.jsx
│   ├── tests/
│   ├── .env
│   ├── package.json
```

```
│   └── vite.config.js
├── docs/
│   ├── manual_usuario.md
│   ├── documentacao_tecnica.md
│   └── guia_manutencao.md
└── deploy.sh
```

4. Backend

4.1 Estrutura do Backend

O backend é organizado seguindo o padrão de arquitetura MVC (Model-View-Controller), com algumas adaptações para APIs RESTful:

- **Models**: Definição das entidades e relacionamentos do banco de dados
- **Resources**: Endpoints da API e lógica de negócios
- **Schemas**: Validação e serialização de dados
- **Utils**: Funções utilitárias e helpers

4.2 Inicialização da Aplicação

O arquivo `run.py` é o ponto de entrada da aplicação. Ele importa a aplicação Flask do módulo `app` e a executa:

```python
from app import create_app

app = create_app()

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

A função `create_app()` está definida em `app/__init__.py` e é responsável por configurar a aplicação Flask, registrar as extensões e blueprints:

```python
def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # Inicializa as extensões
    db.init_app(app)
    migrate.init_app(app, db)
    jwt.init_app(app)
    CORS(app)
```

```
    # Registra os blueprints
    from .resources.auth import auth_bp
    app.register_blueprint(auth_bp, url_prefix='/api/auth')

    from .resources.client import client_bp
    app.register_blueprint(client_bp, url_prefix='/api/clients')

    from .resources.project import project_bp
    app.register_blueprint(project_bp, url_prefix='/api/
projects')

    # Outros blueprints...

    return app
```

4.3 Modelos de Dados

Os modelos de dados são definidos usando SQLAlchemy ORM. Exemplo do modelo de Cliente:

```
class Client(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(100), nullable=True)
    phone = db.Column(db.String(20), nullable=False)
    address = db.Column(db.String(200), nullable=True)
    city = db.Column(db.String(100), nullable=True)
    state = db.Column(db.String(2), nullable=True)
    zip_code = db.Column(db.String(10), nullable=True)
    referral_source = db.Column(db.String(100), nullable=True)
    notes = db.Column(db.Text, nullable=True)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow,
onupdate=datetime.utcnow)

    # Relacionamentos
    projects = db.relationship('Project', backref='client',
lazy=True)
```

4.4 Recursos da API

Os recursos da API são implementados usando blueprints do Flask. Exemplo do recurso de Cliente:

```
client_bp = Blueprint('client', __name__)

@client_bp.route('', methods=['GET'])
```

```
@jwt_required()
def get_clients():
    page = request.args.get('page', 1, type=int)
    per_page = request.args.get('per_page', 10, type=int)

    clients = Client.query.paginate(page=page,
per_page=per_page, error_out=False)

    return {
        'clients': client_schema.dump(clients.items, many=True),
        'count': clients.total,
        'pages': clients.pages,
        'current_page': clients.page
    }

@client_bp.route('/<int:id>', methods=['GET'])
@jwt_required()
def get_client(id):
    client = Client.query.get_or_404(id)
    return {'client': client_schema.dump(client)}

@client_bp.route('', methods=['POST'])
@jwt_required()
def create_client():
    try:
        data = request.get_json()
        errors = client_schema.validate(data)
        if errors:
            return {'message': 'Erro de validação', 'errors':
errors}, 400

        client = Client(**data)
        db.session.add(client)
        db.session.commit()

        return {'message': 'Cliente criado com sucesso',
'client': client_schema.dump(client)}, 201
    except Exception as e:
        db.session.rollback()
        return {'message': str(e)}, 500
```

4.5 Schemas

Os schemas são definidos usando Marshmallow e são responsáveis pela validação e serialização dos dados:

```
class ClientSchema(Schema):
    id = fields.Integer(dump_only=True)
    name = fields.String(required=True,
```

```
validate=validate.Length(min=3, max=100))
    email = fields.Email(allow_none=True)
    phone = fields.String(required=True,
validate=validate.Length(min=8, max=20))
    address = fields.String(allow_none=True)
    city = fields.String(allow_none=True)
    state = fields.String(allow_none=True,
validate=validate.Length(max=2))
    zip_code = fields.String(allow_none=True)
    referral_source = fields.String(allow_none=True)
    notes = fields.String(allow_none=True)
    created_at = fields.DateTime(dump_only=True)
    updated_at = fields.DateTime(dump_only=True)
```

5. Frontend

5.1 Estrutura do Frontend

O frontend é organizado seguindo as melhores práticas do React:

- **components**: Componentes reutilizáveis
- **services**: Serviços para comunicação com a API
- **pages**: Páginas da aplicação
- **hooks**: Hooks personalizados
- **utils**: Funções utilitárias

5.2 Componentes Principais

5.2.1 App.jsx

O componente `App.jsx` é o ponto de entrada da aplicação React e define as rotas da aplicação:

```
function App() {
  return (
    <Router>
      <Routes>
        {/* Rota pública */}
        <Route path="/login" element={<Login />} />

        {/* Rotas protegidas */}
        <Route path="/" element={
          <PrivateRoute>
            <Layout />
          </PrivateRoute>
        }>
```

```
            <Route index element={<Dashboard />} />

            {/* Rotas de clientes */}
            <Route path="clients" element={<ClientList />} />
            <Route path="clients/new" element={<ClientForm />} />
            <Route path="clients/:id" element={<ClientForm />} />

            {/* Outras rotas... */}

            <Route path="*" element={<Navigate to="/" />} />
          </Route>
        </Routes>
      </Router>
    );
}
```

5.2.2 Layout.jsx

O componente `Layout.jsx` define a estrutura básica da aplicação, incluindo o menu lateral e a barra superior:

```
function Layout() {
  return (
    <div className="flex h-screen">
      <Sidebar />
      <div className="flex flex-col flex-1 overflow-hidden">
        <Navbar />
        <main className="flex-1 overflow-y-auto p-6 bg-gray-50">
          <Outlet />
        </main>
      </div>
    </div>
  );
}
```

5.3 Serviços

Os serviços são responsáveis pela comunicação com a API:

```
// services/api.js
import axios from 'axios';

const api = axios.create({
  baseURL: 'http://localhost:5000/api',
});

api.interceptors.request.use(
  (config) => {
```

```
    const token = localStorage.getItem('token');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => Promise.reject(error)
);

api.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response && error.response.status === 401) {
      localStorage.removeItem('token');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);

export default api;
```

```
// services/clients.js
import api from './api';

export const getClients = async (page = 1, perPage = 10) => {
  const response = await api.get(`/clients?page=${page}
&per_page=${perPage}`);
  return response.data;
};

export const getClient = async (id) => {
  const response = await api.get(`/clients/${id}`);
  return response.data;
};

export const createClient = async (data) => {
  const response = await api.post('/clients', data);
  return response.data;
};

export const updateClient = async (id, data) => {
  const response = await api.put(`/clients/${id}`, data);
  return response.data;
};

export const deleteClient = async (id) => {
  const response = await api.delete(`/clients/${id}`);
  return response.data;
};
```

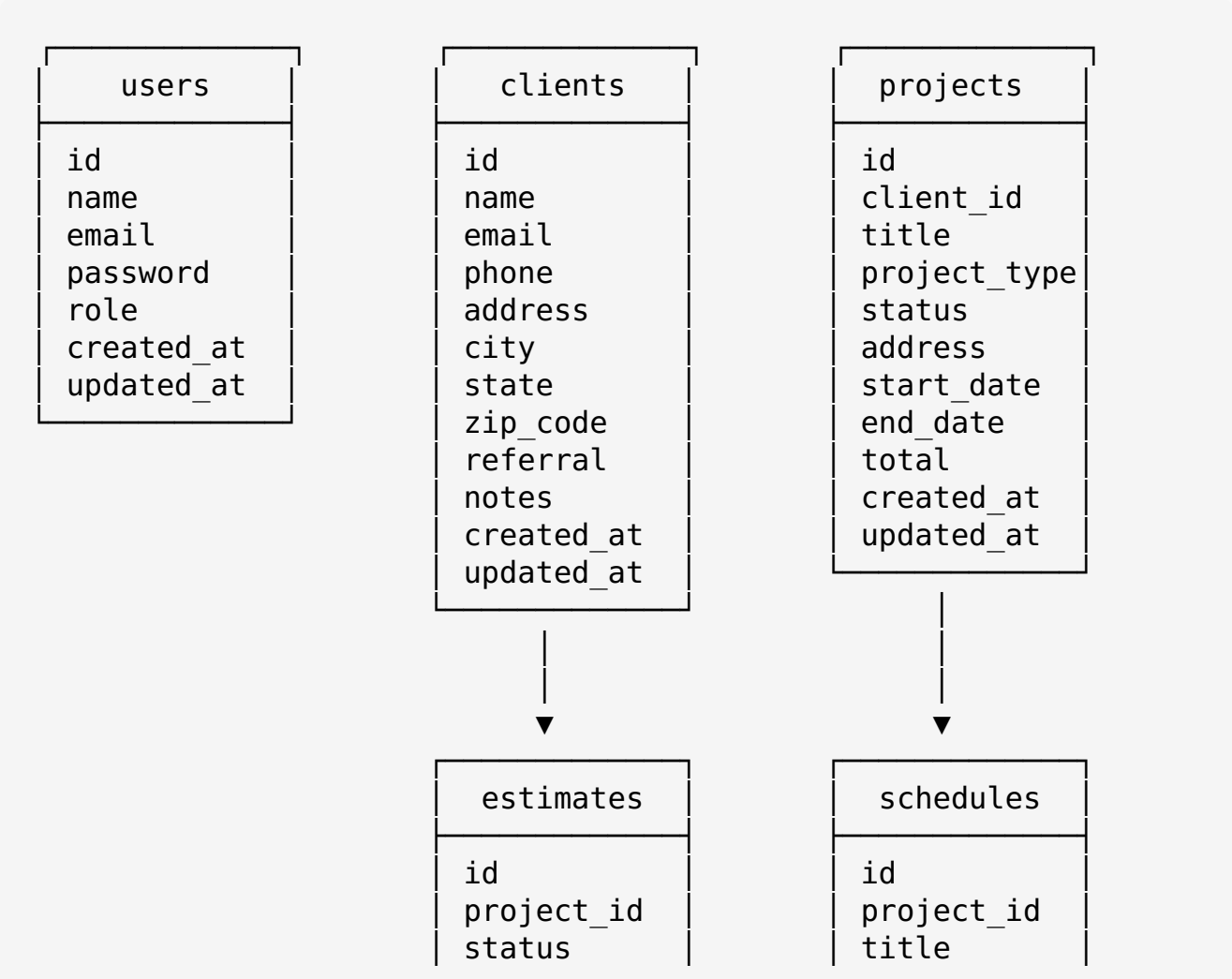
6. Banco de Dados

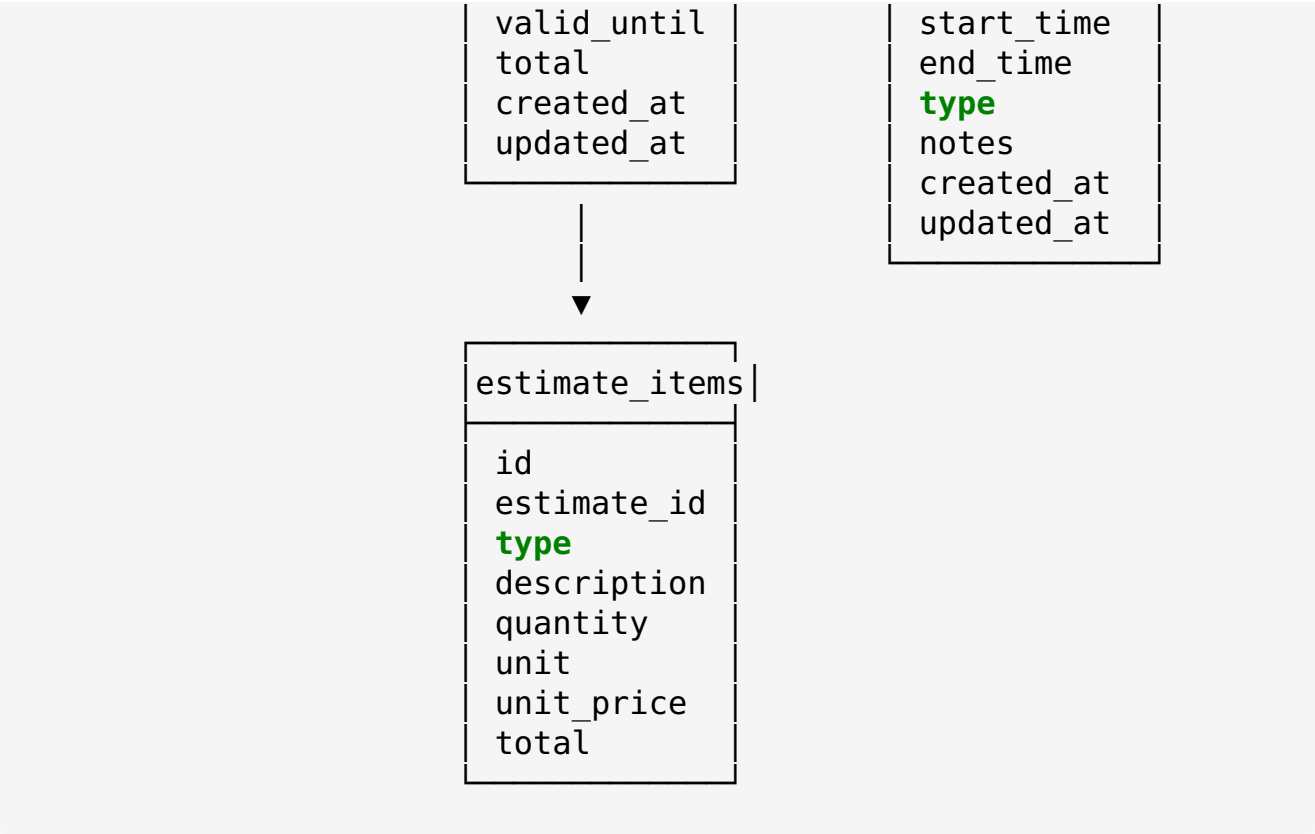
6.1 Modelo de Dados

O sistema utiliza um modelo de dados relacional com as seguintes tabelas principais:

- **users**: Usuários do sistema
- **clients**: Clientes da empresa
- **projects**: Projetos de instalação
- **estimates**: Orçamentos
- **estimate_items**: Itens de orçamento
- **schedules**: Agendamentos
- **materials**: Materiais utilizados
- **suppliers**: Fornecedores
- **payments**: Pagamentos
- **expenses**: Despesas
- **progress**: Registros de progresso dos projetos
- **files**: Arquivos anexados aos projetos

6.2 Diagrama ER





6.3 Migrações

O sistema utiliza Flask-Migrate para gerenciar as migrações do banco de dados. As migrações são armazenadas no diretório `migrations/` e podem ser executadas com os seguintes comandos:

```
# Inicializar as migrações
flask db init

# Criar uma nova migração
flask db migrate -m "Descrição da migração"

# Aplicar as migrações
flask db upgrade

# Reverter a última migração
flask db downgrade
```

7. API

7.1 Endpoints da API

A API do sistema expõe os seguintes endpoints:

7.1.1 Autenticação

- `POST /api/auth/login`: Autenticar usuário
- `POST /api/auth/register`: Registrar novo usuário (apenas para administradores)
- `POST /api/auth/refresh`: Renovar token de acesso
- `POST /api/auth/logout`: Encerrar sessão

7.1.2 Clientes

- `GET /api/clients`: Listar clientes
- `GET /api/clients/<id>`: Obter cliente específico
- `POST /api/clients`: Criar novo cliente
- `PUT /api/clients/<id>`: Atualizar cliente
- `DELETE /api/clients/<id>`: Excluir cliente

7.1.3 Projetos

- `GET /api/projects`: Listar projetos
- `GET /api/projects/<id>`: Obter projeto específico
- `POST /api/projects`: Criar novo projeto
- `PUT /api/projects/<id>`: Atualizar projeto
- `DELETE /api/projects/<id>`: Excluir projeto
- `GET /api/projects/<id>/progress`: Listar registros de progresso
- `POST /api/projects/<id>/progress`: Adicionar registro de progresso

7.1.4 Orçamentos

- `GET /api/estimates`: Listar orçamentos
- `GET /api/estimates/<id>`: Obter orçamento específico
- `POST /api/estimates`: Criar novo orçamento
- `PUT /api/estimates/<id>`: Atualizar orçamento
- `DELETE /api/estimates/<id>`: Excluir orçamento
- `POST /api/estimates/<id>/approve`: Aprovar orçamento
- `POST /api/estimates/<id>/reject`: Rejeitar orçamento

7.1.5 Agendamentos

- `GET /api/schedules`: Listar agendamentos
- `GET /api/schedules/<id>`: Obter agendamento específico
- `POST /api/schedules`: Criar novo agendamento
- `PUT /api/schedules/<id>`: Atualizar agendamento

- `DELETE /api/schedules/<id>`: Excluir agendamento

7.1.6 Materiais

- `GET /api/materials`: Listar materiais
- `GET /api/materials/<id>`: Obter material específico
- `POST /api/materials`: Criar novo material
- `PUT /api/materials/<id>`: Atualizar material
- `DELETE /api/materials/<id>`: Excluir material
- `POST /api/materials/<id>/stock/in`: Registrar entrada de estoque
- `POST /api/materials/<id>/stock/out`: Registrar saída de estoque

7.2 Formato de Resposta

Todas as respostas da API seguem um formato padrão:

```
{
  "message": "Mensagem descritiva",
  "data": { ... },
  "errors": { ... }
}
```

- `message`: Uma mensagem descritiva sobre o resultado da operação
- `data`: Os dados retornados pela operação (opcional)
- `errors`: Erros de validação ou outros erros (opcional)

7.3 Códigos de Status HTTP

A API utiliza os seguintes códigos de status HTTP:

- `200 OK`: Requisição bem-sucedida
- `201 Created`: Recurso criado com sucesso
- `400 Bad Request`: Erro de validação ou requisição inválida
- `401 Unauthorized`: Autenticação necessária ou falha na autenticação
- `403 Forbidden`: Acesso negado
- `404 Not Found`: Recurso não encontrado
- `500 Internal Server Error`: Erro interno do servidor

8. Autenticação e Segurança

8.1 Autenticação JWT

O sistema utiliza JSON Web Tokens (JWT) para autenticação. O fluxo de autenticação é o seguinte:

1. O usuário envia suas credenciais (email e senha) para o endpoint `/api/auth/login`
2. O servidor valida as credenciais e, se válidas, gera um token JWT
3. O token JWT é retornado ao cliente
4. O cliente armazena o token JWT (geralmente no localStorage)
5. O cliente inclui o token JWT no cabeçalho `Authorization` de todas as requisições subsequentes
6. O servidor valida o token JWT e, se válido, processa a requisição

8.2 Proteção de Rotas

No frontend, as rotas protegidas são envolvidas pelo componente `PrivateRoute`, que verifica se o usuário está autenticado:

```
const PrivateRoute = ({ children }) => {
  const isAuthenticated = () => {
    const token = localStorage.getItem('token');
    if (!token) return false;

    try {
      const payload = JSON.parse(atob(token.split('.')[1]));
      return payload.exp > Date.now() / 1000;
    } catch (e) {
      return false;
    }
  };

  return isAuthenticated() ? children : <Navigate to="/login" /
>;
};
```

No backend, as rotas protegidas são decoradas com o decorador `@jwt_required()`:

```
@client_bp.route('', methods=['GET'])
@jwt_required()
def get_clients():
    # ...
```

8.3 Controle de Acesso Baseado em Funções (RBAC)

O sistema implementa controle de acesso baseado em funções (RBAC) para restringir o acesso a determinadas funcionalidades com base na função do usuário:

```
@user_bp.route('', methods=['GET'])
@jwt_required()
def get_users():
    current_user = get_jwt_identity()
    user = User.query.filter_by(email=current_user).first()

    if user.role != 'admin':
        return {'message': 'Acesso negado'}, 403

    # ...
```

9. Implantação

9.1 Requisitos de Sistema

- Python 3.11 ou superior
- Node.js 20.18.0 ou superior
- PostgreSQL 14 ou superior (para produção)
- Nginx (para produção)

9.2 Implantação do Backend

Para implantar o backend em produção:

1. Clone o repositório
2. Crie um ambiente virtual Python
3. Instale as dependências
4. Configure as variáveis de ambiente
5. Inicialize o banco de dados
6. Configure o servidor WSGI (Gunicorn)
7. Configure o servidor web (Nginx)

```
# Clonar o repositório
git clone https://github.com/seu-usuario/tile-system.git
cd tile-system

# Criar ambiente virtual
python3 -m venv venv
source venv/bin/activate
```

```
# Instalar dependências
cd backend
pip install -r requirements.txt

# Configurar variáveis de ambiente
cp .env.example .env
# Edite o arquivo .env com as configurações de produção

# Inicializar banco de dados
flask db upgrade
python init_db.py

# Configurar Gunicorn
pip install gunicorn
gunicorn -b 0.0.0.0:5000 run:app
```

9.3 Implantação do Frontend

Para implantar o frontend em produção:

1. Clone o repositório
2. Instale as dependências
3. Configure as variáveis de ambiente
4. Construa a aplicação
5. Configure o servidor web (Nginx)

```
 # Clonar o repositório
git clone https://github.com/seu-usuario/tile-system.git
cd tile-system

# Instalar dependências
cd frontend
pnpm install

# Configurar variáveis de ambiente
cp .env.example .env
# Edite o arquivo .env com as configurações de produção

# Construir a aplicação
pnpm build

# O resultado estará no diretório dist/
```

9.4 Script de Implantação

O sistema inclui um script de implantação (`deploy.sh`) que automatiza o processo de implantação:

```
# Executar o script de implantação
cd tile-system
./deploy.sh
```

10. Manutenção

10.1 Logs

O sistema gera logs para facilitar a manutenção e a depuração de problemas:

- Logs do backend: `/var/log/tile-system/backend.log`
- Logs do frontend: `/var/log/tile-system/frontend.log`
- Logs do banco de dados: `/var/log/postgresql/postgresql.log`

10.2 Monitoramento

Recomenda-se o uso de ferramentas de monitoramento para acompanhar o desempenho e a disponibilidade do sistema:

- Prometheus: Coleta de métricas
- Grafana: Visualização de métricas
- Sentry: Monitoramento de erros

10.3 Atualizações

Para atualizar o sistema:

1. Faça backup do banco de dados
2. Clone a versão mais recente do repositório
3. Instale as dependências atualizadas
4. Execute as migrações do banco de dados
5. Construa a aplicação frontend
6. Reinicie os serviços

```
# Backup do banco de dados
pg_dump -U postgres tile_system > backup.sql
```

```
# Atualizar o código
cd tile-system
git pull

# Atualizar o backend
cd backend
source venv/bin/activate
pip install -r requirements.txt
flask db upgrade

# Atualizar o frontend
cd ../frontend
pnpm install
pnpm build

# Reiniciar os serviços
sudo systemctl restart tile-system-backend
sudo systemctl restart nginx
```

11. Backup e Recuperação

11.1 Backup do Banco de Dados

Recomenda-se realizar backups diários do banco de dados:

```
# Backup do banco de dados
pg_dump -U postgres tile_system > backup_$(date +%Y%m%d).sql

# Compactar o backup
gzip backup_$(date +%Y%m%d).sql
```

11.2 Recuperação do Banco de Dados

Para restaurar o banco de dados a partir de um backup:

```
# Criar o banco de dados (se não existir)
createdb -U postgres tile_system

# Restaurar o backup
gunzip -c backup_20250606.sql.gz | psql -U postgres tile_system
```

11.3 Backup de Arquivos

Recomenda-se realizar backups regulares dos arquivos do sistema:

```
# Backup dos arquivos
tar -czf tile-system-files_$(date +%Y%m%d).tar.gz /path/to/tile-system/uploads
```

12. Escalabilidade

12.1 Escalabilidade Vertical

O sistema pode ser escalado verticalmente aumentando os recursos do servidor:

- CPU: Adicionar mais núcleos
- Memória: Aumentar a quantidade de RAM
- Disco: Aumentar o espaço em disco ou usar discos mais rápidos

12.2 Escalabilidade Horizontal

O sistema pode ser escalado horizontalmente distribuindo a carga entre vários servidores:

- Backend: Usar um balanceador de carga para distribuir as requisições entre vários servidores de backend
- Frontend: Usar uma CDN para distribuir os arquivos estáticos
- Banco de Dados: Usar replicação para distribuir a carga de leitura

12.3 Otimização de Desempenho

Para otimizar o desempenho do sistema:

- Implementar cache no backend (Redis)
- Otimizar consultas ao banco de dados
- Minimizar e compactar os arquivos do frontend
- Usar lazy loading para carregar componentes sob demanda
- Implementar paginação para grandes conjuntos de dados

© 2025 Tile System. Todos os direitos reservados.