Documentação Completa: Sistema de Gerenciamento de Oficina Mecânica

Sumário

---

1. Introdução

O **Sistema de Gerenciamento de Oficina Mecânica** é uma plataforma web projetada para otimizar o gerenciamento de oficinas, fornecendo uma interface eficiente para controlar agendamentos, serviços, estoque de peças, clientes e veículos, além de gerar relatórios financeiros e operacionais.

Este sistema será desenvolvido utilizando o **Django** (com Django REST Framework) para o back-end, garantindo um ambiente robusto e escalável, e tecnologias modernas no front-end, como **React.js** e **Tailwind CSS**, para uma experiência de usuário fluida e eficiente.

---

2. Requisitos

2.1 Requisitos Funcionais

1. **Cadastro de Clientes**
    - Registro de clientes com informações como nome, telefone, e-mail e endereço.
    - Listagem e visualização detalhada dos dados do cliente.
2. **Cadastro de Veículos**
    - Associações de veículos a clientes com detalhes como modelo, ano, placa e tipo (carro, moto, caminhão, etc.).
3. **Agendamento de Serviços**
    - Agendamento de manutenção e serviços, com possibilidade de adicionar peças e gerar estimativas de custos.
    - Exibição de calendários para visualização de disponibilidade e agendamentos realizados.
4. **Controle de Serviços**
    - Registro de serviços realizados, incluindo descrição, status (pendente, em andamento, concluído) e valor total.
    - Acompanhamento da execução dos serviços.

5. **Controle de Estoque de Peças**
    - Gerenciamento de peças disponíveis, incluindo controle de quantidade, preço e fornecedor.
    - Atualização automática do estoque após a execução de serviços.
6. **Emissão de Orçamentos e Faturas**
    - Geração de orçamentos baseados nos serviços agendados e peças utilizadas.
    - Emissão de faturas detalhadas após a finalização dos serviços.
7. **Relatórios**
    - Geração de relatórios financeiros e operacionais, como número de serviços, faturamento, peças vendidas, etc.

2.2 Requisitos Não Funcionais

- **Desempenho**: O sistema deve ser capaz de suportar grandes volumes de dados, como clientes, veículos e histórico de serviços.
- **Segurança**: A plataforma deve garantir a segurança dos dados sensíveis (informações pessoais, detalhes de pagamento).
- **Escalabilidade**: O sistema deve permitir fácil expansão, permitindo o suporte a múltiplas filiais de oficinas.
- **Usabilidade**: A interface deve ser intuitiva, com feedback visual e fluxos simplificados.

---

3. Tecnologias Utilizadas

Back-End

- **Django**: Framework Python robusto para construção de sistemas web rápidos e seguros.
- **Django REST Framework**: Para a criação de APIs RESTful para comunicação com o front-end.

- **PostgreSQL**: Banco de dados relacional para armazenar dados de clientes, veículos, serviços e peças.
- **Celery**: Para processamento assíncrono de tarefas (por exemplo, envio de e-mails ou geração de relatórios).

Front-End

- **React.js**: Biblioteca JavaScript para construir interfaces dinâmicas e interativas.
- **Tailwind CSS**: Framework de CSS utilitário para criar interfaces modernas e responsivas.
- **Axios**: Para fazer requisições HTTP de maneira eficiente para a API.

Segurança

- **JWT (JSON Web Tokens)**: Para autenticação e autorização segura de usuários.
- **HTTPS**: Para garantir a segurança na comunicação entre o cliente e o servidor.

---

4. Arquitetura do Sistema

4.1 Arquitetura de Camadas

A arquitetura do sistema segue o padrão **MVC (Model-View-Controller)**, com separação clara entre as camadas de apresentação, lógica de negócios e persistência de dados.

1. **Camada de Apresentação (Front-End)**
    - **Responsabilidade**: Interação com o usuário.
    - **Tecnologias**: React.js, Tailwind CSS.
2. **Camada de Lógica de Negócio (Back-End)**
    - **Responsabilidade**: Processamento dos dados e fornecimento de APIs.
    - **Tecnologias**: Django + Django REST Framework.
3. **Camada de Persistência (Banco de Dados)**

- **Responsabilidade**: Armazenamento de dados do sistema.
- **Tecnologias**: PostgreSQL.

4.2 Fluxo de Dados

1. **Front-End** envia requisições HTTP para o **Back-End** através da API RESTful.
2. O **Back-End** processa essas requisições, interage com o **Banco de Dados** para salvar ou recuperar dados e retorna uma resposta ao **Front-End**.
3. O **Front-End** atualiza a interface com base nos dados recebidos da API.

---

5. Estrutura de Diretórios

A estrutura de diretórios para o projeto segue uma organização clara:

```
/gerenciamento_oficina
  ├── /config/              # Arquivos de configuração do Django
  ├── /clientes/            # App de gerenciamento de clientes
  ├── /veiculos/            # App de gerenciamento de veículos
  ├── /servicos/            # App de gerenciamento de serviços
  ├── /pecas/               # App de controle de peças e estoque
  ├── /relatorios/          # App para geração de relatórios
  ├── /templates/           # Templates HTML (para views do Django)
  ├── /static/              # Arquivos estáticos (CSS, JS, imagens)
  ├── /media/               # Arquivos de mídia (ex: imagens de clientes ou veículos)
  ├── /migrations/          # Migrações do banco de dados
  ├── /tests/               # Testes automatizados
  └── manage.py             # Script de gerenciamento
```

---

6. Banco de Dados

O banco de dados será implementado em **PostgreSQL** ou **MySQL**, utilizando a modelagem relacional para garantir a consistência e integridade dos dados.

6.1 Estrutura das Tabelas

1. **Cliente**

- `id`: Inteiro (PK)
- `nome`: String
- `telefone`: String
- `email`: String
- `endereco`: Texto

2. **Veículo**
   - `id`: Inteiro (PK)
   - `cliente_id`: Relacionamento com a tabela Cliente (FK)
   - `modelo`: String
   - `placa`: String
   - `ano`: Inteiro
   - `tipo`: String (carro, moto, caminhão)
3. **Serviço**
   - `id`: Inteiro (PK)
   - `veiculo_id`: Relacionamento com a tabela Veículo (FK)
   - `descricao`: Texto
   - `status`: Enum (pendente, em andamento, concluído)
   - `data_agendamento`: Data
   - `valor_total`: Decimal
4. **Peça**
   - `id`: Inteiro (PK)
   - `descricao`: String

- `quantidade_estoque`: Inteiro
- `preco_unitario`: Decimal

5. **Orçamento**
   - `id`: Inteiro (PK)
   - `servico_id`: Relacionamento com a tabela Serviço (FK)
   - `valor_total`: Decimal
   - `data_emissao`: Data

---

7. Funcionalidades do Sistema

7.1 Front-End

- **Interface de Clientes**: Cadastro, visualização e edição de dados dos clientes.
- **Interface de Veículos**: Cadastro e visualização de veículos associados aos clientes.
- **Agenda de Serviços**: Interface gráfica para agendar e visualizar serviços em um calendário interativo.
- **Feedback Visual**: Mensagens claras para sucesso ou erro em ações, como agendamento ou pagamento de fatura.

7.2 Back-End

- **Autenticação e Autorização**: O uso de JWT permite que usuários façam login de forma segura. O sistema irá suportar múltiplos tipos de usuários (administradores, atendentes e mecânicos).
- **API RESTful**: A comunicação entre o front-end e o back-end será feita através de endpoints RESTful para garantir flexibilidade e desacoplamento entre as camadas.

**8. API

- Endpoints**

Exemplo de endpoints principais:

- **POST /clientes/**: Criação de um novo cliente.
- **GET /clientes/{id}/**: Retorna os dados de um cliente específico.
- **POST /servicos/**: Criação de um novo serviço agendado.
- **GET /servicos/{id}/**: Detalhes de um serviço específico.
- **GET /relatorios/**: Geração de relatórios financeiros e operacionais.

---

9. Fluxo de Trabalho

1. O usuário realiza o login com suas credenciais.
2. O administrador cadastra clientes e veículos.
3. O atendente agenda serviços, e o mecânico registra o progresso.
4. A plataforma gera relatórios financeiros e operacionais com base nos dados armazenados.

---

10. Segurança e Autenticação

- **JWT**: Utilizado para autenticação de usuários. O sistema garante que apenas usuários autenticados e autorizados acessem os dados e funcionalidades.
- **Proteção Contra Ataques**: O Django está configurado para proteger contra ataques de **Cross-Site Scripting (XSS)**, **Cross-Site Request Forgery (CSRF)**, e **SQL Injection**.

---

11. Design de Interface (UI) e Experiência do Usuário (UX)

11.1 UI

- **Estilo Moderno e Limpo**: Design com foco na clareza e acessibilidade. Uso de cores neutras e ícones intuitivos.
- **Responsividade**: O layout será adaptável para dispositivos móveis, proporcionando uma experiência ótima em qualquer tela.
- **Feedback Visual**: O sistema exibe mensagens de erro e sucesso de forma clara e objetiva.

11.2 UX

- **Navegação Intuitiva**: Menus bem estruturados, botões visíveis e uma interface fluída.
- **Velocidade de Interação**: O sistema responde rapidamente às interações do usuário, com carregamento assíncrono de dados.

---

12. Implantação

- **Docker**: A aplicação será containerizada com Docker, permitindo fácil implantação em servidores ou na nuvem.
- **CI/CD**: Integração contínua e entrega contínua utilizando ferramentas como **GitLab CI** ou **GitHub Actions**.

---

13. Testes

- **Testes Unitários**: Implementação de testes unitários utilizando o Django Test Framework.
- **Testes de Integração**: Testes para garantir que a comunicação entre front-end e back-end funcione corretamente.
- **Testes de Usabilidade**: Testes para validar a experiência do usuário em diferentes dispositivos.

---

14. Manutenção e Suporte

- **Documentação de Código**: Comentários claros e documentação de API para facilitar a manutenção futura.

- **Suporte ao Usuário**: Canal de atendimento ao cliente disponível através da plataforma.

- **Atualizações Regulares**: Planejamento de atualizações para corrigir bugs e adicionar novas funcionalidades conforme necessário.
