PROGRAMAÇÃO II

PROJETO INTEGRADOR

João Luis Almeida Santos - Matrícula: 20240002508

Sumário

1 Introdução

1.1 Comanda do trabalho

Este Trabalho Integrador tem como objetivo a prática da integração entre frontend e backend, unindo conceitos de Programação II, Engenharia de Software e Banco de Dados. A proposta consiste no desenvolvimento de uma aplicação web completa utilizando HTML5, CSS3, JavaScript e um banco de dados relacional.

Para atender aos requisitos obrigatórios, a solução deve implementar um sistema de Autenticação e Autorização com controle de sessão e diferenciação de perfis, além de operações completas de CRUD (Criação, Leitura, Atualização e Exclusão) para, no mínimo, três entidades diferentes. O resto dos requisitos foi especificado no documento de especificação do trabalho.

1.2 Desafios Enfrentados

O desenvolvimento do projeto apresentou diversos desafios técnicos e de design que exigiram atenção cuidadosa e soluções criativas. Um dos principais desafios foi manter a modularização adequada do código, garantindo que componentes fossem verdadeiramente reutilizáveis sem acoplamento excessivo. A criação de uma arquitetura onde componentes como formulários, botões e grades de dados pudessem ser utilizados em diferentes contextos, recebendo configurações via propriedades, demandou planejamento cuidadoso e múltiplas iterações de refatoração.

A definição de um esquema de cores coerente e interessante para a aplicação representou outro desafio significativo. Equilibrar estética, usabilidade e acessibilidade exigiu experimentação com diferentes paletas de cores e ajustes iterativos até alcançar um resultado visualmente agradável que também mantivesse boa legibilidade e contraste adequado para diferentes elementos da interface.

A decisão de combinar DaisyUI e Material-UI (MUI), embora tenha propor-

cionado flexibilidade na construção da interface, introduziu complexidades não antecipadas. As duas bibliotecas possuem filosofias de design diferentes e, ocasionalmente, estilos conflitantes que causaram bugs visuais e comportamentais durante o desenvolvimento. Problemas com sobreposição de estilos CSS, inconsistências na aparência de componentes e conflitos entre sistemas de temas das bibliotecas exigiram intervenções manuais e soluções de contorno. Em retrospecto, a utilização de Shadcn/ui teria sido mais apropriada, por ter sido especificamente projetado para integração nativa com Tailwind CSS, evitando esses conflitos.

O processo de implementação e refinamento da interface consumiu vários dias de trabalho. desenvolvimento.

Vale destacar que algumas implementações mais complexas foram adaptadas de outros projetos pessoais em desenvolvimento. Funcionalidades como modais arrastáveis com o mouse, que permitem ao usuário reposicionar janelas de diálogo na tela, foram implementações previamente desenvolvidas de outros projetos pessoais (https://github.com/lyszt/iris) que se adequavam bem ao contexto deste projeto. Esses dois projetos foram feitos ao mesmo tempo, então existem muitas similaridades entre eles.

Esses desafios, embora tenham aumentado a complexidade e o tempo de desenvolvimento, contribuíram significativamente para o aprendizado e resultaram em um produto final mais robusto e profissional.

2 Metodologia

2.1 Stack utilizada

Para o desenvolvimento da aplicação, foram selecionados: Python com o framework Django no backend e SQLite como banco de dados. No frontend, utilizou-se a biblioteca React em conjunto com DaisyUI e MUI para a construção da interface.

A escolha do Django deve-se à facilidade e a praticidade do uso da ferra-

menta e à proficiência prévia na linguagem Python. O SQLite foi adotado pela sua praticidade e configuração simplificada, adequando-se ao contexto do projeto, visto que não estou cursando a disciplina de banco de dados. SQLite foi a escolha balanceada entre usar arrays/JSON e criar um banco no PostgreSQL.

A combinação de DaisyUI e MUI no frontend foi motivada pela preferência pelas ferramentas e pela flexibilidade que o uso conjunto dessas bibliotecas de componentes proporciona na estilização. Foi, além disso, usado Tailwind em conjunto. Ao fim do projeto, percebi que o Shadcdn poderia ter substituído os dois por ter sido criado para ser integrado com o Tailwind.

2.2 Estrutura de arquivos

O projeto frontend foi organizado seguindo princípios de modularização e separação de responsabilidades, adotando uma arquitetura baseada em componentes reutilizáveis e páginas isoladas. A estrutura de diretórios divide-se em três categorias principais: components/, pages/ e utils/.

O diretório components/ concentra componentes reutilizáveis que podem ser utilizados em diferentes contextos da aplicação, como botões, formulários, modais e grades de dados. Essa separação permite que elementos da interface sejam compartilhados entre diferentes páginas sem duplicação de código, facilitando a manutenção e garantindo consistência visual. Por exemplo, o componente DeleteButton pode ser utilizado tanto na página de leads quanto na de clientes, mantendo o mesmo comportamento e aparência.

Já o diretório pages/ organiza componentes que representam páginas completas ou seções específicas da aplicação. Cada página possui seu próprio subdiretório, contendo toda a lógica, componentes específicos e recursos necessários para aquela funcionalidade. Essa abordagem encapsula a complexidade de cada módulo, tornando mais fácil localizar e modificar funcionalidades específicas sem afetar outras partes do sistema.

O diretório utils/ centraliza funções utilitárias e helpers que fornecem funcionalidades compartilhadas, como gerenciamento de autenticação, comunicação com a API e formatação de dados. Essa centralização evita a repetição de lógica em múltiplos arquivos e facilita alterações em comportamentos que afetam toda a aplicação.

Um padrão fundamental adotado na estrutura é o uso de arquivos index.jsx em cada diretório. Essa convenção traz dois benefícios significativos. Primeiro, simplifica os caminhos de importação: ao invés de escrever

```
import Dashboard from './pages/dashboard/Dashboard.jsx'
```

pode-se usar apenas

```
import Dashboard from './pages/dashboard'
```

pois o JavaScript automaticamente busca o arquivo index.jsx quando um diretório é importado. Segundo, facilita a modularização ao permitir que cada diretório funcione como um módulo autocontido, onde o index.jsx atua como ponto de entrada e pode gerenciar exportações de múltiplos arquivos internos.

Essa organização promove escalabilidade, pois novos componentes ou páginas podem ser adicionados seguindo o mesmo padrão estabelecido. Além disso, melhora a navegabilidade do código, permitindo que desenvolvedores localizem rapidamente arquivos relacionados a funcionalidades específicas. A separação clara entre componentes reutilizáveis, páginas específicas e utilitários compartilhados reduz o acoplamento entre diferentes partes do sistema e facilita testes unitários, já que cada módulo possui responsabilidades bem definidas.

2.3 Especificações e Instruções de Instalação

O projeto foi desenvolvido como um sistema de gerenciamento de leads e clientes com dashboard integrado, utilizando uma arquitetura separada entre frontend e backend. Para garantir a execução adequada da aplicação, é necessário atender aos seguintes pré-requisitos: Python 3.13.9 para o backend Django e Node.js 25.2.1 para o frontend React.

2.3.1 Configuração do Backend

O backend utiliza Django 6.0 com SQLite como banco de dados. Uma característica importante do projeto é que o arquivo de banco de dados db.sqlite3 está incluído no repositório, com as tabelas já criadas e prontas para uso imediato. Isso elimina a necessidade de configuração complexa de banco de dados, permitindo que o ambiente esteja operacional logo após a instalação das dependências.

Para configurar o backend, primeiramente deve-se navegar até o diretório backend e criar um ambiente virtual Python. A criação do ambiente virtual pode ser feita com o comando:

```
python3 -m venv venv
```

Após a criação, o ambiente deve ser ativado. Em sistemas Linux ou Mac, utiliza-se:

```
source venv/bin/activate
```

Em sistemas Windows, o comando de ativação é:

```
venv\Scripts\activate
```

Com o ambiente virtual ativado, as dependências do projeto devem ser instaladas através do arquivo requirements.txt:

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

Opcionalmente, para garantir que o banco de dados está atualizado com as migrações mais recentes, pode-se executar:

```
python manage.py migrate
```

Finalmente, o servidor de desenvolvimento Django pode ser iniciado com:

```
python manage.py runserver
```

2.3.2 Configuração do Frontend

O frontend foi desenvolvido com React 18.3.1 e Vite, utilizando Tailwind CSS 4.1.17, Material-UI 7.3.6 e DaisyUI 5.5.8 para estilização. A configura-

ção deve ser feita em um terminal separado, a partir da raiz do projeto.

Primeiro, as dependências Node.js devem ser instaladas:

```
npm install
```

Após a instalação das dependências, o servidor de desenvolvimento Vite pode ser iniciado:

```
npm run dev
```

2.3.3 Gerenciamento do Banco de Dados

Para facilitar a manutenção do banco de dados, o projeto inclui um script utilitário chamado preparar_banco.sh na raiz do projeto. Este script oferece duas funcionalidades principais: atualizar migrações existentes, aplicando alterações de esquema sem perder dados, e recriar o banco de dados do zero, excluindo o arquivo db.sqlite3 existente e recriando todas as tabelas com um estado limpo.

O script pode ser executado com:

```
./preparar_banco.sh
```

Para acessar funcionalidades administrativas como gerenciamento de contas de usuário, é necessário criar um usuário com permissões de administrador. O projeto inclui um comando Django customizado para facilitar este processo:

```
python manage.py create_admin
```

Este comando cria automaticamente um grupo admin com permissão 1 (administrador) e um usuário padrão com as seguintes credenciais:

- Username: admin
- Password: admin123
- Email: admin@praestitia.com

Adicionalmente, o projeto fornece um comando customizado para popular o banco com dados fictícios para teste. Este comando permite especificar a quantidade de leads e clientes a serem gerados:

```
python manage.py gerar_dados_teste --leads 20 --clientes 10
```

Esta funcionalidade é particularmente útil durante o desenvolvimento e testes, permitindo simular um ambiente com dados realistas sem a necessidade de entrada manual.

3 Funcionamento do Código

3.1 Servidor Django

O servidor Django foi estruturado seguindo o padrão MVT (Model-View-Template), adaptado para uma API REST que serve o frontend React. A arquitetura é organizada em aplicações modulares, cada uma responsável por uma entidade específica do sistema: accounts para autenticação e usuários, leads para gerenciamento de prospectos, clientes para clientes convertidos, e concorrentes para análise de concorrência.

3.1.1 Estrutura de URLs e Roteamento

O roteamento de URLs é centralizado no arquivo praestitia/urls.py, que define os endpoints principais da aplicação. O padrão utilizado emprega a função include() do Django para delegar grupos de URLs a cada aplicação específica, promovendo modularidade e separação de responsabilidades. Os endpoints principais são:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('auth/', include('accounts.urls')),
    path('api/leads/', include('leads.urls')),
    path('api/clientes/', include('clientes.urls')),
    path('api/concorrentes/', include('concorrentes.urls')),
]
```

Esta estrutura permite que cada aplicação mantenha suas próprias rotas de forma independente, facilitando manutenção e escalabilidade. O prefixo api/ foi adotado para endpoints de dados, enquanto auth/ é reservado

para operações de autenticação.

3.1.2 Modelos de Dados

Os modelos Django necessita que os modelo do banco sejam declarados através de classes Python. Cada atributo da classe representa uma coluna na tabela, e o Django automaticamente gera e executa as queries SQL necessárias para criar e manipular essas estruturas.

O modelo Lead exemplifica essa abordagem, definindo campos com tipos específicos e validações:

```
class Lead(models.Model):
    nome = models.CharField(
        max_length=200,
        verbose_name='Nome do Lead',
        help_text='Nome completo do lead'
    )

    email = models.EmailField(
        max_length=254,
        validators=[EmailValidator()],
        verbose_name='E-mail'
    )

    status = models.CharField(
        max_length=20,
        choices=STATUS_CHOICES,
        default='novo'
    )
```

Neste exemplo, CharField define campos de texto com tamanho máximo, EmailField adiciona validação automática de formato de email, e o parâmetro choices restringe os valores possíveis a um conjunto predefinido. O Django traduz automaticamente essas definições em colunas VARCHAR e aplica constraints no banco de dados. Para controlar a data da criação

dos dados, o modelo utiliza campos que se atualizam automaticamente:

```
data_cadastro = models.DateTimeField(auto_now_add=True)
data_atualizacao = models.DateTimeField(auto_now=True)
```

O parâmetro auto_now_add registra o timestamp apenas na criação do registro, enquanto auto_now atualiza o campo toda vez que o objeto é salvo, permitindo rastreamento completo do ciclo de vida dos dados sem intervenção manual.

A otimização de consultas é configurada através da Meta class interna ao modelo:

```
class Meta:
    ordering = ['-data_cadastro']
    indexes = [
        models.Index(fields=['email']),
        models.Index(fields=['status']),
        models.Index(fields=['-data_cadastro']),
    ]
```

O atributo ordering define que consultas sem ordenação explícita retornarão leads ordenados por data de cadastro decrescente (o prefixo - indica ordem descendente). Os indexes criam índices no banco de dados para os campos especificados, funcionando como um catálogo que permite ao banco localizar registros rapidamente sem precisar ler toda a tabela. Por exemplo, uma query como Lead.objects.filter(status='novo') seria muito lenta em uma tabela com milhares de registros se o banco precisasse verificar o campo status de cada linha individualmente. Com o índice, o banco consulta o catálogo diretamente e encontra apenas os registros relevantes, acelerando significativamente a operação.

3.1.3 Sistema de Autenticação

O sistema de autenticação foi implementado utilizando Django Knox, uma biblioteca que fornece autenticação baseada em tokens com segurança aprimorada. Knox gera tokens com hash SHA-512 e permite configura-

ção de expiração, definida em 10 horas no projeto. Diferentemente de sistemas básicos de token, Knox suporta múltiplos tokens por usuário e permite invalidação individual de tokens durante logout.

Para gerenciar usuários, o projeto implementa um modelo User customizado que estende AbstractBaseUser, permitindo controle total sobre os campos e comportamentos de autenticação:

```
class User(AbstractBaseUser):
    id = models.BigAutoField(primary_key=True)
    username = models.CharField(max_length=150, unique=True)
    email = models.EmailField(unique=True)
    group = models.ForeignKey(Group, null=True, blank=True,
                              on_delete=models.SET_NULL)

    objects = UserManager()

    USERNAME_FIELD = 'username'
    REQUIRED_FIELDS = ['email']
```

O atributo USERNAME_FIELD indica que o campo username será usado para login, enquanto REQUIRED_FIELDS especifica campos obrigatórios além da senha. O relacionamento com Group através de ForeignKey permite associar cada usuário a um grupo de permissões.

A criação de usuários é gerenciada por um UserManager personalizado que garante o hash correto de senhas:

```
class UserManager(BaseUserManager):
    def create_user(self, username, email, password=None, **extra_fields):
        if not email:
            raise ValueError('Usuários devem ter um endereço de e-mail.')

        email = self.normalize_email(email)
        user = self.model(username=username, email=email, **extra_fields)
        user.set_password(password)  # Hash automático com Argon2
```

```
        user.save(using=self._db)
        return user
```

O método set_password() automaticamente aplica o algoritmo de hash Argon2 configurado no projeto, garantindo que senhas nunca sejam armazenadas em texto plano. O método normalize_email() converte o domínio do email para lowercase, prevenindo duplicações por diferenças de capitalização.

O sistema de grupos e permissões foi simplificado para atender as necessidades específicas do projeto. O modelo Group armazena grupos de usuários com suas respectivas permissões:

```
class Group(models.Model):
    name = models.CharField(max_length=150, unique=True)
    permissions = models.JSONField(default=list, blank=True)
```

Cada grupo possui um nome único e um campo permissions que utiliza JSONField para armazenar uma lista de inteiros, por exemplo [1, 2, 3], onde cada número representa uma permissão específica do sistema.

A associação entre usuários e grupos é feita através de ForeignKey no modelo User:

```
class User(AbstractBaseUser):
    # ... outros campos ...
    group = models.ForeignKey(Group, null=True, blank=True,
                              on_delete=models.SET_NULL, related_name='users')
```

Esta estrutura permite que múltiplos usuários compartilhem o mesmo conjunto de permissões através do grupo, facilitando o gerenciamento de autorizações.

3.1.4 Views e Processamento de Requisições

As views são funções Python que recebem requisições HTTP do frontend e retornam respostas. Quando o usuário clica em um botão no navegador ou a página carrega dados, o JavaScript envia uma requisição HTTP para

o backend Django, que processa através de uma view específica.

O projeto implementa dois estilos de views. O primeiro são funções simples que recebem o objeto request como parâmetro:

```
@csrf_exempt
@require_POST
def login_view(request):
    data = json.loads(request.body)
    username = data.get('username')
    password = data.get('password')
    # ... lógica de autenticação ...
    return JsonResponse({'status': 200, 'token': token})
```

O segundo estilo utiliza decorators do Django REST Framework, que são anotações que adicionam funcionalidades automáticas à função. Por exemplo, o decorator @api_view(['POST']) define que a função só aceita requisições POST, enquanto @authentication_classes verifica automaticamente se o token enviado é válido:

```
@api_view(['POST'])
@authentication_classes([TokenAuthentication])
@permission_classes([IsAuthenticated])
def validate_token_view(request):
    # Neste ponto, o Django já verificou o token automaticamente
    # Se chegou aqui, o usuário está autenticado
    user = request.user
    return Response({'status': 200, 'username': user.username})
```

Quando uma requisição chega ao endpoint /auth/validate/, o Django executa os decorators na ordem: primeiro verifica se é POST, depois extrai o token do header Authorization, consulta o banco de dados Knox para validar o token e sua expiração, e finalmente carrega o objeto User correspondente. Se qualquer etapa falhar, retorna erro 401 automaticamente sem executar o código da função.

O fluxo de processamento de uma requisição típica segue estas etapas: o

frontend envia dados em formato JSON no corpo da requisição, o Django valida o token Knox verificando se não expirou, a view extrai e valida os dados recebidos, executa a operação no banco de dados (criar, ler, atualizar ou deletar), e retorna uma resposta JSON com o resultado. Códigos HTTP padronizados indicam o resultado: 200 para sucesso, 401 quando o token é inválido ou expirado, 400 quando os dados enviados estão incorretos, e 500 para erros internos do servidor.

3.2 Frontend React

O frontend foi desenvolvido com React, uma biblioteca JavaScript que organiza a interface em componentes reutilizáveis. Cada componente é uma peça independente da interface que pode ser combinada com outras para construir páginas completas. Por exemplo, um botão de deletar é um componente que pode ser usado tanto na página de leads quanto na de clientes.

3.2.1 Estrutura da Aplicação e Fluxo de Autenticação

A aplicação inicia no componente App.jsx, que funciona como o ponto de entrada e gerencia o estado de autenticação de toda a aplicação. O estado de autenticação determina se o usuário vê a tela de login ou o dashboard:

```
function App() {
  const [isAuth, setAuth] = useState(() => {
    // Verifica se o token existe no localStorage ao inicializar
    return !!localStorage.getItem('token_acesso')
  });

  return (
    <ThemeProvider theme={theme}>
      {!isAuth && <AuthenticationPage setAuth={setAuth} />}
      {isAuth && <Dashboard setAuth={setAuth} />}
    </ThemeProvider>
  )
```

```
}
```

O código utiliza useState, um hook do React que cria uma variável de estado isAuth e uma função setAuth para modificá-la. Na inicialização, verifica se existe um token salvo no localStorage do navegador. O operador !! converte o resultado em booleano: se existe token retorna true, caso contrário false.

A renderização condicional usa isAuth para decidir qual página mostrar: se false, exibe AuthenticationPage (tela de login), se true, exibe Dashboard. Ambos os componentes recebem setAuth como propriedade, permitindo que modifiquem o estado de autenticação. Quando o usuário faz login com sucesso, AuthenticationPage chama setAuth(true), fazendo a aplicação re-renderizar e mostrar o Dashboard.

3.2.2 Arquitetura do Dashboard

O Dashboard funciona como um componente container que organiza todas as páginas principais da aplicação. Ele gerencia qual página está sendo exibida através do estado currentPage:

```
export default function Dashboard({ setAuth }) {
    const [currentPage, setCurrentPage] = useState('dashboard');

    const handleNavigate = (page) => {
        if (page === 'logout') {
            removeToken()
            setAuth(false)
            return
        }
        setCurrentPage(page);
    };

    return (
        <Navigator onNavigate={handleNavigate}>
            {currentPage === 'dashboard' && <DashboardStats />}
```

```
            {currentPage === 'cliente' && <Clientes />}
            {currentPage === 'lead' && <Leads />}
            {currentPage === 'concorrente' && <Concorrentes />}
        </Navigator>
    );
}
```

O componente recebe setAuth como propriedade do App.jsx e o repassa para o Navigator, criando uma cadeia de comunicação. Quando o usuário clica em "Sair"no menu, o Navigator chama handleNavigate('logout'), que remove o token do localStorage e chama setAuth(false), retornando o usuário à tela de login.

A função handleNavigate centraliza a lógica de navegação. Para páginas normais, apenas atualiza currentPage, fazendo o React re-renderizar e exibir o componente correspondente. Para logout, executa duas ações críticas: removeToken() limpa o token armazenado localmente, e setAuth(false) propaga a mudança até o componente App, que então exibe novamente a tela de login.

3.2.3 Verificação Automática de Autenticação

O Dashboard implementa um sistema de verificação periódica para detectar tokens expirados automaticamente:

```
useEffect(() => {
    const checkAuth = async () => {
        const response = await authenticatedFetch('/auth/validate/', {
            method: 'POST'
        });

        if (response.status === 401) {
            removeToken()
            setAuth(false)
        }
    };
```

```
    checkAuth();  // Executa imediatamente
    const intervalId = setInterval(checkAuth, 10000);  // Repete a cada 10s

    return () => clearInterval(intervalId);  // Cleanup
}, [setAuth]);
```

A função checkAuth envia uma requisição ao backend para validar o token atual. Se retornar 401 (não autorizado), significa que o token expirou, então remove o token e desloga o usuário automaticamente.

O setInterval cria um timer que executa checkAuth a cada 10 segundos, verificando continuamente se o token ainda é válido. A função de cleanup retornada pelo useEffect é executada quando o componente é desmontado, removendo o timer para evitar vazamento de memória e requisições desnecessárias.

3.2.4 Componentes Reutilizáveis e Comunicação com API

Os componentes reutilizáveis encapsulam funcionalidades específicas que são usadas em múltiplas páginas. O EntityForm exemplifica como um único componente pode ser configurado para diferentes propósitos:

```
const EntityForm = ({
    entityType,
    apiEndpoint,
    fields,
    onSubmit,
    setRefresh,
    method = 'POST'
}) => {
    const [formData, setFormData] = useState({});

    async function handleSubmit(e) {
        e.preventDefault();
        const response = await authenticatedFetch(apiEndpoint, {
```

```
            method: method,
            body: JSON.stringify(formData),
        });

        if (response.ok) {
            setRefresh((prev) => prev + 1);
            onSubmit?.();
        }
    }

    return (
        <form onSubmit={handleSubmit}>
            {/* Renderiza campos dinamicamente */}
        </form>
    );
};
```

O componente recebe configurações como propriedades: entityType define o tipo de entidade (lead, cliente), apiEndpoint especifica a URL da API, fields lista os campos do formulário, e method define se é criação (POST) ou edição (PUT). Esta abordagem permite usar o mesmo componente para criar leads, editar clientes, ou qualquer outra operação CRUD.

A função authenticatedFetch é um utilitário centralizado que adiciona automaticamente o token Knox em todas as requisições, evitando repetição de código. Após submeter com sucesso, setRefresh incrementa um contador que força a página pai a recarregar os dados, exibindo a nova informação criada ou modificada.

3.2.5 Autenticação Knox nas Views

Um aspecto importante da implementação é a correta autenticação de requisições protegidas. O projeto utiliza dois padrões distintos para autenticar views: o padrão manual e o padrão com decorators do Django REST Framework.

Inicialmente, algumas views implementavam autenticação manual através de uma função authenticate_request:

```
def authenticate_request(request):
    auth_header = request.headers.get('Authorization')
    if not auth_header or not auth_header.startswith('Token '):
        return None

    token = auth_header.split(' ')[1]
    auth_token = AuthToken.objects.filter(token_key=token[:8]).first()

    if not auth_token or auth_token.expiry < timezone.now():
        return None

    return auth_token.user
```

Esta abordagem apresentava problemas porque tentava validar tokens Knox manualmente usando apenas os primeiros 8 caracteres (token_key). Knox armazena tokens em formato hash por segurança, e a validação manual não considerava toda a complexidade do sistema de autenticação da biblioteca.

A solução correta utiliza os decorators fornecidos pelo Django REST Framework em conjunto com a classe TokenAuthentication do Knox:

```
@api_view(['GET', 'POST'])
@authentication_classes([TokenAuthentication])
@permission_classes([IsAuthenticated])
def users_list_view(request):
    user = request.user

    if not user.group or 1 not in user.group.permissions:
        return JsonResponse({'status': 403, 'message': 'Sem permissão.'})

    # Lógica da view...
```

Os decorators funcionam em camadas sequenciais. O @api_view define

os métodos HTTP permitidos e transforma a função em uma view compatível com Django REST Framework. O @authentication_classes([TokenAuthentication]) instrui o Django a usar o sistema de autenticação Knox, que extrai automaticamente o token do header Authorization (formato "Token <token>"), valida o hash contra o banco de dados, verifica a expiração, e carrega o usuário correspondente no atributo request.user. Por fim, @permission_classes([IsAuthenticated]) garante que apenas requisições autenticadas prossigam, retornando 401 automaticamente se o token for inválido.

Esta abordagem delega toda a complexidade de validação de tokens para o Knox, garantindo que todos os aspectos de segurança sejam tratados corretamente, incluindo validação de hash, verificação de expiração, e proteção contra tokens inválidos ou adulterados.

3.3 Entidades do Sistema

O sistema implementa quatro entidades principais, cada uma com funcionalidades CRUD completas: Leads, Clientes, Concorrentes e Contas (usuários). Todas as entidades seguem padrões consistentes de implementação, mas possuem propósitos e campos específicos para suas funcionalidades.

3.3.1 Leads

Leads representam potenciais clientes que demonstraram interesse inicial no produto ou serviço. Esta entidade é tipicamente o ponto de entrada do funil de vendas, armazenando informações de contatos que ainda não foram convertidos em clientes efetivos.

O modelo Lead possui os seguintes campos principais:

```
class Lead(models.Model):
    nome = models.CharField(max_length=200)
    email = models.EmailField(validators=[EmailValidator()])
    telefone = models.CharField(max_length=20, blank=True)
    empresa = models.CharField(max_length=200, blank=True)
```

```
    status = models.CharField(
        max_length=20,
        choices=[
            ('novo', 'Novo'),
            ('contatado', 'Contatado'),
            ('qualificado', 'Qualificado'),
            ('convertido', 'Convertido'),
            ('perdido', 'Perdido')
        ],
        default='novo'
    )
    origem = models.CharField(max_length=100, blank=True)
    observacoes = models.TextField(blank=True)
    data_cadastro = models.DateTimeField(auto_now_add=True)
    data_atualizacao = models.DateTimeField(auto_now=True)
```

O campo status permite rastrear o progresso do lead através do funil de vendas, desde o contato inicial até a conversão ou perda. O campo origem registra de onde o lead veio (site, indicação, evento), permitindo análise da efetividade de diferentes canais de captação.

3.3.2 Clientes

Clientes representam leads convertidos ou contatos que já realizaram negócios com a empresa. Esta entidade armazena informações mais detalhadas e comerciais do que a entidade Lead.

```
class Cliente(models.Model):
    nome = models.CharField(max_length=200)
    email = models.EmailField(validators=[EmailValidator()])
    telefone = models.CharField(max_length=20, blank=True)
    empresa = models.CharField(max_length=200, blank=True)
    cnpj = models.CharField(max_length=18, blank=True)
    endereco = models.CharField(max_length=300, blank=True)
    cidade = models.CharField(max_length=100, blank=True)
```

```
    estado = models.CharField(max_length=2, blank=True)
    valor_contrato = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        null=True,
        blank=True
    )
    data_inicio = models.DateField(null=True, blank=True)
    observacoes = models.TextField(blank=True)
    data_cadastro = models.DateTimeField(auto_now_add=True)
    data_atualizacao = models.DateTimeField(auto_now=True)
```

A entidade Cliente inclui campos comerciais como cnpj, endereco completo, e valor_contrato. O campo valor_contrato usa DecimalField para garantir precisão em valores monetários, evitando problemas de arredondamento associados a floats.

3.3.3 Concorrentes

A entidade Concorrentes permite análise competitiva, armazenando informações sobre empresas que competem no mesmo mercado. Esta funcionalidade auxilia no posicionamento estratégico e acompanhamento do cenário competitivo.

```
class Concorrente(models.Model):
    nome = models.CharField(max_length=200)
    site = models.URLField(blank=True)
    descricao = models.TextField(blank=True)
    produtos_servicos = models.TextField(blank=True)
    pontos_fortes = models.TextField(blank=True)
    pontos_fracos = models.TextField(blank=True)
    faixa_preco = models.CharField(max_length=100, blank=True)
    market_share = models.CharField(max_length=50, blank=True)
    observacoes = models.TextField(blank=True)
    data_cadastro = models.DateTimeField(auto_now_add=True)
```

```
    data_atualizacao = models.DateTimeField(auto_now=True)
```

Os campos pontos_fortes e pontos_fracos permitem análise SWOT simplificada, enquanto faixa_preco e market_share auxiliam no posicionamento competitivo e estratégias de precificação.

3.3.4 Contas (Usuários)

A entidade de Contas gerencia os usuários do sistema, controlando autenticação e permissões. Esta é a única entidade com controle de acesso baseado em permissões administrativas.

O modelo User já foi detalhado anteriormente, mas vale destacar sua integração com o sistema de permissões. A view users_list_view exemplifica o controle de acesso:

```
@api_view(['GET', 'POST'])
@authentication_classes([TokenAuthentication])
@permission_classes([IsAuthenticated])
def users_list_view(request):
    user = request.user

    # Verifica se o usuário tem permissão de administrador
    if not user.group or 1 not in user.group.permissions:
        return JsonResponse({
            'status': 403,
            'message': 'Sem permissão.'
        }, status=403)

    if request.method == 'GET':
        users = User.objects.select_related('group').all()
        # Retorna lista de todos os usuários
```

A verificação "1 not in user.group.permissions"implementa controle de acesso baseado em permissões, onde o número 1 representa a permissão de administrador. Apenas usuários com esta permissão podem visualizar

e gerenciar contas.

No frontend, o componente Contas em src/pages/dashboard/contas/ implementa a interface de gerenciamento, e o Navigator só exibe o menu "Contas"para usuários com permissão administrativa:

```
const Navigator = ({ userPermissions = [] }) => {
  const hasAdminPermission = userPermissions.includes(1);

  return (
    {/* Outros menus */}

    {hasAdminPermission && (
      <li>
        <button onClick={() => onNavigate('contas')}>
          Contas
        </button>
      </li>
    )}
  );
};
```

Esta arquitetura garante que funcionalidades administrativas sejam acessíveis apenas por usuários autorizados, implementando segurança tanto no backend (validação de permissões nas views) quanto no frontend (ocultação de elementos da interface).

4 Conclusão

O desenvolvimento deste projeto permitiu a aplicação prática de conceitos fundamentais de engenharia de software, integrando conhecimentos de Programação II, Banco de Dados e Engenharia de Software em uma solução web completa e funcional. A implementação bem-sucedida de um sistema de gerenciamento de leads e clientes demonstra a viabilidade de construir aplicações profissionais utilizando tecnologias modernas e

padrões consolidados da indústria.

Como trabalhos futuros, o sistema poderia ser expandido com funcionalidades adicionais como notificações em tempo real, relatórios exportáveis em PDF, integração com serviços externos de email, e implementação de testes automatizados. A parte que possui mais potencial, ao meu ver, de ser expandida, envolve o estudo de concorrentes dentro do aplicativo. A migração do banco de dados SQLite para PostgreSQL seria preferível ao uso atual do SQLite.

Em suma, este projeto representou uma experiência de aprendizado significativa, consolidando conhecimentos teóricos em uma aplicação prática e funcional, preparando para desafios futuros no desenvolvimento de sistemas web complexos.