**Tutorial de Desenvolvimento: Sistema de Gestão de Pedidos com Django**

**Uma adaptação ao material da FAP 2024**

Resumo - Por José Alfredo F. Costa - Novembro de 2024

**1. Introdução**

Este projeto tem como objetivo criar um sistema de gestão de pedidos para uma loja online. Ele será desenvolvido com Django, aproveitando suas funcionalidades robustas para a construção de aplicações web. O sistema permitirá a gestão de usuários (administradores da loja), cadastro de clientes, criação e visualização de pedidos, e controle de produtos disponíveis.

**1.1 Objetivos do Sistema**

1. **Gerenciamento de Usuários**: Administradores podem criar e gerenciar contas de forma segura, com autenticação.
2. **Cadastro de Clientes**: Permite o registro de clientes com informações básicas.
3. **Processamento de Pedidos**: Clientes podem fazer pedidos, que serão armazenados com detalhes de data e produtos.
4. **Detalhes dos Pedidos**: Cada pedido pode ter múltiplos produtos, com valores e descontos individuais.
5. **Catálogo de Produtos**: Cadastro e listagem de produtos, facilitando a adição de novos itens à loja.

**2. Estrutura do Projeto**

A estrutura do projeto no Django incluirá os seguintes diretórios e arquivos principais:

SGE\_Django/

├── config/ # Configurações do Django

├── controllers/ # Lógica dos controladores (views)

├── models/ # Modelos de dados (entidades do sistema)

├── templates/ # Templates HTML para o frontend

├── static/ # Arquivos estáticos (CSS, JavaScript)

├── routes/ # URLs e rotas do sistema

├── manage.py # Script de gerenciamento do Django

└── settings.py # Configurações do projeto

**3. Modelos de Dados**

Para construir o sistema, serão criados cinco modelos principais. Vamos definir cada um deles com os campos necessários.

**3.1 Usuário**

Modelo para gerenciar os administradores da loja.

python

from django.contrib.auth.models import User

Utilizaremos o modelo User nativo do Django para autenticação e gestão de administradores.

**3.2 Cliente**

Modelo para representar os clientes que realizam pedidos.

python

from django.db import models

class Cliente(models.Model):

cliente\_id = models.AutoField(primary\_key=True)

cliente\_nome = models.CharField(max\_length=255)

cliente\_email = models.EmailField(unique=True)

**3.3 Pedido**

Modelo para representar o pedido realizado por um cliente.

python

class Pedido(models.Model):

pedido\_id = models.AutoField(primary\_key=True)

cliente = models.ForeignKey(Cliente, on\_delete=models.CASCADE)

data\_compra = models.DateTimeField(auto\_now\_add=True)

**3.4 DetalhePedido**

Modelo para armazenar os detalhes de cada produto em um pedido.

python

class DetalhePedido(models.Model):

dt\_id = models.AutoField(primary\_key=True)

pedido = models.ForeignKey(Pedido, on\_delete=models.CASCADE)

produto = models.ForeignKey('Produto', on\_delete=models.CASCADE)

dt\_valor = models.DecimalField(max\_digits=10, decimal\_places=2)

dt\_desconto = models.DecimalField(max\_digits=5, decimal\_places=2, default=0)

**3.5 Produto**

Modelo para representar produtos disponíveis na loja.

python

class Produto(models.Model):

produto\_id = models.AutoField(primary\_key=True)

produto\_nome = models.CharField(max\_length=255)

**4. Configuração das Rotas e Views**

No Django, as rotas são definidas nos arquivos de urls.py dentro dos apps, enquanto as views controlam a lógica de negócios.

**4.1 Rotas para Clientes**

Em urls.py do app clientes:

python

from django.urls import path

from . import views

urlpatterns = [

path('clientes/', views.lista\_clientes, name='lista\_clientes'),

path('clientes/<int:id>/', views.detalhe\_cliente, name='detalhe\_cliente'),

]

**4.2 Views para Clientes**

Em views.py:

python

from django.shortcuts import render

from .models import Cliente

def lista\_clientes(request):

clientes = Cliente.objects.all()

return render(request, 'clientes/lista.html', {'clientes': clientes})

def detalhe\_cliente(request, id):

cliente = Cliente.objects.get(id=id)

return render(request, 'clientes/detalhe.html', {'cliente': cliente})

**5. Funcionalidades do Sistema**

**5.1 Cadastro de Usuários e Autenticação**

Usaremos o sistema de autenticação nativo do Django, configurando o User para gerenciar logins e permissões dos administradores.

**5.2 Interface de Cadastro e Visualização de Clientes**

Templates HTML para a exibição e gerenciamento dos clientes.

**5.3 Criação e Visualização de Pedidos**

Permite que o administrador veja todos os pedidos, com detalhes de produtos e descontos aplicados.

**5.4 Registro e Exibição de Produtos**

Cadastro de produtos, possibilitando gerenciar o inventário da loja.

Esse tutorial oferece uma visão inicial da estrutura e do desenvolvimento do sistema de gestão de pedidos utilizando Django, com foco em cadastro e controle de pedidos, clientes e produtos​(Cópia de Projeto SGP)​(FAP JS PYTHON FLASK M07…).

**Vantagens de usar o Django em relação a Flask para esse projeto**

Django e Flask são frameworks populares em Python para desenvolvimento web, mas possuem características distintas que os tornam mais adequados para diferentes tipos de projetos. Aqui estão as vantagens de usar Django em relação ao Flask, exemplificadas para o projeto de Sistema de Gestão de Pedidos (SGP):

**1. Estrutura Completa e Opiniada**

* **Django** é um framework "batteries-included" (baterias inclusas), ou seja, já vem com várias funcionalidades integradas, como ORM (Object-Relational Mapping), sistema de autenticação, painel administrativo e ferramentas de segurança. Isso significa que grande parte da estrutura já está pronta, acelerando o desenvolvimento.
* No projeto SGP, com Django, temos rapidamente a autenticação de usuários e o sistema administrativo, onde o administrador pode gerenciar clientes, pedidos e produtos sem precisar implementar uma interface de administração do zero.

**2. Sistema de Autenticação Integrado**

* Django oferece um sistema de autenticação robusto e completo nativamente, o que facilita o controle de acesso e a criação de usuários com diferentes níveis de permissão.
* No contexto do SGP, o sistema de autenticação permitirá que apenas os administradores da loja acessem certas funcionalidades, como a criação e modificação de produtos e pedidos, sem a necessidade de configurar uma camada de autenticação do zero.

**3. Painel Administrativo Padrão**

* Django possui um painel administrativo pronto para uso, acessível imediatamente após configurar o modelo de dados. Esse painel permite gerenciar todas as entidades (clientes, pedidos, produtos) de forma amigável e com uma interface gráfica.
* Para o SGP, o administrador pode facilmente visualizar e editar informações dos clientes, verificar pedidos e gerenciar o catálogo de produtos diretamente no painel administrativo, sem a necessidade de desenvolver um frontend específico para isso.

**4. ORM Integrado**

* Django possui um ORM poderoso que facilita o mapeamento de modelos Python para tabelas de banco de dados. Isso torna a interação com o banco de dados mais simples e evita o uso de SQL em várias situações.
* No projeto SGP, o ORM permite criar, buscar, atualizar e deletar registros (clientes, produtos, pedidos) usando apenas código Python, sem necessidade de escrever SQL diretamente, o que reduz a complexidade e chances de erro.

**5. Segurança Integrada**

* Django vem com várias configurações de segurança integradas, como proteção contra CSRF (Cross-Site Request Forgery), SQL Injection e XSS (Cross-Site Scripting). Essas proteções tornam a aplicação mais segura contra ataques comuns.
* No SGP, onde informações sensíveis dos clientes e dados de pedidos são armazenados, essas proteções de segurança são essenciais para garantir a confidencialidade e integridade das informações.

**6. Comunidade e Ecossistema de Plugins**

* Django possui uma comunidade ativa e um ecossistema vasto de plugins e bibliotecas prontas para integração. Isso significa que é possível encontrar pacotes já prontos para implementar funcionalidades adicionais.
* No caso do SGP, se precisar de funcionalidades extras, como pagamento online ou integração com sistemas externos, é provável que já existam bibliotecas e plugins que podem ser facilmente integrados com Django.

**7. Estrutura Modular (Apps)**

* Django utiliza uma estrutura modular chamada "apps", que facilita a divisão do projeto em módulos menores e mais organizados. Isso ajuda no desenvolvimento colaborativo e na manutenção.
* No SGP, podemos criar apps distintos para cada entidade principal, como clientes, pedidos e produtos, mantendo o código organizado e facilitando futuras expansões e manutenções.

**8. Rapidez no Desenvolvimento de Funcionalidades**

* Como Django já oferece uma estrutura completa e padrões estabelecidos, o desenvolvimento tende a ser mais rápido para projetos que requerem funcionalidades típicas de aplicações web.
* No SGP, em vez de gastar tempo configurando autenticação, painel administrativo e ORM, o desenvolvedor pode focar nas funcionalidades específicas do sistema, como o cadastro de produtos e o processamento de pedidos.

**Comparação com Flask para o SGP**

* **Flask** é um microframework, o que significa que ele é mais flexível e minimalista. Ele permite maior personalização, mas exige que o desenvolvedor implemente (ou integre) muitas funcionalidades essenciais, como autenticação, painel administrativo e ORM.
* Para o SGP, isso significaria maior tempo de desenvolvimento e mais código personalizado, especialmente para segurança, autenticação e gerenciamento de banco de dados, pois teríamos que construir ou configurar essas partes de forma independente.

**Conclusão**

Para o Sistema de Gestão de Pedidos, Django oferece uma estrutura mais robusta e rápida para implementar todas as funcionalidades essenciais, além de manter o sistema seguro e escalável. A escolha do Django possibilita que o projeto seja desenvolvido com maior velocidade e segurança, além de fornecer uma estrutura pronta para crescimento futuro, caso novas funcionalidades sejam necessárias.

**Análise do FAP JS FLASK SEGURANÇA M07A03.pptx**

Para o sistema de gestão de pedidos, vamos implementar uma estrutura de autenticação com segurança utilizando JWT (JSON Web Tokens) primeiro em Flask e depois em Django. Após isso, analisaremos as vantagens do uso do Django neste cenário.

**Implementação em Flask com JWT**

Para implementar a autenticação com JWT no Flask, utilizaremos a biblioteca Flask-JWT-Extended, que facilita a criação e verificação de tokens JWT para autenticação de APIs. O processo inclui a criação de um ambiente virtual, configuração de rotas de login, proteção de rotas com JWT e exemplos de código para cada etapa.

**Passo a Passo**

1. **Criar o Ambiente Virtual**

python3 -m venv venv

source venv/bin/activate # Para Linux/Mac

venv\Scripts\activate # Para Windows

1. **Instalar as Bibliotecas Necessárias**

pip install Flask Flask-JWT-Extended

1. **Configurar o Aplicativo Flask**

No arquivo principal (app.py), vamos configurar o JWT:

python

from flask import Flask, jsonify, request

from flask\_jwt\_extended import JWTManager, create\_access\_token, jwt\_required

app = Flask(\_\_name\_\_)

app.config['JWT\_SECRET\_KEY'] = 'chave\_secreta\_para\_token'

jwt = JWTManager(app)

1. **Criar Rota de Login**

Nesta rota, verificamos as credenciais do usuário e geramos um token JWT se as credenciais forem válidas.

python

@app.route('/login', methods=['POST'])

def login():

dados = request.get\_json()

if dados['usuario'] == 'admin' and dados['senha'] == 'senha123':

access\_token = create\_access\_token(identity=dados['usuario'])

return jsonify(access\_token=access\_token), 200

return jsonify({"msg": "Credenciais inválidas"}), 401

1. **Proteger Rotas com JWT**

Utilizamos o decorator @jwt\_required para proteger rotas específicas, como a visualização de pedidos e detalhes.

python

@app.route('/pedidos', methods=['GET'])

@jwt\_required()

def listar\_pedidos():

# Exemplo de retorno de pedidos

pedidos = [{"pedido\_id": 1, "cliente": "Cliente 1", "data\_compra": "2024-11-06"}]

return jsonify(pedidos=pedidos)

1. **Executar o Aplicativo**

flask run

Neste exemplo em Flask, implementamos a autenticação básica com JWT, permitindo que apenas usuários autenticados acessem a rota /pedidos.

**Implementação em Django com JWT**

No Django, utilizamos o Django Rest Framework (DRF) em conjunto com o djangorestframework-simplejwt para implementar autenticação com JWT de forma simples e escalável. O processo é semelhante ao do Flask, mas aproveitamos o DRF para gerenciar a API e o sistema de autenticação.

**Passo a Passo**

1. **Criar o Ambiente Virtual**

python3 -m venv venv

source venv/bin/activate

1. **Instalar as Bibliotecas Necessárias**

pip install django djangorestframework djangorestframework-simplejwt

1. **Configurar o Django e DRF**

No arquivo settings.py, adicionamos o DRF e configuramos o JWT:

python

INSTALLED\_APPS = [

# Outros apps

'rest\_framework',

'rest\_framework\_simplejwt',

]

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': (

'rest\_framework\_simplejwt.authentication.JWTAuthentication',

),

}

1. **Configurar URLs e Views para Login**

No arquivo urls.py, configuramos as rotas de autenticação.

python

from django.urls import path

from rest\_framework\_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [

path('api/token/', TokenObtainPairView.as\_view(), name='token\_obtain\_pair'),

path('api/token/refresh/', TokenRefreshView.as\_view(), name='token\_refresh'),

]

1. **Proteger Rotas com JWT**

No Django, utilizamos o DRF para criar views protegidas. Apenas usuários autenticados com um token JWT válido poderão acessar a rota.

python

from rest\_framework.permissions import IsAuthenticated

from rest\_framework.views import APIView

from rest\_framework.response import Response

class PedidoView(APIView):

permission\_classes = [IsAuthenticated]

def get(self, request):

pedidos = [{"pedido\_id": 1, "cliente": "Cliente 1", "data\_compra": "2024-11-06"}]

return Response({"pedidos": pedidos})

1. **Executar o Servidor Django**

python manage.py runserver

Neste exemplo em Django, o DRF e o simplejwt fornecem um sistema robusto para autenticação e autorização. Utilizamos o sistema de permissões do Django para garantir que apenas usuários autenticados acessem as rotas protegidas.

**Vantagens do Uso do Django para o Sistema de Gestão de Pedidos**

1. **Integração com DRF e Suporte Nativo a APIs RESTful**
   * Django, com o DRF, fornece um sistema completo e nativo para criação de APIs RESTful, que é ideal para projetos que exigem segurança e padronização.
   * No SGP, essa estrutura é vantajosa para construir uma API escalável, com autenticação pronta e controles de permissões.
2. **Painel Administrativo para Gestão de Usuários e Dados**
   * Django oferece um painel administrativo nativo, que facilita o gerenciamento de usuários, pedidos e produtos. Isso é útil para o administrador do sistema, permitindo visualização e edição de dados sem necessidade de uma interface adicional.
   * No Flask, seria necessário desenvolver ou integrar um painel administrativo, aumentando o esforço de desenvolvimento.
3. **Controle de Acesso e Permissões Granulares**
   * Com o DRF, é possível definir permissões específicas para cada rota e controle de acesso detalhado, facilitando a personalização das permissões para diferentes tipos de usuários.
   * No Flask, o controle de permissões requer desenvolvimento adicional, pois o framework não possui uma estrutura nativa para isso.
4. **Organização Modular e Manutenibilidade**
   * Django possui uma estrutura de "apps" que organiza o código de maneira modular, facilitando a manutenção e a escalabilidade do sistema.
   * No Flask, a estrutura é mais flexível, mas também pode se tornar desorganizada em projetos maiores, exigindo mais cuidados na organização do código.
5. **Suporte a JWT e Outros Métodos de Autenticação**
   * Embora o Flask permita a implementação de JWT com a biblioteca Flask-JWT-Extended, o Django e o DRF possuem suporte nativo e integrado para JWT e outros métodos de autenticação (como OAuth).
   * No SGP, esse suporte integrado permite maior flexibilidade e segurança, além de menos dependência de bibliotecas externas.
6. **Documentação e Comunidade Ativa**
   * Django tem uma comunidade ativa e vasta documentação, o que facilita o desenvolvimento, resolução de problemas e implementação de melhores práticas.
   * Para projetos como o SGP, uma documentação sólida e o suporte da comunidade ajudam na resolução de dúvidas e garantem um desenvolvimento mais seguro e estruturado.

**Conclusão**

Django oferece uma estrutura mais robusta e escalável para o desenvolvimento de um sistema de gestão de pedidos. A integração com o DRF facilita a criação de APIs, o controle de permissões e a autenticação com JWT. Além disso, o painel administrativo nativo do Django é uma vantagem significativa para a gestão de dados do sistema, algo que exigiria mais trabalho para implementar em Flask.

**Sistema de gestão de pedidos com autenticação JWT em Django**

Passos necessários desde a criação do projeto até a configuração de segurança e autenticação. Esta explicação cobrirá desde a criação do ambiente até a implementação de cada funcionalidade com segurança JWT.

**Passo 1: Criar o Projeto Django e o Ambiente Virtual**

Primeiro, precisamos criar um ambiente virtual para instalar as dependências do projeto e criar o projeto Django.

# Criar o ambiente virtual

python3 -m venv venv

source venv/bin/activate # Para Linux/Mac

venv\Scripts\activate # Para Windows

# Instalar Django e Django REST Framework

pip install django djangorestframework djangorestframework-simplejwt

Agora, vamos criar o projeto Django.

# Criar o projeto Django

django-admin startproject sgp\_django

cd sgp\_django

# Criar o aplicativo para o sistema de pedidos

python manage.py startapp pedidos

**Passo 2: Configurar o Django e o Django REST Framework (DRF)**

No arquivo settings.py, adicione os aplicativos rest\_framework e rest\_framework\_simplejwt para habilitar o uso do DRF e JWT.

python

# settings.py

INSTALLED\_APPS = [

# Apps padrões do Django

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

# Apps do projeto

'pedidos',

# Django REST Framework e JWT

'rest\_framework',

'rest\_framework\_simplejwt',

]

# Configurações do Django REST Framework

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': (

'rest\_framework\_simplejwt.authentication.JWTAuthentication',

),

}

Configurar o DRF com JWT permite que usemos autenticação com tokens JWT em nossas APIs de maneira segura.

**Passo 3: Configurar URLs para o JWT e Login**

Vamos configurar as URLs para gerar tokens JWT e proteger as rotas com autenticação.

No arquivo urls.py do projeto principal, inclua as rotas para obtenção de tokens JWT.

python

# sgp\_django/urls.py

from django.contrib import admin

from django.urls import path, include

from rest\_framework\_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [

path('admin/', admin.site.urls),

path('api/token/', TokenObtainPairView.as\_view(), name='token\_obtain\_pair'),

path('api/token/refresh/', TokenRefreshView.as\_view(), name='token\_refresh'),

path('pedidos/', include('pedidos.urls')), # Incluir as URLs do app "pedidos"

]

* TokenObtainPairView é usado para gerar o token JWT ao fazer login.
* TokenRefreshView permite atualizar o token JWT antes que ele expire.

**Passo 4: Criar Modelos para as Entidades do Sistema**

No app pedidos, vamos criar os modelos para Cliente, Produto, Pedido e DetalhePedido.

python

# pedidos/models.py

from django.db import models

from django.contrib.auth.models import User

class Cliente(models.Model):

nome = models.CharField(max\_length=255)

email = models.EmailField(unique=True)

def \_\_str\_\_(self):

return self.nome

class Produto(models.Model):

nome = models.CharField(max\_length=255)

preco = models.DecimalField(max\_digits=10, decimal\_places=2)

def \_\_str\_\_(self):

return self.nome

class Pedido(models.Model):

cliente = models.ForeignKey(Cliente, on\_delete=models.CASCADE)

data\_compra = models.DateTimeField(auto\_now\_add=True)

def \_\_str\_\_(self):

return f"Pedido {self.id} - {self.cliente.nome}"

class DetalhePedido(models.Model):

pedido = models.ForeignKey(Pedido, on\_delete=models.CASCADE, related\_name='detalhes')

produto = models.ForeignKey(Produto, on\_delete=models.CASCADE)

quantidade = models.IntegerField()

desconto = models.DecimalField(max\_digits=5, decimal\_places=2, default=0.00)

def \_\_str\_\_(self):

return f"{self.produto.nome} - Pedido {self.pedido.id}"

Após definir os modelos, execute as migrações para criar as tabelas no banco de dados:

python manage.py makemigrations

python manage.py migrate

**Passo 5: Criar Serializers para os Modelos**

Os serializers são responsáveis por converter os dados dos modelos para JSON, permitindo que eles sejam utilizados nas APIs.

python

# pedidos/serializers.py

from rest\_framework import serializers

from .models import Cliente, Produto, Pedido, DetalhePedido

class ClienteSerializer(serializers.ModelSerializer):

class Meta:

model = Cliente

fields = ['id', 'nome', 'email']

class ProdutoSerializer(serializers.ModelSerializer):

class Meta:

model = Produto

fields = ['id', 'nome', 'preco']

class PedidoSerializer(serializers.ModelSerializer):

detalhes = serializers.StringRelatedField(many=True)

class Meta:

model = Pedido

fields = ['id', 'cliente', 'data\_compra', 'detalhes']

class DetalhePedidoSerializer(serializers.ModelSerializer):

class Meta:

model = DetalhePedido

fields = ['id', 'pedido', 'produto', 'quantidade', 'desconto']

**Passo 6: Criar Views para as Funcionalidades de API**

Em views.py, vamos criar as views que gerenciarão as operações CRUD para os clientes, produtos e pedidos.

python

# pedidos/views.py

from rest\_framework import viewsets

from rest\_framework.permissions import IsAuthenticated

from .models import Cliente, Produto, Pedido, DetalhePedido

from .serializers import ClienteSerializer, ProdutoSerializer, PedidoSerializer, DetalhePedidoSerializer

class ClienteViewSet(viewsets.ModelViewSet):

queryset = Cliente.objects.all()

serializer\_class = ClienteSerializer

permission\_classes = [IsAuthenticated]

class ProdutoViewSet(viewsets.ModelViewSet):

queryset = Produto.objects.all()

serializer\_class = ProdutoSerializer

permission\_classes = [IsAuthenticated]

class PedidoViewSet(viewsets.ModelViewSet):

queryset = Pedido.objects.all()

serializer\_class = PedidoSerializer

permission\_classes = [IsAuthenticated]

class DetalhePedidoViewSet(viewsets.ModelViewSet):

queryset = DetalhePedido.objects.all()

serializer\_class = DetalhePedidoSerializer

permission\_classes = [IsAuthenticated]

**Passo 7: Configurar URLs do App pedidos**

No arquivo urls.py dentro do app pedidos, vamos registrar as rotas para os viewsets criados.

python

# pedidos/urls.py

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from . import views

router = DefaultRouter()

router.register(r'clientes', views.ClienteViewSet)

router.register(r'produtos', views.ProdutoViewSet)

router.register(r'pedidos', views.PedidoViewSet)

router.register(r'detalhes', views.DetalhePedidoViewSet)

urlpatterns = [

path('', include(router.urls)),

]

**Passo 8: Testar a Autenticação com JWT**

Agora, ao iniciar o servidor e acessar as rotas, a autenticação será necessária para visualizar ou manipular dados. O fluxo de autenticação funciona da seguinte forma:

1. **Obter o Token JWT**: O usuário faz uma solicitação POST para /api/token/ com as credenciais (nome de usuário e senha).

json

{

"username": "admin",

"password": "senha"

}

1. **Utilizar o Token nas Requisições**: O token JWT recebido deve ser enviado no cabeçalho de autorização das requisições protegidas.

http

Authorization: Bearer <token\_jwt>

**Vantagens do Uso do Django para o Sistema de Gestão de Pedidos**

* **Painel Administrativo**: O Django oferece um painel administrativo completo, facilitando a gestão de clientes, produtos e pedidos sem necessidade de criar interfaces específicas.
* **Segurança**: O Django REST Framework e o simplejwt proporcionam uma segurança robusta e controle de permissões mais refinado do que o Flask.
* **Modularidade e Organização**: O Django organiza o projeto de forma modular com "apps", facilitando a manutenção e escalabilidade.
* **Documentação e Comunidade Ativa**: O Django possui ampla documentação e uma comunidade ativa, tornando o desenvolvimento mais seguro e facilitado.

Essa estrutura e implementação proporcionam um sistema robusto, seguro e escalável, ideal para um sistema de gestão de pedidos.