Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Lamego

P. Viseu
Tecnologia e Gestão Lamego

Fundamentos de Programação

Elaboração de uma loja online "ClickFarma"

Lamego, 14 de janeiro de 2024

Fundamentos de Programação

Elaboração de uma loja online "ClickFarma"

CTESP: Tecnologias e Programação de Sistemas de Informação

Unidade Curricular: Fundamentos de Programação

Professor: Carlos Costa

Pv28311 Miguel Rocha

Pv28430 Rita Moreira

Pv29101 Ana Matos

Lamego, 14 de janeiro de 2024

ENQUADRAMENTO

Este trabalho foi realizado no âmbito da disciplina de Fundamentos de Programação em que houve uma implementação de uma aplicação em Python.

O nosso grupo realizou o trabalho de uma Farmácia Online (Clickfarma) em Django.

O Django foi inventado por Lawrence Journal-World em 2003, para atender aos curtos prazos dos jornais e ao mesmo tempo atender às necessidades de programadores web experientes.

Utilizamos esta framework porque facilita a criação de sites usando Python. O Django enfatiza a reutilização de componentes, também conhecida como DRY (Don't Repeat Yourself) e vem com recursos prontos a utilizar como sistema de login, conexão de banco de dados e operações CRUD (Create, Read, Update e Delete). O Django é útil para a criação de sites utilizando banco de dados.

A implementação do nosso site incidiu maioritariamente na gestão dos produtos e na gestão de encomendas.

Definimos requisitos funcionais, mínimos e obrigatórios para o nosso site funcionar sem erros e com os devidos aspetos legais. Estes requisitos são detalhadamente descritos ao longo do trabalho.

Para além dos requisitos mínimos e obrigatórios elaboramos também quais os requisitos de melhoria que gostaríamos de efetuar. Este tema também é abordado ao longo do trabalho. Em relação aos requisitos de melhoria não os conseguimos efetuar, deixando apenas a loja online com os requisitos obrigatórios a funcionar.

Concluindo, apesar de alguma dificuldade sentida na realização deste trabalho sabemos que, no futuro, vai ser fundamental e benéfico para nós termos tido trabalhado com o Django nesta fase.

ÍNDICE

1 DJANGO:

1.1 Visão Geral

O Django é um framework web poderoso, construído em Python, que simplifica o desenvolvimento de aplicações web robustas e escaláveis. Desde o seu lançamento em 2005, o Django tem sido popular devido à sua filosofia "batteries-included", que fornece um conjunto abrangente de ferramentas para facilitar o desenvolvimento.

1.2 Arquitetura MTV (Model-Template-View) no Django

A arquitetura de software desempenha um papel fundamental na determinação da estrutura e do comportamento das aplicações informáticas. No contexto dos frameworks web, diversas arquiteturas foram propostas para simplificar e otimizar o desenvolvimento de aplicações. No caso do Django, um framework escrito em Python, adota-se uma abordagem denominada MTV (Model-Template-View).

1.2.1 Introdução à Arquitetura MTV

Enquanto muitos frameworks web seguem a arquitetura MVC (Model-View-Controller), o Django opta pela MTV. Ambas as arquiteturas partilham objetivos similares: desacoplar a lógica de negócio da interface do utilizador e do controlo de fluxo da aplicação. No entanto, os componentes são definidos e interagem de maneira ligeiramente diferente.

1.2.1.1 Componentes da Arquitetura MTV

1.2.1.2 Model (Modelo)

O componente "Model" diz respeito à representação dos dados e à lógica de negócios associada. Em termos concretos, em Django:

- Define a estrutura da base de dados, incluindo as tabelas e as relações entre elas.
- Contém funções que permitem interagir com esses dados, como consulta, inserção, atualização e remoção.
- Garante a integridade e a validade dos dados através de validações.

Conceito de Modelos no Django

Em Django, um modelo é a representação única e definitiva da informação sobre os dados. Define os campos e o comportamento dos dados que quer armazenar. Cada modelo traduz-se numa tabela na base de dados.

Vantagens dos Modelos

- Abstração: Permitem uma representação de alto nível de estruturas de base de dados, evitando a necessidade de se preocupar com os detalhes específicos do sistema de gestão de base de dados.
- DRY (Don't Repeat Yourself): Uma vez definido o modelo, o Django gere muitas tarefas relacionadas, evitando repetição de código.
- Integridade dos Dados: Ao definir relações e restrições nos modelos, garante-se que os dados armazenados sejam consistentes e íntegros.

Definição dos Modelos

Para definir um modelo, cria-se uma classe no ficheiro models.py da aplicação, cria apartir da classe mãe models.Model fornecida pelo Django.

Exemplo de definição de um modelo:

```
from django.db import models

class Livro(models.Model):
    titulo = models.CharField(max_length=200)
    autor = models.CharField(max_length=100)
    publicado_em = models.DateField()
```

Neste exemplo, foi definido um modelo para um livro com campos para o título, autor e data de publicação.

1.2.1.3 Migração de Modelos

Após definir um modelo, é necessário criar migrações e aplicá-las para que as tabelas correspondentes sejam criadas na base de dados.

Criar Migrações

Para criar migrações com base nas alterações dos modelos, utiliza-se o seguinte comando:

```
python manage.py makemigrations
```

Aplicar Migrações

Depois de criar as migrações, para aplicá-las e efetuar as alterações na base de dados, executa-se:

```
python manage.py migrate
```

Ao aplicar as migrações, o Django cria ou altera as tabelas na base de dados conforme definido nos modelos.

1.2.1.4 Vistas (Views) no Django

As views são componentes fundamentais no Django, pois atuam como intermediárias entre os modelos (dados) e os templates (apresentação). Elas têm a responsabilidade de processar os HTTP Request, interagir com os modelos, e eventualmente, renderizar um template com informações para ser apresentado ao utilizador.

Conceito de Views no Django

Uma view, no contexto do Django, pode ser uma função Python ou uma classe, que recebe HTTTP Request como argumento e devolve um HTTP Response. Essa resposta pode ser a renderização de um template, um redireccionamento para outro URL, ou até mesmo a devolução de dados em formatos como JSON.

Views Baseadas em Funções

As views baseadas em funções são simples e diretas. Aqui está um exemplo básico:

```
from django.http import HttpResponse

def minha_vista(request):
    return HttpResponse('Hello world!')
```

Neste exemplo, a view simplesmente devolve um HTTP Response com o texto "Hello World!".

Views Baseadas em Classes

O Django também suporta views baseadas em classes, que oferecem uma abordagem mais modular e reutilizável. Com elas, pode-se criar views que herdam comportamentos específicos, permitindo uma melhor organização e reutilização de código.

Exemplo de uma view baseada em classe:

```
from django.views import View
from django.http import HttpResponse

class MinhaVista(View):
    def get(self, request):
        return HttpResponse('Hello World, através de uma vista baseada numa
classe!')
```

Integrar views com URLs

Para que uma view seja acessível através de um browser ou qualquer cliente HTTP, é necessário mapeá-la para um URL específico. Isso é feito no ficheiro urls.py da aplicação ou do projeto.

Exemplo de ligação de um URL a uma view:

```
from django.urls import path
from . import views

urlpatterns = [
    path('hello/', views.output_test, name='hello_world'),
]
```

Neste caso, a view "output_test" está ligada para o URL /hello/.

1.2.1.5 Templates no Django

Conceito de Templates no Django

Os templates desempenham um papel crucial na arquitetura do Django, possibilitando a criação de interfaces de utilizador de forma dinâmica. O sistema de templates do Django foi concebido para separar a lógica da aplicação da apresentação, proporcionando flexibilidade e uma manutenção mais eficaz.

Um template no Django é, essencialmente, um ficheiro HTML que pode conter variáveis, bem como tags específicas que permitem inserir lógica diretamente na apresentação. Estas variáveis são substituídas pelos seus valores quando o template é renderizado.

Criar Templates

Para começar a trabalhar com templates, é boa prática criar uma pasta denominada templates na respetiva aplicação. Dentro desta pasta, cada template é geralmente um ficheiro .html.

Exemplo de um template básico:

```
<!DOCTYPE html>
<html>
<head>
    <title>{{ titulo_pagina }}</title>
</head>
<body>
    <h1>{{ cabecalho }}</h1>
    <p>{{ conteudo }}</p>
</body>
</html>
```

Neste exemplo, '{{ titulo_pagina }}', '{{ cabecalho }}' e '{{ conteudo }}' são variáveis que serão substituídas pelos seus valores correspondentes na renderização do template.

Linguagem do Template de Django

A linguagem do template de Django oferece uma variedade de elementos que permitem adicionar lógica diretamente ao ficheiro do template:

- Variáveis: Representadas por {{ variavel }}, são substituídas pelo seu valor quando o template é renderizado.

- Tags: Permite adicionar lógica, como ciclos ou condições. Exemplo: {% for item in lista %} ... {% endfor %}
- Filtros: Modificam a forma como as variáveis são apresentadas. Exemplo: {{ nome|lower }} converte a variável nome para minúsculas.

Integrar Templates com Vistas

Para renderizar um template, é comum usar a função render() em uma vista. Esta função toma o pedido HTTP, o nome do template e um dicionário de variáveis a serem passadas para o template como argumentos.

```
from django.shortcuts import render

def minha_vista(request):
    context = {'titulo_pagina': 'Título', 'cabecalho': 'Bem-vindo!',
'conteudo': 'Este é o conteúdo da página.'}
    return render(request, 'nome_do_template.html', context)
```

1.3 ORM

O Object-Relational Mapping (ORM) é um componente fundamental do framework Django, proporcionando uma ponte entre os modelos de dados definidos em Python e as bases de dados relacionais.

1.3.1 Conceito de ORM

ORM refere-se à técnica de mapear objetos de software a entradas de bases de dados relacionais. Em termos simples, permite que os programadores interajam com a base de dados orientadas a objetos, sem a necessidade de escrever instruções SQL explícitas.

1.3.2 Vantagens do Uso do ORM

Abstração da Base de Dados: Oferece uma camada de abstração, permitindo que os programadores não tenham que lidar diretamente com detalhes específicos da base de dados.
Portabilidade: O código torna-se menos dependente de um Sistema de Gestão de Bases de Dados (SGBD) específico, facilitando a troca entre diferentes SGBDs.
Produtividade: Permite que os programadores trabalhem com a linguagem de programação Python nativa, sem a necessidade de se preocuparem com instruções SQL.

Segurança: Reduz o risco de vulnerabilidades como injeções SQL, uma vez que as consultas são construídas de forma segura pelo ORM.

1.4 ORM no Django

O ORM do Django é poderoso e oferece uma série de características que otimizam o desenvolvimento e manutenção de aplicações.

1.4.1 Definição de Modelos

Em Django, cada modelo, que é uma classe Python, corresponde a uma tabela na base de dados. Os atributos da classe representam as colunas da tabela.

```
from django.db import models

class Autor(models.Model):
    nome = models.CharField(max_length=100)
    data_nascimento = models.DateField()
```

Neste exemplo, um modelo Autor foi definido, que será mapeado para uma tabela na base de dados com as colunas nome e data_nascimento.

1.4.2 Consultas com ORM

O ORM do Django permite realizar consultas à base de dados de forma intuitiva.

Exemplo de uma consulta para buscar todos os autores:

```
autores = Autor.objects.all()
```

Para filtrar autores com um nome específico:

```
autores = Autor.objects.filter(nome="João")
```

1.4.3 Relações entre Modelos

O ORM do Django suporta relações entre modelos, como *ForeignKey* (relação muitos para um), *OneToOneField* (relação um para um) e *ManyToManyField* (relação muitos para muitos), permitindo a modelagem de relações complexas entre as tabelas de forma simples e eficaz.

1.5 Conceito de Aplicação no Django

Em termos do Django, uma "aplicação" é um módulo de Python que é reconhecido pelo Django como tendo relevância administrativa. Uma aplicação pode conter models, views, controladores, ficheiros estáticos, e muito mais. De ressalvar que um projeto poderá ter várias aplicações.

1.5.1 Vantagens de Utilizar Aplicações

- Modularidade: Permite separar diferentes funcionalidades do site ou da plataforma, tornando o código mais organizado.
- Reutilização: Uma aplicação desenvolvida pode ser facilmente reutilizada em diferentes projetos sem a necessidade de duplicar o código.
- Manutenção: Ao separar as funcionalidades em aplicações distintas, a manutenção torna-se mais gerável, pois as alterações em uma aplicação não afetam diretamente outras aplicações.

Criação de uma Aplicação

Para criar uma aplicação no Django, é necessário utilizar uma ferramenta de linha de comando que o Django fornece: manage.py.

O comando para iniciar uma nova aplicação é o startapp, e o seu uso é bastante direto. Após criar um projeto Django, dentro da pasta raiz do projeto, executa-se:

```
python manage.py startapp nome_da_aplicacao
```

Este comando irá gerar uma pasta com o nome fornecido (nome_da_aplicacao, neste caso). Esta pasta contém a estrutura inicial para a aplicação, incluindo diretórios para models, views, tests, entre outros.

1.6 Modelo de Ficheiros e Estrutura de Pastas no Django

Num projeto Django, a organização dos ficheiros e pastas segue um padrão específico para garantir uma estrutura lógica e coerente. Esta estrutura padrão facilita a orientação dos programadores, promovendo a manutenção eficiente e a escalabilidade do projeto.

Estrutura Básica

Após criar um projeto e uma aplicação no Django, a estrutura típica de ficheiros e pastas é a seguinte:

```
meu_projeto/
|-- meu_projeto/
|   |-- __init__.py
|   |-- asgi.py
|   |-- settings.py
|   |-- urls.py
|   |-- wsgi.py
|
|-- minha_aplicacao/
|   |-- migrations/
|   |   |-- __init__.py
|   |
|   |-- __init__.py
|   |-- admin.py
|   |-- apps.py
|   |-- models.py
|   |-- tests.py
|   |-- views.py
|
|-- manage.py
```

1.6.1 Descrição dos Componentes

- meu_projeto/ :Pasta raiz do projeto.

- minha_aplicacao/ :Pasta da aplicação criada.

- migrations/: Contém os ficheiros de migração, que são gerados automaticamente e descrevem as alterações nas tabelas da base de dados.

- settings.py: Contém as configurações do projeto.

- urls.py: Define as rotas URL do projeto.

- models.py: Define os models da aplicação, que são mapeados para tabelas na base de dados.

- views.py: Define as views, que processam os pedidos e retornam respostas.

- admin.py: Configura a interface de administração do Django para a aplicação.

- manage.py: Ferramenta de linha de comando para gerir diversas tarefas do projeto.

Estrutura com Múltiplas Aplicações

Dentro de um projeto Django, a inclusão de múltiplas aplicações resultaria numa estrutura similar à seguinte:

1.6.2 Exemplo prático com várias apps

```
meu_projeto/
|-- meu_projeto/
|    |-- __init__.py
|    |-- asgi.py
|    |-- settings.py
|    |-- urls.py
|    |-- wsgi.py
|
|-- aplicacao_1/
|    |-- migrations/
|    |    |-- __init__.py
|    |
|    |-- __init__.py
|    |-- admin.py
|    |-- apps.py
|    |-- models.py
|    |-- tests.py
|    |-- views.py
|
|-- aplicacao_2/
|    |-- migrations/
|    |    |-- __init__.py
|    |
|    |-- __init__.py
|    |-- admin.py
|    |-- apps.py
|    |-- models.py
|    |-- tests.py
|    |-- views.py
|
|-- manage.py
```

Neste exemplo, “aplicacao_1” e “aplicacao_2” representam aplicações distintas dentro do mesmo projeto Django. Ambas mantêm a sua estrutura individual, mas podem interagir entre si conforme configurado no ficheiro settings.py e outros ficheiros de configuração central do projeto.

Esta abordagem modular permite que os programadores expandam e escalem o projeto de forma ordenada, adicionando ou removendo aplicações conforme as necessidades evoluem.

2 PROJETO CLICKFARMA – "VENDEMOS PRODUTOS PARA A SUA SAÚDE E BEM-ESTAR"`

No decorrer deste projeto de uma farmácia online, definimos alguns requisitos mínimos e obrigatórios. Focamo-nos principalmente na gestão dos produtos.

2.1 Requisitos

Como requisitos mínimos e obrigatórios definimos:

- Efetuar login/logout dos utilizadores e do staff dando acesso à área de membro;
- Página do utilizador;
- Ver produtos na homepage;
- Garantir que o preço de venda ao público (PVP) dos produtos se encontra visível bem como o respetivo botão para adicionar ao carrinho;
- O staff visualizar os produtos que se encontram em stock e alterar o estado da encomenda;
- Na área pessoal do utilizador este consegue ver os seus produtos selecionados no carrinho e após o pagamento visualizar o estado da sua encomenda;
- Ver a página do checkout;
- Colocar 5 categorias e alguns produtos nelas.

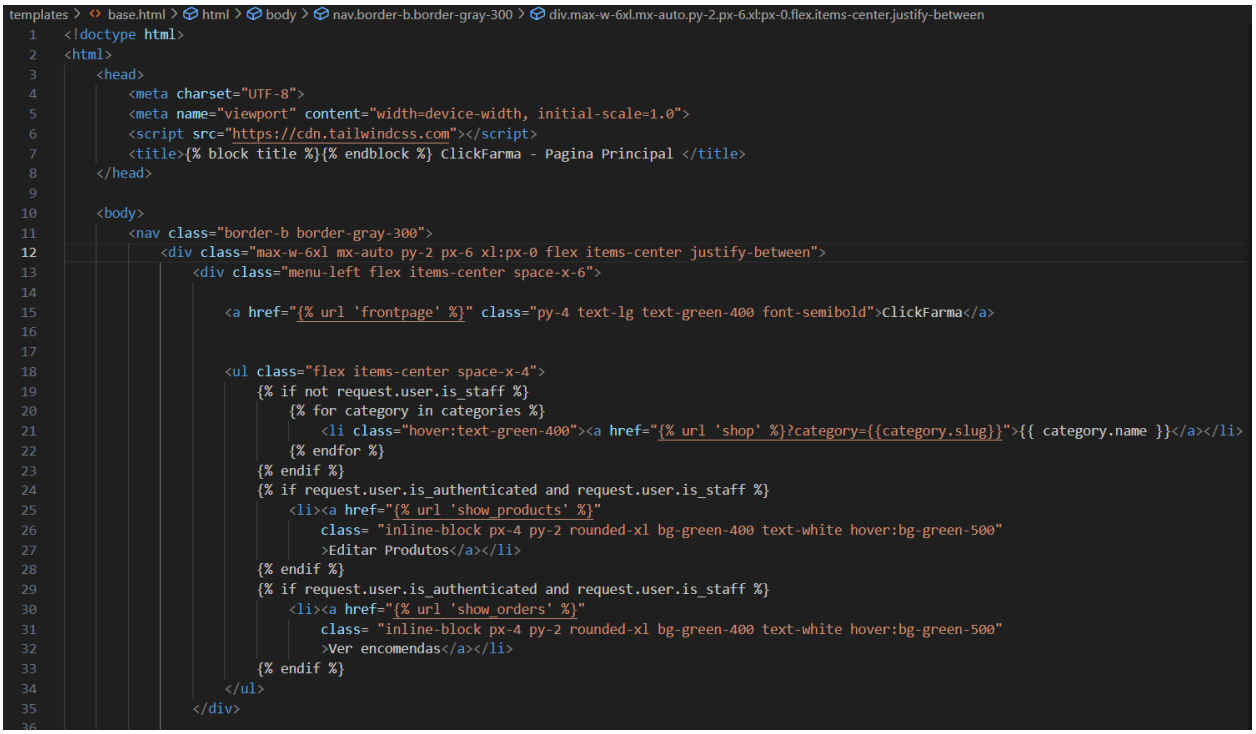
Os requisitos mínimos e obrigatórios foram todos implementados.

Como requisitos de melhoria, definimos:
- Selecionar a morada pretendida;
- Mudar a palavra-passe;
- Colocar mais categorias de produtos;
- Colocar o logotipo da farmácia no cabeçalho;
- Fazer promoções usando o discount;
- Métodos de pagamento (Paypal e MBWAY…).

Os requisitos de melhoria não conseguimos implementar durante a elaboração deste projeto.

2.2 Front-end Web Development

Um programador front end é a pessoa responsável por toda a aparência do site. Podemos dizer que o front-end diz respeito à parte do cliente ou do utlizador. As linguagens básicas para o Front-end são HTML (Hyper Text Markup Language), CSS e JavaScript.
No nosso projeto utilizamos a linguagem HTML com o CSS integrado através de Tailwind. De seguida, apresentamos o nosso código que se encontra na base.html. Este código está associado a todos os outros HTML's existentes na nossa aplicação.



O CSS integrado através da framework Tailwind é implementada através de um link colocado no head do documento HTML (como se pode ver na figura):

```
<script src="https://cdn.tailwindcss.com"></script>
```

No código HTML de seguida representado, podemos ver duas partes bem distintas. O início do código remete para "href" e uma url da frontpage, isto quer dizer que quando

visualizamos no código HTML a palavra “href” é sinónimo de uma hiperligação. Neste caso, ao clicarmos na palavra ClickFarma é direcionado para a frontpage.

Na segunda parte do código, temos o estilo das cores que são apresentadas na palava ClickFarma, como por exemplo, a cor verde e a sua intensidade.

```
<a href="{% url 'frontpage' %}" class="py-4 text-lg text-green-400 font-semibold">ClickFarma</a>
```

Outro exemplo bem interessante que aplicamos durante o nosso projeto foi o facto de que tivemos que adicionar algumas imagens para colocar a nossa loja online legal. Estamos a falar de autorização de venda de medicamentos online (pelo INFARMED e pela DGAV) e o livro de reclamações. Estas imagens remetem para uma hiperligação para o site.
De seguida, apresentamos o código HTML de forma a explicar melhor como se procedeu:

```
<a href="https://www.livroreclamacoes.pt/Inicio/"><img src="{{MEDIA_URL}}livro_rec.png" width="170"/></a>
```

Neste caso, como podemos visualizar, temos a imagem em PNG e o site da hiperligação. Neste código é obrigatório estar entre chavetas a localização da imagem no Django. Neste caso, encontra-se na pasta media.

2.3 Back-End Wb Developemennt

O back-end por sua vez, é tudo o que fica do lado do servidor. É uma combinação de servidores e bases de dados. Os utilizadores controlam como os utilizadores acedem aos arquivos. As bases de dados são coleções de dados organizadas e estruturadas.

Por exemplo, quando entramos num site e colocamos o e-mail ou o nome de utilizador e a palavra-passe, esta informação vai toda para a parte do servidor que "valida" estes dados. Se correr tudo bem, o servidor verifica os dados com a base de dados de forma a garantir que o nome do utilizador e a palavra-passe existem. Se isso acontecer, a base de dados fará login e entra na área pessoal do utilizador.

2.3.1 O que é o módulo Views em Django e por que é importante na ótica do utilizador?

Este módulo é responsável por receber solicitações dos utilizadores, processá-las e retornar respostas. As visualizações podem ser funções ou classes que definem a lógica da aplicação, incluindo o processamento de dados, a renderização de modelos e a criação de respostas para o cliente. Nesse sentido, as views são partes importantes para a criação de páginas Web dinâmicas no Django.

De seguida, apresentamos as views da app Info.

```python
from django.shortcuts import render

# Create your views here
def about(request):
    return render(request, 'info/about.html')

def shipping(request):
    return render(request, 'info/shipping.html')

def payment(request):
    return render(request, 'info/payment.html')

def conbact(request):
    return render(request, 'info/conbact.html')
```

Vamos analisar o que cada função faz:

about(request): a função diz respeito à página "about" do site.

Quando um utilizador acede à página "about", o Django chama esta função.

A função usa o método `render` para renderizar o modelo associado à página "about.html" e retornar a resposta.

shipping(request): Esta função diz respeito à página "shipping" do site.

Semelhante à função anterior, ela renderiza o modelo associado à página "shipping.html" e retorna a resposta.

payment(request): Esta função diz respeito à página "payment" do site.

Renderiza o modelo associado à página "payment.html" e retorna a resposta.

conbact(request): Esta função diz respeito à página de contacto do site.

Assim como nas outras funções, ela renderiza o modelo associado à página "conbact.html" e retorna a resposta.

Cada página do site é tratada por uma função de visualização separada. As funções de visualização recebem uma solicitação de “request” como parâmetro, processam as informações necessárias e retornam uma resposta, geralmente “renderizando” um modelo HTML correspondente à página acedida.

Para alem de renderizar páginas apartir de um ficheiro html, também é possivel adicionar contexto para ser renderizado como uma variavel no ficheiro html ou então a utilização de decoradores para limitar funcionalidade.

Para demonstrar isto temos de seguida a view que renderiza a pagina onde o staff pode gerir os produtos:

```python
@staff_member_required
def show_products(request):
    products = Product.objects.all()
    categories = Category.objects.all()
    category = request.GET.get('category', '')
    if category:
        products = products.filter(category__slug=category)
    query = request.GET.get('query', '')
    if query:
        products = products.filter(name__icontains=query)

    active_category = request.GET.get('category', '')
    context = {
        'products': products,
        'categories': categories,
        'active_category': active_category
    }

    return render(request, 'Products/show_products.html', context)
```

Esta função pode ser dividida em várias partes:

Primeiro, a função está decorada com @staff_member_required, o que significa que só pode ser acedida por utilizadores que sejam Staff e que tenham feito login.

Depois temos os seguintes elementos:

- Buscar todas as instâncias (ou objectos) da classe Product e Category, usando o método all(), estes objectos são guardados nas variaveis products e categories respetivamente;

- Verifica se category está presente no pedido GET. Se estiver, filtra o queryset products para incluir apenas produtos cuja categoria correspondente tenha um slug que combine com o parâmetro category. Isso é feito usando o método filter(), que retorna um novo queryset incluindo apenas as instâncias que correspondem aos parâmetros de pesquisa fornecidos;

- Também verifica se um parâmetro query está presente no pedido GET. Se estiver, filtra o queryset products para incluir apenas produtos cujo nome contenha o parâmetro query, independentemente de maiúsculas ou minúsculas. Isso é feito usando o método filter() com a pesquisa icontains, que realiza um teste de contenção insensível a maiúsculas/minúsculas;

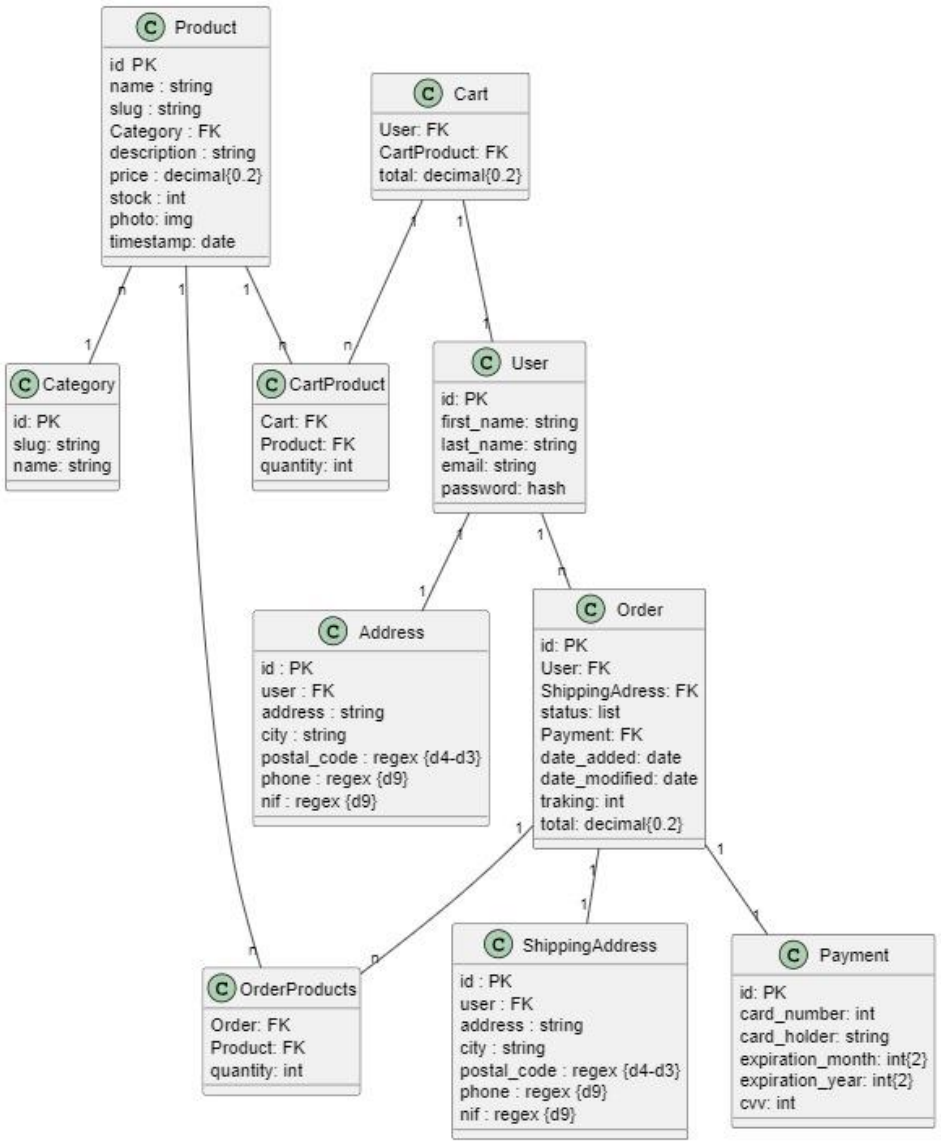
- Obtém novamente o parâmetro category do pedido GET e atribui-o a active_category. Isto é utilizado para acompanhar qual categoria está a ser visualizada no momento;

-Finalmente, cria um dicionário de contexto contendo os products, categories, e active_category, e retorna este contexto na função render. Este contexto é utilizado para renderizar um modelo com os produtos, categorias e a categoria ativa.

2.4 Models (Classes)

Em Django, "models" são classes Python que representam tabelas no base de dados. Cada atributo da classe representa uma coluna da tabela e os tipos de dados desses atributos determinam os tipos de colunas no base de dados.

Na ilustração seguinte temos a representação gráfica das classes usadas no nosso projecto:



A seguir, apresentamos o código do *models.py* (código que armazena a programação das Classes) referente à app Orders.

```python
class ShippingAddress(models.Model):
    first_name = models.CharField(max_length=100, null=False)
    last_name = models.CharField(max_length=100, null=False)
    address = models.CharField(max_length=100, null=False)
    postal_code = models.CharField(validators=[RegexValidator(r'^\d{4}-\d{3}$')], max_length=8, null=False)
    city = models.CharField(max_length=100, null=False)
    nif = models.CharField(validators=[RegexValidator(r'^\d{9}$')], max_length=9, null=True)
    phone = models.CharField(validators=[RegexValidator(r'^\d{9}$')], max_length=9, null=False)



    def __str__(self):
        return self.address
    class Meta:
        db_table = 'ShippingAdress'
        verbose_name = 'ShippingAdress'
        verbose_name_plural = 'ShippingAdresses'
        ordering = ['id']

class Payment(models.Model):
    id = models.AutoField(primary_key=True)
    card_number = models.CharField(validators=[RegexValidator(r'^\d{16}$')], max_length=16, null=False)
    card_holder = models.CharField(max_length=100, null=False)
    expiration_month = models.CharField(validators=[RegexValidator(r'^\d{2}$')], max_length=2, null=False)
    expiration_year = models.CharField(validators=[RegexValidator(r'^\d{2}$')], max_length=2, null=False)
    cvv = models.CharField(validators=[RegexValidator(r'^\d{3}$')], max_length=3, null=False)
    class Meta:
        verbose_name = 'Payment'
        verbose_name_plural = 'Payments'
    def clean(self):
        super().clean()
        expiry_date = datetime.strptime(f'{self.expiration_year}-{self.expiration_month}-01', '%y-%m-%d')
        last_day_of_expiry_month = expiry_date.replace(month=expiry_date.month%12+1, day=1) - timedelta(days=1)
        if last_day_of_expiry_month.date() < datetime.now().date():
            raise ValidationError("The credit card has expired.")
    def __str__(self):
        return str(self.id)
```

A classe Shipping Adress armazena informações sobre a morada de envio. Os campos que a constituem são nome, apelido, morada, código postal, cidade, NIF e contacto telefónico contendo uma validação para o formato do código postal (XXXX-XXX), NIF (9) e contacto telefónico (9).

A classe Payment é responsável por armazenar os dados de pagamento como o número de cartão multibanco, o titular do cartão, o mês/ano validade e o CVV. Inclui validação para o formato desses campos e implementa “clean” para validar a data expiração do cartão.

Ambas têm um método “__str__” para representação de uma string.

2.5 O que são templates em Django e um exemplo prático no projeto ClickFarma

Os templates em Django são arquivos que definem a aparência e o layout das páginas Web. É um arquivo de texto que pode ser transformado em outro arquivo (um arquivo HTML, um CSS, um CSV, etc…).

Exemplo de template da frontpage:

```
{% extends 'base.html' %}

{% block content %}
<header class="px-6 py-10 lg:py-20 bg-green-400">
    <div class="max-w-3xl mx-auto text-center">
        <p class="mb-2 text-3xl lg:text-5xl text-white">Bem-vindo(a) à ClickFarma!</p>

        <p class="mb-10 text-white">Vendemos produtos para a sua saude e bem estar!</p>

        <a href="{% url 'shop' %}" class="inline-block px-8 py-4 rounded-xl bg-white text-green-400 hover:bg-gray-200">Iniciar Compras</a>
    </div>
</header>

<div class="max-w-6xl mx-auto py-2 px-6 xl:px-0">
    <div class="products flex items-center flex-wrap">
        {% for product in products %}
            {%include 'Products/partials/grid.html'%}
        {% endfor %}

    </div>
</div>
{% endblock %}
```

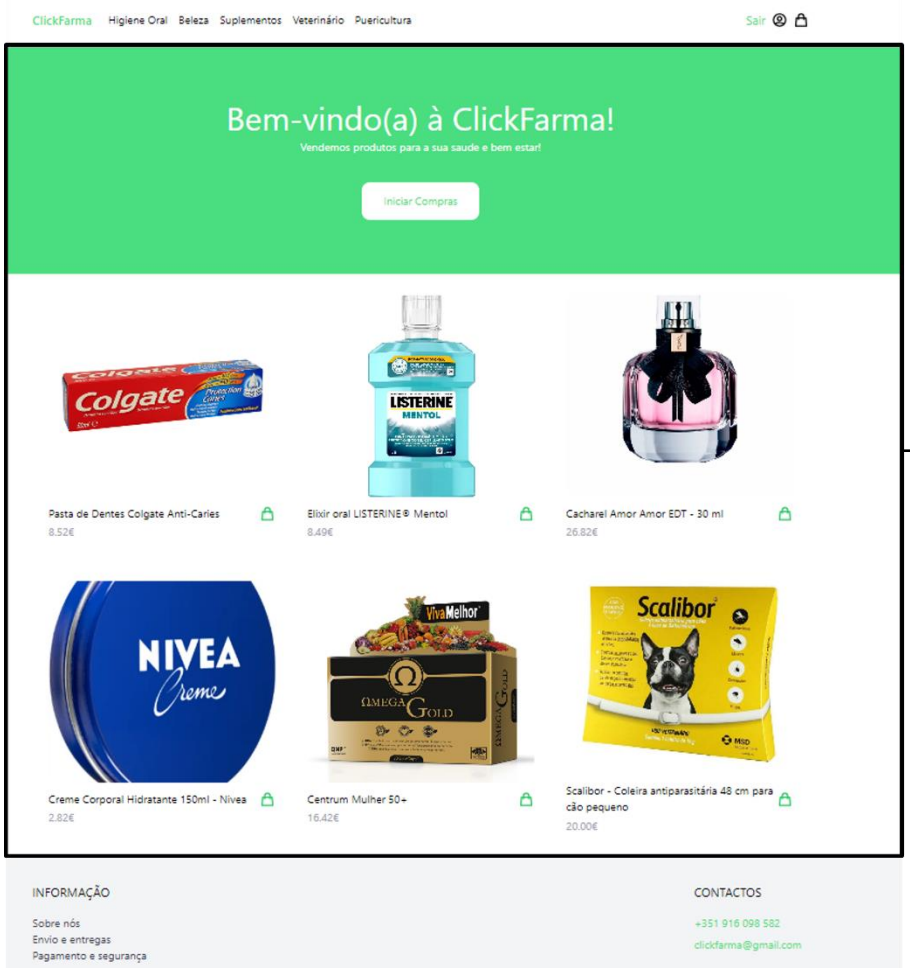
Um template contém:

- Variáveis (products);
- Tags que controlam a lógica do template;
- Filtros que adicionam funcionalidades ao template.

O que cada parte do código significa?

- `{% extends 'base.html' %}`: Herança da estrutura definida no template base.
- `{% block content %} ... {% endblock %}`: O bloco "content" contém o conteúdo específico desta template.
- O conteúdo dentro do bloco "content": `<header>`: Define um cabeçalho com algumas classes de estilo. Exibe uma mensagem de boas-vindas, uma descrição e um botão "Iniciar Compras" que redireciona para a página de compras quando clicado.
- `<div class="max-w-6xl mx-auto py-2 px-6 xl:px-0"> ... </div>`: Contém uma seção para exibir produtos. Usa um loop `{% for product in products %}` para iterar sobre uma lista de produtos e incluir um template parcial chamado 'Products/partials/grid.html' para cada produto.

- `{% endblock %}`: Marca o final do bloco "content".



Template frontpage

SUPORTE DOCUMENTAL

Vincent, W. S. (2022). Django for Beginners. Packt Publishing.

https://docs.djangoproject.com/en/4.2/ , visitado no mês de dezembro de 2023

https://www.w3schools.com/django/ , visitado no mês de dezembro de 2023

https://github.com/Jofralso/UPskill---Intensive-Python/tree/main/DJANGO, visitado no mês de dezembro de 2023

Para população de base de dados:

https://www.colgate.com/pt-pt/products/cavity-prevention-toothpaste

https://www.listerine.pt/produtos/listerine-mentol#benef%C3%ADcios

https://wells.pt/cacharel-amor-amor-edt-287911.html

https://wells.pt/mala-da-maternidade-azul-5409379.html?gad_source=1&gclid=Cj0KCQiAwP6sBhDAARIsAPfK_wadfhCvSrTpmBaTYLOfSF3fI2p4ZtdN3b5YnU0AbkLmVqXoHYJUEeMaAgGUEALw_wcB

https://wells.pt/centrum-mulher-50-5293730.html?gad_source=1&gclid=Cj0KCQiAwP6sBhDAARIsAPfK_wYIERU4T3Et9DcSB5ahjVfhM9-4G6lG1Z1QAGlZMVXmfr8DAKCPEP0aAt6_EALw_wcB

https://www.continente.pt/produto/creme-corporal-hidratante-nivea-2053684.html?gad_source=1&gclid=Cj0KCQiAwP6sBhDAARIsAPfK_wYxxGZGaiLO_lcdVn3-qFkTQD_BjKgaCIlQD8N0w-sGSvU5wTsj_5UaAlbdEALw_wcB

CONCLUSÃO

No decorrer deste projeto de implementação de uma Farmácia Online, que denominamos por ClickFarma, usamos a framework Django, exploramos diversas áreas fundamentais do desenvolvimento web, desde o front-end até o back-end. Com a ajuda da framework Django conseguimos colocar a nossa loja funcional.

Ao abordarmos os requisitos mínimos e obrigatórios, conseguimos implementar funcionalidades fundamentais, como a gestão de produtos, login/logout de utilizadores e staff, visualização de produtos na homepage, e a gestão de encomendas. Todos estes requisitos são essenciais para que qualquer loja online seja minimamente funcional.

Mesmo com todos os desafios que enfrentamos durante a elaboração, principalmente na implementação dos requisitos de melhoria, é importante reconhecer que estes foram de grande importância para o desenvolvimento das nossas habilidades em Django.

No contexto do back-end, a exploração do ORM do Django trouxe à tona a facilidade de interação com bases de dados relacionais, proporcionando abstração e segurança. A criação e utilização de aplicações dentro do projeto demonstrou modularidade e reutilização de código, facilitando a manutenção e escalabilidade.

O desenvolvimento do front-end foi conduzido com a utilização de HTML e CSS, integrando a framework Tailwind para estilização. A apresentação de templates exemplificou a separação eficaz de lógica e apresentação, fundamental para a criação de interfaces dinâmicas.