

Desenvolvimento de Webapps em Django

- **Criação de ambiente de desenvolvimento**

conda create -n aula_django python=3.9

conda activate aula_django

- **Instalação do Framework Django**

pip install Django

- **Criação de slugs automáticos a partir de um campo do Model**

pip install django-autoslug

- **Criação de campos de timestamp:**

Biblioteca que auxilia na criação de Models que utilizam data e hora de criação e modificação.

pip install django-model-utils

- **Biblioteca que auxilia no shell mais iterativo**

pip install ipython

- **Biblioteca Pillow**

Biblioteca necessária em projetos que irão trabalhar com imagens

pip install pillow

Outras funcionalidades para aumentar a produtividade:

pip install autopep8

pip install pylint

- **Instalação de requisitos de projeto:**

pip install -r requirements.txt

flag -r (--requirements) : requisitos a serem instalados

ctrl+shift+p

select linter -> pylint

- **Criação de projeto:**

django-admin startproject store (ou django-admin startproject store . #para não criar a segunda pasta)

- **Criação de apps:**

Primeiro app irá tratar da apresentação geral dos produtos

django-admin startapp pages (ou python manage.py startapp pages)

Segundo app irá tratar dos registros dos usuários

django-admin startapp users (ou python manage.py startapp users)

Terceiro app irá tratar dos registros dos produtos

django-admin startapp products (ou python manage.py startapp products)

Vamos criar mais apps posteriormente...

Antes de iniciar a programação do projeto e criar os models, vamos entender um pouco sobre o mapeamento de urls e templates no Django:

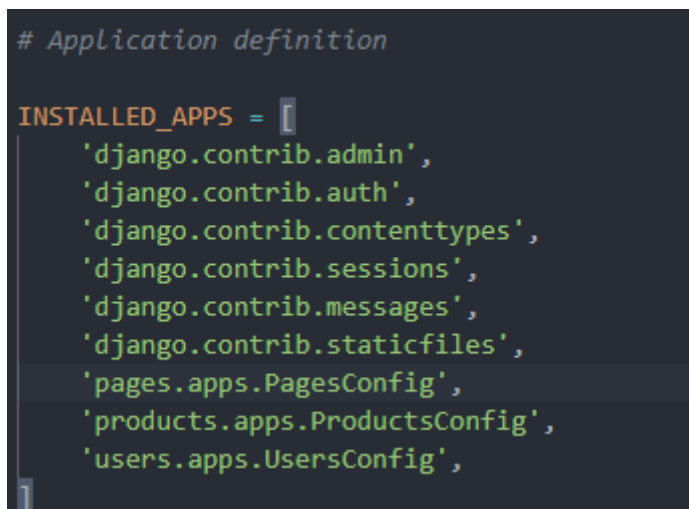
Primeiro passo é incluir nossos apps no projeto

Em settings.py -> INSTALLED_APPS, adicione:

'pages.apps.PagesConfig',

'products.apps.ProductsConfig',

'users.apps.UsersConfig',



```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pages.apps.PagesConfig',
    'products.apps.ProductsConfig',
    'users.apps.UsersConfig',
]
```

Agora vamos criar os arquivos urls.py nas pastas de cada app, assim teremos:

Projeto: store -> urls.py

App: pages -> urls.py

App: users -> urls.py

App: products -> urls.py

Na pasta do projeto store, em urls.py:

Faça a importação do include e adicione as rotas para as urls de cada app:

path('pages/', include('pages.urls')),

path('users/', include('users.urls')),

path('products/', include('products.urls')),

```

from django.contrib import admin
from django.urls import path, include

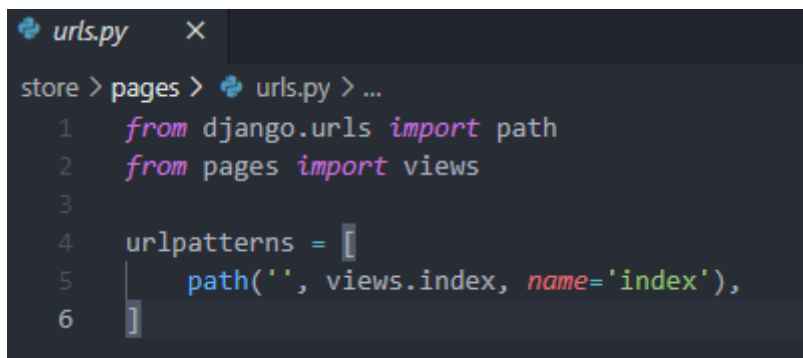
# fiz a importação para apresentar a duplicação de chamada a mesma página
from pages import views

urlpatterns = [
    path('admin/', admin.site.urls),

    # Essa linha está chamando um método em views dentro do app pages,
    # duplicando a chamada que está em pages.urls.py
    path('', views.index, name='index'),
    path('pages/', include('pages.urls')),
    path('users/', include('users.urls')),
    path('products/', include('products.urls')),
]

```

Dentro de urls.py de cada app crie um padrão de chamadas de urls



```

store > pages > urls.py > ...
1  from django.urls import path
2  from pages import views
3
4  urlpatterns = [
5      path('', views.index, name='index'),
6  ]

```

Procure ir testando seu projeto (subindo o server) ao longo das modificações, isso facilita na hora de encontrar problemas de codificação.

No app pages, em views.py, inclua o seguinte código:

```

from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def index(request):

    return HttpResponse('Olá Mundo!')

```

```
views.py X
store > pages > views.py > ...
1  from django.shortcuts import render
2  from django.http import HttpResponse
3
4  # Create your views here.
5  def index(request):
6      return HttpResponse('Olá Mundo!')
```

Agora acesse:

<http://127.0.0.1:8000/>

<http://127.0.0.1:8000/pages/>

Você verá o mesmo código, pois tanto o urls.py (projeto store) quanto o urls.py (app pages) estão chamando o mesmo método index de view.py (app pages)

- **Criação e configuração de pastas para templates:**

No projeto store, em settings.py, vamos incluir caminhos para as pastas de templates e arquivos estáticos para funcionar de forma modular.

Será necessário importar a biblioteca os e utilizar métodos da mesma:

```
import os
```

```
TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
```

```
from pathlib import Path
import os

# Build paths inside the project like this: BASE_DIR / 'subdir'.
# Não siga a informação acima, pois em um server baseado em sistema Windows não irá funcionar.

BASE_DIR = Path(__file__).resolve().parent.parent
# print(__file__)
# print(BASE_DIR)

TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
# print(TEMPLATE_DIR)
```

```

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR,],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

```

- Criação e configuração de pastas para static files:

```

from pathlib import Path
import os

# Build paths inside the project like this: BASE_DIR / 'subdir'.
# Não siga a informação acima, pois em um server baseado em docker

BASE_DIR = Path(__file__).resolve().parent.parent
# print(__file__)
# print(BASE_DIR)

TEMPLATE_DIR = os.path.join(BASE_DIR, 'templates')
# print(TEMPLATE_DIR)

STATIC_DIR = os.path.join(BASE_DIR, 'static')
# print(STATIC_DIR)

STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
print(STATIC_ROOT)

```

```

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.2/howto/static-files/

STATIC_URL = '/static/'
STATICFILES_DIRS = [
    STATIC_DIR,
]

```

- **Reconfiguração das Rotas**

Agora que as configurações para utilização de páginas html e arquivos estáticos estão concluídas, vamos criar nossas primeiras páginas html, rotarizar corretamente e criar métodos em views para renderizar as páginas para os usuários.

Primeiramente vamos consertar a rota no projeto store -> urls.py

```
from django.contrib import admin
from django.urls import path, include

# fiz a importação para apresentar a duplicação de chamada a mesma página
# from pages import views

urlpatterns = [
    path('admin/', admin.site.urls),

    # Essa linha está chamando um método em views dentro do app pages,
    # duplicando a chamada que está em pages.urls.py
    # path('', views.index, name='index'),
    path('', include('pages.urls')),
    path('users/', include('users.urls')),
    path('products/', include('products.urls')),
]
```

Agora vamos criar rotas nos arquivos urls.py dos apps:

```
store > pages > urls.py > ...
1  from django.urls import path
2  from pages import views
3
4  app_name = 'pages'
5
6  urlpatterns = [
7      path('', views.index, name='index'),
8      path('about/', views.about, name='about'),
9  ]
```

```
store > products > urls.py > ...
1  from django.urls import path
2  from products import views
3
4  urlpatterns = [
5      path('', views.index, name='index'),
6  ]
```

```
store > users > urls.py > ...
1  from django.urls import path
2  from users import views
3
4  urlpatterns = [
5      path('', views.index, name='index'),
6  ]
```

- **Criação dos templates HTML**

Nesse momento crie os arquivos html:

- **Utilização de tags de extensão do Python/Django:**

{%block title%} {%endblock title%} -> Substitui o título da página

{%block content%} {%endblock content%} -> insere o bloco de conteúdo de outras páginas

{% url 'app:name' %} -> app: nome do app que contém o padrão URL; name: nome do padrão da URL

{% static 'endereço' %} -> endereço: subpasta onde contém o arquivo que deseja inserir

```
store > templates > base.html > html > body
14  <title>{% block title %} Projeto E-commerce Django {% endblock title %}</title>
15  </head>
16
17  <body>
18      <nav class="navbar navbar-expand-sm py-3 border-bottom navbar-light">
19          <div class="container">
20              <a href="{% url 'pages:index' %}" class="navbar-brand">
21                  <span class="font-weight-bold h3">Loja Django UCAM</span>
22              </a>
23
24              <button class="navbar-toggler" data-toggle="collapse" data-target="#navbarCollapse">
25                  <span class="navbar-toggler-icon"> </span>
26              </button>
27              <div class="collapse navbar-collapse" id="navbarCollapse">
28                  <ul class="navbar-nav ml-auto">
29                      <li class="nav-item">
30                          <a href="{% url 'pages:about' %}" class="nav-link lead mr-4 font-weight-bold">Sobre Nós</a>
31                      </li>
32                  </ul>
33              </div>
34          </div>
35      </nav>
36      {% block content %}
37      {% endblock content %}
38
39      <!-- Optional JavaScript -->
40      <!-- jQuery first, then Popper.js, then Bootstrap JS -->
41      <script src="{% static 'js/jquery-3.2.1.slim.min.js' %}"></script>
42      <script src="{% static 'js/popper.min.js' %}"></script>
43      <script src="{% static 'js/bootstrap.min.js' %}"></script>
44  </body>
45
46  </html>
```


{{ variável }} -> apresentação de informação da variável

```
store > templates > <> index.html > div.container.my-3
1  {% extends 'base.html' %}
2
3  {% block content %}
4  <div class="container my-3">
5      <h1> {{intro}} </h1>
6      <p> {{info}} </p>
7      <p> {{intro2}} </p>
8  </div>
9
10 {% endblock content %}
```

```
store > templates > <> about.html > ...
1  {% extends 'base.html' %}
2
3  {% load static %}
4
5  {% block title %}Sobre Nós{% endblock title %}
6
7  {% block content %}
8
9  <div class="container my-3">
10     <h1 class="font-weight-bold">{{intro}}</h1>
11     <p>{{info}}</p>
12     
13 </div>
14
15 {% endblock content %}
```

- **Modificação do views.py para renderização dos templates**

Finalizamos com o views.py, do app pages:

```

store > pages > views.py > ...
1  from django.shortcuts import render
2  # from django.http import HttpResponse
3
4  # Create your views here.
5  def index(request):
6      context = {
7          'intro': 'Bem vindos a aula de Desenvolvimento Web I',
8          'info': 'Turma 2021-1',
9          'intro2': 'Mastering Django Web Dev'
10     }
11     return render(request, 'index.html', context=context)
12
13  def about(request):
14      context = {
15          'intro': 'Sobre nós',
16          'info': 'Em construção',
17      }
18     return render(request, 'about.html', context=context)

```

Como estamos utilizando arquivos estáticos, é necessário salvá-los nas pastas referente a cada arquivo:

```

static
├── css
│   ├── bootstrap.min.css
│   └── my_stylesheet.css
├── images
│   └── under_construction.jpg
└── js
    ├── bootstrap.min.js
    ├── jquery-3.2.1.slim.min.js
    └── popper.min.js

```

- **Criação de Models:**

No arquivo /<nome_do_app>/models.py crie suas classes que irão representar os modelos equivalente a uma tabela do banco de dados

- **Criação do model Users:**

O app users, bem como o seu model, fará toda a gestão de usuários do sistema. O Django já possui uma classe padrão para criação de usuários.

Basta utilizar a AbstractUser:

```
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    pass
```

No entanto, em projetos de maior porte geralmente a AbstractUser não comporta todos os campos necessários para o cadastro de um usuário, pois é necessário a informação de outros campos, como por exemplo o CPF.

Portanto, vamos alterar a AbstractUser ampliando sua funcionalidade.

Tenha sempre em mente que estamos trabalhando sob paradigma de OO, portanto, vamos herdar as classes e sobrescrever os métodos.

```
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager

# Como estamos criando um usuário diferente do padrão do Django temos que criar um gerenciador para este usuário
class UsersManager(BaseUserManager):

    #Sobrescrevendo o método da BaseUserManager
    def create_user(self, email, name, cpf, address, phone, password=None):
        if not email:
            raise ValueError('Favor inserir seu e-mail')
        if not name:
            raise ValueError('Favor inserir seu nome')
        if not cpf:
            raise ValueError('Favor inserir seu CPF')
        if not address:
            raise ValueError('Favor inserir seu Endereço')
        if not phone:
            raise ValueError('Favor inserir seu Telefone')

        user = self.model(
            # normaliza o email colando todas as letras minusculas
            email = self.normalize_email(email),
            name = name,
            cpf = cpf,
            address = address,
            phone = phone
        )

        user.set_password(password)
        user.save(using=self._db)
        return user

    # Sobreescrita do método create_superuser para inserção de outras variáveis
    def create_superuser(self, email, name, cpf, address, phone, password):
        user = self.create_user(
            email=self.normalize_email(email),
            name = name,
            cpf = cpf,
            address = address,
            phone = phone,
```

```

        address = address,
        phone = phone,
        password = password,
    )
    user.is_admin = True
    user.is_staff = True
    user.is_superuser = True
    user.save(using=self._db)
    return user

# Criando um usuário modificado
class Users(AbstractBaseUser):
    email = models.EmailField(verbose_name = 'E-mail', max_length = 100, unique=True)
    name = models.CharField(verbose_name = 'Nome', max_length = 200,
        null = False, blank = False)
    cpf = models.CharField(verbose_name = 'CPF', max_length = 14,
        null = False, blank = False)
    address = models.CharField(verbose_name = 'Endereço',
        max_length = 200, null = False, blank = False)
    phone = models.CharField(verbose_name = 'Telefone',
        max_length = 14, null = False, blank = False)

    # Campos obrigatórios
    date_joined = models.DateTimeField(verbose_name='date joined', auto_now_add=True)
    last_login = models.DateTimeField(verbose_name='last login', auto_now=True)
    is_admin = models.BooleanField(default=False)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    is_superuser = models.BooleanField(default=False)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['name', 'cpf', 'address', 'phone']

# Adiciona o Gerenciador de conta sobreescrevendo alguns parametros do Django
objects = UserManager()

```

```

# Adiciona o Gerenciador de conta sobreescrevendo alguns parametros do Django
objects = UserManager()

def __str__(self):
    return self.cpf

# Campos obrigatórios - Admin tem nível de autorização maior, vamos deixar por questões de simplicidade
def has_perm(self, perm, obj=None):
    return self.is_admin

# Verifica se o usuário tem permissão para visualizar a aplicação
def has_module_perms(self, app_label):
    return True

# APÓS A FINALIZAÇÃO DA CRIAÇÃO DO USUÁRIO MODIFICADO, ATUALIZE O ARQUIVO SETTINGS.PY
# INCLUINDO AUTH_USER_MODEL

```

Altere o dunder methods das classes para retornar o nome ou algum outro identificador que facilite a observação através do admin do django.

Alterando o dunder str	Sem alterar o dunder str
<input type="checkbox"/> PRODUTO	<input type="checkbox"/> ITEM VENDA
<input type="checkbox"/> Produto 01	<input type="checkbox"/> ItemVenda object (1)
	1 item venda

OBSERVAÇÃO: Adicione no settings.py a autenticação para usuários customizados.

```
# Deve ser inserido após a customização do usuário do Django
AUTH_USER_MODEL = 'users.Users' #cliente.Cliente
```

Após a configuração do model Users, fazer a criação de formulário padrão que irá aparecer no admin e, também, fazer a alteração do admin.py para aparecer o model na página de admin.

Crie o arquivo: forms.py

```
from django.contrib.auth import forms
from .models import User

# Formulário padrão para modificação de usuário
class UserChangeForm(forms.UserChangeForm):
    class Meta(forms.UserChangeForm.Meta):
        # O modelo que será utilizado para criação do formulário de usuário é de acordo com nosso Users customizado
        model = User

# Formulário padrão para criação de usuário
class UserCreationForm(forms.UserCreationForm):
    class Meta(forms.UserCreationForm.Meta):
        # O modelo que será utilizado para criação do formulário de usuário é de acordo com nosso Users customizado
        model = User
```

No admin.py crie o UserAdmin e faça o registro na página de admin

```
store > users > admin.py > ...
1  from django.contrib import admin
2  from django.contrib.auth import admin as auth_admin
3
4  from .forms import UserChangeForm, UserCreationForm
5  from .models import User
6
7
8  @admin.register(User)
9  class UserAdmin(auth_admin.UserAdmin):
10     form = UserChangeForm
11     add_form = UserCreationForm
12
13     list_display = ['id', 'name', 'cpf', 'address', 'phone']
14     list_filter = ['name', 'cpf']
15
16     admin.site.register(User, UserAdmin)
```

- **Realizando as migrações para o Banco de Dados:**

Após a criação dos modelos é necessário informar ao Django que ocorreram alterações e que essas alterações devem ser propagadas para o banco de dados, isso se chama Migração;

Altere o settings.py para direcionar para o banco de dados que deseja utilizar;

```
# Database
# https://docs.djangoproject.com/en/3.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

(Vamos alterar para utilização do MySQL posteriormente)

Comando: python manage.py makemigrations

Após o makemigrations é necessário aplicar as mudanças ao banco de dados, sendo necessário:

- Configuração do banco de dados no arquivo settings.py;
- Tenha sido gerados os modelos migrations com o makemigrations;

Comando: python manage.py migrate

- **Criando um Super Usuário para acessar o Admin:**

Comando: python manage.py createsuperuser

Digite os dados necessários

Rode o server e entre no admin: <http://127.0.0.1:8000/admin>

- **Criação do model Products:**

Vamos criar duas classes para cuidar dos produtos.

A primeira são as Categorias de Produtos:

Atenção a utilização dos slugs:

Slugs são labels contendo somente letras, números, underscores ou hifens.

Serão utilizados para criação de urls amigáveis para os usuários.

```
from autoslug import AutoSlugField
from django.db import models
from django.urls import reverse
from model_utils.models import TimeStampedModel

class Category(TimeStampedModel):
    name = models.CharField(verbose_name = 'Descrição', max_length=255, unique=True)
    slug = AutoSlugField(unique=True, always_update=False, populate_from="name")

    class Meta:
        ordering = ("name",)
        verbose_name = "category"
        verbose_name_plural = "categories"

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse("products:list_by_category", kwargs={"slug": self.slug})
```


A segunda classe refere-se ao produto em si.

Atenção a herança da classe `TimeStampedModel`, que simplifica a criação de classes que utilizam o campo data e hora de criação e modificação de algum cadastro.

```
class Product(TimeStampedModel):
    category = models.ForeignKey(
        Category, related_name="products", on_delete=models.CASCADE
    )
    name = models.CharField(max_length=255)
    slug = AutoSlugField(unique=True, always_update=False, populate_from="name")
    image = models.ImageField(upload_to="products/%Y/%m/%d", blank=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    # Podemos controlar também por um controle de estoque
    # Esse será o desafio para turma... criar um controle de estoque para controlar os produtos
    # que serão apresentados na tela para o usuário
    is_available = models.BooleanField(default=True)

    class Meta:
        ordering = ("name",)

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse("products:detail", kwargs={"slug": self.slug})
```

○ Criação do model Orders:

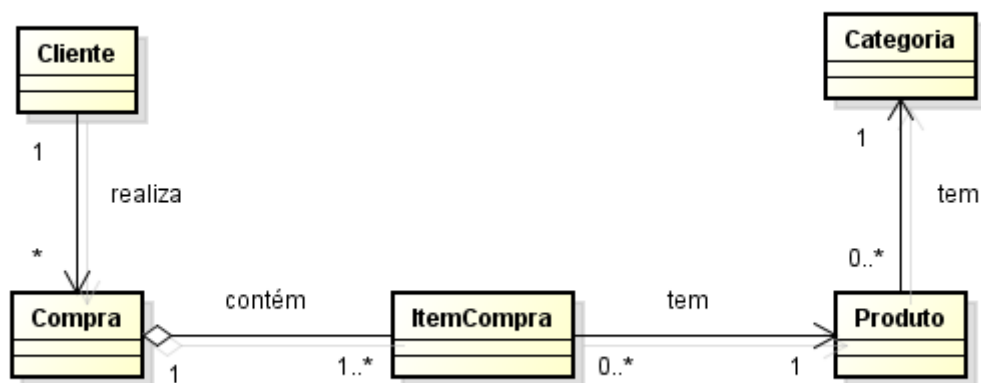
Vamos criar o app Orders e criar seus respectivos models

django-admin startapp orders

Nesse caso teremos que criar duas classes:

A primeira Orders, que armazenará informações das compras dos usuários.

E a segunda OrdersItems, que armazenará quais foram os itens comprados.



```

from django.db import models
from model_utils.models import TimeStampedModel
from products.models import Product
from users.models import User
# Create your models here.

class Order(TimeStampedModel):
    user = models.ForeignKey(User, related_name="users", on_delete=models.CASCADE)
    name = models.CharField("Nome Completo", max_length=250)
    postal_code = models.CharField("CEP", max_length=9)
    address = models.CharField("Endereço", max_length=250)

    def __str__(self):
        return str(self.id)

    @property
    def get_total_price(self):
        total_cost = sum(item.get_cost for item in self.items.all())
        return total_cost

```

```

class OrderItem(models.Model):
    order = models.ForeignKey(Order, related_name="items", on_delete=models.CASCADE)
    product = models.ForeignKey(
        Product, related_name="order_items", on_delete=models.CASCADE
    )
    quantity = models.PositiveSmallIntegerField(blank=False, null=False)

    def __str__(self):
        return str(self.id)

    @property
    def get_cost(self):
        return self.product.price * self.quantity

```

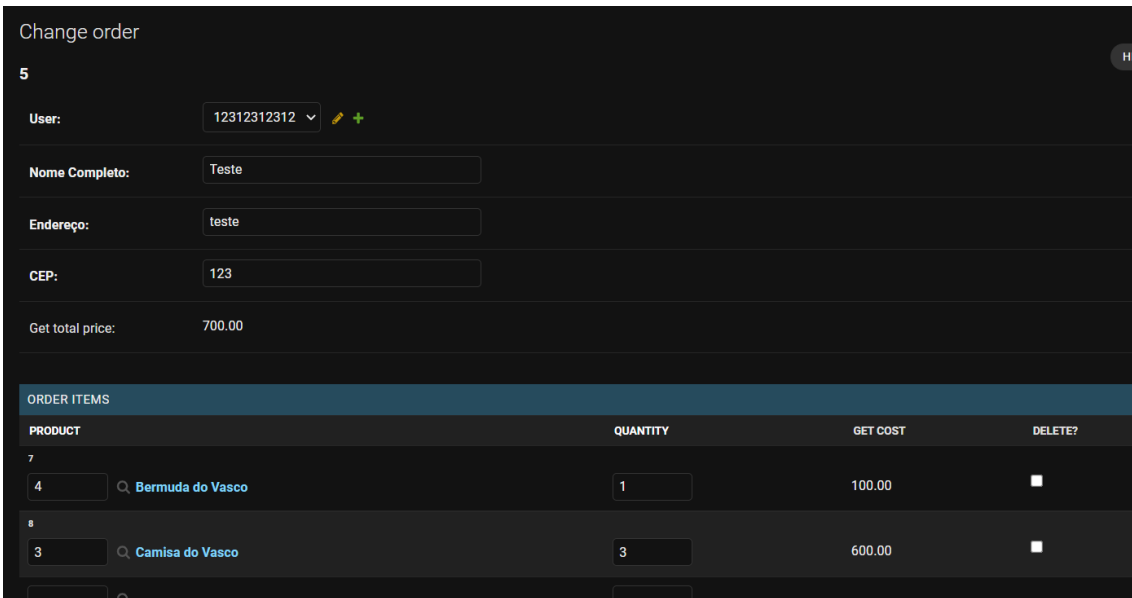
No admin:

```
from django.contrib import admin
from .models import OrderItem, Order

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    raw_id_fields = ['product']
    fields = ('product', 'quantity', 'get_cost')
    readonly_fields = ('get_cost',)

class OrderAdmin(admin.ModelAdmin):
    fields = ('user', 'name', 'address', 'postal_code', 'get_total_price')
    list_display = ['__str__', 'user', 'name', 'address', 'get_total_price']
    inlines = [OrderItemInline]
    readonly_fields = ('get_total_price',)
    admin.site.register(Order, OrderAdmin)
```

Agora conseguimos criar vendas diretamente no admin:



Change order

5

User: 12312312312

Nome Completo: Teste

Endereço: teste

CEP: 123

Get total price: 700.00

ORDER ITEMS			
PRODUCT	QUANTITY	GET COST	DELETE?
4 Bermuda do Vasco	1	100.00	<input type="checkbox"/>
3 Camisa do Vasco	3	600.00	<input type="checkbox"/>
		-	

Na sequência precisamos criar o carrinho de compra, onde os clientes poderão adicionar os produtos que desejarem comprar.

Para tal necessitamos fazer uso das Sessões (Session).

Vamos utilizar o Session do Django Framework para nos auxiliar nessa tarefa

A Sessão será valida até expirar ou o Usuário finalizar a compra

Verifique no settings.py na seção de Middleware:

django.contrib.sessions.middleware.SessionMiddleware

Vamos criar o app cart e adicioná-lo no settings.py

django-admin startapp cart

```
✓ INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'pages.apps.PagesConfig', #pages  
    'products.apps.ProductsConfig', #products  
    'users.apps.UsersConfig', #users  
    'orders.apps.OrdersConfig', #orders  
    'cart.apps.CartConfig', #cart  
]
```

Vamos adicionar o Cart_Session_ID:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')  
MEDIA_URL = '/media/'  
  
CART_SESSION_ID = 'cart'  
  
# Default primary key field type  
# https://docs.djangoproject.com/en/3.2/ref/settings/#default-auto-field  
  
DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
```

Na sequência precisamos criar o arquivo cart.py dentro do app cart.

Esse arquivo se encarregará de manipular as sessões do carrinho

Apresentar o código no VSCode

Criando o formulário que permitirá os usuários a colocarem produtos no carrinho:

Em cart.py -> forms.py

```
from django import forms

# Quantidade de produtos até 20 unidades
PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(choices=PRODUCT_QUANTITY_CHOICES, coerce=int)
    update = forms.BooleanField(required=False, initial=False, widget=forms.HiddenInput)
```

Vamos criar as views agora:

```
from django.shortcuts import render, redirect, get_object_or_404
from django.views.decorators.http import require_POST
from products.models import Product
from .cart import Cart
from .forms import CartAddProductForm

@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)

    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product, quantity=cd['quantity'], update_quantity=cd['update'])

    return redirect('cart:cart_detail')

def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')

def cart_detail(request):
    cart = Cart(request)
    return render(request, 'cart/detail.html', {'cart': cart})
```

Finalizando com urls.py:

```
from django.conf.urls import url
from cart import views

app_name = "cart"

urlpatterns = [
    url('', views.cart_detail, name='cart_detail'),
    url(r'^add/(?P<product_id>\d+)/$', views.cart_add, name='cart_add'),
    url(r'^remove/(?P<product_id>\d+)/$', views.cart_remove, name='cart_remove'),
]
```

Para encerrar vamos criar o template do carrinho: