**Факультет Python-разработчик**

**ДИПЛОМНАЯ РАБОТА**

**Разработка интернет-магазина на django**

**Разработал: студент Шаньшеров**

**Николай Аркадьевич**

**Содержание**

|  |  |  |
| --- | --- | --- |
| № | Раздел | страница |
|  | Введение | 3 |
| 1. | Подготовка к созданию проекта. | 6 |
| 1.1 | Установка Django. | 6 |
| 1.2. | Создание структуры проекта | 6 |
| 2. | Создание интернет-магазина. | 7 |
| 2.1 | Создание приложения shop. | 7 |
| 2.1.1 | Создание моделей. | 7 |
| 2.1.2 | Регистрация моделей на сайте администрирования. | 9 |
| 2.1.3 | Создание представлений. | 9 |
| 2.1.4 | Создание шаблонов приложения shop. | 11 |
| 2.2 | Создание корзины покупок. | 14 |
| 2.2.1 | Создание приложения для управления корзинами. | 15 |
| 2.2.2 | Создание представлений корзины. | 17 |
| 2.2.3 | Добавление элементов в корзину. | 18 |
| 2.2.4 | Создание шаблона отображения корзины | 19 |
| 2.2.5 | Настройка корзины в контексте запроса. | 20 |
| 2.3 | Регистрация заказов. | 22 |
| 2.3.1 | Создание модели order. | 22 |
| 2.3.2 | Добавление моделей заказов на сайт администрирования. | 23 |
| 2.3.3 | Создание заказов клиентов. | 24 |
| 2.3.4 | Создание шаблонов заказов. | 26 |
|  | Заключение | 28 |

**Введение**

**Почему Django?**

Django — это фреймворк, то есть набор готовых инструментов и функций. С его помощью можно быстрее и проще реализовывать на Python сайты и приложения, которые работают в браузере.

Конечно, можно создать веб-сервис на Python и без Django, но тогда очень многое придется писать и настраивать с нуля самостоятельно. Если использовать программирование на Django, можно сосредоточиться только на уникальных функциях веб-сервиса. Вот основные возможности, которые даёт этот фреймворк:

● Настроенный веб-сервер, который будет обрабатывать запросы от пользователей к веб-сервису.

● Готовые механизмы для авторизации пользователей.

● Простые шаблоны веб-страниц.

● Административный интерфейс для управления контентом сервиса — наполнения, изменения, обновления используемых данных.

● Система кэширования для увеличения скорости загрузки и открытия страниц через браузеры, внешние клиенты или приложения.

● Интерфейсы и адаптеры для подключения к различным типам баз данных.

С помощью Django можно очень быстро, как из конструктора, настроить и запустить работающий веб-сервис — а потом программировать только специфичные функции и бизнес-логику.

Django работает по модели MVT — Model-View-Template, или «Модель–Представление–Шаблон». Она разделяет внутреннюю логику работы сайта, внешний вид страниц для пользователя и реакции веб-сервиса на внешние воздействия:

● **Модель** обеспечивает внутреннюю работу сайта: подключение к базе данных, формат хранения информации и другие функции.

● **Представление** определяет данные, которые нужно показать пользователю, и отправляет их туда, где они должны быть показаны. Также оно принимает запросы пользователей и реагирует на них — например, обновляет страницу после отправки данных формы.

● **Шаблон** — это каркас страницы на Django, которую увидит пользователь.

**Преимущества Django:**

● Простота. Создать простой сайт с помощью программирования на Django можно максимально быстро, обладая минимальными знаниями в Python.

● Гибкость. Базовые функции фреймворка можно изменять и настраивать под потребности конкретного проекта.

● Стандартизация. Разобраться в проекте на Django просто, если уже имел с ним дело.

● Безопасность. В фреймворк уже встроены методы защиты, обеспечивающие безопасность данных пользователей и всего веб-приложения.

● Удобное администрирование. Благодаря графическому административному интерфейсу проще управлять БД и содержимым сайта.

● Популярность. У Django обширная документация и огромное сообщество лояльных разработчиков. Всегда можно найти ответ на возникший вопрос или получить совет.

● Поддержка разных баз данных. Благодаря Django можно легко интегрировать веб-сервис с любой популярной БД без большого количества сложных настроек.

**Недостатки** **Django:**

● Некоторая ограниченность. Есть вещи, которые в Django заданы строго, и изменить их нельзя. Если нет желания писать их с нуля, придется смириться с ограничениями.

● Требовательность к ресурсам. Из-за обилия встроенных инструментов Django требует больше мощностей для запуска, чем нативный код на Python.

● Слишком большая сложность для простых проектов. Маленький одностраничный сайт может быть быстрее собрать с нуля, чем писать полную структуру проекта и программировать на Django.

Руководствуясь вышеизложенным, и тем, что проекты на Django являются расширяемыми, я выбрал для своего дипломного проекта разработку простого интернет-магазина с помощью этого фреймворка.

1. **Подготовка к созданию проекта.**
   1. **Установка Django.**

Установка Django предполагает установку фреймворка в изолированную среду python.

В данном проекте я использовал предварительно установленную IDE PyCharm Community Edition 2023.1.2, которая облегчает написание кода, отлично подходит для разработки на языке python.

PyCharm позволяет создать проект python в изолированной среде.

Находясь в папке проекта python, с помощью средств установки PyCharm, устанавливаем Django. Вместе с ним устанавливаются все необходимые пакеты.

* 1. **Создание структуры проекта**

Для создания проекта Django выполняем следующую команду из терминала:

django-admin startproject diplom

Это создает проект django с именем diplom.

Структура проекта:

diplom/

manage.py

diplom /

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

1. **Создание интернет-магазина.**
   1. **Создание приложения shop.**

В терминале переходим в папку проекта и вводим команду:

django-admin startapp shop

Это создает в нашем проекте приложение shop в котором будет происходить основная работа.

Следующим шагом необходимо зарегистрировать наше приложение в проекте, для этого в файле settings.py проекта добавим в INSTALLED\_APPS наше приложение:

INSTALLED\_APPS = [

# ...

'shop',

]

* + 1. **Создание моделей.**

Для создания моделей необходимо отредактировать файл model.py приложения shop следующим образом:

from django.db import models

from django.urls import reverse

class Category(models.Model):

name = models.CharField(max\_length=200, db\_index=True)

slug = models.SlugField(max\_length=200, db\_index=True, unique=True)

class Meta:

ordering = ('name',)

verbose\_name = 'Категория'

verbose\_name\_plural = 'Категории'

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

return self.name

def get\_absolute\_url(self):

return reverse('product\_list\_by\_category', args=[self.slug])

class Product(models.Model):

category = models.ForeignKey(Category, related\_name='products', on\_delete=models.CASCADE)

name = models.CharField(max\_length=200, db\_index=True)

slug = models.SlugField(max\_length=200, db\_index=True)

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)

stock = models.PositiveIntegerField()

available = models.BooleanField(default=True)

created = models.DateTimeField(auto\_now\_add=True)

updated = models.DateTimeField(auto\_now=True)

class Meta:

ordering = ('name',)

index\_together = (('id', 'slug'),)

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

return self.name

def get\_absolute\_url(self):

return reverse('product\_detail', args=[self.id, self.slug])

Это модели **Category** и **Product**. Модель Category состоит из поля **name** и **slug**. Рассмотрим поля модели Product:

**category** : Это ForeignKey модели **Category**. Это отношение "многие к одному": продукт относится к одной категории, а категория содержит несколько продуктов

**name** : Название продукта.

**slug** : Алиас продукта(его URL).

**image** : Изображение продукта.

**description** : Необязательное описание для продукта.

**price** : Это поле DecimalField. В нем используется десятичное число Python. Десятичный тип для хранения десятичного числа с фиксированной точностью.

**stock** : Это поле PositiveIntegerField для хранения остатков данного продукта.

**available** : Это булево значение, указывающее, доступен ли продукт или нет. Позволяет включить/отключить продукт в каталоге.

**created** : Это поле хранит дату когда был создан объект.

**updated** : В этом поле хранится время последнего обновления объекта.

В классе **Meta** модели **Product** используем параметр мета **index\_together**, чтобы задать индекс для полей **id** и **slug**. Определяем этот индекс, поскольку планирую запросить продукты с помощью id и slug. Оба поля индексируются вместе для улучшения представлений для запросов, использующих эти два поля.

Затем выполняем миграции и синхронизируем модели с базой данных.

Для этого в терминале вводим команды:

python manage.py makemigrations

python manage.py migrate

* + 1. **Регистрация моделей на сайте администрирования.**

Добавляем модели на сайт администрирования, чтобы легко управлять категориями и продуктами. Изменяем файл **admin.py** приложения **shop** и добавляем в него следующий код:

from django.contrib import admin

from .models import Category, Product

class CategoryAdmin(admin.ModelAdmin):

list\_display = ['name', 'slug']

prepopulated\_fields = {'slug': ('name',)}

admin.site.register(Category, CategoryAdmin)

class ProductAdmin(admin.ModelAdmin):

list\_display = ['name', 'slug', 'price', 'stock', 'available', 'created', 'updated']

list\_filter = ['available', 'created', 'updated']

list\_editable = ['price', 'stock', 'available']

prepopulated\_fields = {'slug': ('name',)}

admin.site.register(Product, ProductAdmin)

Теперь можно создать суперпользователя для работы в админке Django с помощью команды в терминале:

python manage.py createsuperuser

* + 1. **Создание представлений.**

Чтобы отображать продукты и категории продуктов, необходимо создать представление для списка всех продуктов или фильтрации продуктов по данной категории. Изменяем файл **views.py** приложения **shop** и добавляем в него следующий код:

from django.shortcuts import render, get\_object\_or\_404

from .models import Category, Product

from cart.forms import CartAddProductForm

def product\_list(request, category\_slug=None):

category = None

categories = Category.objects.all()

products = Product.objects.filter(available=True)

if category\_slug:

category = get\_object\_or\_404(Category, slug=category\_slug)

products = products.filter(category=category)

return render(request, 'shop/product/list.html', {'category': category, 'categories': categories, 'products': products})

def product\_detail(request, id, slug):

product = get\_object\_or\_404(Product, id=id, slug=slug, available=True)

cart\_product\_form = CartAddProductForm()

return render(request, 'shop/product/detail.html', {'product': product,

'cart\_product\_form': cart\_product\_form})

Для получения экземпляра **Product** в **product\_detail** рассчитываются параметры **id** и **slug**. Этот экземпляр можно получить только с помощью id, поскольку он является уникальным атрибутом. Тем не менее, в URL-адрес можно добавить slug для создания удобных URL-адресов для продуктов.

Далее необходимо определить шаблоны URL-адресов. Создаем новый файл в каталоге приложения shop – **urls.py**.

Добавляем в него следующий код:

from django.urls import re\_path

from . import views

urlpatterns = [

re\_path(r'^$', views.product\_list, name='product\_list'),

re\_path(r'^(?P<category\_slug>[-\w]+)/$', views.product\_list, name='product\_list\_by\_category'),

re\_path(r'^(?P<id>\d+)/(?P<slug>[-\w]+)/$', views.product\_detail, name='product\_detail'),

]

Это шаблоны URL-адресов для продуктов. Я определил два различных шаблона URL-адреса для представления **product\_list**: шаблон с именем product\_list, который вызывает product\_list view без каких-либо параметров; и шаблон с именем **product\_list\_by\_category**, который предоставляет параметр category\_slug в представлении для фильтрации продуктов по данной категории. Добавил шаблон для выносного элемента **product\_detail**, который передает в представление параметры id и slug для извлечения конкретного продукта.

Теперь изменяем файл **urls.py** проекта **diplom**:

from django.contrib import admin

from django.urls import re\_path, include

urlpatterns = [

re\_path(r'admin/', admin.site.urls),

re\_path(r'^', include('shop.urls')),

]

* + 1. **Создание шаблонов приложения shop.**

Создаем шаблоны для списка продуктов. Для этого в каталоге приложения создаем следующую структуру:

templates/

shop/

base.html

product/

list.html

detail.html

Сначала создаем базовый шаблон – **base.html**:

<!DOCTYPE html>

{% load static %}

<html>

<head>

<meta charset="UTF-8">

<title>{% block title %}Магазин{% endblock %}</title>

<link href="{% static 'css/base.css' %}" rel="stylesheet">

</head>

<body>

<div>

<a href="/" class="logo">Магазин</a>

</div><br>

<div class="cart">

{% with total\_items=cart|length %}

{% if cart|length > 0 %}

Ваша корзина:

<a href="{% url 'cart\_detail' %}">

Продуктов {{ total\_items }},

на сумму {{ cart.get\_total\_price }} руб.

</a>

{% else %}

Ваша корзина пуста.

{% endif %}

{% endwith %}

</div>

<div>

{% block content %}

{% endblock %}

</div>

</body>

</html>

Шаблон **list.html** для отображения списка продуктов наследуем от **base.html:**

{% extends "shop/base.html" %}

{% load static %}

{% block title %}

{% if category %}{{ category.name }}{% else %}Продукты{% endif %}

{% endblock %}

{% block content %}

<div>

<h1>Категории</h1>

<ul>

<li {% if not category %} class="selected" {% endif %}>

<a href="{% url 'product\_list' %}">Все</a>

</li>

{% for c in categories %}

<li {% if category.slug == c.slug %} class="selected" {% endif %}>

<a href="{{ c.get\_absolute\_url }}">{{ c.name }}</a>

</li>

{% endfor %}

</ul>

</div>

<div class="product-list">

<h1>{% if category %}{{ category.name }}{% else %}Продукты{% endif %}</h1>

{% for product in products %}

<div class="item">

<a href="{{ product.get\_absolute\_url }}">

<img src="{% if product.image %}{{ product.image.url }}{% else %}{% static 'img/no\_image.jpg' %}{% endif %}">

</a><br>

<a href="{{ product.get\_absolute\_url }}">{{ product.name }}</a><br>

цена {{ product.price }} руб.

</div><br><br>

{% endfor %}

</div>

{% endblock %}

Также шаблон **detail.html** для отображения детализации наследуем от **base.html**:

{% extends "shop/base.html" %}

{% load static %}

{% block title %}

{% if category %}{{ category.title }}{% else %}Продукты{% endif %}

{% endblock %}

{% block content %}

<div class="product-detail">

<img src="{% if product.image %}{{ product.image.url }}{% else %}{% static 'img/no\_image.jpg' %}{% endif %}">

<h1>{{ product.name }}</h1>

<h2><a href="{{ product.category.get\_absolute\_url }}">{{ product.category }}</a></h2>

<p class="price">цена {{ product.price }} руб.</p>

{{ product.description|linebreaks }}

</div><br>

<form action="{% url 'cart\_add' product.id %}" method="post">

{{ cart\_product\_form }}

{% csrf\_token %}<br><br>

<input type="submit" value="Добавить в корзину">

</form>

{% endblock %}

Поскольку используем ImageField для хранения изображений продуктов, нам необходим сервер разработки для обслуживания загруженных файлов изображений. Добавляем в файл **settings.py** проекта **diplom** следующие параметры:

MEDIA\_URL = '/media/'

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media/')

Чтобы Джанго обрабатывал загруженные мультимедийные файлы с помощью сервера разработки, добавим в файл **urls.py diplom** следующий код:

from django.conf import settings

from django.conf.urls.static import static

urlpatterns = [

# ...

]

if settings.DEBUG:

urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

**2.2 Создание корзины покупок.**

После создания каталога продуктов следующим шагом является создание корзины покупок, которая позволит пользователям выбирать продукты, которые они хотят приобрести. Корзина позволяет пользователям выбирать нужные продукты и временно хранить их во время просмотра сайта до тех пор, пока не будет размещен заказ. Корзина должна быть сохранена в сессии, чтобы элементы корзины хранились во время визита пользователя.

Необходимо использовать Django's session framework для сохранения товаров корзины. Корзина будет храниться в сессии до тех пор, пока она не завершится. Также потребуется построить дополнительные модели Джанго для корзины и ее товаров.

Необходимо создать простую структуру, которая может быть сериализована в JSON для хранения элементов корзины в сессии. Корзина должна включать следующие данные для каждого содержащегося в ней элемента:

- id экземпляра Product

- Количество товара, выбранное для данного продукта

- Цена единицы для данного продукта

Поскольку цены на продукцию могут различаться, мы приближаемся к сохранению цены продукта вместе с продуктом, когда он добавляется в корзину. Таким образом, будем сохранять ту же цену, которую пользователи увидели при добавлении товара в корзину, даже если цена продукта изменится после этого.

Теперь необходимо управлять созданием корзин и связывать их с сеансами. Корзина покупок должна работать следующим образом:

- Когда требуется корзина, мы проверяем, установлен ли пользовательский session key. Если в сессии не задана корзина, мы создаем новую корзину и сохраняем ее в session key корзины.

- Для последовательных запросов мы выполняем одну и ту же проверку и получая номенклатуры корзины из session key корзины. Мы извлекаем элементы корзины из базы данных и связанные с ними объекты продукта.

Изменяем файл **settings.py** проекта и добавляем в него следующий параметр:

CART\_SESSION\_ID = 'cart'

Это ключ, который используется для хранения корзины в сессии пользователя.

**2.2.1 Создание приложения для управления корзинами.**

В терминале запускаем команду для создания приложения **cart**:

python manage.py startapp cart

Затем добавляем в файл **settings.py** проекта наше приложение.

Создаем новый файл **cart.py** в каталоге приложения **cart** со следующим кодом:

from decimal import Decimal

from django.conf import settings

from shop.models import Product

class Cart(object):

def \_\_init\_\_(self, request):

"""

Инициализируем корзину

"""

self.session = request.session

cart = self.session.get(settings.CART\_SESSION\_ID)

if not cart:

# save an empty cart in the session

cart = self.session[settings.CART\_SESSION\_ID] = {}

self.cart = cart

def add(self, product, quantity=1, update\_quantity=False):

"""

Добавить продукт в корзину или обновить его количество.

"""

product\_id = str(product.id)

if product\_id not in self.cart:

self.cart[product\_id] = {'quantity': 0,

'price': str(product.price)}

if update\_quantity:

self.cart[product\_id]['quantity'] = quantity

else:

self.cart[product\_id]['quantity'] += quantity

self.save()

def save(self):

# Обновление сессии cart

self.session[settings.CART\_SESSION\_ID] = self.cart

# Отметить сеанс как "измененный", чтобы убедиться, что он сохранен

self.session.modified = True

def remove(self, product):

"""

Удаление товара из корзины.

"""

product\_id = str(product.id)

if product\_id in self.cart:

del self.cart[product\_id]

self.save()

def \_\_iter\_\_(self):

"""

Перебор элементов в корзине и получение продуктов из базы данных.

"""

product\_ids = self.cart.keys()

# получение объектов product и добавление их в корзину

products = Product.objects.filter(id\_\_in=product\_ids)

for product in products:

self.cart[str(product.id)]['product'] = product

for item in self.cart.values():

item['price'] = Decimal(item['price'])

item['total\_price'] = item['price'] \* item['quantity']

yield item

def \_\_len\_\_(self):

"""

Подсчет всех товаров в корзине.

"""

return sum(item['quantity'] for item in self.cart.values())

def get\_total\_price(self):

"""

Подсчет стоимости товаров в корзине.

"""

return sum(Decimal(item['price']) \* item['quantity'] for item in

self.cart.values())

def clear(self):

# удаление корзины из сессии

del self.session[settings.CART\_SESSION\_ID]

self.session.modified = True

**2.2.2 Создание представлений корзины.**

Теперь, когда у есть класс Cart для управления корзиной, необходимо создать представления для добавления, обновления или удаления элементов из нее. Необходимо создать следующие представления:

- Представление для добавления или обновления номенклатур в корзине, которое может обрабатывать текущие и новые количества

- Представление для удаления товаров из тележки

- Представление для отображения элементов корзины и итоговых значений

Создадаем представление для добавления элементов в корзину. Редактируем файл **views.py** приложения **cart**:

from django.shortcuts import render, redirect, get\_object\_or\_404

from django.views.decorators.http import require\_POST

from shop.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\_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\_detail')

def cart\_detail(request):

cart = Cart(request)

return render(request, 'cart/detail.html', {'cart': cart})

Представления для добавления товаров в корзину, обновления количества, удаления товаров из корзины и отображения корзины созданы. Создаеме новый файл в каталоге приложения **cart** - **urls.py**. Добавляем следующие URL-адреса:

from django.urls import re\_path

from . import views

urlpatterns = [

re\_path(r'^$', views.cart\_detail, name='cart\_detail'),

re\_path(r'^add/(?P<product\_id>\d+)/$', views.cart\_add, name='cart\_add'),

re\_path(r'^remove/(?P<product\_id>\d+)/$', views.cart\_remove, name='cart\_remove'),

]

В файл **urls.py** проекта **diplom** добавляем следующий адрес:

re\_path(r'^cart/', include('cart.urls')),

**2.2.3 Добавление элементов в корзину.**

Чтобы добавить элементы в корзину, нужна форма, позволяющая пользователю выбрать количество добавляемого товара. Создадим файл **forms.py** в каталоге приложения **cart** и добавляем в него следующий код:

from django import forms

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, label="Количество")

update = forms.BooleanField(required=False, initial=False, widget=forms.HiddenInput)

Эта форма будет использоваться для добавления продуктов в корзину. Класс **CartAddProductForm** содержит следующие поля:

**quantity**: позволяет пользователю выбрать количество между 1-20. Поле TypedChoiceField с coerce=int служит для преобразования ввода в целое число.

**update**: позволяет указать, следует ли добавлять сумму к любому существующему значению в корзине для данного продукта (False) или если существующее значение должно быть обновлено с заданным значением (True). Для этого поля используется графический элемент HiddenInput, поскольку не требуется показывать его пользователю.

**2.2.4 Создание шаблона отображения корзины**

Представления cart\_add и cart\_remove не нужнаются в шаблонах, но необходимо создать шаблон для представления cart\_detail для отображения элементов корзины и итоговых значений.

Создаем следующую структуру файла в каталоге приложения cart:

templates/

cart/

detail.html

Изменяем шаблон cart/detail.html и добавляем в него следующий код:

{% extends "shop/base.html" %}

{% load static %}

{% block title %}

Ваша корзина покупок

{% endblock %}

{% block content %}

<h1>Ваша корзина покупок</h1>

<table class="cart2">

<thead>

<tr>

<th>Изображение</th>

<th> Продукт </th>

<th> Количество </th>

<th> Удалить </th>

<th> Цена за единицу </th>

<th> Стоимость </th>

</tr>

</thead>

<tbody>

{% for item in cart %}

{% with product=item.product %}

<tr>

<td>

<a href="{{ product.get\_absolute\_url }}">

<img src="

{% if product.image %}

{{ product.image.url }}

{% else %}

{% static 'img/no\_image.jpg' %}

{% endif %}">

</a>

<td>{{ product.name }}</td>

<td>{{ item.quantity }}</td>

<td><a href="{% url 'cart\_remove' product.id%}">Удалить</a></td>

<td class="num">{{ item.price }} руб.</td>

<td class="num">{{ item.total\_price }} руб.</td>

</tr>

{% endwith %}

{% endfor %}

<tr>

<td>Итого стоимость корзины</td>

<td colspan="4"></td>

<td class="num">{{ cart.get\_total\_price }} руб.</td>

</tr>

</tbody>

</table>

<p class="text-center">

<a href="{% url 'product\_list' %}">Продолжить выбор продуктов</a>

<br><br>

<a href="{% url 'order\_create' %}">

Заказать

</a>

</p>

{% endblock %}

Это шаблон, используется для отображения содержимого корзины. Он содержит таблицу с элементами, хранящимися в текущей корзине. Мы разрешаем пользователям изменять количество выбранных продуктов, используя форму, которая учитывается в представлении cart\_add. Мы также разрешаем пользователям удалять элементы из корзины, предоставляя для каждого из них ссылку Удалить.

**2.2.5 Настройка корзины в контексте запроса.**

Создадим обработчик контекста для установки текущей корзины в контекст запроса для шаблонов. Так сможем получить доступ к этой корзине в любом шаблоне.

Создаем новый файл в каталоге приложения **cart** - **context\_processors.py**. Контекстные процессоры могут размещаться в любом месте кода, но создание их в отдельном документе помогает лучше организовать структуру проекта. Добавляем в файл следующий код:

from .cart import Cart

def cart(request):

return {'cart': Cart(request)}

Контекстный процессор — это функция, которая получает объект запроса в качестве параметра и возвращает словарь объектов, которые будут доступны всем шаблонам, визуализированным с помощью RequestContext. В обработчике контекста мы создаем объект корзины с помощью объекта request и делаем его доступным для шаблонов в виде переменной с именем cart.

Редактируем файл **settings.py** проекта и добавляем  'cart.context\_processors.cart'  для параметра  context\_processors  в параметрах TEMPLATES. После изменения этот параметр будет выглядеть следующим образом:

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [os.path.join(BASE\_DIR, 'templates')]

,

'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',

'cart.context\_processors.cart',

],

},

},

]

Теперь контекстный процессор будет выполняться при каждом просмотре шаблона с использованием RequestContext Джанго. Переменная cart будет задана в контексте для шаблонов.

* 1. **Регистрация заказов.**

При извлечении товаров из корзины для покупок необходимо сохранить заказ в базе данных. Заказы будут содержать информацию о клиентах и продуктах, которые они покупают.

Создаем новое приложение для управления заказами клиентов, используя следующую команду:

python manage.py startapp orders

В  settings.py  проекта добавляем приложение  **orders**  в параметры **INSTALLED\_APPS**:

INSTALLED\_APPS = (

# ...

'orders',

)

Тем самым мы активировали приложение **orders**.

**2.3.1 Создание модели order.**

Потребуется модель для хранения сведений о заказе и модель для хранения купленных товаров, включая их количество и цену. Изменяем файл **models.py** приложения **orders** и добавим в него следующий код:

from django.db import models

from shop.models import Product

class Order(models.Model):

first\_name = models.CharField(max\_length=50)

last\_name = models.CharField(max\_length=50)

email = models.EmailField()

address = models.CharField(max\_length=250)

postal\_code = models.CharField(max\_length=20)

city = models.CharField(max\_length=100)

created = models.DateTimeField(auto\_now\_add=True)

updated = models.DateTimeField(auto\_now=True)

paid = models.BooleanField(default=False)

class Meta:

ordering = ('-created',)

verbose\_name = 'Заказ'

verbose\_name\_plural = 'Заказы'

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

return 'Order {}'.format(self.id)

def get\_total\_cost(self):

return sum(item.get\_cost() for item in self.items.all())

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)

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

quantity = models.PositiveIntegerField(default=1)

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

return '{}'.format(self.id)

def get\_cost(self):

return self.price \* self.quantity

Модель **Order** содержит несколько полей для сведений о клиенте и поле paid, которое по умолчанию имеет значение False. Будем использовать это поле для различения оплаченных и неоплаченных заказов. Также определяем метод get\_total\_cost(), чтобы получить общую стоимость товаров, купленных в этом заказе.

Модель **OrderItem** позволяет хранить продукт, количество и цену, уплаченную за каждый товар. Метод get\_cost() – для возврата стоимости товара.

Далее необходимо создать и применить новые миграции.

**2.3.2 Добавление моделей заказов на сайт администрирования.**

Теперь добавим модели заказов на сайт администрирования. Изменяем файл **admin.py** приложения **orders**:

from django.contrib import admin

from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):

model = OrderItem

raw\_id\_fields = ['product']

class OrderAdmin(admin.ModelAdmin):

list\_display = ['id', 'first\_name', 'last\_name', 'email',

'address', 'postal\_code', 'city', 'paid',

'created', 'updated']

list\_filter = ['paid', 'created', 'updated']

inlines = [OrderItemInline]

admin.site.register(Order, OrderAdmin)

Используем ModelInline для модели OrderItem, чтобы включить ее в качестве inline встроенного в класс OrderAdmin. Inline режим позволяет включить модель для отображения на той же странице редактирования, что и родительская модель.

**2.3.3 Создание заказов клиентов.**

Теперь необходимо использовать только что созданные модели заказов для сохранения товаров, содержащихся в корзине для покупок, когда пользователь наконец пожелает разместить заказ. Функции создания нового заказа будут работать следующим образом:

- Мы предоставляем форму заказа для заполнения пользовательских данных.

- Создается новый экземпляр заказа с данными, введенными пользователями, а затем создается связанный экземпляр OrderItem для каждого товара в корзине.

- Очищаем все содержимое корзины и перенаправляем пользователей на страницу финальную страницу.

Для ввода сведений о заказе необходима форма. Создаем новый файл в каталоге приложения **orders** и назоваем его **forms.py**. Добавляем в него следующий код:

from django import forms

from .models import Order

class OrderCreateForm(forms.ModelForm):

class Meta:

model = Order

fields = ['first\_name', 'last\_name', 'email', 'address', 'postal\_code', 'city']

Теперь нам необходимо представление, чтобы обработать форму и создать новый заказ. Изменяем файл **views.py** приложения **orders** и добавляем в него следующий код:

from django.shortcuts import render

from .models import OrderItem

from .forms import OrderCreateForm

from cart.cart import Cart

def order\_create(request):

cart = Cart(request)

if request.method == 'POST':

form = OrderCreateForm(request.POST)

if form.is\_valid():

order = form.save()

for item in cart:

OrderItem.objects.create(order=order,

product=item['product'],

price=item['price'],

quantity=item['quantity'])

# очистка корзины

cart.clear()

return render(request, 'orders/order/created.html',

{'order': order})

else:

form = OrderCreateForm

return render(request, 'orders/order/create.html',

{'cart': cart, 'form': form})

В представлении order\_create мы получаем текущую корзину из сессии с cart = Cart(request). В зависимости от метода запроса мы будем выполнять следующие задачи:

- **GET request** : Создается экземпляр формы OrderCreateForm и отображается шаблон orders/order/create.html

- **POST request** : Проверяет валидность введенных данных. Если данные являются допустимыми, то для создания нового экземпляра заказа будет использоваться order = form.save(). Затем мы сохраняем его в базу данных, а затем храним в переменной order. После создания заказа мы перейдем по товарам корзины и создадим OrderItem для каждого из них. Наконец, мы очищаем содержимое корзины

Создаем новый файл **urls.py** в каталоге приложения **orders**. Добавляем в него следующий код:

from django.urls import re\_path

from . import views

urlpatterns = [

re\_path(r'^create/$', views.order\_create, name='order\_create'),

]

Это шаблон URL-адреса для представления order\_create. Далее редактируем файл **urls.py** проекта **diplom** и включаем в следующий шаблон:

re\_path(r'^orders/', include('orders.urls')),

**2.3.4. Создание шаблонов заказов.**

Создаем следующую структуру в каталоге **orders**:

templates/

orders/

order/

create.html

created.html

Затем редактируем шаблон **create.html**:

{% extends "shop/base.html" %}

{% block title %}Checkout{% endblock %}

{% block content %}

<h1>Разместить заказ</h1>

<div class="order-info">

<h3>Ваш заказ</h3>

<ul>

{% for item in cart %}

<li>

{{ item.product.name }}, количество - {{ item.quantity }},

<span>на сумму {{ item.total\_price }} руб.</span>

</li>

{% endfor %}

</ul>

<p>Стоимость заказа: {{ cart.get\_total\_price }} руб.</p>

</div>

<form action="." method="post" class="order-info">

{{ form.as\_p }}<br>

<p><input type="submit" value="Разместить заказ" class="order-info"></p>

{% csrf\_token %}

</form>

{% endblock %}

Этот шаблон отображает товары корзины, включая итоговую сумму, и форму для размещения заказа.

Теперь редактируем шаблон **created.html**:

{% extends "shop/base.html" %}

{% block title %}Thank you{% endblock %}

{% block content %}

<h1>Спасибо что вы с нами</h1>

<h2>

<strong>Ваш заказ № {{ order.id }} успешно сохранен. Продукты будут доставлены Вам курьером.

Оплата при получении</strong>.</h2>

{% endblock %}

Это шаблон, который мы показываем при успешном создании заказа.

**Заключение.**

В заключении я бы хотел отметить, что фреймворк **Django** является большим подспорьем для разработки веб приложений.

Конечно, в данной работе использованы далеко не все возможности данного фреймворка. Но в развитии данного проекта имеется возможность добавления новых приложений (добавление платежных систем, авторизации клиентов, добавления скидочных купонов).

Особую благодарность хотел бы выразить **персоналу GeekBrains**!