Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Ивановский государственный энергетический университет
имени В.И. Ленина»

Кафедра систем управления

Пояснительная записка к курсовой работе
Разработка back-end части магазина одежды Iconic
Дисциплина: «Разработка Web-приложений»

Выполнил: студент группы З-44
Болонин Е.В.
Проверил: к.т.н., доцент кафедры СУ
Маршалов Е.Д.

Иваново 2023

Содержание

1. Цель работы

Целью данной курсовой работы является разработка back-end части магазина одежды Iconic в рамках дисциплины «Разработка Web-приложений». Для достижения этой цели мы воспользуемся современными инструментами веб-разработки, а именно фреймворками Django и Django Rest Framework (DRF). Django обеспечит нам удобное и эффективное создание веб-приложения, а DRF – мощный инструмент для построения RESTful API.

На протяжении работы мы рассмотрим основные принципы проектирования и реализации back-end-сервера, включая обработку HTTP-запросов, взаимодействие с базой данных, обеспечение безопасности данных и авторизации пользователей.

2. Анализ предметной области

Так, в ходе выполнения данной курсовой работы будет реализовано web-приложение для удобного взаимодействия пользователей и продуктов. Таким образом, предметная область данной курсовой работы включает в себя разработку back-end части магазина, специализированного на продаже одежды. Рассмотрим ключевые аспекты данной предметной области:

1. Магазин одежды

Основной объект продажи является – продукт, которые представляют собой одежду или различные аксессуары.

2. Регистрация и управление пользователями:

Пользователи могут регистрироваться в магазине, создавая аккаунты для покупки и создания товаров.

3. Каталог:

Реализация каталога, где одежда классифицируются по категориям, цене, названию и другим параметрам. Система поиска должна обеспечивать удобный доступ к разнообразной одежде.

4. Административная часть:

Администратор должен иметь возможность управлять контентом, модерировать продукты, решать спорные ситуации и обеспечивать безопасность платформы.

3. Задачи проекта

Так, в соответствии с определенной предметной областью был выделен следующий ключевой функционал, который необходимо реализовать с помощью Django и Django REST Framework:

• Создание основного приложения, включающего в себя модели, формы, представления, URL-маршруты и шаблоны, работу с API.

• Определение моделей данных, таких как Product (продукты), Productcategory (категории), стандартная модель user (пользователь) для хранения информации о товарах, их категориях, а также пользователей сайта.

• Создание форм, например для добавления новых товаров и для регистрации новых пользователей, фильтрации и так далее.

• Реализация представлений для отображения списка продуктов, отдельных страниц для каждого продукта, категории, а также для добавления и редактирования товаров.

• Определение URL-маршрутов для соответствия различным представлениям и функционалу приложения.

• Создание шаблонов HTML для отображения страниц, включая домашнюю страницу, страницы «Каталог», «О сайте», а также страницы добавления товаров, страницы отдельных товаров, страницы входа и регистрации пользователей.

• Использование фильтров для динамического отображения товаров по имени, категории и сортировки их по цене.

• Использование встроенной административной панели Django для управления базой данных.

• Применение механизмов безопасности Django, таких как проверка аутентификации пользователей.

4. Разработка web-приложения

4.1. Создание основных моделей БД

Концепция приложения «магазина по продаже одежды» подразумевает в себе следующие модели:

1. Продукт. Товар, которому посвящено всё наше приложение. В данной модели должны быть следующие поля:
   - Название
   - Описание
   - Краткое описание
   - Цена
   - Количество
   - Фото
   - Категория

- Создатель
- Флаг наличия товара
- SLUG, предназначенный для формирования url-адреса того или иного продукта.

2. Категория. Соответствует категории нашего товара и содержит следующие поля:
   - Имя
   - Описание категории
   - SLUG

После того, как были определены основные модели и их атрибуты, они были реализованы с помощью фреймворка Django. Для этого был создан файл «models.py». В нём будут храниться все наши модели. В нём и были указаны все наши модели. На рисунке №1, можно увидеть код, соответствующий модели «Product».

```python
class Product(models.Model):
    name = models.CharField(max_length=256, verbose_name="Товар")
    description = models.TextField(blank=True, verbose_name="Описание")
    short_description = models.CharField(max_length=64, verbose_name="Краткое описание")
    price = models.DecimalField(decimal_places=2, max_digits=8, verbose_name="Цена")
    image = models.ImageField(upload_to=user_directory_path, verbose_name="Изображение", blank=True)
    category = models.ForeignKey(to=ProductCategory, verbose_name=("Категория"), on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(verbose_name="Количество", default=1)
    is_available = models.BooleanField(verbose_name="В наличии", default=True)
    user = models.ForeignKey(User, verbose_name='Пользователь', on_delete=models.CASCADE)
    slug = models.SlugField(max_length=255, unique=True, db_index=True, verbose_name="URL")


    def __str__(self) -> str:
        return self.name


    def get_absolute_url(self):
        return reverse('item', kwargs={'item_slug': self.slug})

    class Meta:
        verbose_name = "Товар"
        verbose_name_plural = "Товары"
        ordering = ['name', 'category']
```

Рис.1. Код модели «Product».

Так, данный код представляет модель Django под названием Product. Она представляет информацию о продуктах. Так, она содержит поля, которые были описаны выше. Кроме того, есть магический методы __str__, который возвращает название того или иного продукта; get_absolute_url, который используется для получения абсолютного url-адреса объекта «Product» а также определены некоторые метаданные, управляющие отображением в административном интерфейсе Django, а так же на сайте.

Для того, чтобы добавленные модели появились в нашей БД, мы опять же воспользуемся встроенными в фреймворк Django функциями «makemigrations» и «migrate». После этого наша БД успешно создана. На рисунке №2 можно увидеть результат.

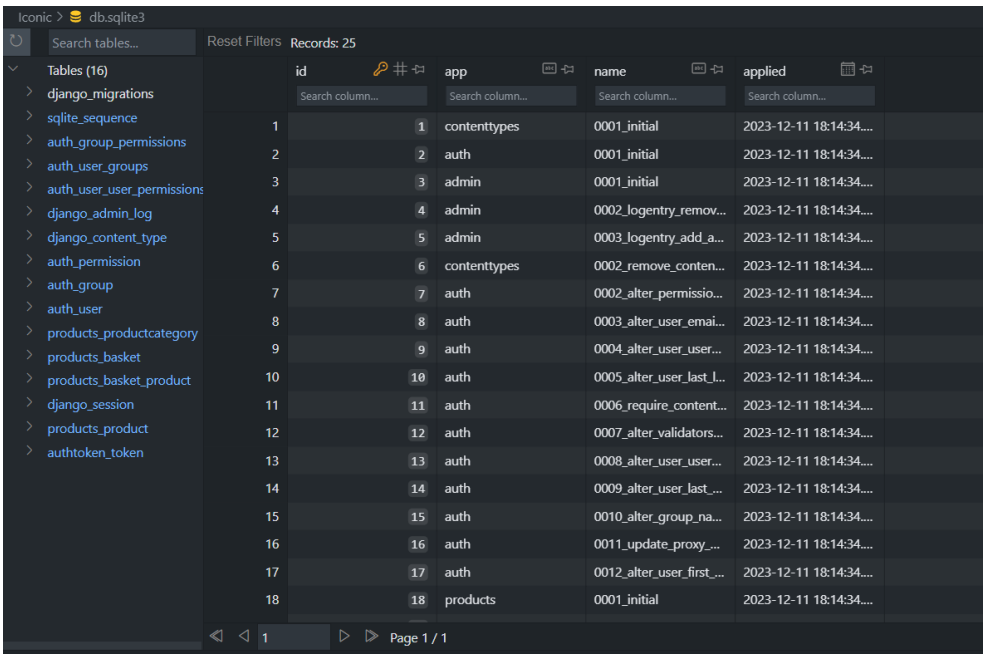

Рис.2. Созданная БД.

4.2. Настройка админ-панели Django

Перед тем, как переходить к взаимодействиям с созданными моделями, давайте сначала настроим нашу админ-панель, которую предоставляет фреймворк Django. Для этого создадим новый файл «admin.py», в котором будут описаны правила отображения тех или иных моделей в нашей админ-панели. Рассмотрим настройку на примере модели Product.

```
class ProductsAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'description',
                    'short_description', 'price', 'image', 'category', 'is_available',
                    'quantity', 'user')
    list_display_links = ('name', )
    search_fields = ('name', 'short_description', 'price', 'category')
    list_editable = ('description', 'short_description', 'price', 'image', 'quantity', 'is_available', 'user')
    list_filter = ('price', 'name', 'is_available', 'quantity')
    prepopulated_fields = {"slug": ('name', )}
```

Рис.3. Настройка админ-панели. Класс «Product Admin».

Так, с помощью переменной «list_display» мы задаём то, какие поля у нас будут отображаться в админ-панели для модели «Product». Переменная «list_display_links» указывает, какие поля в списке записей будут представлять собой ссылки на детали этой записи. Переменная «search_fields» задает список полей, по которым можно выполнять поиск в административном интерфейсе. Поле «list_editable» задает список полей, которые мы можем менять в админке. Поле «list_filter» задает список полей, по которым можно выполнять фильтрацию в административном интерфейсе. Кроме того, важно отметить, что для удобного отображения всех названий в админ-панели мы заранее в модели добавили атрибут «verbose-name» для всех полей модели, а также в классе Meta указали то, как админ-панель должна отображать нам название нашей модели в единственном и множественном числе.

Теперь давайте посмотрим на полученный результат. Для этого предварительно необходимо создать пользователя, который будет являться администратором нашего сайта с помощью команды «py manage.py createsuperuser», после чего запустить сервер с

помощью команды «py manage.py runserver» и дописать в url-адрес «/admin». Таким образом, мы откроем админ-панель Django. Таким образом видим следующее:

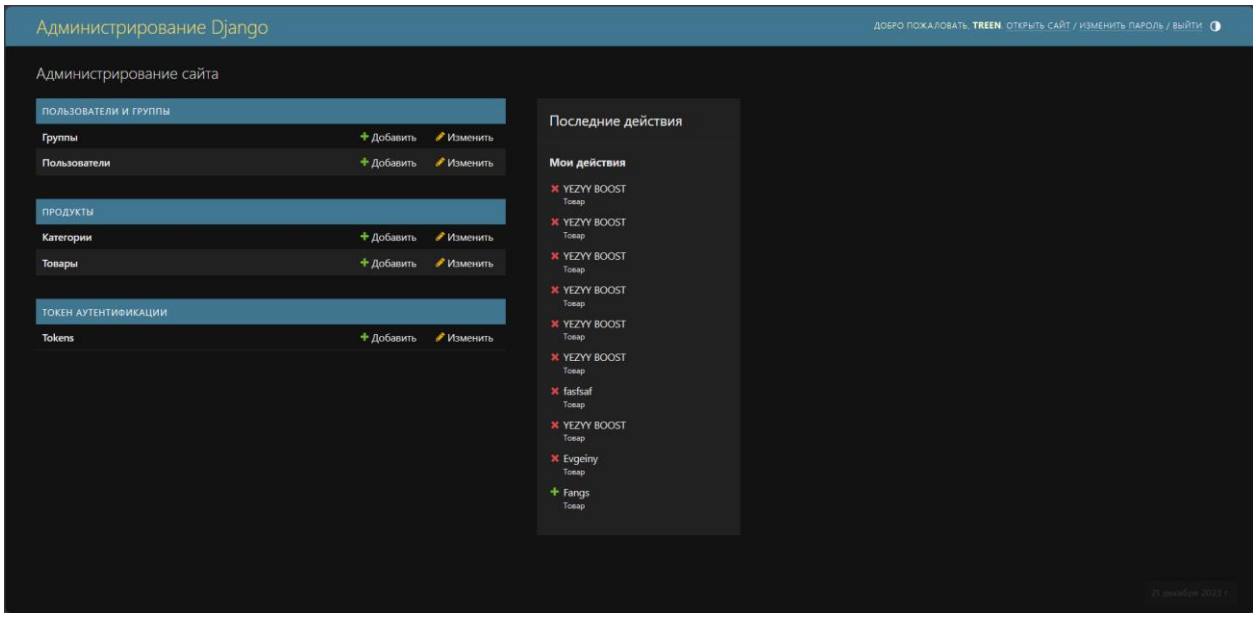

Рис.4. Главная страница админ-панели.

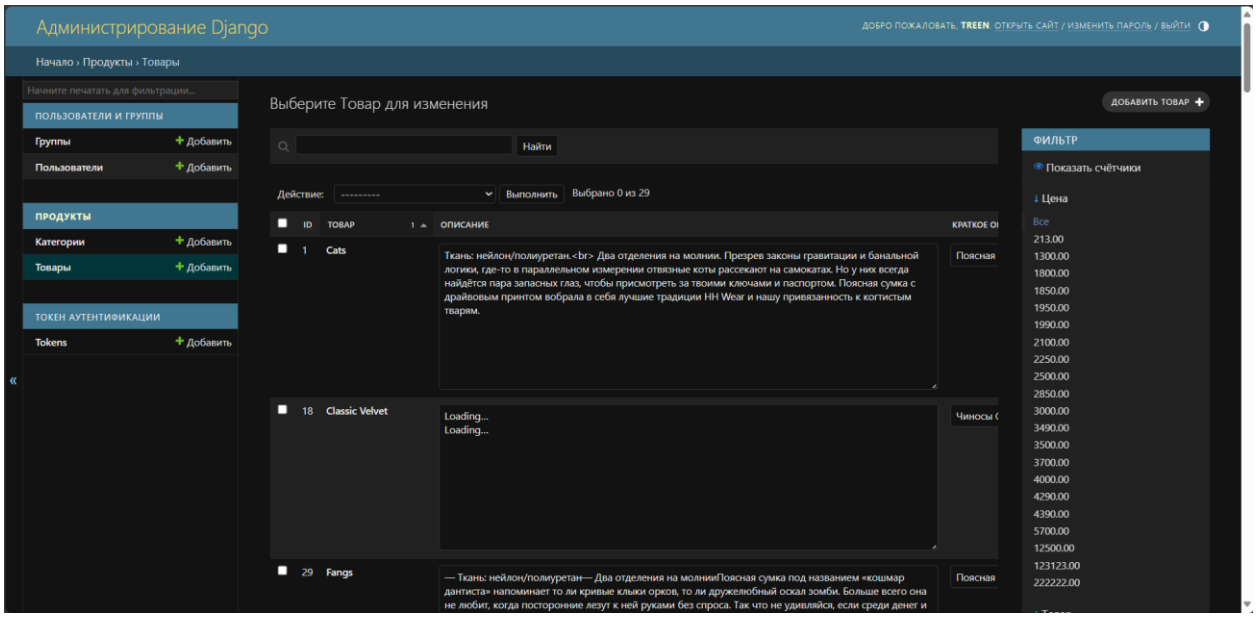

Рис.5. Админ-панель. Вкладка «Продукты».

Таким образом, мы получили полностью настроенную для удобной работы админ-панель Django.

4.3. Работа с шаблонами в Django

После того, как мы создали нашу БД с нужными для корректной работы приложения моделями и админ-панелью необходимо реализовать взаимодействия с БД непосредственно на сайте. Это можно реализовать несколькими путями: с помощью шаблонов и с помощью API. Для начала разберём реализацию с помощью шаблонов, а после уже с помощью API.

В контексте нашей темы, на сайте должны быть следующие страницы и взаимодействия с моделями:

1. Главная страница.
2. Каталог.
3. Каталог по категориям
4. Авторизация и регистрация

Для удобства дальнейшей разработки для начала создадим файлы «views.py», «urls.py», «filters.py», «utils.py», а также «forms.py». После этого создадим файл «base.html», который будет являться основой каждой из наших страниц сайта. В нём нам необходимо написать желаемый html-код для нашего сайта. Тут важно отметить, что для того, чтобы использовать css, а также загружать различные картинки, нам необходимо предварительно указать «static» путь, в котором будут находиться внешние файлы. Для этого в файле «settings.py» был добавлен код, представленный на рисунке №6.

```
STATIC_URL = 'static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
STATICFILES_DIRS = []
```

Рис. 6. Указание пути «static» в файле «settings.py».

После этого в начале шаблона «base.html» необходимо написать следующий код «{% load static %}». Таким образом наш base.html будет использоваться во всех шаблонах.

Так, в начале мы указываем, что любой файл html является расширением шаблона «base.html», после чего в блоке контента описываем содержимое страницы. Это используется для избежания дублирования кода в шаблонах и для повышения читаемости кода.

Но это еще не всё, что нужно для корректного отображения наших страниц. Как можно увидеть на рисунке №7, в шаблонах мы можем использовать различные переменные и передавать информацию из них для отображения в html-шаблон. Для этого в Django используются представления, которые описываются в файле «views.py». Ниже представлен код представления, соответствующий главной странице нашего сайта.

```
class CatalogHome(DataMixin, ListView):
    model = Product
    template_name = 'products/index.html'
    context_object_name = 'products'

    def get_context_data(self,*, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title="Каталог", form=FilterProductForm(self.request.GET), cat_shearch=1)
        return dict(list(context.items()) + list(c_def.items()))

    def get_queryset(self) -> QuerySet[Any]: #  Какие записи должны быть на странице отображены
        queryset = super().get_queryset()
        st_filter = ProductFilter(self.request.GET, queryset)
        return st_filter.qs
```

Рис.8. Код представления «HomePage»

Т.к. на странице каталога могут отображаться самые популярные товары. В данном представлении моделью выбрана модель Product, а в качестве шаблона в переменной «template_name» используется файл index.html. После этого мы описываем функцию «get_context_data», которая используется для получения данных, используемых в контексте этой страницы, что как раз-таки нужно для передачи данных из переменных непосредственно в html-шаблон.

Также для корректной работы представлений нам необходимо указывать url-адреса, которые соответствуют тому или иному представлению. Это делается в файле «urls.py», код которого представлен на рисунке №9.

```
urlpatterns = [
    path('', CatalogHome.as_view(), name='home'),
    path('item/<slug:item_slug>/', ShowItem.as_view(), name='item'),
    path('catalog/<slug:cat_slug>/', ShowCatalog.as_view(), name='category'),
    path('about/', about.as_view(), name='about'),
    path('addpage/', AddItem.as_view(), name='add_page'),
    path('logout/', logout_user, name='logout'),
    path('login/', LoginUser.as_view(), name='login'),
    path('register/', RegisterUser.as_view(), name='register'),
]
```

Рис.9. Файл «urls.py». URL-адреса существующих страниц.

Страница «Каталог», на которой должны отображаться все продукты, а также должен быть реализован поиск по названию и цене, сортировка по категории.

Для корректного отображения данной страницы был написан миксин под названием «DataMixin». Механизм миксинов применяется для случаев, когда нужно дополнительно к уже существующей иерархии добавить какие-либо общие для разнородных классов данные или методы. В нашем случае они используются для избежания дублирования кода в представлениях. Так, наш «DataMixin» будет содержать в себе функцию, которая будет передавать в представления ссылки, использующиеся в меню нашего сайта, а также дополнительную информацию о моделях. Реализация данного миксина представлена ниже.

```
class DataMixin:
    paginate_by = 3
    def get_user_context(self, **kwargs):
        context = kwargs
        user_menu = menu.copy()
        if not self.request.user.is_authenticated:
            user_menu.pop(1)

        context['menu'] = user_menu

        if 'cat_shearch' not in context:
            context['cat_shearch'] = 0

        if 'category_selected' not in context:
            context['category_selected'] = 0
        return context
```

Рис. 11. Реализация миксина DataMixin

Далее сразу опишем то, как устроен фильтр, который будет использоваться для поиска и сортировки на странице «Каталог».

Код данного фильтра будет находиться в файле «filters.py» и выглядеть так, как показано на рисунке №13.

```python
class ProductFilter(FilterSet):
    name = CharFilter(field_name='name', lookup_expr='contains', label='Название')
    price = CharFilter(field_name='price', lookup_expr='contains', label='Цена')

    class Meta:
        model = Product
        fields = ['name', 'price', 'category']
```

Рис.12. Код фильтра «ProductFilter».

Так, в нём описаны поля, по которым мы с можем осуществлять поиск (поля name и price), а также поле category, отвечающее за сортировку по категории.

Класс Meta определяет метаданные для фильтра. Здесь указывается модель (Product) и поля, по которым можно выполнять фильтрацию.

Таким образом, представление нашей страницы «Каталог» будет выглядеть так, как показано на рисунке №14.

Важно отметить, что данное представление наследуется от встроенного в Django класса «ListView», который отвечает за отображение наших данных в виде списка. Так, для пагинации мы будем использовать встроенное в данный класс свойство «paginate_by». Для этого в коде задаётся переменная «paginate_by». Ей присваивается значение того, сколько объектов у нас должно находится на странице. Для сохранения состояния фильтрации при пагинации страницы был использован пакет spurl.

```
{% if page_obj.has_other_pages %}
    <ul class="list-pages">
        {% if page_obj.has_previous %}
            <a href="?page={{ page_obj.previous_page_number }}" class="page-num">&lt;</a>
        {% endif %}
        {% for p in paginator.page_range %}
            {% if page_obj.number == p %}
                <li class="page-num selected">{{ p }}</li>
            {% elif p >= page_obj.number|add:-2 and p <= page_obj.number|add:2 %}
                <li>
                    <a href="{% spurl query=request.GET set_query='page={{ p }}'%}" class="page-num">{{ p }}</a>
                </li>
            {% endif %}
        {% endfor %}
        {% if page_obj.has_next %}
            <a href="?page={{ page_obj.next_page_number }}" class="page-num">&gt;</a>
        {% endif %}
    </ul>
{% endif %}
```

Рис.13. Шаблон «base.html».Условия для корректной работы пагинации

Так, мы проверяем применяется ли пагинация для данной страницы, если да, то проверяем на какой странице мы находимся, и в зависимости от этого выводим кнопки для переключения на следующую и предыдущую страницы.

Таким образом, страница «Каталог» выглядит следующим образом:

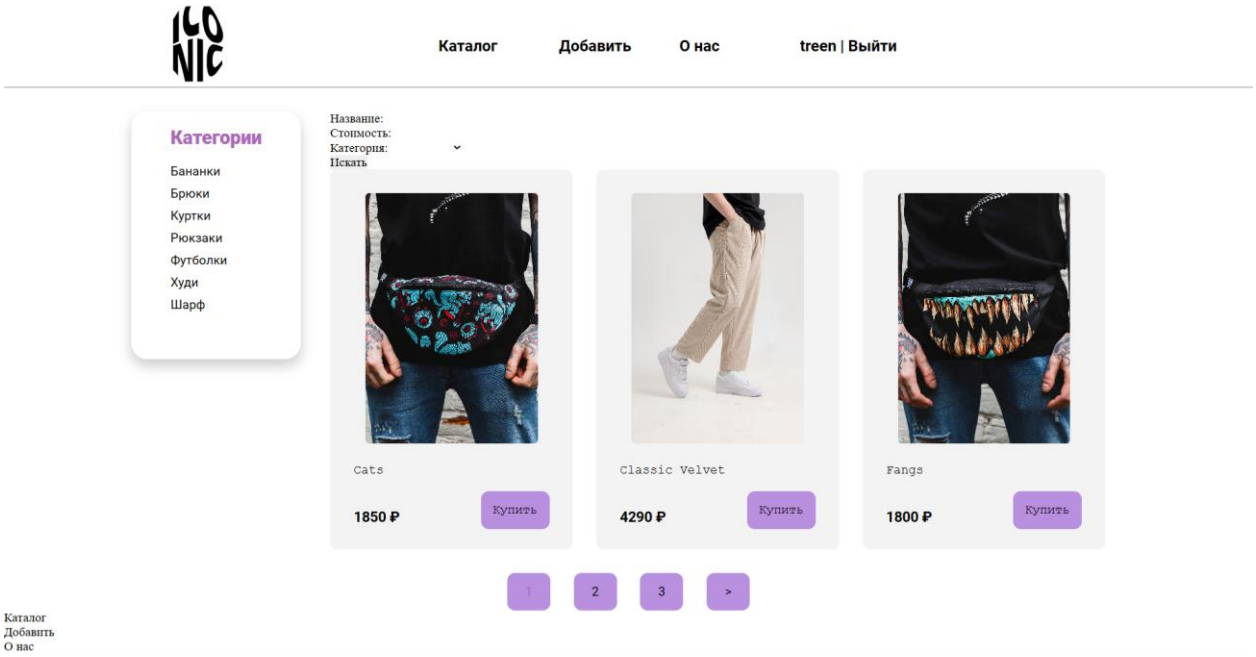

Рис.16. Страница «Каталог».

Далее разберём, то как реализована страница самого товара. Для этого используются: представление «ShowItem» и HTML-шаблон «item.html».

Так, представление «ShowItem» реализовано аналогично ранее описанным представлениям, единственное отличие состоит в том, что здесь мы используем встроенную функцию «get_object» в класс DetailView для получения конкретного объекта, соответствующего модели. Код данного представления показан ниже.

```
class ShowItem(DataMixin, DetailView):
    model = Product
    template_name = 'products/item.html'
    slug_url_kwarg = 'item_slug'
    context_object_name = 'products'

    def get_context_data(self,*, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title=context['products'].name)
        return dict(list(context.items()) + list(c_def.items()))
```

Рис.17. Представление «ShowItem».

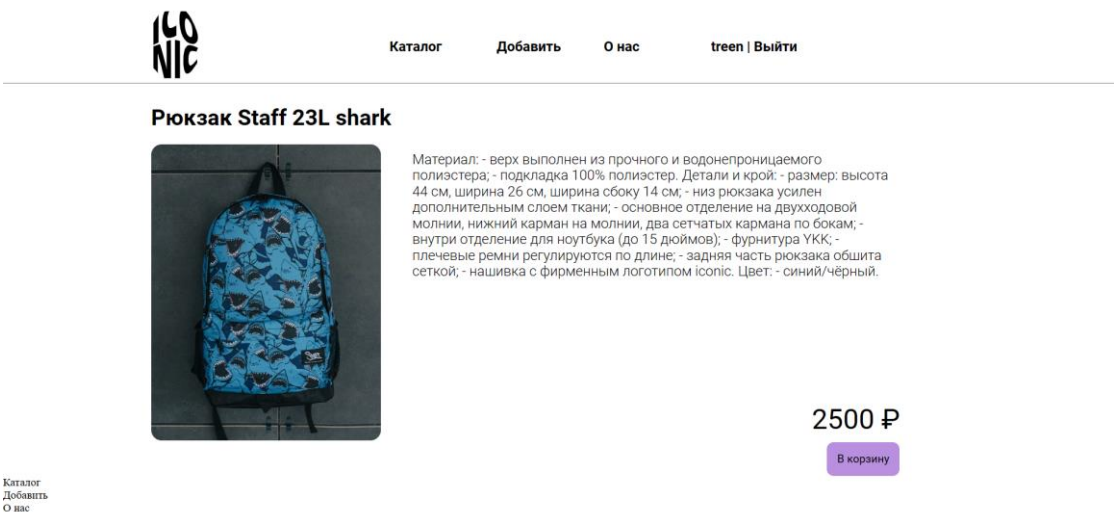

Рис.23. Страница отдельного предмета.

4.4 Работа с формами. Авторизация и регистрация пользователей

Теперь разберём, как реализована страница «Добавление товара». Для этого используется форма, связанная с моделью, а именно «AddProductForm», которая находится в файле «forms.py». Её код представлен ниже:

```python
class AddProductForm(forms.ModelForm):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['category'].empty_label = "Категория не выбрана"

    class Meta:
        model = Product
        fields = ['name', 'description', 'short_description','image', 'price', 'slug', 'category', 'quantity', 'is_available']
        widgets = {
            'name': forms.TextInput(attrs={'class': 'form__input'}),
            'short_description': forms.TextInput(attrs={'class': 'form__input'}),
            'slug': forms.TextInput(attrs={'class': 'form__input'}),
            'description': forms.Textarea(attrs={'cols':60, 'rows': 10}),
        }

    def clean_name(self): #  Пользовательская валидация
        name = self.cleaned_data['name']
        if len(name)>200:
            raise forms.ValidationError("Длинна превышает 200 символов")
        return name
```

Рис.24. Страница отдельного предмета.

Она работает следующим образом:

1. Для того, чтобы привязать форму к какой-либо из моделей, мы наследуем её от базового класса ModelForm, которая автоматически создает форму, соответствующую структуре модели, и обрабатывает валидацию данных в соответствии с правилами модели.
2. В классе Meta указываются необходимые для заполнения поля модели, а также сама модель, с которой мы работаем. Так же там указаны классы для оформления полей ввода в модель.
3. В функции «__init__» мы можем указать начальное состояние формы. Так, в данной форме указано автозаполнение поля category.
4. В функции, которая является валидатором для поля name, «clean_name» прописано условие, что если пользователь ввёл название длиной выше 200 символов, то мы

должны вывести ошибку, если всё в норме, то имя записывается в БД с остальной информацией.

Представление страницы добавления выглядит следующим образом:

```
class AddItem(LoginRequiredMixin, DataMixin, CreateView):
    form_class = AddProductForm
    template_name = 'products/addpage.html'
    login_url = reverse_lazy('home')
    raise_exception = True #  Генерирует 403 (нет доступа) если не авторизован
    # success_url = reverse_lazy('home') #  Перенаправление после добавления поста

    def get_context_data(self,*, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title="Добавление товара")
        return dict(list(context.items()) + list(c_def.items()))

    def form_valid(self, form):
            form.instance.user = self.request.user
            return super().form_valid(form)
```

Рис.25. Представление «AddItem»

Данное представление уже наследуется от другого базового класса, а именно «CreateView», которое позволяет удобно сделать страницу создания какого-либо объекта. А также мы наследуем миксин «LoginRequiredMixin», который позволяет воспользоваться формой только авторизированным пользователям.

Таким образом, сначала мы указываем, какую форму нам необходимо использовать на данной странице, указываем шаблон. В функции «form_valid» мы получаем данные из формы и проверяем их на валидность, так же мы добавляем к каждому продукту текущего пользователя, который создает этот товар.

Далее разберём регистрацию и авторизацию пользователей. («RegisterUser» и «LoginUser»). Разберём их реализацию, начиная с регистрации. Код данного контроллера представлен ниже:

```
class RegisterUser(DataMixin, CreateView):
    form_class = RegisterUserForm
    template_name = 'products/register.html'
    success_url = reverse_lazy('login')

    def get_context_data(self,*, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title="Регистрация")
        return dict(list(context.items()) + list(c_def.items()))

    def form_valid(self, form):
        user = form.save()
        login(self.request, user)
        return redirect('home')
```

Рис.26. Представление «RegisterUser»

Данная функция берёт форму RegisterUserForm и использует её в register.html. Если форма была заполнена успешно, то пользователь сохраняется в БД и нас перенаправляют на страницу каталога.

Код формы:

```
class RegisterUserForm(UserCreationForm):
    username = forms.CharField(label='Логин', widget=forms.TextInput(attrs={'class': 'form__input'}))
    email = forms.EmailField(label='Email', widget=forms.EmailInput(attrs={'class': 'form__input'}))
    password1 = forms.CharField(label='Пароль', widget=forms.PasswordInput(attrs={'class': 'form__input'}))
    password2 = forms.CharField(label='Пароль повтор', widget=forms.PasswordInput(attrs={'class': 'form__input'}))

    class Meta:
        model = User
        fields = ('username', 'email', 'password1', 'password2')
        widget = {
            'username': forms.TextInput(attrs={'class': 'form__input'}),
            'password1': forms.PasswordInput(attrs={'class': 'form__input'}),
            'password2': forms.PasswordInput(attrs={'class': 'form__input'}),
        }
```

Рис.27. Класс «RegisterUserForm»

Так, данная форма наследуется от базового класса UserCreationForm, которая автоматически создаёт форму регистрации, таким образом, в самой форме указываем поля, необходимые для регистрации, а в классе Meta указываем используемые поля.

В представлении RegisterUser, наследуемом от базового класса «CreateView» определяем используемую форму, шаблон и функцию «form_valid» для проверки валидности введённых данных. В остальном представление ничем не отличается от ранее описанных представлений. Код представлен на рисунке №26.

Для авторизации используется форма «LoginUserForm», которую мы наследуем от базового класса «AuthenticationForm». Данная форма предельно проста, тут мы указываем поля, которые необходимо заполнить пользователю для входа в аккаунт. Кроме того, определяем тип данных полей. Код данной формы представлен на рисунке №28.

```
class LoginUserForm(AuthenticationForm):
    username = forms.CharField(label='Логин', widget=forms.TextInput(attrs={'class': 'form__input'}))
    password = forms.CharField(label='Логин', widget=forms.PasswordInput(attrs={'class': 'form__input'}))
```

Рис.28. Форма «LoginUserForm»

Представление «LoginUser» наследуем от базового класса «LoginView» и определяем функцию «get_success_url», которая отвечает за перенаправление пользователя на определённую страницу после успешной авторизации. Код данного представления показан на рисунке №29.

```python
class LoginUser(DataMixin, LoginView):
    form_class = LoginUserForm
    template_name = 'products/login.html'

    def get_context_data(self, *, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        c_def = self.get_user_context(title='Авторизация')
        return dict(list(context.items()) + list(c_def.items()))

    def get_success_url(self):
        return reverse_lazy('home')
```

Рис.29. Класс «LoginUser»

5 Работа с Django REST Framework

Последним пунктом выполнения курсовой работы является использование Django REST Framework. С помощью него мы прописываем взаимодействие между сервером и различными клиентами. Т.е. мы воссоздаём возможность захождения на наш сайт с различных устройств. На стороне сервера создается специальный программный интерфейс, который сокращенно называется: API и Django REST Framework как раз обеспечивает взаимодействие любого конечного устройства через API с сайтом на сервере.

Для начала добавим новые url-адреса в файл «urls.py». Они будут соответствовать страницам, на которых мы можем отправлять различные запросы, взаимодействуя с API.

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

    path('api/v1/drf-auth/', include('rest_framework.urls')),

    path('api/v1/auth/', include('djoser.urls')),
    re_path(r'^auth/', include('djoser.urls.authtoken')),


    path('api/v1/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/v1/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/v1/token/verify/', TokenVerifyView.as_view(), name='token_verify'),


    path('api/v1/product-control/', ProductAPIList.as_view()),
    path('api/v1/product-control/<int:pk>/', ProductAPIUpdate.as_view()),
    path('api/v1/product-control-del/<int:pk>/', ProductAPIDestroy.as_view()),

    path('api/v1/', include(router.urls)), # https://127.0.0.1:8000/api/v1/product/ Viewset

    path('api/v1/category/', ProductCategoryAPIList.as_view(), name='category_api_get_post'), # get Class
    path('api/v1/categorydetail/<int:pk>/', ProductCategoryAPIDetailView.as_view(), name='category_detail_api'), # for item

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

]
```

Рис. 30. URL-адреса для работы с API

Начнём с разбора сериализаторов, представлений и ограничения доступай для модели Product. Здесь сериализатор наследуется от базового класса «ModelSerializer», встроенный в Django REST Framework. Как понятно из названия данных сериализатор связывает данный класс с моделью Product. В классе Meta указаны какие поля он будет брать из модели БД, а так же прописано своё поле user, куда мы заносим текущего пользователя.

```python
class ProductSerializer(serializers.ModelSerializer):
    user = serializers.HiddenField(default=serializers.CurrentUserDefault())
    class Meta:
        model = Product
        fields = "__all__"
```

Рис. 31. Сериализатор модели Product

Рассмотрим представления, которые используются для работы с товарами. Оно наследуется от класса ListCreateAPIView, который предоставляет функционал вывода записей (GET запрос) и добавление записей в БД(POST). Здесь поле queryset говорит от том, какие записи надо отправить json ответом, serializer_class указывает на то, какой сериализатор надо использовать, permission_classes указывает на модификатор доступа, в нашем случае, если пользователь вошел в свой аккаунт, то он может читать и добавлять данные, иначе он может только читать.

```python
class ProductAPIList(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    permission_classes = (IsAuthenticatedOrReadOnly,)
    pagination_class = ProductAPIListPagination
    # authentication_classes = (TokenAuthentication, )
```

Рис. 32. Представление модели ProductAPIList

Так же в этом классе указано поле pagination_class, которое содержит класс с настройками пагинации внутри интерфейса API.

```python
class ProductAPIListPagination(PageNumberPagination):
    page_size = 5 # Число записей на странице
    page_size_query_param = 'page_size'
    max_page_size = 10000 # Максимальное значение записей
    # На странице API
```

Рис. 33. Представление модели ProductAPIListPagination

Тут мы наследуемся от класса PageNumberPagination, который содержит в себе стандартные поля для настройки пагинации.

В результате API запрос в браузере выглядит следующим образом:

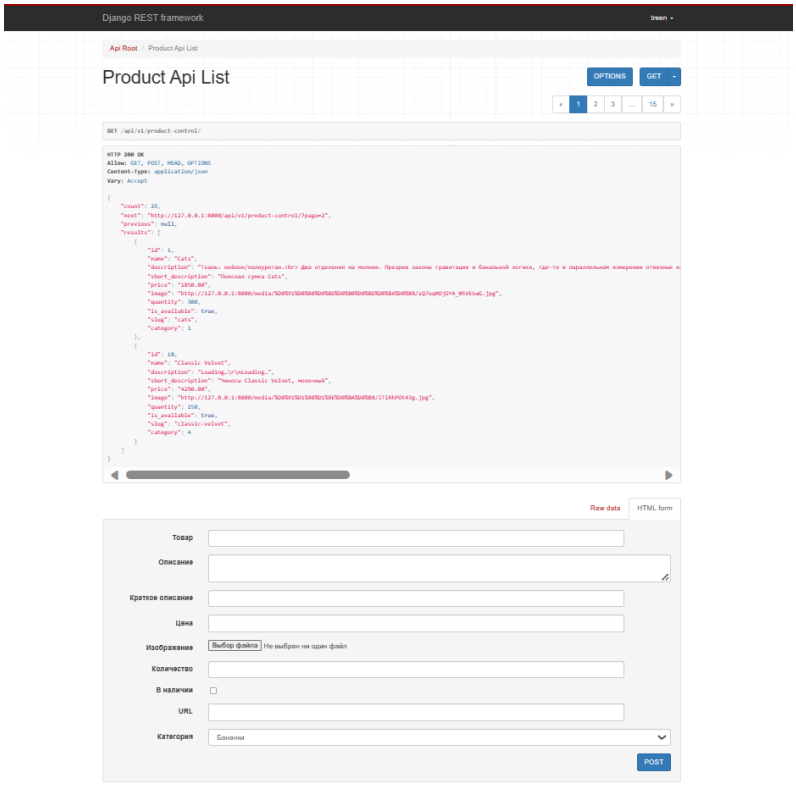

Рис.34. Страница продуктов. REST Framework. Вход с аккаунта администратора

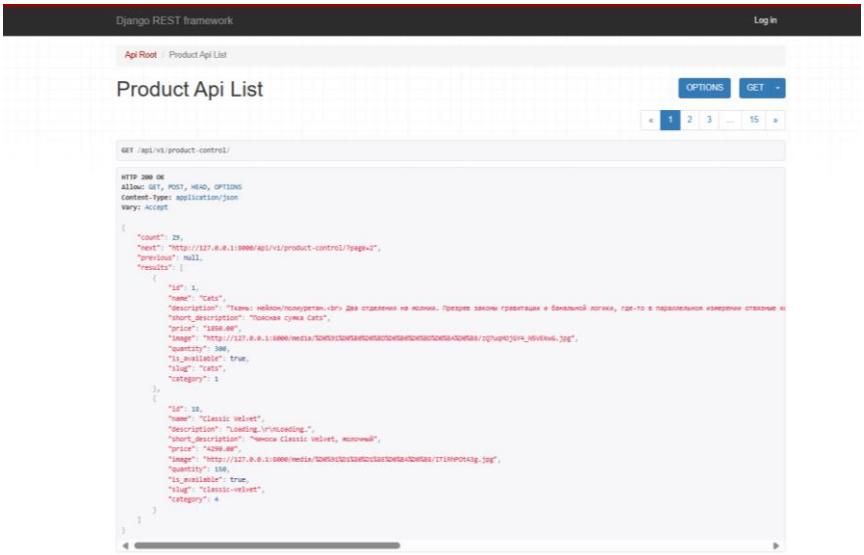

Рис.35. Страница продуктов. REST Framework. Вход без авторизации

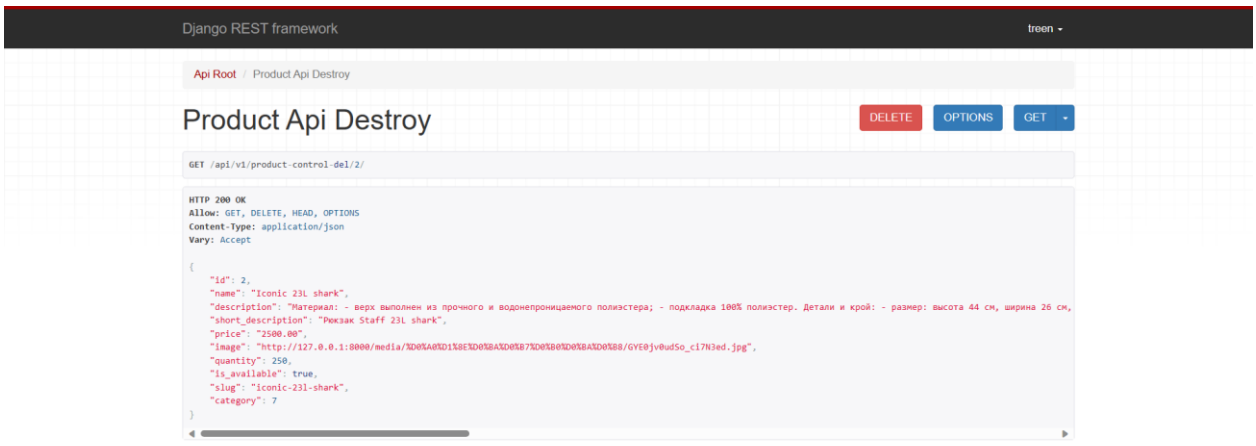

Рис.35. Страница удаления продуктов. REST Framework.

В данном случае приведу примеры работы данных представлений. Далее они показаны не будут, т.к. логика остаётся той же, отличие только в реализации.

Разберём реализацию авторизации и исполнения запросов по средствам токенов. Для демонстрации работы воспользуюсь программой «Postman».

Для начала войдём в аккаунт с помощью токена. Для этого отправим запрос на url-адрес «http://127.0.0.1:8000/auth/token/login». В ответ получаем токен для входа.

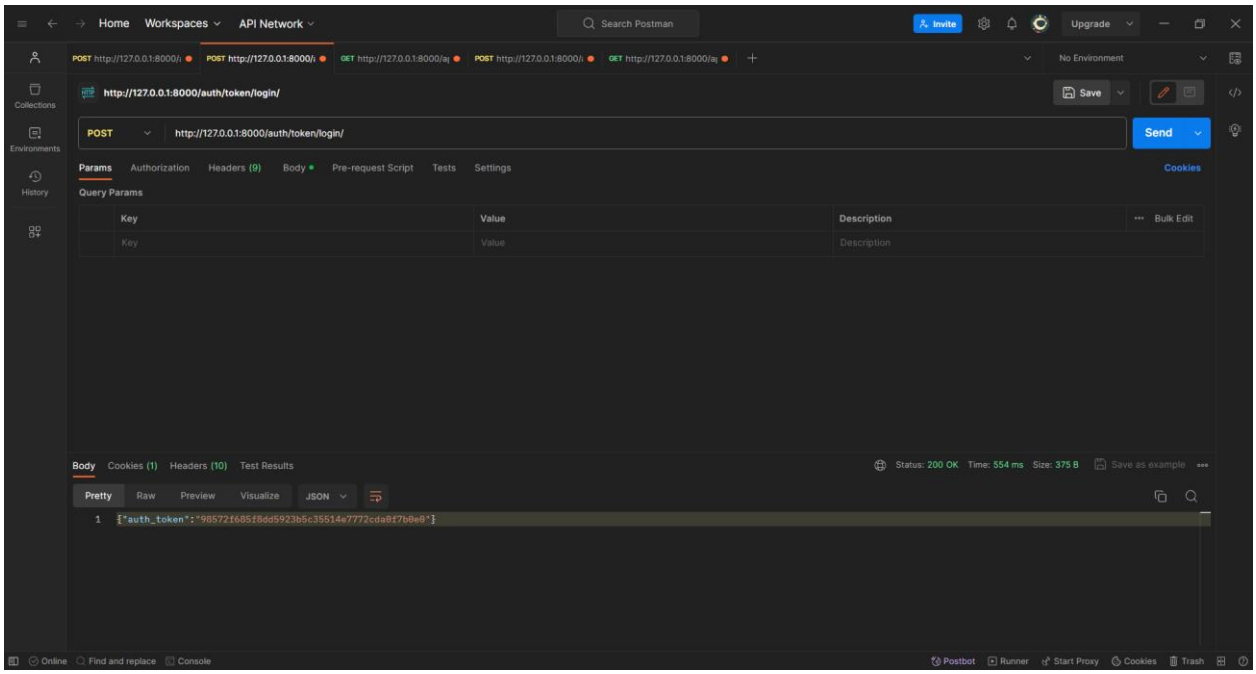

Рис.36. Авторизация по токену. Получение токена.

Далее введём его в поле «Authorization»:

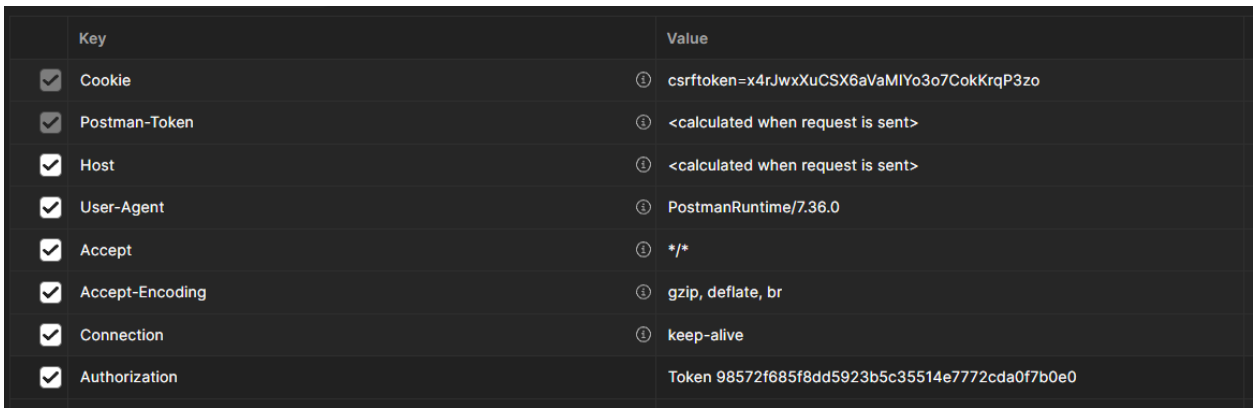

Рис.37. Авторизация по токену. Введение токена.

После чего отправим POST-запрос по url-адресу «http://127.0.0.1:8000/api/genres/». В ответ получаем данные о добавленном продуктах в виде JSON-ответа (см. рис. №38).

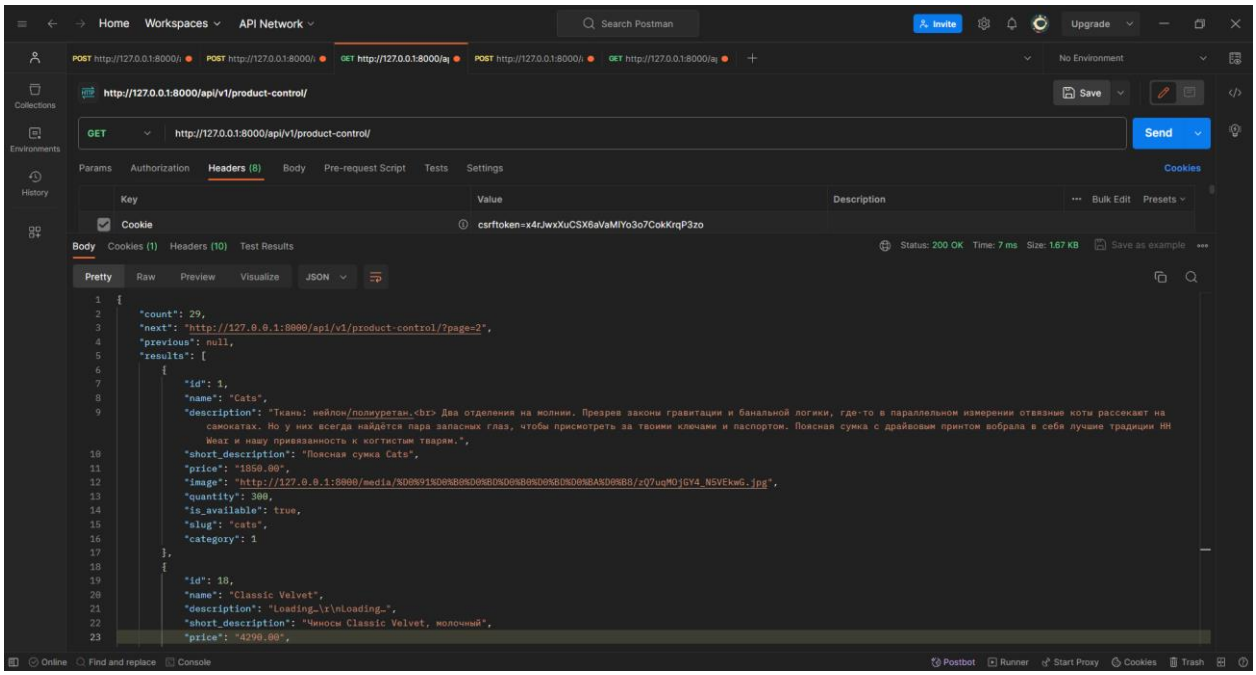

Рис.38. Отправка GET-запроса для получения продуктов.

В представлениях, предназначенных для работы с категориями через API всё практически идентично, отличие состоит лишь в том, какие ограничения доступа применяются, и от каких базовый классов мы наследуем представления.

Так же хочется отметить, что можно реализовывать кастомные модификаторы доступа. В примере ниже мы определяем модификатор доступа IsAdminOrReadOnly, который разрешает не авторизовавшимся пользователям только доступ ко всем “безопасным” методам(“GET”…), а админу даёт полный контроль

```python
class IsAdminOrReadOnly(permissions.BasePermission):
    def has_object_permission(self, request, view, ob):
        if request.method in permissions.SAFE_METHODS:
            return True

        return bool(request.user and request.user.is_staff)
```

Рис.39. Кастомный permission

Рассмотрим самый функциональный API класс, который дает доступ ко всем запросам в рамках одного представления – это ViewSet. Его код представлен ниже:

```
class ProductViewSet(viewsets.ModelViewSet):
    serializer_class = ProductSerializer
    queryset = Product.objects.all()
    permission_classes = (IsAdminUser,)
    pagination_class = ProductAPIListPagination


    def get_queryset(self):
        pk = self.kwargs.get('pk')
        if not pk:
            return Product.objects.all()[:15]
        return Product.objects.all()

    @action(methods=['get'], detail = True)
    def category(self, request, pk=None):
        category = ProductCategory.objects.get(pk=pk)
        return Response({'category': category.name})
```

Рис.40. Viewset модели product

В нем мы указываем те же поля, что и указывали в ViewCreateAPIList, но тут мы переопределяем метод get_queryset для того, чтобы выводить 15 записей, а не целый список.

Здесь так же указана функция, которая будет реализовывать взаимодействие с другой моделью БД (category). Для того, чтобы указать что эта функция должна представлять функционал API с GET-запросом мы прописываем соответствующий декоратор. А в теле функции указываем выборку из нужных объектов.

Для того, чтобы эта функция, а также само представления viewset работало по определённому адресу её нужно зарегистрировать в router:

```
router = routers.DefaultRouter()
router.register(r'product', ProductViewSet, basename='product')
```

Рис.41. Viewset регистрация

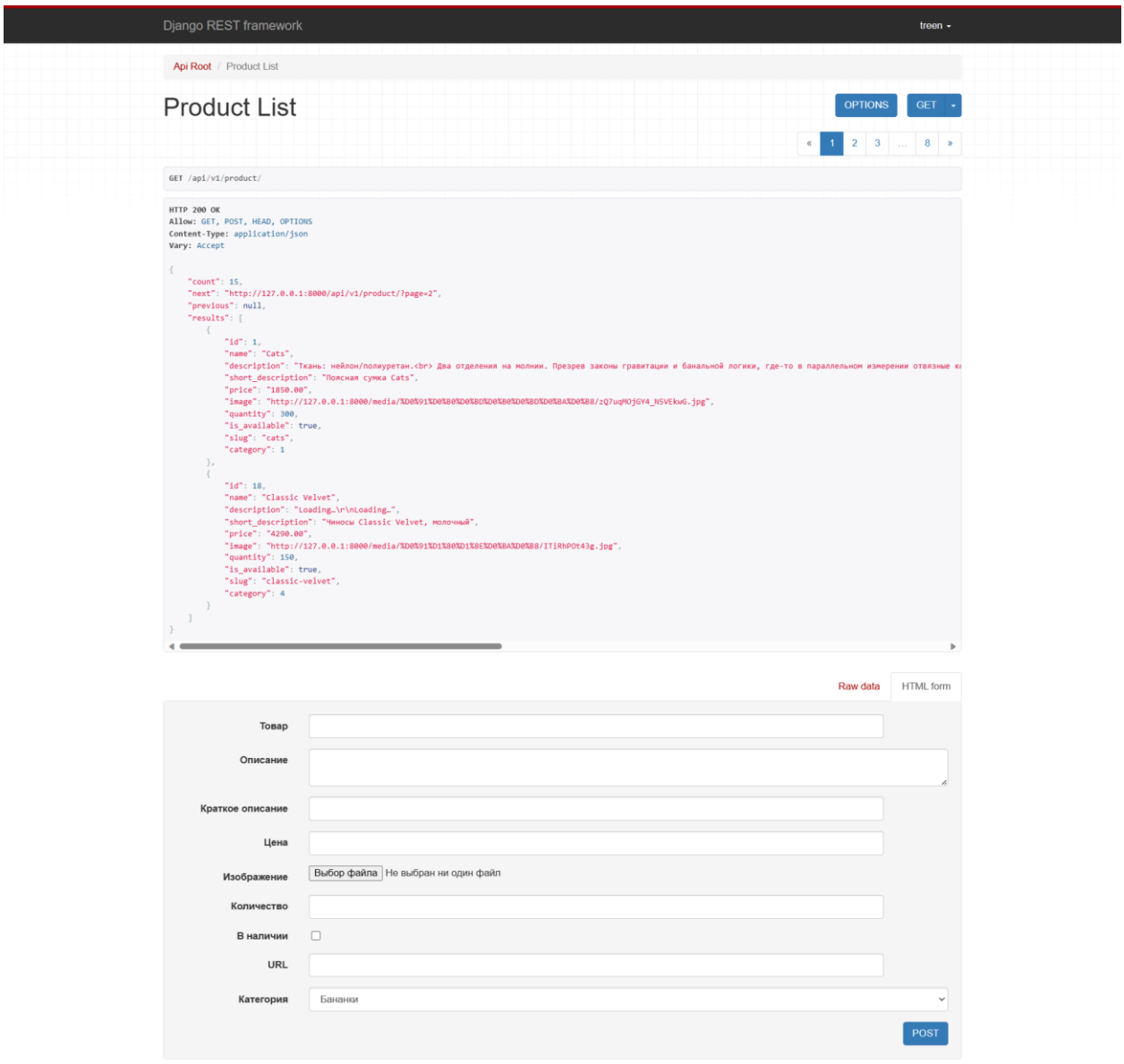

Рис.42. Viewset API интерфейс POST, GET

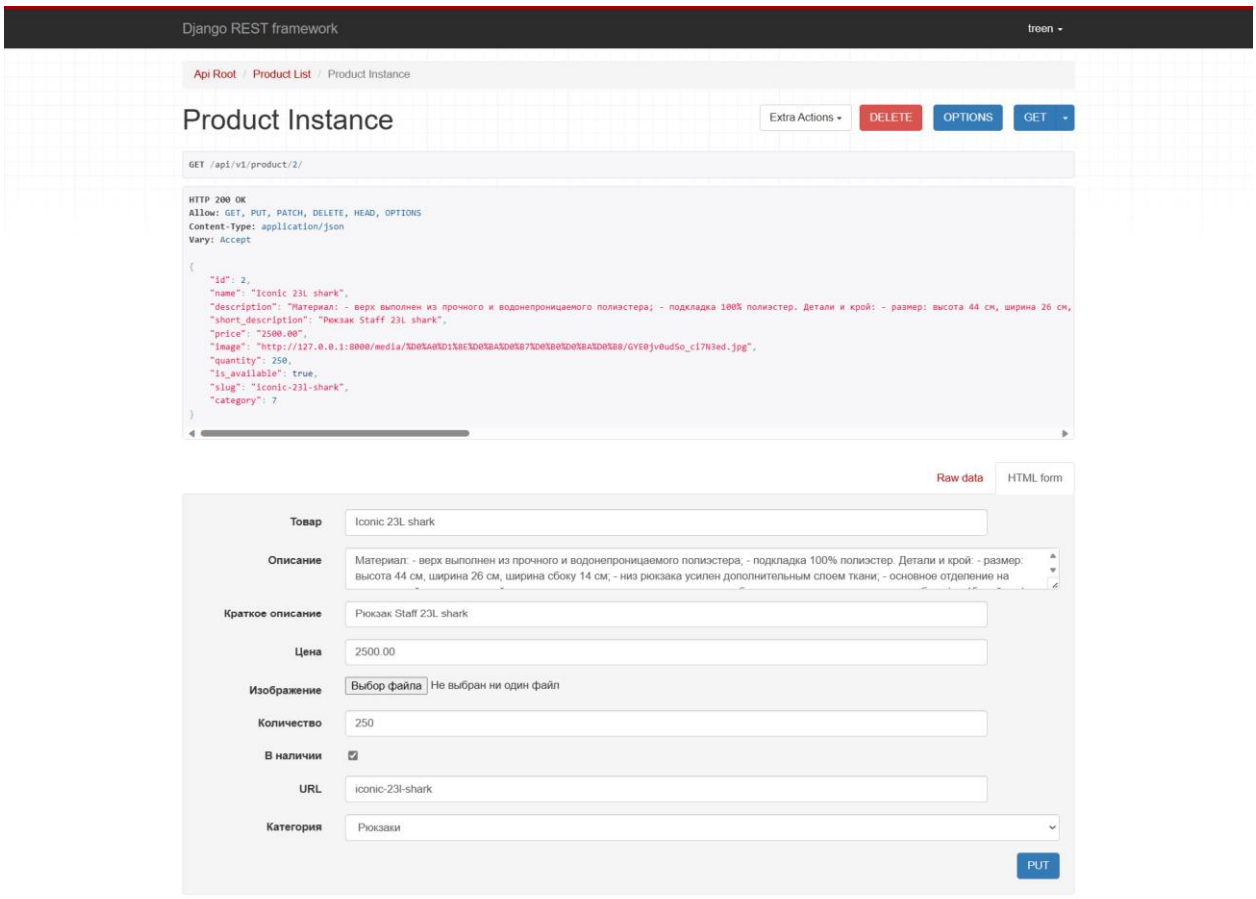

Рис.43. Viewset API интерфейс UPDATE\PATCH, DELETE

Теперь реализуем авторизацию с помощью cookies и с помощью токенов.

Начнём с реализации авторизации по «кукам». Для её реализации необходимо всего лишь дописать следующую строку кода в наш файл «urls.py»: «path('api/drf-auth/', include('rest_framework.urls'))».

Таким образом, перейдя по адресу «http://127.0.0.1:8000/api/drf-auth/login/» попадаем на страницу входа в аккаунт, которая выглядит следующим образом:

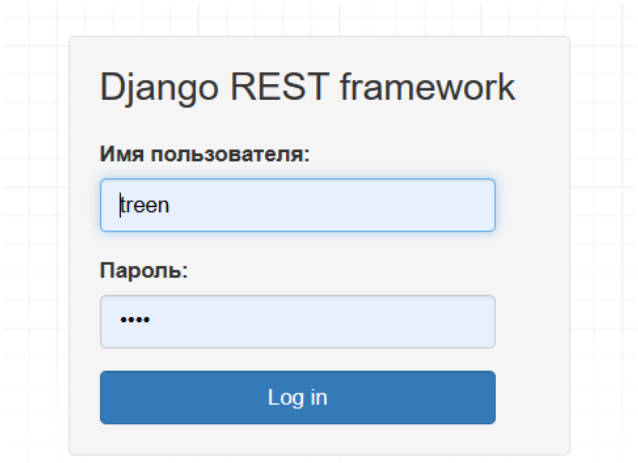

Рис.44. Страница авторизации по кукам.

Для выхода из аккаунта достаточно нажать кнопку «Log out» на любой из страниц, предназначенной для работы с API.

Теперь реализуем авторизацию с помощью токенов, которую мы использовали для входа при тестировании get и post запросов.

Для этого нам потребуется библиотека Djoser, устанавливаем её регистрируем её в нашем приложении и приступаем к реализации. Для реализации допишем новые пути в наш файл «urls.py», а также добавим возможность авторизации по токенам в нашем приложении.

```
path('api/v1/drf-auth/', include('rest_framework.urls')),
path('api/v1/auth/', include('djoser.urls')),
re_path(r'^auth/', include('djoser.urls.authtoken')),
```

Рис.45. Пути, добавленные для работы с Djoser.

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 5,


    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
        'rest_framework.renderers.BrowsableAPIRenderer',
    ],

    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ],

    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ),
}
SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": timedelta(minutes=5),
    "REFRESH_TOKEN_LIFETIME": timedelta(days=1),
    "ROTATE_REFRESH_TOKENS": False,
    "BLACKLIST_AFTER_ROTATION": False,
    "UPDATE_LAST_LOGIN": False,

    "ALGORITHM": "HS256",
    "SIGNING_KEY": SECRET_KEY,
    "VERIFYING_KEY": "",
    "AUDIENCE": None,
    "ISSUER": None,
    "JSON_ENCODER": None,
    "JWK_URL": None,
    "LEEWAY": 0,

    "AUTH_HEADER_TYPES": ("JWT",),
    "AUTH_HEADER_NAME": "HTTP_AUTHORIZATION",
    "USER_ID_FIELD": "id",
    "USER_ID_CLAIM": "user_id",
    "USER_AUTHENTICATION_RULE": "rest_framework_simplejwt.authentication.default_user_authentication_rule",

    "AUTH_TOKEN_CLASSES": ("rest_framework_simplejwt.tokens.AccessToken",),
    "TOKEN_TYPE_CLAIM": "token_type",
    "TOKEN_USER_CLASS": "rest_framework_simplejwt.models.TokenUser",

    "JTI_CLAIM": "jti",

    "SLIDING_TOKEN_REFRESH_EXP_CLAIM": "refresh_exp",
    "SLIDING_TOKEN_LIFETIME": timedelta(minutes=5),
    "SLIDING_TOKEN_REFRESH_LIFETIME": timedelta(days=1),

    "TOKEN_OBTAIN_SERIALIZER": "rest_framework_simplejwt.serializers.TokenObtainPairSerializer",
    "TOKEN_REFRESH_SERIALIZER": "rest_framework_simplejwt.serializers.TokenRefreshSerializer",
    "TOKEN_VERIFY_SERIALIZER": "rest_framework_simplejwt.serializers.TokenVerifySerializer",
    "TOKEN_BLACKLIST_SERIALIZER": "rest_framework_simplejwt.serializers.TokenBlacklistSerializer",
    "SLIDING_TOKEN_OBTAIN_SERIALIZER": "rest_framework_simplejwt.serializers.TokenObtainSlidingSerializer",
    "SLIDING_TOKEN_REFRESH_SERIALIZER": "rest_framework_simplejwt.serializers.TokenRefreshSlidingSerializer",
}
```

Рис.46. Разрешение аутентификации с помощью токенов.

Пример авторизации с помощью токенов был показан на рисунках. Сейчас лишь посмотрим на то, как выглядят сайты, соответствующие добавленным маршрутам.

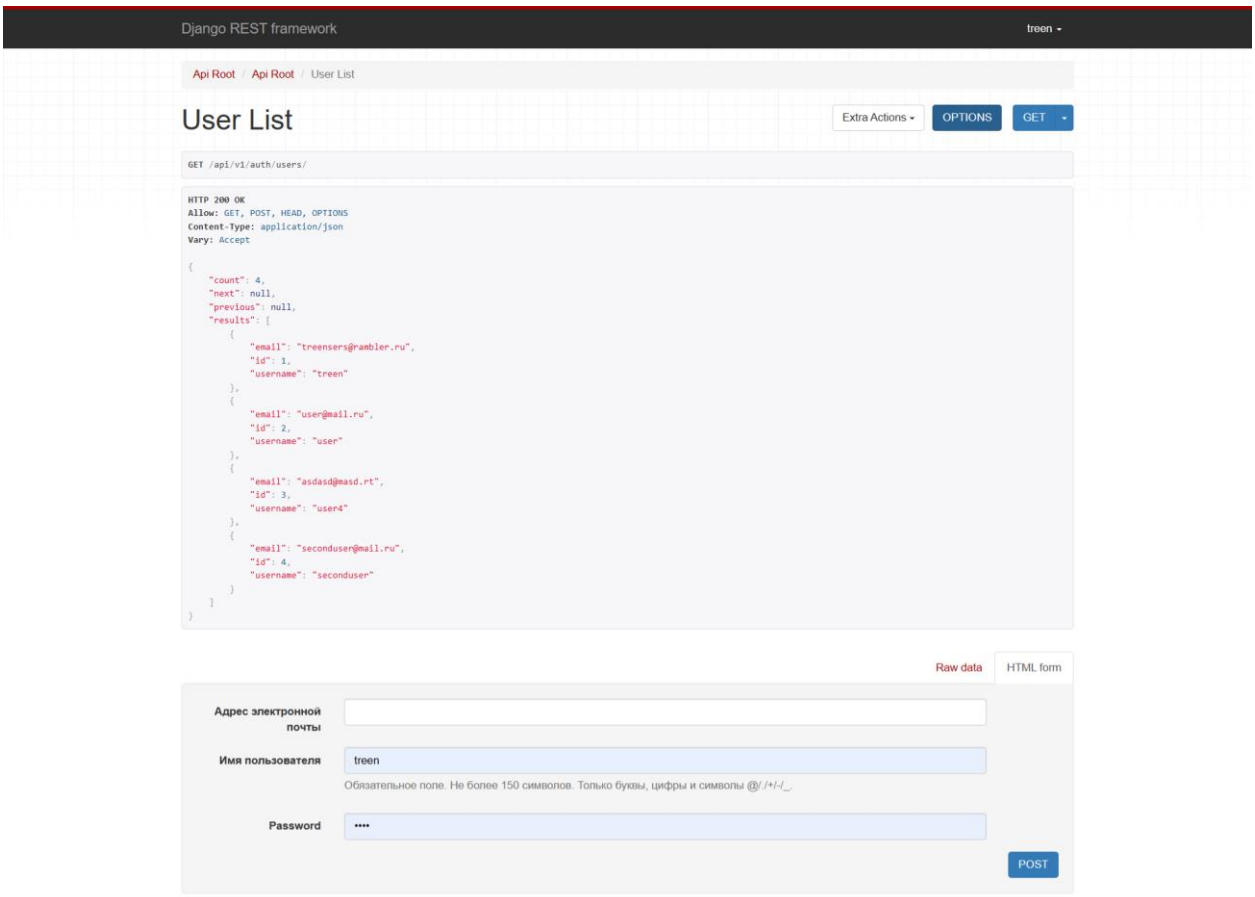

Рис.69. Список пользователей и форма для получения токена для входа.

Таким образом, разработка приложения окончена.

6. Вывод по выполненной работе

Таким образом, в ходе выполнения курсовой работы по дисциплине «Разработка Web-приложений» было реализовано веб-приложение «магазин одежды». Для этого были изучены и использованы Django, Django REST Framework, API. По итогу мы получили почти готовое веб-приложение, которое при должной доработке может грамотно функционировать.

Были реализованы следующие ключевые функциональности:

• Создание основного приложения, включающего в себя модели, формы, представления, URL-маршруты и шаблоны.

• Определение моделей данных, таких как Product, ProductCategory для хранения информации о товарах

• Создание форм, примеру таких как AddProductForm для добавления новых продуктов и RegisterUserForm для регистрации новых пользователей.

• Реализация представлений для отображения списка продуктов, отдельных страниц, а также для добавления и редактирования продуктов.

• Определение URL-маршрутов для соответствия различным представлениям и функциональностям приложения.

• Создание шаблонов HTML для отображения страниц, включая домашнюю страницу, страницы «Каталог», «Каталог по категориям», страницы добавления товаров, страницы отдельных товаров.

• Использование фильтров для динамического отображения товаров по имени, категории и сортировки их по цене.

• Использование встроенной административной панели Django для управления базой данных.

• Применение механизмов безопасности Django, таких как проверка аутентификации пользователей.

Таким образом, выполнение данной курсовой работы дало базовые знания о Django в области back-end разработки, которые при желании можно использовать для развития своих навыков.