Geekbrains

**Разработка серверной части веб-приложения, используя Django фреймворк на примере интернет-магазина компьютерных комплектующих “UPGrade PC”**

Программа:

Программист Python Цифровые профессии

Ерёменко Роман Юрьевич

г. Прохладный

2023г.

**Содержание**

1. Введение:

1.1 Вступление

1.2 Цель и задачи проекта

1.3 Обзор технологий и инструментов

2. Обзор Django фреймворка:

2.1 История и основные принципы Django

2.2 Архитектура и основные компоненты Django

2.3 Преимущества и возможности использования Django для web-разработки

3. Анализ требований к интернет-магазину "UPGrade PC":

3.1 Описание функциональных и нефункциональных требований

3.1.1 Функциональные требования

3.1.2 Нефункциональные требования

4. Проектирование базы данных:

4.1 Определение сущностей и их атрибутов

4.2 Разработка схемы базы данных

4.3 Описание связей между таблицами

5. Разработка серверной части веб-приложения:

5.1 Создание моделей Django для представления данных

5.2 Реализация логики проекта с использованием Django views и templates

5.3 Интеграция с базой данных и обработка запросов

6. Реализация функциональности интернет-магазина "UPGrade PC":

6.1 Регистрация и аутентификация пользователей

6.2 Верификация электронной почты

6.3 Добавление товаров в корзину и оформление заказа

6.4 Управление заказами и администрирование магазина

7. Тестирование и отладка:

7.1 Тестирование функциональности

7.2 Линтеры и форматеры

8. Заключение:

8.1 Выводы по результатам разработки серверной части веб-приложения

8.2 Дальнейшие перспективы развития и улучшения интернет-магазина "UPGrade PC"

9. Список использованных источников:

9.1 Официальная документация и руководства

9.2 Интернет-ресурсы и статьи

10. Приложения:

10.1 Код серверной части веб-приложения на Django

10.1.1 Модели Django для представления данных:

# 10.1.2 Реализация логики проекта с использованием Django views и templates

10.2 Снимки экрана интерфейса интернет-магазина "UPGrade PC"

10.3 Дополнительные материалы и данные

**Введение**

**1.1 Вступление.**

В современном мире электронной коммерции создание эффективного и надежного интернет-магазина становится ключевой задачей для предпринимателей. Данный дипломный проект посвящен разработке серверной части веб-приложения с использованием Django фреймворка на примере интернет-магазина компьютерных комплектующих "UPGrade PC".

Целью данной работы является создание полнофункциональной backend части интернет-магазина, способной эффективно управлять товарами, заказами и пользователями. Для достижения этой цели был выбран Django, популярный и мощный фреймворк для быстрой и надежной разработки веб-приложений.

Работа будет охватывать анализ требований к интернет-магазину "UPGrade PC", проектирование базы данных, разработку функциональности, тестирование, развёртывание проекта и профилирование.

Этот документ представляет собой описание шагов, предпринятых для создания серверной части веб-приложения, и вклад в область веб-разработки с использованием современных технологий. В ходе работы над проектом были изучены принципы построения эффективной и масштабируемой backend системы для интернет-магазина, что представляет актуальный интерес в контексте современного онлайн-бизнеса.

Данный дипломный проект представляет собой результат работы над созданием веб-приложения, способного удовлетворить требования современного рынка электронной коммерции.

**1.2 Цель и задачи проекта.**

Цель:

Целью данного дипломного проекта является создание функциональной и масштабируемой серверной части веб-приложения для интернет-магазина "UPGrade PC" с использованием Django фреймворка. Основной целью является предоставление надежной платформы для продажи компьютерных комплектующих, обеспечивающей удобство в использовании как для клиентов, так и для администраторов магазина.

Задачи:

1. Анализ требований:

- Провести детальный анализ функциональных и нефункциональных требований к интернет-магазину "UPGrade PC".

2. Проектирование базы данных:

- Определить структуру базы данных, соответствующую потребностям магазина.

- Разработать схему базы данных, обеспечивающую эффективное хранение и доступ к информации о товарах, пользователях и заказах.

3. Разработка функциональности:

- Создать модели данных с использованием Django ORM для представления информации о продуктах, пользователях и заказах.

- Реализовать логику приложения для управления товарами, процессом заказа и аутентификации пользователей.

4. Тестирование и отладка:

- Провести тестирование функциональности и проверку на соответствие требованиям.

- Отладить систему, исправить ошибки и обеспечить стабильную работу приложения.

5. Дальнейшее развитие:

- Определить потенциальные области улучшения и развития интернет-магазина.

- Предложить идеи по расширению функциональности и улучшению пользовательского опыта.

**1.3 Обзор технологий и инструментов.**

В данном разделе представлен обзор использованных технологий и инструментов для разработки веб-приложения "UPGrade PC".

Frontend:

1. HTML5 и CSS3:

- HTML5 используется для создания структуры веб-страниц, обеспечивающей семантичность и доступность контента.

- CSS3 применяется для оформления и стилизации веб-страниц, обеспечивая удобство использования и привлекательный дизайн.

2. JavaScript и jQuery:

- JavaScript используется для добавления интерактивности и динамической функциональности на веб-страницы.

- jQuery используется для упрощения манипуляции с DOM-элементами и создания анимаций.

3. Bootstrap:

- Bootstrap — это фреймворк с отзывчивым дизайном, который обеспечивает быструю и простую разработку адаптивных веб-интерфейсов.

Backend:

1. Django Framework:

Django — это мощный фреймворк для разработки веб-приложений на языке Python. Он обеспечивает эффективное управление запросами, обработку данных и управление базой данных.

2. Python:

Python — это язык программирования, который обладает простым синтаксисом и широкими возможностями для разработки веб-приложений.

3. PostgreSQL:

PostgreSQL — это реляционная база данных с открытым кодом, которая поддерживается в течение 30 лет разработки и является одной из наиболее известных среди всех существующих реляционных баз данных.

4. Redis:

Redis (REmote DIctionary Server, «удалённый серверный словарь») — это нереляционная резидентная СУБД, хранящая данные в виде пар «ключ-значение».

5. Celery:

Celery это простая, гибкая и надежная распределенная система для обработки большого количества сообщений, предоставляющая операции с инструментами, необходимыми для обслуживания такой системы. Очереди используются как механизм для распределения работы по потокам или процессам. Celery использует брокера для посредничества между клиентами и воркерами.

6. OAuth 2.0:

OAuth 2.0 — протокол авторизации, позволяющий выдать одному сервису (приложению) права на доступ к ресурсам пользователя на другом сервисе.

7. Nginx:

Nginx — это программное обеспечение с открытым исходным кодом, которое позволяет создавать веб-сервер. Также его используют как почтовый сервер или обратный прокси-сервер. Веб-сервер — это программное обеспечение, основная задача которого — обработка входящих и исходящих запросов на различных протоколах передачи данных.

8. Gunicorn:

Gunicorn — это Application-сервер для запуска Web-приложений написанных на Python. Основная его задача — это работа в режиме демона и поддержка постоянной работы Web-приложений.

**2. Обзор Django фреймворка**

***2.1 История и основные принципы Django.***

Django — это популярный веб-фреймворк, разработанный для создания мощных и масштабируемых веб-приложений. Он был создан в 2003 году в Lawrence Journal-World, местной газете в США. Изначально Django разрабатывался для упрощения процесса создания веб-приложений, обладал интуитивным интерфейсом и хорошо структурированным кодом.

Основными принципами, лежащими в основе Django, являются:

1. Модель-представление-контроллер (MVC): Django следует архитектуре MVC, которая разделяет компоненты веб-приложения на модели (данные), представления (визуализация) и контроллеры (логика обработки запросов). Это помогает разработчикам создавать чистый и модульный код.

2. ORM (Object-Relational Mapping): Django предоставляет ORM, который позволяет работать с базами данных в объектно-ориентированном стиле. Это упрощает процесс работы с данными и уменьшает необходимость вручную писать SQL-запросы.

3. URL-маршрутизация: Django использует файл urls.py для маршрутизации запросов от клиента к соответствующим представлениям. Это позволяет легко определять, какой контроллер должен обрабатывать каждый конкретный URL.

4. Шаблонизация: Django поддерживает использование шаблонов, которые позволяют разделять логику приложения и представления. Шаблоны позволяют разработчикам создавать динамические веб-страницы, в которых данные могут быть вставлены и отображены с помощью специальных тегов и фильтров.

5. Автоматическая административная панель: Django предоставляет встроенную административную панель, которая позволяет управлять данными в базе данных, без необходимости вручную писать код. Это очень удобно для управления контентом и администрирования веб-приложений.

6. Безопасность: Django обеспечивает встроенные механизмы безопасности, такие как защита от CSRF-атак (межсайтовая подделка запроса) и инструменты для обработки аутентификации и авторизации пользователей.

7. Международная поддержка: Django обладает мощными функциями для локализации и интернационализации веб-приложений. Он предоставляет инструменты для перевода текста на разные языки и поддержку разных форматов даты и времени.

И это только малая часть возможностей, которые предоставляет Django. Фреймворк активно развивается и имеет большое сообщество разработчиков, что делает его надежным и поддерживаемым инструментом для создания веб-приложений.

**2.2 Архитектура и основные компоненты Django.**

Архитектура Django состоит из четырех основных компонентов:

1) *Модели данных* являются сердцевиной любого современного веб-приложения. Модель — это важнейшая часть приложения, которое постоянно обращается к данным при любом запросе из любой сессии. Любая модель является стандартным Python классом. Объектно-ориентированный mapper (ORM) обеспечивает таким классам доступ непосредственно к базам данных. Если бы не было ORM, пришлось бы писать запросы непосредственно на SQL. Модель обеспечивает облегченный механизм доступа к слою данных, инкапсулирует бизнес-логику. Модель не зависит от конкретного приложения. Данными можно манипулировать даже из командной строки, не используя при этом веб-сервер;

2) *Представления* выполняют разнообразные функции, в том числе контролируют запросы пользователя, выдают контекст в зависимости от его роли. View — это обычная функция, которая вызывается в ответ на запрос какого-то адреса (URL) и возвращает контекст;

3) *Шаблоны* являются формой представления данных. Шаблоны имеют свой собственный простой метаязык и являются одним из основных средств вывода на экран;

4) *URL* механизм внешнего доступа к представлениям (view). Встроенные в URL регулярные выражения делают механизм достаточно гибким. При этом одно представление может быть сконфигурировано к нескольким URL, предоставляя доступ различным приложениям. Здесь поддерживается философия закладок: URL становятся самодостаточными и начинают жить независимо от представления.

**2.3 Преимущества и возможности использования Django для web-разработки.**

Вот несколько преимуществ и возможностей, которые он предоставляет:

1. Простота использования: Django имеет простой и понятный синтаксис, что делает разработку быстрой и эффективной. Это особенно полезно для начинающих разработчиков.

2. Мощный ORM: Django поставляется с ORM (Object-Relational Mapping), который облегчает работу с базами данных. Он позволяет вам взаимодействовать с базой данных с использованием объектно-ориентированного подхода, что делает код более читаемым и поддерживаемым.

3. Гибкость: Django предоставляет множество готовых компонентов и приложений, что значительно упрощает разработку. Вы можете использовать эти компоненты по мере необходимости и создавать свои собственные приложения, чтобы адаптировать Django под ваши потребности.

4. Безопасность: Django имеет встроенные механизмы безопасности, которые помогают защитить ваше приложение от атак. Он предоставляет защиту от CSRF-атак, предотвращает инъекции SQL и обеспечивает безопасную аутентификацию и авторизацию.

5. Масштабируемость: Django разработан с учетом масштабируемости. Он поддерживает горизонтальное масштабирование и легко масштабируется на платформы cloud-серверов.

6. Административный интерфейс: Django поставляется с готовым административным интерфейсом. Он считывает метаданные из ваших моделей, чтобы обеспечить быстрый, ориентированный на модели интерфейс, где доверенные пользователи могут управлять контентом на вашем сайте.

**3. Анализ требований к интернет-магазину "UPGrade PC":**

***3.1 Описание функциональных и нефункциональных требований***

*3.1.1 Функциональные требования*

1. Регистрация и Аутентификация:

- Пользователи могут создавать учетные записи в системе.

- Система должна обеспечивать безопасный процесс аутентификации.

- Авторизованные пользователи имеют доступ к персонализированным функциям.

2. Каталог товаров:

- Просмотр списка компьютерных комплектующих с детальной информацией о каждом товаре.

- Фильтрация товаров по категориям.

- Возможность добавления товаров в корзину напрямую из каталога.

3. Корзина и Оформление заказа:

- Добавление товаров в корзину и управление ею.

- Просмотр общей стоимости заказа перед оформлением.

- Оформление заказа с указанием адреса доставки и выбором способа оплаты.

4. Личный кабинет пользователя:

- Просмотр и редактирование личной информации.

- Просмотр истории заказов и их статусов.

- Просмотр корзины и возможность сделать заказ.

5. Административная панель:

- Работа с товарами, категориями и пользователями.

- Мониторинг заказов и обработка запросов от клиентов.

- Управление содержимым сайта.

*3.1.2 Нефункциональные требования*

1. Производительность:

- Система должна обеспечивать отзывчивость при большом количестве пользователей и товаров.

- Время загрузки страниц должно быть минимальным.

2. Безопасность:

- Защита от SQL-инъекций, XSS-атак и других видов веб-атак.

- Использование протокола HTTPS для шифрования данных.

3. Масштабируемость:

- Возможность горизонтального масштабирования серверной части при увеличении нагрузки.

4. Совместимость:

- Поддержка различных браузеров (Chrome, Firefox, Safari, Edge) и устройств (планшеты, мобильные телефоны).

5. Доступность:

- Обеспечение доступности сервиса 24/7 с минимальным временем простоя.

- Резервное копирование данных для предотвращения потери информации.

6. Удобство использования:

- Интуитивный интерфейс для удобства навигации и покупок.

- Поддержка уведомлений и подсказок для пользователя.

7. Язык и технологии:

- Разработка с использованием Django фреймворка с версией 4.2.7.

- Использование языка программирования Python для серверной части.

8. Требования к базе данных:

- Использование реляционной базы данных (например, PostgreSQL) для хранения информации о пользователях, заказах и товарах.

9. Интеграция с платежными системами:

- Поддержка популярных платежных систем для обеспечения удобства клиентов при оплате заказов.

10. Документация:

- Наличие подробной документации по развертыванию, настройке и обслуживанию системы.

**4. Проектирование базы данных:**

***4.1 Определение сущностей и их атрибутов***

При разработке серверной части веб-приложения "UPGrade PC" на основе Django фреймворка, были определены следующие сущности и их атрибуты:

1.Пользователь (User)

- Идентификатор (id): целое число, автоматическое создание, первичный ключ.

- Имя (first\_name): строка, обязательное поле

- Фамилия (last\_name): строка, обязательное поле

- Электронная почта (email): строка, уникальное обязательное поле

- Подтвержденная электронная почта (is\_verified\_email) поле «истина/ложь»

- Пароль (password): строка, обязательное поле

- Дата регистрации (date\_joined): дата и время, автоматически генерируемое поле

- Изображение (image): изображение профиля пользователя

2.Товар (Product)

- Идентификатор (id): целое число, автоматическое создание, первичный ключ.

- Название (name): строка, уникальное обязательное поле

- Описание (description): текстовое поле

- Спецификации (specifications): текстовое поле

- Цена (price): десятичное число, обязательное поле

- Категория (category): внешний ключ к сущности "Категория"

- Количество на складе (quantity): целое число, обязательное поле

- Наличие на складе (is\_available): поле «истина/ложь»

- Изображение (image): ссылка на изображение товара

3.Категория (ProductCategory)

- Идентификатор (id): целое число, автоматическое создание, первичный ключ.

- Название (name): строка, уникальное обязательное поле

- Описание (description): текстовое поле

- Наличие на складе (is\_available): поле «истина/ложь»

4. Заказ (Order)

- Идентификатор (id): целое число, автоматическое создание, первичный ключ.

- Имя (first\_name): строка, обязательное поле

- Фамилия (last\_name): строка, обязательное поле

- Электронная почта (email): строка, уникальное обязательное поле

- Адрес доставки (address): строка, обязательное поле

- Заказанные товары (basket\_history): строка, обязательное поле

- Пользователь (initiator): внешний ключ к сущности "Пользователь"

- Дата создания (created): дата и время, автоматически генерируемое поле

- Статус (status): строка, например: “создан”, “оплачен”, “в пути”, “доставлен”

5. Корзина (Basket)

- Идентификатор (id): целое число, автоматическое создание, первичный ключ.

- Количество (quantity): целое число, обязательное поле

- Дата создания (created\_timestamp): дата и время, автоматически генерируемое поле

- Пользователь (user): внешний ключ к сущности "Пользователь"

- Товар (product): внешний ключ к сущности "Товар"

6. Верификация электронной почты (EmailVerification)

- Идентификатор (id): целое число, автоматическое создание, первичный ключ

- Код подтверждения (code): строка, уникальное обязательное поле

- Дата создания (created): дата и время, автоматически генерируемое поле

- Срок действия (expiration): дата и время, автоматически генерируемое поле

- Пользователь (user): внешний ключ к сущности "Пользователь"

Эти сущности и их атрибуты обеспечат необходимую функциональность для работы интернет-магазина компьютерных комплектующих "UPGrade PC".

***4.2 Разработка схемы базы данных***

Схема базы данных разработана с помощью онлайн-инструмента для разработки и моделирования схем баз данных и представлена в приложении №1.

***4.3 Описание связей между таблицами***

1. Связь между таблицей "User" (Пользователь) и таблицей "Order" (Заказ):

- Поле "initiator" в таблице " Order " является внешним ключом, связывающим каждый заказ с соответствующим пользователем из таблицы "User". Таким образом, каждый заказ принадлежит определенному пользователю.

2. Связь между таблицей "User" (Пользователь) и таблицей "EmailVerification" (Верификация по email):

- Поле "user" в таблице " EmailVerification " является внешним ключом, связывающим код верификации с соответствующим пользователем из таблицы "User". Таким образом, каждый уникальный код верификации принадлежит определенному пользователю.

3. Связь между таблицей "User" (Пользователь) и таблицей "Basket" (Корзина):

- Поле "user" в таблице " Basket " является внешним ключом, связывающим каждую корзину, включающую товар и его количество с соответствующим пользователем из таблицы "User". Таким образом, каждая корзина принадлежит определенному пользователю.

4. Связь между таблицей "Product" (Товар) и таблицей "Basket" (Корзина):

- Поле "product" в таблице " Basket " является внешним ключом, связывающим каждую корзину, включающую пользователя с соответствующим товаром из таблицы " Product ". Таким образом, в каждой корзине находится определенный товар.

5. Связь между таблицей "ProductCategory" (Категория товара) и таблицей "Product" (Товар):

- Поле "category" в таблице " Product " является внешним ключом, связывающим каждый товар с соответствующей категорией товаров из таблицы " ProductCategory ".

**5. Разработка серверной части веб-приложения:**

***5.1 Создание моделей Django для представления данных***

Для начала рассмотрим, что же такое Django ORM.

Согласно Википедии, ORM (Object-Relational Mapping, рус. объектно-реляционное отображение) — это **технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования**, создавая «виртуальную объектную базу данных». Более простым языком — это механизм, который позволит нам описывать таблицы базы данных с помощью объектов.

Вообще, ORM Django имеет очень богатый функционал и вам вряд ли понадобится переходить на уровень SQL-запросов, все можно сделать на уровне этого API. Кроме того, начинающим разработчикам уровень ORM позволит писать грамотные и оптимизированные запросы к БД любого типа: SQLite, MySQL, PostgreSQL, Oracle, то есть, приложение будет совершенно независимым от типа СУБД. Все это и привело к тому, что сейчас, в основном, используются различные ORM при работе с таблицами БД.

Сначала мы создаем образец базы данных, используя модель Django, состоящую из некоторых данных, и затем мы сможем выполнить интересующий нас запрос в этой базе данных.

Используемые при создании моделей типы полей:

* CharField - используется для определения строк фиксированной длины от короткой до средней.
* TextField – большое текстовое поле.
* DecimalField - Десятичное число с фиксированной точностью, представленное в Python экземпляром Decimal.
* BigAutoField - 64-битное целое число, которое автоматически увеличивается в соответствии с доступными идентификаторами, в него гарантированно помещаются числа от 1 до 9223372036854775807.
* DateTimeField - дата и время.
* BooleanField - поле «истина/ложь».
* EmailField - поле CharField, которое проверяет, является ли значение действительным адресом электронной почты.
* ImageField - наследует все атрибуты и методы от FileField, но при этом проверяет, является ли загружаемый объект действительным изображением. В дополнение к специальным атрибутам, доступным для FileField, ImageField также имеет атрибуты высоты и ширины.
* JSONField - поле для хранения данных в кодировке JSON. В Python данные представляются в родном для Python формате: словари, списки, строки, числа, булевы и None.
* SmallIntegerField - целое число, но допускает значения только ниже определенной (зависящей от базы данных) точки. Значения от -32768 до 32767 безопасны во всех базах данных, поддерживаемых Django.
* PositiveSmallIntegerField - Подобно SmallIntegerField, но допускает значения только ниже определенного (зависящего от базы данных) предела. Значения от 0 до 32767 безопасны во всех базах данных, поддерживаемых Django.
* UUIDField - Поле для хранения универсальных уникальных идентификаторов. Использует класс UUID из Python. При использовании в PostgreSQL хранится в типе данных uuid, в противном случае - в char(32).
* ForeignKey - Отношение "многие к одному". Требует два позиционных аргумента: класс, с которым связана модель, и параметр on\_delete.

Код реализации моделей представлен в пункте 10.1.1

***5.2 Реализация логики проекта с использованием Django views и templates***

Представления или views — главные составляющие веб-приложений на основе Django. Еще их называют джангистами или вьюхами. Приложение может не взаимодействовать с базой данных, может не использовать шаблоны, но views в нем будут обязательно — приложение должно как-то отвечать на запросы.

В Django используются два вида представлений:

* Представления-функции — view functions
* Представления-классы — class based views

В одном проекте могут одновременно использоваться оба вида. В моем проекте преимущественно используются представления классы. Все представления этого вида наследуются от класса django.views.View. Представления-классы хороши тем, что дают возможность легко переиспользовать поведение через наследование.

Шаблоны Django — это файлы, содержащие html, css и javascript код, которые используются для построения веб-страниц в целях отображения информации. В Django, шаблоны используются для отделения логики представления от логики работы приложения. Вместо того, чтобы вставлять html код в Python файлы, мы можем использовать шаблоны. Для каждой веб-страницы в приложении Django, мы можем создать свой шаблон.

В моём проекте создано три Django приложения, в каждом свой файл views.py с реализацией представлений и папка templates с html шаблонами. Рассмотрим каждое приложение подробнее:

1)***orders*** – тут содержится логика, связанная с заказами:

- class OrderListView, где реализовано отображение списка заказов, наследуемый от django.views.generic.list.ListView, где шаблоном выступает orders.html.

- class OrderCreateView, где реализовано создание нового заказа, наследуемый от django.views.generic.edit.CreateView, где шаблоном выступает order-create.html.

- class OrderDetailView, где реализовано отображение деталей конкретного заказа, наследуемый от django.views.generic.detail.DetailView, где шаблоном выступает order.html.

- class CanceledTemplateView, где реализовано отображение неправильного оформления заказа, наследуемый от django.views.generic.base.TemplateView, где шаблоном выступает canceled.html.

- class SuccessTemplateView, где реализовано отображение правильного оформления заказа, наследуемый от django.views.generic.base.TemplateView, где шаблоном выступает success.html.

2) ***products*** – тут содержится логика, связанная с товарами, категориями товаров и работа с корзиной товаров:

- class IndexView, где реализовано отображение главной страницы c кнопками для перехода в каталог товаров, на страницу профиля, авторизации или регистрации пользователя, наследуемый от django.views.generic.base.TemplateView, где шаблоном выступает index.html.

- class ProductsListView, где реализовано отображение каталога товаров, наследуемый от django.views.generic.list.ListView, где шаблоном выступает products.html.

- class ProductDetailView, где реализовано отображение конкретного товара, наследуемый от django.views.generic.detail.DetailView, где шаблоном выступает product.html.

- также в данном приложении реализованы функции добавления(basket\_add) и удаления(basket\_remove) товаров в корзине.

3) ***users*** – куда, соответственно, поместили логику, связанную с пользователями: авторизация, регистрация, профиль пользователя и верификация электронной почты пользователя. Рассмотрим подробнее:

- class UserLoginView, где реализована логика авторизации пользователя, наследуемый от django.contrib.auth.views.LoginView, где шаблоном выступает login.html.

- class UserRegistrationView, где реализована логика регистрации пользователя, наследуемый от django.views.generic.edit.CreateView, где шаблоном выступает registration.html.

- class UserProfileView, где реализована логика отображения личного кабинета пользователя, наследуемый от django.views.generic.edit.UpdateView, где шаблоном выступает profile.html.

- class EmailVerificationView, где реализована логика подтверждения электронной почты пользователя путем отправки email со ссылкой для подтверждения, наследуемый от django.views.generic.base.TemplateView, где шаблоном выступает email\_verification.html.

Подробнее с кодом реализации представлений и шаблонов можно ознакомиться в приложении 10.1.2

***5.3 Интеграция с базой данных и обработка запросов***

Теперь рассмотрим более детально реализацию логики приложения с использованием Django views и templates на примере приложения ***products***:

**\*\*models.py:\*\***

```python

class Product(models.Model):

name = models.CharField(max\_length=128, unique=True)  
description = models.TextField()  
specifications = models.TextField()  
price = models.DecimalField(max\_digits=6, decimal\_places=0)  
quantity = models.PositiveSmallIntegerField(default=0)  
image = models.ImageField(upload\_to='products\_images')  
category = models.ForeignKey(to=ProductCategory, on\_delete=models.PROTECT)  
is\_available = models.BooleanField(default=True)  
  
 class Meta:  
 verbose\_name = 'Товар'  
 verbose\_name\_plural = "Товары"  
  
 def \_\_str\_\_(self):  
 return self.name

```

Мы создали модель `Product`, которая представляет товары в нашем приложении. Модель содержит поля для имени товара, описания, характеристик товара(спецификации), цены, количества, изображения товара, категории и доступности на складе.

**\*\*views.py:\*\***

```python

class ProductsListView(TitleMixin, ListView):  
 model = Product  
 template\_name = 'products/products.html'  
 paginate\_by = 6  
 title = "UPGrade PC - Каталог"  
  
 def get\_queryset(self):  
 queryset = super(ProductsListView, self).get\_queryset()  
 category\_id = self.kwargs.get('category\_id')  
 return queryset.filter(category\_\_id=category\_id) if category\_id else queryset  
  
 def get\_context\_data(self, \*, object\_list=None, \*\*kwargs):  
 context = super(ProductsListView, self).get\_context\_data(\*\*kwargs)  
 categories = cache.get('categories')  
 if not categories:  
 context['categories'] = ProductCategory.objects.all()  
 cache.set('categories', context['categories'], 30)  
 else:  
 context['categories'] = categories  
 return context

```

Здесь мы создали представление ` ProductsListView `, которое получает все объекты `Product` из базы данных и передает их в шаблон `products.html`. Также здесь реализованы:

- пагинация товаров по 6 штук на странице (Пагинация — это процесс разделения документа на отдельные страницы, электронные или печатные.);

- возможность сортировки товаров по конкретной категории товаров;

- кэширование.

Рассмотрим подробнее логику кэширования данных:

Redis (REmote DIctionary Server, «удалённый серверный словарь») — это нереляционная резидентная СУБД, хранящая данные в виде пар «ключ-значение». Устанавливаем данную программу на наш текущий хост.

Вносим необходимые настройки в файл settings.py, где добавляем новые переменные:

REDIS\_HOST = env('REDIS\_HOST')  
REDIS\_PORT = env('REDIS\_PORT')

CACHES = {  
 'default': {  
 'BACKEND': 'django.core.cache.backends.redis.RedisCache',  
 'LOCATION': f'redis://{REDIS\_HOST}:{REDIS\_PORT}',  
 }  
}

Метод `get\_context\_data` в Django используется для получения контекстных данных, которые будут переданы в шаблон представления.

В данном коде метод `get\_context\_data` переопределен в классе представления. Контекстные данные возвращаются в виде словаря, где ключами являются имена переменных контекста, а значениями - сами данные. Этот словарь затем доступен в шаблоне представления для отображения информации. В данном случае, если есть заданный список объектов (`object\_list`), он будет включен в контекстные данные. Кроме того, метод просто передает все другие ключевые аргументы (`\*\*kwargs`) в контекстные данные без изменений. В данном коде используется функция `cache.get()` для получения значения из кэша, связанного с ключом `'categories'`. Кэширование данных может быть полезным, когда требуется избежать повторного выполнения дорогостоящих операций, например запросов к базе данных или вычислений. При первом вызове `cache.get('categories’) ` данные будут получены из кэша, если они там уже сохранены. Если данных нет в кэше или они устарели, необходимо выполнить необходимую операцию для получения данных и сохранить их в кэше для последующих запросов. В данном коде использована функция `cache`, которая представляет собой механизм кэширования данных. Перед использованием данной функции, необходимо убедиться, что модуль кэширования правильно настроен и подключен к проекту (в моем проекте для работы с кэшированием мы используем Redis.

Если переменная `categories` пуста или равна `None`, то выполняется код внутри условия. Код `context['categories'] = ProductCategory.objects.all()` получает все объекты модели `ProductCategory` из базы данных и сохраняет их в переменную `categories`. Теперь переменная `categories` содержит список всех категорий товаров. Код `cache.set('categories', context['categories'], 30)` устанавливает значение переменной `categories` в кэше на 30 секунд. Теперь мы сможем получить доступ к этому значению из кэша в течение указанного периода времени. Это полезно, если вы хотите избежать повторного выполнения запроса к базе данных и использовать уже сохраненные данные из кэша.

**\*\*products.html:\*\***

**Логика отображения товаров в шаблоне:**

```html

{% for product in object\_list %}  
 <div class="col-lg-4 col-md-6 mb-4">  
 <div class="card h-100">  
 <a href="{% url 'products:product' product.id %}">  
 <img class="card-img-top"  
 src="{{ product.image.url }}"  
 alt="{{ product.description }}">  
 </a>  
 <div class="card-body">  
 <h4 class="card-title">  
 <a href="{% url 'products:product' product.id %}">{{ product.name }}</a>  
 </h4>  
 <h5>{{ product.price|intcomma }} руб.</h5>  
 <p class="card-text">{{ product.description }}</p>  
  
 </div>  
 <div class="card-footer text-center">  
 <a class="btn btn-outline-success"  
 href="{% url 'products:basket\_add' product.id %}">  
 Отправить в корзину  
 </a>  
  
 </div>  
 </div>  
 </div>  
{% endfor %}

```

В шаблоне `products.html` мы используем цикл `for` для прохода по всем товарам и отображения их изображения, имени, описания и цены. Теперь, когда у нас есть модель `Product`, представление ` ProductsListView ` и соответствующий шаблон, мы можем отобразить список товаров на веб-странице.

**Логика пагинации страниц в шаблоне:**

```html

{% if is\_paginated %}  
 <nav aria-label="Page navigation example">  
 <ul class="pagination justify-content-center">  
 <li class="page-item {% if not page\_obj.has\_previous %} disabled {% endif %}">  
 <a class="page-link"  
 href="{% if page\_obj.has\_previous %} {% url 'products:paginator' page\_obj.previous\_page\_number %} {% else %} # {% endif %}"  
 tabindex="-1" aria-disabled="true">  
 Предыдущая  
 </a>  
 </li>  
 {% for page in paginator.page\_range %}  
 <li class="page-item">  
 <a class="page-link" href="{% url 'products:paginator' page %}">  
 {{ page }}  
 </a>  
 </li>  
 {% endfor %}  
  
 <li class="page-item {% if not page\_obj.has\_next %} disabled {% endif %}">  
 <a class="page-link"  
 href="{% if page\_obj.has\_next %} {% url 'products:paginator' page\_obj.next\_page\_number %} {% else %} # {% endif %}">  
 Следующая  
 </a>  
 </li>  
 </ul>  
 </nav>  
{% endif %}

```

Код `{% if is\_paginated %}` представляет собой условную конструкцию в шаблонизаторе, который используется во многих системах управления контентом, таких как Django, Jinja, Liquid и других. Этот код проверяет переменную `is\_paginated` на истинность. Если `is\_paginated` содержит значение, которое интерпретируется как истинное (например, `True` в Python), то выполнится код, следующий за этой конструкцией. Если же `is\_paginated` содержит ложное значение (например, `False` в Python), то код, следующий за этой конструкцией, будет проигнорирован.

- `{% if not page\_obj.has\_previous %} disabled {% endif %}`: Это является шаблонным тегом, используемым в фреймворке, который проверяет условие. Здесь проверяется, есть ли предыдущая страница (`page\_obj.has\_previous`). Если предыдущей страницы нет (`not page\_obj.has\_previous`), то добавляется класс "disabled" к элементу списка. Класс "disabled" используется для стилизации и указания на то, что данная страница является недоступной или неактивной.

- `{% if page\_obj.has\_previous %}`: Это условие проверяет наличие предыдущей страницы (`page\_obj.has\_previous`). Если условие истинно, выполняется следующая часть кода.

- `{% url 'products:paginator' page\_obj.previous\_page\_number %}`: Этот шаблонный тег используется для генерации URL-адреса. В данном случае, он генерирует URL для предыдущей страницы с помощью имени маршрута `'products:paginator'` и номера предыдущей страницы (`page\_obj.previous\_page\_number`).

- `{% else %} # {% endif %}`: Если предыдущей страницы нет (условие ложно), то атрибут `href` устанавливается в символ `#`. Это использовано для создания ссылки, которая не приводит к конкретной странице в случае отсутствия предыдущей страницы.

Выше рассмотрен пример логики приложения с использованием Django views и templates на примере приложения ***products***, я не буду описывать все сценарии использованные в проекте, поскольку это займет слишком много места, а принцип работы у всех представлений примерно одинаков. С кодом всех остальных представлений и шаблонов можно ознакомиться в приложениях.

# **6. Реализация функциональности интернет-магазина "UPGrade PC":**

# ***6.1 Регистрация и аутентификация пользователей***

Для начала детально рассмотрим представления авторизации и регистрации пользователя

*Авторизация пользователей реализована в представлении UserLoginView:*

```python

users/views.py

from django.contrib.auth.views import LoginView

from common.views import TitleMixin

from users.forms import UserLoginForm

class UserLoginView(TitleMixin, LoginView):

template\_name = 'users/login.html'  
form\_class = UserLoginForm  
title = "Авторизация пользователя UPGrade PC"

```

Это представление наследуется от стандартного класса LoginView Django который обрабатывает запросы на вход пользователя, используя стандартный функционал Django для аутентификации и также использует миксин TitleMixin для управления заголовками страницы.

Атрибуты:

title (str): Заголовок страницы, устанавливаемый миксином TitleMixin.

Указывает на цель страницы - в данном случае, на авторизацию пользователя UPGrade PC.

template\_name (str): Путь к шаблону, используемому для отображения страницы входа.

form\_class: Класс формы, используемой для сбора данных во время входа пользователя.

*Рассмотрим реализацию url путей в приложении users:*

from django.contrib.auth.decorators import login\_required  
from django.contrib.auth.views import LogoutView  
from django.urls import path  
  
from users.views import (EmailVerificationView, UserLoginView, UserProfileView,  
 UserRegistrationView)  
  
app\_name = 'users'  
  
urlpatterns = [  
 path('login/', UserLoginView.as\_view(), name='login'),  
 path('registration/', UserRegistrationView.as\_view(), name='registration'),  
 path('profile/<int:pk>/', login\_required(UserProfileView.as\_view()), name='profile'),  
 path('logout/', LogoutView.as\_view(), name='logout'),  
 path('verify/<str:email>/<uuid:code>/', EmailVerificationView.as\_view(), name='email\_verification'),  
]

Как мы видим, реализованы пять различных путей: авторизация, регистрация, личный кабинет, выход из системы и верификация электронной почты пользователя.

*Регистрация пользователей реализована в представлении UserRegistrationView:*

```python

users/views.py

from django.views.generic.edit import CreateView

from common.views import TitleMixin

from users.forms import UserRegistrationForm

class UserRegistrationView(TitleMixin, SuccessMessageMixin, CreateView):  
 model = User  
 form\_class = UserRegistrationForm  
 template\_name = 'users/registration.html'  
 success\_url = reverse\_lazy('users:login')  
 success\_message = 'Вы успешно зарегистрированы'  
 title = "UPGrade PC - Регистрация"

```

Это представление наследуется от стандартного класса CreateView Django который отображает форму для создания объекта, повторного отображения формы с ошибками проверки (если таковые имеются) и сохранения объекта, а также использует миксин TitleMixin для управления заголовками страницы.

Атрибуты:

model: класс модели, используемой для сохранения пользователя в базе данных.

form\_class: класс формы, используемой для сбора данных во время входа пользователя.

template\_name (str): Путь к шаблону, используемому для отображения страницы входа.

success\_url (str): это атрибут, который определяет, куда будет перенаправлен пользователь после успешной аутентификации или выполнения какого-либо действия.

success\_message (str): Это атрибут, который определяет, какое сообщение пользователь увидит после успешной аутентификации или выполнения какого-либо действия.

title (str): Заголовок страницы, устанавливаемый миксином TitleMixin.

Указывает на цель страницы.

Остановимся подробнее на миксине TitleMixin:

class TitleMixin:  
 title = None  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super(TitleMixin, self).get\_context\_data(\*\*kwargs)  
 context['title'] = self.title  
 return context

**Миксин** (дословно: "**примесь**) - класс (class) или интерфейс, в котором некоторые или все его методы (methods) и/или свойства (properties) не реализованы, требуя, чтобы другой класс или интерфейс обеспечивал недостающие реализации. Новый класс или интерфейс затем включает в себя как свойства и методы из **миксина**, так и те, которые он определяет сам.

Как видно из реализации выше, мы добавляем в класс атрибут title, а также добавляем в context значение этого атрибута, что в дальнейшем, после наследования от миксина, позволяет легко передавать в переменную context нужное нам значение для заголовка страницы.

*Аналогичным образом реализовано представление личного кабинета пользователя:*

“class UserProfileView(TitleMixin, UpdateView):  
 model = User  
 template\_name = 'users/profile.html'  
 form\_class = UserProfileForm  
 title = 'UPGrade PC - Личный кабинет'”

# ***6.2 Верификация электронной почты***

Вносим необходимые изменения в файл settings.py, где находятся все основные настройки нашего проекта.

```python

store/settings.py

if DEBUG:  
 EMAIL\_BACKEND = 'django.core.mail.backends.console.EmailBackend'  
else:  
 EMAIL\_HOST = env('EMAIL\_HOST')  
 EMAIL\_PORT = env('EMAIL\_PORT')  
 EMAIL\_HOST\_USER = env('EMAIL\_HOST\_USER')  
 EMAIL\_HOST\_PASSWORD = env('EMAIL\_HOST\_PASSWORD')  
 EMAIL\_USE\_SSL = env('EMAIL\_USE\_SSL')

```

Конструкция if-else, в данном случае, позволяет при разработке и проведении тестирования не менять текущие настройки. Если переменной DEBUG присвоено значение TRUE, то текст электронного письма со ссылкой на подтверждение адреса электронной почты будет выводиться в консоль, в противном случае пользователю будет отправлено электронное письмо на указанный им при регистрации email.

Для того, чтобы проверить электронную почту пользователя, нужно создать уникальную ссылку, при переходе по которой пользователь сможет подтвердить свою электронную почту. Поэтому удобнее создать новую модель, где можно хранить уникальный код и проверять верификацию:

```python

users/views.py

class EmailVerification(models.Model):  
 code = models.UUIDField(unique=True)  
 user = models.ForeignKey(to=User, on\_delete=models.CASCADE)  
 created = models.DateTimeField(auto\_now\_add=True)  
 expiration = models.DateTimeField()  
  
 def \_\_str\_\_(self):  
 return f'EmailVerification object for {self.user.email}'  
  
 def send\_verification\_email(self):  
 link = reverse('users:email\_verification', kwargs={'email': self.user.email, 'code': self.code})  
 verification\_link = f'{settings.DOMAIN\_NAME}{link}'  
 subject = f'Подтверждение учетной записи для {self.user.username}'  
 message = 'Для подтверждения учетной записи пользователя {} перейдите по ссылке: {}'.format(  
 self.user.username,  
 verification\_link  
 )  
 send\_mail(  
 subject=subject,  
 message=message,  
 from\_email=settings.EMAIL\_HOST\_USER,  
 recipient\_list=[self.user.email],  
 fail\_silently=False,  
 )  
  
 def is\_expired(self):  
 return True if now() >= self.expiration else False

```

Поля:

сode – здесь хранится автоматически генерируемый уникальный идентификатор;

user – внешний ключ к пользователю, где указано, что при удалении пользователя удаляется и его объект для верификации email;

created – автоматически заполняемое поле с указанием даты и времени создания объекта;

expiration – дата и время, окончания срока действия ссылки для верификации.

(В модели пользователя необходимо Boolean поле с указанием подтвердил ли пользователь свой email.)

Для того, чтобы после заполнения всех необходимых полей при нажатии пользователем на кнопку регистрация, ему отправлялось письмо с верификацией, нужно переопределить метод save в форме регистрации пользователя добавив следующий код:

def save(self, commit=True):  
 user = super(UserRegistrationForm, self).save(commit=True)  
 send\_email\_verification.delay(user.id)  
 return user

Для того, чтобы пользователю при нажатии на кнопку регистрации не ждать ответа от email сервера, мы реализуем отложенную задачу при помощи Celery (Celery это простая, гибкая и надежная распределенная система для обработки большого количества сообщений, предоставляющая операции с инструментами, необходимыми для обслуживания такой системы.) В корневом каталоге создаем файл ‘store/celery.py’ с необходимыми настройками в соответствии с официальной документацией Celery:

```

import os  
from celery import Celery  
  
os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'store.settings')  
  
app = Celery('store')  
app.config\_from\_object('django.conf:settings', namespace='CELERY')  
app.autodiscover\_tasks()

```

Также в файле “store/\_\_init\_\_.py” добавляем следующий код:

from .celery import app as celery\_app  
  
\_\_all\_\_ = ('celery\_app',)

Указываем необходимые настройки в файле конфигурации проекта store/settings.py:  
CELERY\_BROKER\_URL = f'redis://{REDIS\_HOST}:{REDIS\_PORT}'  
CELERY\_RESULT\_BACKEND = f'redis://{REDIS\_HOST}:{REDIS\_PORT}'

Прописываем функцию send\_email\_verification, указанную в форме регистрации пользователя в файле users/tasks.py для отложенных задач:

@shared\_task()  
def send\_email\_verification(user\_id):  
 user = User.objects.get(id=user\_id)  
 expiration = now() + timedelta(hours=48)  
 record = EmailVerification.objects.create(code=uuid.uuid4(), user=user, expiration=expiration)  
 record.send\_verification\_email()

В данной функции мы берем пользователя из нашей базы данных по его id. Устанавливаем срок действия нашей ссылки на верификацию в значение 48 часов. Создаем объект модели EmailVerification и запускаем у него функцию send\_verification\_email в которой прописана логика формирования и отправки email.

Рассмотрим представление, реализованное в приложении users:

```python

users/views.py

class EmailVerificationView(TitleMixin, TemplateView):  
 title = "UPGrade PC - Подтверждение электронной почты"  
 template\_name = 'users/email\_verification.html'  
  
 def get(self, request, \*args, \*\*kwargs):  
 code = kwargs['code']  
 user = User.objects.get(email=kwargs['email'])  
 email\_verifications = EmailVerification.objects.filter(user=user, code=code)  
 if email\_verifications.exists() and not email\_verifications.first().is\_expired():  
 user.is\_verified\_email = True  
 user.save()  
 return super(EmailVerificationView, self).get(request, \*args, \*\*kwargs)  
 else:  
 return HttpResponseRedirect(reverse('index'))

```

Это представление наследуется от стандартного класса TemplateView Django и также использует миксин TitleMixin для управления заголовками страницы.

# ***6.3 Добавление товаров в корзину и оформление заказа***

*Добавление товаров в корзину:*

Для реализации логики добавления товаров в корзину реализованы модели ‘Product’ (Товар) и ‘Basket’ (Корзина). При нажатии пользователем на кнопку «Добавить в корзину» выполняется работа функции ‘basket\_add’(посмотреть реализацию данной функции можно в приложении 10.1.2), которая принимает на вход аргумент ‘product\_id’. Далее по указанному идентификатору товара создает переменную ‘product’. Создает переменную ‘baskets’, в которую помещает ранее созданный объект Basket, если у пользователя уже был добавлен ранее такой же товар. Затем идет условие if-else, в котором проверяется переменная ‘baskets’ и если значение переменной отличается от ‘None’, то количество указанного товара увеличивается на единицу. В противном случае создается новый объект класса ‘Basket’ с указанным пользователем и товаром в количестве одна штука. Затем идет перенаправление пользователя на ту же страницу, с которой был отправлен запрос.

*Оформление заказа:*

Оформление заказа происходит в личном кабинете пользователя кнопкой оформить заказ, расположенной под интерфейсом отображения текущей корзины пользователя. Логика реализована с помощью класса представления ‘OrderCreateView’ наследуемого от миксина ‘TitleMixin’ и класса ‘CreateView’ из пакета Django django.views.generic.edit.

Класс ‘OrderCreateView’ содержит следующие поля:

template\_name = 'orders/order-create.html'  
form\_class = OrderForm  
success\_url = reverse\_lazy('orders:order\_create')  
title = 'UPGrade PC - Оформление заказа'

Также в данном классе переопределена функция ‘post’ которая проверяет, что корзина пользователя не пуста, и если это так, то запускает функцию ‘update\_after\_payment’ прописанную в модели ‘Order’, которая в свою очередь, реализует добавление в поле ‘basket\_history’ модели ‘Order’ всех товаров из текущей корзины пользователя, очищение текущей корзины и изменения поля ‘status’ на значение оплачен. Затем идет перенаправление пользователя на страницу с сообщением об успешном оформлении заказа. Если же корзина пользователя пуста, то функция ‘post’ перенаправляет пользователя на страницу с сообщением об отмене заказа и просит вернуться к покупкам.

# ***6.4 Управление заказами и администрирование магазина***

*Управление заказами:*

Классом представления ‘OrderDetailView’, наследованным от класса ‘DetailView’, позволяющим удобно отображать детальную информацию переданной модели, из пакета Django django.views.generic.detail, реализована логика отображения данных конкретного заказа.

Класс ‘OrderDetailView’ содержит следующие поля:

template\_name = 'orders/order.html'  
model = Order

Для отображения информации в шаблоне используем такую конструкцию:

<table class="table">  
 <thead>  
 <tr>  
 <th scope="col">Название</th>  
 <th scope="col">Кол-во</th>  
 <th scope="col">Цена</th>  
 <th scope="col">Сумма</th>  
 </tr>  
 </thead>  
 <tbody>  
 {% for purchased\_item in object.basket\_history.purchased\_items %}  
 <tr>  
 <th scope="row">  
 {{ purchased\_item.product\_name }}  
 </th>  
 <td>{{ purchased\_item.quantity }}</td>  
 <td>{{ purchased\_item.price|intcomma }} руб.</td>  
 <td>{{ purchased\_item.sum|intcomma }} руб.</td>  
 </tr>  
 {% endfor %}  
 </tbody>  
</table>

Где в ранее указанной модели ‘Order’ берем поле ‘basket\_history’, затем в нем по ключу ‘purchased\_items’ берем последовательно все значения и по каждому выводим необходимую информацию.

За отображение списка заказов с указанием идентификатора, статуса, даты создания, итоговой суммы заказа и возможностью перехода к конкретному заказу, отвечает класс представления ‘OrderListView’, наследованный от миксина ‘TitleMixin’ и от класса ‘ListView’, из пакета Django django.views.generic.list, позволяющий удобно отобразить список объектов.

Класс ‘OrderListView’ содержит следующие поля:

template\_name = 'orders/orders.html'  
title = 'UPGrade PC - Заказы'  
queryset = Order.objects.all()  
ordering = '-created'

Все поля нам уже известны, за исключением ‘ordering’, которое отвечает за сортировку, в данном случае сортировка осуществляется по атрибуту ‘created’ в обратном порядке.

Для отображения информации в шаблоне используем такую конструкцию:

<table class="table">  
 <thead>  
 <tr class="table-light">  
 <th scope="col">#</th>  
 <th scope="col">Статус</th>  
 <th scope="col">Создан</th>  
 <th scope="col">Итого</th>  
 <th scope="col">Действие</th>  
 </tr>  
 </thead>  
 <tbody>  
 {% for order in object\_list %}  
 <tr>  
 <th scope="row">{{ order.id }}</th>  
 <td>{{ order.get\_status\_display }}</td>  
 <td>{{ order.created|naturaltime }}</td>  
 <td>{{ order.basket\_history.total\_sum|intcomma }} руб.</td>  
 <td>  
 <a href="{% url "orders:order" order.id %}">просмотреть</a>  
 </td>  
 </tr>  
 {% endfor %}  
  
 </tbody>  
</table>

Где в ранее указанном queryset по всем объектам класса ‘Order’ берем последовательно все необходимые значения каждого объекта и выводим необходимую информацию.

# *Администрирование магазина:*

Одна из самых мощных частей Django - автоматический интерфейс администратора. Он считывает метаданные из наших моделей, чтобы обеспечить быстрый, ориентированный на модели интерфейс, в котором доверенные пользователи могут управлять содержимым нашего сайта. Здесь реализовано много возможностей для настройки, если нужно предоставить более ориентированный на процесс интерфейс, абстрагируясь от деталей реализации таблиц и полей базы данных, то, можно написать собственные представления. Для этого проекта этого будет вполне достаточно.

*Регистрация модели ‘Order’ из приложения ‘orders’:*

@admin.register(Order)  
class OrderAdmin(admin.ModelAdmin):  
 list\_display = ('\_\_str\_\_', 'status')  
 fields = (  
 'id', 'created',  
 ('first\_name', 'last\_name'),  
 ('email', 'address'),  
 'basket\_history', 'status', 'initiator',  
 )  
 readonly\_fields = ('id', 'created')

Первой строкой регистрируем текущую модель для отображения в административной панели.

В поле ‘list\_display’ указываем отображение для списка всех заказов.

Поле ‘fields’ отвечает за отображение деталей конкретного заказа. Тут имеет значение последовательность, поскольку в такой же последовательности мы и будем видеть информацию. Также при помощи кортежей можно группировать несколько атрибутов в один блок.

В поле ‘readonly\_fields’ мы указали те поля, у которые будут неизменяемыми.

*Регистрация моделей из приложения ‘products’:*

admin.site.register(ProductCategory)

Здесь мы регистрируем модель ‘ProductCategory’ в административной панели, не меняя никаких настроек для отображения и взаимодействия.

@admin.register(Product)  
class ProductAdmin(admin.ModelAdmin):  
 list\_display = ['name', 'price', 'quantity', 'category']  
 fields = ['image', 'name', 'description', 'specifications', ('price', 'quantity'), 'category', 'is\_available']  
 readonly\_fields = ('is\_available',)  
 search\_fields = ('name',)  
 ordering = ('-quantity',)

Тут показана регистрация модели ‘Product’ с уже знакомыми нам настройками, кроме поля ‘search\_fields’, которое, как понятно из названия, добавляет строку для поиска по наименованию товара.

В интерфейсе администратора есть возможность редактировать модели на той же странице, что и родительская модель. Это называется вставками. Django предоставляет два подкласса InlineModelAdmin, и это:

TabularInline

StackedInline .

Разница между ними заключается лишь в шаблоне, используемом для их отображения. В проекте реализован класс ‘BasketAdmin’, который позволяет нам редактировать корзину пользователя на странице этого пользователя, в котором указана модель корзина и её поля для отображения.

class BasketAdmin(admin.TabularInline):  
 model = Basket  
 fields = ('product', 'quantity')  
 extra = 0

Параметр `extra = 0` указывает, что по умолчанию на странице администрирования не будут отображаться дополнительные пустые формы для добавления объектов `Basket`.

*Регистрация моделей из приложения ‘users’:*

@admin.register(User)  
class UserAdmin(admin.ModelAdmin):  
 list\_display = ('username',)  
 inlines = (BasketAdmin,)

Тут всё уже знакомо, кроме поля ‘inlines’ в котором мы используем наш класс-вставку ‘BasketAdmin’ на странице конкретного пользователя.

@admin.register(EmailVerification)  
class EmailVerificationAdmin(admin.ModelAdmin):  
 list\_display = ('user', 'code', 'created', 'expiration')  
 fields = ('code', 'user', 'created', 'expiration')  
 readonly\_fields = ('created',)

Тут тоже всё знакомо, просто добавили отображение модели ‘EmailVerification’ с удобными для просмотра и редактирования в этом проекте параметрами.

# **7. Тестирование и отладка:**

# ***7.1 Тестирование функциональности***

В данном проекте реализованы некоторые автоматические тесты для проверки основной функциональности:

*Тесты приложения ‘products’:*

* Проверка правильности отображения главной страницы;
* Проверка правильности заголовка главной страницы;
* Проверка правильности использования конкретного шаблона для отображения главной страницы;
* Проверка отображаемых товаров;
* Проверка отображаемых товаров при выборе конкретной категории для отображения;
* Проверка правильности отображения страницы с товарами;
* Проверка правильности отображения заголовка страницы с товарами;
* Проверка правильности использования конкретного шаблона для отображения главной страницы.

*Тесты приложения ‘users’:*

* Проверка правильности отображения страницы регистрации;
* Проверка правильности отображения заголовка страницы регистрации;
* Проверка правильности использования конкретного шаблона для отображения страницы регистрации;
* Проверка, что в базе данных не существует незарегистрированного пользователя;
* Проверка успешной регистрации пользователя;
* Проверка, что новый зарегистрированный пользователь, существует в базе данных;
* Проверка верификации электронной почты пользователя;
* Проверка времени действия уникальной ссылки для верификации электронной почты пользователя;
* Проверка, что при регистрации невозможно указать имя пользователя, которое уже существует в базе данных.

# ***7.2 Линтеры и форматеры***

**Линтеры** (linters) – это инструменты, которые используются в программировании для автоматической проверки и анализа исходного кода на предмет соответствия стандартам кодирования и обнаружения потенциальных проблем. Они играют важную роль в обеспечении качества кода, повышают читаемость и поддерживаемость программного обеспечения, а также помогают предотвратить ошибки и баги.

В качестве линтера в проекте использовался ‘flake8’, который помогал автоматически анализировать и проверять код на соответствие стандартам оформления и выявлять потенциальные проблемы. Flake8 — инструмент, позволяющий просканировать код проекта и обнаружить в нем стилистические ошибки и нарушения различных конвенций кода на Python.

**Форматеры** **кода** (code formatters) — это автоматизированные инструменты, которые помогают автоматически форматировать исходный **код**. Их основная цель - стандартизировать форматирование кода в рамках проекта или команды, облегчая чтение и понимание кода.

Для форматирования кода в соответствии с определенными стилями в проекте использовали ‘isort’, который автоматически приводил код к общему стилю и форматированию. Isort — это библиотека Python для сортировки импорта по алфавиту с автоматическим разделением на разделы и по типу.

Эти инструменты позволили поддерживать единый стиль кодирования в проекте, улучшить читаемость кода и обеспечить соответствие кода определенным стандартам. В результате мы смогли значительно упростить процесс разработки, обнаруживать потенциальные проблемы и улучшать качество кода веб-приложения.

# **8. Заключение:**

# ***8.1 Выводы по результатам разработки серверной части веб-приложения***

В ходе разработки серверной части веб-приложения "UPGrade PC" с использованием Django фреймворка были достигнуты следующие результаты:

Была создана стабильная и масштабируемая серверная часть, способная обрабатывать запросы от пользователей, обеспечивать безопасность данных и управлять базой данных эффективным образом.

Мы успешно реализовали основные функции, такие как управление товарами, корзиной покупок, обработка заказов и аутентификация пользователей. Также была обеспечена поддержка административной панели для управления данными и мониторинга деятельности интернет-магазина.

В процессе разработки мы использовали передовые методы и инструменты, такие как использование Django ORM для взаимодействия с базой данных, использование механизмов аутентификации и авторизации Django для обеспечения безопасности приложения.

Несмотря на успешное выполнение поставленных задач, были выявлены некоторые аспекты, требующие улучшения, такие как оптимизация производительности и расширение функциональности. Кроме того, для дальнейшего развития приложения необходимо провести более глубокий анализ требований пользователей и рыночной конкуренции.

В целом, разработка серверной части веб-приложения "UPGrade PC" была успешной, и полученный опыт и результаты позволят нам продолжить развитие и совершенствование приложения в будущем.

# ***8.2 Дальнейшие перспективы развития и улучшения интернет-магазина "UPGrade PC"***

"Интернет-магазин 'UPGrade PC' имеет потенциал для дальнейшего развития и улучшения в различных аспектах. Основываясь на полученном опыте и результатах разработки серверной части веб-приложения, можно выделить следующие перспективы для улучшения и расширения функциональности интернет-магазина:

1. Расширение ассортимента товаров: в рамках дальнейшего развития интернет-магазина планируется расширение ассортимента товаров, добавление новых категорий и брендов, что позволит привлечь больше клиентов и увеличить объем продаж.

2. Оптимизация производительности: Проведение дополнительной оптимизации серверной части приложения для обеспечения более быстрой загрузки страниц, улучшения отзывчивости и общего пользовательского опыта.

3. Внедрение механизмов персонализации: Разработка и внедрение механизмов персонализации, таких как рекомендательные системы, учет предпочтений пользователей, что позволит улучшить качество обслуживания и повысить лояльность клиентов.

4. Доработка мобильной версии: Улучшение мобильной версии интернет-магазина, учитывая растущее количество пользователей, предпочитающих мобильные устройства для онлайн-покупок.

5. Внедрение аналитики и отчетности: Разработка и внедрение системы аналитики для мониторинга ключевых показателей производительности интернет-магазина, что поможет принимать обоснованные решения и улучшать бизнес-процессы.

Эти перспективы развития и улучшения интернет-магазина 'UPGrade PC' станут основой для дальнейшего развития и укрепления позиций на рынке электронной коммерции, обеспечивая конкурентное преимущество и удовлетворение потребностей клиентов."

# **9. Список использованных источников:**

# ***9.1 Официальная документация и руководства***

Django - <https://www.djangoproject.com/>

Celery - <https://docs.celeryq.dev/en/stable/getting-started/introduction.html>

Redis - <https://redis.io/>

Bootstrap - <https://getbootstrap.ru/>

PostgreSQL - <https://www.postgresql.org/>

OAuth 2.0 - <https://oauth.net/2/>

# ***9.2 Интернет-ресурсы и статьи***

<https://studbooks.net/2015118/informatika/arhitektura_struktura_sayta_freymvorke_django>

<https://developer.mozilla.org/ru/docs/Learn/Server-side/Django/Models>

<https://www.dbdesigner.net>

<https://pythonpip.ru/django/zaprosy-django-orm>

<https://docs.celeryq.dev/en/stable/getting-started/introduction.html>

<https://alimbekov.com/beautiful-python-code-simple-steps/>

# **10. Приложения:**

# ***10.1 Код серверной части веб-приложения на Django***

10.1.1 Модели Django для представления данных:

Модели приложения orders:

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Модели приложения products:

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Модели приложения users:

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

# 10.1.2 Реализация логики проекта с использованием Django views и templates

Представления приложения orders:

Изображение выглядит как текст, снимок экрана, меню

Автоматически созданное описание

Представления приложения products:

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Представления приложения users:

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

# 10.1.3 Реализация автоматического тестирования приложений ‘products’ и ‘users’:

Products/tests.py:

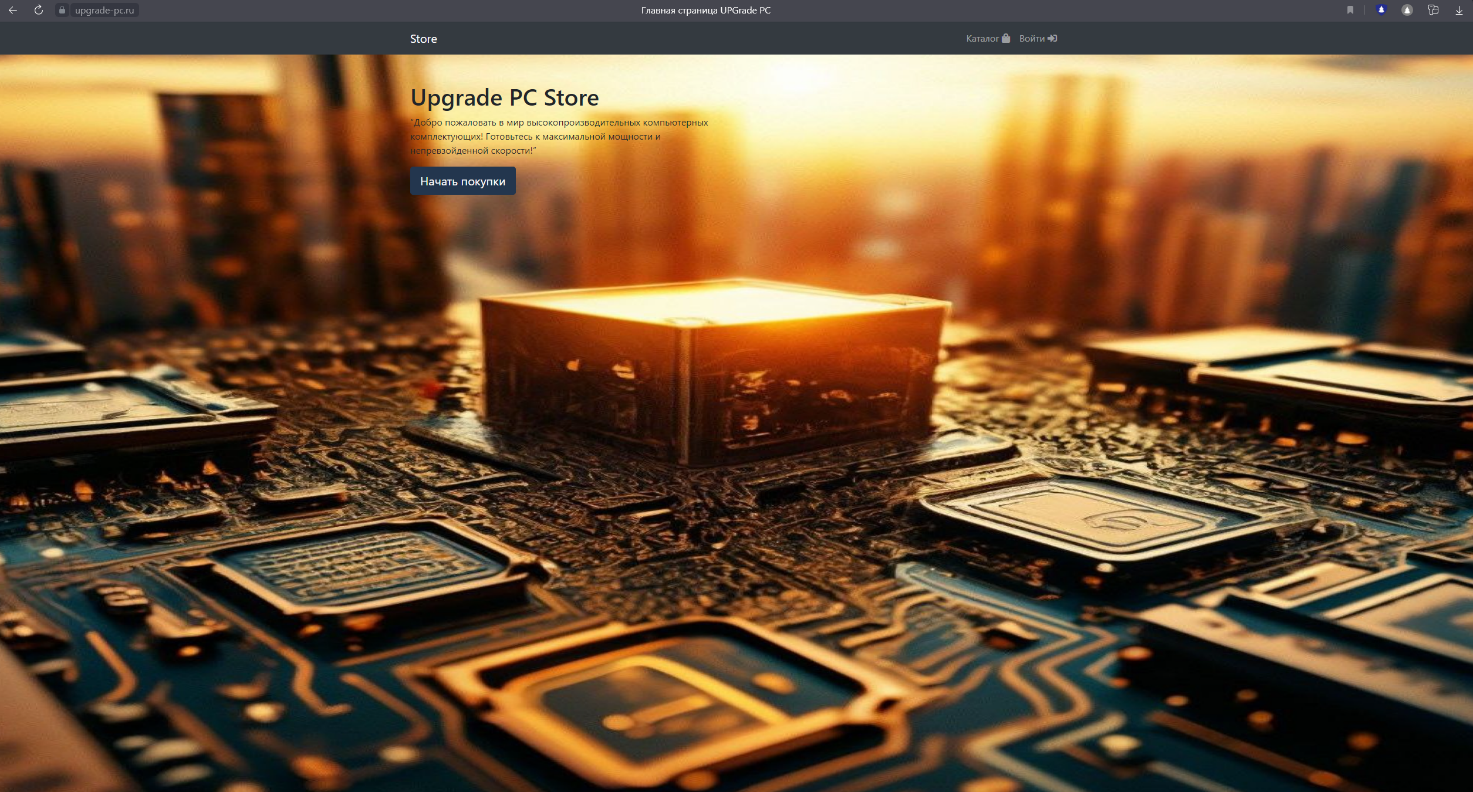


Users/tests.py:

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

# ***10.2 Снимки экрана интерфейса интернет-магазина "UPGrade PC"***



Главная страница

Изображение выглядит как текст, программное обеспечение, снимок экрана, Мультимедийное программное обеспечение

Автоматически созданное описание

Страница каталога товаров

Изображение выглядит как текст, снимок экрана, дизайн

Автоматически созданное описание

Страница конкретного товара

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Страница регистрации

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Страница авторизации

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Страница личного кабинета пользователя

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Страница отображения заказа

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Страница отображения всех заказов

# ***10.3 Дополнительные материалы и данные:***

10.3.1 Приложение №1: Схема Базы Данных.

Изображение выглядит как текст, снимок экрана, диаграмма, число

Автоматически созданное описание