

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Отделение информационных технологий и вычислительной техники

Утверждаю
Руководитель
отделения ИТВТ _____.

Рощенко О.Е.
(подпись, фамилия, инициалы)
«___» _____ 20__ г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

ДИПЛОМНЫЙ ПРОЕКТ

НА ТЕМУ

РАЗРАБОТКА САЙТА МАГАЗИНА

ОДЕЖДЫ И ОБУВИ

Автор дипломного проекта.....

(подпись студента, выполнившего дипломный проект)

Василенко М.И...... Группа И-93

(фамилия, инициалы студента)

(в которой обучался студент)

Институт социальных технологий

(факультет)

Специальность 09.02.03 Программирование в компьютерных системах

(код и наименование специальности)

Руководитель дипломного проекта И.Н. Прохорова

(подпись, инициалы, фамилия)

Новосибирск, 2023 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Отделение информационных технологий и вычислительной техники

Утверждаю
Руководитель
отделения ИТВТ _____

Рощенко О.Е.
(подпись, фамилия, инициалы)
« ____ » _____ 20__ г.

ЗАДАНИЕ НА ДИПЛОМНЫЙ ПРОЕКТ

студенту(ке) Василенко Михаилу Ивановичу группы И-93
(фамилия, инициалы) (обучения)

1. Тема Разработка сайта магазина одежды и обуви

Утверждена приказом по НГТУ № 5575/2 от « 26 » декабря 2022 г.

2. Дата представления проекта к защите « ____ » _____ 20__ г.

3. Цели проекта (исходные данные) Разработать рабочую версию сайта магазина одежды и обуви. На сайте должна быть реализована логика каталога товаров, учетных записей пользователей, работы с корзиной, обработки заказов, товаров и работы со складом.

4. Содержание пояснительной записки

- 4.1 Введение
- 4.2 Глава 1. Анализ предметной области и теоретические основы разработки
- 4.3 Постановка задачи
- 4.4 Основная аудитория и её потребности
- 4.5 Основные категории товаров
- 4.6 Анализ требований к функциональности
- 4.8 Обоснование выбора инструментальных средств разработки
- 4.9 Особенности веб-разработки

4.10 Глава 2. Практическая реализация проекта

4.11 План работы по проекту

4.11 Дизайн проекта

4.12 Функциональное описание разработки

4.13 Конфигурирование

4.14 Платежные системы

4.15 Заключение

4.16 Список источников

4.17 Приложение

5. Перечень графического и (или) иллюстрационного материала *Рис.1 ER-диаграмма, рис.2 скриншот из редактора, рис.3 макет главной страницы, рис.4 макет страницы с фильтрацией, рис.5 макет страницы товара, рис.6 макет личного кабинета, рис.7 макет корзины товаров, рис.8 макет формы регистрации, рис.9 макет формы авторизации, рис.10 макет формы заказа, рис.11 организация файлов в проекте, рис.12 одновременный запуск Flask и React приложений, рис.13 скриншот главной страницы проекта из примера.*

Руководитель проекта

(подпись, дата)

Прохорова И.Н.

(фамилия, инициалы)

Задание принял к исполнению

(подпись студента, дата)

Василенко М.И.

(фамилия, инициалы студента)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	5
ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ.....	6
1.1. Постановка задачи.....	6
1.2. Основная аудитория и её потребности	6
1.3. Основные категории товаров	7
1.4. Анализ требований к функциональности	7
1.5. Обоснование выбора инструментальных средств разработки	8
1.6. Особенности веб-разработки	14
ГЛАВА 2. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ ПРОЕКТА	15
2.1. План работ по проекту.....	15
2.2. Дизайн проекта	20
2.3. Функциональное описание разработки.....	27
2.3.1. Конфигурирование.....	47
2.3.2. Платежные системы.....	53
ЗАКЛЮЧЕНИЕ	56
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	58
ПРИЛОЖЕНИЕ	59

ВВЕДЕНИЕ

Интернет-магазины становятся все более популярными в мире бизнеса. Они позволяют компаниям привлекать новых клиентов и увеличивать доходы, обеспечивая покупателям удобство и простоту при покупке товаров. В настоящее время создание интернет-магазина является одним из самых важных шагов для успешного бизнеса в онлайн-среде. В данной работе предполагается рассмотреть все важные аспекты создания интернет-магазина, включая разработку всей необходимой логики для работы магазина.

Цель проекта – разработка сайта магазина одежды и обуви.

Главные задачи проекта:

1. Проектирование и разработка базы данных.
2. Разработка Flask приложения, обладающего необходимой функциональностью и возможность модернизации для использования с другим интерфейсом.
3. Подключение базы данных PostgreSQL к Flask приложению.
4. Разработка функций и написание запросов к базе данных с использованием ORM.
5. Разработка макетов с использованием графического редактора Figma.
6. Разработка шаблонов с помощью шаблонизатора Jinja2.

ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ

1.1. Постановка задачи

Цель проекта – разработка сайта магазина одежды и обуви, который бы отвечал всем требованиям современного интернет-бизнеса. Сайт должен из себя представлять многостраничный магазин одежды и обуви с главной страницей, на которую будут выводиться товары из базы данных, функциональная корзина товаров, доступная только авторизованным пользователям, страничка товара, которая будет генерироваться для каждого товара автоматически, поиск товара и фильтрация по категориям, авторизация и регистрация.

Главные задачи на разработку:

1. Проектирование и разработка базы данных.
2. Разработка Flask приложения, обладающего необходимой функциональностью и возможность модернизации для использования с другим интерфейсом.
3. Подключение базы данных PostgreSQL к Flask приложению.
4. Разработка функций и написание запросов к базе данных с использованием ORM.
5. Разработка макетов с использованием графического редактора Figma.
6. Разработка шаблонов с помощью шаблонизатора Jinja2.

1.2. Основная аудитория и её потребности

Общими характеристиками аудитории сайта магазина одежды и обуви, скорее всего, будут: возраст от 18 до 40 лет, преимущественно женщины, заинтересованные в моде и стиле. Однако это только общие характеристики, и в итоге аудитория может отличаться в зависимости от бренда и продуктов.

Обычно определение аудитории и другие задачи, связанные с функциональной частью сайта и его дизайном, решаются командой продуктовых менеджеров. Они занимаются анализом рынка, конкурентов, определением потребностей и желаний клиентов.

1.3. Основные категории товаров

Определение основных категорий товаров — это очень важный шаг при разработке сайта магазина одежды и обуви. Категории товаров помогают клиентам быстро найти то, что им нужно, и сокращают время поиска товаров.

Основные категории товаров можно определить, как:

- кеды;
- кроссовки;
- футболки;
- толстовки.

Каждая категория может иметь подкатегории, например, обувь может быть разбита по полу, возрасту, материалу и т.д. Важно, чтобы категории были логичными и понятными для клиентов, чтобы они могли быстро и без труда найти нужный товар.

1.4. Анализ требований к функциональности

Необходимо определить, какие функции будут доступны пользователям на сайте интернет-магазина, как они будут реализовываться и как они будут взаимодействовать с базой данных.

1. Отображение каталога товаров - интернет-магазин должен иметь возможность отображать свой ассортимент товаров, включая информацию о каждом товаре, такую как его название, описание, цена, фотографии, наличие и т.д.
2. Корзина покупок - покупатели должны иметь возможность добавлять товары в корзину покупок, удалять их и просматривать содержимое корзины в любое время.
3. Поиск и фильтрация товаров - интернет-магазин должен иметь возможность поиска и фильтрации товаров по различным параметрам, таким как цена, бренд, размер, цвет и т.д.
4. Оформление заказа - покупатели должны иметь возможность оформления заказа, включая ввод информации о доставке и оплате.

5. Регистрация и авторизация покупателей - интернет-магазин должен иметь возможность регистрации и авторизации покупателей, включая возможность создания профиля покупателя и хранения истории заказов.
6. Управление заказами - интернет-магазин должен иметь возможность управления заказами, включая уведомления о новых заказах, обработку заказов, отслеживание статуса и доставки заказов и т.д.

1.5. Обоснование выбора инструментальных средств разработки

Был выбран **Visual Studio Code** для работы над проектом веб-разработки по нескольким причинам. Во-первых, VS Code имеет множество интеграций с плагинами и расширениями, что позволяет расширять его функциональность и делает его более удобным в использовании. Во-вторых, VS Code обладает интеграцией с Git, что позволяет контролировать версию своего кода. В-третьих, VS Code мультиплатформенный и легковесный редактор кода, который не нагружает процессор и позволяет сохранять продуктивность при работе с большими объемами кода.

Кроме того, Visual Studio Code имеет богатый функционал, который делает его удобным для разработки веб-приложений. Он имеет инструментарий для отладки кода, подсветки синтаксиса, автодополнения и много других функций, которые позволяют быстро и удобно создавать качественный код.

Sublime Text, Atom, Pycharm и другие редакторы также могут быть хорошими инструментами для веб-разработки, но выбор Visual Studio Code был сделан в связи с его особенностями и преимуществами, которые делают его удобным для использования в качестве редактора кода для веб-приложений.

Язык программирования Python был выбран для разработки интернет-магазина по нескольким причинам. Первая из них заключается в том, что Python является одним из наиболее популярных языков программирования в мире. Он широко используется в различных областях, в том числе в разработке веб-приложений и интернет-магазинов.

Кроме того, Python — это язык с динамической типизацией, что означает, что тип переменной определяется в процессе выполнения программы, а не на

этапе компиляции. Это позволяет разработчикам быстро и легко вносить изменения в код, не переживая о его типизации.

Для разработки интернет-магазина, помимо Python можно было использовать язык программирования PHP, который является одним из наиболее распространенных языков программирования для веб-разработки и особенно популярен в разработке динамических веб-страниц. Однако, выбор между Python и PHP зависит от конкретных потребностей проекта.

Python и PHP обладают схожими преимуществами, такими как простота использования, мощный функционал и популярность в веб-разработке. Однако, Python обеспечивает динамическую типизацию, что позволяет быстро и легко вносить изменения в код, в то время как PHP является статически типизированным языком программирования.

В целом, Python и PHP являются хорошими инструментами для разработки интернет-магазина и выбор зависит от требований проекта и личных предпочтений разработчика.

Фреймворк Flask был выбран для разработки интернет-магазина по нескольким причинам. Первая из них заключается в том, что Flask является легковесным и гибким фреймворком для веб-приложений. Он предоставляет набор инструментов для создания веб-сайтов и позволяет разработчикам свободно настраивать их. Благодаря этому Flask позволяет быстро разрабатывать веб-приложения.

Кроме того, Flask очень быстрый и производительный фреймворк. Он способен обрабатывать большие объемы данных и обеспечивать высокую скорость работы веб-сайта. Это особенно важно для интернет-магазинов, где каждая секунда задержки может привести к потере клиента. Также следует отметить богатый набор инструментов, предоставляемых Flask. Он включает в себя множество расширений и плагинов, которые упрощают разработку и позволяют быстро добавлять новые функции.

Django, с другой стороны, является полноценным фреймворком для веб-разработки, который предоставляет большое число инструментов для создания

web-приложений любой сложности. Использование Django может занять больше времени, чем Flask, из-за более сложного уровня проектирования и использования учебных кривых. Однако, Django более подходит для крупных и сложных веб-приложений.

В целом Flask подходит для создания простых интернет-магазинов и маленьких сайтов, в то время как Django более подходит для крупных, сложных веб-приложений.

Ниже перечислены основные причины выбора **PostgreSQL**.

PostgreSQL обладает высокой степенью надежности и защиты данных, благодаря своей архитектуре и механизмам безопасности. Он обеспечивает сохранность и целостность данных, а также защищает их от несанкционированного доступа и взломов.

PostgreSQL обладает большим количеством расширений и плагинов, которые позволяют улучшать функциональность и масштабировать БД под нужды проекта.

PostgreSQL имеет активное сообщество разработчиков и пользователей, которые обеспечивают ее постоянную поддержку и развитие. Это позволяет решать проблемы и находить ответы на вопросы быстро и эффективно.

PostgreSQL обладает простым и интуитивно понятным интерфейсом, что делает его легким в освоении и использовании как для новичков, так и для профессионалов. Использование PostgreSQL для управления данными позволяет сохранять целостность и конфиденциальность данных, а также облегчает их обработку и управление.

Одним из основных конкурентов PostgreSQL является MySQL. В отличие от PostgreSQL, MySQL более легковесная СУБД, что делает ее лучшим выбором для простых веб-приложений или приложений со средней нагрузкой.

Однако она не обладает таким высоким уровнем безопасности и не обладает такими же мощными возможностями масштабирования, как PostgreSQL.

SQLite, которая также является легковесной СУБД, используется обычно для локального хранения данных. Использование SQLite в качестве серверной СУБД может быть лучшим выбором только для простых проектов с очень низкой нагрузкой на сервер.

PostgreSQL также конкурирует с MS SQL Server. MS SQL Server более распространен на корпоративном уровне и часто используется для управления большими объемами данных на платформе Microsoft.

Однако, открытый и бесплатный характер PostgreSQL является преимуществом для проектов с ограниченным бюджетом. В целом, выбор между PostgreSQL и другими СУБД зависит от потребностей проекта. PostgreSQL является хорошим выбором для проектов, которые требуют высокой надежности и безопасности данных, а также мощных возможностей масштабирования.

Менеджером баз данных был выбран **pgAdmin**, по ниже перечисленным причинам.

pgAdmin — это бесплатный инструмент для управления базами данных PostgreSQL. Данный инструмент выбран неслучайно, так как он обладает множеством преимуществ и особенностей, которые делают его незаменимым инструментом для управления базами данных PostgreSQL. Основные причины выбора pgAdmin:

pgAdmin обладает простым и интуитивно понятным интерфейсом, что делает его легким в освоении и использовании как для новичков, так и для профессионалов.

pgAdmin поддерживает работу на различных операционных системах, что делает его удобным для использования в различных условиях и на различных устройствах.

pgAdmin обладает множеством функций и инструментов, которые позволяют управлять базами данных PostgreSQL на высоком уровне и обеспечивать их сохранность и целостность данных.

pgAdmin обладает функционалом для безопасности баз данных, таких как шифрование, аутентификация и авторизация. Данный функционал обеспечивает сохранность данных и защиту от несанкционированного доступа.

pgAdmin имеет активное сообщество разработчиков и пользователей, которые обеспечивают его постоянную поддержку и развитие. Это позволяет решать проблемы и находить ответы на вопросы быстро и эффективно.

Использование pgAdmin в качестве менеджера баз данных для PostgreSQL позволяет обеспечить комфортное и удобное управление базами данных, а также повысить их сохранность и целостность.

В выборе между pgAdmin и DBeaver для управления базами данных PostgreSQL, был выбран pgAdmin. Это связано с тем, что pgAdmin имеет множество преимуществ и особенностей, которые делают его незаменимым инструментом для управления базами данных PostgreSQL.

Во-первых, pgAdmin обладает простым и интуитивно понятным интерфейсом.

Во-вторых, pgAdmin поддерживает работу на различных операционных системах.

В-третьих, pgAdmin обладает множеством функций и инструментов.

Для редактирования фотографий товаров в проекте будет использоваться программа **Photoshop**. Такое решение было принято, так как данная программа обладает множеством особенностей и инструментов, которые делают ее одним из наиболее популярных инструментов для графического дизайна

Для проектирования ER-диаграмм будет использоваться веб-сервис **app.diagrams.net**, ниже перечислены причины его использования.

Во-первых, app.diagrams.net позволяет создавать диаграммы с помощью интуитивно понятного интерфейса и широкого набора инструментов. Это особенно актуально для создания ER-диаграмм, которые позволяют визуализировать связи между сущностями базы данных.

Во-вторых, app.diagrams.net является бесплатным онлайн-сервисом, который не требует установки дополнительного программного обеспечения на

компьютере. Это позволяет быстро запустить инструмент и начать работу над диаграммой, что экономит время и упрощает процесс разработки.

Одним из основных недостатков ER-диаграмм в pgAdmin является ограниченный набор функций и настроек для отрисовки диаграмм. PgAdmin позволяет создавать ER-диаграммы, но он не обладает такими широкими возможностями по настройке и управлению элементами диаграммы, как app.diagrams.net. Возможности pgAdmin ограничены в плане добавления пользовательских элементов или настройки параметров элементов.

[App.diagrams.net](https://app.diagrams.net), с другой стороны, предоставляет широкий набор инструментов для работы с ER-диаграммами, что позволяет более точно настроить отображение диаграммы в соответствии с требованиями проекта. Он также обладает более продвинутым интерфейсом и более высокой скоростью работы. В целом, если нужно создать высококачественную, настраиваемую ER-диаграмму, app.diagrams.net более подходящий выбор, чем pgAdmin.

Для работы с графическими макетами был выбран онлайн-сервис **Figma**. Этот выбор был обусловлен несколькими причинами.

Во-первых, Figma позволяет быстро создавать и редактировать графические макеты независимо от того, где вы находитесь и каким устройством вы пользуетесь. Это особенно важно в случае распределенной команды разработчиков, когда разработчики находятся в разных местах и используют различные устройства.

Во-вторых, Figma позволяет создавать макеты с высокой степенью точности и детализации. Он предоставляет широкий набор инструментов для создания различных элементов интерфейса, включая кнопки, поля ввода, выпадающие списки и многое другое. Это позволяет разработчикам создавать качественные макеты, которые полностью соответствуют требованиям проекта.

В-третьих, Figma позволяет легко и быстро создавать прототипы макетов. Это позволяет проверять макеты на практике, тестировать их на пользователях и вносить корректировки с учетом полученной обратной связи.

В-четвертых, Figma предоставляет возможность совместной работы над макетами в режиме реального времени. Это позволяет разработчикам быстро общаться и синхронизироваться друг с другом, что упрощает процесс разработки и позволяет быстро решать возникающие проблемы.

Использование редактора Figma для создания макетов сайта было выбрано в результате анализа всех доступных инструментов и было признано наиболее удобным и эффективным способом для создания качественных макетов.

1.6. Особенности веб-разработки

Веб-разработка — это процесс создания веб-приложений, которые могут быть запущены в браузере. Этот процесс включает в себя различные этапы, такие как проектирование пользовательского интерфейса, разработка базы данных, создание серверной части и т.д. Ниже перечислены основные особенности веб-разработки.

Frontend разработка — это разработка пользовательского интерфейса веб-страницы. Она включает в себя создание дизайна страницы, написание кода на языках HTML, CSS и JavaScript, а также тестирование и отладку интерфейса. Frontend разработка направлена на то, чтобы сделать веб-приложение удобным и привлекательным для пользователя.

Backend разработка — это разработка серверной части веб-приложения. Она включает в себя создание базы данных, написание серверного кода на языках программирования, таких как Python, Ruby, PHP, JavaScript и др., а также тестирование и отладку функционала. Она направлена на обработку данных и сохранение их в базе данных, а также на управление функционалом сайта.

ГЛАВА 2. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ ПРОЕКТА

2.1. План работ по проекту

Выделим основные задачи:

1. Разработать и описать структуру данных ER-диаграммой.
2. Создать базу данных по имеющейся ER-диаграмме и наполнить её тестовыми данными.
3. Создать Flask приложение, подключить базу данных.
4. Создать и описать функции принимающие и возвращающие данные из БД по запросу:
 - a. Для главной страницы (отображение карточек товаров).
 - b. Для регистрации и авторизации пользователей.
 - c. Для поиска товаров.
 - d. Для фильтрации товаров по его характеристикам.
 - e. Для отслеживания популярности товаров.
 - f. Для реализации корзины товаров.
 - g. Для отслеживания наличия товаров в наличии.
 - h. Для оформления заказов.
5. Разработать шаблоны с использованием Jinja2 для демонстрации работоспособности сайта.

2.1.1. Проектирование структуры данных

Изучив предметную область, были составлены требования к базе данных магазина: хранить информацию о товарах, контактную информацию покупателей и информацию о заказах. База данных для интернет-магазина должна состоять из таблиц, перечисленных ниже.

Товары (Products), брэнды (Brands), категории (Category), цвет (Color), размеры (Sizes), склад (Storage), Пользователи (Users), корзины товаров (Cart), заказы (Orders), информация о заказанных продуктах (Order_items), статусы заказа (Order_status), информация о закупках (Purchase).

Для удобства и наглядности была нарисована ER-диаграмма, которая предоставлена на рисунке (рис.1). Код создания базы данных (см. Приложение 1).

Таблицы: **Brands, Category, Color, Sizes** - классификаторы, в них есть поля: «id» - Индекс и «name» - название.

Код создания таблиц: Brands (см. Приложение 2), Category (см. Приложение 3), Color (см. Приложение 4), Sizes (см. Приложение 5).

Классификаторы в структурах данных используются для группирования различных данных, которые могут быть представлены одним и тем же типом, но имеют различия в значениях.

Например, если у нас есть набор данных, таких как список продуктов в магазине, то мы можем использовать классификаторы для группирования продуктов по категориям, таким как "электроника", "одежда", "книги" и т.д. Это позволяет легче и быстрее искать определенные продукты и делать выборки по конкретным классификаторам.

Классификаторы также могут использоваться для категоризации данных в базе знаний или поисковых системах, чтобы упростить их поиск и улучшить производительность приложений. Кроме того, они могут использоваться в системах машинного обучения для классификации различных объектов и данных по различным критериям.

Таблица **Users** определяет структуру таблицы в базе данных, которая содержит информацию о пользователях. Каждая строка этой таблицы представляет отдельного пользователя и содержит информацию о его идентификаторе (поле id), имени (name), логине (login), хэше пароля (password_hash) и электронной почте (email).

Одно из применений этой таблицы реализация авторизации и аутентификации пользователей в приложении, используя данные из таблицы. Кроме того, она может быть использована для хранения других данных, связанных с пользователями, таких как их заказы или избранные товары.

Код создания таблицы Users (см. Приложение 6).

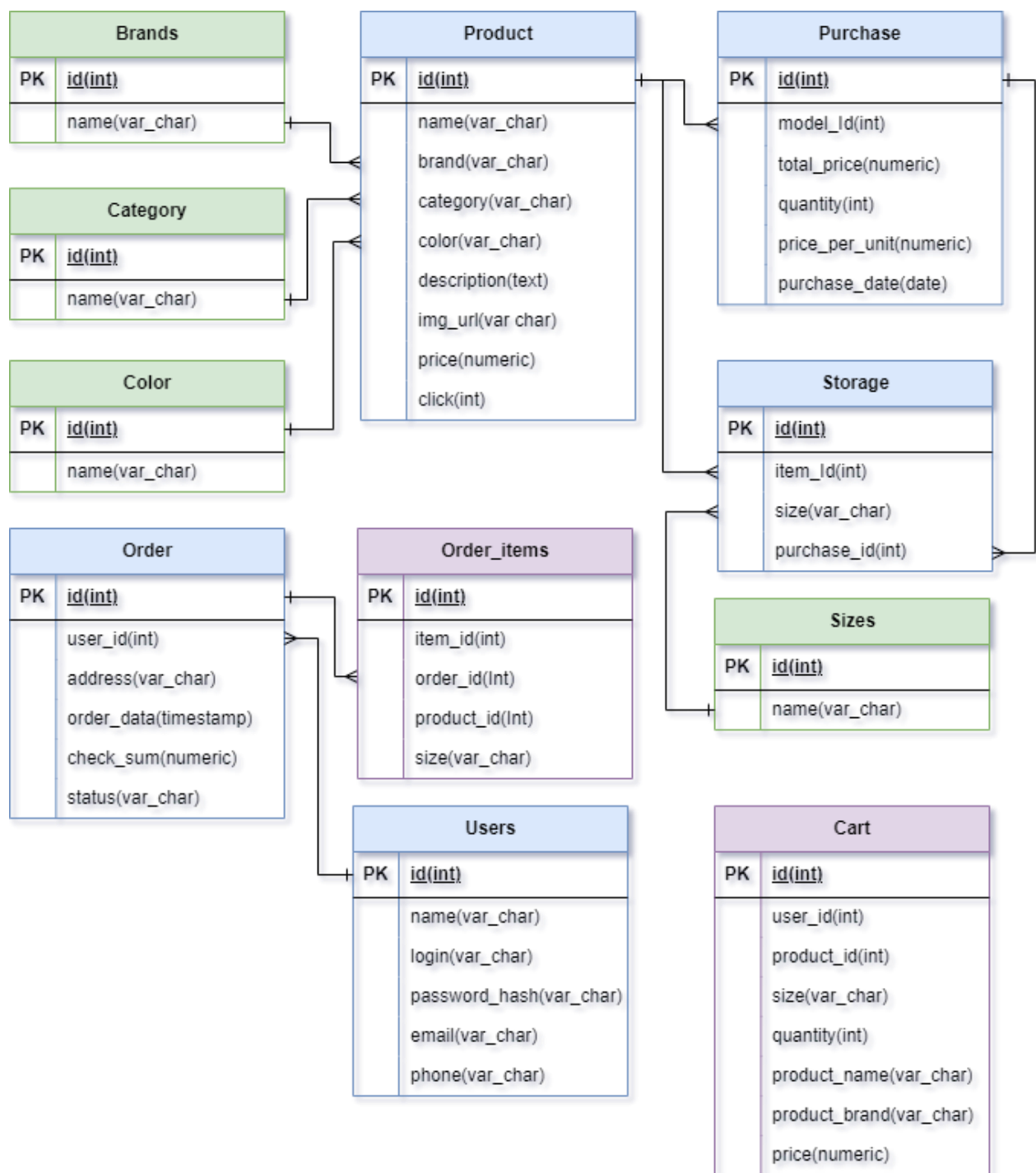


Рис.1. ER-диаграмма

Таблица **Products** определяет структуру таблицы в базе данных, которая содержит информацию о продуктах. Каждая строка этой таблицы представляет отдельный продукт и содержит информацию о его идентификаторе (поле id), названии (name), бренде (brand), категории (category), цвете (color), описании (description), URL-адресе изображения (img_url), цене (price) и количестве кликов (click).

Поле **brand** является внешним ключом на таблицу **Brands**, где содержится информация о брендах продуктов, а поля **category** и **color** также являются внешними ключами на таблицы **Category** и **Color**, соответственно.

Эта таблица используется для хранения списка продуктов в базе данных интернет-магазина. В приложении можно использовать данные из этой таблицы для отображения каталога товаров, для поиска продуктов по категориям, цвету или бренду.

Код создания таблицы **Products** (см. Приложение 7).

Таблица **Purchase** - определяет структуру таблицы в базе данных, используется для учета закупок интернет-магазина.

Поле **id** является уникальным идентификатором каждой закупки и автоматически генерируется при каждой новой записи в таблицу.

Поле **product_id** является внешним ключом на таблицу "**Products**", где содержится информация о продуктах.

Поля **total_price**, **quantity** и **price_per_unit** отражают информацию о каждой конкретной закупке.

total_price - общая стоимость, **quantity** - количество продуктов, **price_per_unit** - цена за единицу продукта.

Код создания таблицы **Purchase** (см. Приложение 8).

Таблица **Storage** используется для учета товаров на складе интернет-магазина.

Поле **id** является уникальным идентификатором каждой записи в таблице и автоматически генерируется при каждой новой записи.

Поле **item_id** является внешним ключом на таблицу "**Products**", где содержится информация о продуктах.

Поле **size** указывает на размер товара.

Поле **purchase_id** является внешним ключом на таблицу "**Purchase**", где содержится информация о закупках товаров.

Используя данные этой таблицы, интернет-магазин может отслеживать количество товаров на складе, размеры и другую информацию. Также это может упростить процесс заказа и отправки товаров клиентам, так как интернет-магазин может знать, какой товар и в каком количестве лежит на складе в данный момент времени.

Код создания таблицы Storage (см. Приложение 9).

таблица **Cart** используется для хранения информации о товарах, которые были добавлены в корзину покупателем в интернет-магазине.

Поле `id` является уникальным идентификатором каждой записи в таблице и автоматически генерируется при каждой новой записи.

Поле `user_id` хранит информацию о пользователе, который добавил товар в корзину.

Поле `product_id` содержит идентификатор продукта, который был добавлен в корзину.

Поле `size` указывает на размер продукта.

Поле `quantity` указывает на количество продуктов, добавленных в корзину.

Поля `product_name`, `product_brand` и `price` содержат информацию о названии, бренде и цене продукта.

Используя данные этой таблицы, интернет-магазин может отслеживать, какие продукты и в каком количестве были добавлены в корзину каждым пользователем, что позволяет упростить процесс оформления заказа клиентом. Кроме того, эта таблица может использоваться для отображения истории заказов пользователей.

Код создания таблицы Cart (см. Приложение 10).

Таблица **Order** используется для хранения информации о заказах, которые были сделаны покупателем.

Поле `id` является уникальным идентификатором каждого заказа и автоматически генерируется при каждой новой записи.

Поле `user_id` хранит информацию о пользователе, который сделал заказ.

Поле `address` указывает на адрес доставки заказа.

Поле `order_data` содержит информацию о дате совершения заказа.

Поле `check_sum` указывает на сумму заказа.

Поле `status_name` содержит информацию о статусе заказа.

Используя данные этой таблицы, интернет-магазин может отслеживать информацию о каждом заказе, включая идентификатор пользователя, адрес доставки, сумму заказа и историю изменения статуса заказа. Также эта таблица может быть использована для отображения заказов пользователей и упрощения процесса обработки заказов.

Код создания таблицы `Order` (см. Приложение 11).

таблица **`Order_items`** используется для хранения информации о продуктах в заказе.

Поле `order_id` хранит информацию о заказе, в котором был размещен этот продукт.

Поле `item_id` - уникальный идентификатор каждого продукта в заказе.

Поле `id` является уникальным идентификатором каждого товара в заказе и автоматически генерируется при каждой новой записи.

Поле `product_id` содержит идентификатор продукта, который был добавлен в заказ.

Используя данные этой таблицы, интернет-магазин может отслеживать, какие продукты были включены в каждый заказ, что позволяет упростить процесс обработки заказов и определения, какие товары нужно отправлять клиентам.

Код создания таблицы `Order_items` (см. Приложение 12).

2.2. Дизайн проекта

Так как в этом проекте идет упор на Backend разработку, для демонстрации будет использован простой дизайн с использованием шаблонов, в дальнейшем будет разъяснение, как использовать разработанное API с другим дизайном, на примере с простым React проектом.

Макет был создан с помощью графического редактора Figma, скриншот из которого представлен на рисунке (рис.1), сами макеты представлены на рисунках ниже (рис.3, рис.4, рис.5, рис.6, рис.7, рис.8, рис.9, рис.10).

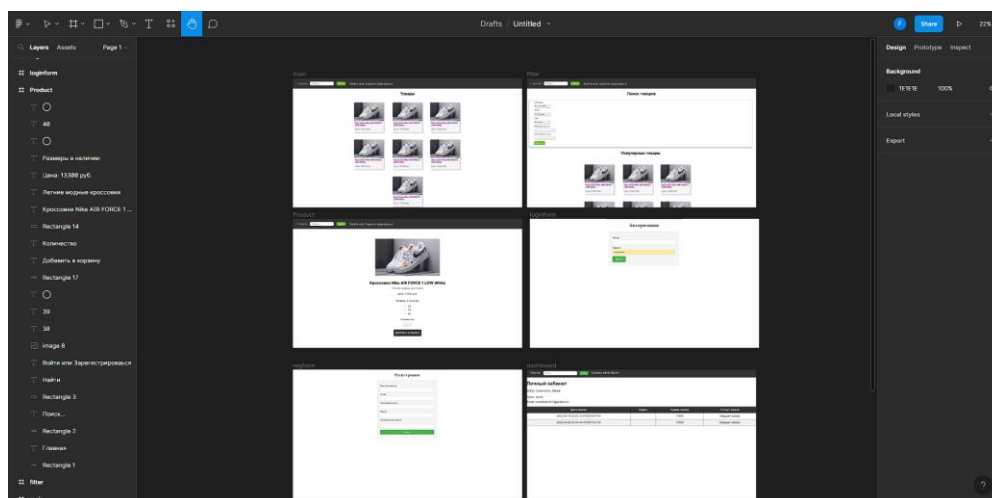


Рис.2. Скриншот из редактора

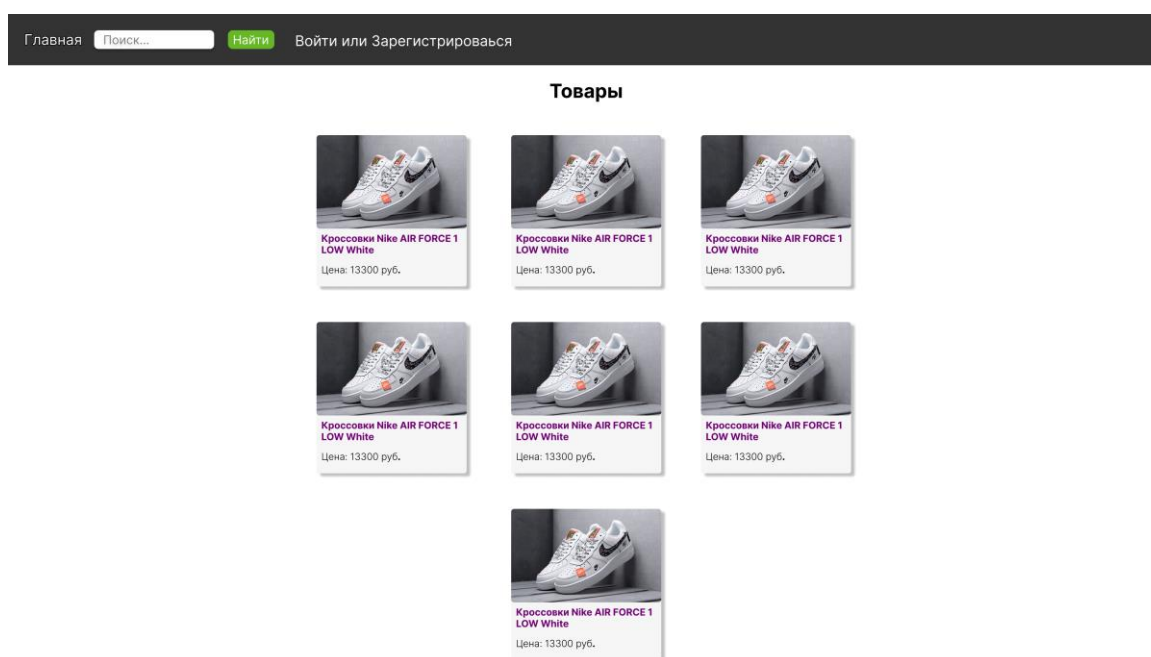


Рис.3. Макет главной страницы

Для создания шаблонов сайта использовался шаблонизатор Jinja2.

Листинг базового шаблона предоставлен в приложении (см. Приложение 13).

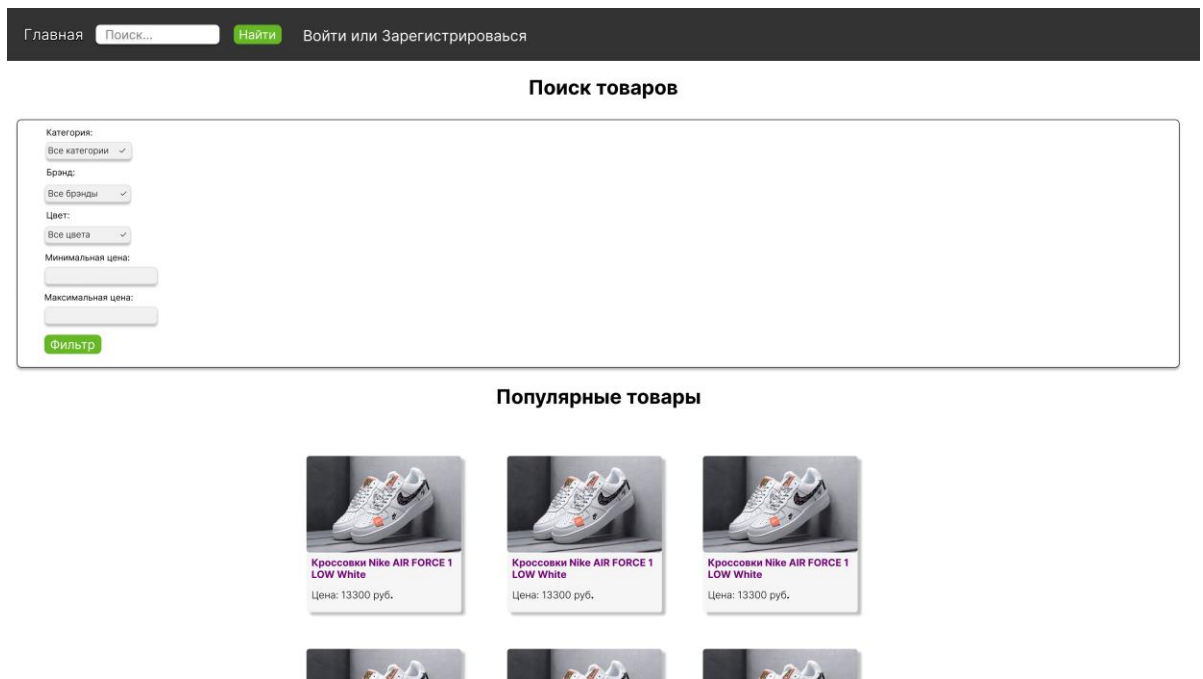


Рис.4. Макет страницы с фильтрацией

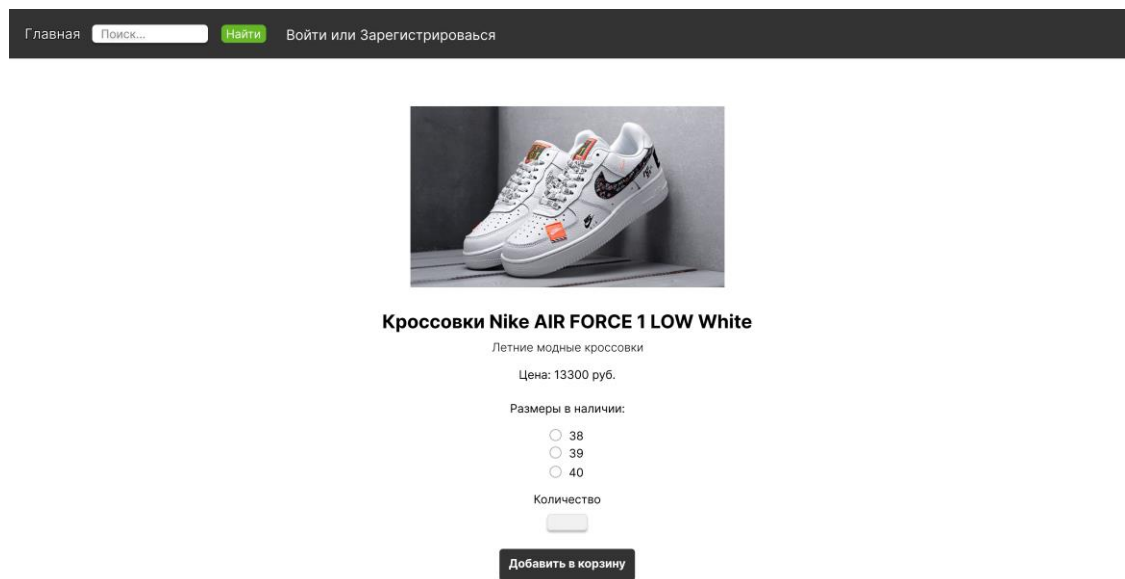


Рис.5. Макет страницы товара

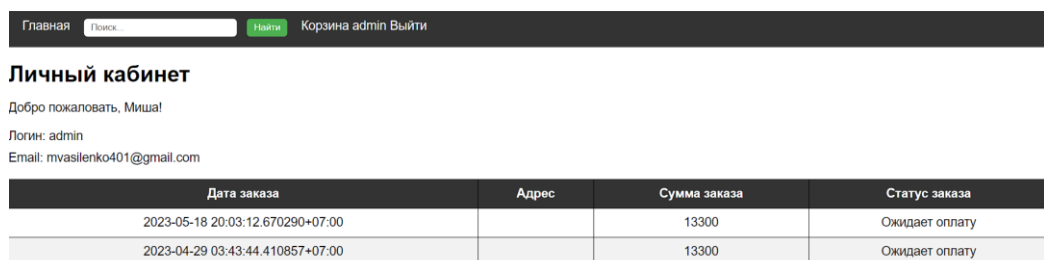


Рис.6. Макет личного кабинета

[Главная](#)

[Найти](#)
[Корзина admin](#)
[Выйти](#)

Корзина товаров

Название товара	Размер	Количество	Цена	Сумма	Наличие
Общая стоимость:	0 руб.	Корзина пуста или не все товары доступны.			

Рис.7. Макет корзины товаров

Регистрация

Имя пользователя:

Логин:

Электронная почта:

Пароль:

Подтверждение пароля:

Register

Рис.8. Макет формы регистрации

Авторизация

Логин:

Пароль:

Войти

Рис.9. Макет формы авторизации

Оформление заказа

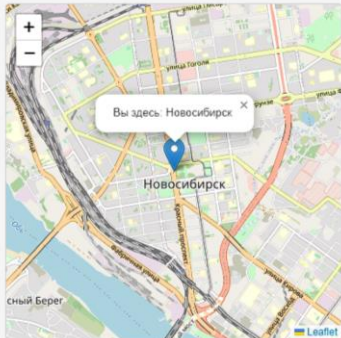
Телефон

Адрес

Квартира

+

−



Оформить заказ

Рис.10. Макет формы заказа

Существует множество геосервисов, которые можно использовать на сайте магазина: Google Maps API, Bing Maps API, Mapbox, Here Maps, OpenStreetMap.

Каждый из этих сервисов имеет сильные и слабые стороны. Например, Google Maps API обладает обширными функциональными возможностями, но стоимость использования может быть слишком высокой для многих проектов. Bing Maps API может быть более доступным с точки зрения стоимости, но может быть менее точным и не иметь такого большого сообщества разработчиков.

В проекте был выбран OpenStreetMap, перечислим ниже обоснования выбора:

1. Это бесплатный геосервис, что может значительно снизить стоимость проекта.
2. OpenStreetMap является проектом с открытыми исходными кодами, что позволяет разработчикам свободно использовать API и расширять его функциональность.
3. OpenStreetMap имеет широкое сообщество разработчиков, которые постоянно вносят изменения и улучшения, что гарантирует высокую точность и актуальность карт.

4. OpenStreetMap имеет подробные карты многих регионов мира, в том числе и тех, где другие геосервисы могут быть менее полными.
5. OpenStreetMap имеет большую гибкость настройки, что позволяет интегрировать карты и их функции в ваш сайт точно так, как вам это нужно.

Таким образом, выбор OpenStreetMap позволяет существенно сэкономить деньги и сделать разработку вашего интернет-магазина менее затратной, а также обеспечить высокую точность и актуальность карт.

Для реализации функциональной карты в форме заказа был написан Java Script код (см. Приложение 14).

Этот код отвечает за определение местоположения пользователя, отображение его на карте, обработку кликов и получение информации о местоположении, соответствующем выбранной точке.

- 1) `var map = L.map('mapid').setView([55.028554, 82.920532], 13);` - создает экземпляр карты и добавляет ее на веб-страницу в элемент с `id="mapid"`. `setView` задает центральную точку карты и масштаб (в данном случае 13);
- 2) `var previousMarker = null;` - создает переменную `previousMarker` и устанавливает ее значение `null`. Это позволяет контролировать, был ли нарисован предыдущий маркер на карте;
- 3) `L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', { maxZoom: 18, }).addTo(map);` - добавляет слой карты (тайл-слой) на карту, используя URL-адрес плиток OpenStreetMap. `maxZoom: 18` ограничивает максимальный размер масштабирования карты;
- 4) `if ('geolocation' in navigator) {` - проверяет, поддерживает ли браузер геолокацию;
- 5) `navigator.geolocation.getCurrentPosition(function(position) {` - получает текущее местоположение пользователя;
- 6) `var lat = position.coords.latitude;` и `var lon = position.coords.longitude;` - сохраняет широту и долготу местоположения пользователя;

- 7) `var url = 'https://nominatim.openstreetmap.org/reverse?format=json&lat=' + lat + '&lon=' + lon + '&zoom=10';` - формирует URL-адрес для получения информации о городе, соответствующем местоположению пользователя, используя координаты местоположения;
- 8) `fetch(url)` - выполняет HTTP-запрос, чтобы получить информацию о городе;
- 9) `.then(function(response) { return response.json(); })` - преобразует ответ в формате JSON в объект JavaScript;
- 10) `.then(function(data) {` - обрабатывает полученные данные;
- 11) `var city = data.address.city || data.address.town || data.address.village || data.address.hamlet || data.address.locality;`
- 12) `|| data.address.suburb || data.address.district || data.address.county || data.address.region || data.address.state || data.address.country;` - получает данные об адресе в определенном городе;
- 13) `L.marker([lat, lon]).addTo(map).bindPopup('Вы здесь: ' + city).openPopup();` - добавляет маркер на карту, указывая координаты местоположения пользователя, и открывает всплывающее окно с названием города;
- 14) `map.setView([lat, lon], 13);` - устанавливает масштаб карты на определенный уровень и центрирует карту на местоположении пользователя;
- 15) таким образом, этот код позволяет определить местоположение пользователя и отобразить его на карте, а также показать информацию о городе, соответствующем местоположению;
- 16) `function onMapClick(e) {` - определяет функцию обработки кликов на карте с координатами, переданными в параметре `e`;
- 17) `document.getElementById('adres').value = '';` - очищает элемент с идентификатором 'adres', если в нем был введен текст;
- 18) `if (previousMarker) { map.removeLayer(previousMarker); }` - удаляет предыдущую метку, если она существует;

- 19) `var url = 'https://nominatim.openstreetmap.org/reverse?format=json&lat=' + e.latlng.lat + '&lon=' + e.latlng.lng;` - формирует URL-адрес для получения информации о местоположении, соответствующем выбранной точке на карте, используя координаты выбранной точки;
- 20) `fetch(url)` - выполняет HTTP-запрос, чтобы получить информацию о местоположении;
- 21) `.then(function(response) { return response.json(); })` - преобразует ответ в формате JSON в объект JavaScript;
- 22) `.then(function(data) {` - обрабатывает полученные данные;
- 23) `document.getElementById('adres').value = data.display_name;` - выводит текстовый адрес в поле "Адрес";
- 24) `var marker = L.marker(e.latlng).addTo(map); previousMarker = marker;` - добавляет маркер на карту в выбранной точке и сохраняет его в переменной `previousMarker`, чтобы удалить его при следующем клике на карте;
- 25) `map.on('click', onMapClick);` - устанавливает обработчик событий 'click' для карты, вызывая нашу функцию `onMapClick` при каждом клике на карте;
- 26) таким образом, этот код позволяет обрабатывать клики на карте, получать информацию о местоположении, соответствующем выбранной точке, и выводить эту информацию в поле "Адрес".

2.3. Функциональное описание разработки

Для комфортной работы с Flask важна правильная организация файлов, в теории все функции можно расписать в одном файле, но это станет проблемой, когда другой программист попытается разобраться в этом коде. Организация файлов в данном проекте предоставлена на рисунке (рис.11).

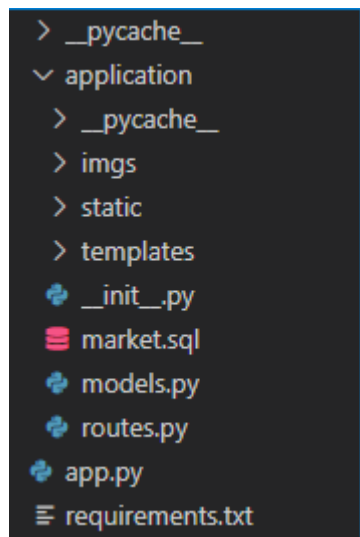


Рис.11. Организация файлов в проекте

Файл «__init__.py» нужен для инициализации Flask приложения, базы данных и логин менеджера.

«market.sql» - выгрузка базы данных.

В файле «models.py» описание моделей БД.

Файл «routes.py» содержит маршруты приложения.

Файл «app.py» нужен для запуска самого приложения.

В файле «requirements» хранится список всех модулей и пакетов Python, которые нужны для полноценной работы программы.

В папке «templates» находятся html шаблоны, а в папке «static» стили и скрипты.

Инициализация – это процесс подготовки к работе, в данном случае этим можно назвать «подключением» базы данных к проекту.

Код инициализации:

```
from flask import Flask
```

```
from flask_sqlalchemy import SQLAlchemy
```

```
from flask_login import LoginManager
```

```
from flask_migrate import Migrate
```

```
from flask_migrate import upgrade as _upgrade
```

```
app = Flask(__name__)
```

```
app.config['SQLALCHEMY_DATABASE_URI'] =  
'postgres://postgres:123@localhost/market'  
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False  
app.secret_key = 'secretkey'  
db = SQLAlchemy(app)  
manager = LoginManager(app)
```

Помимо БД и Flask тут же инициализирован LoginManager, он нужен для работы с расширением Flask-Login, для функционала аутентификации пользователей.

«app.py» - нужен для запуска Flask приложения.

Код инициализации:

```
from application import app  
from application.routes import *  
  
if __name__ == '__main__':  
    app.run()
```

ORM или Object Relational Mapper (объектно-реляционное отображение) позволяет работать с базой данных с помощью объектно-ориентированного кода, не используя SQL-запросы. Благодаря этой технологии можно описать таблицы в виде классов для дальнейшей работы. Модели предоставлены в приложении (см Приложение 15).

В файле «routes.py» прописаны страницы веб сайта.

Импорты в маршрутах:

```
from flask import Flask, render_template, request, redirect, url_for, flash, Blueprint  
from flask_login import login_user, logout_user, login_required, current_user  
from flask_sqlalchemy import SQLAlchemy  
from sqlalchemy import func  
from werkzeug.security import generate_password_hash  
from werkzeug.security import check_password_hash  
from application.models import *
```

```
from application import app, db
from sqlalchemy import or_, text, and_
from sqlalchemy.exc import SQLAlchemyError
```

Это импорт нескольких модулей и классов, необходимых для работы приложения на Flask. - Flask используется для создания веб-приложений.

- render_template используется для рендеринга HTML-шаблонов. - request используется для доступа к параметрам запроса.

- redirect используется для перенаправления пользователя на другую страницу. - url_for используется для генерации URL-адресов на основе имен функций.

- flash используется для отображения флэш-сообщений, таких как ошибки или уведомления об успешном выполнении действия.

- Blueprint - это механизм организации кода в Flask, чтобы разбить его на отдельные модули и разделить функциональность приложения на более мелкие куски.

- login_user, logout_user и login_required - используются в сочетании с Flask-Login для регистрации пользователей, их выхода и защиты доступа к некоторым страницам. - SQLAlchemy используется для работы с базами данных.

- generate_password_hash и check_password_hash используются для хеширования и проверки паролей пользователей.

- func используется для доступа к функциям агрегирования базы данных, таким как SUM, AVG, MAX, MIN и т.д.

- or_, and_ и text используются для создания более сложных SQL-запросов.

- SQLAlchemyError используется для обработки ошибок СУБД SQLAlchemy.

Проверка наличия товара, функция **get_size**:

```
# узнать размеры и товары в наличии
def get_size(product_id):
    sizes =
    Storage.query.filter_by(item_id=product_id).with_entities(Storage.size).all()
```

if not sizes:

return "Товара нет в наличии"

else:

return f"Размеры товара: {' '.join([size[0] for size in sizes])}"

Функция `get_size` используется для получения информации о размерах товара и его наличии на складе.

Она принимает идентификатор продукта в качестве аргумента. Затем она использует запрос к таблице `Storage` и фильтрует ее по значению `item_id` (идентификатор продукта), чтобы получить размер товара.

Если размеры продукта не найдены в таблице `Storage`, функция возвращает сообщение "Товара нет в наличии".

Если размеры продукта найдены в таблице `Storage`, функция возвращает список размеров в виде строки, разделенных запятыми.

Функция **`increment_click`** используется для увеличения значения поля `click` для заданного продукта в таблице `Products`:

инкремент кликов для отслеживания популярности товара

def increment_click(product_id):

Обновляем значение поля click для записи где id = product_id

*db.session.query(Products).filter(Products.id ==
product_id).update({Products.click: Products.click + 1})*

Сохраняем изменения в базе данных

db.session.commit()

Она принимает идентификатор продукта и использует запрос для обновления значения поля `click` на 1 для записи с заданным `product_id`.

После этого функция сохраняет изменения в базу данных, вызывая метод `commit` для текущей сессии базы данных.

Этот код используется для отслеживания популярности продуктов, увеличивая значение поля `click` при каждом клике пользователя на продукт.

Функция **index** обрабатывает запросы на главной странице веб-приложения:

```
# главная страница
```

```
@app.route('/')
```

```
def index():
```

```
    brands = Brands.query.all()
```

```
    categorys = Category.query.all()
```

```
    products = Products.query.order_by(Products.click.desc()).all() # сортировка по количеству кликов
```

```
    return render_template('index.html', brands=brands, categorys=categorys, products=products)
```

Она определяет несколько переменных, используя запросы к таблицам базы данных - brands - бренды, categorys - категории и products - продукты.

products сортируется в порядке убывания значений поля click, что позволяет отображать наиболее популярные продукты.

Затем она передает эти переменные в шаблон index.html с помощью функции render_template, которая отображает HTML-шаблон на клиентской стороне.

Функция **product** отображает информацию о товаре на странице с идентификатором product_id:

```
# страница товара
```

```
@app.route('/product/<int:product_id>')
```

```
def product(product_id):
```

```
    increment_click(product_id) # вызываю функцию инкремента кликов
```

```
    # код для получения информации о товаре по его идентификатору
```

```
    product = Products.query.get(product_id)
```

```
    model = Storage.query.filter_by(item_id=product_id).all()
```

```
    sizes = Sizes.query.all()
```

```
    storage = get_size(product_id)
```

```
    size_lst = []
```



```

app.jinja_env.globals['count_available_items'] = count_available_items

for size in sizes:
    if size.name in storage and size.name not in size_lst:
        size_lst.append(size.name)
    size_lst=sorted(size_lst)

return render_template('product.html', product=product, storage=storage,
                        model=model,size_lst=size_lst,count_available_items=count_availa
                        ble_items)

```

Она принимает `product_id` в качестве аргумента и вызывает функцию `increment_click`, чтобы увеличить количество кликов для данного продукта.

Затем она использует запросы к таблицам базы данных, чтобы получить информацию о продукте, его наличии на складе и доступных размерах.

- `product` - информация о продукте из таблицы `Products`.
- `model` - модель товара в наличии из таблицы `Storage`.
- `sizes` - доступные размеры из таблицы `Sizes`.
- `storage` - информация о наличии товаров и их размерах из таблицы `Storage`.

Затем она использует цикл `for` для создания списка доступных размеров для данного продукта, используя строку `storage` и список доступных размеров из таблицы `Sizes`.

Наконец, она передает эти переменные в шаблон HTML `product.html`, который будет отображен на странице.

Функция **`add_to_cart`** используется для добавления товара в корзину покупателя:

```

# функция добавления товара в корзину
@app.route('/add_to_cart', methods=['POST'])
@login_required
def add_to_cart():

```

```

# Получаем данные товара из запроса
prod_id = request.form['prod_id']
prod_price = request.form['prod_price']
prod_name = request.form['prod_name']
prod_brand = request.form['prod_brand']
size_inf = request.form['size_inf']
quantity_inf = request.form['quantity_inf']

# Проверяем, есть ли товар уже в корзине
cart_item = Cart.query.filter_by(user_id=current_user.id, product_id=prod_id,
size=size_inf).first()
if cart_item:
    # Если товар уже есть в корзине, увеличиваем его количество на указанное
    значение
    cart_item.quantity += int(quantity_inf)
else:
    # Если товара еще нет в корзине, добавляем его в корзину
    cart_item = Cart(user_id=current_user.id, product_id=prod_id, size=size_inf,
        quantity=int(quantity_inf), price=prod_price,
product_name=prod_name,
        product_brand=prod_brand)
    db.session.add(cart_item)

# Сохраняем изменения в базе данных
db.session.commit()

flash('Товар добавлен в корзину')
return redirect(url_for('cart'))

```

Функция принимает POST-запрос от формы, отправленной на странице товара, с данными о продукте и количестве. Эта информация извлекается из

запроса, используя объект `request.form`. Затем функция проверяет, есть ли товар уже в корзине на основании `id` пользователя, `id` продукта и `size`. Если товар уже есть в корзине, количество товара увеличивается на указанное значение. Если товар еще не добавлен в корзину, создается новый объект `Cart` с информацией о товаре и добавляется в базу данных. После этого функция сохраняет изменения в базе данных и перенаправляет пользователя на страницу корзины.

Функция **cart** используется для отображения корзины покупателя:

```
# страница корзины
@app.route('/cart', methods=['POST', 'GET'])
@login_required
def cart():
    cart_items = Cart.query.filter_by(user_id=current_user.id).all()
    total=total_price()
    all_available = all(count_available_items(item.size, item.product_id) >=
item.quantity for item in cart_items)
    # Добавляем функцию count_available_items в глобальный контекст шаблона
    app.jinja_env.globals['count_available_items'] = count_available_items

    return render_template('cart.html',cart_items=cart_items,
total=total,all_available=all_available)
```

Функция использует запрос к таблице `Cart` для получения списка элементов корзины пользователя, а также вычисляет общую стоимость всех предметов в корзине.

Функция также определяет, все ли предметы из корзины доступны в наличии, используя функцию `all`, которая проверяет, что для каждого элемента корзины количество товаров, доступных в заданном размере, больше или равно запрошенному количеству. Затем функция передает информацию о корзине, общей стоимости и доступности товаров на страницу `cart.html` с помощью функции `render_template`. Функция **remove_from_cart** используется для удаления продукта из корзины. Функция принимает `cart_id` в качестве аргумента.

Затем она использует `cart_id` для получения информации о продукте из таблицы `Cart` и удаляет этот элемент из базы данных с помощью `db.session.delete(cart_item)`. После этого функция сохраняет изменения в базе данных и перенаправляет пользователя на страницу корзины. В этой функции также используется функция `flash`, чтобы отобразить сообщение об успешном удалении товара из корзины.

Функция `decrease_cart_item_quantity`:

```
# декремент количества товара
@app.route('/decrease_cart_item_quantity/<int:cart_id>', methods=['POST'])
def decrease_cart_item_quantity(cart_id):
    cart_item = Cart.query.get(cart_id)
    if cart_item:
        if cart_item.quantity > 1:
            cart_item.quantity -= 1
            db.session.commit()
        else:
            db.session.delete(cart_item)
            db.session.commit()
    return redirect(url_for('cart'))
```

Эта функция используется для уменьшения количества товаров определенного элемента корзины на единицу.

Она принимает `cart_id` в качестве аргумента, используя `cart_id`, функция получает информацию о продукте из таблицы `Cart`. Если элемент существует в корзине и его количество больше 1, функция уменьшает количество товаров элемента на 1 и сохраняет изменения базы данных. Если количество товаров элемента равно 1, элемент удаляется из корзины. После выполнения любого из этих действий, функция перенаправляет пользователя на страницу корзины.

Функция `increase_cart_item_quantity`:

```
# инкремент количества товара
@app.route('/increase_cart_item_quantity/<int:cart_id>', methods=['POST'])
```

```
def increase_cart_item_quantity(cart_id):
    cart_item = Cart.query.get(cart_id)
    if cart_item:
        cart_item.quantity += 1
        db.session.commit()
    return redirect(url_for('cart'))
```

Эта функция используется для увеличения количества товаров определенного элемента корзины на единицу. Она принимает `cart_id` в качестве аргумента, используя `cart_id`, функция получает информацию о продукте из таблицы `Cart`. Если элемент существует в корзине, функция добавляет 1 к количеству товаров элемента и сохраняет изменения базы данных. После этого функция перенаправляет пользователя на страницу корзины.

Функция **`count_available_items`** используется для подсчета количества товаров выбранной модели определенного размера, которые доступны для покупки в базе данных:

```
# подсчет количества товара одной модели одного размера в БД
def count_available_items(size, item_id):
    # Получаем список item_id из таблицы Order_items
    ordered_items = [item.item_id for item in Order_items.query.all()]
    # Считаем количество записей в таблице Storage, удовлетворяющих
    заданным условиям
    count = Storage.query.filter_by(size=size,
    item_id=item_id).filter(Storage.id.notin_(ordered_items)).count()
    return count
```

Функция принимает `size` и `item_id` в качестве аргументов. Затем она использует запрос к таблице `Storage`, чтобы получить количество наличия товара с заданным `size` и `item_id`.

Функция использует метод `not_in_`, чтобы исключить товары, которые уже были заказаны в прошлом, на основании списка `item_id`, полученного из таблицы `Order_items`.

На выходе функция возвращает количество доступных товаров в указанном размере.

Функция `total_price`:

подсчет полной стоимости корзины

def total_price():

cart_items = Cart.query.filter_by(user_id=current_user.id).all()

total = 0

for item in cart_items:

*total = total + item.price * item.quantity*

return total

Эта функция используется для вычисления общей стоимости элементов корзины пользователя.

Функция получает информацию о корзине пользователя, используя запрос к таблице `Cart`, и затем суммирует цену продукта, умноженную на его количество для каждого элемента корзины с помощью цикла `for`.

На выходе функция возвращает общую стоимость всех элементов корзины пользователя.

Функция `register`:

регистрация

@app.route('/register', methods=['GET', 'POST'])

def register():

if current_user.is_authenticated:

return redirect(url_for('logout'))

if request.method == 'POST':

name = request.form['name']

login = request.form['login']

password = request.form['password']

```

confirm_password = request.form['confirm_password']
email = request.form['email']
user = User.query.filter((User.login==login) | (User.email==email)).first()

if password != confirm_password:
    flash('Пароли не совпадают')
elif user:
    flash('Логин или Email заняты')
else:
    new_user = User(name=name, login=login,
password_hash=generate_password_hash(password), email=email)
    db.session.add(new_user)
    db.session.commit()
    flash('Вы успешно зарегистрировались!')
    return redirect(url_for('login'))
return render_template('register.html')

```

Эта функция используется для регистрации нового пользователя. Сначала проверяется, аутентифицирован ли пользователь уже на сервере.

Если пользователь уже вошел в систему, он перенаправляется на страницу выхода. Затем проверяется метод запроса, и если он равен «POST», информация из формы получается с помощью объекта request.form.

Затем функция проверяет введенные пользователем пароли на соответствие. Если пароли не совпадают, функция возвращает сообщение об ошибке.

Затем функция пробует найти пользователя в таблице User.

Если пользователь уже существует, функция возвращает сообщение об ошибке.

В противном случае функция создает нового пользователя в таблице User с помощью модели User, хешируя пароль пользователя с помощью generate_password_hash. Добавление нового пользователя в базу данных

осуществляется с помощью `db.session.add` и сохранения изменений в базе данных с помощью `db.session.commit`.

После успешной регистрации пользователь перенаправляется на страницу входа. Если метод запроса равен «GET», функция возвращает страницу `register.html` для заполнения пользователем.

Функция **login**:

```
# авторизация
@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('dashboard'))
    if request.method == 'POST':
        login = request.form['login']
        password = request.form['password']

        user = User.query.filter_by(login=login).first()

        if user and check_password_hash(user.password_hash, password):
            login_user(user)
            flash('Вы успешно авторизовались!', 'success')
            return redirect(url_for('dashboard'))
        else:
            flash('Ошибка авторизации', 'danger')

    return render_template('login.html')
```

Эта функция используется для авторизации пользователя.

Сначала проверяется, аутентифицирован ли пользователь уже на сервере. Если пользователь уже вошел в систему, он перенаправляется на страницу с панелью управления.

Затем проверяется метод запроса, и если он равен «POST», информация из формы получается с помощью объекта `request.form`. Затем функция пытается найти пользователя по логину в таблице `User`. Если пользователь найден, проверяется, соответствует ли введенный пользователем пароль хешу пароля пользователя. Если проверка успешна, пользователь авторизуется на сайте и перенаправляется на страницу панели управления. В противном случае пользователь получает сообщение об ошибке.

Если метод запроса равен «GET», функция возвращает страницу `login.html` для ожидания ввода пользователем логина и пароля.

Функция **logout**:

#выход и учетки

@app.route('/logout')

@login_required

def logout():

logout_user()

return redirect(url_for('index'))

Эта функция используется для выхода пользователя из учетной записи.

Функция сначала проверяет, вошел ли пользователь в систему. Если пользователь вошел в систему, функция вызывает функцию `logout_user` из `Flask-Login`, чтобы завершить сеанс пользователя.

Затем функция перенаправляет пользователя на главную страницу (страницу входа или страницу с продуктами). Если пользователь не вошел в систему, функция просто перенаправляет его на главную страницу.

Функция **dashboard**:

#личный кабинет

@app.route('/dashboard')

@login_required

def dashboard():

*orders = Order.query.filter(Order.user_id ==
current_user.id).order_by(Order.id.desc()).all()*

```
return render_template('dashboard.html', orders=orders)
```

Эта функция используется для отображения личного кабинета пользователя.

Функция также содержит запрос к таблице "Order", чтобы получить список заказов пользователя. Для этого используется метод `filter` из объекта `Order`, метод `order_by` для сортировки заказов по убыванию идентификаторов, и метод `all()` для получения списка всех заказов.

Затем функция передает список заказов в шаблон HTML `dashboard.html` с помощью функции `render_template`.

Функция **`add_order`** используется для добавления нового заказа:

```
# добавление заказа
```

```
@app.route('/add_order', methods=['POST', 'GET'])
```

```
def add_order():
```

```
    if request.method == 'POST':
```

```
        apart = request.form['apart']
```

```
        address = ' '.join([request.form['address'], 'квартира', apart])
```

```
        phone = request.form['phone']
```

```
        total = total_price()
```

```
        cart= Cart.query.filter_by(user_id=current_user.id).all()
```

```
        # добавляем номер телефона
```

```
        update_user_phone(id=current_user.id, phone=phone)
```

```
        # Создаем объект модели Order
```

```
        new_order = Order(user_id=current_user.id, address=address,
```

```
order_data=datetime.now(), check_sum=total)
```

```
        # Добавляем объект в сессию
```

```
        db.session.add(new_order)
```

```
        # Сохраняем изменения в базе данных
```

```
        db.session.commit()
```

```
return redirect(url_for('dashboard'))  
return render_template('order_form.html')
```

Функция проверяет метод запроса. Если метод POST, функция получает данные из формы (номер телефона, адрес и стоимость) и создает новый заказ в таблице Order с помощью объекта Order.

Функция также использует total_price для подсчета полной стоимости заказа и запрос Cart для получения списка элементов корзины пользователя. Затем функция добавляет новый заказ в базу данных с помощью метода db.session.add и сохраняет изменения в базе данных с помощью db.session.commit. После успешного добавления заказа функция перенаправляет пользователя на страницу личного кабинета. Если метод запроса равен GET, функция возвращает страницу формы заказа.

```
Функция find_storage_id:  
# Функция поиска записи в таблице Storage  
def find_storage_id(item_id, size):  
    # Ищем запись в таблице Storage по item_id и size, с условием что её id не  
    # встречается в Order_items.item_id  
    storage = Storage.query.outerjoin(Order_items, Storage.item_id ==  
    Order_items.item_id).filter(Order_items.item_id == None).first()  
    if storage:  
        # Если запись найдена, возвращаем её id  
        return storage.id  
    else:  
        flash('Товара нет в наличии', 'danger')
```

Эта функция используется для поиска записи в таблице Storage, которая соответствует item_id и size и не была использована в предыдущих заказах.

Функция использует запрос outerjoin для объединения таблиц Storage и Order_items.

Затем функция применяет фильтр, чтобы найти только записи, которые не встречаются в Order_items и имеют заданный item_id и size.

Если запись найдена, возвращается ее id. В противном случае, функция возвращает сообщение об ошибке.

Функция **update_user_phone** используется для обновления поля phone в таблице User.

Функция обновления записи в таблице User

```
def update_user_phone(id, phone):
```

```
    # Получаем объект записи из таблицы User по id
```

```
    user = User.query.filter_by(id=id).first()
```

```
    # Обновляем значение поля phone
```

```
    user.phone = phone
```

```
    # Сохраняем изменения в базе данных
```

```
    db.session.commit()
```

Функция принимает id и phone в качестве аргументов, где id - это идентификатор пользователя. с помощью метода filter_by из объекта User функция получает информацию о пользователе с заданным идентификатором id.

Затем функция обновляет значение поля phone в объекте пользователя и сохраняет изменения в базе данных с помощью db.session.commit().

Функция **search** используется для поиска продуктов в базе данных:

страница поиска

```
@app.route('/search')
```

```
def search():
```

```
    query = request.args.get('query')
```

```
    brands = Brands.query.all()
```

```
    categorys = Category.query.all()
```

```
    colors = Color.query.all()
```

```
    products = Products.query.order_by(Products.click).all()
```

```
    if query:
```

```
        results = Products.query.filter(
```

```

        Products.name.ilike(f'%{query}%') |
        Products.brand.ilike(f'%{query}%') |
        Products.category.ilike(f'%{query}%')
    )

    print(str(Products.query.filter(
        Products.name.ilike(f'%{query}%') |
        Products.brand.ilike(f'%{query}%') |
        Products.category.ilike(f'%{query}%')
    ).statement))

    return render_template('search_results.html', results=results)

```

else:

```

    return render_template('search.html', brands=brands, categorys=categorys,
products=products, colors=colors)

```

Функция начинается с получения запроса `query` с помощью метода `request.args.get`. Затем функция извлекает данные из таблиц `Brands`, `Category` и `Color` и информацию о продуктах из таблицы `Products`.

Если запрос не пустой (то есть пользователь выполнил поиск), функция использует фильтр `ilike` и метод `filter` для поиска продуктов, содержащих заданный запрос в их названии, марке или категории.

Затем функция передает полученные результаты поиска в шаблон HTML `search_results.html` с помощью функции `render_template`. Если поисковый запрос пуст, функция возвращает страницу `search.html`.

Функция **filter** используется для фильтрации продуктов в базе данных:

фильтрация

```
@app.route('/filter')
```

```
def filter():
```

```
    category = request.args.get('category')
```

```
    brand = request.args.get('brand')
```

```

color = request.args.get('color')
min_price = request.args.get('min_price')
max_price = request.args.get('max_price')

filters = []

if category:
    filters.append(Products.category.ilike(f'%{category}%'))
if brand:
    filters.append(Products.brand.ilike(f'%{brand}%'))
if color:
    filters.append(Products.color.ilike(f'%{color}%'))
if min_price:
    filters.append(Products.price >= int(min_price))
if max_price:
    filters.append(Products.price <= int(max_price))

if filters:
    results = Products.query.filter(and_(*filters)).all()
else:
    results = Products.query.all()

return render_template('search_results.html', results=results)

```

Функция начинается с получения значений фильтров и запроса их из формы с помощью метода `request.args.get`. Затем функция создает список фильтров, куда добавляет элементы в зависимости от того, были ли заполнены соответствующие поля формы.

Затем функция использует `query.filter` для фильтрации продуктов в соответствии с заданными фильтрами. Затем функция передает полученные

результаты в шаблон HTML `search_results.html` с помощью функции `render_template`.

Если ни один из фильтров не был задан пользователем, функция возвращает список всех продуктов из таблицы `Products`.

функция **`delete_cart_items`** используется для удаления всех элементов корзины пользователя из таблицы `Cart`:

эта функция удаляет все записи из корзины пользователя

```
def delete_cart_items():
```

```
    Cart.query.filter_by(user_id=current_user.id).delete()
```

```
    db.session.commit()
```

Функция использует метод `delete()`, который предоставляется объектом `Cart`, для удаления всех записей из корзины пользователя, определяемого его `id`.

Затем функция сохраняет изменения в базе данных с помощью `db.session.commit()`.

2.3.1. Конфигурирование

Часто бывает, что у заказчика уже есть пользовательский интерфейс, и он хочет создать RESTful API для обращения к этому интерфейсу через серверную часть проекта.

Для использования данного проекта с другим пользовательским интерфейсом необходимо выполнить модификации, которые позволят ему вернуть данные в формате JSON, а не HTML.

REST (Representational State Transfer) — это стиль архитектуры для разработки веб-сервисов, которые могут быть доступным на удаленном компьютере. RESTful API — это интерфейс программирования приложений, который использует протокол HTTP для запросов и передачи данных.

RESTful API предоставляет ресурсы в формате JSON или XML, и может быть использован для обработки запросов на чтение, запись, обновление и удаление данных. Каждый ресурс имеет уникальный URI (Uniform Resource Identifier), который может быть использован для доступа к нему.

RESTful API использует общие методы HTTP (GET, POST, PUT и DELETE) для управления ресурсами. GET используется для чтения данных, POST для создания новых записей, PUT для обновления существующих записей и DELETE для удаления записей.

RESTful API является очень популярным подходом для создания веб-сервисов, которые могут быть использованы как веб-приложениями, так и мобильными приложениями. Он является гибким и масштабируемым решением, которое может быть использовано во многих разных сценариях.

Для примера покажу как использовать REST с React приложением, созданным в образовательных целях.

Класс ProductEncoder представляет собой классификатор JSON, который позволяет переопределить функцию default для модели Product.

```
class ProductEncoder(json.JSONEncoder):
```

```
    def default(self, obj):
```

```
        if not isinstance(obj.__class__, DeclarativeMeta):
```

```
            return super(ProductEncoder, self).default(obj)
```

```
    fields = {}
```

```
    for field in [x for x in dir(obj) if not x.startswith('_') and x != 'metadata']:
```

```
        data = obj.__getattr__(field)
```

```
        try:
```

```
            json.dumps(data)
```

```
            fields[field] = data
```

```
        except TypeError:
```

```
            fields[field] = None
```

```
    # a json-encodable dict
```

```
    return fields
```


Класс определяет функцию `default`, которая возвращает словарь, содержащий данные из модели `Product`. Функция обрабатывает все атрибуты класса, и если атрибут может быть сериализован в JSON, он добавляется в словарь.

Для атрибутов, которые не могут быть сериализованы в JSON, возвращается `None`.

Класс `ProductEncoder` может быть использован для преобразования объектов из базы данных типа `Product` в формат JSON.

Вместо класса `ProductEncoder` можно использовать встроенный метод модуля `json` - `json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)`.

Пример:

```
from flask import jsonify  
from sqlalchemy.ext.declarative import DeclarativeMeta
```

```
@app.route('/products')  
def get_products():  
    products = Product.query.all()  
    return jsonify(products)
```

И с использованием ранее описаного ProductEncoder:

```
@app.route('/')  
def index():  
    products = Products.query.order_by(Products.clicks.desc()).all() # Получаем  
продукты, отсортированные по кликам  
    return json.dumps([product for product in products], cls=ProductEncoder), 200,  
{'Content-Type': 'application/json'} # Возвращаем данные в формате JSON
```

Этот код представляет функцию представления (view function), которая отображает главную страницу приложения.

Функция использует метод `query.order_by` для получения списка продуктов из таблицы `Products`, отсортированных по количеству кликов на продукт.

Затем функция использует модуль `json` для преобразования списка продуктов в формат JSON, используя класс `ProductEncoder`. Возвращается JSON строковое представление этого списка продуктов, а также статус 200 OK и заголовок с типом контента `application/json`.

Клиентский код может использовать данные, возвращаемые этой функцией, для отображения списка продуктов на главной странице приложения.

Для получения этих данных в React приложении используется ниже предоставленный код:

```
useEffect(() => {  
  axios.get('http://localhost:5000/')  
    .then((response) => {  
      setItems(response.data);  
    });  
}, []);
```

Этот код использует хук `useEffect` в React, чтобы получить данные из API.

Хук `useEffect` позволяет выполнять побочные эффекты в функциональных компонентах React. В данном случае, он используется для выполнения HTTP-запроса, используя метод `axios.get` для получения данных из API.

Когда компонент отрисовывается в первый раз, хук `useEffect` вызывает функцию, которая выполняет запрос, и сохраняет полученные данные в локальном состоянии с помощью функции `setItems`.

Второй аргумент функции `useEffect` - это массив зависимостей. Если значение в этом массиве не изменяется между рендерами, то эффект вызывается только один раз при первом рендере. В данном случае, массив зависимостей пуст, что означает, что эффект будет вызван только один раз при первом рендере.

```
{products.map((product) => (  
  <Card  
    key={product.id}
```

```

    title = { product.product_type + ' ' + product.gender + ' ' + product.brand + '
' + product.model_name}
    price = {product.price}
    imageUrl = {product.img_url}
    onFavorite={() => console.log('добавили в избранное')}
    onPlus={(obj) => onAddToCart(obj)}
  />
  )))}

```

Этот код использует метод `map` для отображения массива продуктов в компоненты `Card` в `React`.

`map` - это метод массива `JavaScript`, который позволяет пройти по каждому элементу массива и вернуть новый массив, содержащий измененные элементы. В данном случае, метод `map` выполняется на массиве `products`, который содержит данные, полученные из `API`.

Внутри метода `map` каждый элемент массива `products` обрабатывается функцией, которая возвращает новый компонент `Card`. Каждый компонент содержит информацию о продукте, включая название, цену и изображение. Также в компоненте есть обработчики, которые вызывают функции `onFavorite` и `onPlus`, передавая в качестве аргументов соответствующие данные.

Здесь также используется атрибут `key`, который помогает `React` оптимизировать производительность компонентов. Атрибут `key` должен быть уникальным для каждого элемента массива. В данном случае, уникальным идентификатором является `product.id`.

Запускается сразу серверная и клиентская часть, скриншот на рисунке (рис.12).

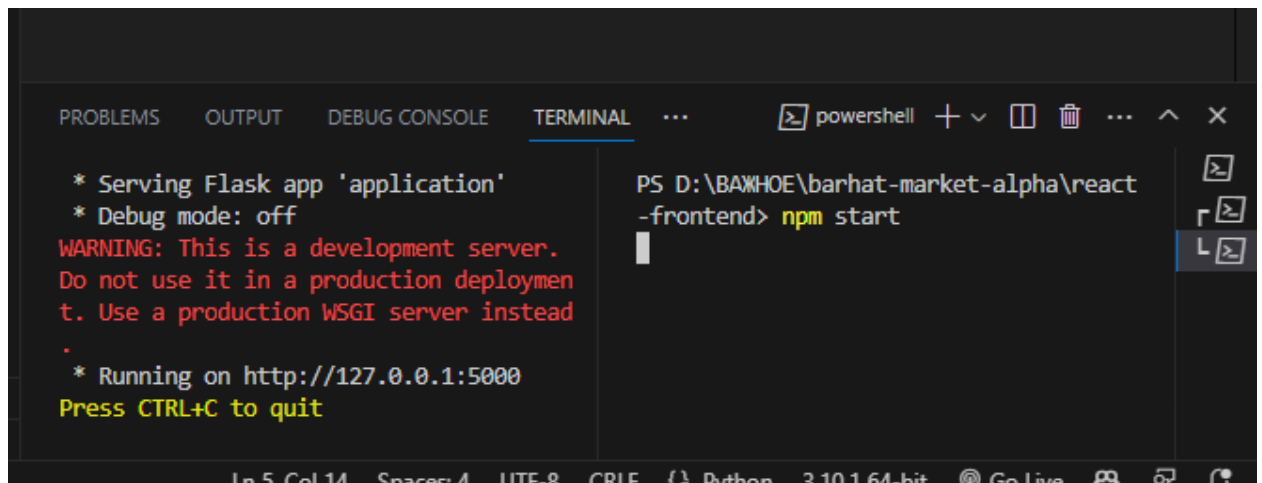


Рис.12. одновременный запуск React и Flask приложений

И теперь на главной странице отображаются товары из БД этого проекта, скриншот на рисунке (рис.13).

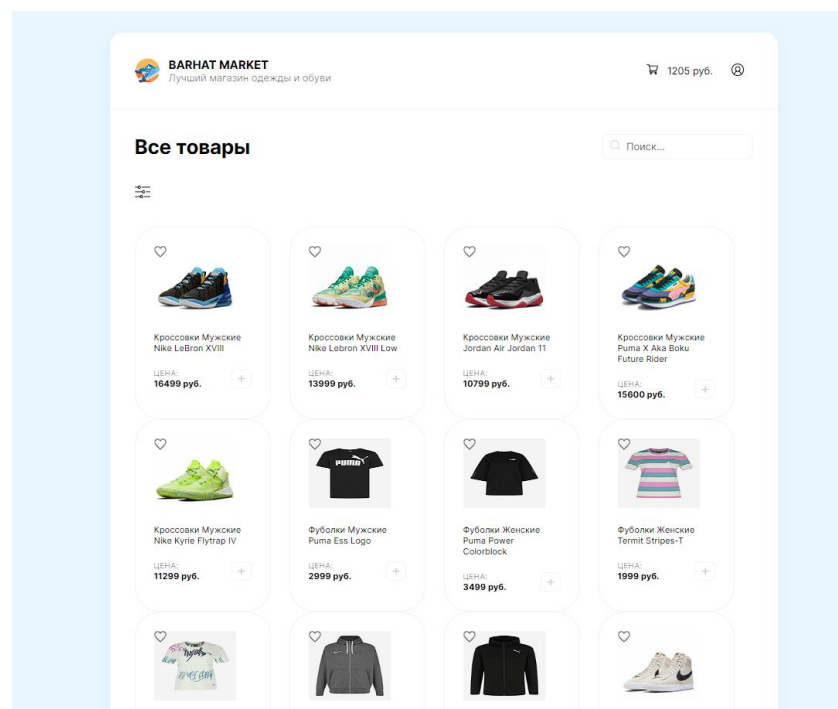


Рис.13. Скриншот главной страницы проекта из примера

Существуют и другие архитектуры стилей API:

1. REST — это простая и гибкая архитектура для создания веб-сервисов, которая использует HTTP-запросы для передачи сообщений между клиентом и сервером. REST хорошо подходит для создания веб-сайтов, которые должны обслуживать большое

количество клиентов, и для которых гибкость и масштабируемость являются ключевыми показателями.

2. SOAP — это более сложная архитектура, которая также используется для создания веб-сервисов, но использует XML для описания контента и затем использует HTTP для передачи сообщений. SOAP менее гибкая и масштабируемая, что делает его менее популярным среди разработчиков. Однако, SOAP имеет свои преимущества, например, встроенную поддержку безопасности. GraphQL - это относительно новая архитектура стиля API, которая становится все более популярной среди разработчиков.
3. GraphQL позволяет клиентам самостоятельно определять формат ответа на запросы и запрашивать только те данные, которые им нужны, а не все данные, которые предоставляет сервер. Это делает GraphQL более эффективной архитектурой для создания мобильных приложений и других клиентских приложений.

Каждая из этих архитектур имеет свои преимущества и недостатки, и выбор между ними зависит от требований проекта и задач, которые они должны выполнять. REST хорошо подходит для создания веб-сайтов с большим количеством клиентов, GraphQL подходит для создания клиентских приложений, а SOAP может быть полезен для приложений, требующих высокой степени безопасности.

2.3.2. Платежные системы

Для использования платежных систем и интеграции их на сайт, необходимо оформить документы и пройти регистрацию у соответствующих платежных сервисов.

Как правило, нужно предоставить такие документы, как паспортные данные, данные о компании, выписку из ЕГРЮЛ, свидетельство о регистрации, ИНН, ОГРН и т.д.

Кроме того, нужно подписать договор на использование платежной системы и ознакомиться с условиями использования и комиссиями за проведение платежей.

Подробнее об оформлении документов и условиях использования платежных систем можно узнать на сайтах самих систем и в их документации или обратившись к консультантам.

Выше были перечислены причины отказа от интеграции системы оплаты, далее будет описано как интегрировать платежную систему на примере Сбербанк Эквайринга.

Один из способов - использование API, которое позволяет осуществлять прием платежей с помощью создания соответствующих запросов. Ниже приведен пример интеграции Сбербанк Эквайринга, с использованием Python и Flask.

Сначала необходимо зарегистрироваться в Сбербанк Эквайринге и получить идентификатор магазина и секретный ключ.

- Установить библиотеку для работы с Сбербанк Эквайрингом с помощью команды:

```
pip install sberbank-acquiring
```

Инициализировать переменные окружения:

```
import os
```

```
os.environ['SBERBANK_POS_ID'] = 'ВАШ ИДЕНТИФИКАТОР МАГАЗИНА'
```

```
os.environ['SBERBANK_POS_SECRET'] = 'ВАШ СЕКРЕТНЫЙ КЛЮЧ'
```

- Создание функции для генерации платежной формы на сайте:

```
@app.route('/payment', methods=['POST'])
```

```
def payment():
```

```
    form_data = request.form
```

```
    amount = float(form_data.get('amount'))
```

```
    description = form_data.get('description')
```

```
    api = AcquiringApi()
```

```
params = RegisterParams(  
    amount=int(amount * 100),  
    order_number='123',  
    description=description,  
    return_url='http://localhost:5000/success'  
)  
response = api.register(params)  
return render_template('payment.html', form_url=response.form_url)  
@app.route('/success')  
def success():  
    return 'Payment successful!'
```

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе был разработан сайт магазина одежды и обуви, с использованием языка программирования Python и фреймворка Flask. Были выполнены задачи по проектированию и реализации бэкэнд-части веб-приложения, которое включает в себя работу с базами данных, формами и запросами.

Основной целью проекта была разработка сайта магазина одежды и обуви, который позволял бы пользователям просматривать и заказывать различные товары и управлять корзиной товаров. Были созданы различные модели базы данных для хранения информации о продуктах, пользователях, заказах и корзинах.

В ходе разработки были приобретены практические умения и знания, такие как:

1. Проектирование и разработка структур данных.
2. Работа с языками программирования Python и библиотекой Flask.
3. Использование технологии REST.
4. Использование технологии ORM.
5. Работа с шаблонизатором Jinja2.
6. Работа с графическим редактором Figma.
7. Использование PostgreSQL.
8. Работа с языком программирования JavaScript и библиотекой React.
9. Подключение платежных систем.

В результате выполненной работы был разработан функциональный сайт магазина, который позволяет пользователям удобно и быстро ознакомиться с каталогом и покупать товары, просматривать информацию о них, управлять своими заказами и корзиной.

В ходе создания проекта были реализованы следующие задачи:

1. Проектирование и разработка базы данных.

2. Разработка Flask приложения, обладающего необходимой функциональностью и возможность модернизации для использования с другим интерфейсом.
3. Подключение базы данных PostgreSQL к Flask приложению.
4. Разработка функций и написание запросов к базе данных с использованием ORM.
5. Разработка макетов с использованием графического редактора Figma.
6. Разработка шаблонов с помощью шаблонизатора Jinja2.

Данный проект может послужить основой для дальнейшего развития и расширения функциональности интернет-магазина, а также может быть использован в учебных целях для изучения веб-разработки на языке Python с использованием фреймворка Flask.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Flask Documentation [Электронный ресурс]. - Режим доступа: <https://flask.palletsprojects.com/en/2.0.x/> (дата обращения: 23.06.2021).
2. SQLAlchemy Documentation [Электронный ресурс]. - Режим доступа: <https://docs.sqlalchemy.org/en/14/> (дата обращения: 23.06.2021).
3. Marshmallow Documentation [Электронный ресурс]. - Режим доступа: <https://marshmallow.readthedocs.io/en/stable/> (дата обращения: 23.06.2021).
4. JavaScript Reference [Электронный ресурс]. - Режим доступа: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference> (дата обращения: 23.06.2021).
5. React Documentation [Электронный ресурс]. - Режим доступа: <https://reactjs.org/docs/> (дата обращения: 23.06.2021).
6. Axios Documentation [Электронный ресурс]. - Режим доступа: <https://axios-http.com/docs/intro> (дата обращения: 23.06.2021).
7. JSON [Электронный ресурс]. - Режим доступа: <https://www.json.org/json-en.html> (дата обращения: 23.06.2021).
8. Representational State Transfer (REST) [Электронный ресурс]. - Режим доступа: <https://www.ibm.com/cloud/learn/rest-api> (дата обращения: 23.06.2021).
9. JavaScript Object Notation (JSON) - W3 schools [Электронный ресурс]. - Режим доступа: https://www.w3schools.com/js/js_json_intro.asp (дата обращения: 23.06.2021).
10. Flask Web Development with Python Tutorial [Электронный ресурс]. - Режим доступа: <https://www.tutorialspoint.com/flask/index.htm> (дата обращения: 23.06.2021).
11. Документация Сбербанк Эквайринг, 2023. [Электронный ресурс]. - Режим доступа: <https://securepayments.sberbank.ru/wiki/doku.php/start>.

ПРИЛОЖЕНИЕ

Приложение 1

Создание базы данных «market»

-- Database: market

-- DROP DATABASE IF EXISTS market;

CREATE DATABASE market

WITH

OWNER = postgres

ENCODING = 'UTF8'

LC_COLLATE = 'Russian_Russia.1251'

LC_CTYPE = 'Russian_Russia.1251'

TABLESPACE = pg_default

CONNECTION LIMIT = -1

IS_TEMPLATE = False;

Создание таблицы «Brands»

-- Table: Market_schem.Brands

-- DROP TABLE IF EXISTS "Market_schem"."Brands";

CREATE TABLE IF NOT EXISTS "Market_schem"."Brands"

(
 id integer NOT NULL GENERATED ALWAYS AS IDENTITY (INCREMENT 1
 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1),
 name character varying COLLATE pg_catalog."default" NOT NULL,
 CONSTRAINT "Brands_pkey" PRIMARY KEY (id),
 CONSTRAINT "Brands_name_key" UNIQUE (name)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Brands"

OWNER to postgres;

Создание таблицы «Category»

-- Table: Market_schem.Category

-- DROP TABLE IF EXISTS "Market_schem"."Category";

CREATE TABLE IF NOT EXISTS "Market_schem"."Category"

(
 id integer NOT NULL GENERATED ALWAYS AS IDENTITY (INCREMENT 1
 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1),
 name character varying COLLATE pg_catalog."default" NOT NULL,
 CONSTRAINT "Category_pkey" PRIMARY KEY (id),
 CONSTRAINT "Category_name_key" UNIQUE (name)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Category"

OWNER to postgres;

Создание таблицы «Color»

-- Table: Market_schem.Color

-- DROP TABLE IF EXISTS "Market_schem"."Color";

CREATE TABLE IF NOT EXISTS "Market_schem"."Color"

(

id integer NOT NULL GENERATED ALWAYS AS IDENTITY (INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1),

name character varying COLLATE pg_catalog."default" NOT NULL,

CONSTRAINT "Color_pkey" PRIMARY KEY (id),

CONSTRAINT "Color_name_key" UNIQUE (name)

)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Color"

OWNER to postgres;

Создание таблицы «Sizes»

-- Table: Market_schem.Sizes

-- DROP TABLE IF EXISTS "Market_schem"."Sizes";

CREATE TABLE IF NOT EXISTS "Market_schem"."Sizes"

(
 id integer NOT NULL GENERATED ALWAYS AS IDENTITY (INCREMENT 1
 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1),
 name character varying COLLATE pg_catalog."default" NOT NULL,
 CONSTRAINT "Sizes_pkey" PRIMARY KEY (id),
 CONSTRAINT "Sizes_name_key" UNIQUE (name)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Sizes"

OWNER to postgres;

Создание таблицы «Users»

-- Table: Market_schem.Users

-- DROP TABLE IF EXISTS "Market_schem"."Users";

CREATE TABLE IF NOT EXISTS "Market_schem"."Users"

(
 id integer NOT NULL GENERATED ALWAYS AS IDENTITY (INCREMENT 1
 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1),
 name character varying(32) COLLATE pg_catalog."default" NOT NULL,
 login character varying(32) COLLATE pg_catalog."default" NOT NULL,
 password_hash character varying COLLATE pg_catalog."default" NOT NULL,
 email character varying(50) COLLATE pg_catalog."default" NOT NULL,
 phone character varying(10) COLLATE pg_catalog."default",
 CONSTRAINT "Users_pkey" PRIMARY KEY (id),
 CONSTRAINT "Users_email_key" UNIQUE (email),
 CONSTRAINT "Users_login_key" UNIQUE (login)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Users"

OWNER to postgres;

Создание таблицы «Products»

```

CREATE TABLE IF NOT EXISTS "Market_schem"."Products"
(
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    description text COLLATE pg_catalog."default" NOT NULL,
    img_url character varying COLLATE pg_catalog."default" NOT NULL,
    price numeric NOT NULL,
    name character varying COLLATE pg_catalog."default" NOT NULL,
    brand character varying COLLATE pg_catalog."default" NOT NULL,
    category character varying COLLATE pg_catalog."default" NOT NULL,
    color character varying COLLATE pg_catalog."default" NOT NULL,
    click integer NOT NULL DEFAULT 1,
    CONSTRAINT "Models_pkey" PRIMARY KEY (id),
    CONSTRAINT "Models_img_url_key" UNIQUE (img_url),
    CONSTRAINT "Products_brand_fkey" FOREIGN KEY (brand)
        REFERENCES "Market_schem"."Brands" (name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    CONSTRAINT "Products_category_fkey" FOREIGN KEY (category)
        REFERENCES "Market_schem"."Category" (name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    CONSTRAINT "Products_color_fkey" FOREIGN KEY (color)
        REFERENCES "Market_schem"."Color" (name) MATCH SIMPLE
        ON UPDATE NO ACTION

```

ON DELETE NO ACTION

NOT VALID

)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Products"

OWNER to postgres;

Создание таблицы «Purchases»

```
CREATE TABLE IF NOT EXISTS "Market_schem"."Purchase"  
(  
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1  
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),  
    product_id integer NOT NULL,  
    total numeric NOT NULL,  
    quantity integer NOT NULL,  
    price_per_unit numeric NOT NULL,  
    purchase_date date,  
    CONSTRAINT "Purchase_pkey" PRIMARY KEY (id),  
    CONSTRAINT "Purchase_model_id_fkey" FOREIGN KEY (product_id)  
REFERENCES "Market_schem"."Products" (id) MATCH SIMPLE  
ON UPDATE NO ACTION  
ON DELETE NO ACTION  
)  
  
TABLESPACE pg_default;  
  
ALTER TABLE IF EXISTS "Market_schem"."Purchase"  
OWNER to postgres;
```

Создание таблицы «Storages»

```

CREATE TABLE IF NOT EXISTS "Market_schem"."Storage"
(
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    item_id integer NOT NULL,
    purchase_id integer NOT NULL,
    size character varying COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "Products_pkey" PRIMARY KEY (id),
    CONSTRAINT "Products_model_id_fkey" FOREIGN KEY (item_id)
REFERENCES "Market_schem"."Products" (id) MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION
NOT VALID,
    CONSTRAINT "Products_size_fkey" FOREIGN KEY (size)
REFERENCES "Market_schem"."Sizes" (name) MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION
NOT VALID
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Storage"
OWNER to postgres;

```

Создание таблицы «Cart»

```
CREATE TABLE IF NOT EXISTS "Market_schem"."Cart"
(
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    user_id integer NOT NULL,
    product_id integer NOT NULL,
    size character varying COLLATE pg_catalog."default" NOT NULL,
    quantity integer NOT NULL,
    product_name character varying COLLATE pg_catalog."default" NOT NULL,
    product_brand character varying COLLATE pg_catalog."default" NOT NULL,
    price numeric NOT NULL,
    CONSTRAINT "Cart_pkey" PRIMARY KEY (id)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Cart"
    OWNER to postgres;
```

Создание таблицы «Order»

```
CREATE TABLE IF NOT EXISTS "Market_schem"."Order"
(
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    user_id integer NOT NULL,
    address character varying(300) COLLATE pg_catalog."default" NOT NULL,
    order_data timestamp with time zone NOT NULL,
    check_sum numeric NOT NULL,
    status_name character varying COLLATE pg_catalog."default" NOT NULL
DEFAULT 'Ожидает оплаты'::character varying,
    CONSTRAINT "Order_pkey" PRIMARY KEY (id)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Order"
    OWNER to postgres;
```

Создание таблицы «Order_items»

```

CREATE TABLE IF NOT EXISTS "Market_schem"."Order_items"
(
    order_id integer NOT NULL,
    item_id integer NOT NULL,
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    product_id integer NOT NULL,
    CONSTRAINT "Order_products_pkey" PRIMARY KEY (id),
    CONSTRAINT "Order_products_prod_id_key" UNIQUE (item_id),
    CONSTRAINT "Order_products_order_id_fkey" FOREIGN KEY (order_id)
        REFERENCES "Market_schem"."Order" (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT "Order_products_prod_id_fkey" FOREIGN KEY (item_id)
        REFERENCES "Market_schem"."Storage" (id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
    NOT VALID
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS "Market_schem"."Order_items"
    OWNER to postgres;

```

Базовый шаблон «base.html»

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 0;
    }
    nav {
      background-color: #333;
      color: #ffffff;
      display: flex;
      justify-content: space-between;
      align-items: center;
      padding: 10px 20px;
    }

    nav ul {
      list-style: none;
      margin: 0;
      padding: 0;
      display: flex;
      gap: 20px;
    }
```



```
nav a {  
  color: #fff;  
  text-decoration: none;  
  font-size: 18px;  
}
```

```
nav a:hover {  
  text-decoration: underline;  
}
```

```
.userprofile {  
  display: flex;  
  align-items: center;  
}
```

```
.userprofile ul {  
  display: none;  
}
```

```
@media screen and (min-width: 768px) {  
  .userprofile {  
  }  
  display: block;  
}  
  
input[type="text"] {  
  width: 200px;  
  padding: 5px 10px;  
  border: none;
```

```
border-radius: 5px;  
margin-right: 10px;  
}  
button[type="submit"] {  
background-color: #4CAF50;  
color: #fff;  
border: none;  
padding: 5px 10px;  
border-radius: 5px;  
cursor: pointer;  
}  
button[type="submit"]:hover {  
background-color: #3e8e41;  
}  
.product-card {  
display: inline-block;  
width: 300px;  
height: 320px;  
background-color: #fff;  
border-radius: 5px;  
box-shadow: 0px 4px 6px rgba(0, 0, 0, 0.1);  
margin-right: 20px;  
margin-bottom: 20px;  
transition: box-shadow 0.3s ease-in-out;  
margin: 50px auto;  
}  
  
.product-card:hover {
```

```
box-shadow: 0px 8px 12px rgba(0, 0, 0, 0.2);  
}
```

```
.product-card img {  
  width: 100%;  
  height: 200px;  
  object-fit: cover;  
  border-top-left-radius: 5px;  
  border-top-right-radius: 5px;  
}
```

```
.product-card h2 {  
  font-size: 20px;  
  font-weight: bold;  
  margin: 10px;  
}
```

```
.product-card p {  
  font-size: 16px;  
  margin: 10px;  
}
```

```
.product-card a {  
  text-decoration: none;  
  color: #333;  
}
```

```
.product-card a:hover {  
  text-decoration: underline;
```

```
}  
  
.product-wrapper {  
  display: flex;  
  justify-content: center;  
  flex-wrap: wrap;  
  margin: 0 auto;  
  max-width: 1000px;  
  gap: 20px;  
}
```

```
#product-title {  
  text-align: center;  
}
```

```
button[disabled] {  
  opacity: 0.5;  
  cursor: not-allowed;  
}
```

```
.product-page {  
  display: flex;  
  flex-direction: column;  
  align-items: center;  
  margin-top: 50px;  
  margin-bottom: 50px;  
}
```

```
.product-page img {
```

```
max-width: 400px;
max-height: 400px;
margin-bottom: 20px;
}

.product-page h2 {
font-size: 24px;
text-align: center;
margin-bottom: 10px;
}

.product-page p {
font-size: 18px;
text-align: justify;
margin: 0 20px 20px 20px;
}

.product-page form {
display: flex;
flex-direction: column;
align-items: center;
}

.product-page label {
font-size: 18px;
margin-top: 10px;
}

.product-page input[type="radio"] {
```

```
margin-right: 10px;
}

.product-page input[type="number"] {
width: 50px;
margin-left: 10px;
margin-right: 10px;
text-align: center;
}

.product-page button[type="submit"] {
background-color: #333;
color: #fff;
border: none;
border-radius: 5px;
padding: 10px 20px;
font-size: 18px;
cursor: pointer;
margin-top: 20px;
}

.product-page button[type="submit"]:hover {
background-color: #555;
}

h1 {
font-size: 32px;
margin-bottom: 20px;
}
```

```
p {  
    font-size: 18px;  
    margin-bottom: 20px;  
}  
  
ul {  
    list-style: none;  
    margin: 0;  
    padding: 0;  
    font-size: 18px;  
    margin-bottom: 20px;  
}  
  
ul li {  
    margin-bottom: 10px;  
}  
  
table {  
    border-collapse: collapse;  
    font-size: 18px;  
    width: 100%;  
    margin-bottom: 20px;  
}  
  
table th, table td {  
    padding: 10px;  
    border: 1px solid black;  
    text-align: center;
```

```

}

table th {
    background-color: #333;
    color: #fff;
}

tbody tr:nth-child(even) {
    background-color: #f2f2f2;
}

</style>
</head>
<body>

<nav>
<ul>
<li><a href="{{ url_for('index') }}">Главная</a></li>
<form action="{{ url_for('search') }}" method="GET">
    <input type="text" name="query" placeholder="Поиск...">

    <button type="submit">Найти</button>
</form>
<div class="userprofile">
    {% if current_user.is_authenticated %}
        <a href="{{ url_for('cart') }}">Корзина</a>
        <a href="{{ url_for('dashboard') }}">{{ current_user.login }}</a>
        <a href="{{ url_for('logout') }}">Выйти</a>
    
```



```
{% else %}  
  <a href="{{ url_for('login') }}">Войти</a> или  
  <a href="{{ url_for('register') }}">зарегистрироваться</a>  
{% endif %}  
</div>  
</ul>  
</ul>  
</nav>  
{% block content %}{% endblock %}  
</body>  
</html>
```

Интеграция карты для формы заказа

```

<link rel="stylesheet" href="https://unpkg.com/leaflet/dist/leaflet.css" />
<script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
<script>
  var map = L.map('mapid').setView([55.028554, 82.920532], 13);
  var previousMarker = null;

  L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 18,
  }).addTo(map);

  // Получаем координаты местоположения пользователя
  if ('geolocation' in navigator) {
    navigator.geolocation.getCurrentPosition(function(position) {
      var lat = position.coords.latitude;
      var lon = position.coords.longitude;

      // Получаем информацию о городе по координатам
      var url = 'https://nominatim.openstreetmap.org/reverse?format=json&lat='
+ lat + '&lon=' + lon + '&zoom=10';
      fetch(url)
        .then(function(response) {
          return response.json();
        })
        .then(function(data) {
          // Выводим информацию о городе
          var city = data.address.city || data.address.town || data.address.village
          || data.address.hamlet || data.address.locality || data.address.suburb

```

```

// data.address.district // data.address.county // data.address.region
// data.address.state // data.address.country;
// document.getElementById('city').textContent = city;

// Показываем местоположение пользователя на карте
L.marker([lat, lon]).addTo(map)
    .bindPopup('Вы здесь: ' + city)
    .openPopup();
map.setView([lat, lon], 13);
})
.catch(function(error) {
    console.log('Ошибка получения города: ' + error.message);
});
}, function(error) {
    console.log('Ошибка получения местоположения: ' + error.message);
});
} else {
    console.log('Geolocation не поддерживается браузером');
}

function onMapClick(e) {
    // Удаляем предыдущий адрес, если он существует
    document.getElementById('adres').value = "";

    // Удаляем предыдущую метку, если она существует
    if (previousMarker) {
        map.removeLayer(previousMarker);
    }
}

```

```

// Декодируем координаты местоположения в текстовый адрес
var url = 'https://nominatim.openstreetmap.org/reverse?format=json&lat=' +
e.latlng.lat + '&lon=' + e.latlng.lng;
fetch(url)
  .then(function(response) {
    return response.json();
  })
  .then(function(data) {
    // Выводим текстовый адрес в поле "Адрес"
    document.getElementById('adres').value = data.display_name;
  })
  .catch(function(error) {
    console.log('Ошибка получения адреса: ' + error.message);
  });

// Добавляем маркер на карту
var marker = L.marker(e.latlng).addTo(map);
previousMarker = marker;
}

map.on('click', onMapClick);
</script>

```

Код файла «Models.py»

```
from flask_sqlalchemy import SQLAlchemy  
from flask_login import UserMixin  
from werkzeug.security import generate_password_hash, check_password_hash  
from datetime import datetime  
from application import db, manager
```

```
@manager.user_loader  
def load_user(user_id):  
    return User.query.get(user_id)
```

```
# описываем модели
```

```
class Cart(db.Model):  
    __tablename__ = 'Cart'  
    __table_args__ = {'schema': 'Market_schem'}  
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)  
    user_id = db.Column(db.Integer, nullable=False)  
    product_id = db.Column(db.Integer, nullable=False)  
    size = db.Column(db.String(10), nullable=False)  
    quantity = db.Column(db.Integer, nullable=False)  
    product_name = db.Column(db.String(50), nullable=False)  
    product_brand = db.Column(db.String(50), nullable=False)  
    price = db.Column(db.Integer, nullable=False)
```

```
# размеры
```

```
class Sizes(db.Model):  
    __tablename__ = 'Sizes'
```

```

__table_args__ = {'schema': 'Market_schem'}
id = db.Column(db.Integer, primary_key=True, autoincrement=True)
name = db.Column(db.String(50), nullable=False, unique=True)

# брэнды
class Brands(db.Model):
    __tablename__ = 'Brands'
    __table_args__ = {'schema': 'Market_schem'}
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(50), nullable=False )

class Category(db.Model):
    __tablename__ = 'Category'
    __table_args__ = {'schema': 'Market_schem'}
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(50), nullable=False)

class Color(db.Model):
    __tablename__ = 'Color'
    __table_args__ = {'schema': 'Market_schem'}
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(50), nullable=False)

#класс юзер наследует UserMixin для работы с flask-login
class User(UserMixin, db.Model):
    __tablename__ = 'Users'
    __table_args__ = {'schema': 'Market_schem'}
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)

```

```

name = db.Column(db.String(50), nullable=False)
login = db.Column(db.String(50), nullable=False, unique=True)
password_hash = db.Column(db.String(128), nullable=False)
email = db.Column(db.String(120), nullable=False, unique=True)

#функция для получения пароля
def get_id(self):
    return str(self.id)

#сравнение хэшей пароля
def check_password(self, password):
    return check_password_hash(self.password_hash, password)

#шифрование пароля с использованием веб-сервиса werkzeug.security
@staticmethod
def hash_password(password):
    return generate_password_hash(password)

# склад
class Storage(db.Model):
    __tablename__ = 'Storage'
    __table_args__ = {'schema': 'Market_schem'}
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    item_id = db.Column(db.Integer, db.ForeignKey("Products.id"), nullable=False)
    size = db.Column(db.String(50), db.ForeignKey("Sizes.name"), nullable=False)
    purchase_id = db.Column(db.Integer, db.ForeignKey("Purchase.id"),
nullable=False)

```

закупки

```
class Purchase(db.Model):
    __tablename__ = 'Purchase'
    __table_args__ = {'schema': 'Market_schem'}
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    product_id = db.Column(db.Integer, db.ForeignKey("Products.id"),
        nullable=False)
    total_price = db.Column(db.Integer)
    quantity = db.Column(db.Integer)
    price_per_unit = db.Column(db.Integer)
    # price_per_unit = total_price // quantity
```

карточка товара

```
class Products(db.Model):
    __tablename__ = 'Products'
    __table_args__ = {'schema': 'Market_schem'}
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(50), nullable=False)
    brand = db.Column(db.String(50), db.ForeignKey("Brands.name"),
        nullable=False)
    category = db.Column(db.String(50), db.ForeignKey("Category.name"),
        nullable=False)
    color = db.Column(db.String(50), db.ForeignKey("Color.name"), nullable=False)
    description = db.Column(db.String(100000))
    img_url = db.Column(db.String(100000), unique=True)
    price = db.Column(db.Integer, nullable=False)
    click = db.Column(db.Integer)
    ## статус заказа
```


заказ

class Order(db.Model):

__tablename__ = 'Order'

__table_args__ = {'schema': 'Market_schem'}

id = db.Column(db.Integer, primary_key=True, autoincrement=True)

user_id = db.Column(db.Integer, nullable=False)

address = db.Column(db.String(50), nullable=False)

order_data = db.Column(db.DateTime)

check_sum = db.Column(db.Integer)

status_name = db.Column(db.String(50), nullable=False, default='Ожидает оплаты')

товар в заказе

class Order_items(db.Model):

__tablename__ = 'Order_items'

__table_args__ = {'schema': 'Market_schem'}

order_id = db.Column(db.Integer, db.ForeignKey("Order.id"), nullable=False)

item_id = db.Column(db.Integer, nullable=False, unique=True)

id = db.Column(db.Integer, primary_key=True, autoincrement=True)

product_id = db.Column(db.Integer, db.ForeignKey("Order.id"), nullable=False)