УНИВЕРСИТЕТ «УРБАН»

Специализация: Разработчик программ в Python.

**ДИПЛОМНАЯ РАБОТА**

**Тема дипломной работы:** Анализ и сравнение написания web-приложений с использованием разных фреймворков: Разработать простые веб-приложения с использованием Django, Flask и FastAPI, провести их сравнение.

**Студент (ка):**

Ф.И.О Гостевских Светлана Александровна

**Работу проверил:**

Должность:\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ф.И.О \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2024г

**ОГЛАВЛЕНИЕ:**

**[1. Введение.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692639)   
  
[Обоснование выбора темы:](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692640)   
  
[Определение цели и задач исследования:](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692641)  
  
[2. Основные понятия и определения.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692642)   
  
[3. Методы и подходы к разработке.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692643)**

**[Архитектура веб-приложения.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692645)   
  
[Обеспечение безопасности.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692646)   
  
[4. Обзор популярных инструментов для разработки веб-приложений на Python](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692647)   
  
[Django.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692648)   
  
[Flask.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692649)   
  
[FastAPI](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692650)   
  
[5. Проектирование приложения.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692651)**

**Требования к проекту.**

**[6. Разработка в соответствии с созданной документацией.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692655)   
  
[Разработка.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692657)**

1. **Подведение итогов по разделу «Разработка в соответствии с созданной документацией».**

**[8. Анализ и интерпретация результатов.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692658)   
  
[Рекомендации по выбору модели в зависимости от проекта.](https://urban-university.ru/members/courses/course999421818026/20240313-0000vybor-temy-po-diplomnoj-rabote-po-kursu-pythonrazrabotcik-648056632363" \l "_Toc167692661)**

**9.Список используемой литературы.**

**1.ВВЕДЕНИЕ.**

**Основание выбора темы.**

В нашем быстро развивающемся и меняющемся, информационно насыщенном мире, очень важны помощники в виде различных веб приложений. Сайты, приложения для телефонов, программ для бухучета, программное обеспечение для разных сфер промышленности, в медицине, учёбе и т.д. Создаваемые веб-приложения облегчают сбор информации и особенно сбор и обработку больших объёмов информации. Из-за роста жизненных задач человечество с давних пор изобретало разные информационные носители начиная с палочки с нанесёнными на неё зарубками отмечая количество чего либо, до изобретения различных счётных механических машин, а также простых счёт и калькулятора. А сейчас уже прогресс дошёл до создания компьютеров и создания искусственного интеллекта.

С каждым новым изобретением человечеству даётся возможность, решать более глобальные задачи для мира. А так же и помочь людям в решении их повседневных задач, упрощать им жизнь и улучшать качество жизни.

Когда мы создаём веб-приложения под конкретную задачу создаём что-то , что может быть полезным для других людей, это приносит удовлетворение и чувство принесённой пользу обществу.

**Определение целей и задач исследования.**

1. В рамках дипломного проекта глубже изучить основные функции фреймворков Django, Flask и FastAPI.
2. Разработать простые веб-приложения с использованием Django, FastAPI на основе учебных материалов Университета Urban.
3. Изучить способы создания приложения на Flask по собранным материалам из открытых источников. И разработать простое веб-приложение применив изученные материалы.
4. Произвести сравнение и анализ Django, Flask и FastAPI.
5. Определить для каких видов разработки целесообразно применять какой фреймворк и применять для работы в будущем.
6. **Основные понятия и определения.**

**Веб-приложение** - это клиентоориентированное сервисное приложение, в котором клиент взаимодействует с веб-сервером при помощи браузера.

Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется преимущественно на сервере, обмен информацией происходит по сети.

**Фреймворк** - это набор инструментов, компонентов и методов, которые облегчают разработку программного обеспечения. Это готовый шаблон для написания программы(основа, каркас).

**Django («Джанго»)** — это свободный фреймворк для разработки быстрых и безопасных веб-приложений и сайтов на языке Python.

**Flask** — это микровеб-фреймворк, написанный на Python. Он не требует специальных инструментов или библиотек и поддерживает расширения для добавления функций приложения.

**FastAPI** - это современный и высокопроизводительный веб-фреймворк для создания API на Python 3.6+, основанный на стандартных подсказках типов Python. FastAPI это своего рода язык общения между сервисами и программами. Чтобы они понимали друг друга, API включает в себя набор инструкций, правил, способов, инструментов, посредством которых и происходит обмен данными.

1. **Методы и подходы к разработке**

**Выбор методов**

Выбор методов веб-разработки с использованием фреймворков зависит от конкретных потребностей и предпочтений.

Некоторые критерии выбора фреймворка:

**Производительность.** Насколько быстро фреймворк обрабатывает запросы и отвечает на них. Производительность может быть критически важной для приложений с высокой нагрузкой, таких как социальные сети или интернет-магазины.

**Простота использования.** Лёгкость освоения и использования фреймворка. Это особенно важно для новичков, которые только начинают свой путь в веб-разработке.

**Сообщество.** Размер и активность сообщества разработчиков, которые могут помочь с решением проблем. Большое сообщество обычно означает наличие множества готовых решений и библиотек, которые можно использовать в своём проекте, а так же специалисты у которых можно получить опыт в разработке.

**Документация.** Качество и полнота документации, которая поможет в изучении и использовании фреймворка. Хорошая документация может значительно ускорить процесс обучения и разработки.

**Расширяемость.** Возможность добавления новых функций и модулей. Это важно для проектов, которые могут расти и изменяться со временем.

**Безопасность.** Наличие встроенных механизмов для защиты от распространённых угроз. Безопасность должна быть приоритетом для любого веб-приложения, особенно если оно обрабатывает конфиденциальные данные пользователей.

**Существуют два вида веб-фреймворков:**

**Фронтенд-фреймворки.** Используются для разработки внешнего (пользовательского) интерфейса. Они чаще всего основаны на таких языках программирования, как JavaScript, HTML и CSS. Среди решаемых задач — разработка дизайна UX/UI, SEO-оптимизация, фрагменты кода, шаблоны, управление взаимодействием с пользователем и многое другое.

**Бэкэнд-фреймворки.** Используются для разработки серверной части, которая отвечает за функционирование ИТ-продукта. Они основаны на таких языках программирования, как Python, .NET, Ruby, Java и PHP. Среди решаемых задач — функционирование сервера и базы данных, протоколы маршрутизации, логика и архитектура сервиса, параметры авторизации, безопасность и многие другие.

Некоторые популярные фронтенд-фреймворки: React, Vue.js, Ember, Bootstrap и Angular. Среди бэкенд-фреймворков — Django, Laravel, Ruby On Rails, Spring, Express, ASP.NET Core.

**Вывод:** для данной дипломной работы напишем простые приложения для каждого фреймворка и в дальнейшем сможем подбирать какой лучше фреймворк подходит для разработки.

**Архитектура веб-приложения**

Архитектура веб-приложения — это подход к его созданию, проектированию и развёртыванию. Она включает в себя такие элементы, как серверы приложений, базы данных, фреймворки, системы управления контентом и многое другое.

**Обычно архитектура веб-приложения состоит из трёх основных компонентов:**

**Веб-браузер.** Ключевой компонент, который взаимодействует с пользователем, получает входные данные и управляет логикой представления, одновременно контролируя взаимодействие пользователя с приложением.

**Веб-сервер.** Веб-сервер обрабатывает бизнес-логику и запросы пользователей, направляя запросы к нужному компоненту и управляя всеми операциями приложения.

**Сервер базы данных**. Сервер базы данных предоставляет необходимые данные для приложения и решает задачи, связанные с данными.

**Разделить ответственность между компонентами приложения помогают слои архитектуры:**

**Презентационный.** Назначение — взаимодействие с пользователем и отображение данных. Компоненты: HTML/CSS, JavaScript, а также фреймворки и библиотеки (React, Angular, Vue.js).

**Логический.** Назначение — выполнение операций. Компоненты: языки (программирование веб-приложений может включать использование Python, Java, JavaScript, Node.js, PHP, Ruby и т. д.), а также фреймворки (Django, Spring, Express, Laravel, Ruby on Rails).

**Слой доступа к данным.** Назначение — взаимодействие с базой данных. Компоненты: базы данных: реляционные (MySQL, PostgreSQL) и нереляционные (MongoDB, Cassandra).

**Вывод:**Для моей дипломной работы подойдёт архитектура сайта в виде Интернет-магазин, или пример небольшой программы для учёта товаров на складе. Для данных приложений можно использовать микросервисную и макросервисную архитектуру. Каждый сервис будет отвечать за определённую функциональность, например, управление товарами, каталог товаров, учёт товаров, корзину покупок и обработку заказов. Это позволит работать и с небольшими объёмами и легко масштабировать систему и обновлять отдельные сервисы без влияния на другие.

**Обеспечение безопасности**

Обеспечение безопасности веб-разработки с использованием фреймворков включает в себя ряд мер:

Выбор правильной архитектуры. При разработке приложения нужно определить эффективную техническую модель, соответствующую конкретным бизнес-целям.

Использование встроенных механизмов безопасности фреймворка. Например, в Django есть инструменты для защиты от CSRF, XSS, SQL-инъекций и других атак.

Контроль доступа. Веб-приложения нуждаются в механизмах, которые определяют, к каким функциям и данным пользователь имеет доступ. Например, в Django есть встроенный инструмент для управления доступом — Django Admin.

Регулярные обновления и автоматизированное тестирование. Это помогает исправлять уязвимости и следить за последними рекомендациями по безопасности.

Важно, писать код, который защищён от распространённых атак, и проводить мониторинг новых угроз и возможные риски безопасности

1. **Обзор популярных инструментов для разработки веб-приложений Python.**

**Django**

Django («Джанго») — это свободный фреймворк для разработки быстрых и безопасных веб-приложений и сайтов на языке Python.

Его создали в 2003–2005 годах веб-программисты онлайн-версии Lawrence Journal-World Адриан Головати и Саймон Уиллисон.

С 2008 года поддержкой и развитием Django занимается организация Django Software Foundation (DSF).

**Вот некоторые преимущества этого фреймворка:**

**Быстрота.** Django помогает разработчикам создавать приложения как можно быстрее, экономя время и ресурсы на каждом этапе разработки.

**Полная комплектация.** Фреймворк работает с десятками дополнительных функций, которые помогают с аутентификацией пользователя, картами сайта, администрированием содержимого и другим.

**Безопасность.**  Django защищает приложения от распространённых ошибок безопасности, связанных с Python, например, от межсайтового скриптинга, подделки запросов и SQL-инъекций.

**Масштабируемость.** Фреймворк подходит для работы с самыми высокими трафиками.

Работа с большинством баз данных. Django позволяет разработчикам работать с несколькими базами данных одновременно, а также переходить из одной базы данных в другую и выполнять стандартные операции без необходимости написания большого количества дополнительного кода.

**Flask**

Flask — это микровеб-фреймворк, написанный на Python. Он не требует специальных инструментов или библиотек и поддерживает расширения для добавления функций приложения.Расширения для микрофреймворка позволяют коммуницировать с базами данных, проверять формы, контролировать загрузку на сервер, работать с аутентификацией и многое другое.

Flask был создан Армином Ронахером из Pocoo — международной группы энтузиастов Python, созданной в 2004 году. По словам Ронахера, изначально идея была первоапрельской шуткой, которая была достаточно популярна, чтобы превратить её в серьёзное приложение.

Первая публичная версия Flask вышла 16 апреля 2010 года.

**FastAPL**

FastAPI был выпущен в 2018 году.

Это современный и высокопроизводительный веб-фреймворк для создания API на Python 3.6+, основанный на стандартных подсказках типов Python.

FastAPI работает очень быстро благодаря встроенной поддержке функции async в Python 3.6+. (Асинхронное программирование позволяет улучшить производительность приложения, особенно при работе с вводом-выводом (I/O), таким как чтение и запись файлов, обращение к базам данных и веб-сервисам).

Сейчас многие крупные технологические компании, такие как Uber, Netflix и Microsoft, используют FastAPI для создания своих приложений.

В переводе на русский API — программный интерфейс, с помощью которого приложения, веб-сервисы и программы обмениваются информацией.

Говоря простыми словами, это своего рода язык общения между сервисами и программами. Чтобы они понимали друг друга, API включает в себя набор инструкций, правил, способов, инструментов, посредством которых и происходит обмен данными.

1. **Проектирование приложений**

**Для данной дипломной работы:**

**Требования к проект, к приложениям.**

1. Разработать простые веб-приложения с простым но наглядным функционалом.
2. Проработать моменты:

- с чего начинается проект в каждом фреймворке

- как выстраивается структура в каждом фреймворке, насколько быстро создаётся проект.

- какой функционал используется для создания проекта в каждом фреймворке.

- создать проекты и собрать информацию в таком виде чтоб была возможность выполнить задание проекта, а именно провести анализ и сравнение фреймворков Django, Flask и FastAPI и в итоге определить для каких веб-приложений, какой фреймворк лучше подходит.

1. Рассмотреть возможность объединения веб-приложения Django и базы данные в FastAPI.

Это пригодиться нам для дальнейшей работы.

1. **Разработка в соответствии с созданной документацией.**

Для выполнения задачи по сравнению работы фреймворков Django, Flask и FastAPI и чтоб произвести анализ их работы, напишем простые веб-приложения с использованием инструментов каждого фреймворка и затем сделаем анализ и сравнение назначения и работы фреймворков на конкретном примерах.

**Django Project**

1.Создаём новый проект в Python.

**Название: pythonBrushikadbcars*.*** *Приложение* **brusshika.**

1. Создаём среду окружения.

**python -m venv venv**

1. Активируем python.

**venv\scripts\activate**

1. Проверяем установку Python в настройках и проверяем активацию среды и активацию Python. И что все сохраняется в нужный нам каталог/файл.

Все установилось и работает. Продолжаем работу над нашим проектом.

1. Устанавливаем в созданную среду окружение нужно нам программное обеспечение через pip.

Устанавливаем **Django**.

Команда в терминале:

**pip install django**

Установилось django успешно.

**python.exe -m pip install --upgrade pip** (обновляем pip)

1. Проверяем что установилось какие программы:

Команда в терминале:

**pip freeze**

Программы установлены:

asgiref==3.8.1

Django==5.1.3

sqlparse==0.5.1

tzdata==2024.2

Для начала работы этого достаточно. В дальнейшем по ходу создания проекта будем до устанавливать нужные нам программы.

1. **Создаём приложение в нашем проекте, название brusshika. В нем будем писать наш сайт.**

Команда в терминале:

**django-admin startproject brusshika**

1. **Проверяем, нашу приложение успешно создано.**

В нем видим 2 паки brussnika.

Открываем вторую паку видим файлы, которые будут нам помогать создавать приложение-сайт и будут нам помогать связываться со сторонним сервером.

Сейчас структура выглядит так:

**brusshika**

**brusshika**

\_\_init\_\_ - указывает на то что наш проект пакет

asgi.py - для запуска нашего приложения на стороннем сервере

settings.py - файл с настройками проекта

urls.py - маршрутизация пользователей по страницам

wsgi.py - для запуска нашего приложения на стороннем сервере

manage.py - управляет нашим сервером

1. **Проверяем работу. Переходим в brusshika.**

Команда в терминале:

**cd brusshika**

1. **Запускаем сервер:**

Команда в терминале:

**python manage.py runserver**

**Получаем:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

November 05, 2024 - 16:04:39

Django version 5.1.3, using settings '**brusnika**.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CTRL-BREAK.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Связь установилась. Переходим на http://127.0.0.1:8000/.

Все работает. Открылось приложение внутри браузера.

Останавливаем сервер **CTRL-С**.

1. **Создаём внутри нашего проекта приложения brusshika 2 приложения:**

приложение **brdbcarsapp** и **cart**.

Команда в терминале:

**python manage.py startapp**  **brdbcarsapp**

**python manage.py startapp**  **cart**

Смотрим в нашей структуре в приложении **brusshika** появилась 2 папки: папка **brdbcarsapp** и папка **cart**.

Структура папок **brdbcarsapp** и **cart** :

\_\_init \_\_.py - пакет

admin.py - для админ панели для нашего сайта

apps.py - с настройками нашего приложения

models.py - модели базы данных

tests.py - тест для тестов

venws.py - для визуального представления и всей логики нашего приложения на сервере.

1. **Создаём файл requirements.txt** (Этот файл часто используется с инструментами управления пакетами, такими как pip, для автоматической установки и управления зависимостями проекта.)

Команда в терминале:

**pip freeze > requirements.txt**

1. **Создаём для каждого приложения brdbcarsapp** и **cart логику нашего проекта:**

Далее работаем с файлом views.py - логика проекта. Создаём функции которые будет логику обрабатывать и возвращать шаблон пользователю.

И файлом urls.py - маршрутизация и вывод информации на сайт.

В папке **brdbcarsapp** views.py меню проекта и вход на сайт:

Импортируем необходимые инструменты для логики нашего проекта и прописываем логику/

**brdbcarsapp/views.py**:

from django.shortcuts import render  
from .forms import UserRegister  
from django.http import HttpResponse  
import webbrowser  
  
def menu(request):  
 return render(request, 'menu.html')  
  
def registration\_page(request):  
 users = {  
 'Vasya': 'VasyokUrban',  
 'Terminator2000': 'lolkekcheburek1894'  
 }  
 info = {  
 }  
 if request.method == "POST":  
 form = UserRegister(request.POST)  
 if form.is\_valid():  
 # Обработка данных формы  
 username = form.cleaned\_data['username']  
 password = form.cleaned\_data['password']  
 repeat\_password = form.cleaned\_data['repeat\_password']  
 age = form.cleaned\_data['age']  
 if password == repeat\_password and int(age) >= 18 and username not in users:  
 return HttpResponse(  
 f"Приветствуем, {username}!, {webbrowser.open("http://127.0.0.1:8000/menu/")} Перейдите по ссылке в магазин Брусника")  
 elif password != repeat\_password:  
 info['error'] = 'Пароли не совпадают'  
 elif int(age) < 18:  
 info['error'] = 'Вы должны быть старше 18'  
 elif username in users:  
 info['error'] = 'Пользователь уже существует'  
  
 else:  
 form = UserRegister()  
 return render(request, 'registration\_page.html', {'form': form})

В папке **cart/** **views.py** прописывает работу админ панели, с базой данных, и работу корзины покупок:

**cart/** **views.py**:

from django.shortcuts import render  
from django.http import HttpResponse  
from django.shortcuts import render, redirect  
from .models import Product, CartItem, Category  
  
  
def pokupka(request):  
 return render(request, 'pokupka.html')  
  
def product\_list(request):  
 products = Product.objects.all()  
 return render(request, 'index.html', {'products': products})  
  
def view\_cart(request):  
 cart\_items = CartItem.objects.filter(user=request.user)  
 total\_price = sum(item.product.price \* item.quantity for item in cart\_items)  
 return render(request, 'cart.html', {'cart\_items': cart\_items, 'total\_price': total\_price})  
  
def add\_to\_cart(request, product\_id):  
 product = Product.objects.get(id=product\_id)  
 cart\_item, created = CartItem.objects.get\_or\_create(product=product,  
 user=request.user)  
 cart\_item.quantity += 1  
 cart\_item.save()  
 return redirect('cart:view\_cart')  
  
def remove\_from\_cart(request, item\_id):  
 cart\_item = CartItem.objects.get(id=item\_id)  
 cart\_item.delete()  
 return redirect('cart:view\_cart')  
  
def home(request):  
 return HttpResponse('ДОБРЫЙ ДЕНЬ!')

**И файлом settings.py прописываем: [BASE\_DIR / 'templates'] в блоке:**

EMPLATES = [  
 {  
 'BACKEND': 'django.template.backends.django.DjangoTemplates',  
 'DIRS': [BASE\_DIR / 'templates'],

BASE\_DIR - переменная Django которая хранит информацию о нашем проекте, его месте где он у нас расположен.

'templates' - имя папки из которой Django будет брать шаблоны.

Папку мы создаём там же где manage.py.

Так оформляется по стандарту.

**Подключаем наше приложение ‘brdbcarsapp’** и **‘cart’ так же в файле settings.py в блоке:**INSTALLED\_APPS = [  
 'django.contrib.admin',  
 'django.contrib.auth',  
 'django.contrib.contenttypes',  
 'django.contrib.sessions',  
 'django.contrib.messages',  
 'django.contrib.staticfiles',  
  **'brdbcarsapp’,**

‘**cart**’,  
]

Теперь Django может обращаться к приложению  **‘brdbcarsapp’** и **‘cart’** и брать из него всю логику.

**Создаём HTML файлы для наших приложений ‘brdbcarsapp’** и **‘cart’ и логики в файле views.py в приложениях HTML файлы. В папке templates.**

С помощью: языка Django Template Language (DTL) — это текстовый язык шаблонов, который обеспечивает мост между скриптами (HTML, CSS, JS и т. д.) и языками программирования.

Ссылка примеры основных тегов для создания шаблонов HTML файлов.

<https://uguide.ru/tablica-osnovnykh-tegov-html-s-primerami>

В папке **templates**:

<> cart.html

<> index.html

<> info.html

<> menu.html

<> pokupka.html

<> registration\_page.html

**Пример файла html для приложения ‘brdbcarsapp’:**

**menu.html:**

{% load static %}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>{% block title %} МАГАЗИН ТОВАРОВ ДЛЯ ЗДОРОВЬЯ "БРУСНИКА"{% endblock %}</title>  
 <link rel = 'stylesheet' type="text/css" href = '{% static "style.css" %}'>  
</head>  
<adm>  
 <button>  
 <a href="http://127.0.0.1:8000/admin/">ВХОД ДЛЯ АДМИНИСТРАТОРОВ</a>  
 </button>  
</adm>  
<body>  
 <img src="{% static '/IMG\_20240521\_094111.jpg' %}" alt="My image" height="260px" width="1455px"/>  
 <pivetctvie>  
 {% block pivetctvie %}  
 <h1>ВАС ПРИВЕТСТВУЕТ МАГАЗИН "БРУСНИКА"! </h1>  
 <h2>Природные товары для вас.</h2>  
 <h2>Польза и красота от природы.</h2>  
 <h2>Подбираем только качественные товары для вашего здоровья.</h2>  
 {% endblock %}  
 </pivetctvie>  
 <nav>  
 <ul>  
 <menu>  
 {% block menu %}  
 <h1>МАГАЗИН</h1>  
 <button>  
 <a href="http://127.0.0.1:8000/product\_list/">ПЕРЕЙТИ В КАТАЛОГ ТОВАРОВ</a>  
 </button>  
  
 <h1>МАСЛА</h1>  
 <h1>СМОЛКА ЖИВАТЕЛЬНА</h1>  
 <h1>ЧАИ</h1>  
 <h1>БАЛЬЗАМЫ</h1>  
 <h1>ВОДА ПРОПОЛИСНАЯ</h1>  
 <a href='http://127.0.0.1:8000/cart/'>КАРЗИНА</a>  
 {% endblock %}  
 </menu>  
 </ul>  
 </nav>  
 <content>  
 {% block content %}  
 <p>НАШИ ПРИЕМУЩЕСТВА</p>  
 <li> БЫСТРОЕ ОФОРМЛЕНИЕ</li>  
 <li> ВЫСОКОЕ КАЧЕСТВО ПРОДУКЦИ</li>  
 <li> БЫСТРАЯ ДОСТАВКА </li>  
 {% endblock %}  
 </content>  
  
 <section>  
 {% block section %}  
 <p>Контакты: 620000, г.Новосибирск, ул. Ленина, д.50.</p>  
 <p>т.8-800-00000009, brusnica@mail.ru</p>  
 {% endblock %}  
 </section>  
  
 <info>  
 {% block info %}  
 {% include 'info.html' %}  
 {% endblock %}  
 </info>  
</body>  
</html>

**registration\_page.html:**

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <meta http-equiv="X-UA-Compatible" content="ie=edge">  
 <title>Авторизация пользователя</title>  
 <style>  
 body {  
 font-family: Arial, sans-serif;  
 display: flex;  
 justify-content: center;  
 align-items: center;  
 height: 100vh;  
 margin: 0;  
 background-color: #f3f4f6;  
 }  
 form {  
 background-color: #fff;  
 padding: 20px;  
 border-radius: 8px;  
 box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);  
 max-width: 400px;  
 width: 100%;  
 }  
 h2 {  
 text-align: center;  
 color: #333;  
 }  
 label {  
 font-weight: bold;  
 display: block;  
 margin-top: 15px;  
 color: #555;  
 }  
 input[type="text"],  
 input[type="password"],  
 input[type="number"] {  
 width: 100%;  
 padding: 10px;  
 margin-top: 5px;  
 border-radius: 5px;  
 border: 1px solid #ccc;  
 font-size: 14px;  
 }  
 button {  
 width: 100%;  
 padding: 12px;  
 border: none;  
 border-radius: 5px;  
 margin-top: 20px;  
 background-color: #4CAF50;  
 color: white;  
 font-size: 16px;  
 cursor: pointer;  
 }  
 button:hover {  
 background-color: #45a049;  
 }  
 p {  
 text-align: center;  
 color: #777;  
 margin-top: 20px;  
 font-size: 14px;  
 }  
 </style>  
</head>  
<body>  
 <form method="post">  
 {% csrf\_token %}  
 <h2>Регистрация пользователя</h2>  
  
 <label for="username">Введите логин:</label>  
 <input type="text" id="username" name="username" maxlength="30" required>  
  
 <label for="password">Введите пароль:</label>  
 <input type="password" id="password" name="password" maxlength="8" required>  
  
 <label for="repeat\_password">Подтвердите пароль:</label>  
 <input type="password" id="repeat\_password" name="repeat\_password" maxlength="8" required>  
  
 <label for="age">Введите свой возраст:</label>  
 <input type="number" id="age" name="age" maxlength="3" required>  
  
 <button type="submit">Зарегистрироваться</button>  
  
 <p>Пройдите регистрацию и зайдите на сайт товаров для здоровья "Брусника"</p>  
 </form>  
</body>  
</html>

**Пример файлa HTML для приложения ‘cart’ :**

**cart.html**

{% load static %}  
<!DOCTYPE html>  
<html lang="en">  
  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <link rel = 'stylesheet' type="text/css" href = '{% static "style.css" %}'>  
 <title>КАРЗИНА ПОКУПОК МАГАЗИН "БРУСНИКА"</title>  
 <style>  
 /\* Add CSS styles for flex container and items \*/  
 .product-list {  
 display: flex;  
 flex-wrap: wrap; /\* Allow items to wrap to the next row if necessary \*/  
 justify-content: space-between; /\* Space items evenly along the main axis \*/  
 list-style: none; /\* Remove list styles \*/  
 padding: 0;  
 }  
  
 .product-item {  
  
 flex: 1; /\* Grow to fill available space evenly \*/  
 /\* Limit item width to avoid overcrowding \*/  
 margin: 10px; /\* Add spacing between items \*/  
 border: 1px solid #ccc; /\* Add a border for visual separation \*/  
 padding: 10px;  
 text-align: center;  
 }  
  
 /\* Style the "Buy Now" button \*/  
 .buy-now-button {  
 display: block;  
 margin-top: 10px;  
 background-color: #007bff;  
 color: #fff;  
 text-decoration: none;  
 padding: 5px 10px;  
 border-radius: 5px;  
 }  
 </style>  
</head>  
  
<body>  
 <h1> КАРЗИНА </h1>  
 <div class="product-list">  
  
 {% for item in cart\_items %}  
 <div class="product-item">  
 <h1>Вы купили товар</h1>  
 <h2>{{ item.product.name }} ({{ item.quantity }})</h2>  
 <h2>Цена: {{ item.product.price }} руб. </h2>  
 <a href="{% url 'cart:remove\_from\_cart' item.id %}">Удалить из карзины товар</a>  
 </div>  
 {% empty %}  
 <p>ВАША КАРЗИНА ПУСТА</p>  
 {% endfor %}  
  
 </div>  
  
 <h2>СУММА К ОПЛАТЕ: {{ total\_price }} руб.</h2>  
 <button>  
 <a href="{% url 'cart:product\_list' %}"> В КАТАЛОГ \* ПРОДОЛЖИТЬ ПОКУПКИ</a>  
 </button>  
  
 <button>  
 <a href="http://127.0.0.1:8000/pokupka/">ОПЛАТИТЬ ТОВАРЫ</a>  
 </button>  
 <button>  
 <a href="http://127.0.0.1:8000/menu/">ВЕРНУТСЯ НА ГЛАВНУЮ СТРАНИЦУ</a>  
 </button>  
</body>  
</html>

1. **Создание urls.py маршрутов для взаимодействия приложений и логики проекта.**

Так же для удобства маршрутизации создаем в приложении  **‘cart’ файл urls.py и в нем размещаем маршруты для работы карзины и админ:**

**cart/urls.py :**

from django.urls import path  
from . import views  
  
app\_name = 'cart'  
  
urlpatterns = [  
 path('product\_list/', views.product\_list, name='product\_list'),  
 path('home/', views.home, name='home'),  
 path('cart/', views.view\_cart, name='view\_cart'),  
 path('pokupka/', views.pokupka, name='pokupka.html'),  
 path('add/<int:product\_id>/', views.add\_to\_cart, name='add\_to\_cart'),  
 path('remove/<int:item\_id>/', views.remove\_from\_cart, name='remove\_from\_cart'),  
]

**И все основные маршнуты прописываем в основной папке brusshika в urls.py который объединяет работу всех приложений файла через маршруты взаимодействия:**

**Brusshika/urls.py**  
from django.contrib import admin  
from django.urls import path, include  
from django.conf.urls.static import static  
from django.conf import settings  
from brdbcarsapp.views import registration\_page, menu  
  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('', registration\_page),  
 path('menu/', menu),  
 path('', include('cart.urls')),  
] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

1. **Для создания стиля и медиа для нашего проекта прописываем в файле settings.py:**

import os

STATIC\_URL = 'static/'  
  
STATICFILES\_DIRS = [  
 os.path.join(BASE\_DIR, "statiс")  
]

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

Static и media находится в директории с проектом BASE\_DIR в папке statis.

Импортируем библиотеку os.

**Прописываем стиль. В папке static. И HTML шаблоне menu.html. То как будет выглядеть наш сайт.**

**В menu.html:**

{% load static %}  
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>{% block title %} МАГАЗИН ТОВАРОВ ДЛЯ ЗДОРОВЬЯ "БРУСНИКА"{% endblock %}</title>  
 <link rel = 'stylesheet' type="text/css" href = '{% static "style.css" %}'>  
</head>

**В**  **папке static прописываем стиль нашего сайта, то как будет его видеть пользователь:**

**static/style.css:**

body {  
 font-family: Monaco, monospace;  
 font-style: italic;  
 margin: 0;  
 padding: 0;  
 background-color: green;  
 text-align: center;  
}  
adm {  
 text-align: left;  
}  
style {  
 background-color: brown;  
 color: #fff;  
 padding: 1em 0;  
 text-align: center;  
}  
nav ul {  
 background-color: brown;  
 color: #fff;  
 text-align: center;  
 padding: 0em 0;  
 position: moving;  
 width: 100%;  
 bottom: 0;  
 text-align: center;  
}  
  
info {  
 padding: 1em;  
}  
section {  
 list-style-type: none;  
 padding: 0;  
 background-color: pink;  
 overflow: hidden;  
 text-align: center;  
}  
cars {  
 background-color: pink;  
 color: #fff;  
 padding: 1em 0;  
 text-align: center;  
}

1. **Создаём в файле models модели для таблиц, которые будет создана в базе данных для применения далее в работе хранения информации по товарам и для работы корзины:**

**В приложении ‘cart’ в файле models создаём модели:**

class Category(models.Model):  
  
 name = models.CharField(max\_length=150, db\_index=True, verbose\_name='Название категории')  
 slug = models.SlugField(max\_length=150, unique=True)  
  
 class Meta:  
 ordering = ('name',)  
 verbose\_name = 'Категория'  
 verbose\_name\_plural = 'Категории'  
  
 def \_\_str\_\_(self):  
 return self.name  
  
 def get\_absolute\_url(self):  
 return reverse('shop:product\_list\_by\_category', args=[self.slug])  
  
class Product(models.Model):  
 category = models.ForeignKey(Category, related\_name='products', on\_delete=models.CASCADE,  
 verbose\_name='Выберите категорию')  
 name = models.CharField(max\_length=200, db\_index=True, verbose\_name='Наменование')  
 slug = models.SlugField(max\_length=200, db\_index=True)  
 image = models.ImageField(upload\_to='products/%Y/%m/%d', blank=True, verbose\_name='Фото')  
 description = models.TextField(blank=True, verbose\_name='Описание')  
 price = models.DecimalField(max\_digits=15, decimal\_places=2, verbose\_name='Цена')  
 available = models.DecimalField(max\_digits=15, decimal\_places=2, verbose\_name='Наличие')  
 created = models.DateTimeField(auto\_now\_add=True)  
 updated = models.DateTimeField(auto\_now=True)  
  
 class Meta:  
 ordering = ('name',)  
 #index\_together = (('id', 'slug'),)  
 verbose\_name = 'Продукт'  
 verbose\_name\_plural = 'Продукты'  
  
 def \_\_str\_\_(self):  
 return self.name  
  
 def get\_absolute\_url(self):  
 return reverse('shop:product\_detail', args=[self.id, self.slug])  
  
from django.db import models  
from django.contrib.auth.models import User  
  
  
class CartItem(models.Model):  
 product = models.ForeignKey(Product, on\_delete=models.CASCADE)  
 quantity = models.PositiveIntegerField(default=0)  
 user = models.ForeignKey(User, on\_delete=models.CASCADE)  
 date\_added = models.DateTimeField(auto\_now\_add=True)  
  
 def \_\_str\_\_(self):  
 return f'{self.quantity} x {self.product.name}'  
  
category - является экземпляром ForeignKey, содержит ключ на другую запись в базе данных, в нашем случае на Category. Все создаваемые продукты будут относится к какой-нибудь категории. on\_delete=models.CASCADE - сообщает Django, что при удалении категории, будут удаляться все продукты, относящиеся к данной категории. related\_name='products' - указывает имя обратной связи, так мы легко получим доступ ко всем продуктам категории.   
name - название продукта. Содержит данные CharField - строковое поле ограниченной длины, в нашем случае максимум 200 символов.   
slug - служит для построения удобочитаемых URLов.  
image - фотография товара. Так как в параметрах стоит blank=True, значит поле не обязательно к заполнению.   
description - описание вашего продукта  
price - цена товара. Тип поля DecimalField, которое хранит значение с фиксированной точностью. Парамаетр max\_digits задает общее количество цифр, включая десятичные, decimal\_places - количество цифр после запятой.   
available - наличие товара на складе. Тип BooleanField, так как есть два варианта True - в наличие, False - отсутствует на складе.  
created - дата и время создания товара  
updated - дата и время последнего изменения   
Класс Meta, метод \_\_str\_\_ и get\_absolute\_url несут такой же функционал, как и в модели Category.

1. **Создаём для взаимодействия с базой данных, заполнения таблиц по товарам в базу данных и работы корзины “админ панель” в файле админ прописываем(создаём и регистрируем):**

**admin.ModelAdmin класс позволяет работать с моделями для админ панелей от него все наследуется.**

from django.contrib import admin  
from .models import Category, Product  
  
class CategoryAdmin(admin.ModelAdmin):  
 list\_display = ['name', 'slug'] прописываем поля для админ панели  
 prepopulated\_fields = {'slug': ('name',)} прописываем поля для админ панели можно делать поиск  
  
admin.site.register(Category, CategoryAdmin)  
  
class ProductAdmin(admin.ModelAdmin):  
 list\_display = ['name', 'slug', 'price', 'available', 'created', 'updated']  
 list\_filter = ['available', 'created', 'updated']  
 list\_editable = ['price', 'available']  
 prepopulated\_fields = {'slug': ('name',)}  
  
admin.site.register(Product, ProductAdmin)

from .models import CartItem  
admin.site.register(CartItem)

После миграций, миграции пропишут, нужные данные в программе по зарегистрированным модулям админ и мы создадим суперпользователя командой **python manage.py createsuperuser** (см. ниже в миграциях)

1. **Создаём с помощью Django Form модели для создаём блок взаимодействия сайта и пользователя. На примере Регистрации пользователя.**

**Создаём файл forms.py в структуре ‘brdbcarsapp’ импортируем из Django -> forms. Логика прописана в файле views.py в файле HTML в приложении ‘brdbcarsapp’ (см. выше в тексте):**

**brdbcarsapp/forms:**

from django import forms  
from django.forms import models  
  
  
class UserRegister(forms.Form):  
 username = forms.CharField(max\_length=30, label='Введите логин:')  
 password = forms.CharField(max\_length=8, label='Введите пароль:', widget=forms.PasswordInput)  
 repeat\_password = forms.CharField(max\_length=8, label='Подтвердите пароль:', widget=forms.PasswordInput)  
 age = forms.CharField(max\_length=3, label='Введите свой возраст:')

1. **Миграции, база данных.**

Первая миграция Django создаёт первые технические таблицы в базе данных.

Далее будем прописывать модели таблиц для нашей базы данных с товарами.

Команда в терминале:

**python manage.py migrate**

Активировался файл базы данных. **db.sqlite3**

Зайдя в него через программу DB Browse, видим в терминале что все сработало нормально. Проверяем в базе данных видим что Django создал 11 первых технических таблиц.

По созданным моделям в models.py **в** приложении **‘cart’** будут созданы таблицы после мигруций **.**

Таблицы создаются через классы. Каждый класс это отдельная таблица.

И каждый класс наследуется от класса Model. Каждая переменная создана под каждое поле.

По созданным моделям, проводим миграции для создания таблиц:

Команда в термирале:

**python manage.py makemigrations**

**python manage.py migrate**

Так же нам понадобится:

python -m pip install Pillow

Создаём супер пользователя для работы с **админ панелью** с именем и паролем.

python manage.py createsuperuser

Получаем:

Username (leave blank to use 'dns'): svetlana

Email address: info@egyn.ru

Password:

Password (again):

This password is too common.

This password is entirely numeric.

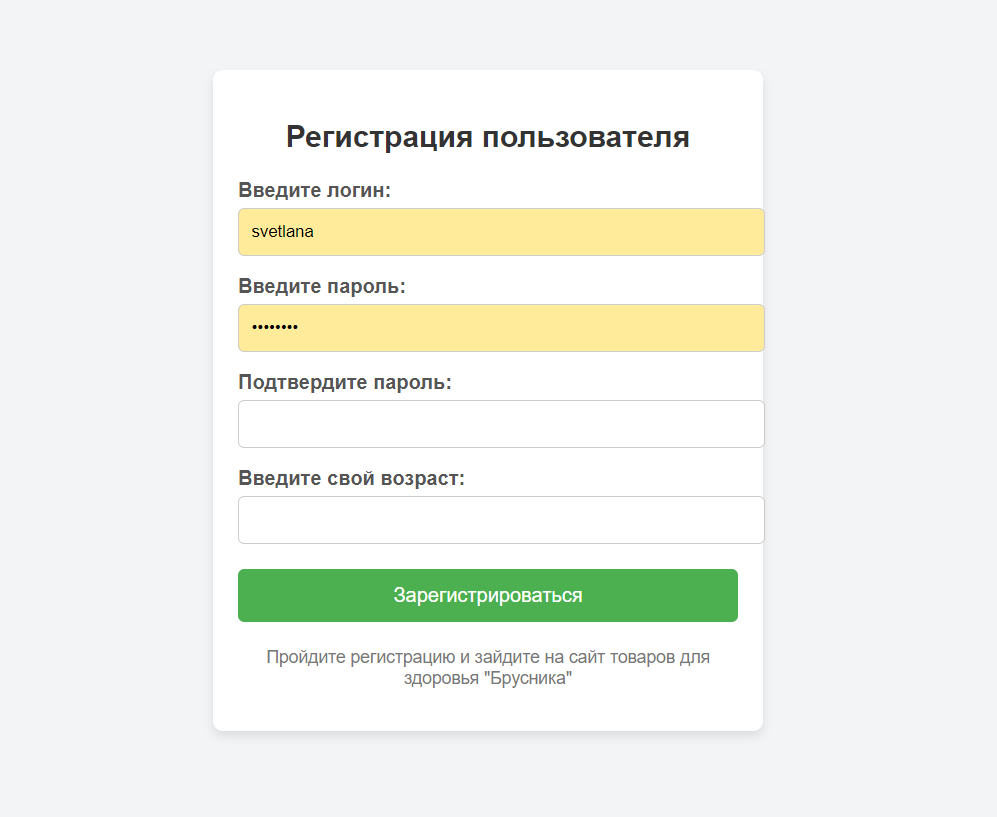
Bypass password validation and create user anyway? [y/N]: y

Superuser created successfully.

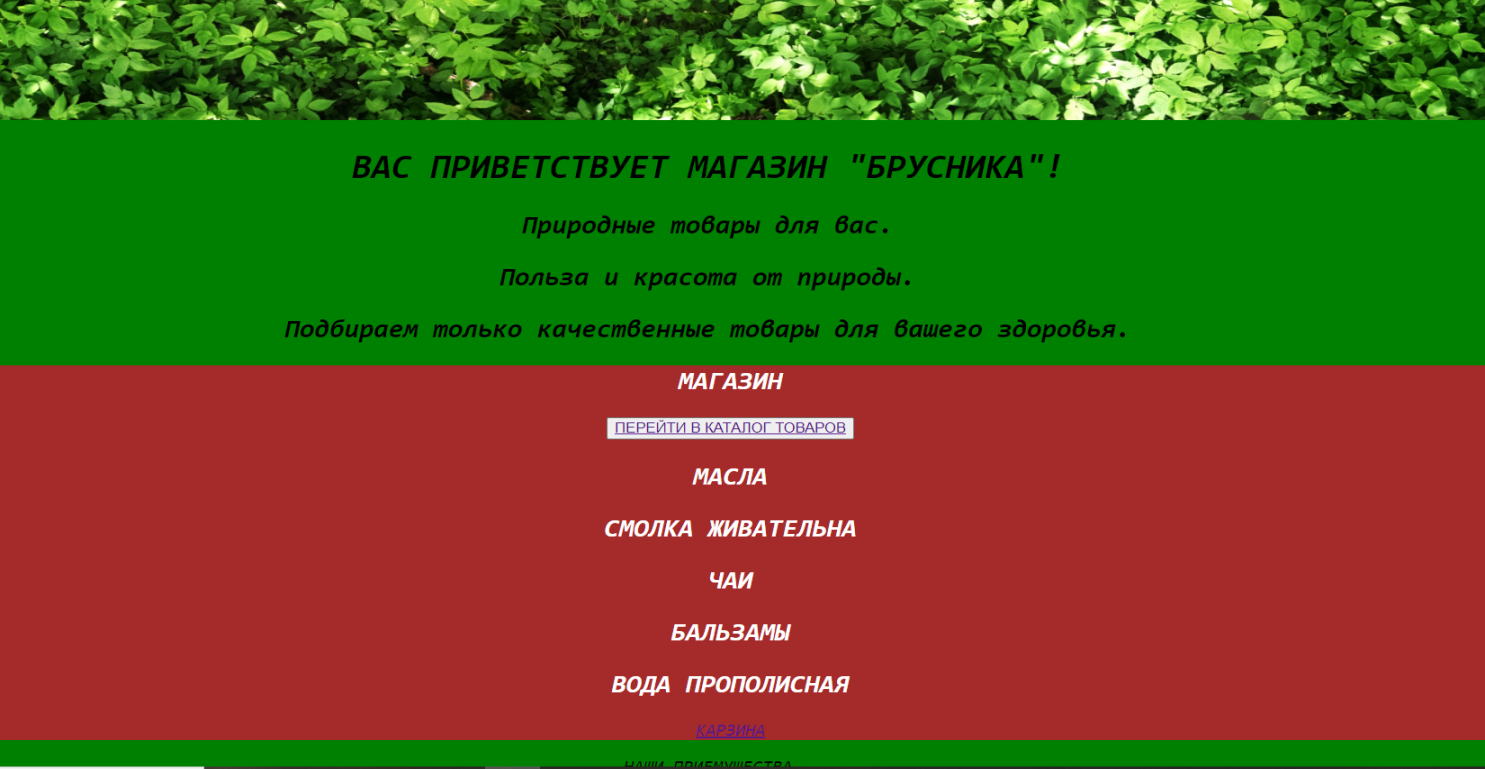
Теперь сможем входить в админ панель и вносить категории товары формировать базу данных. Наш сайт начинает наполняться товарами и работать.

**Создав проект по всем правилам Django мы получем простое приложение в виде Интеренет-магазина «Брусника» с симпатичным видом и у нас появился функционал в виде «Входа в магазин», «Главная страница», «входа и работы административного блока», магазин с товарами и корзина в дальнейшем можно добавить блок оплаты.**

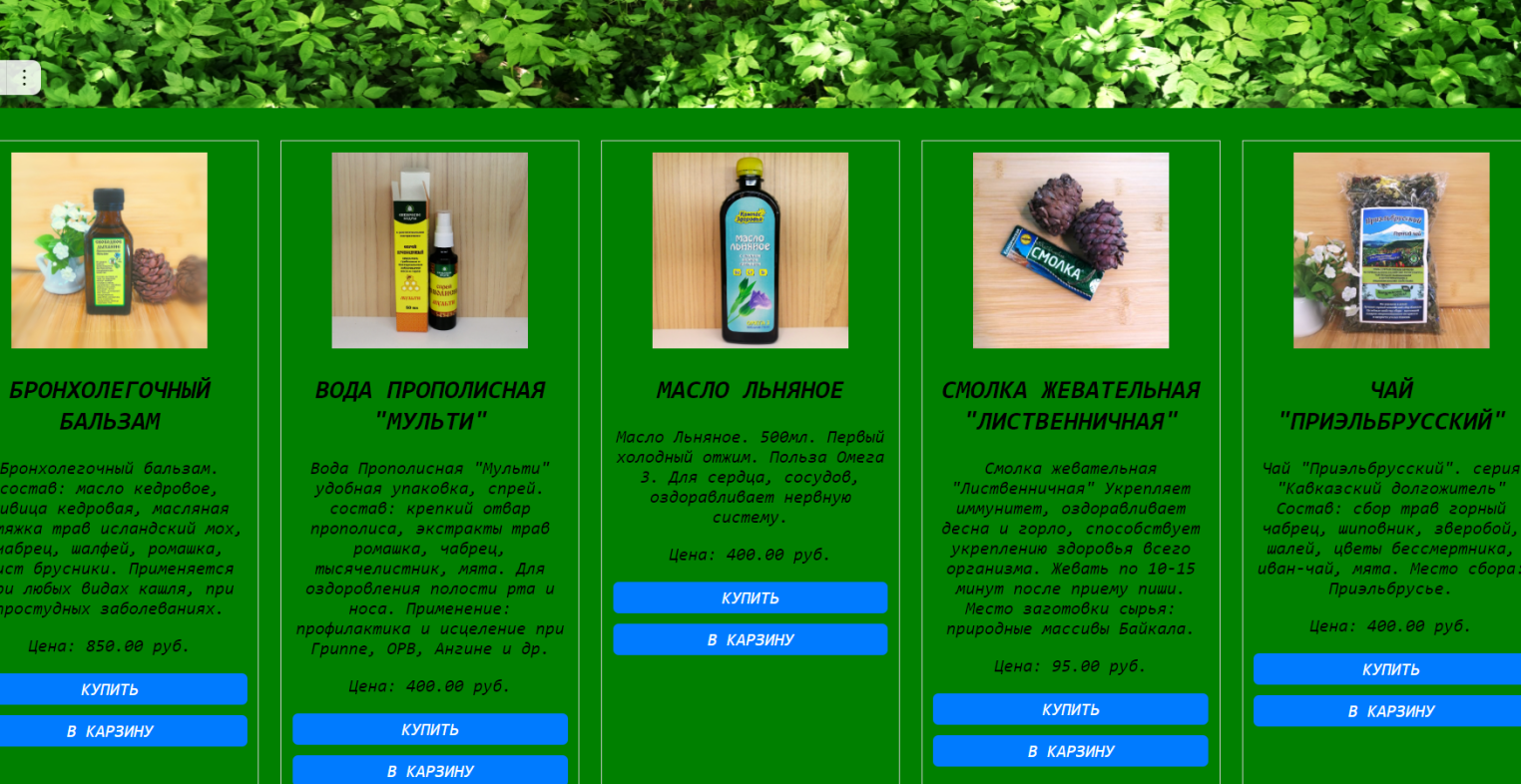
**Регистрация для входа на сайт:**



**Главная страница:**

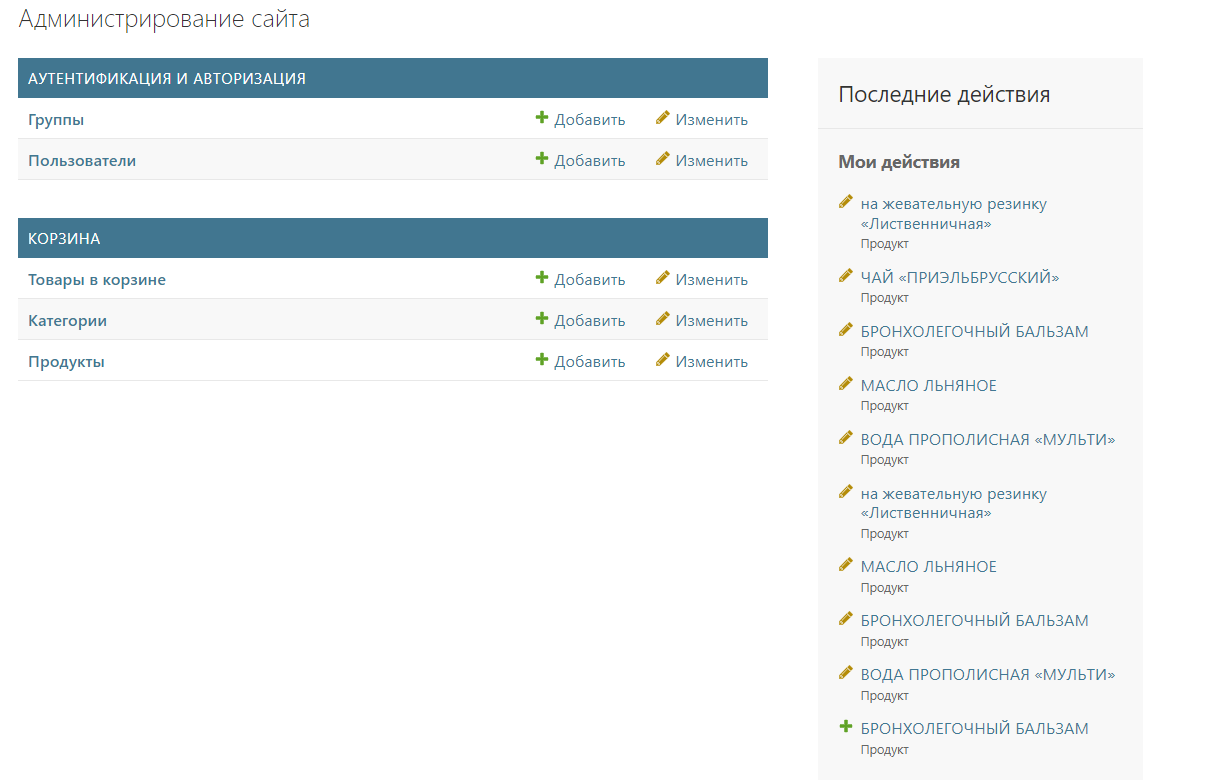


**Магазин «Брусника»:**



**Карзина:**



**Админ:**

**Структура проекта:**

Папка brusshika

папка brdbcarsapp

папка migrations отображаем миграции изменения в приложении

\_\_init\_\_.py указывает что приложение пакет

admin.py для админ панели для нашего сайта

apps.py с настройками нашего приложения

forms.py прописываются формы с назначением

models.py модели базы данных для приложения

tests.py тест для тестов

views.p прописывается логика для данного приложения и в целом

папка brusshika

\_\_init\_\_.py указывает на то что нашпроект пакет

asgi.py для запуска нашего приложения на стороннем сервере

settings.py файл с настройками проекта

urls.py маршрутизация по страницам

wsgi.py для записка приложения на сторонем сервере

папка cart

папка migrations отображаем миграции изменения в приложении

\_\_init\_\_.py указывает что приложение пакет

admin.py для админ панели для нашего сайта

apps.py с настройками нашего приложения

models.py модели базы данных

tests.py тест для тестов

urls.py спец файл для приложения cart с маршрутами

views.py прописывается логика для приложения

папка media

папка products содержит фото, видео для проекта

IMG\_20240403\_172638.jpg

папка static

style.css содержит стили и фото, видео проекта

папка templates содержит HTML файлы для проекта

<>cart.html

<>index.html

<>info.html

<>menu.html

<>pokupka.html

<>registration\_page.html

=db.sqlite3 файл база данных

manage.py

папка venv окружение

**Вывод: Мы рассмотрели работы по созданию сайтов в Django, на простом примере.**

**И это достаточно интересно, просто, функционально и удобно.**

**Далее рассмотрим создание приложений в других фреймворках.**

**FastAPI. Создаём простое приложение с базой данных для сайта интернет-магазин Брусника.**

**Начинаем проект с установки среды окружения.**

1.Создаём новый проект в Python.

Название: **ProjectKatalogTovarovBRUCNIKA*.***

1. Создём среду окружения.

**python -m venv venv**

3.Активируем python

**venv\scripts\activate**

4.Проверяем установку Python в настройках и проверяем активацию среды и активацию Python. И что все сохраняется в нужный нам каталог/файл.

Все установилось и работает. Продолжаем работу над нашим проектом.

5.Устанавливаем в созданную среду окружение нужно нам программное обеспечение через pip.

Устанавливаем uvicorn

**pip install uvicorn**

Uvicorn — это реализация веб-сервера ASGI (Asynchronous Server Gateway Interface) для Python.

ASGI предоставляет общий набор инструментов, применимых ко всем асинхронным фреймворкам. Uvicorn обеспечивает поддержку HTTP/1.1 и WebSockets.

Uvicorn позволяет запускать асинхронные приложения на Python, например, FastAPI.

**Так же применим в проекте для ORM баз данных SQL Aлхимия.**

Устанавливаем fastapi.

**pip instell fatsapi**

**При создании проекта будем импортировать нужные нам инструменты:**

**from fastapi import Fastapi**

**from fastapi import APIRouter, Depends, status, HTTPException** - позволяет организовывать операции с путями в приложении FastAPI, а система Dependency Injection упрощает интеграцию различных компонентов.

**from sqlalchemy.orm import Session** - взаимодействия с базой данных

**from typing import Annotated** -  добавлять метаданные к существующим типам.

**from sqlalchemy import insert** - создавать инструкции для вставки данных в базу данных.

**from slugify import slugify** - функция преобразует строку в удобный для URL слаг.

**from sqlalchemy import select** - выполнять SELECT-запросы, принимая список таблиц или колонок, из которых требуется получить данные.

**from sqlalchemy import update** - обновлять данные, изменяя значения столбцов по заданным условиям

**from pydantic import BaseModel** - класс для наследования и построения моделей **Pydantic**.  
**from sqlalchemy import create\_engine** - создание глобального объекта, движка, создаётся один раз для взаимодействий с сервером.  
**from sqlalchemy.orm import sessionmaker, DeclarativeBase** -создание объектов сессий для безопасное управлени соединениями и класс моделей для работы с таблицами  
**from sqlalchemy import Column, Integer, String** - классы используются для определения атрибутов-столбцов в моделях SQLAlchemy, задавая тип данных для каждого столбца таблицы.

**from sqlalchemy.schema import CreateTable** - конструктор позволяет создавать операторы CREATE TABLE, взаимодействует с таблицами, объясняя что нужно делать.,

**from sqlalchemy.orm import relationship** - для установки отношений между моделями в SQLAlchemy

**Первоначально создаём структуру:**

app -главная папка

routers - в папке app

category.py - файл в папке routers

products.py - файл в папке routers

\_\_init\_\_.py - файл указывает что папка пакет

\_\_init\_\_ - файл в папке app ( папка пакет)

main.py - файл в пакете app

**В файле main создаём основу для подключения.**

from fastapi import FastAPI  
from app.routers import category  
from app.routers import products  
  
app = FastAPI()  
  
@app.get("/")  
async def welcome() -> dict:  
 return{"message": "My shop"}  
  
app.include\_router(category.router)  
app.include\_router(products.router)

**Запускаем приложение:**

**python -m uvicorn main:app**

Получаем:

**INFO: Started server process [11976]**

**INFO: Waiting for application startup.**

**INFO: Application startup complete.**

**INFO: Uvicorn running on http://127.0.0.1:8000**

Связь установлена с приложением, все работает. Пишем проект далее.

**Создаём файл schemas.py:** - модели которые используются для хранения.

from pydantic import BaseModel  
  
class CreateProduct(BaseModel):  
 name: str  
 description: str  
 price: int  
 image\_url: str  
 stock: int  
 category: int  
  
class CreateCategory(BaseModel):  
 name: str  
 parent\_id: int

**Далее создаём БД и подключение к БД применением sqlalchemy:**

**В структура дополняем папку backend. В папку добавляем файлы.**

Папка backend

\_\_init\_\_.py

db.py

db\_depends.py

**В файле db.py:**

from sqlalchemy import create\_engine  
from sqlalchemy.orm import sessionmaker, DeclarativeBase  
from sqlalchemy import Column, Integer, String  
  
engine = create\_engine("sqlite:///ecommerce.db", echo = True)  
  
SessionLocal = sessionmaker(bind=engine)  
  
class Base(DeclarativeBase):  
 pass

**В файл db\_depends.py прописывается функция для запуска.**

from .db import SessionLocal  
  
async def get\_db():  
 db = SessionLocal()  
 try:  
 yield db  
 finally:  
 db.close()

**Далее:**

**В структуру добавляем ещё одну папку models. В ней прописываем файлы и взаимодействие между базами данных.**

**Один к одному, один ко могим и многие ко многим.**

**Рассмотрим в проекте один к одному, один ко многим.**

Папка **models**

\_\_init\_\_.py

category.py

products.py

**Файл models/ \_\_init\_\_.py прописываем:**

from app.models.products import Product  
from app.models.category import Category

**Файл models/category.py прописываем:**

from app.backend.db import Base  
from sqlalchemy import Column, Integer, String, Boolean, ForeignKey, Float  
from sqlalchemy.orm import relationship  
from app.models import \*  
  
class Category(Base):  
 \_\_tablename\_\_ = "categories"  
 \_\_table\_args\_\_ = {'extend\_existing': True}  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String)  
 slug = Column(String, unique=True, index=True)  
 is\_active = Column(Boolean, default=True)  
 parent\_id = Column(Integer, ForeignKey("categories.id"), nullable= True)  
  
 products = relationship("Product", back\_populates="category")  
  
from sqlalchemy.schema import CreateTable  
print(CreateTable(Category.\_\_table\_\_))

**Файл models/products.py прописываем:**

from app.backend.db import Base  
from sqlalchemy import Column, ForeignKey, Integer, String, Boolean, Float  
from sqlalchemy.orm import relationship  
from app.models import \*  
  
class Product(Base):  
 \_\_tablename\_\_="products"  
 \_\_table\_args\_\_ = {'keep\_existing': True}  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String)  
 slug = Column(String, unique=True, index=True)  
 description = Column(String)  
 price = Column(Integer)  
 image\_url = Column(String)  
 stock = Column(Integer)  
 category\_id = Column(Integer, ForeignKey('categories.id'))  
 rating = Column(Float)  
 is\_active = Column(Boolean, default=True)  
  
 category = relationship("Category", back\_populates="products")

**В файле category.py проектируем базу.**

**Прописываем роутинг. И др функции.**

from fastapi import APIRouter, Depends, status, HTTPException  
from sqlalchemy.orm import Session  
from backend.db\_depends import get\_db  
from typing import Annotated  
  
from models import \*  
from sqlalchemy import insert  
from schemas import CreateCategory  
#from schemas import CreateCategory  
  
from slugify import slugify  
from sqlalchemy import select  
from sqlalchemy import update  
  
from app.models.category import Category  
  
router = APIRouter(prefix="/category", tags=["category"])  
  
@router.post("/create")  
async def create\_category(db: Annotated[Session, Depends(get\_db)], create\_category: CreateCategory):  
 db.execute(insert(Category).values(name=create\_category.name,  
 parent\_id=create\_category.parent\_id,  
 slug=slugify(create\_category.name)))  
 db.commit()  
 return {  
 'status\_code':status.HTTP\_201\_CREATED,  
 'transaction': 'Successful'  
 }  
  
@router.get("/all\_categories")  
async def get\_all\_categories(db: Annotated[Session, Depends(get\_db)]):  
 categories = db.scalars(select(Category).where(Category.is\_active == True)).all()  
 return categories

@router.put("/update\_category")  
async def update\_category(db: Annotated[Session, Depends(get\_db)], category\_id: int, update\_category: CreateCategory):  
 category = db.scalars(select(Category).where(Category.id == category\_id))  
 if category is None:  
 raise HTTPException(  
 status\_code=status.HTTP\_404\_NOT\_FOUND,  
 detail='There is no category found'  
 )  
 db.execute(update(Category).where(Category.id == category\_id).values(  
 name=update\_category.name,  
 slug=slugify(update\_category.name),  
 parent\_id=update\_category.parent\_id))  
  
 db.commit()  
 return {  
 'status\_code': status.HTTP\_200\_OK,  
 'transaction': 'Category update is successful'  
 }  
  
@router.delete("/delete")  
async def delete\_category(db: Annotated[Session, Depends(get\_db)], category\_id: int):  
 category = db.scalar(select(Category).where(Category.id == category\_id))  
 if category is None:  
 raise HTTPException(  
 status\_code=status.HTTP\_404\_NOT\_FOUND,  
 detail='There is no category found'  
 )  
 db.execute(update(Category).where(Category.id == category\_id).values(is\_active=False))  
 db.commit()  
 return {  
 'status\_code': status.HTTP\_200\_OK,  
 'transaction': "Category delete is successful"  
 }

**Прописываем файл products.py**

from fastapi import APIRouter, Depends, status, HTTPException  
from sqlalchemy.orm import Session  
from backend.db\_depends import get\_db  
from typing import Annotated  
  
from models import \*  
from sqlalchemy import insert  
from schemas import CreateProduct  
  
from slugify import slugify  
from sqlalchemy import select  
from sqlalchemy import update  
  
from app.models.category import Category  
from app.models.products import Product  
  
router = APIRouter(prefix="/products", tags=["products"])  
  
@router.get("/")  
async def all\_products(db: Annotated[Session, Depends(get\_db)]):  
 products = db.scalars(select(Product).where(Product.is\_active == True, Product.stock > 0)).all()  
 if products is None:  
 return HTTPException(  
 status\_code=status.HTTP\_404\_NOT\_FOUND,  
 detail="There are no product"  
 )  
 return products  
  
@router.post("/create")  
async def create\_products(db: Annotated[Session, Depends(get\_db)], create\_product: CreateProduct):  
 db.execute(insert(Product).values(name=create\_product.name,  
 description=create\_product.description,  
 price=create\_product.price,  
 image\_url=create\_product.stock,  
 category\_id=create\_product.category,  
 rating=0.0,  
 slug=slugify(create\_product.name)))  
 db.commit()  
 return {  
 'status\_code':status.HTTP\_201\_CREATED,  
 'transaction': 'Successful'  
 }  
  
@router.get('/{category\_slug}')  
async def product\_by\_category(db: Annotated[Session, Depends(get\_db)], category\_slug: str):  
 category = db.scalar(select(Category).where(Category.slug == category\_slug))  
 if category is None:  
 raise HTTPException(  
 status\_code=status.HTTP\_404\_NOT\_FOUND,  
 detail="Category not found"  
 )  
 subcategories = db.scalars(select(Category).where(Category.parent\_id == category.id)).all()  
 categories\_and\_subcategories = [category.id] + [i.id for i in subcategories]  
 products\_category = db.scalars(  
 select(Product).where(Product.category\_id.in\_(categories\_and\_subcategories), Product.is\_active,  
 Product.stock > 0)).all()  
 return products\_category  
  
@router.get('/detail/{product\_slug}')  
async def product\_detail(db: Annotated[Session, Depends(get\_db)], product\_slug: str):  
 product = db.scalar(  
 select(Product).where(Product.slug == product\_slug, Product.is\_active == True, Product.stock > 0))  
 if not product:  
 return HTTPException(  
 status\_code=status.HTTP\_404\_NOT\_FOUND,  
 detail='There are no product'  
 )  
 return product  
  
@router.put("/datail/{product\_slug}")  
async def update\_product(db: Annotated[Session, Depends(get\_db)], product\_slug: str, update\_product\_model: CreateProduct):  
 product\_update = db.scalar(select(Product).where(Product.slug == product\_slug))  
 if product\_update is None:  
 raise HTTPException(  
 status\_code=status.HTTP\_404\_NOT\_FOUND,  
 detail='There is no product found'  
 )  
 db.execute(update(Product).where(Product.slug == product\_slug).values(  
 name=update\_product\_model.name,  
 description=update\_product\_model.description,  
 price=update\_product\_model.price,  
 image\_url=update\_product\_model.image\_url,  
 stock=update\_product\_model.stock,  
 category\_id=update\_product\_model.category,  
 slug=slugify(update\_product\_model.name)))  
 db.commit()  
 return {  
 'status\_code': status.HTTP\_200\_OK,  
 'transaction': 'Category update is successful'  
 }  
  
@router.delete("/delete")  
async def delete\_product(db: Annotated[Session, Depends(get\_db)], product\_id: int):  
 product\_delete = db.scalar(select(Product).where(Product.id == product\_id))  
 if product\_delete is None:  
 raise HTTPException(  
 status\_code=status.HTTP\_404\_NOT\_FOUND,  
 detail='There is no category found'  
 )  
 db.execute(update(Product).where(Product.id == product\_id).values(is\_active=False))  
 db.commit()  
 return {  
 'status\_code': status.HTTP\_200\_OK,  
 'transaction': "Product delete is successful"  
 }

**МИГРАЦИИ**

**Это методология построения баз данных и внесения изменений.**

Применяем alembic.

**pip install alembic**

**Далее создаём начальные файлы**

**alembic init app/migrations**

**Появился в структуре проекта файл alembic**

в нем вносим изменения указываем адрес в строке

sqlalchemy.url = "sqlite:///ecommerce.db"

**Далее в блок env.py**

В target\_metadata = None

**Вносим изменения**

Получаем  
from app.models.category import Category  
from app.models.products import Product  
from app.backend.db import Base  
target\_metadata = Base.metadata

Далее программа будет обращаться к этим данным.

**Далее чтоб произвести миграцию пишем команду обращаемся к alembic в терминале:**

**alembic revision --autogenerate -m "Initial migration"**

Появилась база данных

ecommerce.db

Заходим в versions смотрим версию нашей миграции

44cc7f20d253\_initial\_migration.py

Можем если нужно вносить изменения в БД.

И далее делаем миграцию и программа вносит изменения.

**alembic upgrade head**

**В итоге структура проекта FastAPI:**

Папка app

Папака beckend

- \_\_init\_\_.py

- db.py.py

- db\_depends.py

Папка migrations

Папка models

- \_\_init\_\_

- category.py

- products.py

Папка routers

- \_\_init\_\_

- category.py

- products.py

\_\_init\_\_

- schemas.py

-alembic.ini

-ecommerce.db

-main.py

**При создании окружающей среды для FastApi установлены программы:**

**Команда в терминале:**

**pip freeze**

alembic==1.14.0

annotated-types==0.7.0

anyio==4.6.2.post1

click==8.1.7

colorama==0.4.6

fastapi==0.115.4

functions==0.7.0

greenlet==3.1.1

h11==0.14.0

idna==3.10

iniconfig==2.0.0

Mako==1.3.6

MarkupSafe==3.0.2

packaging==24.1

pluggy==1.5.0

pydantic==2.9.2

pydantic\_core==2.23.4

pytest==8.3.3

python-slugify==8.0.4

schemas==0.7.1

sniffio==1.3.1

SQLAlchemy==2.0.36

starlette==0.41.2

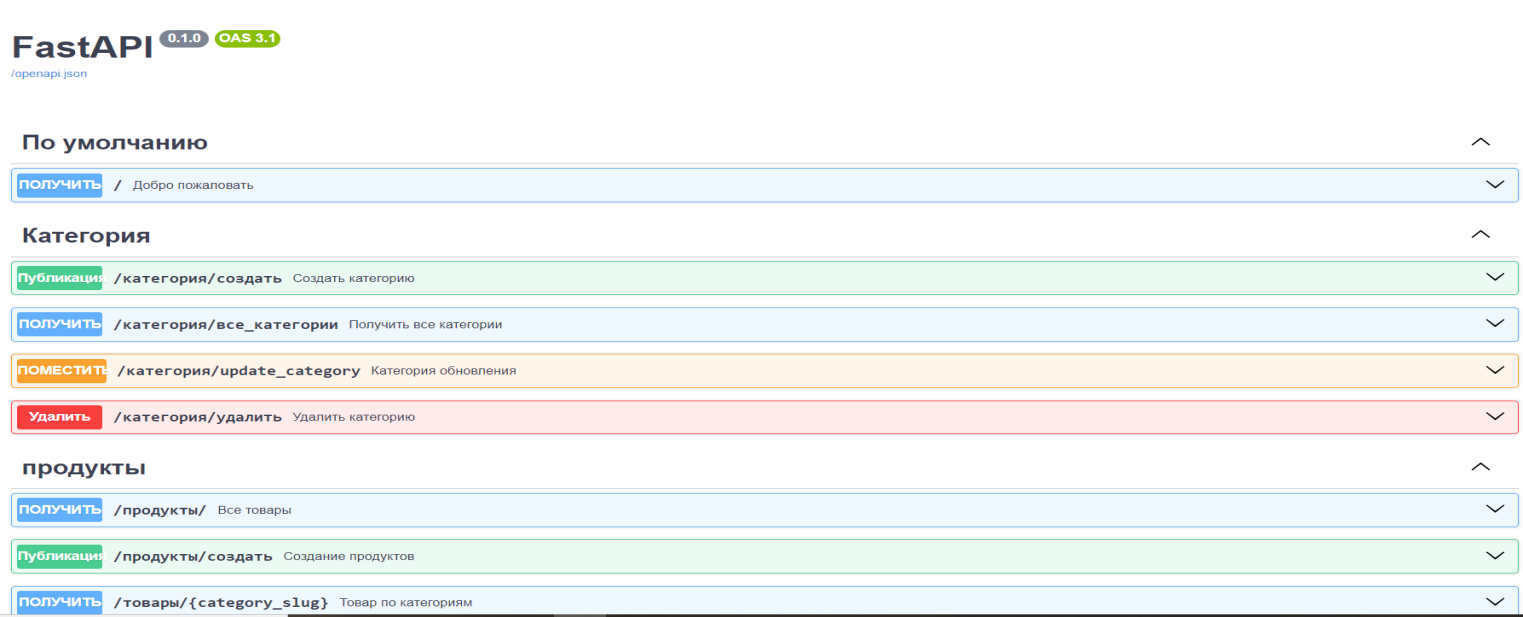
text-unidecode==1.3

typing\_extensions==4.12.2

uvicorn==0.32.0

(venv) PS C:\Urban\_Dip\FastAP

**Вывод:Создав проект с помощью фрайм ворка FastAPI получаем небольшой приложение с функционалом сознания в базе данных категорий товаров и продуктов и возможностью менять добавляя или удаляя позиции категорий или товаров.**



**Flask создаём простой проект учёт товаров.**

Пример изучения работы фреймворка взят с сайтов и др. видео материалов:

<https://habr.com/ru/articles/783574/>

https://proglib.io/p/samouchitel-po-python-dlya-nachinayushchih-chast-23-osnovy-veb-razrabotki-na-flask-2023-06-27/amp/

**Название проекта: DPFlasknovpr.**

1. Создаём среду окружения.

**python -m venv DPFlasknovpr\venv**

**Сохранение приложения внутри среды окружения предотвращает многие ошибки по взаимодействию. Далее активируем среду окружения с приложением.**

1. Активируем:

**venv\scripts\activate**

3.Проверяем установку Python в настройках и проверяем активацию среды и активацию Python. И что все сохраняется в нужный нам каталог/файл.

Все установилось и работает. Продолжаем работу над нашим проектом

4.Устанавливаем в созданную среду окружение нужно нам программное обеспечение через pip.

5.Устанавливаем Flask

**pip install flask**

1. Эта команда установит Flask и все необходимые зависимости.
2. Создаём файл requirements.txt (Этот файл часто используется с инструментами управления пакетами, такими как pip, для автоматической установки и управления зависимостями проекта.)

Команда в терминале:

**pip freeze > requirements.txt**

1. Структура проекта:

папка **DPFlasknovpr**

папка **DPFlasknovpr** основаная папка проекта

папка **instace** база данных создаёт программа

**= groceries.db**

папка **templates** папка для html файлов

**add.html**

**basa.html**

**edit.html**

**Index.html**

**Infopost.html**

**login.html**

**vhod.html**

**zakazi.html**

**app.py** файл логика проекта

**create\_db.py** файл создает базу даных

**models.py** файл формы для таблиц базы данных

**=requirements.txt**

1. **Необходимые программы для проекта:**

Можно посмотреть через команду в терминале: **pip freeze**

**blinker==1.9.0**

**click==8.1.7**

**colorama==0.4.6**

**Flask==3.1.0**

**Flask-SQLAlchemy==3.1.1**

**greenlet==3.1.1**

**itsdangerous==2.2.0**

**Jinja2==3.1.4**

**MarkupSafe==3.0.2**

**SQLAlchemy==2.0.36**

**typing\_extensions==4.12.2**

**Werkzeug==3.1.3**

1. Создаём в папке **DPFlasknovpr** файл основу проекта.

В папке создаём файл app.рy:

from flask import Flask

app = Flask(name)

@app.route('/')

def hello\_world():

return 'ДОБРО ПОЖАЛОВАТЬ!'

if \_\_name\_\_ ==\_\_ 'main'\_\_:

app.run(debug=True)

app - это основной файл приложения, где мы определяем маршруты, представления и логику. В этом файле вы создаёте экземпляр приложения Flask и запускаем сервер, проверяем работу.

Код создаёт в браузере путь и базовое веб-приложение, которое отображает 'ДОБРО ПОЖАЛОВАТЬ!' на главной странице.

Проверяем запускаем приложение.

Команда в терминале:

**python app.py**

Получаем:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\* Running on http://127.0.0.1:5000

\* Restarting with stat

\* Debugger is active!

\* Debugger PIN: 122-509-112

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверяем. Да приложение работает вывело на главную страницу сообщение:

'ДОБРО ПОЖАЛОВАТЬ!'

Flask предлагает гибкий и интуитивно понятный способ создания веб-приложений. Основой Flask является маршрутизация запросов и генерация ответов.

Теперь создаём логику проекта.

**Файл app для простой программы складского учёта.**  
from flask import Flask, request, render\_template, redirect  
from flask\_sqlalchemy import SQLAlchemy  
from models import db, Grocery  
  
app = Flask(\_\_name\_\_)  
app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///groceries.db'  
app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  
db.init\_app(app)  
  
@app.route('/', methods=['GET', 'POST'])  
def login():  
 if request.method == 'POST':  
 username = request.form['username']  
 password = request.form['password']  
 # проверка логина и пароля  
 return redirect('/vhod')  
 else:  
 return render\_template('login.html')

@app.route('/vhod')  
def vhod():  
 return render\_template('vhod.html')

@app.route('/infopost')  
def infopost():  
 return render\_template('infopost.html')

@app.route('/zakazi')  
def zakazi():  
 return render\_template('zakazi.html')

@app.route('/index', methods=['GET'])  
def index():  
 groceries = Grocery.query.all()  
 return render\_template('index.html', groceries=groceries)

@app.route('/add', methods=['GET', 'POST'])  
def add\_grocery():  
 if request.method == 'POST':  
 name = request.form['name']  
 description = request.form['description']  
 weight = float(request.form['weight'])  
 quantity = float(request.form['quantity'])  
 price = request.form['price']  
 groceries = Grocery(name=name, description=description, weight=weight, quantity=quantity, price=price)  
 db.session.add(groceries)  
 db.session.commit()  
 return redirect('/index')  
 return render\_template('add.html')  
  
@app.route('/edit/<int:id>', methods=['GET', 'POST'])  
def edit\_grocery(id):  
 grocery = Grocery.query.get\_or\_404(id)  
 if request.method == 'POST':  
 grocery.name = request.form['name']  
 grocery.description = request.form['description']  
 grocery.weight = float(request.form['weight'])  
 grocery.quantity = float(request.form['quantity'])  
 grocery.price = request.form['price']  
 db.session.commit()  
 return redirect('/')  
 return render\_template('edit.html', grocery=grocery)  
  
@app.route('/delete/<int:id>', methods=['POST'])  
def delete\_grocery(id):  
 patient = Grocery.query.get\_or\_404(id)  
 db.session.delete(patient)  
 db.session.commit()  
 return redirect('/index')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

**Файл models.py для формирования структуры таблиц для базы данных.**

from flask\_sqlalchemy import SQLAlchemy  
  
db = SQLAlchemy()  
  
  
class Grocery(db.Model):  
 id = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String(100), nullable=False)  
 description = db.Column(db.String(200), nullable=False)  
 weight = db.Column(db.Integer, nullable=False)  
 quantity = db.Column(db.Integer, nullable=False, default=0)  
 price = db.Column(db.Integer, nullable=False)  
  
 def \_\_repr\_\_(self):  
 return f"Patient(name='{self.name}', description='{self.description}', weight='{self.weight}', quantity='{self.quantity}', price='{self.price}')"

**Файл create\_db.py для создания базы данных на основе модели таблицы.**

**Устанавливаем:**

**pip install Flask-SQLAlchemy**

from flask import Flask  
from models import Grocery, db  
  
app = Flask(\_\_name\_\_)  
app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///groceries.db'  
db.init\_app(app)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with app.app\_context():  
 db.create\_all()  
 print('Создана база данных groceries')

**В папке templates создаём html модели которые будут выводить функционал в определённом виде на сервер сайта для пользователя для возможности работы с программой.**

add.html

basa.html

edit.html

Index.html

Infopost.html

login.html

vhod.html

zakazi.html

**Пример htmlфайла add.html:**

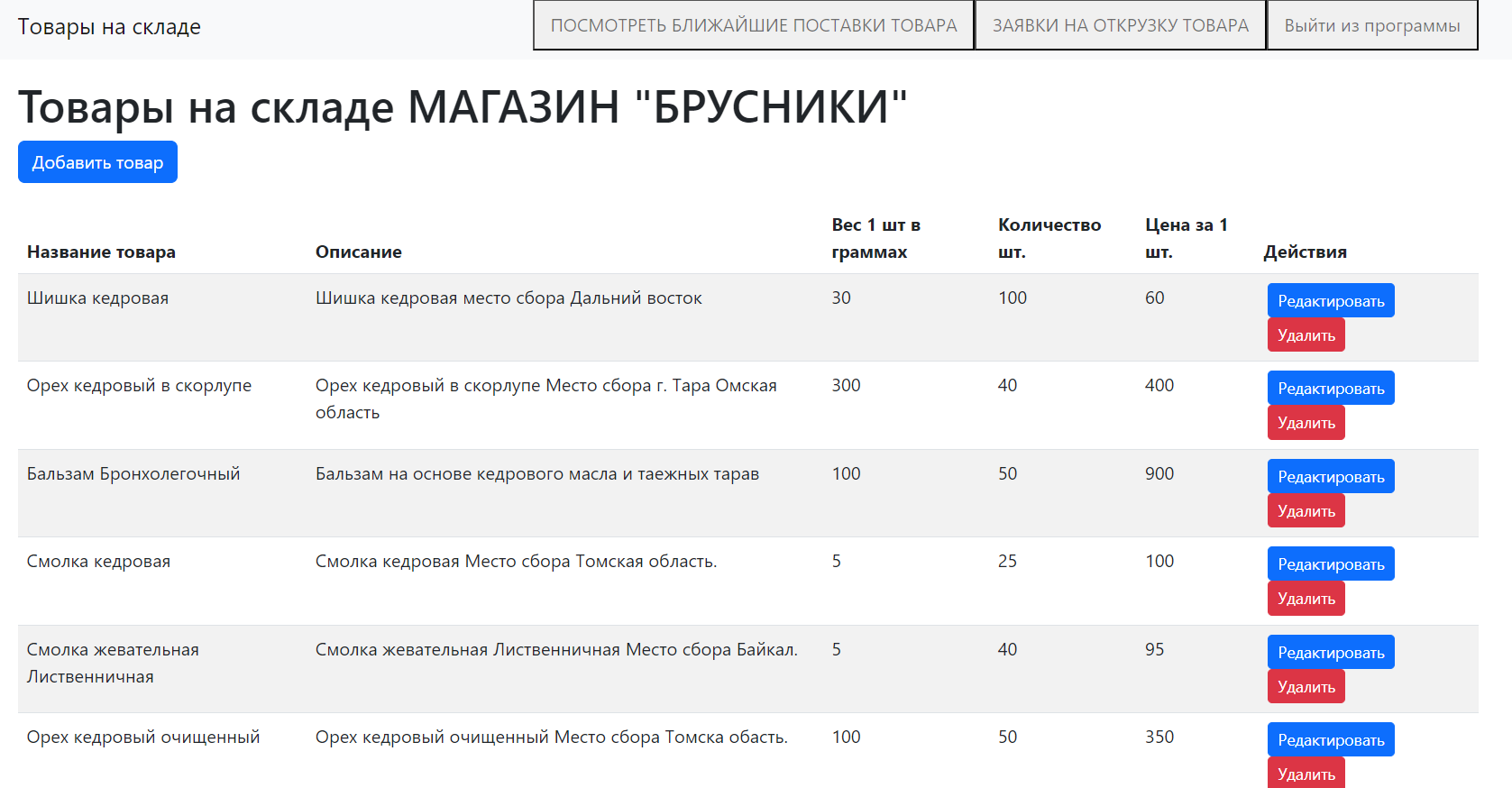
{% extends 'base.html' %}  
  
{% block content %}  
 <h1>Добавить товар</h1>  
 <div class="col-6">  
 <form method="post" action="{{ url\_for('add\_grocery') }}">  
 <div class="mb-3">  
 <label for="name" class="form-label">Название товара</label>  
 <input type="text" class="form-control" id="name" name="name" required>  
 </div>  
 <div class="mb-3">  
 <label for="description" class="form-label">Описание</label>  
 <textarea class="form-control" id="description" name="description" rows="3" required></textarea>  
 </div>  
 <div class="mb-3">  
 <label for="weight" class="form-label">Вес 1 шт в граммах</label>  
 <input type="text" class="form-control" id="weight" name="weight" required>  
 </div>  
 <div class="mb-3">  
 <label for="quantity" class="form-label">Количество шт.</label>  
 <input type="text" class="form-control" id="quantity" name="quantity" required>  
 </div>  
 <div class="mb-3">  
 <label for="price" class="form-label">Цена за 1 шт.</label>  
 <input type="text" class="form-control" id="price" name="price" required>  
 </div>  
 <button type="submit" class="btn btn-primary">Добавить</button>  
 </form>  
 </div>  
{% endblock %}

**Создав все папки и файлы в проекте. И запустив проект на сервере получаем простую программу по учёту товаров в которую можно заносить товары, редактировать данные, удалять. И просмотреть заказы и поставки.**

**Применили стили для создания приятного вида проэкта.**

<!-- Bootstrap стили -->  
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">  
<!-- иконки fontawesome -->  
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">  
<!-- кастомные стили -->  
<link rel="stylesheet" href="{{ url\_for('static', filename='custom.css') }}">

Страница работы с товарами Добавить, Редактирвать, Удалить, просмотреть другую информацию:



**Вывод: Flask показался мне лёгким, но при этом мощный инструмент для создания веб-приложений на Python. Его простота и гибкость делают очень привлекательным при выборе фреймворка при разработке. Приятно работать с таким фреймворком.**

**7.Подведение итогов по разделу «Разработка в соответствии с созданной документацией»:**

Все фреймворкиDjango, Flask и FastAPI по праву занимают место в разработке веб-приложений в нашем современном мире. Можно создавать приложения и Django и в Flask это очень удобно и функционально.

В FastAPI удобно создавать приложения объединяющие различные программы.

Материалы изложенному ниже представлены показывают что в дальнейшем можно объединить работу приложения написанного в Django и базу данных товаров написанную в FastAPI.

Это один из предметов для моего дальнейшего изучения работы фреймворков.

1.Ссылка на github проект объединения веб- приложения Django c проектом в FatsAGI.

<https://github.com/drmacsika/fastapi-django-combo>

2.Учебник как использовать Django c фреймворкам FastAPI. «Как использовать FastAPI с Django ORM и администратором»

Ссылка:

https://nsikakimoh.com/learn/django-and-fastapi-combo-tutorials

**8.Анализ и интерпретация результата.**

**Рекомендации по выбору модели в зависимости от проекта.**

**Написав простые проекты для каждого из фреймворков могу заключить следующее,**

**выбор фреймворка** Django, Flask и FastAPI зависит от конкретных потребностей и требований проекта.

**Django** подходит для крупных проектов со сложной функциональностью, включая системы управления контентом, платформы социальных сетей и сайты электронной коммерции. Например Социальная сеть. Для неё можно использовать архитектуру вертикального среза. Каждый модуль будет включать все необходимые слои для выполнения определённой функциональности, например, профиль пользователя, лента новостей, сообщения и уведомления. Безопасность: этот фреймворк уделяет значительное внимание защите от угроз.

Очень интересный фреймворк достаточно лёгкий, если поглубже погрузиться в документацию и поднабраться опыты можно с интересом работать создавая приложения.

**Flask** — лёгкий и гибкий микрофреймворк для небольших и сред. Подойдёт например для Интернет-магазина или для программы учета товаров на складе. Для интернет-магазина можно использовать микросервисную архитектуру. Каждый сервис будет отвечать за определённую функциональность, например, управление пользователями, каталог товаров, корзину покупок и обработку заказов. Это позволит легко масштабировать систему и обновлять отдельные сервисы без влияния на другие. Написав приложение в Python можно достаточно быстро интегрировать разрабатываемый проект с другими веб-приложениями либо обеспечить его взаимодействие с существующим API, этот фреймворк — прекрасный выбор для решения задачи.

**FastAPI**— быстрый и производительный веб-фреймворк Python, который подходит для создания высокопроизводительных API и микросервисов. Он поддерживает асинхронное программирование и имеет высокую пропускную способность.

Применять можно для Корпоративной системы управления. Для неё можно использовать сервис-ориентированную архитектуру. Общие сервисы, такие как аутентификация, управление пользователями и отчётность, будут использоваться различными бизнес-приложениями. Объединить все эти сервисы вполне можно с помощью FastAPI прописав программу -язык для удобной интеграции между собой этих сервисов и разных программ.

**9.Используемая литература:**

1. Обучающие материалы Университет Urban.

2.Ссылка Пример изучения работы фреймворка взят с сайта:

<https://habr.com/ru/articles/783574/>

3.Ссылка на github проект объединения веб- приложения Django c проекторм в FatsAGI.

<https://github.com/drmacsika/fastapi-django-combo>

4.Учебник как использовать Django c фреймворкам FastAPI.

Ссылка:https://nsikakimoh.com/learn/django-and-fastapi-combo-tutorials

5.Ссылка по HTML шаблонам для написания файлов HTML/

<https://uguide.ru/tablica-osnovnykh-tegov-html-s-primerami>

1. Учебник Python сборник упражнений

Ссылка: <https://drive.google.com/file/d/1oyT9kMusk3Iaj6zB-8MTb94Q-ctx6oMS/view>

1. Профессиональная работа в пайтон.

Ссылка:https://drive.google.com/file/d/1J9Wu53bWaXhzph9KlBr6VmslmC3vVmAZ/view

1. Определения, описания фреймворков взяты из открытых источников с просторов интернета.