Geekbrains

**Создание веб-приложения для организации мероприятий**

**и автоматического анализа отзывов о событие.**

Автор:

Родонов А.Ю.

Россия

2024

**Содержание**

1. Введение ------------------------------------------------------------------- 4

1.1 Актуальность проекта --------------------------------------------------- 7

1.2 Цели и задачи проекта -------------------------------------------------- 9

1.4 Инструменты разработки ---------------------------------------------- 10

2. Теоретическая часть ----------------------------------------------------- 11

2.1 Структура приложения ------------------------------------------------ 11

2.2 Выбор среды разработки ---------------------------------------------- 12

2.3 Обновление базы данных ----------------------------------------------13

2.4 Базовый функционал --------------------------------------------------- 13

2.5 Расширенный функционал -------------------------------------------- 13

3. Практическая часть ------------------------------------------------------ 14

3.1 Подготовка среды разработки --------------------------------------- 14

3.2 Установка необходимых библиотек ------------------------------- 19

3.2 Разработка приложения ----------------------------------------------- 21

3.3 Создание конфигурационных файлов Flask ---------------------- 24

3.4 Создание файла окружения .env ------------------------------------ 25

3.5 Создание config.py ----------------------------------------------------- 26

3.6 Модуль расширений extensions ------------------------------------- 27

3.7 Создание библиотеки models ---------------------------------------- 28

3.8 Создание модуля forms ----------------------------------------------- 31

3.9 Инициализация Migrate ---------------------------------------------- 34

3.10 Создание модулей main и user библиотеки routes -------------- 34

3.11 Создание модуля post библиотеки routes ------------------------ 39

3.12 Создание модуля comment.py библиотеки routes -------------- 45

3.13 Фронтенд секция. Создание и наполнение static --------------- 47

3.14 Фронтенд секция. Разработка библиотеки templates ----------- 52

3.15 Разработка шаблонов библиотеки templates/user -------------- 56

3.16 Разработка шаблонов библиотеки templates/post ------------- 57

3.17 Разработка шаблонов библиотеки templates/comment ------- 61

3.18 Предоставление прав доступа ------------------------------------- 63

3.19 Итоги разработки --------------------------------------------------- 63

4. Добавление расширенного функционала -------------------------- 64

4.1 Языковая модель ------------------------------------------------------ 64

4.2 Тестирование языковой модели ------------------------------------ 68

5. Подготовка проекта к внедрению ------------------------------------ 69

6. Перспективы дальнейшей проработки проекта ------------------- 74

7. Итоге проекта ------------------------------------------------------------- 74

Список используемых источников --------------------------------------- 75

1. **Введение**

Современное общество уверенно вступило в цифровую эпоху. Сегодня уже невозможно представить ни одно сообщество или организацию людей не использующей информационные технологии в своей повседневной жизни и работе.

Цифровые системы связи и социальные сети становятся основными каналами связи, обмена и взаимодействия между людьми.

По сути у общества 21-го века появилась еще одна базовая потребность на уровне необходимости в еде, жилье, сне, безопасности и стабильности.

Можно констатировать, что в последнее десятилетие развитие информационных технологий привело к значительным изменениям во сферах экономики и общественной жизни. Выражение «если твоего бизнеса нет в интернете — то вас нет в бизнесе» сегодня можно перенести уже на уровень отдельной личности.

Инструменты цифровой экономики становятся важнейшим средство социально-экономического развития, модернизации инфраструктуры и прогресса самого человечества. С развитием систем искусственного интеллекта эти процессы несоизмеримо ускоряются.

Вместе с огромными возможностями данная тенденция так же несет в себе серьезные фундаментальный риски для общества. Само развитие социума, социальных институтов и ментальных навыков отдельного человека не поспевают за цифровизацией мира.

Уже сегодня существенная часть общества воспринимающая процессы цифровизации как абстрактное явление не подконтрольное их воле, что приводит к отстраненному восприятию нового цифрового окружения и принятию его «as is». Этот юридический термин является основой лицензионных соглашений, подчас настолько сложных для понимания простыми пользователями и в конечном счете сводящимся к принципу «принимаешь то что тебе дают или ты не получишь ничего».

Люди начинают относиться к инструментам от которых зависит их жизнь, как к повседневной магии - «оно как то работает» и пока работает - их это устраивает.

Здесь же проявляется вторая актуальная проблема повсеместной цифровизации общества. Это сам источник этих технологических решений в лице нескольких частных транснациональных корпораций.

Развитие новых цифровых ресурсов требует огромного количества инвестиций, которые окупаются только при доминировании этих продуктов на рынке. По этой причине для корпораций первичной целью является доминирование на рынке, а не удовлетворение потребностей клиентов.

Эта вроде бы парадоксальная ситуация является тем самым опасным искажением взаимосвязи мира реального и цифрового, где информационный поток настолько мощный, что отдельный человек не способен его полностью осознать и сделать разумный выбор.

Этим и пользуются владельцы цифрового потока самостоятельно определяя потребности и мотивации своих клиентов.

Примеров такой эксплуатации множество — когда человек покупает товар не по его реальным характеристикам, а на основании рекомендации системы формирующей его картину мира. Когда люди вместо решения своих реальных проблем , замещают их виртуальным опытом. Когда вместо поддержания себя в хорошей физической форме и заботе о своем здоровье, готовы инвестировать время и ресурсы в виртуальных аватаров. При этом по факту даже понимая нецелесообразность такого поведения - часто не имеют доступных альтернатив деятельности, так как их социальные контакты находятся в той же цифровой среде и подвержены тем же рекомендательным активностям.

Здесь проявляется следующая проблема цифровизации. При теоретически неограниченном потенциале и возможностей связи между людьми в рамках цифрового мира — в реальности каналов связи между людьми становится крайне мало и все они принадлежат конкретной корпорации и используются для целей этой корпорации.

Несмотря на то что условных мессенджеров множество, основная масса населения пользуется одним или двумя. Это относится как маркетплейсам, так и к поисковым системам. Даже условные СМИ, при казалось бы огромном разнообразии — на самом деле собраны в структурированные сети продвижения конкретного контента и интересах все тех же корпораций.

Эти проблемы будут только усугубляться приводя к ситуации монополизации информационного пространства и обществу еще только предстоит найти ответы на этот вызов.

Одним из направлений возможного решения этой проблемы является создание систем ноу-код программирования для создание альтернативных сервисов и продуктов созданных в интересах самих потребителей. Однако это очень масштабная задача и самые успешные проекты пока что так же принадлежат конкретным корпорациям.

Потому определенные перспективы есть у опенсорс разработки, хотя очевидно что это направление не поспевает за текущими трендами и продуктами корпораций.

Однако как временное решение конкретных запросов от общества подход открытой разработки выглядит наиболее реалистичным и эффективным. Целью такой разработки является создание продуктов и сервисов, достаточно простых для внедрения даже не профессиональными разработчиками — но с достаточным функционалом для закрытия конкретных потребностей пользователей.

Примером подобной разработки будет этот учебный проект.

**1.1 Актуальность проекта**

Несмотря на ряд выше озвученных проблем, отказаться от цифровизации всех аспектов жизни практически невозможно. По этой причине актуальным решением является создание веб приложения для коммуникации между людьми, которое можно развернуть локально в масштабах отдельного населенного пункта или даже района города.

Само понимание потребности в подобном продукте возникло из повседневной деятельности и общения с простыми людьми, вне зависимости являются ли они клиентами бизнеса или нет.

Существующие приложения вроде «телеграмма» закрывают ряд озвученных задач, но не позволяют пользователю как либо расширять функционал для решения конкретных задач сообщества.

В тех же группах телеграмма возникают проблемы с идентификацией собеседника при росте числа активных участников сообщества. Это вызывает проблемы с доверием к информации в таких сообществах и открывает дорогу для мошенников.

Маркетплейсы из системы выбора актуального поставщика превратились в механизмы продвижения маркетинговых кампаний. Честный поставщик без внушительного бюджета на продвижение не может найти клиента, а клиент не может совершить сделку на действительно выгодных и актуальных условиях. Ситуация настолько усугубляется что корпорации открыто в своих лицензионных соглашениях указывают что среди первых 2000 рекомендациях по запросу до 30% контента носят характер прямого продвижения за рекламный бюджет. Притом нет никакого четкого понимания работы алгоритма о последующей выдаче и о порядке отображения. Часто получается так что 30% это 600 карточек товара идет подряд и занимают все первые 10 страниц выдачи. Расчет на то что пользователь просто не захочет тратить свое время и будет вынужден взять то что ему навязывают.

Схожие проблемы актуальны и для досок объявлений вроде «Авито» и для сервисоа по поиску работы вроде «hh.ru». Являясь по сути монополистом на рынке и оптимизируя свои рекомендательные системы, корпорация просто не оставляет инструментов для своих пользователей решить их реальные проблемы. Люди просто отказываются от изучения принципов работы алгоритмов системы и отдают все поле предложений «профессиональным» участникам рынка. В каком то смысле примитивные доски объявлений начало нулевых решали задачи людей лучше, чем современная мегакорпорация со всеми доступными им вычислительными ресурсами.

Подобный пользовательский опыт испытывает большинство людей, потому и запрос на создание альтернативного инструмента взаимодействия между людьми понятен и обоснован.

Люди хотят получить инструмент для самоорганизации, для продвижения своих услуг или товаров, для совместной деятельности и организации мероприятий. При этом людей интересует конкретика и актуальность информации, возможность быстро оценить имеющиеся предложения и не провалиться в воронку навязывания услуг или товаров.

Интересы и запросы людей отличаются, но есть условный запрос на минимальный функционал - отвечающий решению конкретных целей пользователей по получению доступа к информации. Этому запросу и будет отвечать разрабатываемое приложение.

**1.2 Цели и задачи проекта**

Целью этого проекта является создание веб приложения для организации совместных мероприятий. Сам процесс проектирования и создания подобного продукта будет подробно описан, чтобы быть примером для других проектов.

Притом важным условием реализации является использование доступных простым пользователям инструментов для разработки, развертывание и модернизации этого приложения.

Задачи проекта:

* Выбор и изучение инструментов для самостоятельной разработки приложения
* Выбор инфраструктуры для развертывания приложения
* Компиляция исходного кода
* Настройка необходимой конфигурации
* Описание всего процесса разработки и запуска приложения для целей обучения целевых пользователей

**1.4 Инструменты разработки**

Одной из ключевых целей проекта является возможность его масштабирования и массового использования. Эта цель ограничивает нас в выборе инструментов разработки — потому в приоритете будут бесплатные опенсорс продукты, доступ к которым может получить любой пользователь из открытых источников.

Это проблема крайне актуальна сегодня , когда из-за санкций против нашей страны ограничивается доступ ко многим сервисам и продуктам лидеров рынка, которые до этого повсеместно использовались и на их работоспособности завязаны бизнес процессы.

|  |  |
| --- | --- |
| Инструмент | Описание |
| Visual Studio Code | Редактор кода с функциями IDE |
| Windows10, Ubuntu Linux | Операционная система |
| PostgreSQL | Система управления базами данных |
| Flask | Фреймворк для создания веб-приложений |
| Docker | Платформа контейнеризации,  используемая для оптимизации управления приложениями |
| Python | Язык программирования |
| Git, GitHub | Управление версиями, хранилище программного кода |
|  |  |

**2. Теоретическая часть.**

Ввиду ограниченности времени на реализацию проекта в рамках учебной программы принято решение о поэтапной разработке и фокусе внимания на получение MVP продукта. Дополнительный функционал будет добавлять после получения базового рабочего продукта.

Такой выбор сделан с целью отработки комплексного понимания самой сути разработки, взаимосвязи разных областей знаний и технологий в процессе создания реального продукта.

По возможности к проекту могут присоединиться другие студенты курса, чтобы в процессе командной работы лучше изучить методологию Agile.

**2.1 Структура приложения.**

Структурно приложение будет состоять из нескольких модулей.

Так как основной изучаемый язык это Python, а целью является разработка веб приложения — то основным фреймворком будет Flask.

Потому основой всего приложения будет точка входа и сборки всех компонентов в виде исполняемого файла.

В минимальной версии приложения фронтенд часть будет имеет второстепенное значение и будет выполнять исключительно утилитарные функции. Состоять будет из двух модулей — в одном будут храниться шаблоны страниц в формате html, во втором будет храниться вся статика в виде стилей, шрифтов и изображений. Для ускорения разработки будем использовать стандартные бесплатный CSS-фреймворк для адаптивной верстки сайтов и веб-приложений — bootstrap.

Бекенд будет состоять из исполняемого файла, его окружения и конфигов, откуда мы будем подтягивать всю логику работы приложения. Для flask создалим библиотеку маршрутов.

Для работы с базой данных будет отдельный модуль где будут храниться все модели таблиц.

Так как планируется возможность развертывания приложения на облачных серверах, то помимо самого приложения будут конфигурационные файлы виртуального окружения, докера и nginx.

**2.2 Выбор среды разработки.**

Одной из основных задач проекта является удобство и доступность его разработки и развертывания.

Потому при выборе инструментов и среды разработки мы исходим из фактической доступности инструментов и навыков простых пользователей. Для большинства людей более привычной операционной системой является Windows. Однако среди разработчиков промышленным стандартом является Linux.

Потому принято решение в качестве основной операционной системы использовать Windows, но саму разработку проводить в подсистеме Windows Subsystem for Linux. В качестве дистрибутива Линукс мы выберем Ubuntu.

В качестве редактора кода и IDE выбираем Visual Studio Code, так как это бесплатный опенсорс продукт с огромным количеством расширений и достаточным функционалом. Так как мы приняли решения работать на виндовс с эмуляцией линукс — то нам важна полноценная поддержка WSL. VSC умеет это делать из коробки , в отличие от PyCharm в котором этот функционал доступен только в платной версии.

**2.3 Обновление базы данных**

Реляционные базы данных сосредоточены вокруг структурированных данных, однако как в процессе разработки, так и в процессе обновления приложения у нас можем возникнуть потребность в изменении структуры данных. Чтобы постоянно не удалять и не создавать новые базы данных воспользуемся расширением Flask-Migrate.

**2.4 Базовый функционал**

Основой приложения является возможность пользователей создавать информационные карточки планируемых мероприятий, а другим пользователям записываться на эти мероприятия и по итогам оставлять свои отзывы о качестве предоставленной услуги.

Админ приложения будет иметь возможность проводить анализ отзывов и на его основе определять рейтинг организаторов.

Для достоверности данных будет реализован механизм когда только участник мероприятия может оставить отзыв.

**2.5 Расширенный функционал.**

После реализации основного функционала планируется внедрение новых возможностей. Для админа мы внедрим языковую модель , позволяющую автоматически определять тональность и эмоции отзывов.

Для пользователей мы можно внедрить систему меток на карте, чтобы было удобно выбирать мероприятия подходящие по месту проведения. Так же можно внедрить чат бота для ответа на основные вопросы , но это отдельная разработка требующая существенных затрат времени.

**3. Практическая часть**

**3.1 Подготовка среды разработки.**

Скачаем и установим наш редактор кода с официального сайта

https/code.visualstudio.com/Download

либо

https://github.com/microsoft/vscode

Подключим компоненты WSL и Hyper-V в windows. Для этого в поисковой строке виндовс набираем «Включение или отключение компонентов Windows». Подключаем нужные нам компоненты.

С оф.сайта скачиваем последнюю версию ядра Linux WSL 2.

Запускаем Windows PowerShell от администратора и задаем

wsl --set-default-version 2

Устанавливаем выбранные дистрибутив линукс.

wsl --install -d ubuntu

После установки дистрибутива создаем суперпользователя и задаем пароль.

После установки создаем конфигурационный файл для управления ресурсами системы выделяемыми для WSL. Стандартный путь для конфига C:\Users\YourUsername где и создаем файл с расширением .wslconfig

В зависимости от ресурсов компьютера выделяем объем памяти и ядер процессора под эту службу. Рекомендуется выделить не менее 6 Гб.

[wsl2]

memory=6GB

processors=2

Интегрируем WSL в наш редактор кода. Для этого в VSC перейдем во вкладку extensions(расширения) и наберем там WSL. После установки расширения можно подтвердить запуск «get started with wsl». Либо в меню дополнительных команд VSC (ctrl + shift + p) выберем «Connect to WSL using distro..” и выберем установленный дистрибутив линукс, в нашем случае ubuntu. После соединения мы получаем доступ к терминалу линукса.

Следующим этапом подготовки будет устновка docker. Так как мы работаем из windows, то нам понадобится docker desktop. Скачиваем и устанавливаем приложение, не забыв поставить галочку в приоритете WSL2.0 перед Hyper-V. После запуска докера заходим в настройки, выбираем вкладку Resources > WSL integration. Включаем Enable integration и активируем ползунок с нашим дистрибутивом. После применения настроек докер так же доступен из нашего терминала.

Проверим это командой из терминала.

docker version

Фактически все подготовительные работы выполнены и дальше разработка будет проходить из терминала и IDE.

Для удобства разработки создадим нового пользователя линукс и будем производить разработку от его лица. Для удобста читаемости примера кода мы будем использовать конкретного пользователя с ником devel. Наделим его правами root.

В терминале пишем команду

sudo adduser username

Заполним все формы и создадим пароль. После этого добавим нового пользователя в группу sudo командой

sudo usermod -aG sudo username

Далее все действия будет производить от имени созданного пользователя. Выберем активного пользователя командой

su username

Установим актуальную версию python. Для этого добавим репозиторий с Python командами

sudo apt install software-properties-common

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt-get install python3-launchpadlib -y

sudo apt update -y

sudo apt install python3.11

Проверим установку командой python --version

Актуализируем используемую версию python командой

sudo update-alternative --install /usr/bin/python python /usr/bin/python3.11 1

После чего установим другие пакеты нужные для разработки и создания виртуального окружения.

sudo apt install python3.11-dev python3.11-venv python3.11-distutils python3.11-gdbm python3.11-tk python3.11-lib2to3

Так же установим пакетный менеджер Python

sudo apt install python3-pip -y

Создадим в домашней директории пользователя папку для нашего проекта и выберем ее в качестве основной для работы в VSC. Всю остальные действия мы будем проводить из директории проекта.

Установим виртуальное окружение для нашего проекта командой

python -m venv venv

Активируем виртуальное окружение для текущего проекта

source venv/bin/activate

Установим выбранную для нашего проекта СУБД. Выбор в пользу Postgres сделан из-за ее распространенности и достаточном для наших задач функционале.

sudo apt install postgresql postgresql-contrib -y

Создадим пароль для суперпользователя postgres

sudo passwd postgres

Нужно понимать что текущая версия postgres работает из под виртуального сервера линукс и может конфликтовать с СУБД работающей на основной операционной системе, потому заменим используем порт на незанятый. Для этого перейдем в основной каталог postgres установленной на нашем виртуальном сервере. Для текущей версии это директория

cd /etc/postgresql/14/main

Откроем конфигурационный файл текстовым редактором и заменим значение порта на уникальное(в текущем проекте порт 5532). Сохраняем изменения и возвращаемся в папку проекта.

Для удобной работы с базой данных мы подключим ее напрямую к VSC. Для этого создадим пользователя базы данных и наделим его всеми правами и доступом. Так как у нас есть уникальный супер пользователь postgres на линукс для которого выше мы создали пароль, то перейдем на него и от его имени создадим нашу базу данных.

su postgres

Запустим postgres командой

psql

Создадим базу данных, пользователя project и наделим его всеми нужными правами на использование этой базы данных.

CREATE ROLE devel PASSWORD 'password' SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;

CREATE DATABASE projectdb;

GRANT ALL PRIVILEGES ON DATABASE "projectdb" to project;

ALTER USER project CREATEDB;

Перейдем в созданную базу данных и предоставим пользователю все привелегии для схему public

GRANT ALL ON schema public TO project;

exit

Выйдем из postgres, переключимся на владельца проекта и вернемся в основную директорию проекта. Проверим активно ли ли виртуальное окружение.

Теперь подключим PostgreSQL к VSC. Установить расширение «PostgreSQL». Откроем командную палитру «Ctrl + Shift + P» и выберем «PostgreSQL: New Query». Создайте новый профиль указав атрибуты соединения, такие как имя хоста, имя базы данных, имя пользователя, порт подключения и прочее.

Это расширению удобно для работы с классическим языком запросов SQL. Для удобства непосредственно разработки приложения так же установим PostgreSQL database manager – с удобным графическим инструментарием и возможностью просмотра содержания таблиц.

На этапе разработки приложения в качестве хоста мы будем использовать localhost , при запуске приложения укажем хост используемого сервера. Все остальные атрибуты задаем для ранее созданного пользователя coder.

Так как разработку мы будем производить поэтапно установим систему контроля версий git.

sudo apt install git

После чего в каталоге проекта инициализируем git

git init

Создадим первый комит

git commit -m "start project"

Переименуем основную ветку в main

git branch -m main

И подключим удаленный репозиторий

git remote add origin «адрес репозитория»

Передадим все изменения во внешний репозиторий

git push -u origin main

У нас есть контроль версий разрабатываемого приложения. Если над разработкой будет работать несколько человек , то каждый из них может добавлять свои изменения пулл реквестами.

Среда разработки подготовлена и можно переходить непосредственно к разработке приложения.

**3.2 Установка необходимых библиотек**

Установим flask и SQLAlchemy командой

pip install flask

pip install Flask-SQLAlchemy

Установим библиотеку python dotenv, которая позволяет загружать переменные среды из файла .env в проект

pip install python-dotenv

Установим модуль для взаимодействия с базой данных PostgreSQL

python -m pip install psycopg2-binary

pip install psycopg2

Установим библиотека PostgreSQL, которая позволяет клиентам отправлять запросы и получать ответы от бэкенд-сервера

sudo apt-get install libpq-dev

Установим расширение Flask для управления состоянием входа пользователя в систему.

pip install flask-login

Установим расширение для Flask, которое позволяет работать с ресурсами проекта

pip install -U Flask-Assets

Установим библиотеку Pillow для работы с изображениями

pip install --upgrade Pillow

Установим расширение Flask Migrate для обработки миграций базы данных для SQLAlchemy

pip install Flask-Migrate

Установим расширение для Flask, которое добавляет функционал для хеширования и проверки паролей.

pip install flask-bcrypt

Установим расширение Flask-WTF использует его для защиты веб-форм

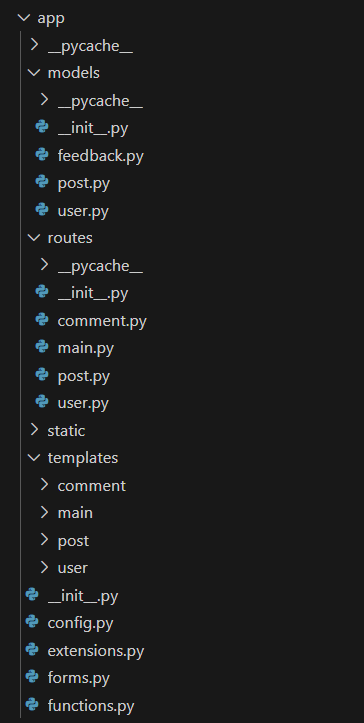
pip install Flask-WTF

Для удобства отладки приложения установим библиотеку

python -m pip install --upgrade debugpy

**3.2 Разработка приложения**

Общая структура нашего приложения будет иметь следующий вид

****

Для того чтобы наше приложение можно было легко масштабировать и не потерять компоненты и зависимости, всю разработку мы будем производить в виртуальном окружении. Активируем его

source venv/bin/activate

Перейдем в директорию нашего проекта и создадим python package в котором будут храниться файлы и библиотеки нашего приложения.

mkdir mainapp

Перейдем в созданный каталог и создадим файл \_\_init\_\_.py

touch \_\_init\_\_.py

Это будет наш основной файл и точка сборки всех компонентов приложения.

Создадим функцию сборки всех компонентов приложения и назовем ее create\_app, эта функция будет возвращать нам уже собранное приложение.

def create\_app():

return app

Основной фреймворк разработки это фласк, потому внутри созданной функции определим метод для нашего приложения.

app = Flask(\_\_name\_\_)

Атрибутом будет параметр \_\_name\_\_ , то есть текущий файл.

Так как по задумке мы в

Определим логику сборки приложения следующей конструкцией

with app.app\_context():

db.create\_all()

Теперь мы можем инициализировать все объекты импортируемые из файлов окружения внутри функции сборки приложения. Для каждого импортируемого объекта проведем инициализацию.

Для SQLAlchemy укажем параметр текущего приложения

db.init\_app(app)

Для flask\_migrate укажем как текущее приложение ,так и базу данных

migrate.init\_app(app, db)

login\_manager.init\_app(app)

Теперь в момент создания приложения будет создаваться и база данных. Потому создадим конфигурацию приложения для подключения и получения доступа к базе данных. Для это в нашу функцию сборки мы будем передавать новый атрибум с данными из файла **config.py**

app.config.from\_object(config)

Из этого же конфига мы сможем получать любые другие переменные, которые нам могут понадобиться в процессе расширения функционала приложения. Сам файл мы создадим чуть позже.

Далее нам нужно будет зарегистрировать используемые blueprint марштуры routes для каждого маршрута. Пример для user

app.register\_blueprint(user)

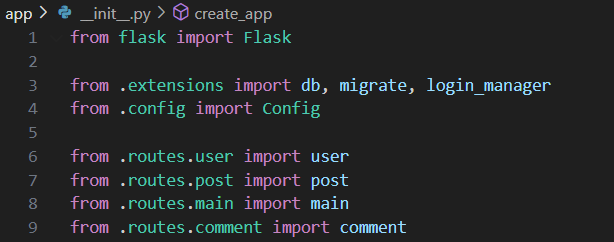
Так же зададим маршруты для логин менеджера.

login\_manager.login\_view = 'user.login'

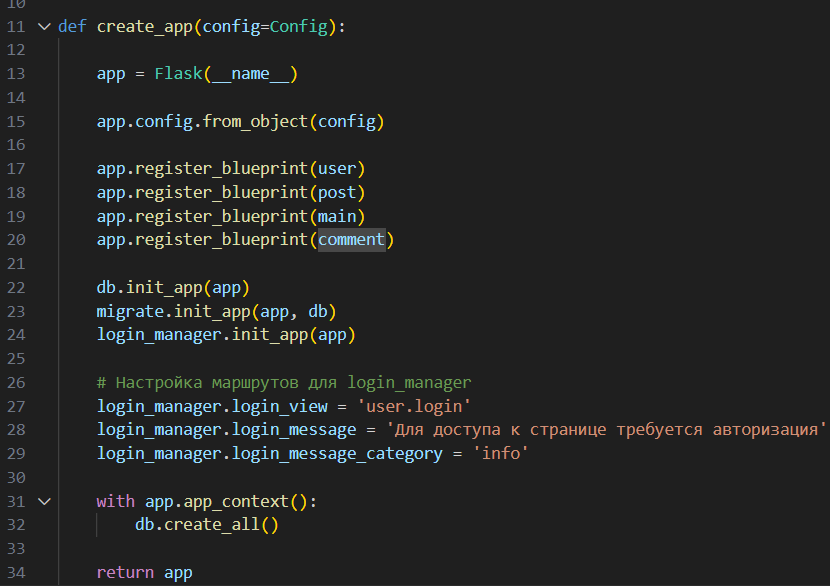
login\_manager.login\_message = 'Для доступа к странице требуется авторизация'

login\_manager.login\_message\_category = 'info'

Итоговый импорт библиотек окружения имеет вид



Функция сборки приложениями



**3.3 Создание конфигурационных файлов Flask**

В основной директории проекта создадим конфигурационный файл для настроек Flask

touch .flaskenv

Зададим переменные окружения работы flask. В зависимости от нейминга функций определяем конфигурацию работы flask. Для нашей структуры проекта настройки будут следующие, где значение app

это название создаваемого приложение и по сути объект python package. Flask автоматически распознает стандартные название функций и методов. Если мы хотим назвать приложение уникальны именем, как и функцию собирающую приложение, то указывать ссылку для них нужн следующим способом. Вместо значения app указываем имя папки с приложением и через разделитель : указываем имя функции сборки приложения. FLASK\_APP=nameapp:namefunc

Для переменной FLASK\_ENV зададим параметр development на стадии разработки и production при развертывании на сервере.

Для переменной FLASK\_DEBUG зададим параметр True, это активирует режим отладки чтобы вносимые в код изменения сразу же отображались в веб приложении.

Для того чтобы система подтягивала этот конфиг нужно в нашем виртуальном окружении установить модуль pip install python-dotenv

в нашем виртуальном окружении. В итоге для нашего проекта мы получаем следующие перменные

FLASK\_APP=app

FLASK\_ENV=development

FLASK\_DEBUG=True

**3.4 Создание файла окружения .env**

Этот файл будет нужен для хранения переменных окружения всего проекта, а не только разрабатываемого приложения. При развертывание приложения на сервере здесь будут указываться уникальные значения переменных для конкретной реализации.

На стадии разработки здесь мы будем хранить переменные нужные для работы приложения в наших условиях работы из под WSL.

Зададим значения переменных для нашего подключения к Postrges

POSTGRES\_USER=project

POSTGRES\_PASSWORD=project

POSTGRES\_HOST=127.0.0.1

POSTGRES\_PORT=5532

POSTGRES\_DB=projectdb

Здесь мы меняем параметры используемого во время разработки порт на котором будет работать postgres. При запуске приложения на рабочем сервере , достаточно будет задать переменные конкретной реализации только в этом файле окружения и они подтянутся во все модули приложения.

**3.5 Создание config.py**

Создадим конфигурационный файл из которого будем брать данные для инициализации базы данных и других объектов при сборке приложения.

Импортируем библиотеку для работы с операционной системой. Это нам будет нужно чтобы получить доступ к переменным окружения и определения абсолютных путей расположения файлов.

import os

Создадим класс Config в котором и будем хранить переменные.

Значения ряда используемых переменных мы будем брать из файла .env. Если же такого файла нет, то на этапе разработки можем задать значения нужных переменных по умолчанию.

class Config(object):

APPNAME = 'app'

ROOT = os.path.abspath(APPNAME)

UPLOAD\_PATH = '/static/upload/'

SERVER\_PATH = ROOT + UPLOAD\_PATH

USER = os.environ.get('POSTGRES\_USER', 'devel')

PASSWORD = os.environ.get('POSTGRES\_PASSWORD', '223344')

HOST = os.environ.get('POSTGRES\_HOST', '127.0.0.1')

PORT = os.environ.get('POSTGRES\_PORT', 5532)

DB = os.environ.get('POSTGRES\_DB', 'projectdb')

SQLALCHEMY\_DATABASE\_URI=f'postgresql://{USER}:{PASSWORD}@{HOST}:{PORT}/{DB}'

# SECRET\_KEY = '123456'

SQLALCHEMY\_TRACK\_MODIFICATIONS = True

Как и в файле окружения всего проекта при развертывании приложения эти данные имеют первостепенное значение и их нужно тщательно охранять. В процессе разработки будем использовать простые значения для удобства.

**3.6 Модуль расширений extensions**

Создадим в директории app файл extensions.py в котором будем проводить инициализацию всех нужным нам переменных и процессов.

Смысл этого файла будет в том ,что здесь мы будем импортировать все нужные библиотеки и функции, создадим здесь экземпляры этих объектов и после чего будем использовать эти объекты во всех других частях приложения.

Для работы с базой данных создадим объект

db = SQLAlchemy()

Для работы с изменениями базы данных создадим объект

migrate = Migrate()

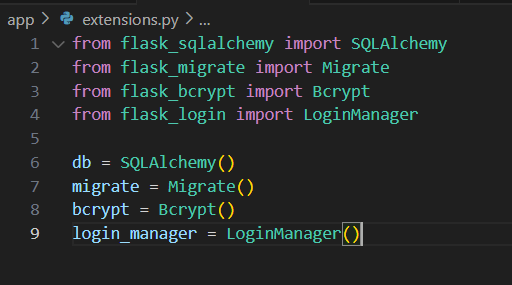
Так же создадим объект криптографической хеш функции формирования ключа , для защиты паролей.

bcrypt = Bcrypt()

Для авторизации пользователей создадим объектов

login\_manager = Login Manager()

Импортируем все нужные библиотеки, в итоге получим файл расширений следующего содержания.



**3.7 Создание библиотеки models**

В этой библиотеки мы будем хранить классы с описанием структуры и поведения объектов нашего приложения. Для реализации базового функционала нам понадобятся классы для описания структуры хранения информации в базе данных.

У нас будет таблица user, в которой мы будем хранить все сведения о нашем пользователе. Это наш ключевой класс, так как на него завязана как регистрация с аутентификацией пользователя, так и параметры доступа к разным функция приложения.

Так же в этом модуле будет реализован функционал управления сеансами пользователя.

В директории нашего приложения создадим каталог models

mkdir models

Преобразуем каталог в Python Packag создав файл \_\_init\_\_.py

Создадим первую модель user.py

Импортируем объекты из файла расширений приложения

from ..extensions import db, login\_manager

Так как модель юзер будет связана с моделью post.py , то импортируем нужный для работы класс.

from .post import Post

Импортируем класс UserMixin для реализации функционала аутентификаци, проверки активность пользователя или правильности заполнения учетных данных.

from flask\_login import UserMixin

Для учета времени создания пользователя импортируем билиотеку

from datetime import datetime

Для работы login\_manager создадим функцию подгрузки данных о текущем пользователе. Для корректной работы обернем эту функцию в декоратор

@login\_manager.user\_loader

def load\_user(user\_id):

return User.query.get(int(user\_id))

Создадим наш основной класс User.

Чтобы нас класс адекватно взаимодействовал с библиотекой login\_manager , наследуем его от класса UserMixin. Это позволит фласку получать доступ у нужным атрибутам этого класса создавая их самостоятельно.

class User(db.Model, UserMixin):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(50))

login = db.Column(db.String(50))

password = db.Column(db.String(200))

avatar = db.Column(db.String(200))

status = db.Column(db.String(50), default='user')

posts = db.relationship(Post, backref='author')

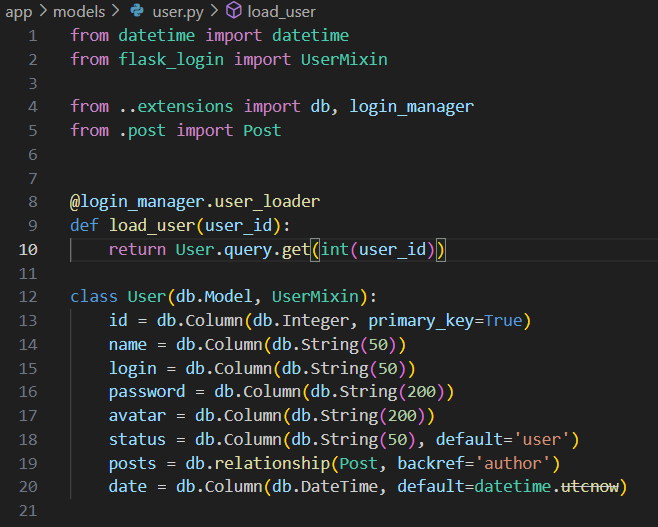
date = db.Column(db.DateTime, default=datetime.utcnow)

Используем простую схему для формирования таблицы. Единственный момент связи таблиц будет между user и posts будет через колонку автора поста в таблице post и колонку posts в таблице user.

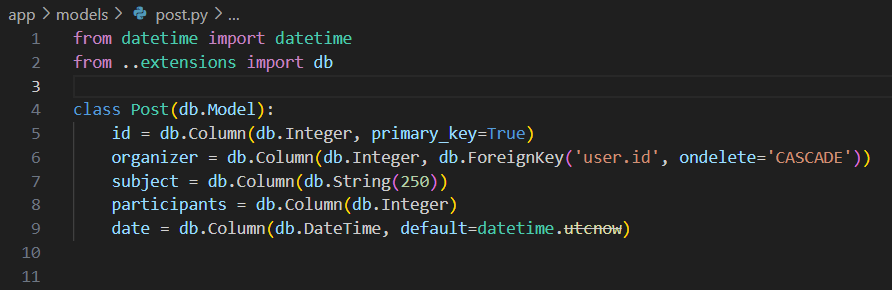
При расширении приложения можно будет реализовать дополнительную логику связей между таблицами в зависимости от бизнес логики проекта. Благодаря фласк-миграции многие изменения базы данных можно будет сделать на ходу. Однако прорабатывать логику зависимостей между элементами таблицы лучше заранее, так как некоторые измемения могут радикально менять уже созданные связи, что приведет к неработоспособности БД. Для отката изменений можно использовать функционал фласк-миграции. Но для этого обновлять структуру хранения данных и логику связей поэтапно, после чего эти изменения комитить и апгрейдить.

В нашем случае мы разрабатываем максимально простую и безопасную структуру хранения данных, потому проблем быть не должно.

Общий вид этого модуля будет следующим.

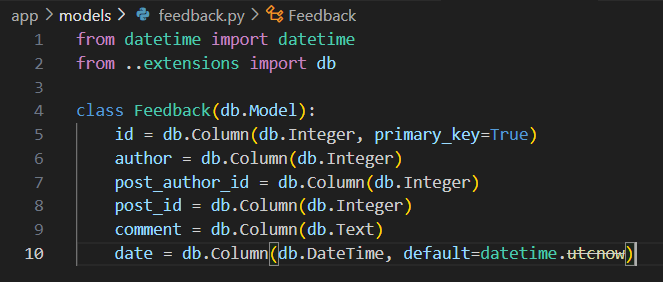


Точно таким же образом создадим модуль для класса Post.



Большая часть полей заполняется стандартно, единственная связанная колонка через ForeignKey это organizer с атрибутом ondelete='CASCADE'. Это сделано для реализации функционала удаления пользователя. Если админ приложения примет решения удалить организатора некачественных мероприятий или мошенника, то будут удалены все связанные с ним записи в базе данных.

Создадим модуль для класса Feedback.



**3.8 Создание модуля forms**

На этапе проектирования приложения мы должны понимать какие инструменты для какого функционал мы будем использовать. Выбор фреймворка Flask был обусловлен предлагаемым функционалом, который полностью закрывает потребности в создания и обработки форм в веб-приложении.

Импортируем библиотеку FlaskForm, валидатор изображений FileAllowed, валидаторы для заполнения всех используемых форм.

Это позволит при минимуме кода закрыть целый спектр задач по валидации ввозимых пользователем данных, их формату и прочему. Так же это существенно упрощает тестирование приложения, когда есть четкое понимание типов используемых и допустимых данных.

Все импорты будут иметь следующий вид

from flask\_wtf import Flask Form

from flask\_wtf.file import FileAllowed

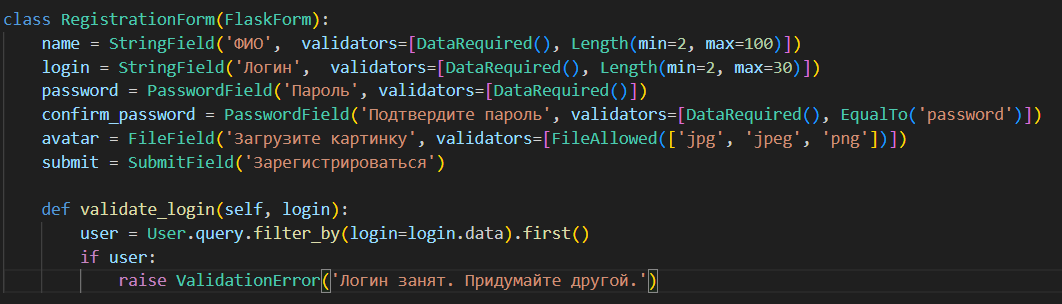
from wtforms import StringField, PasswordField, SubmitField, FileField, BooleanField, SelectField, TextAreaField

from wtforms.validators import DataRequired, Length, EqualTo, ValidationError

from .models.user import User

Все создаваемые классы будут наследоваться от FlaskForm

Создадим класс для регистрации.



Принцип создания элементов класса следующий.

Мы определяем поле веб формы, именуем ее, определяем для нее валидатор проверяющий вводимые пользователем данные на допустимый формат хранения данных.

В случае нашей реализации мы задаем следующие поля формы регистрации.

Имя пользователя — для удобства социального взаимодействия.

Логин — именно это имя будет идентификатором пользователя при авторизации. По этой причине внутри класса определяем метод validate\_login для проверки уникальности задаваемого логина.

Пароль и подтверждение пароля, притом в рамках единой формы, где данные на корректность ввода берутся из соответствующих полей.

В нашем случае для удобства идентификации пользователей между собой мы реализуем возможность загрузки аватара или фотографии. Можно реализовать механизм автоприсвоения картинки через атрибут default: object, либо в интересах экономии хранимой информации ограничить разрешение загружаемых картинок.

По такой же схеме создадим классы для других используемых форм ввода данных в приложении.

Для логирования пользователя

class LoginForm(Flask Form):

login = StringField('Логин', validators=[Data Required(), Length(min=2, max=30)])

password = PasswordField('Пароль', validators=[DataRequired()])

remember = BooleanField('Запомнить меня')

submit = SubmitField('Войти')

Форма для выбора участников меропрития

class Participants Form(Flask Form):

participants=SelectField('participants',choices=[],

render\_kw={'class':'form-control'})

Форма для выбора организаторов мероприятий

class Organizer Form(FlaskForm):

organizer = SelectField('organizer', choices=[],

render\_kw={'class':'form-control'})

**3.9 Инициализация Migrate**

После создания моделей нашего приложения, мы можем инициализировать создание базы данных и начать отрабатывать логику взаимодействия элементов. Но перед этим создадим копию версии нашей базы данных. Если в процессе разработки мы столкнемся с какими либо проблемами. То у нас будет возможность откатить вносимые в с руктуру базы данных изменения.

Инициализируем расширение для работы с миграциями баз данных

flask db init

Инициализируем генерацию автоматической миграции. Эта команда создает сценарии изменений, не внося изменений в бд.

flask db migrate

Для внесения изменений в базу данных используем команду

flask db upgrade

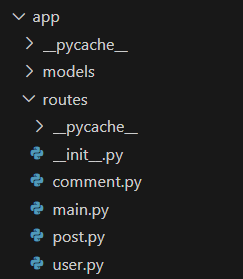
**3.10 Создание модулей main и user библиотеки routes**

На этом этапе разработки все ключевые элементы такие, как модели объектов с которыми мы будем работать и база данных где эти модели будут храниться , уже созданы — потому мы можем перейти к ключевой библиотеке нашего приложения. По сути именно маршруты и функции представления, которые будут выполнять какие-то действия при переходе по определенному адресу являются основой всего функционала приложения с которым будет взаимодействовать пользователь.

Создадим отдельную python packages где будем хранить отдельные модели для реализации конкретного функционала.

Чтобы у нас была точка входа в приложения создадим маршрут для главной страницы. Для организации кода приложения в отдельные модули, которые будут иметь свои собственные шаблоны, маршруты и статику мы используем blueprint flask. Это улучшит структуру приложения позволяя масштабировать или отлаживать части приложения отдельно.

Общая структура библиотеки будет иметь следующий вид



Модуль main будет стартовым с самым минимальным функционалом.

По его структуре будут созданы все остальные ,но с расширенным функционалом под конкретные задачи.

Импортируем библиотеку Blueprint и функцию для отрисовки шаблонов render\_template.

from flask import Blueprint, render\_template

Создадим экземпляр класса Blueprint. В качестве первого аргумента будет задано название route/маршрута. Вторым аргументом будет имя текущего файла.

main = Blueprint('main', \_\_name\_\_)

Создадим функцию которая будет используя импортированный render\_template перенаправлять пользователя на главную страницу приложения. Обернем функцию декоратором route.

@main.route('/')

@main.route('/index')

def index():

return render\_template('main/index.html')

После создания html шаблонов указанных страниц мы фактически уже сможем начать взаимодействовать с приложением через локальный сервер.

Перейдем к созданию главного модуля этой библиотеки user. Именно в этом модуле будем реализовывать весь ключевой функционал приложения по взаимодействию с пользователем.

Импортируем все необходимые библиотеки.

from flask import Blueprint, render\_template, redirect, request, url\_for, flash

from ..forms import RegistrationForm, LoginForm

from flask\_login import login\_user, logout\_user

from ..extensions import db, bcrypt

from ..models.user import User

from ..functions import save\_picture

Создадим объект класса Blueprint

user = Blueprint('user', \_\_name\_\_)

Реализуем функцию регистрации пользователя. Обернем ее декоратором

@user.route('/user/registration', methods=['POST', 'GET'])

Так как функция регистрации является ключевой и стартовой для взаимодействия с приложением , то мы будем проверять корректность вводимой пользователем информации в удобной для самого пользователя форме.

Для этого импортируем класс RegistrationForm из ранее написанного модуля forms. И методом if будем проверять соответствие вводимых данных заданным в форме.

Чувствительные данные вроде пароля будем хранить в виде хеша заданной длины, используя метод generate\_password\_hash() объекта класса bcrypt.

На этой стадии будет первое взаимодействие пользователя с базой данных. Так как приложение будет работать на облачном сервере, то важно точно понимать была ли закончена сессия отправки и регистрации данных. Для этого мы будем использовать конструкцию try-except. И только в случае удачной передачи данных отправлять пользователя на следующий маршрут логирования.

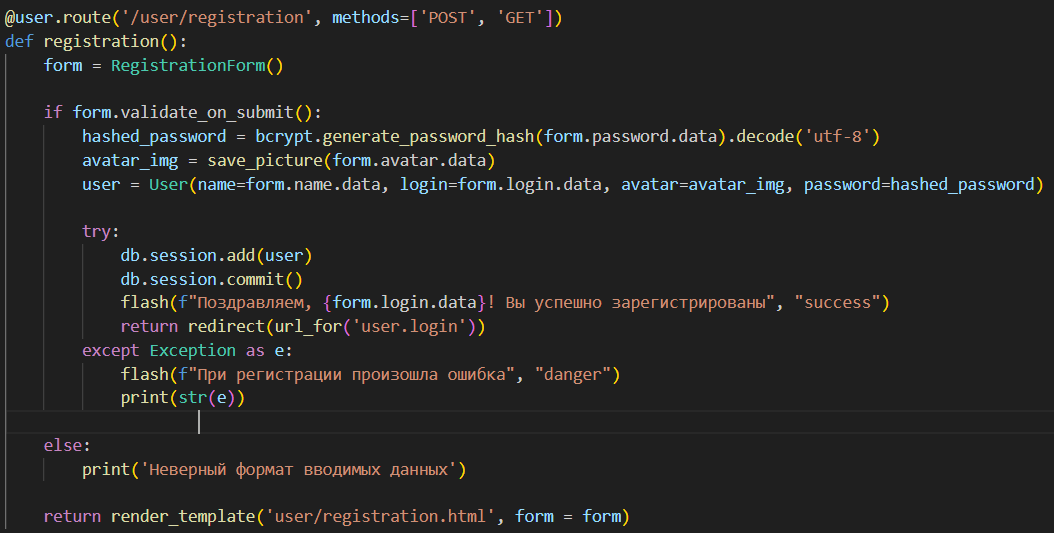
Так же реализуем функционал flash на стороне фронтенда ,чтобы информировать пользователя о статусе регистрации.

Так же для адекватного отображения функционала реализуемого в маршруте на сторне фронта приложения нужно указать пусть к шаблон страницы и контекст передаваемых данных.

Для нашей функции регистрации это будет объект form определенный внутри тела функции.

return render\_template('user/registration.html', form = form)

Законченная функция регистрации будет выглядеть следующим образом



По аналогии с функцией регистрации напишем другие функции не забывая о структуре и последовательности взаимодействий клиента с приложением.

Для логирования напишем следуюший код

@user.route('/user/login', methods=['POST', 'GET'])

def login():

form = LoginForm()

if form.validate\_on\_submit():

user = User.query.filter\_by(login=form.login.data).first()

if user and bcrypt.check\_password\_hash(user.password, form.password.data):

login\_user(user, remember=form.remember.data)

next\_page = request.args.get('next')

flash(f"Пользователь {form.login.data} успешно авторизован", "success")

return redirect(next\_page) if next\_page else redirect(url\_for('post.all'))

else:

flash(f"Ошибка входа, проверьте логин и пароль!", "danger")

return render\_template('user/login.html', form=form)

По введенной переменной user из соотетствующего поля формы мы берем данные для поиска совпадения в таблице user нашей базы данных.

Если такой пользователь существует и он вводит корректный пароль совпадающий с хранимым в БД, то мы авторизуем пользователя и перенаправляем его запрашиваю им страницу. Либо если это первый запрос, то на основную страницу с доступным контентом.

Последней в этом модуле будет функция выхода из системы. Этот функционал уже реализован в flask\_login. Однако необходимо соблюдать структуру взаимодействия и маршрутизации ,потому напишем отдельную функцию.

@user.route('/user/logout', methods=['POST', 'GET'])

def logout():

logout\_user()

return redirect(url\_for('post.all'))

**3.11 Создание модуля post библиотеки routes**

Создадим модуль post для реализации функционала создания мероприятий и регистрации в них участников.

Для всех функций у которых есть взаимодействие с базой данных мы будем использовать декоратор @login\_required библиотеки flask\_login для защиты функции просмотра от анонимных пользователей.

Импортируем все необходимые бибилотеки

from flask import Blueprint, render\_template, request, redirect, url\_for, abort

from flask\_login import login\_required, current\_user

from ..extensions import db

from ..models.post import Post

from ..models.feedback import Feedback

from ..forms import Organizer Form, ParticipantsForm

from ..models.user import User

Создадим объект класса Blueprint

post = Blueprint('post', \_\_name\_\_)

Напишем функцию отображения всех имеющихся мероприятий

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

def all():

form = Organizer Form()

form.organizer.choices = [t.name for t in User.query.filter\_by(status='admin')]

if request.method == "POST":

organizer = request.form.get('organizer')

organizer\_id = User.query.filter\_by(name=organizer).first().id

posts=Post.query.filter\_by(organizer=organizer\_id).order\_by(Post.date.

desc()).all()

else:

posts = Post.query.order\_by(Post.date.desc()).limit(35).all()

return render\_template('post/all.html', posts=posts, user=User, form=form)

Через метод organizer.choices() мы реализуем механизм сортировки вывода данных по организаторам мероприятий.

Через метод render\_template() мы передаем указание на шаблон и все данные которые будут использовать в функционале отображаемой страницы.

Реализуем функция создания мероприятий. На данном этапе логика приложения работает по следующей схеме. Организатор создаем мероприятие и добавляет в него зарегистрированного в базе данных пользователя. Такая упрощенная схема выбрана для ускорения разработки минимальной работающей версии приложения.

Создадим функцию создания мероприятия.

@post.route('/post/create', methods=['POST', 'GET'])

@login\_required

def create():

form = Participants Form()

form.participants.choices=[i.name for i in User.query.filter\_by(status='user')]

if request.method == "POST":

subject = request.form.get('subject')

participants = request.form.get('participants')

participant\_id = User.query.filter\_by(name=participants).first().identified

post = Post(organizer=current\_user.id, subject=subject, participants=participant\_id)

try:

db.session.add(post)

db.session.commit()

return redirect('/')

except Exception as e:

print(str(e))

else:

return render\_template('post/create.html', form=form)

Мы снова взаимодействуем с базой данных, а потому используем конструкцию try except .

Механизм выбора мероприятия и записанного на него участника реализуется через веб форму.

Следующая функция расшивает функционал приложения возможностью редактирования мероприятий. Притом будет реализована защита от несанкционированного использования.

Мероприятия созданные одним пользователем могут быть редактированы или удалены только им же. Для всех остальных пользователей такой функционал будет заблокирован.

Реализуем функцию редактирования меропрития.

@post.route('/post/<int:id>/update', methods=['POST', 'GET'])

@login\_required

def update(id):

post = Post.query.get(id)

form = ParticipantsForm()

form.participants.data = User.query.filter\_by(id=post.participants).first().name

form.participants.choices = [s.name for s in User.query.filter\_by(status='user')]

if post.author.id == current\_user.id:

if request.method == "POST":

post.subject = request.form.get('subject')

act\_user = request.form.get('participants')

post.participants = User.query.filter\_by(name=act\_user).first().id

try:

db.session.commit()

return redirect(url\_for('post.all'))

except Exception as e:

print(str(e))

else:

return render\_template('post/update.html', post = post, form = form)

else:

abort(403)

Условием if post.author.id == current\_user.id: Мы проверяем является ли текущий пользователь автором поста, который он пытается редактировать. Если нет — то вместо перехода в соответствую веб форму будет выведено сообщение об ошибке 403.

Создадим последнюю функцию управления мероприятием доступную организатору.

@post.route('/post/<int:id>/delete', methods=['POST', 'GET'])

@login\_required

def delete(id):

post = Post.query.get(id)

if post.author.id == current\_user.id:

try:

db.session.delete(post)

db.session.commit()

return redirect(url\_for('post.all'))

except Exception as e:

print(str(e))

return str(e)

else:

abort(403)

По аналогии с редактированием, доступ к этому функционалу может получить только авторизованный пользователь, являющийся автором карточки мероприятия.

Последней функцией реализованной в этом модуле будет добавление комментария участником этого мероприятия.

@post.route('/post/<int:id>/feedback', methods=['POST', 'GET'])

@login\_required

def feedback(id):

if request.method == "POST":

post\_id = Post.query.get(id).id

author = current\_user.id

post\_author\_id = Post.query.get(id).organizer

comment = request.form.get('text')

feedback = Feedback(post\_id=post\_id, author=author, post\_author\_id=post\_author\_id, comment=comment)

try:

db.session.add(feedback)

db.session.commit()

return redirect('/')

except Exception as e:

print(str(e))

else:

return render\_template('comment/feedback.html')

Так как у объекта класса Blueprint в этом модуле есть только ограниченные набор параметров, то механизм доступа к комментированию мероприятия только участником этого меропрития реализован альтернативным способом на стороне отображения данных в шаблоне страницы.

**3.12 Создание модуля comment.py библиотеки routes**

Это модуль будет нести в себе отдельный функционал доступный только админу приложения. Этот модуль является расширением базовой версии приложения и по своей сути является реальным функционалом приложения несущем в себе потенциал вычислительных систем.

В модуле будет реализован потенциал языковых моделей по анализу тональности и настроения текста отзывов о мероприятии.

Функционал до конца не доработан, а потому записей в базу данных производиться не будет. Админ приложения сможет провести автоматическое определение тональности отзывов и увидеть статистику по каждому отзыву отдельного организатора и совокупную оценку качества мероприятий этого пользователя.

Загрузим все используемые библиотеки

from flask import Blueprint, render\_template, redirect, url\_for  
from flask\_login import login\_required  
from ..models.feedback import Feedback  
from ..models.user import User

from ..functions import review, overall\_rating

Создадим объект Blueprint

comment = Blueprint('comment', \_\_name\_\_)

Создадим функцию администратора для выбора конкретного организатора мероприятий для оценки отзывов по этому событию.

@comment.route('/comment/administration', methods=['POST', 'GET'])

def administration():

feedbacks = Feedback.query.distinct('post\_author\_id')

list = []

for i in feedbacks:

list.append(i.post\_author\_id)

return render\_template('comment/administration.html', feedbacks=feedbacks, user=User)

Создадим функция обработки отзывов выбранного организатора.

@comment.route('/comment/<int:id>/work', methods=['POST', 'GET'])

def work(id):

ratings = Feedback.query.filter\_by(post\_author\_id=id).all()

id = id

list\_comments = []

list\_review = []

for post in ratings:

list\_comments.append(post.comment)

for post in list\_comments:

list\_review.append(review(post))

feedback\_rating = dict(zip(list\_comments, list\_review))

general\_opinion = overall\_rating(list\_review)

return render\_template('comment/rating.html', feedback\_rating=feedback\_rating, id = id, go = general\_opinion )

Логика работы функции следующая. Мы определяем id автора, сортируем все отзывы в таблице feedback по колонке организатора меропрития.

Создаем список этих постов для передачи в шаблон html страницы для их отображения.

Через цикл прогоняем все отзывы через самописную функцию использующую языковую модель. Результаты записываем в список и так же передаем в шаблон html страницы для отображения.

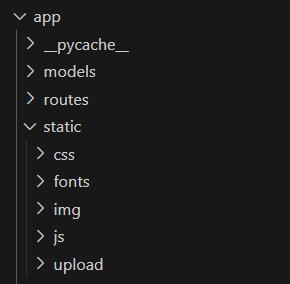
Также все числовые значения анализа текста приводим к бинарной классификации для оценки каждого отзыва и выводим обобщенную оценку о качестве мероприятий конкретного пользователя.

На этом по сути разработки бекенда минимальной версии приложения закончена. Следующий этап это задача фронтенд разработки и написание модулей отображения данных.

**3.13 Фронтенд секция. Создание и наполнение static.**

Для юзерфрендли отображения нашего приложения мы будем использовать уже имеющиеся в открытом доступе шаблоны, стили и шрифты. Все эти компоненты мы будем хранить в директории static.

Общая структура будет иметь следующий вид

****

В папке css мы будем хранить стили для шаблонов наших html страниц. Для удобства стили для отдельных элементы мы будем хранить в папке css/blocks.

Так как ряд данных мы будем отображать в табличной форме, то создадим файл table.css выбраным стилем отображения элемента.

.custom-table {

min-width: 900px; }

.custom-table thead tr, .custom-table thead th {

border-top: none;

border-bottom: none !important;

background: #2e2e36;

color: #b3b3b3;

}

.custom-table tbody th, .custom-table tbody td {

color: #777;

font-weight: 400;

padding-bottom: 20px;

padding-top: 20px;

font-weight: 300; }

.custom-table tbody th small, .custom-table tbody td small {

color: #b3b3b3;

font-weight: 300; }

.custom-table tbody tr:not(.spacer) {

border-radius: 7px;

overflow: hidden;

-webkit-transition: .3s all ease;

-o-transition: .3s all ease;

transition: .3s all ease; }

.custom-table tbody tr:not(.spacer):hover {

-webkit-box-shadow: 0 2px 10px -5px rgba(0, 0, 0, 0.1);

box-shadow: 0 2px 10px -5px rgba(0, 0, 0, 0.1); }

.custom-table tbody tr th, .custom-table tbody tr td {

background: #25252b;

border: none;

-webkit-transition: .3s all ease;

-o-transition: .3s all ease;

transition: .3s all ease; }

.custom-table tbody tr th a, .custom-table tbody tr td a {

color: #b3b3b3; }

.custom-table tbody tr th:first-child, .custom-table tbody tr td:first-child {

border-top-left-radius: 0px;

border-bottom-left-radius: 0px; }

.custom-table tbody tr th:last-child, .custom-table tbody tr td:last-child {

border-top-right-radius: 0px;

border-bottom-right-radius: 0px; }

.custom-table tbody tr.spacer td {

padding: 0 !important;

height: 3px;

border-radius: 0 !important;

background: transparent !important; }

.custom-table tbody tr.active th, .custom-table tbody tr.active td, .custom-table tbody tr:hover th, .custom-table tbody tr:hover td {

color: #fff;

background: #2e2e36; }

.custom-table tbody tr.active th a, .custom-table tbody tr.active td a, .custom-table tbody tr:hover th a, .custom-table tbody tr:hover td a {

color: #fff; }

.custom-table td {

text-align: center;

vertical-align: middle;

}

Библиотеки стилей будем хранить в папке css/libs

Для ускорения разработки будем использовать самые распространенные и доступные стили с сайта https://getbootstrap.com/

и https://fontawesome.com/

Скачаем нужные выбранные стили.

fontawesome.min.css

bootstrap.min.css

Опишем стили основных элементов используемых в шаблонах страниц.

В файле fonts.css в котором мы будем создавать правила для подключения выбранных шрифтов для отображения на веб-странице

Используя языковую конструкцию @font-face подключим все выбранные нами шрифты.

@font-face {

font-family:'Montserrat';

font-style: normal;

font-weight: 400 ;

src: url('../fonts/Montserrat/Montserrat-Regular.ttf');

}

@font-face {

font-family:'Montserrat';

font-style: normal;

font-weight: 600 ;

src: url('../fonts/Montserrat/Montserrat-Bold.ttf');

}

@font-face {

font-family: 'FontAwesome';

src: url('../fonts/FontAwesome/fontawesome-webfont.eot');

}

@font-face {

font-family: 'FontAwesome';

src: url('../fonts/FontAwesome/fontawesome-webfont.ttf');

}

@font-face {

font-family: 'FontAwesome';

src: url('../fonts/FontAwesome/fontawesome-webfont.woff');

}

@font-face {

font-family: 'FontAwesome';

src: url('../fonts/FontAwesome/fontawesome-webfont.woff2');

}

@font-face {

font-family: 'FontAwesome';

src: url('../fonts/FontAwesome/FontAwesome.otf');

}

Теперь в файле main.css мы сможем импортировать эти шрифты.

@import url(fonts.css);

Так же в этом файле определим параметры оформления для используемых нами блоков в разметке страниц.

.page {

background-color: antiquewhite;

overflow: hidden;

display: flex;

flex-direction: column;

min-height: 100vh;

}

.content {

max-width: 1200px;

margin: 0 auto;

padding: 40pt 20px;

flex: 1 1 auto;

}

.header {

position: sticky;

top: 0;

padding: 0px 5px;

box-shadow: 6px 6px 6px rgba(12, 11, 13, 0.5);

}

h1 {

font-family: Montserrat, sans-serif;

font-weight: 600;

}

Папку img будем использовать для локального хранения используемых в оформлении приложения изображений.

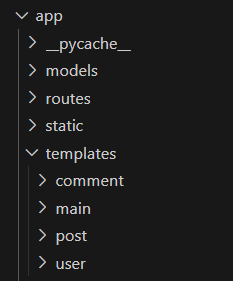
Для хранения загружаемых пользователем изображений и файлов создадим папку upload.

**3.14 Фронтенд секция. Разработка библиотеки templates.**

Создадим папку templates в которой будут хранится шаблоны страниц для каждой отдельной части нашего приложения.

Следуя принципам ООП для каждого маршрута в routes создадим отдельную библиотеку шаблонов страниц.

Структура будет следующая



В библиотеке main мы создадим базовые шаблоны, от которых будут наследоваться все остальные шаблоны.

Создадим файл base.html с базовым каркасом для всех шаблонов.

<head>

<meta charset="UTF-8">

<link rel="stylesheet" href="{{url\_for('static', filename='css/libs/bootstrap.min.css') }}">

<link rel="stylesheet" href="{{url\_for('static', filename='css/main.css') }}">

{% block style %}{% endblock %}

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

</head>

В блоке head мы определим последовательность используемых стилей. Притом первым определяем бутстрап, а после него наши собственные стили. Это позволит где нам нужно использовать стиль отличный от основного.

Определяем блок Заголовок через конструкцию

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

Теперь заполняя соответствующий блок в любых шаблонах который наследуется от базового — эти значения будут подставляться в базовый шаблон.

В блоке <body> так же определим директивы наследования используя {% include %}

<body>

<header class="header">{% include 'main/header.html' %}</header>

<div class="page">

<main class="content">

{% include 'main/flash.html' %}

<h1>{{ self.title() }}</h1>

{% block content %}{% endblock %}

</main>

<footer>{% include 'main/footer.html' %}</footer>

</div>

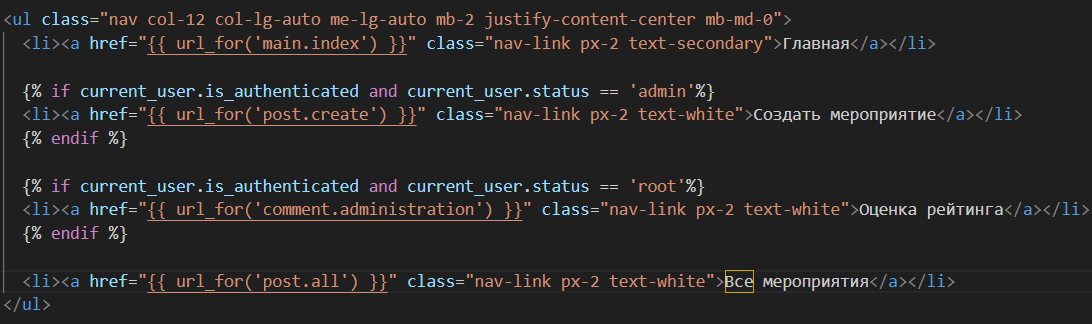
Теперь в базовый шаблон будут подставлять нужные нам стили или элементы страницы, каждый из которых мы сможем настроить в соответствующих шаблонах.

Основным шаблоном в котором будет использоваться логика маршрутов и определяться права доступа будет header.html

Определим блочный элемент с классом container

<div class="container">

Создадим маркированный список управляющих элементов



В этом блоке мы определяем отображаемые элементы для пользователей с разным статусом доступа.

В базовой версии у нас будет три статуса. Для каждого из них хедер страницы будет иметь разные элементы.

Создадим блок регистрация и авторизации.

<div class="text-end">

{% if current\_user.is\_authenticated %}

<span>Добро пожаловать, {{ current\_user.login }}</span>

<a href="{{ url\_for('user.logout') }}" class="btn btn-outline-light me-2">Выйти</a>

{% else %}

<a href="{{ url\_for('user.login') }}" class="btn btn-outline-light me-2">Войти</a>

<a href="{{ url\_for('user.registration') }}" class="btn btn-warning">Регистрация</a>

{% endif %}

</div>

В этом блоке так же реализована логика отображения разных элементов управления в зависимости от того авторизован пользователь или нет.

Шаблон для уведомления пользователя о результатах его действий реализован следующим образом.

{% with messages = get\_flashed\_messages(with\_categories=true) %}

{% if messages %}

{% for category, message in messages %}

<div class="alert alert-{{ category }}">

{{ message }}

</div>

{% endfor %}

{% endif %}

{% endwith %}

footer.html пока что выступает в качестве заглушки и несет на себе никаких интерактивных элементов.

**3.15 Разработка шаблонов библиотеки templates/user.**

Состоит из трех шаблонов. Основной из них это registration.html

Именно на этой странице будет реализовываться логика регистрации пользователя.

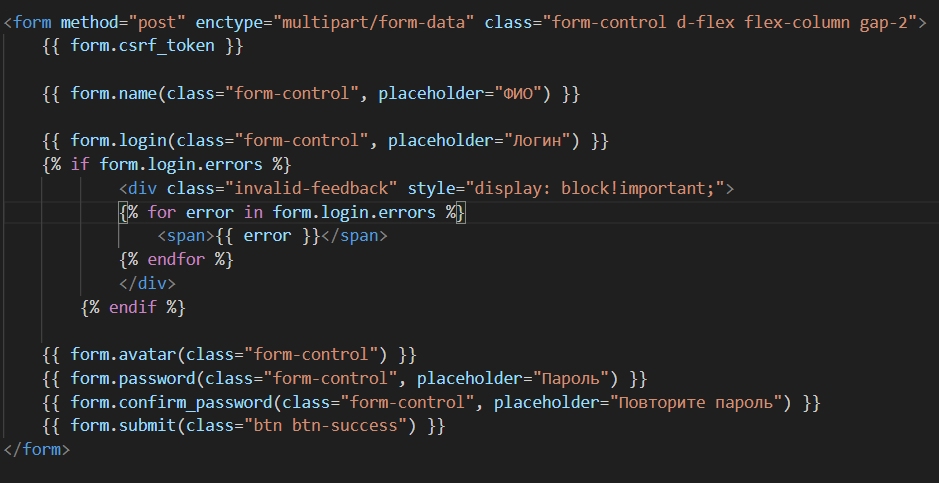
Как любой дочерний шаблон он будет наследоваться от base.html.

{% extends 'main/base.html' %}

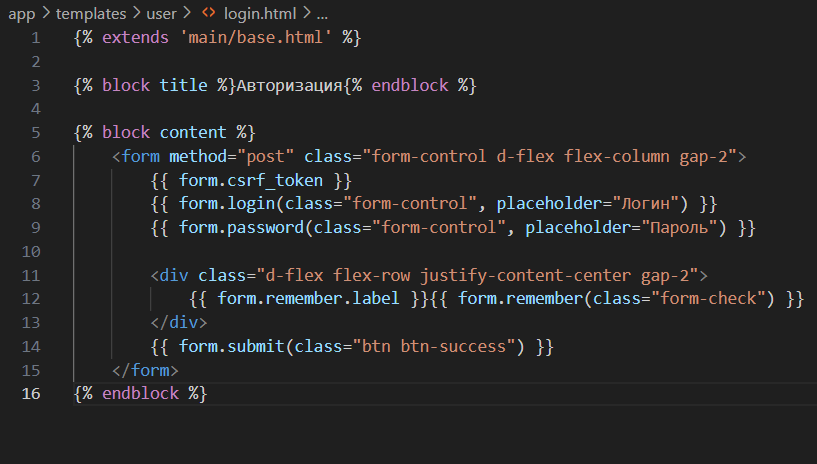
{% block title %}Регистрация{% endblock %}

Весь функционал будет реализован в блоке{% block content %}

Используем форму передаваемую в маршруте /user/registration



Здесь так же будет создан csrf\_token и проведена валидация вводимых данных.

Шаблон login.html будет иметь схожую структуру и функционал прописанный в соответствующем ему маршруте.

Шаблон user.html пока что не несут функционала. В перспективе он будет использован для отображения личного кабинета пользовтаеля.

**3.16 Разработка шаблонов библиотеки templates/post.**

Основным шаблоном будет all.html

Наследуется от main/base.html

{% extends 'main/base.html' %}

{% block title %}Все мероприятия{% endblock %}

{% block style %}

<link rel="stylesheet" href="{{ url\_for('static', filename='css/blocks/table.css') }}">

<<link rel="stylesheet" href="{{ url\_for('static', filename='css/libs/font awesome.min.css') }}">

{% endblock %}

Использует заданную веб форму и определяет токен сессии

<form method="post" class="form-control d-flex flex-row gap-2" id="author Form">

{{ form.csrf\_token }}

{{ form.organizer }}

<input type="submit" class="btn btn-success" value="Показать">

</form>

В блоке content основным элементом является таблица отображающие все актуальные мероприятия и содержащая доступные для каждой категории пользователей действия. Заголовок таблицы

<thead>

<tr >

<th scope="col">Организатор</th>

<th scope="col">Описание меропрития</th>

<th scope="col">Участники</th>

<th scope="col">Действия</th>

</tr>

</thead>

Тело таблицы

<tbody>

{% for post in posts %}

<tr scope="row">

<td>

<img width="24px" height="24px" src="{{ url\_for('static', filename='upload/'+post.author.avatar) }}" alt="Организатор">

{{post.author.name}}

</td>

<td>{{post.subject}}</td>

<td>

{% with user = user.query.get(post.participants) %}

<img width="24px" height="24px" src="{{ url\_for('static', filename='upload/'+user.avatar) }}" alt="Участник">

{{ user.name }}

{% endwith %}

</td>

<td>

{% if current\_user.is\_authenticated and current\_user.id == user.query.get(post.participants).id %}

<a href="{{ url\_for('post.feedback', id=post.id) }}" class="btn btn-sm btn-secondary"><span class="fa fa-comments"></span></a>

{% endif %}

{% if current\_user.is\_authenticated and post.author.id == current\_user.id %}

<a href="{{ url\_for('post.update', id=post.id) }}" class="btn btn-sm btn-secondary"><span class="fa fa-edit"></span></a>

<a href="{{ url\_for('post.delete', id=post.id) }}" class="btn btn-sm btn-danger"><span class="fa fa-trash"></span></a>

{% endif %}

</td>

</tr>

{% endfor %}

</tbody>

Для элементов действия реализована логика ограничения отображения для пользователей не обладающих соответствующими правами доступа.

Шаблон cteate.html использует передаваемую form.

{% extends 'main/base.html' %}

{% block title %}Добавление мероприятия{% endblock %}

{% block content %}

<form method="post" class="form-control d-flex flex-column gap-2">

<!-- <input name="organizer" type="text" placeholder="Организатор" class="form-control"> -->

<input name="subject" type="text" placeholder="Описание мероприятия" class="form-control">

<!-- <input name="participants" type="text" placeholder="Кто участвует" class="form-control"> -->

{{ form.participants }}

<input value="Добавить" type="submit" class="btn btn-success">

</form>

{% endblock %}

Тот же самый принцип работает и для шаблона update.html

{% extends 'main/base.html' %}

{% block title %}Обновление мероприятия {{ post.subject }} {% endblock %}

{% block content %}

<form method="post" class="form-control d-flex flex-column gap-2">

<input value="{{ post.subject }}" name="subject" type="text" placeholder="Описание мероприятия" class="form-control">

{{ form.participants }}

<input value="Обновить" type="submit" class="btn btn-success">

</form>

{% endblock %}

**3.17 Разработка шаблонов библиотеки templates/comment.**

Все шаблоны наследуются от base.html и отличаются только блоком content. Шаблон feedback.html

{% block content %}

<form method="post" class="form-control d-flex flex-column gap-2">

<div>

<<label for="example Form Control Textarea 1" class="form-label">Напишите ваш отзыв о мероприятии</label>

<textarea name="text" class="form-control" rows="5"></textarea>

</div>

<input value="Добавить" type="submit" class="btn btn-success">

</form>

{% endblock %}

Шаблон administration.html

{% block content %}

<table class="table custom-table">

<<thread>

<tr >

<th scope="col">Организатор</th>

<th scope="col">Действия</th>

</tr>

</</thread>

<tbody>

{% for temp in feedbacks %}

<tr scope="row">

<td>

{{user.query.get(temp.post\_author\_id).name}}

</td>

<td>

{% if current\_user.is\_authenticated and current\_user.status == 'root' %}

<a href="{{ url\_for('comment.work', id=temp.post\_author\_id) }}" class="btn btn-sm btn-danger"><span class="fa fa-trophy"></span></a>

{% endif %}

</td>

</tr>

{% endfor %}

</tbody>

</table>

{% endblock %}

Шаблон rating.html

{% block content %}

<table class="table custom-table">

<<thread>

<tr >

<th scope="col">Отзыв</th>

<th scope="col">Оценка</th>

</tr>

</</thread>

<tbody>

{% {% for i, j in feedback\_rating.items() %}

<tr scope="row">

<td> {{i}} </td>

<td> {{j}} </td>

</tr>

{% endfor %}

<td>

{% if current\_user.is\_authenticated and current\_user.status == 'root' %}

<a href="{{ url\_for('comment.add\_rating', author\_id = id )}}" class="btn btn-sm btn-secondary"><span class="fa fa-edit"> Опубликовать текущий рейтинг</span></a>

{% endif %}

</td>

<td>Средний рейтинг {{go}}</td>

</tbody>

</table>

{% endblock %}

**3.18 Предоставление прав доступа.**

Для безопасности и сохранения контроля над приложением функционал наделения правами доступа реализован через ручное изменение записи в базе данных. Ни один пользователь используя вебформы и функционал приложения не имеет доступа к редактированию этого параметра.

**3.19 Итоги разработки.**

Все основные компоненты нашего приложения написаны и интегрированы между собой. По сути у нас есть работающая минимальная версия приложения.

Завершающим этапом будет контейнеризация приложения и создания doker образов для переноса приложения на сервер.

Так же расширим функционал приложения добавлением нового модуля оценки обзывов.

**4. Добавление расширенного функционала**

Основная идея этого проекта это создание и отработка навыков создания модульных приложений. Мы создали приложение с базовым функционалом регистрации и создания карточек мероприятий.

Теперь мы расширим функционал добавив языковую модель для автоматического определения настроения отзывов.

Первой идеей было разработка собственной модели под конкретные требования сообщества, однако ограничения по времени и фокус на основной задаче вынудил отказаться от этой идеи. К тому же поиски актуальных размеченных датасетов для русского языка показал что таких данных практически нет или они имеют специфику непригодную для наших задач. В том числе и этот факт сместил фокус внимания с выбора проекта аналитического характера в пользу проекта разработки инструмента для сбора данных под конкретные задачи.

Были рассмотрены уже обученные модели в отрытом доступе, такие как ruBERT или ruGPT-3.5 Однако большинство моделей слишком тяжелы и требовательны к вычислительным ресурсам. Потому в качестве инструмента внедрения была выбрана достаточно простая и легкая модели из библиотеки dostoevsky.

**4.1 Языковая модель.**

Для того чтобы установить dostoevsky нужны зависимые библиотеки

pip install wheel

После чего устанавливаем

pip install dostoevsky

Загружаем уже предобученную модель

python -m dostoevsky download fasttext-social-network-model

На этапе тестирования разных моделей для определения их совместимости с нашими задачами мы использовали вычислительные ресурсы google collab. Однако по каким то причинам не все элементы интегрируются с коллабом. На этот случай загрузить модель можно из внешнего источника по ссылке

https://storage.b-labs.pro/models/fasttext-social-network-model.bin

Так же часто проявлялась ошибка несовместимости библиотеки с современными версиями взаимосвязанных библиотек.

Для устранения ошибки конфликта версий заменим в файле /venv/lib/python3.11/site-packages/fasttext/FastText.py в строках 46, 61 и 232 значение атрибута np.array( copy=False на copy=True)

Для реализации задуманного функционала нам нужно предобрабатывать текст ,удалять стоп слова и лемматизировать форму слов. Для этого установим библиотеку spacy и загрузим для нее модель работающую с русским языком.

pip install -U pip setuptools wheel

pip install -U spacy

python3 -m spacy download ru\_core\_news\_sm

Функционал будем реализовывать в модуле functions.py расположенном в основном каталоге нашего приложения.

Для удаления предупреждения о некорректности работы некоторых устаревших функций пропишем следующую строку

fasttext.FastText.eprint = lambda x: None

Импортируем нужные бибилотеки

import space

from dostoevsky.tokenization import Regexptokenizer

from dostoevsky.models import Fast Text Social Network Model

import fasttext

Напишем функцию которая будет принимать текст отзыва, предобрабатывать его передавать языковой модели. На выходе мы получим предсказание модели о том какая эмоциональная окраска у этого текста - положительная, нейтральная или негативная. На основании этого предсказания мы будем определять комментарий и формировать рейтинг участника к которому относится отзыв.

def review(text):

lemma = []

nlp2 = spacy.load("ru\_core\_news\_sm")

text = text

doc = nlp2(text)

# Удаляем стопслова

filtered\_words = [token.text for token in doc if not token.is\_stop]

# Получим очищенную строку слов

clean\_text = ' '.join(filtered\_words)

# Проведем лематизацию слов

for doc in nlp2.pipe([clean\_text]):

lemma.append([n.lemma\_ for n in doc])

# Получим очищенную строку слов приведенных к нормальной форме.

text\_lemma = ' '.join(lemma[0])

tokenizer = RegexTokenizer()

model = FastTextSocialNetworkModel(tokenizer=tokenizer)

# Посмотрим на результаты работы модели для оригинального и обработанного текста

results = model.predict([text])

results\_lem = model.predict([text\_lemma])

for items in results:

res = items.get('positive') - items.get('negative')

for items in results\_lem:

res\_lemm = items.get('positive') - items.get('negative')

# В итоговый вывод отдаем усредyенный результат обеих подходов

return round((res + res\_lemm)/2, 2)

Функция для суммирования результатов оценки отзывов.

def overall\_rating(list):

''' Функция для оценки общего настроения отзывов. Так как используемая модель не очень точна, то мы будем учитывать

только полярные значения, без значения веса предсказания'''

tmp = 0

for i in list:

if i > 0:

tmp += 1

elif i < 0:

tmp -= 1

tmp = tmp/len(list)

if tmp > 0.2:

res = 'Хороший'

elif tmp < -0.2:

res = 'Плохой'

else:

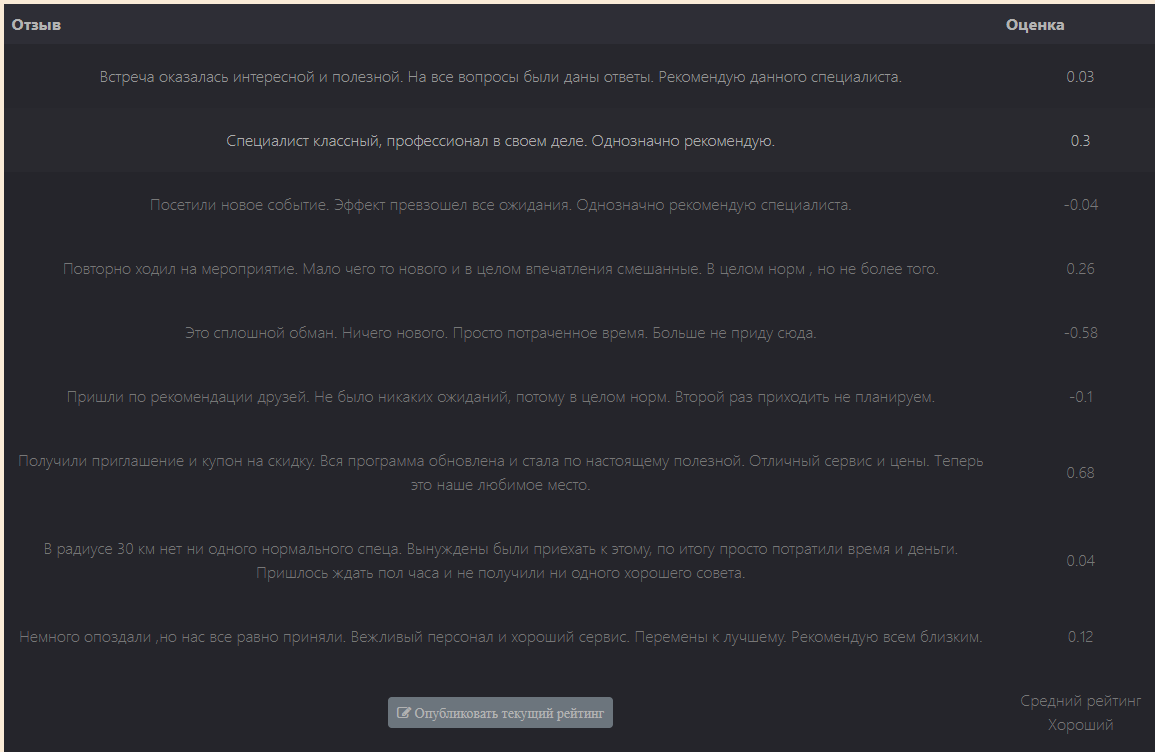
res = 'Нейтральный'

return res

**4.2 Тестирование языковой модели**

Модель весит всего около 80 мб, но вполне сносно справляется со своей задачей. К сожалению модель ошибается при анализе простых предложений с частицей «не». Но на более развернутых отзывах дает достаточно адекватные предсказания.

Ниже приведен пример работы модели уже непосредственно в разрабатываемом приложении.



**5. Подготовка проекта к внедрению.**

У нас есть готовое приложение и файлы внешнего окружения и конфигураций.

Создадим файл .dockerignore для того чтобы docker понимал какие элементы не нужно собирать в контейнер. А нашем случае это будет все виртуальное окружение и версии миграции нашей базы данных. Все виртуальное окружение будет автоматически устанавливаться из файла requirements.txt

Так же нам не нужны все файлы \_\_pychache\_\_/

Создадим докерфайл

touch Dockerfile

Опишем процедуру создания образа Docker

FROM python:3.11

WORKDIR /app

ADD . /app

RUN apt install gcc -y

RUN pip install -r requirements.txt

CMD ["uwsgi", "app.ini"]

Установим компилятор языка C, C++.

sudo apt install gcc

Нам нужно установить контейнер сервера приложений в свое виртуальное окружение.

pip install uwsgi

Создадим файл конфигурации uwsgi

touch app.ini

Определим конфигурацию

[uwsgi]

wsgi-file = run.py

callable = application

socket = :8080

processes = 4

threads = 2

master = true

chmod-socket = 660

vacuum = true

die-on-term = true

Создадим файл точки входа

touch run.py

В файле run.py импортируем функцию сборки нашего приложения и файл переменного окружения.

from app import create\_app

from dotenv import load\_dotenv

load\_dotenv('.env')

application = create\_app()

Создадим файл docker-compose.yml чтобы настроить все сервисы приложения и с помощью одной команды запусткать их в изолированных контейнерах.

touch docker-compose.yml

Определим параметры запуска

services:

postgres:

image: postgres

container\_name: postgres

volumes:

- ~/.pg/pg\_data/flask-yt:/var/lib/postgresql/data

env\_file:

- .env

ports:

- "5532:5432"

flask:

build:

dockerfile: Dockerfile

context: .

container\_name: flask

volumes:

- ~/.uploads/images/flask-yt:/app/app/static/upload

env\_file:

- .env

environment:

- POSTGRES\_HOST=${POSTGRES\_HOST}

- POSTGRES\_PORT=${POSTGRES\_PORT}

- POSTGRES\_USER=${POSTGRES\_USER}

- POSTGRES\_PASSWORD=${POSTGRES\_PASSWORD}

- POSTGRES\_DB=${POSTGRES\_DB}

depends\_on:

- postgres

expose:

- 8080

nginx:

build:

dockerfile: ./Dockerfile

context: ./nginx/

container\_name: nginx

env\_file:

- .env

depends\_on:

- flask

ports:

- "8080:80"

Определим конфигурацию работы сервера Ngnix.

Для это создадим директорию Ngnix с двумя файлами настройки

mkdir nginx

touch nginx.conf

Опишем параметры запуска для конфига nginx

server {

listen 80;

client\_max\_body\_size 1G;

location / {

include uwsgi\_params;

uwsgi\_pass flask:8080;

}

}

Создадим докерфайл для ngnix

touch Dockerfile

Опишем параметры

FROM nginx

RUN rm /etc/nginx/conf.d/default.conf

COPY nginx.conf /etc/nginx/conf.d/

Создадим файл, который содержит перечень всех модулей и пакетов, необходимых для корректной работы приложения

pip freeze >requirements.txt

Все файлы окружения и настройки сборки готовы. Можно собирать контейнер. Так как основная ОС у нас Windows, запустим докер десктоп.

Проверим активная ли интеграция с WSL.

Дадим пользовтаелю от имени которого собиарется контейнер доступ к группе docker

sudo usermod -aG docker devel

Перед началом сборки контейнера проверяем и актализируем все настройки и переменные для запуска на внешнем сервере. Host меняем с локального для разработки на указанный в сборщике контейнера.

Запустим сборку контейнера с нашим приложением.

sudo docker compose build

Сборка контейнера завершена без ошибок после чего мы можем запустить контейнер.

sudo docker compose up

Теперь мы можем получить доступ к нашему приложения по порту указаному в сборке. В нашем случае это http://127.0.0.1:8080

**6. Перспективы дальнейшей проработки проекта.**

У нас есть работающее приложение. Мы изучили и освоили комплекс технологий позволяющий создавать реальный продукт .

Идей как улучшить приложение множество. Актуальной задачей является внедрение в приложение интерактивной карты мероприятий, чтобы у пользователей была возможность подбора в зависимости от их текущей локации.

Можно создать модуль анкетирования, чтобы собирать данные по интересующей теме — на основании которых можно обучать нейросети под конкретные задачи.

Востребован функционал организации и координации локальной экономической кооперацией, осуществить которую можно на основе схожего приложения.

Важно оттачивать навыки полноценной разработки и делать акцент на разработках ,которые могут быть полезны реальным пользователям.

**7. Итоге проекта.**

Нам удалось достичь поставленной цели и получить функционально работающее приложение и развернуть его на внешнем сервере.

К сожалению найти партнеров для совместной разработки не удалось и ввиду дефицита ресурсов и времени проект не получил расширенного функционала.

Однако использованные технологии и навыки позволяют продолжить освоение специальности и гораздо актуальней понимать возникающие в процессе проектирования и разработки проблемы.

Ссылка на проект

https://github.com/AlexCorpROS/diplom

**Список используемых источников:**

1. **Курсы GeekBrains https://gb.ru/lessons/**
2. **https://pythonru.com/uroki/11-rabota-s-formami-vo-flask?ysclid=m1c8obdc40457018245**
3. **https://habr.com/ru/articles/193242/**
4. [**https://pypi.org/project/dostoevsky/**](https://pypi.org/project/dostoevsky/)