Реализация приложения

1. Подготовка проекта.

1.1. Создание виртуального окружения и установка фреймворка Flask

При работе с проектами, используется виртуальное окружения. Таким образом изолируем наш проект от других, хранящихся на этом же ПК. Кроме того изменения внутри проекта не влияют на "эталонную" версию Python, которую только что установили.

Для создания виртуального окружения в каталоге нашего проект выполним команду **python -m venv venv**, таким образом внутри проекта будет создана папка **venv** в которой будет находиться локальная версия Python с которой мы будем работать. Далее следует активировать виртуальное окружение командой **venv\Scripts\activate**. В папке **venv** будут храниться все зависимости проекта.

Командой **pip install Flask** устанавливаем фреймворк Flask в виртуальное окружение.

1.2. Инициируем приложение Flask в файле**\_\_init\_\_.py**.

В директорию проекта flask\_blog поместим вложенную директорию с именем **blog\_app** . Здесь будут находиться непосредственно компоненты проекта. Во вложенную директорию **blog\_app** поместим специальный файл **\_\_init\_\_.py**, тем самым определим директорию в качестве пакета. Благодаря пакетам обеспечивается возможность работы с компонентами проекта при импортах посредством определения уровней вложенности (через точку). Важно отметить, что, если файл **\_\_init\_\_.py** присутствует в директории, он вызывается при импорте этого пакета или входящего в него модуля. Обычно файл **\_\_init\_\_.py** оставляют пустым, но он также может содержать выражения, обеспечивающие инициализацию некоторых важных параметров на уровне пакета. Наше приложение будем выполнять в файле **\_\_init\_\_.py**

На рисунке 1.1 показано создание приложения на Flask в файле **\_\_init\_\_.py**

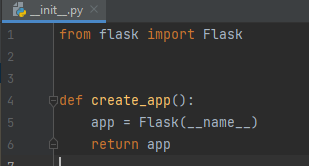


Рисунок 1.1 – Создание объекта приложения -Flask

В файле **\_\_init\_\_.py** создаем экземпляр класса Flask и сохраняем его в переменную app (хранит объект приложения). При создании экземпляра этого класса передается обязательный параметр **\_\_name\_\_**, который соответствует названию пакета, в котором Flask будет искать статические файлы, шаблоны и т.д.

1.3 Создание файла для запуска локального сервера проекта

Создаем в корне проекта файл **wsgi.py** с помощью которого будет производиться запуск проекта.

На рисунке 1.2 код в файле **wsgi.py** для запуска проекта

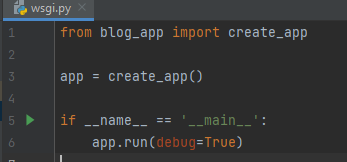


Рисунок 1.2 - Код в файле **wsgi.py** для запуска проекта

В файле **wsgi.py** мы создаем объект Flask-приложения на основе написанной ранее функции **create\_app()**. Функцию **run(**) используем для запуска локального сервера с приложение. Условие **If \_\_name\_\_==’\_\_main\_\_’:** означает что сервер будет запущен только при запуске текущего скрипта (**wsgi.py**) из Python-интерпретатора, т.е. в качестве запускаемой программы, а не при импорте **wsgi.py**. Режим отладки (**debug = True**) используется только на этапе разработки.

На рисунке 1.3 представлен скрин консоли после запуска локального сервера.

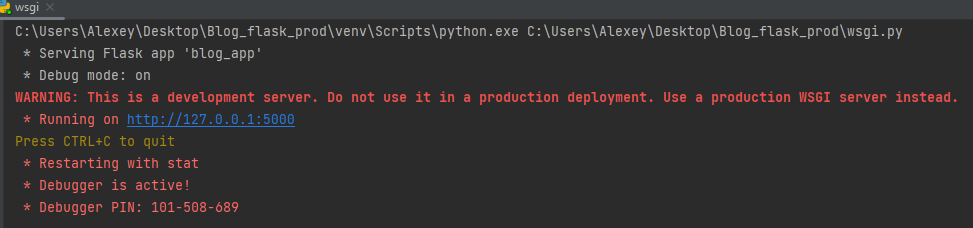


Рисунок 1.3 - Скрин консоли после запуска локального сервера

1.4 Создание файла конфигурации

Приложениям требуется определенная конфигурация. Существуют различные настройки, которые вы, возможно, захотите изменить в зависимости от среды приложения, такие как переключение режима отладки, установка секретного ключа и другие подобные вещи, зависящие от среды. Создадим в директории для компонентов проекта файл **config.py** и в нем разместим путь и название базы данных и секретный ключ для реализации системы аутентификации (рис 1.4)

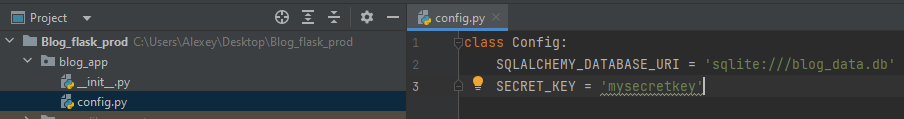


Рисунок 1.4 – Создание файла конфигурации

2. Создание основных зависимостей (компонентов) приложения

2.1 Реализация системы аутентификации

Для реализации аутентификации пользователей приложения используем готовый модуль Flask-Login который обеспечивает управление сеансами пользователей для Flask. Он выполняет обычные задачи по входу в систему, выходу из системы и запоминанию сеансов ваших пользователей в течение длительных периодов времени. Для установки расширения используем команду **pip install flask-login**. В файл **\_\_init\_\_.py** импортируем из библиотеки **flask\_login** класс **LoginManager** и создаем объект login\_manager класса LoginManager (рис. 2.2)

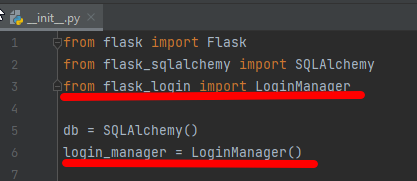
\

Рисунок 2.1 - Создаем объект login\_manager класса LoginManager

Для аутентификации пользователей в файл **models.py**, где описываются модели базы данных импортируем объект login\_manager класса LoginManager и класс-помощник UserMixin из модуля flask\_login который добавляет дополнительный функционал. Декоратор **@login\_manager.user\_loader** (рис.2.2). указывает, как загружать пользователя на основе его идентификатора. Он используется для регистрации функции load\_user() в Flask-Login в качестве загрузчика пользователя. Эта функция вызывается при каждом запросе, когда необходимо получить информацию о текущем пользователе на основе его идентификатора. Функция load\_user() принимает один аргумент - идентификатор пользователя (user\_id). Она должна определить, как получить объект пользователя на основе переданного идентификатора. В данном случае, она использует метод User.query.get(int(user\_id)) для получения пользователя из базы данных на основе его идентификатора. В результате, функция load\_user() возвращает объект пользователя, соответствующий переданному идентификатору.

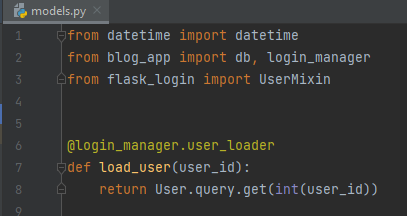


Рисунок 2.2 – Система аутентификации пользователей

2.2 Реализация взаимодействия с базой данных.

2.2.1 Создание и инициализацию SQLAlchemy для Flask-приложения

В нашем приложении в качестве системы хранения данных будем использовать встраиваемую кроссплатформенная БД SQLite. Для создания моделей данных и взаимодействия с БД SQLite используем расширение Flask-SQLAlchemy. Для установки расширения используем команду в терминале **pip install Flask-SQLAlchemy**.

В файл **\_\_init\_\_.**py, где реализуется приложение, импортируем из библиотеки flask\_sqlalchemy класс-конструктор SQLAlchemy и создаем объект db класса SQLAlchemy (рис. 2.2) для взаимодействия с базой данных.

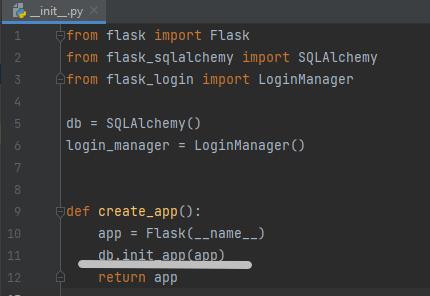


Рисунок 2.2 – Создание объекта и инициализация db класса SQLAlchemy

В файле **\_\_init\_\_.py** связываем объект db (экземпляр класса SQLAlchemy) с Flask-приложением через выражение **db.init\_app(app)**. Метод init\_app() выполняет инициализацию SQLAlchemy для Flask-приложения. Он принимает объект app, который является экземпляром класса Flask- приложения. Таким образом получили возможность использовать SQLAlchemy в Flask-приложении для работы с базой данных.

2.3 Создание моделей базы данных приложения

В Flask модели – это классы, которые представляют сущности в базе данных. Модели используются для организации и хранения данных в структурированном формате. Одним из популярных способов работы с базами данных в Flask является использование ORM библиотеки, таких как SQLAlchemy.

Проект – блог подразумевает регистрацию пользователей, написание ими своих постов, оставление комментариев и реакций в виде лайков к чужим постам. Для реализации этих возможностей были определены следующие модели: User для хранения данных о пользователе, Post – для хранения записей пользователя, Comment – для оставления комментариев к постам пользователей, Like – для возможности оставления реакции на содержание постов.

Создадим в пакете **blog\_app** файл **models.py** для описания моделей проекта. В него импортируем объект db класса SQLAlchemy для создания моделей.

Создаем класс User (рис. 2.3) который наследуется от класса Model, находящегося внутри объекта db. Внутри класса User создаем переменные id для хранения идентификатора пользователя (является первичным ключом), переменная username для хранения имени пользователя, email для хранения электронной почты, image\_file для хранения фото (аватарки) пользователя, password для хранения пароля для входа, переменная post определяет отношение между пользователем и его постами, т.е. указывает SQLAlchemy, что у пользователя может быть несколько постов.

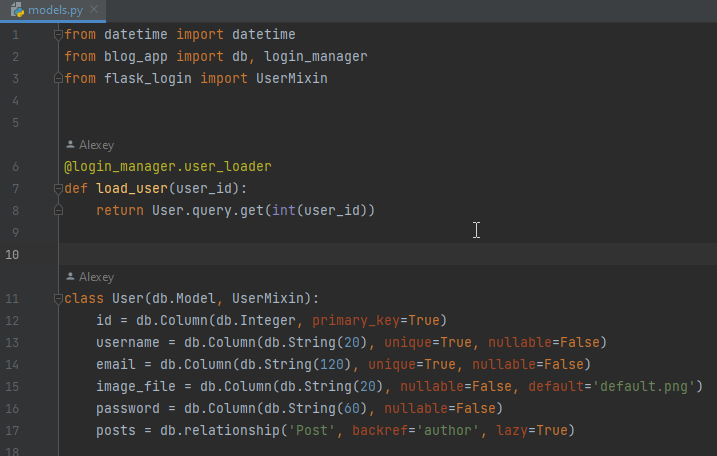


Рисунок 2.3 – Модель пользователя (User) приложения

На рисунке XXX представлена таблица созданная на основе модели User

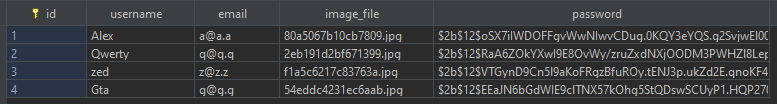


Рисунок ХХХ – Таблица пользователя приложения

Создаем класс Post (рис. 2.4), наследуемый от класса Model, находящегося внутри объекта db. Внутри класса Post создаем переменные **id** для хранения идентификатора поста (является первичным ключом), title для хранения имени пользователя, **date\_post** для хранения даты публикации поста, **content** для хранения текста поста, **user\_id** идентификатор автора поста, **comments** – определяет отношение между постом и комментариями к нему, т.е получение всех комментариев относящихся к посту (связь один ко многим). Следует отметить, что использование **cascade='all, delete-orphan'** указывает SQLAlchemy, что каскадное удаление должно применяться при удалении поста. Это означает, что все комментарии, связанные с удаляемым постом, также будут удалены.

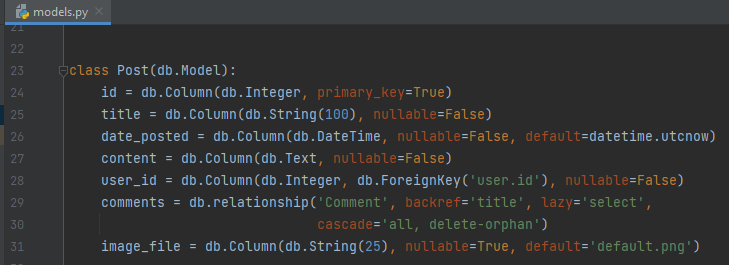


Рисунок 2.4 – Модель поста (Post) приложения

Создаем класс Comment (рис. 2.5), наследуемый от класса Model. Внутри класса Comment создаем переменные **id** для хранения идентификатора комментария, text\_comment для хранения текста комментария, date\_comment для хранения времени публикации комментария, post\_id для хранения id поста к которому пишем комментарий, username для хранения имени (логина) автора поста.

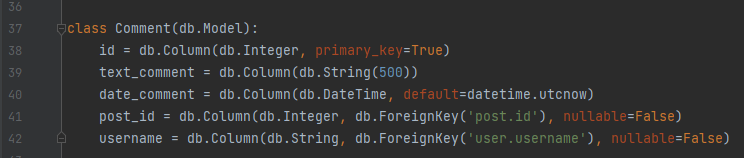
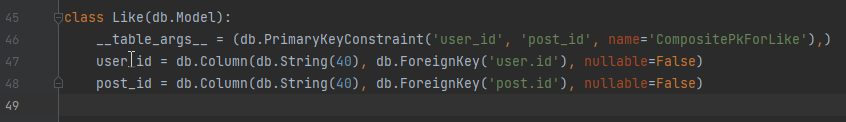


Рисунок 2.5 – Модель комментариев (Comment) приложения

Создаем класс Like (рис. 2.6), наследуемый от класса Model. Внутри класса Like создаем переменные user\_id для хранения id пользователя, post\_id для хранения id поста.



ОПИСАНИЕ И ПРО МИГРАЦИИИ