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**

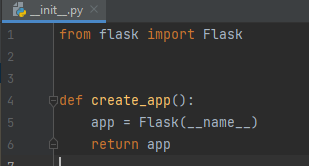


Рисунок 1.1 – Создание объекта приложения -Flask

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

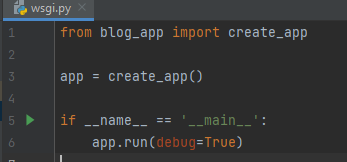


Рисунок 1.2 - Код в файле **wsgi.py** для запуска проекта

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.

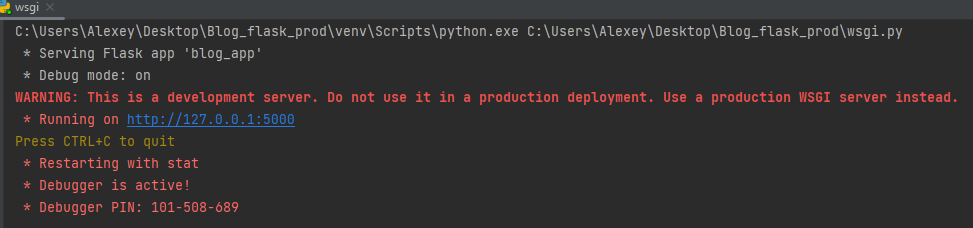


Рисунок 1.3 - Скрин консоли после запуска локального сервера

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)

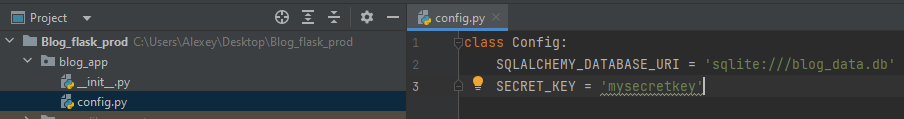


Рисунок 1.4 – Создание файла конфигурации

В файл \_\_init\_\_.py импортируем класс `Config` из модуля `blog\_app.config` и загружаем конфигурацию в наше приложение из объекта `Config, т.е. устанавливаем конфигурацию Flask-приложения.

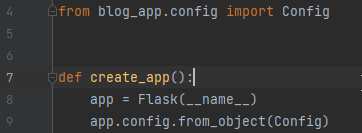


Рисунок ХХХ -установка конфигурации приложения

2. Реализация системы аутентификации приложения на Flask

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). Внутри функции мы инициализируем объект login\_manager, вызывая метод init\_app() и передавая объект app в качестве аргумента.

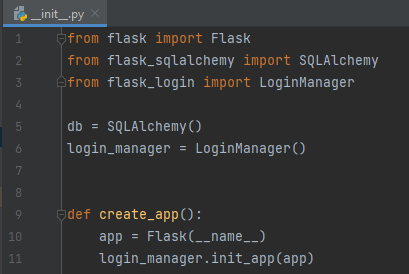
 \

Рисунок 2.1 - Создаем объект login\_manager класса LoginManager и ициирование

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.

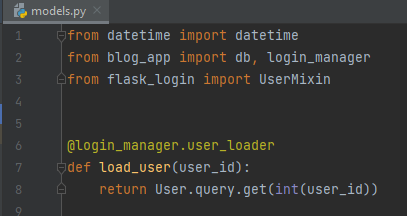


Рисунок 2.2 – Система аутентификации пользователей

3 Реализация взаимодействия с базой данных.

3.1 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.

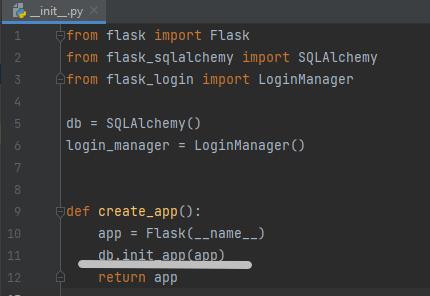


Рисунок 2.2 – Создание объекта и инициализация db класса SQLAlchemy

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.

3.2 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.

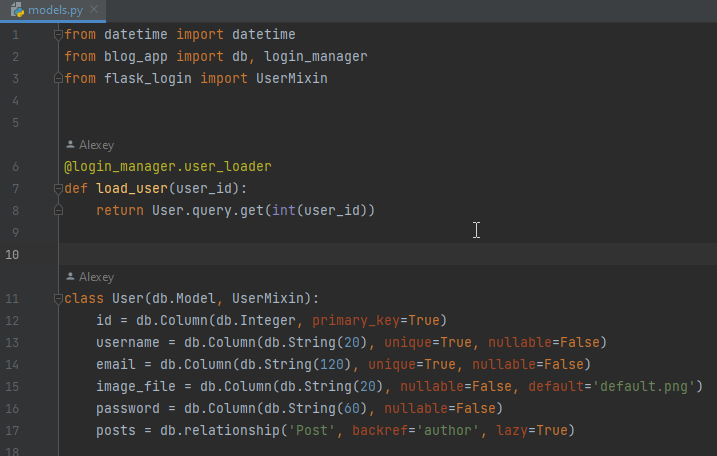


Рисунок 2.3 – Модель пользователя (User) приложения

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.

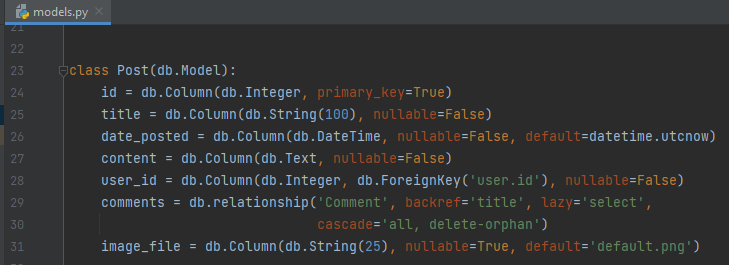


Рисунок 2.4 – Модель поста (Post) приложения

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.

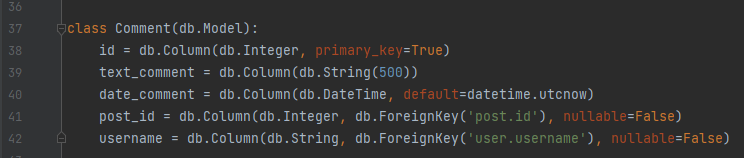


Рисунок 2.5 – Модель комментариев (Comment) приложения

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, а также \_\_table\_args\_\_ являющаяся дополнительным аргументом таблицы, используемым для определения составного первичного ключа, состоящего из двух столбцов user\_id и post\_id.

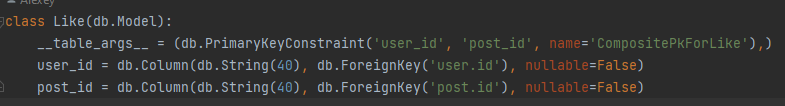


Рисунок ХХХ– Модель реакций (Like) приложения

3.3 Создание и таблиц базы данных приложения

После описания моделей нашего приложения необходимо создать соответствующие им таблицы в базе данных. Для этого использовалась функция внутри которой использовался метод **create\_all()** через консольную команду **flask init-db** создает таблицы в базе данных (рис.ХХХ).

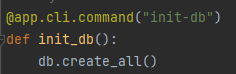


Рисунок ХХХ –

. Сгенерированная на основе моделей база данных будет храниться в папке **instance** на уровне нашего проекта (рис.ХХХ).

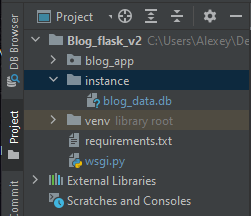


Рисунок ХХХ – Расположение базы данных в проекте

Ниже представлены получившиеся на основания написанных моделей таблицы

На рисунке XXX представлена таблица созданная на основе модели User

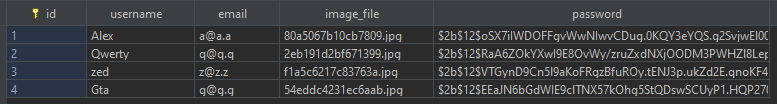


Рисунок ХХХ – Таблица пользователя приложения

На рисунке XXX представлена таблица созданная на основе модели Post

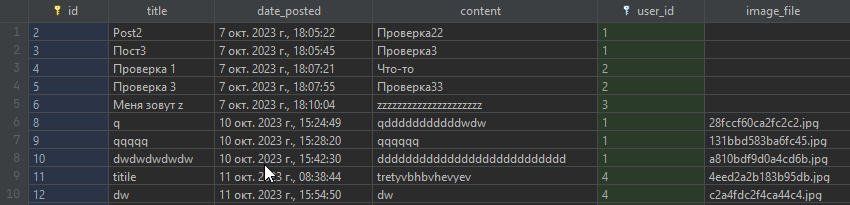


Рисунок ХХХ – Таблица постов приложения

На рисунке XXX представлена таблица созданная на основе модели Comment

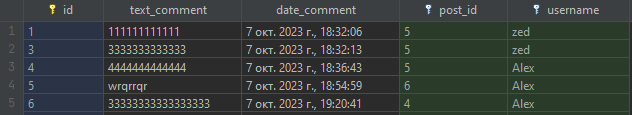


Рисунок ХХХ – Таблица комментариев приложения

На рисунке XXX представлена таблица созданная на основе модели Like

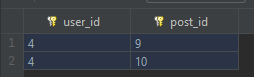


Рисунок ХХХ – Таблица реакций (лайков) приложения

3.4 Миграции базы данных

При разработке приложения нередко приходится менять схемы таблицы, добавляя или удаляя новый функционал. Развитие приложения потребует вносить подобные изменения. метод create\_all() создает базу данных с таблицами. Он может добавить новую таблицу в базу из модели, но изменить поля и связи этот метод не даст, вызвав исключение. У flask есть расширение, flask-migrate, которое интегрирует инструмент для миграции базы данных для SQLAlchemy, которое устанавливается командой **pip install flask-migrate.**

Для настройки механизма миграций базы данных в модуль \_\_init\_\_.py приложения добавим следующий код (рис.ХХХ)

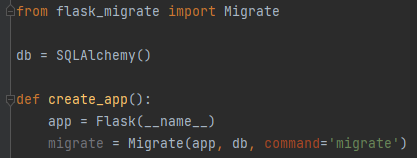


Рисунок ХХХ – Настройка механизма миграции

Импортируем класс **Migrate** из модуля **flask\_migrate**, который предоставляет инструменты для управления миграциями базы данных. Создается объект migrate типа Migrate, который связывает экземпляр Flask-приложения app с экземпляром базы данных db и определяет команду для выполнения миграций (команда - **migrate**). Чтобы использовать обращение из flask к инструменту миграций, создадим в консоли переменную окружения FLASK\_APP (рис. ХХХ)



Рисунок ХХХ – Установка переменной окружения FLASK\_APP

Репозиторий для миграций создается командой терминала **flask db init**, после чего появится папка **migrations** на уровне проекта**.** Далее создаем миграцию командой **flask db migrate**, и применяем ее командой **flask db upgrade.**

4 Создание приложения -модуля main для главной страницы блога

4.1. Создание макета приложения (blueprint) для главной страницы

Для создания независимых модулей приложения будем пользоваться макетами (blueprint). Создадим в приложении пакет **main** в который добавим модуль routes.py. В нем создадим макет приложения main для главной страницы (рис. ХХХ)

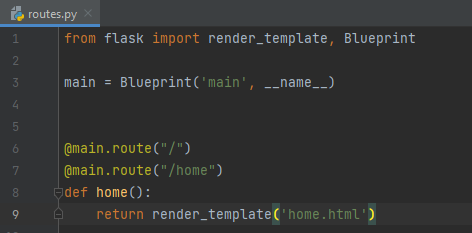


Рисунок ХХХ – Модуль **routes.py**

Импортируем две функции render\_template и Blueprint из модуля flask. Создаем объект main с использованием класса Blueprint, которому передаем имя main и атрибут name. Декораторы @main.route("/") и @main.route("/home") указывают на то, что функция-контроллер home() будет обрабатывать запросы для корневого пути ("/") и пути "/home". Внутри функции home() мы вызываем функцию render\_template, которая возвращает HTML-шаблон с именем "home.html". Этот шаблон будет отображаться в браузере пользователя.

На рисунке ХХХ показана регистрация приложение main для главной страницы в файле \_\_init\_\_.py нашего приложения

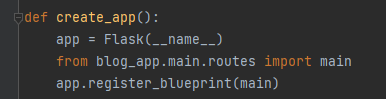


Рисунок ХХХ -Регистрация приложения **main** в файле **\_\_init\_\_.py**

4.1.2 Подключение базового шаблона и шаблона главной страницы приложения

Для взаимодействия в браузере пользователя с приложением используются шаблоны. Хранение шаблонов (HTML-файлов) осуществляется в папке **templates** внутри нашего приложения.

На рисунке ХХХ представлен базовый шаблон приложения. От него будут наследоваться все остальные шаблоны приложения. Этот шаблон использует Jinja2, шаблонный движок, который позволяет использовать условные операторы (if-else) и циклы (for) непосредственно в HTML-коде. Он также использует переменные и функции Flask для динамического отображения данных

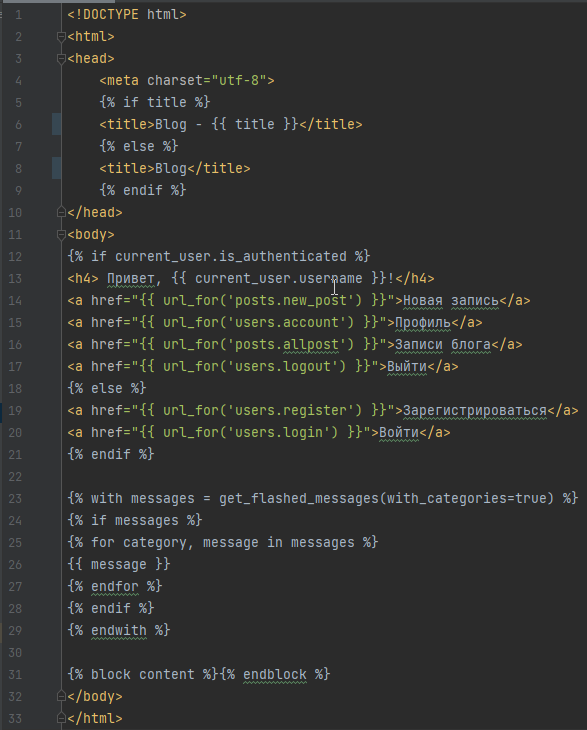


Рисунок ХХХ – Базовый шаблон приложения

В базовом шаблоне происходит проверка **{% if current\_user.is\_authenticated %}** и **{% else %}**, аутентифицирован ли текущий пользователь. Если пользователь аутентифицирован, то отображается приветствие и ссылки на разные части приложения. В противном случае, отображаются ссылки на регистрацию и вход.

Ниже приведен скрин главной страницы приложения (файл home.html – Приложение 1), наследуемого от базового шаблона содержащего гиперссылки навигации приложения

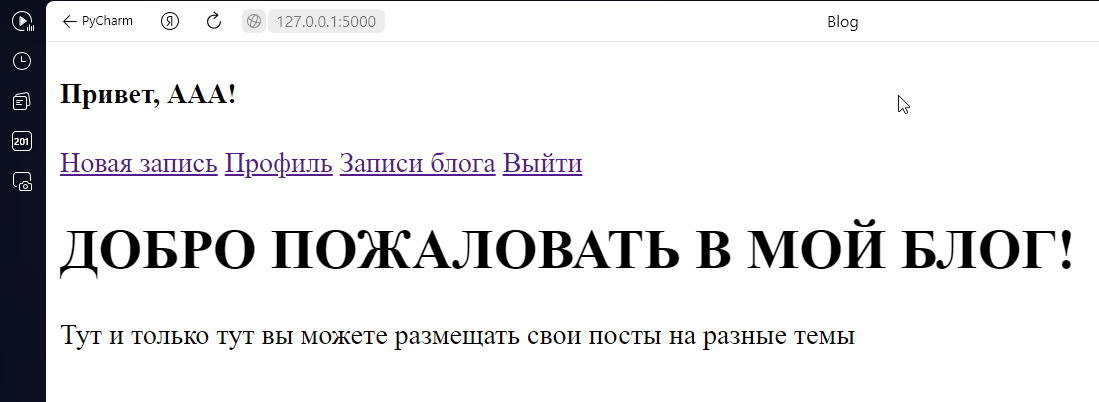


Рисунок ХХХ-Главная страница приложения в браузере

На главной страницы приложения

5. Реализация приложения (модуля **users)** для взаимодействия с пользователями блога

5.1 Создание форм для взаимодействия с пользователем

Формы являются средствами взаимодействия пользователя с приложением. В проекте будем использовать модуль Flask-WTForm, которое является расширением [Flask](https://translated.turbopages.org/proxy_u/en-ru.ru.ac5da8f8-65808bab-18b8d7b6-74722d776562/https/www.geeksforgeeks.org/python-introduction-to-web-development-using-flask/), интегрирующее библиотеку WTForms. Установки расширения используем команду **pip install Flask-WTF**.

В пакете **users** внутри нашего приложения **blog\_app** создадим файл **forms.py** в котором находиться код форм. Импортируем в файл **forms.py** модули необходимые для создания форм (рис. ХХХ)

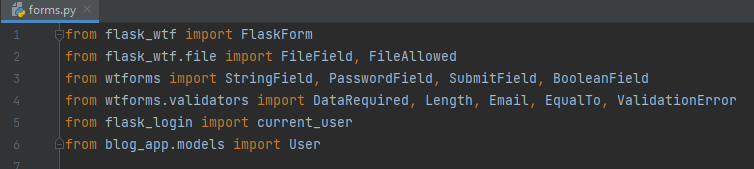


Рисунок ХХХ -Импорт модулей в файл **forms.py**

От импортированного из библиотеки **flask\_wtf** класса **FlaskForm** будут наследоваться все формы приложения. Поле формы **FileField** предназначено для загрузки файлов. Поле формы **FileAllowed** - функция-валидатор, используется для проверки типов разрешенных файлов, которые могут быть загружены через файловое поле формы. **StringField**, **PasswordField**, **SubmitField**, **BooleanField** - поля формы для ввода строковых значений, пароля, кнопки отправки формы и логического флажка. Импортируемы из модуля **wtforms.validators** (предоставляет набор встроенных валидаторов) **DataRequired**, **Length**, **Email**, **EqualTo** - функции-валидаторы, которые проверяют различные условия ввода данных. **ValidationError** - класс исключения, используемый для обработки ошибок валидации форм. **Сurrent\_user** - объект, представляющий текущего пользователя, который предоставляется **Flask-Login**. Импортируем из нашего приложения **blog\_app** модель пользователя **User.**

5.2.1 Форма для регистрации нового пользователя.

Код формы для регистрации пользователем своего аккаунта (рис.ХХХ) реализован в файле **forms.py**

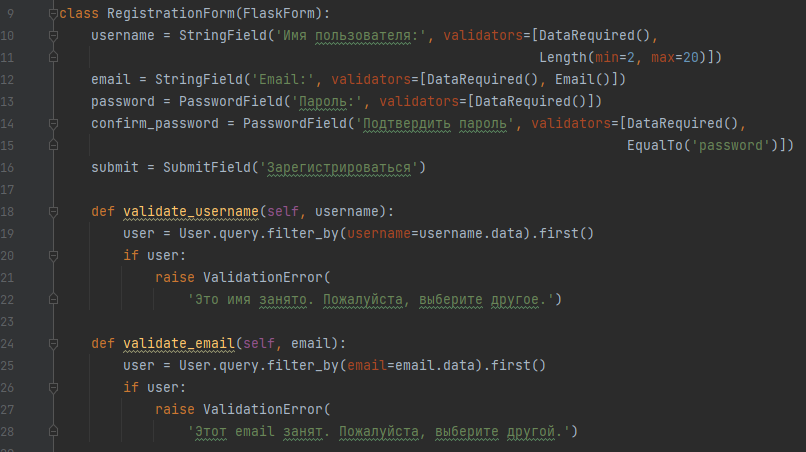


Рисунок ХХХ– Код формы для регистрации нового пользователя

Определяем класс формы **RegistrationForm** который наследуется от класса **FlaskForm**, предоставляемого Flask-WTF. Внутри класса создаются следующие поля:

- поле **username** типа **StringField**, предназначенное для ввода имени пользователя. Этому полю добавляются валидаторы: **DataRequired**, для проверки, что поле было заполнено, и **Length**, чтобы ограничить длину введенных данных от 2 до 20 символов;

- поле **email** также типа **StringField** для ввода адреса электронной почты. Ему также добавляются валидаторы **DataRequired** и **Email**, чтобы проверить корректность введенного email адреса;

- поле **password** типа **PasswordField**, предназначенно для ввода пароля, с валидатором **DataRequired**, чтобы проверить, что поле было заполнено;

- поле **confirm\_password** типа **PasswordField** для повторного ввода пароля для подтверждения. Ему добавляются валидаторы **DataRequired**, чтобы проверить, что поле было заполнено, и **EqualTo**, чтобы проверить, совпадает ли значение этого поля с паролем в поле **password**;

**-** поле **submit** типа **SubmitField**, которое представляет кнопку отправки формы с надписью "Зарегистрироваться".

Внутри класса определяем метод **validate\_username**, который выполняет пользовательскую проверку на уникальность имени пользователя. Он осуществляет запрос к базе данных, чтобы убедиться, что имя пользователя еще не занято. В случае обнаружения совпадения, вызывается исключение **ValidationError**. Здесь определяется метод **validate\_email**, выполняющий аналогичную пользовательскую проверку для определения уникальности email. Он также осуществляет запрос к базе данных для проверки уникальности email, и если email уже занят, вызывается исключение **ValidationError**.

5.2.2 Форма для авторизации пользователя

Код формы для авторизации пользователя (рис.ХХХ) реализован в файле **forms.py**

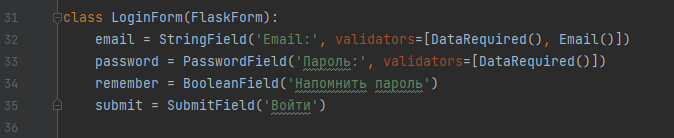


Рисунок ХХХ - Код формы для авторизации пользователя

Определяем класс формы **LoginForm**, который наследуется от класса **FlaskForm**. Внутри класса создаются следующие поля:

- поле **email** типа **StringField** для ввода адреса электронной почты. В него добавляются валидаторы **DataRequired** для проверки, что поле было заполнено, и **Email** для проверки корректности введенного email;

- поле **password** типа **PasswordField** для ввода пароля. Ему также добавляется валидатор **DataRequired**, чтобы проверить, что поле было заполнено;

- поле **remember** типа **BooleanField**, которое представляет чекбокс (в данном случае возможность поставить галочку для запоминания пароля ) для пользовательской опции "Напомнить пароль".

- поле **submit** типа **SubmitField**, которое представляет кнопку отправки формы с надписью "Войти"

5.2.3 Форма для изменения данных профиля пользователя.

Код формы для авторизации пользователя (рис.ХХХ) реализован в файле **forms.py**

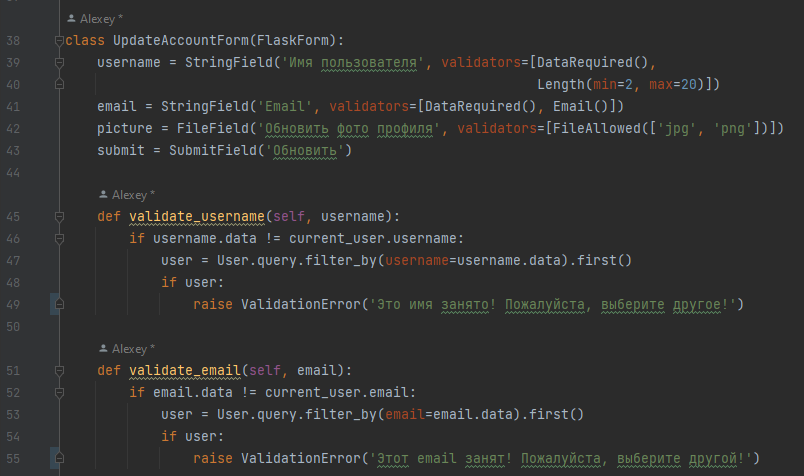


Рисунок ХХХ - Код формы для изменения данных профиля пользователя

Определяем класс формы **UpdateAccountForm**, который наследуется от класса **FlaskForm**. Внутри класса создаются следующие поля:

- поле **username** типа **StringField** для ввода имени пользователя. Для этого поля добавлены валидаторы **DataRequired** для проверки, что поле было заполнено и **Length** для ограничения длины имени от 2 до 20 символов;

- поле **email** также типа **StringField** для ввода адреса электронной почты. Также добавлены валидаторы **DataRequired** и **Email** для проверки корректности введенного email;

- поле **picture** типа **FileField** для обновления фото профиля. К нему добавляется валидатор **FileAllowed**, который проверяет, что разрешены только файлы с расширениями "jpg" и "png";

- поле **submit** типа **SubmitField,** которое представляет кнопку отправки формы с надписью "Обновить";

Внутри класса определяется метод **validate\_username**, который выполняет пользовательскую проверку на уникальность имени пользователя. Если введенное имя пользователя не совпадает с текущим или оно уже занято другим пользователем, вызывается исключение **ValidationError**. Метод **validate\_email**, выполняет аналогичную пользовательскую проверку на уникальность email. Если введенный email не совпадает с текущим или он уже занят другим пользователем, вызывается исключение **ValidationError**.

5.2.4 Формы для запроса на изменение (обновления) пароля пользователя и форма для изменения пароля пользователя.

На рисунке ХХХ приведен код формы **RequestResetForm** предназначенной для запроса на изменение пароля пользователя.

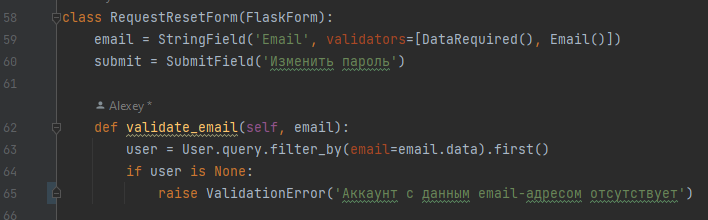


Рисунок ХХХ - Код формы запроса на изменение пароля пользователя

Определяется класс формы **RequestResetForm**, который наследуется от класса **FlaskForm**. Внутри класса создаются следующие поля:

- поле **email** типа **StringField** для ввода адреса электронной почты. В него добавляются валидаторы **DataRequired** для проверки, что поле было заполнено, и **Email** для проверки корректности введенного email;

- поле **submit** типа **SubmitField**, представляющее кнопку отправки формы с надписью "Изменить пароль".

Внутри класса определяется метод **validate\_email**, который выполняет пользовательскую проверку наличия аккаунта с указанным email-адресом. Если пользователя с указанным email-адресом не существует, вызывается исключение **ValidationError**.

На рисунке ХХХ приведен код формы предназначенной для изменения пароля пользователя.

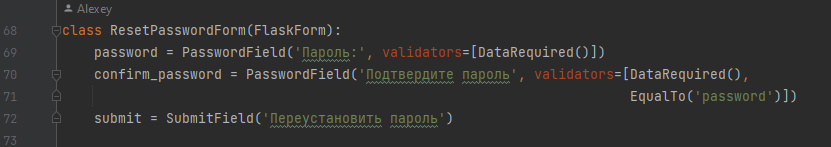


Рисунок ХХХ - Код формы изменения пароля пользователя

Определяется класс формы **ResetPasswordForm**, который наследуется от класса **FlaskForm**. Внутри класса создаются следующие поля:

- поле **password** типа **PasswordField**, предназначенное для ввода нового пароля. К нему добавляется валидатор **DataRequired**, чтобы проверить, что поле было заполнено;

- поле **confirm\_password** типа **PasswordField** для повторного ввода нового пароля для подтверждения. К нему добавляются валидаторы **DataRequired** для проверки что поле было заполнено и **EqualTo**, чтобы проверить, совпадает ли значение этого поля с значением поля **password**;

- поле **submit** типа **SubmitField**, которое представляет кнопку отправки формы с надписью "Переустановить пароль".

5.3 Реализация функции для добавления фото (аватарки) пользователя.

В пакете users приложения **blog\_app** создаем файл **utils.py** в котором реализуем код добавления фото к профилю пользователя.

5.4 Определение маршрутов приложения для взаимодействия с пользователями

5.4.1. Создание макета приложения (blueprint) для работы с пользователями и импорт необходимых модулей.

Как и в случае с главной страницей приложения создадим независимый модуль приложения для работы с пользователем с помощью макета blueprint. Создадим в приложении пакет **users** в который добавим модуль **routes.py** (файл маршрутизации). В нем создадим макет приложения **users** для взаимодействия с пользователем (рис. ХХХ)

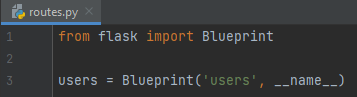


Рисунок ХХХ – Модуль **routes.py**

Импортируем **Blueprint** из модуля **flask**. Создаем объект **users** с использованием класса **Blueprint**, которому передаем имя **users** и атрибут \_\_**name\_\_**.

На рисунке ХХХ показана регистрация приложение **users** для главной страницы в файле **\_\_init\_\_.py** нашего приложения

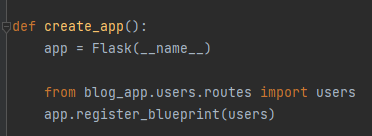


Рисунок ХХХ - Регистрация приложения **users** в файле **\_\_init\_\_.py**

Для описания функций-маршрутов приложения импортируем в файл **routes.py** необходимые модули (рис. ХХХ)

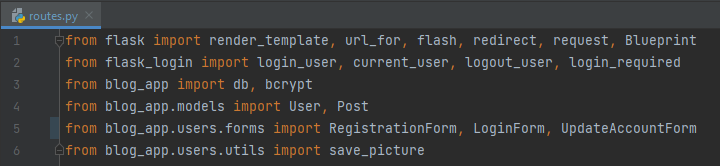


Рисунок ХХХ -Импорт модулей в файл **routes.py**

Из модуля **flask** импортируем функцию **render\_template** для рендеринга html-шаблонов, функцию **url\_for** для построения URL-адреса на основе имени функции представления, **flash** – функцию для показа сообщений пользователю, функцию **redirect** для перенаправления пользователя на другую страницу, функция **request** для доступа к запросам пользователя. Из модуля **flask\_ogin** импортируем функцию **login\_user** для выполнения входа пользователя, функцию **current\_user** для доступа к объекту текущего пользователя, функцию **logout\_user** из модуля Flask-Login для выхода пользователя, декоратор **login\_required** который требует аутентификацию пользователя для доступа к определенным представлениям. Импорт объектов **db** (экземпляр SQLAlchemy) и **bcrypt** (хеширование паролей), которые определены внутри приложения в файле **\_\_init\_\_.py**. Импорт моделей данных **User** и **Post** из файла **models.py**, которые представляют таблицы базы данных пользователя и постов в приложении. Импорт форм для регистрации пользователя (**RegistrationForm**), авторизации пользователя (**LoginForm**) и формы для изменения данных профиля пользователя (**UpdateAccountForm**). Импорт функции **save\_picture**, определенной внутри пакета **users.utils**, используемой для сохранения загруженного изображения.

5.4.2 Реализация функции регистрации нового пользователя

Для выполнения регистрации нового пользователя в файл **routes.py** реализуем функцию **register** (рис.ХХХ)которая определяет представление и будет обеспечивает обработку регистрации пользователей, включая взаимодействие с формой, хеширование пароля, добавление пользователя в базу данных, сообщения об успешной регистрации и перенаправление на соответствующие страницы.

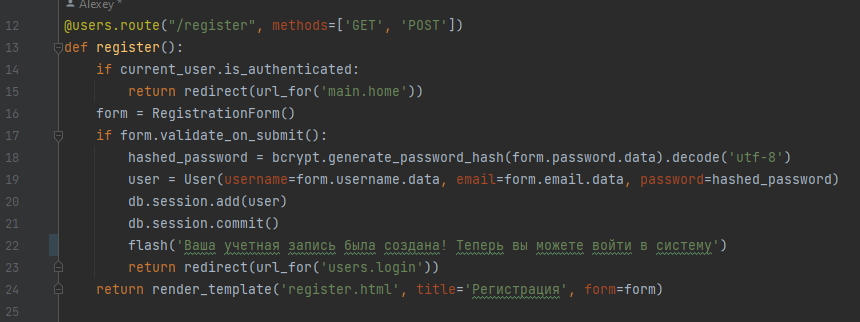


Рисунок ХХХ - Функции регистрации нового пользователя

Декоратор **@users.route("/register", methods=['GET', 'POST'])** определяет, что будем работать с пространством маршрутов нашего текущего «блюпринта» **users**. Функция **register** будет обрабатывать запросы на URL "/register" и поддерживать методы запросов GET и POST. Внутри функции проверяем, аутентифицирован ли текущий пользователь **if current\_user.is\_authenticated** Если пользователь уже аутентифицирован (вошел в систему), он будет перенаправлен на домашнюю страницу (main.). Если нет, то создаем экземпляр формы **form = RegistrationForm()** используемой для регистрации нового пользователя. При отправке формы (метод POST) и ее успешной валидации (**if form.validate\_on\_submit())** выполняется следующий код: пароль пользователя хешируется с использованием **bcrypt.generate\_password\_hash**, создаем новый объект пользователя **user** на основе данных введенных пользователем в форму, добавление нового пользователя в сессию базы данных **db.session.add(user)**, фиксируем сессию **db.session.commit()**, отображается сообщение об успешном создании учетной записи с помощью функции **flash**, перенаправление пользователя на страницу входа после успешной регистрации **redirect(url\_for('users.login')**. В противном случае, если форма не была отправлена или не прошла валидацию, будет отображена страница **register.html (рисХХХ)** передав объект формы в качестве аргумента, а также указаны заголовок страницы ("Регистрация"). Код шаблона для регистрации нового пользователя размещен в приложении 2.

Таким образом, функция **register** обеспечивает обработку регистрации пользователей, включая взаимодействие с формой, хеширование пароля, добавление пользователя в базу данных, сообщения об успешной регистрации и перенаправление на соответствующие страницы.

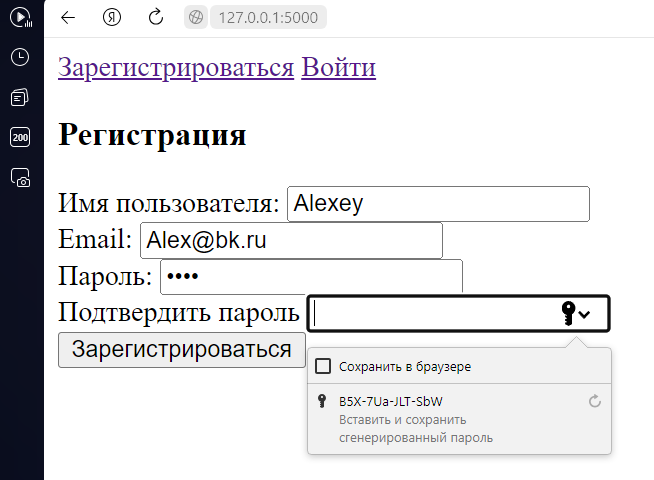


Рисунок ХХХ – Страница регистрации нового пользователя в браузере

5.4.3 Реализация функции авторизации пользователя

Пройдя процедуру регистрации нового пользователя требуется процедура авторизации для чего в файле **router.py** реализуем функцию представления **login** (рисХХХ)

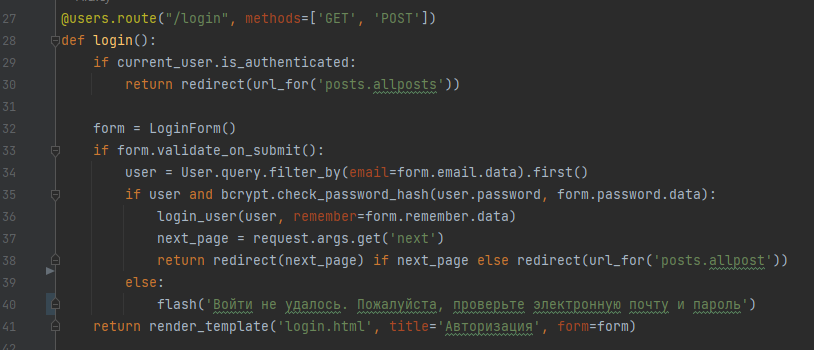


Рисунок ХХХ – Функции авторизации пользователя в браузере

Декоратор **@users.route("/login", methods=['GET', 'POST'])** определяет, что будем работать с пространством маршрутов нашего текущего «блюпринта» **users**. Функция **login** будет обрабатывать запросы на URL "/login" и поддерживать методы GET и POST запросов. Внутри функции первым условием **if current\_user.is\_authenticated** проверяем, аутентифицирован ли текущий пользователь. Если пользователь уже аутентифицирован (вошел в систему), он будет перенаправлен на страницу списка всех постов. Если нет, то создается экземпляр **form = LoginForm()** используемый для входа пользователя. Если условие **if form.validate\_on\_submit()** была ли форма отправлена и прошла валидацию верно, то будет выполнен код внутри этого блока: производится поиск пользователя в базе данных по введенной электронной почте **User.query.filter\_by(email=form.email.data).first()**, производится проверка найден ли пользователь и совпадает ли хэш его пароля с введенным пользователем паролем. Если условие проверки **if user and bcrypt.check\_password\_hash(user.password, form.password.data)** верно, то выполняется вход пользователя в систему с помощью функции **login\_user(user, remember=form.remember.data)** которая также производит установку специальной сессионной куки, для автоматической аутентификации пользователя при последующих запросах. Затем, если параметр next получен из GET запроса, происходит перенаправление пользователя на предыдущую страницу, на которой он пытался залогиниться, либо на страницу со всеми постами блога **posts.allposts**. Если предыдущее условие не верно, то появляется flash сообщение о неудачной попытке входа. Если условие аутентификации пользователя и условие валидации и отправки формы не выполнились, будет выполнен переход на страницу **login.html** (рис.ХХХ) с заголовком "Авторизация". Код шаблона в файле login.html для авторизации пользователя размещен в приложении 3

Таким образом, эта функция обеспечивает обработку входа пользователя в систему, включая взаимодействие с формой, аутентификацию пользователя, управление сеансом и перенаправление на соответствующие страницы в зависимости от результата аутентификации.

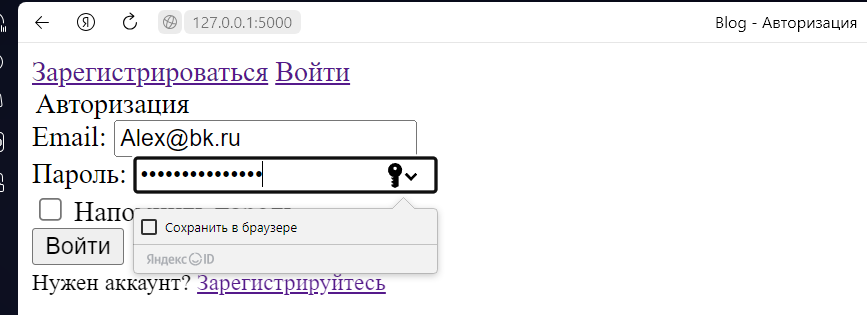


Рисунок ХХХ – Страница авторизации пользователя в браузере

5.4.4 Реализация профиля пользователя

Создадим личный кабинет пользователя, где будет реализуем возможность редактирования его данных после авторизации. Для этого в файле **router.py** реализуем функцию представления **account** (рисХХХ). Личный кабинет содержит имя пользователя, адрес электронной почты и при желании фото (аватарка) пользователя.(рис ХХХ)

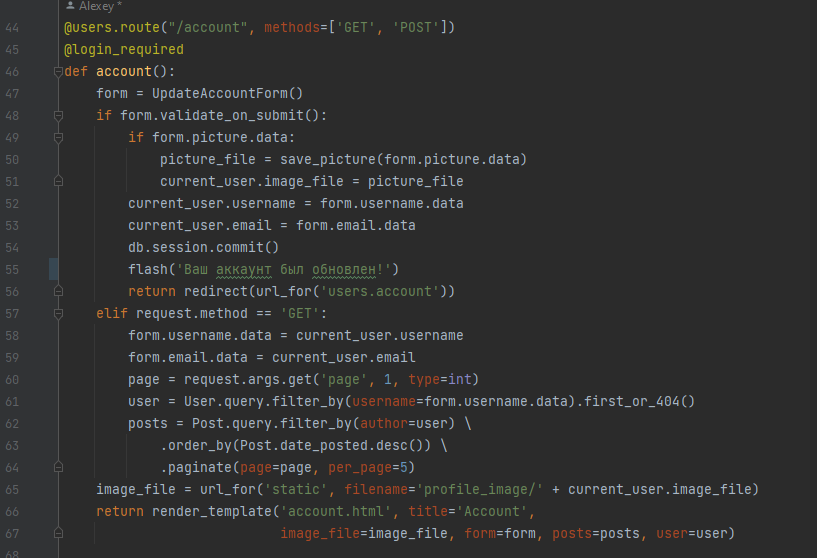


Рисунок ХХХ – Функция обновления данных пользователя

Используем декоратор **@users.route("/account", methods=['GET', 'POST'])** для маршрутизации GET POST запросов к URL-адресу **/account**. Обращение к этому URL будет вызывать функцию **account**. Декоратор **@login\_required** выполняет функцию проверки на аутентификацию пользователя, прежде чем получить доступ к адресу **/account**. Создаем экземпляр формы **form = UpdateAccountForm()** используемый для работы с учетной записью пользователя. Если условие **if form.validate\_on\_submit()** была ли форма отправлена и прошла валидацию верно, то будет выполнен код внутри этого блока: **if form.picture.data** – условие проверки передачи файла изображения в форму. Если файл загружен то происходит сохранение этого изображения **picture\_file = save\_picture(form.picture.data)** и путь к нему записывается в учетную запись текущего пользователя **current\_user.image\_file = picture\_file**. Обновляем имя для текущего пользователя на значение, введенное в форму **current\_user.username = form.username.data** и адрес электронной почты пользователя на значение, введенное в форму **current\_user.email = form.email.data**. Сохраняем изменения в базе данных **db.session.commit()**. Добавляем flash-сообщение об успешном обновлении аккаунта и перенаправление пользователя **return redirect(url\_for('users.account'))** на страницу с адресом **/account**.

При GET-запросе **elif request.method == 'GET',**  т.е. когда мы обращаемся к нашей форме личного кабинета, выполняются следующий код: заполнение поля **username** формы значением текущего имени пользователя **form.username.data = current\_user.username**, заполнение поля **email** формы значением текущего адреса электронной почты пользователя **form.email.data = current\_user.email**. Указываем пагинацию, т.е. получение параметра **page** из GET-запроса. Если он не указан, присваивается значение 1 **page = request.args.get('page', 1, type=int)** для вывода постов текущего пользователя. Извлекаем пользователя из базы данных с именем, указанным в форме **user = User.query.filter\_by(username=form.username.data).first\_or\_404().** Если пользователя нет, будет выдана ошибка 404. Извлекаем посты пользователя из базы данных **posts = Post.query.filter\_by(author=user).order\_by(Post.date\_posted.desc()).paginate(page=page, per\_page=5)** и сортируем их по дате публикации в порядке убывания и разбиением по страницам.

Также создаем объект **image\_file = url\_for('static', filename='profile\_image/' + current\_user.image\_file)** в котором храниться путь к файлу изображения пользователя. Далее происходить отрисовка HTML-шаблона **"account.html"** (рис. ХХХ) с передачей заголовка страницы **'Account'**, пути к изображению профиля, объекта формы **form**, списка постов и объекта пользователя. Код шаблона в файле account.html для изменения данных пользователя размещен в приложении 4

Таким образом, эта функция обрабатывает обновление учетной записи пользователя, включая загрузку изображения профиля, изменение имени пользователя и адреса электронной почты, сохранение обновленных данных в базе данных, отображение сообщений об успешном обновлении учетной записи и отрисовку страницы учетной записи.

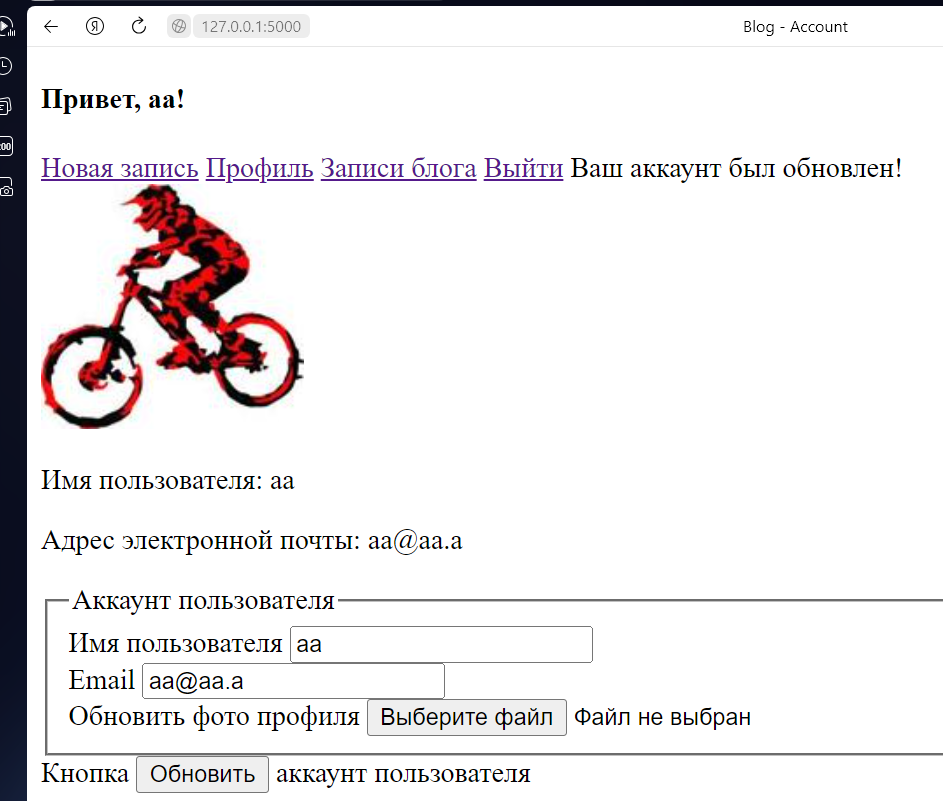


Рисунок ХХХ - – Страница личного кабинета пользователя в браузере