

Знакомство с flask-sqlalchemy

- 1 Базы данных и ORM
- 2 Установка
- 3 Постановка задачи для первого примера
- 4 Начальный каркас приложения
- 5 Таблица с пользователями
- 6 Таблица с новостями
- 7 Взаимодействие с базой данных
- 8 Небольшая шпаргалка по filter
- 9 Отображение публичных записей
- 10 Регистрация пользователей

Аннотация

На этом и следующем занятиях мы создадим полноценное веб-приложение с регистрацией и авторизацией пользователей, работающее с базой данных через ORM.

1. Базы данных и ORM

Мы уже достаточно тесно познакомились с реляционными базами данных: решали задачи с использованием БД, многие из вас уже использовали БД в своих проектах для хранения информации. Однако пока мы работали с данными исключительно с помощью запросов на языке SQL, которые составляли самостоятельно. Для небольших проектов это вполне оправдано и достаточно часто используется в реальной жизни. Flask «из коробки» не имеет собственных инструментов для работы с базами данных (в отличие, например, от Django) и предоставляет нам полную свободу выбора инструментов для работы.

Согласитесь: очень хочется перейти на более высокий уровень работы с данными. Представьте, что в вашем распоряжении есть объект, который связан с базой данных. Этот объект берет на себя всю работу

по организации общения с данными. Вам лишь остается давать ему команды: получить данные, отфильтровать их по заданному условию, записать данные и т. д., а преобразование команд в SQL-запросы — это уже забота объекта. Работать с данными как с объектами чрезвычайно удобно — не зря же мы так много времени уделяем изучению объектного подхода.

К сожалению, написать универсальный модуль, который преобразовывает результат SQL-запроса в объекты и наоборот, достаточно нетривиальная задача. Хорошо, что она уже решена за нас.

В больших приложениях (не только для web) достаточно часто используется технология **ORM** (Object-Relational Mapping — объектно-реляционное отображение) — прослойка, позволяющая работать с базой данных через объекты языка. Кроме того, большинство ORM позволяют генерировать скрипты миграции базы данных для поддержания версионности (отдаленно можно сравнить с git, но для баз данных), а также предоставляют разработчику еще немало другой полезной функциональности. Мы будем использовать библиотеку sqlalchemy. Ее можно использовать не только при создании веб-приложений, но и при разработке любых программ, которые взаимодействуют с базами данных.

2. Установка

Прежде чем мы начнем что-то делать, давайте установим sqlalchemy:

```
pip install sqlalchemy
```

3. Постановка задачи для первого примера

Давайте создадим веб-приложение, которое будет использовать базу данных и ORM для работы с ней. Для начала определимся с функциональностью.

У нас будет простое веб-приложение, в котором пользователи могут авторизовываться, просматривать, добавлять и удалять новости — такой мини-вариант личных блогов. Получается, нам надо хранить информацию о двух сущностях:

1. Пользователи
2. Новости

Причем каждая новость должна быть связана с автором новости, который ее написал.

4. Начальный каркас приложения

Начнем создавать наше приложение с создания основного файла, который назовем main.py. Для начала разместим в нем уже привычный код:

```
from flask import Flask

app = Flask(__name__)
app.config['SECRET_KEY'] = 'yandexlyceum_secret_key'
```

```
def main():
    app.run()

if __name__ == '__main__':
    main()
```

Создадим папки:

- db — для хранения одного единственного файла базы данных
- data — для хранения классов, необходимых для взаимодействия с базой данных

Внутри папки data создадим два файла, один назовем `__all_models.py` (в нем будем хранить модели для работы с базой данных), а — другой `db_session.py` — будет отвечать за подключение к базе данных и создание сессии для работы с ней.

Следующий шаг самый сложный. Не переживайте, этот код будет стандартным для всех ваших приложений и может переезжать от одного проекта к другому. Напишем в `db_session.py` вот такой код:

```
import sqlalchemy as sa
import sqlalchemy.orm as orm
from sqlalchemy.orm import Session
import sqlalchemy.ext.declarative as dec

SqlAlchemyBase = dec.declarative_base()

__factory = None
```

Сначала импортируем необходимое — саму библиотеку `sqlalchemy` (в принципе, этого достаточно, остальные импорты нужны только для избавления от длинных путей), затем часть библиотеки, которая отвечает за функциональность ORM, потом объект `Session`, отвечающий за соединение с базой данных, и модуль `declarative` — он поможет нам объявить нашу базу данных.

Создадим две переменные: `SqlAlchemyBase` — некоторую абстрактную декларативную базу, в которую позднее будем наследовать все наши модели, и `__factory`, которую будем использовать для получения сессий подключения к нашей базе данных.

Кроме того, в файле `db_session.py` нам понадобится сделать еще две функции `global_init` и `create_session`.

```
def global_init(db_file):
    global __factory

    if __factory:
        return

    if not db_file or not db_file.strip():
        raise Exception("Необходимо указать файл базы данных.")

    conn_str = f'sqlite:/// {db_file.strip()}?check_same_thread=False'
```

```

print(f"Подключение к базе данных по адресу {conn_str}")

engine = sa.create_engine(conn_str, echo=False)
__factory = orm.sessionmaker(bind=engine)

from . import __all_models

SqlAlchemyBase.metadata.create_all(engine)

```

`global_init` принимает на вход адрес базы данных, затем проверяет, не создали ли мы уже фабрику подключений (то есть не вызываем ли мы функцию не первый раз). Если уже создали, то завершаем работу, так как начальную инициализацию надо проводить только единожды.

Проверяем, что нам указали непустой адрес базы данных, а затем создаем строку подключения `conn_str` (она состоит из типа базы данных, адреса до базы данных и параметров подключения), которую передаем SQLAlchemy для того, чтобы она выбрала правильный движок работы с базой данных (переменная `engine`). В нашем случае это будет движок для работы с SQLite базами данных.

Если в функцию `create_engine()` передать параметр `echo` со значением `True`, в консоль будут выводиться все SQL-запросы, которые сделает SQLAlchemy, что очень удобно для отладки.

Наконец, создаем фабрику подключений к нашей базе данных, которая будет работать с нужным нам движком.

Импортируем все из файла `__all_models.py` — именно тут SQLAlchemy узнает о всех наших моделях.

Наконец, заставляем нашу базу данных создать все объекты, которые она пока не создала. Обратите внимание: все таблицы, которые были уже созданы в базе данных, останутся без изменений.

```

def create_session() -> Session:
    global __factory
    return __factory()

```

Функция `create_session` нужна для получения сессии подключения к нашей базе данных. Часть `-> Session` нужна лишь для того, чтобы явно указать PyCharm, что наша функция возвращает объект типа `sqlalchemy.orm.Session` и среда могла показывать нам подсказки далее.

В качестве завершающего штриха давайте добавим в `main.py` импорт содержимого файла `db_session`:

```

from data import db_session

```

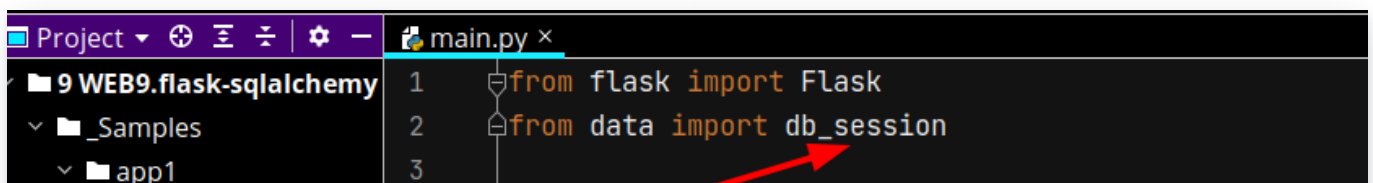
И перед запуском приложения `app.run()` добавим вызов глобальной инициализации всего, что связано с базой данных:

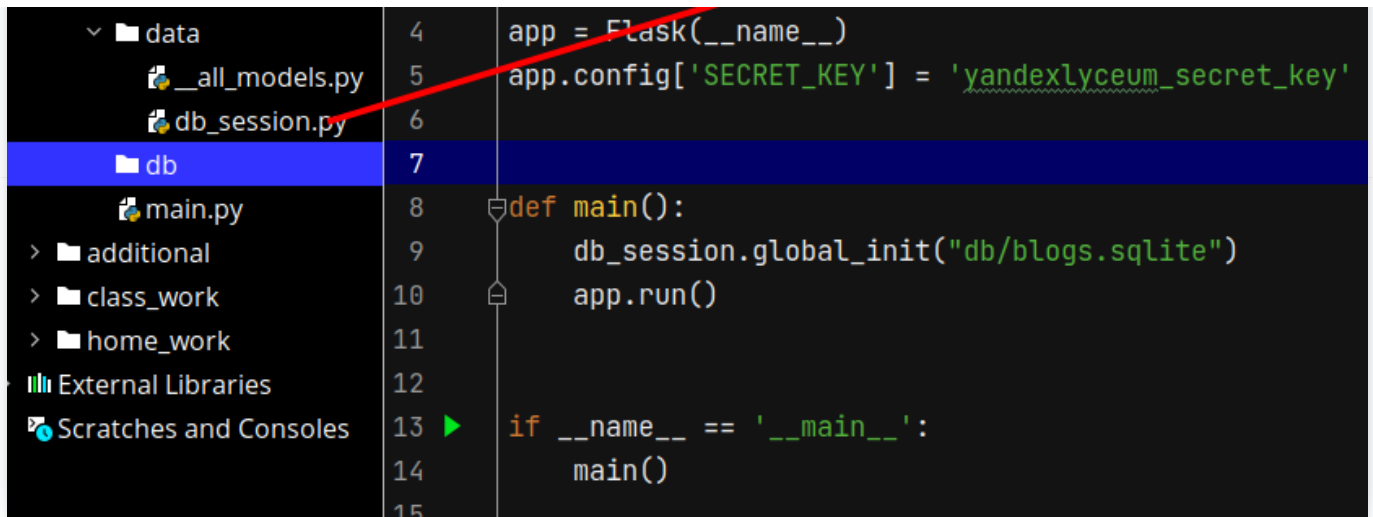
```

db_session.global_init("db/blogs.db")

```

В результате должна получиться примерно такая структура папок и файлов:





```
4 app = Flask(__name__)
5 app.config['SECRET_KEY'] = 'yandexlyceum_secret_key'
6
7
8 def main():
9     db_session.global_init("db/blogs.sqlite")
10    app.run()
11
12
13 if __name__ == '__main__':
14     main()
15
```

Запустим наше приложение. Мы увидим, что оно работает и после запуска создало пустую базу данных по адресу db/blogs.db.

Теперь давайте создадим таблицы.

5. Таблица с пользователями

Так как мы используем ORM, никакого SQL мы писать не будем и позволим sqlalchemy сделать всю работу за нас, а сами лишь опишем те классы, которые хотим получить для работы с базой в Python.

Давайте создадим в папке data файл users.py и опишем в нем класс `User` — модели для работы с таблицей, которая будет содержать информацию о пользователях нашего веб-приложения.

Подумаем, какая информация нам нужна о наших пользователях:

1. Уникальный идентификатор пользователя — некоторый его номер в нашей базе данных, желательно, чтобы он генерировался автоматически без нашего участия.
2. Имя пользователя — некоторое строковое значение.
3. Описание пользователя — некоторая текстовая информация, которой пользователь хочет поделиться о себе.
4. Адрес электронной почты — уникальная строка (чтобы мы могли точно знать, для какого пользователя восстанавливать пароль). Так как мы достаточно часто будем искать пользователя по адресу электронной почты (как минимум, при логине), было бы здорово, если бы база могла как-то ускорить такой поиск.
5. Зашифрованный пароль пользователя — строка. **Ни в коем случае нельзя хранить пароль пользователя в открытом виде!**
6. Дата создания пользователя — потому что нам хочется знать, когда пользователь зарегистрировался в нашем веб-приложении.

```
import datetime
import sqlalchemy
from .db_session import SQLAlchemyBase

class User(SQLAlchemyBase):
    __tablename__ = 'users'
```

```

id = sqlalchemy.Column(sqlalchemy.Integer,
                        primary_key=True, autoincrement=True)
name = sqlalchemy.Column(sqlalchemy.String, nullable=True)
about = sqlalchemy.Column(sqlalchemy.String, nullable=True)
email = sqlalchemy.Column(sqlalchemy.String,
                          index=True, unique=True, nullable=True)
hashed_password = sqlalchemy.Column(sqlalchemy.String, nullable=True)
created_date = sqlalchemy.Column(sqlalchemy.DateTime,
                                 default=datetime.datetime.now)

```

Чтобы обозначить, что `User` — не обычный класс, а именно класс модели, его необходимо унаследовать от объекта класса `SqlAlchemyBase`, который мы создали ранее.

В служебном атрибуте `__tablename__` указывается таблица, которая будет создана для хранения данных этой модели. Название таблицы можно не указывать, тогда `sqlalchemy` сделает таблицу сама, исходя из названия класса.

Для каждого из атрибутов типа `sqlalchemy.Column` будет создан одноименный столбец в базе данных согласно его описанию:

- `sqlalchemy.Integer` (`sqlalchemy.String`, `sqlalchemy.DateTime` и т. д.) — указания типа данных
- `primary_key=True` — указание на то, что столбец является первичным ключом. Обычно первичный ключ — некоторый числовой идентификатор, который однозначно идентифицирует каждую запись в таблице
- `autoincrement=True` — признак автоинкрементного поля. Используется для увеличения значения первичного ключа на единицу при вставке каждой новой записи
- `nullable=True/False` — может ли поле не содержать никакой информации и быть пустым
- `unique=True/False` — содержит ли поле только уникальные значения или они могут повторяться
- `default=datetime.datetime.now` — значение по умолчанию. В данном случае мы говорим, что при вставке нового пользователя будет вставлена дата и время на момент его создания. Обратите внимание: мы не вызываем функцию, а передаем ее. Если бы мы вызвали функцию, то текущее время было бы вычислено только один раз при запуске сервера и было бы одинаковое для всех пользователей, которые были бы созданы после этого
- `index=True` — создать индекс по этому полю. **Индекс**, если говорить упрощенно, позволяет значительно повысить скорость поиска по одному или нескольким полям базы данных. Цена этого — уменьшение скорости вставки данных, поэтому не стоит делать индексы на абсолютно все поля, а только на те (и ту их комбинацию), по которым будет часто осуществляться поиск

Запустим наше приложение. Но почему ничего не произошло? Почему не создалась таблица?

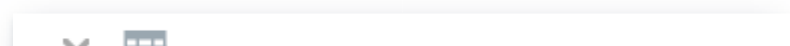
На самом деле все просто. Наша база пока не знает про нашу модель. Давайте добавим информацию о ней в файл `__all_models.py`.

```

from . import users

```

Запустим еще раз.



users	
id	INTEGER
name	VARCHAR
about	VARCHAR
email	VARCHAR
hashed_password	VARCHAR
created_date	DATETIME
key #1	(id)
ix_users_email	(email) UNIQUE

Ура, все заработало!

6. Таблица с новостями

По аналогии рядом с файлом `users.py` создадим файл `news.py`, а в нем — класс `News` для хранения следующей информации о новости:

- Уникальный идентификатор
- Заголовок новости
- Текст новости
- Дата создания новости
- Приватность новости (показывать ли ее всем или только автору)
- Идентификатор автора новости

```
import datetime
import sqlalchemy
from sqlalchemy import orm

from .db_session import SQLAlchemyBase

class News(SQLAlchemyBase):
    __tablename__ = 'news'

    id = sqlalchemy.Column(sqlalchemy.Integer,
                           primary_key=True, autoincrement=True)
    title = sqlalchemy.Column(sqlalchemy.String, nullable=True)
    content = sqlalchemy.Column(sqlalchemy.String, nullable=True)
    created_date = sqlalchemy.Column(sqlalchemy.DateTime,
                                     default=datetime.datetime.now)
    is_private = sqlalchemy.Column(sqlalchemy.Boolean, default=True)

    user_id = sqlalchemy.Column(sqlalchemy.Integer,
```

```
sqlalchemy.ForeignKey("users.id"))
user = orm.relationship('User')
```

Здесь стоит обратить внимание на поля `user_id` и `user`.

`user_id` — колонка, в которой указывается, что она ссылается на поле `id` таблицы `users`. А `user` — атрибут, который позволит нам получить для новости полноценный объект класса `User`.

Немного модифицируем наш класс с пользователями, добавим туда строчку:

```
news = orm.relationship("News", back_populates='user')
```

И в блок импортов строчку:

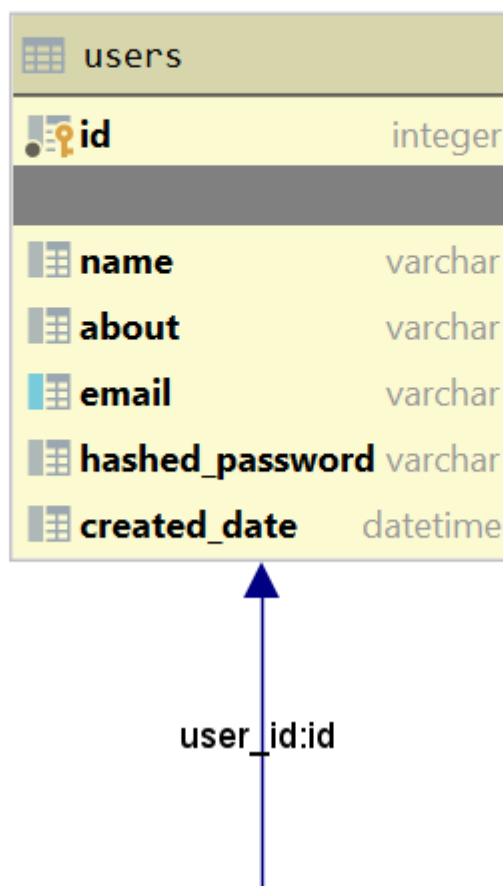
```
from sqlalchemy import orm
```







Это позволит нам легко получать для пользователя все его новости. Обратите внимание: значение параметра `back_populates` должно указывать не на таблицу, а на атрибут класса `orm.relationship`.

Чтобы подключить модель `News` к нашей базе данных, в файл `__all_models.py` надо добавить строчку:

```
from . import news
```

Удалим базу данных и запустим наше приложение заново, чтобы все таблицы пересоздались. Вот какие две таблицы у нас получились в базе данных:



news	
 id	integer
 title	varchar
 content	varchar
 created_date	datetime
 is_private	boolean
 user_id	integer

7. Взаимодействие с базой данных

Давайте ненадолго сместим фокус от создания веб-приложения на взаимодействие с базой данных с помощью sqlalchemy.

Можно пока временно закомментировать строку `app.run()` в файле `main.py`.

Давайте напишем код, который создает пользователя в нашей базе данных.

Добавьте в секцию импортов строку







```
from data.users import User
```

чтобы можно было использовать класс `User`.

За добавление объектов отвечает метод `add` у объекта `session`.

```
user = User()
user.name = "Пользователь 1"
user.about = "биография пользователя 1"
user.email = "email@email.ru"
db_sess = db_session.create_session()
db_sess.add(user)
db_sess.commit()
```

Создадим еще нескольких пользователей, чтобы данные в таблице выглядели примерно вот так:

	 id	 name	 about	 email	 hashed_password	 created_date
1	1	Пользователь 1	биография пользователя 1	email@email.ru	<null>	2020-01-02 15:00:52.273
2	2	Пользователь 2	биография пользователя 2	email2@email.ru	<null>	2020-01-02 15:01:52.273892
3	3	Пользователь 3	биография пользователя 3	email1@email.ru	<null>	2020-01-02 15:02:41.110581

За получение данных отвечает метод `query` объекта `session`. В качестве параметра передаются классы

объектов, которые мы хотим достать. Например, давайте достанем первого пользователя в выборке:

```
user = db_sess.query(User).first()
print(user.name)
```

Пользователь 1

А теперь пройдемся вообще по всем пользователям в таблице. Для более красивого представления переопределим метод `__repr__` у класса `User`:

```
for user in db_sess.query(User).all():
    print(user)
```

```
<User> 1 Пользователь 1 email@email.ru
<User> 2 Пользователь 2 email2@email.ru
<User> 3 Пользователь 3 email1@email.ru
```

Метод `filter()` позволяет отфильтровать результаты с помощью оператора `WHERE`, примененного к запросу. Он принимает колонку, оператор или значение. Например, давайте отфильтруем пользователей и выберем только тех, у которых `id > 1`, а почта не содержит 1.

```
for user in db_sess.query(User).filter(User.id > 1, User.email.notlike("%1%")):
    print(user)
```

```
<User> 2 Пользователь 2 email2@email.ru
```

Условия, перечисленные в скобках `filter` через запятую, соединяются в запросе через `AND`. Давайте изменим запрос, чтобы условия фильтра соединялись через `OR`:

```
for user in db_sess.query(User).filter((User.id > 1) | (User.email.notlike("%1%"))):
    print(user)
```

Обратите внимание: в этом случае скобки вокруг частей фильтра **обязательны**.

Разумеется, `sqlalchemy` позволяет нам изменять и удалять записи в таблицах. Изменение записи сделать очень просто, надо выбрать нужную запись, поменять нужные атрибуты, а потом вызвать у сессии метод `commit`.

Переименуем пользователя с `id 1`:

```
user = db_sess.query(User).filter(User.id == 1).first()
print(user)
user.name = "Измененное имя пользователя"
user.created_date = datetime.datetime.now()
db_sess.commit()
```

	id	name	about	email	hashed_password	created_date
1	1	Измененное имя пользователя	биография пользователя 1	email@email.ru	<null>	2020-01-03 12:12:30.592533

За удаление записей отвечает метод `delete()`.

Мы можем удалить как несколько записей по фильтру:

```
db_sess.query(User).filter(User.id >= 3).delete()  
db_sess.commit()
```

Так и какую-то уже заранее выбранную запись:

```
user = db_sess.query(User).filter(User.id == 2).first()  
db_sess.delete(user)  
db_sess.commit()
```

После всех изменений в базе данных не забывайте сделать `commit`.

После всех этих операций у нас должен остаться в базе всего один пользователь. Давайте добавим ему записи. Это можно сделать несколькими способами. Мы можем создать объект класса `News` и заполнить его поля, в том числе указать явно `id` записи автора:

```
news = News(title="Первая новость", content="Привет блог!",  
            user_id=1, is_private=False)  
db_sess.add(news)  
db_sess.commit()
```

Можем в качестве `user` указать объект класса `User`, выбранный (или созданный) заранее:

```
user = db_sess.query(User).filter(User.id == 1).first()  
news = News(title="Вторая новость", content="Уже вторая запись!",  
            user=user, is_private=False)  
db_sess.add(news)  
db_sess.commit()
```

И самый удобный на наш взгляд вариант — использовать то, что через объект класса `User` мы можем взаимодействовать с его записями в таблице `News` почти как со списком:

```
user = db_sess.query(User).filter(User.id == 1).first()  
news = News(title="Личная запись", content="Эта запись личная",  
            is_private=True)  
user.news.append(news)  
db_sess.commit()
```

Так же легко мы можем обойти все новости конкретного пользователя:

```
for news in user.news:  
    print(news)
```

`sqlalchemy` — невероятно мощный и гибкий инструмент. Обязательно загляните в **документацию**, потому что мы рассмотрели только самый минимум, необходимый для создания небольших веб-приложений и учебных проектов.

8. Небольшая шпаргалка по filter

Операция	Синтаксис ORM
EQUALS	query(User).filter(User.name == 'Иван')
NOT EQUAL	query(User).filter(User.name != 'Иван')
LIKE	query(User).filter(User.name.like('%Иван%'))
NOT LIKE	query(User).filter(User.name.notlike('%Иван%'))
IN	query(User).filter(User.name.in_(['Иван', 'Петр', 'Максим']))
NOT IN	query(User).filter(User.name.notin_(['Иван', 'Петр', 'Максим'])) или query(User).filter(~User.name.in_(['Иван', 'Петр', 'Максим']))
NULL	query(User).filter(User.name == None)
AND	query(User).filter(User.name == 'Иван', User.id > 3) или query(User).filter(User.name == 'Иван').filter(User.id > 3)
OR	query(User).filter((User.name == 'Иван') (User.id > 3)) или query(User).filter(or_(User.name == 'Иван', User.id > 3))

9. Отображение публичных записей

Воспользуемся наработками прошлых уроков и создадим новый шаблон index.html на основе базового шаблона base.html, который мы делали на прошлом уроке. Скопируйте папку templates со всем ее содержимым из материалов, созданных на прошлом уроке, и отредактируйте файл index.html, чтобы он выглядел так:

```
{% extends "base.html" %}

{% block content %}
<h1>Записи в блоге</h1>
{% for item in news%}
<div class="col-md6 border rounded">
    <h2>{{item.title}}</h2>
    <div>
        {{item.content}}
    </div>
    <div>
        Автор - {{item.user.name}}, Дата написания - {{item.created_date}}
    </div>
</div>
{% endfor %}
{% endblock %}
```

Обратите внимание: из каждой записи мы можем легко добраться до имени автора этой записи. А теперь добавим обработчик, чтобы при попадании на главную страницу нашего приложения все пользователи видели

все публичные записи от всех авторов.

```
@app.route("/")
def index():
    db_sess = db_session.create_session()
    news = db_sess.query(News).filter(News.is_private != True)
    return render_template("index.html", news=news)
```

Наше приложение

Записи в блоге

Первая новость

Привет блог!

Автор - Измененное имя пользователя, Дата написания - 2020-01-03 12:33:39.329514

Вторая новость

Уже вторая запись!

Автор - Измененное имя пользователя, Дата написания - 2020-01-03 12:40:01.685179

10. Регистрация пользователей

Редко какое веб-приложение обходится без функциональности, доступной лишь зарегистрированным пользователям. Давайте добавим возможность пользователям регистрироваться в нашем приложении. Все классы для форм разместим в отдельном каталоге forms. Создадим в каталоге forms файл user.py, а в нем — класс, описывающий форму RegisterForm, в которую добавим все поля, которые будут в форме регистрации:

```
from flask_wtf import FlaskForm
from wtforms import PasswordField, StringField, TextAreaField, SubmitField, EmailField
from wtforms.validators import DataRequired

class RegisterForm(FlaskForm):
    email = EmailField('Почта', validators=[DataRequired()])
    password = PasswordField('Пароль', validators=[DataRequired()])
    password_again = PasswordField('Повторите пароль', validators=[DataRequired()])
    name = StringField('Имя пользователя', validators=[DataRequired()])
    about = TextAreaField("Немного о себе")
    submit = SubmitField('Войти')
```

Спросим пароль у пользователя несколько раз, чтобы убедиться, что он нигде не опечатался.

Теперь в папке templates создадим шаблон register.html для отображения нашей формы, он может выглядеть так:

```
{% extends "base.html" %}

{% block content %}
<h1>Регистрация</h1>
<form action="" method="post">
    {{ form.hidden_tag() }}
    <p>
        {{ form.email.label }}<br>
        {{ form.email(class="form-control", type="email") }}<br>
        {% for error in form.email.errors %}
            <p class="alert alert-danger" role="alert">
                {{ error }}
            </p>
        {% endfor %}
    </p>
    <p>
        {{ form.password.label }}<br>
        {{ form.password(class="form-control", type="password") }}<br>
        {% for error in form.password.errors %}
            <p class="alert alert-danger" role="alert">
                {{ error }}
            </p>
        {% endfor %}
    </p>
    <p>
        {{ form.password_again.label }}<br>
        {{ form.password_again(class="form-control", type="password") }}<br>
        {% for error in form.password_again.errors %}
            <p class="alert alert-danger" role="alert">
                {{ error }}
            </p>
        {% endfor %}
    </p>
    <p>
        {{ form.name.label }}<br>
        {{ form.name(class="form-control") }}<br>
        {% for error in form.name.errors %}
            <p class="alert alert-danger" role="alert">
                {{ error }}
            </p>
        {% endfor %}
    </p>
    <p>
        {{ form.about.label }}<br>
        {{ form.about(class="form-control") }}<br>
        {% for error in form.about.errors %}
            <p class="alert alert-danger" role="alert">
                {{ error }}
            </p>
        {% endfor %}
    </p>
</form>
</block>
```

```

        </p>
        {% endfor %}
    </p>
    <p>{{ form.submit(type="submit", class="btn btn-primary") }}</p>
    {{message}}
</form>
{% endblock %}

```

Прежде чем создавать обработчик URL, который будет отвечать за форму регистрации, давайте немного доработаем наш класс User. Как мы уже говорили, хранить пароль в открытом виде нельзя, поэтому во Flask есть инструменты, которые позволяют получить хешированное значение по строке и проверить, соответствует ли пароль хешу, который хранится в нашей базе данных. Функции `generate_password_hash` и `check_password_hash` хранятся в модуле `werkzeug.security`. Для большего удобства добавим в класс User две функции:

```

def set_password(self, password):
    self.hashed_password = generate_password_hash(password)

def check_password(self, password):
    return check_password_hash(self.hashed_password, password)

```

Первая устанавливает значение хэша пароля для переданной строки. А вторая проверяет, правильный ли пароль ввел пользователь. Одна понадобится нам для регистрации пользователя, а другая — позднее, когда мы будем делать авторизацию пользователей в нашем приложении.

А в раздел импортов добавим:

```

from werkzeug.security import generate_password_hash, check_password_hash

```

Добавим обработчик на адрес `/register`.

```

@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        if form.password.data != form.password_again.data:
            return render_template('register.html', title='Регистрация',
                                   form=form,
                                   message="Пароли не совпадают")
        db_sess = db_session.create_session()
        if db_sess.query(User).filter(User.email == form.email.data).first():
            return render_template('register.html', title='Регистрация',
                                   form=form,
                                   message="Такой пользователь уже есть")

        user = User(
            name=form.name.data,
            email=form.email.data,
            about=form.about.data
        )

```

```
user.set_password(form.password.data)
db_sess.add(user)
db_sess.commit()
return redirect('/login')
return render_template('register.html', title='Регистрация', form=form)
```

Не забудьте импортировать форму:

```
from forms.user import RegisterForm
```

После отправки формы на сервер проверяем, что пароли совпадают, а также что пользователя с таким адресом электронной почты пока нет в нашей базе данных.

Если все хорошо, добавляем пользователя в базу данных и отправляем его на страницу авторизации, которую сделаем на следующем уроке.

Наше приложение

Регистрация

Почта

Пароль

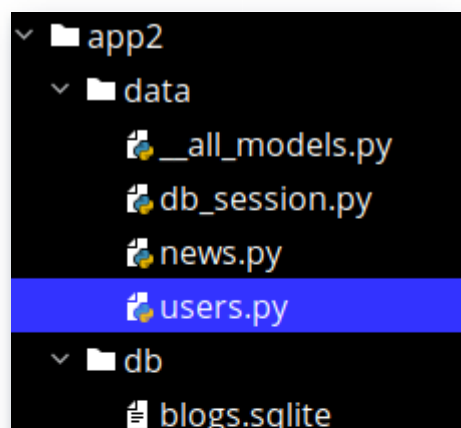
Повторите пароль

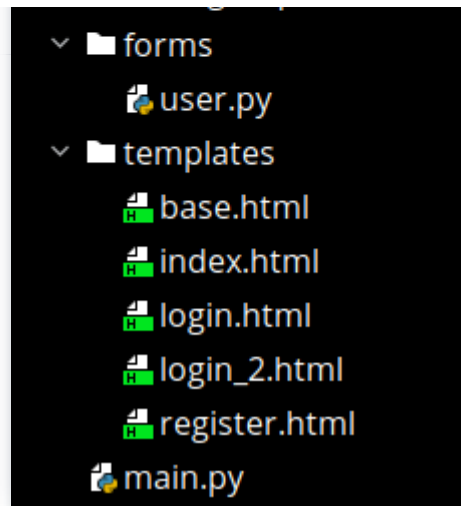
Имя пользователя

Немного о себе

Войти

После добавления всех необходимых файлов в проект он должен выглядеть так:





Заключение.

На этом уроке мы научились работать с ORM sqlalchemy и написали каркас нашего приложения, страницу отображения новостей, а также страницу регистрации пользователей. На следующем уроке мы сделаем авторизацию и ее проверку и поговорим еще о нескольких важных вещах.

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках сервиса, принадлежат АНО ДПО «Образовательные технологии Яндекса». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «Образовательные технологии Яндекса».

[Пользовательское соглашение.](#)

© 2018 – 2024 ООО «Яндекс»