Урок sqlalchemy

Flask и sqlalchemy

Аннотация

Сегодня мы доделаем процесс авторизации пользователей в веб-приложении, а также рассмотрим еще несколько важных моментов, связанных с работой с библиотеками flask и sqlalchemy.

1. Куки и сессии

Продолжим работать над веб-приложением, которое мы начали делать на прошлом уроке. Прежде чем писать форму логина, давайте поговорим о том, как работает процесс аутентификации в вебе. Как вы наверняка заметили, пока наше веб-приложение не запоминает никакой информации о клиенте между его запросами, то есть мы никак не проверяем, пришел ли следующий запрос от того же клиента или от другого. Чтобы удостовериться в том, что пользователь был на нашем сайте и уже делал какие-то действия, есть несколько способов.

Первый из них — **куки**. Куки — это небольшой фрагмент данных, который сервер устанавливает в браузере клиента. Это работает следующим образом:

1. Клиент отправляет запрос на получение страницы от сервера, то есть вызывает какой-то из наших flask-обработчиков url.
2. Сервер отвечает на запрос и вместе со страницей ответа отправляет одно или несколько куки.
3. При всех следующих запросах клиент отправляет информацию из полученных куки серверу (пока не истечет их «срок годности»).

Как и со всеми остальными параметрами, которые мы получаем от пользователя, работа с куки очень похожа

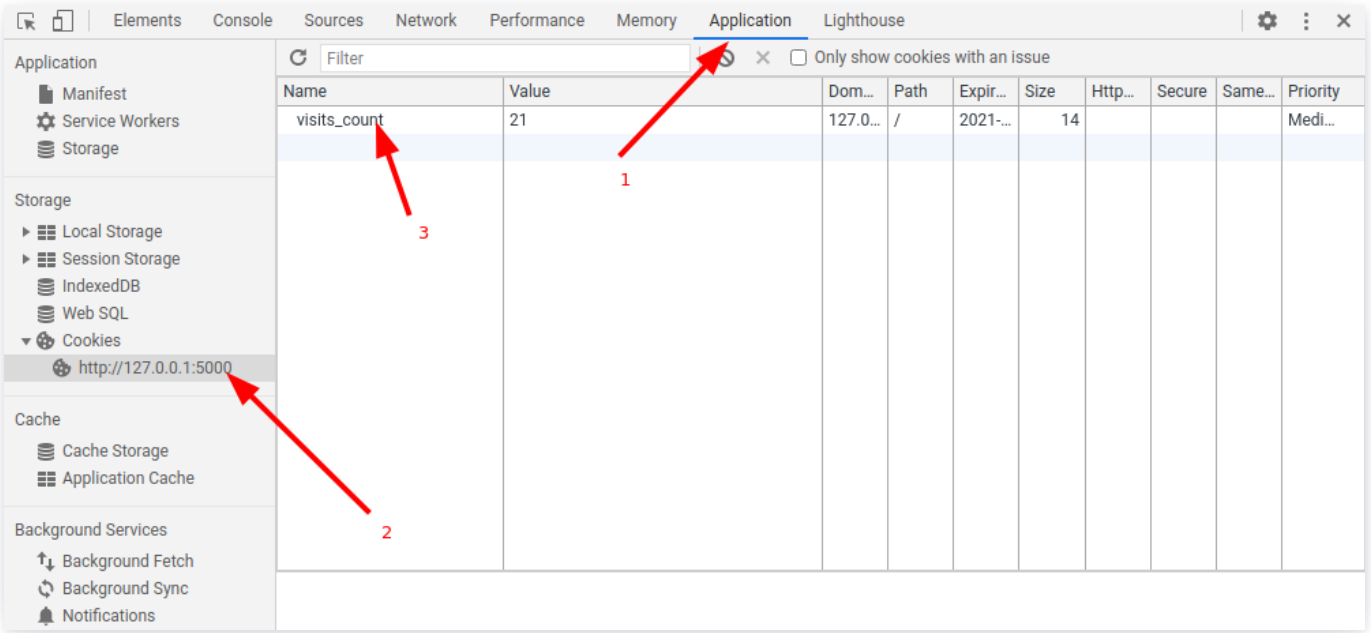
на работу со словарями. Давайте сделаем вот такой небольшой пример:

```
@app.route("/cookie_test")
def cookie_test():
    visits_count = int(request.cookies.get("visits_count", 0))
    if visits_count:
        res = make_response(
            f"Вы пришли на эту страницу {visits_count + 1} раз")
        res.set_cookie("visits_count", str(visits_count + 1),
                       max_age=60 * 60 * 24 * 365 * 2)
    else:
        res = make_response(
            "Вы пришли на эту страницу в первый раз за последние 2 года")
        res.set_cookie("visits_count", '1',
                       max_age=60 * 60 * 24 * 365 * 2)
    return res
```

Сначала мы пытаемся получить куки по ключу `visits_count`, и если она не установлена у пользователя, получаем ноль. Ноль означает, что пользователь пришел на нашу страницу первый раз, о чем мы ему и сообщаем с помощью строки, которую передаем в функцию `flask.make_response`. После чего у получившегося объекта вызываем метод `set_cookie`, куда передаем имя куки, значение, а также максимальное время жизни, после которого браузер удалит куки.

Если куки уже установлено, мы увеличиваем счетчик на 1 и переустанавливаем куки, обновляя срок жизни. Обратите внимание: у куки ключи могут быть типа `str`, а значения — типа `str` или `bytes`.

Запустите программу, перейдите по адресу `http://127.0.0.1:5000/cookie_test` и посмотрите, как будет меняться отображаемая информация в браузере, когда вы обновляете страницу (это можно делать при помощи клавиши F5). Также значение куки можно посмотреть непосредственно в браузере, перейдя в режим разработчика при помощи клавиши F12.



Если мы хотим сделать не простой текстовый ответ, необходимо передать в функцию `make_response` результат функции `render_template`.

Например, в обработчике для страницы новостей это могло выглядеть примерно вот так:

```
res = make_response(render_template("index.html", news=news))
res.set_cookie("visits_count", '1', max_age=60 * 60 * 24 * 365 * 2)
```

Для удаления куки достаточно установить для нее нулевое время жизни:

```
res.set_cookie("visits_count", '1', max_age=0)
```

Куки классные, но у них есть недостатки:

1. Вся информация, записанная в куки, хранится в открытом виде (не шифруется) и может быть доступна любому человеку для чтения и изменения. Поэтому в куки нельзя хранить пароли, данные банковских карт и другие чувствительные для потери данные.
2. Куки можно отключить в браузере, и если пользователь их отключит, мы об об этом не узнаем.
3. Куки не безразмерные. Каждая может хранить до 4 КБ данных, кроме того, у любого браузера есть свое ограничение на количество куки, которое может установить каждый сайт (это число всегда меньше 50).
4. Куки отправляются с каждым запросом к серверу, поэтому, если установить большое число больших куки, запросы к серверу будут тяжелыми и сайт будет работать медленно.

Сессии во Flask очень похожи на куки, но имеют большое преимущество: гарантируется, что содержимое сессии не может быть изменено пользователем (если у него нет нашего секретного ключа). Для работы с сессиями есть специальный объект `flask.session`, его надо импортировать. Давайте перепишем часть со счетчиком посещений на сессии:

```
@app.route("/session_test")
def session_test():
    visits_count = session.get('visits_count', 0)
    session['visits_count'] = visits_count + 1
    return make_response(
        f"Вы пришли на эту страницу {visits_count + 1} раз")
```

Сессии также хранятся в куках, но в зашифрованном виде. По умолчанию сессии существуют до тех пор, пока пользователь не закроет браузер. Чтобы продлить жизнь сессии, нужно присвоить атрибуту `session.permanent` значение True. В таком случае срок жизни сессии будет продлен до 31 дня. Если нужно еще больше (например, год) тогда после создания нашего приложения надо установить параметр PERMANENT_SESSION_LIFETIME.

```
app = Flask(__name__)
app.config['PERMANENT_SESSION_LIFETIME'] = datetime.timedelta(
    days=365
)
```

Удаление данных из сессии происходит так же, как и удаление пары «ключ-значение» из словаря. Например, это можно сделать так:

```
session.pop('visits_count', None)
```

На использовании объекта сессии можно построить простую систему авторизации для нашего приложения. Для этого можно после правильного ввода пользователем логина и пароля записывать в его сессию некоторое сложноподбираемое значение, которое использовать как ключ доступа к личным разделам пользователя. Несмотря на то, что значение сессии нельзя изменить, злоумышленник может перехватить значение сессии и начать отправлять свои запросы, представляясь другим пользователем, так как при передаче данных от клиента до сервера и обратно по протоколу HTTP данные не шифруются. Для шифрованной передачи используется протокол HTTPS.

2. Авторизация пользователя

Для добавления функциональности авторизации пользователей можно воспользоваться библиотекой flask-login. Для начала установим библиотеку:

```
pip install flask-login
```

Выполним первоначальную настройку модуля. Сначала импортируем нужный класс:

```
from flask_login import LoginManager
```

Затем сразу после создания приложения flask инициализируем `LoginManager`:

```
login_manager = LoginManager()
login_manager.init_app(app)
```

Для верной работы flask-login у нас должна быть функция для получения пользователя, украшенная декоратором `login_manager.user_loader`. Добавим ее:

```
@login_manager.user_loader
def load_user(user_id):
    db_sess = db_session.create_session()
    return db_sess.query(User).get(user_id)
```

Кроме того, наша модель для пользователей должна содержать ряд методов для корректной работы flask-login, но мы не будем создавать их руками, а воспользуемся множественным наследованием. И помимо `SqlAlchemyBase` унаследуем `User` от `UserMixin` из модуля flask-login, то есть заголовок класса модели пользователей будет выглядеть так:

```
class User(SqlAlchemyBase, UserMixin):
```

Сделаем форму авторизации пользователя, назовем ее `LoginForm`. Она будет практически совпадать с той, что мы делали на уроке знакомства с flask-wtf:

```
class LoginForm(FlaskForm):
    email = EmailField('Почта', validators=[DataRequired()])
    password = PasswordField('Пароль', validators=[DataRequired()])
    remember_me = BooleanField('Запомнить меня')
    submit = SubmitField('Войти')
```

Сделаем к ней шаблон login.html:

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

{% block content %}
    <h1>Авторизация</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 %}
                <div class="alert alert-danger" role="alert">
                    {{ error }}
                </div>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(class="form-control", type="password") }}<br>
            {% for error in form.password.errors %}
                <div class="alert alert-danger" role="alert">
                    {{ error }}
                </div>
            {% endfor %}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit(type="submit", class="btn btn-primary") }}</p>
        <div>{{ message }}</div>
    </form>
{% endblock %}
```

И, наконец, сделаем обработчик адреса /login:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        db_sess = db_session.create_session()
        user = db_sess.query(User).filter(User.email == form.email.data).first()
        if user and user.check_password(form.password.data):
            login_user(user, remember=form.remember_me.data)
            return redirect("/")
        return render_template('login.html',
                               message="Неправильный логин или пароль",
                               form=form)
    return render_template('login.html', title='Авторизация', form=form)
```

(Не забудьте импортировать класс `LoginForm` и метод `login_user` из модуля flask-login.)

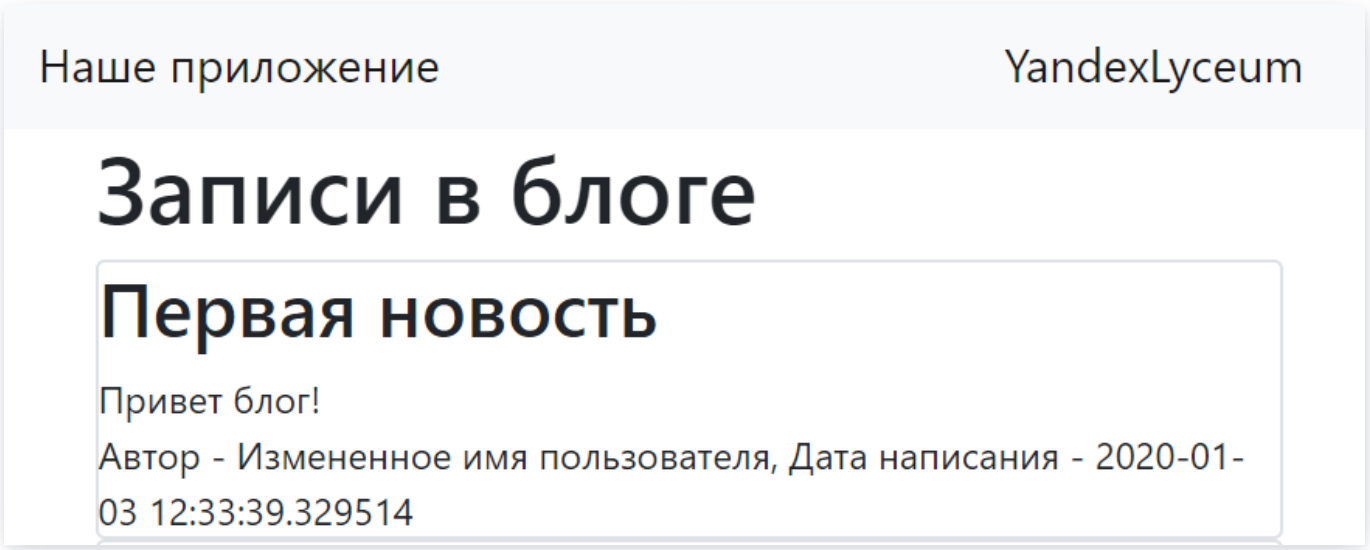
Если форма логина прошла валидацию, мы находим пользователя с введенной почтой, проверяем, введен ли для него правильный пароль, если да, вызываем функцию `login_user` модуля flask-login и передаем туда объект нашего пользователя, а также значение галочки «Запомнить меня». После чего перенаправляем пользователя на главную страницу нашего приложения.

Давайте запустим и попробуем. После ввода правильного логина и пароля нас действительно перенаправляет на главную страницу веб-приложения. Но как понять что что-то поменялось? Для этого существует атрибут `flask_login.current_user`, доступный в любом обработчике URL и в шаблонах. Если пользователь залогинен, то там содержится объект класса `User` текущего пользователя, а если никто не авторизовался — анонимного пользователя.

Давайте добавим следующий код в элемент nav базового шаблона:

```
{% if current_user.is_authenticated %}
    <a class="navbar-brand" href="/logout">{{ current_user.name }}</a>
{% else %}
    <p>
        <a class="btn btn-primary " href="/register">Зарегистрироваться</a>
        <a class="btn btn-success" href="/login">Войти</a>
    </p>
{% endif %}
```

Теперь после входа и перенаправления на главную страницу мы увидим имя залогиненного пользователя:



Теперь добавим обработчик адреса `/logout`. Для него нам не понадобится отдельный шаблон, поскольку это не отдельная страница, а действие.

```
@app.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect("/")
```

Тут все просто — мы «забываем» пользователя при помощи функции `logout_user` и перенаправляем его

на главную страницу нашего приложения. Из интересного здесь — декоратор `login_required` (не забудьте это импортировать). Таким декоратором можно украшать обработчики страниц, на которые может попасть только авторизованный пользователь.

Давайте добавим небольшое изменение в главную страницу нашего приложения, чтобы для авторизованного пользователя отображались и его личные записи.

```
if current_user.is_authenticated:
    news = db_sess.query(News).filter(
        (News.user == current_user) | (News.is_private != True))
else:
    news = db_sess.query(News).filter(News.is_private != True)
```

3. Добавление, изменение и удаление данных

Чтобы далеко не ходить, давайте рассмотрим добавление, изменение и удаление данных на примере новостей. Начнем с добавления. Разумеется, добавлять новости у нас могут только авторизованные пользователи, поэтому давайте добавим в шаблон отображения списка новостей кнопку, доступную только им:

```
{% if current_user.is_authenticated %}
    <a href="news" class="btn btn-secondary">Добавить новость</a>
{% endif %}
```

Создадим форму добавления новости `NewsForm`, в каталог forms добавим файл news.py:

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


class NewsForm(FlaskForm):
    title = StringField('Заголовок', validators=[DataRequired()])
    content = TextAreaField("Содержание")
    is_private = BooleanField("Личное")
    submit = SubmitField('Применить')
```

Шаблон для редактирования новости news.html:

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

{% block content %}
<h1>Добавление новости</h1>
<form action="" method="post">
    {{ form.hidden_tag() }}
    <p>
        {{ form.title.label }}<br>
        {{ form.title(class="form-control") }}<br>
```

```
        {% for error in form.title.errors %}
            <p class="alert alert-danger" role="alert">
                {{ error }}
            </p>
        {% endfor %}
    </p>
    <p>
        {{ form.content.label }}<br>
        {{ form.content(class="form-control") }}<br>
        {% for error in form.content.errors %}
            <p content="alert alert-danger" role="alert">
                {{ error }}
            </p>
        {% endfor %}
    </p>
    <p>{{ form.is_private() }} {{ form.is_private.label }}</p>
    <p>{{ form.submit(type="submit", class="btn btn-primary") }}</p>
    {{message}}
</form>
{% endblock %}
```

И обработчик:

```
@app.route('/news',  methods=['GET', 'POST'])
@login_required
def add_news():
    form = NewsForm()
    if form.validate_on_submit():
        db_sess = db_session.create_session()
        news = News()
        news.title = form.title.data
        news.content = form.content.data
        news.is_private = form.is_private.data
        current_user.news.append(news)
        db_sess.merge(current_user)
        db_sess.commit()
        return redirect('/')
    return render_template('news.html', title='Добавление новости',
                           form=form)
```

Тут нет ничего такого, что мы уже не делали. Единственный интересный момент — сказать сессии, что мы изменили текущего пользователя с помощью метода `merge`.

Сделаем редактирование новости. Будем использовать уже созданную форму и шаблон, напишем только другой обработчик:

```
@app.route('/news/<int:id>', methods=['GET', 'POST'])
@login_required
def edit_news(id):
```

```
    form = NewsForm()
    if request.method == "GET":
        db_sess = db_session.create_session()
        news = db_sess.query(News).filter(News.id == id,
                                          News.user == current_user
                                          ).first()
        if news:
            form.title.data = news.title
            form.content.data = news.content
            form.is_private.data = news.is_private
        else:
            abort(404)
    if form.validate_on_submit():
        db_sess = db_session.create_session()
        news = db_sess.query(News).filter(News.id == id,
                                          News.user == current_user
                                          ).first()
        if news:
            news.title = form.title.data
            news.content = form.content.data
            news.is_private = form.is_private.data
            db_sess.commit()
            return redirect('/')
        else:
            abort(404)
    return render_template('news.html',
                           title='Редактирование новости',
                           form=form
                           )
```

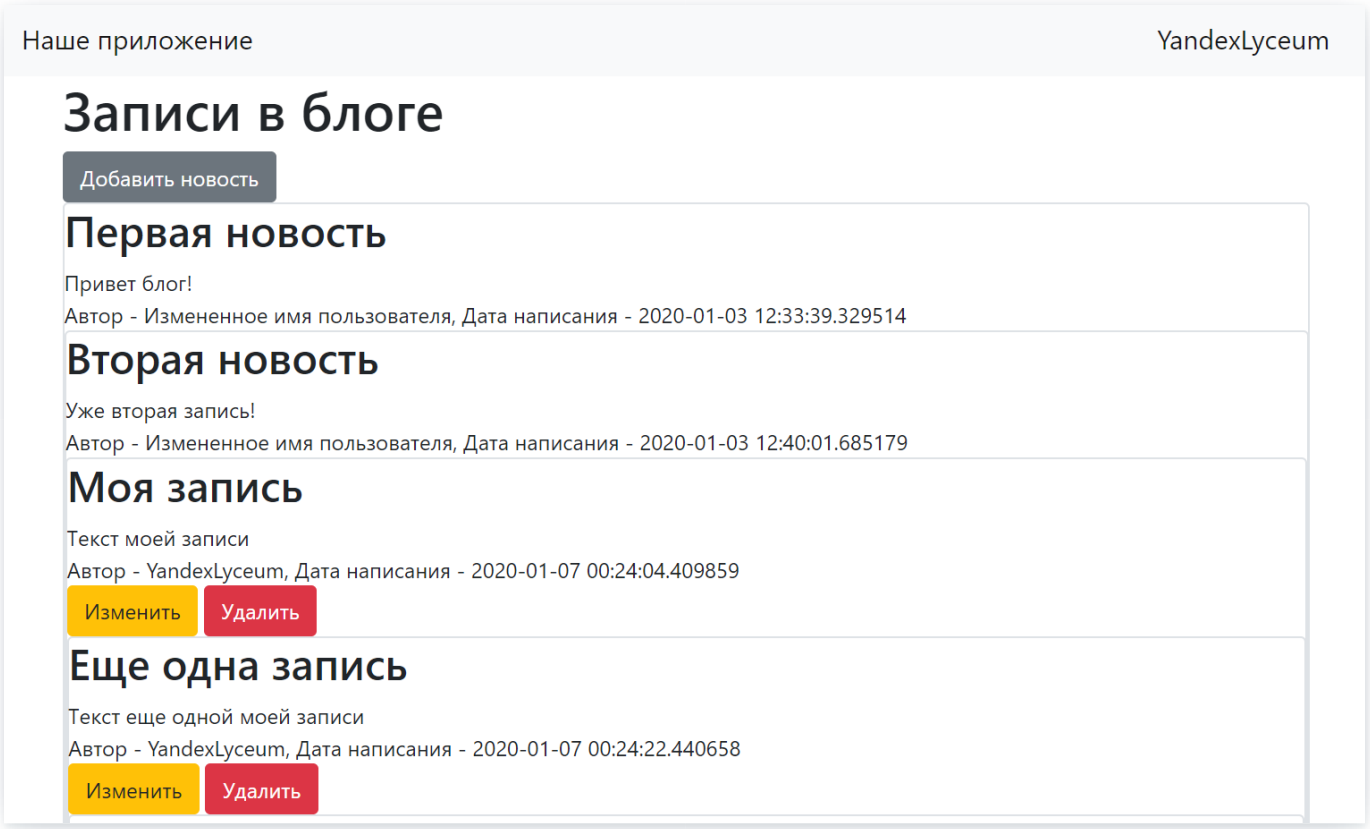
Если мы запросили страницу записи, ищем ее в базе по `id`, причем автор новости должен совпадать с текущим пользователем. Если что-то нашли, предзаполняем форму, иначе показываем пользователю страницу 404. Такую же проверку на всякий случай делаем перед изменением новости.

Добавим кнопки «Изменить» и «Удалить» к каждой новости в списке новостей, но только для тех записей, автором которых является `current_user`. Немного изменим шаблон index.html.

```
{% if current_user.is_authenticated and current_user == item.user %}
    <div>
        <a href="/news/{{ item.id }}" class="btn btn-warning">
            Изменить
        </a>
        <a href="/news_delete/{{ item.id }}" class="btn btn-danger">
            Удалить
        </a>
    </div>
{% endif %}
```

В адреса каждой из ссылок допишем `id` новости.

У нас получится что-то вроде:



Добавим еще обработчик удаления записи:

```
@app.route('/news_delete/<int:id>', methods=['GET', 'POST'])
@login_required
def news_delete(id):
    db_sess = db_session.create_session()
    news = db_sess.query(News).filter(News.id == id,
                                      News.user == current_user
                                      ).first()
    if news:
        db_sess.delete(news)
        db_sess.commit()
    else:
        abort(404)
    return redirect('/')
```

Итак, у нас получилось достаточно функциональное рабочее приложение. Давайте добавим еще пару штрихов.

4. Отношение многие ко многим

Когда вы создавали базы данных, наверняка обратили внимание, что когда сущности связаны **одна ко многим** (у одного пользователя есть несколько записей, в одном жанре есть несколько фильмов), то проблем с проектированием базы данных не возникает. Трудности начинаются, когда сущности связаны **многие ко многим** (в одном заказе может быть несколько наименований товаров, но эти же товары могут быть во многих заказах). Те, кто столкнулся с этой проблемой, вероятно, провели исследования и уже знают, что в SQL такая ситуация решается созданием промежуточной таблицы. Давайте посмотрим, как связь **многие**

ко многим можно реализовать с помощью sqlalchemy.

Давайте представим, что наши записи могут принадлежать к одной или нескольким категориям, чтобы можно было проще их фильтровать по интересующей пользователя теме.

Добавим модель `Category`, сделаем для этого новый файл category.py в папке data:

```
import sqlalchemy
from .db_session import SqlAlchemyBase


class Category(SqlAlchemyBase):
    __tablename__ = 'category'
    id = sqlalchemy.Column(sqlalchemy.Integer, primary_key=True,
                           autoincrement=True)
    name = sqlalchemy.Column(sqlalchemy.String, nullable=True)
```

Кроме того, добавим перед моделью с категориями информацию о промежуточной таблице:

```
association_table = sqlalchemy.Table(
    'association',
    SqlAlchemyBase.metadata,
    sqlalchemy.Column('news', sqlalchemy.Integer,
                      sqlalchemy.ForeignKey('news.id')),
    sqlalchemy.Column('category', sqlalchemy.Integer,
                      sqlalchemy.ForeignKey('category.id'))
)
```

Тут мы говорим sqlalchemy, что нам нужна вспомогательная таблица `association` (не обязательно такое имя, часто их называют `имя_сущности1_to_имя_сущности2`, то есть в нашем случае `news_to_category`), которая будет содержать только несколько внешних ключей на каждую из таблиц.

Немного обновим модель `News`, чтобы можно было получать доступ к категориям новости как к списку:

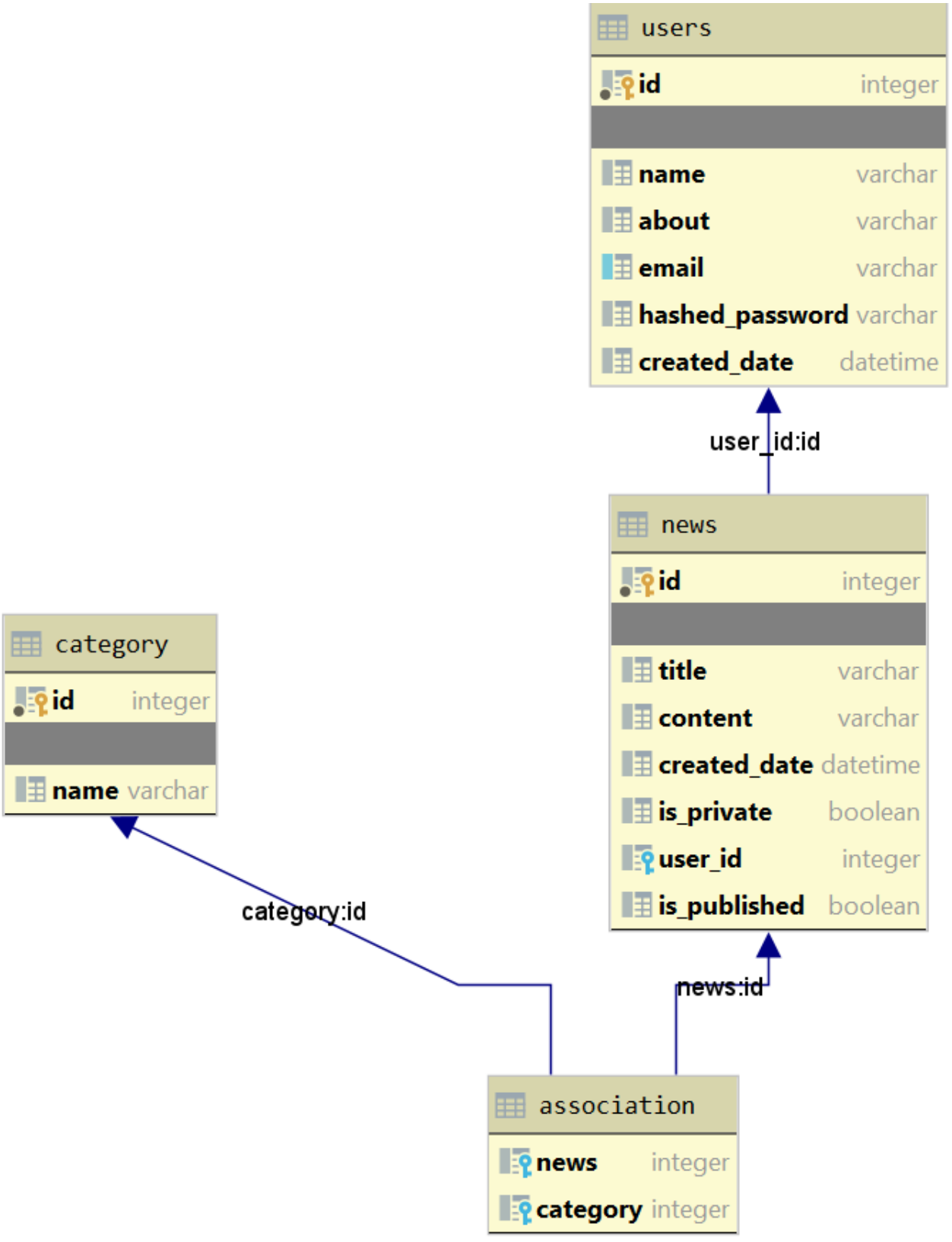
```
categories = orm.relationship("Category",
                              secondary="association",
                              backref="news")
```

К сожалению, чтобы sqlalchemy применил изменения, надо удалить уже существующие таблицы в базе данных, которые затрагивают эти изменения. В данном случае необходимо удалить таблицу `news`. После перезапуска приложения таблицы будут созданы уже с правильной структурой. Такое поведение не очень удобно, скоро мы узнаем, как с этим можно справиться.

Не забудьте добавить в файл __all_models.py импорт новой модели:

```
from . import category
```

Если мы все сделали правильно, наша база данных станет выглядеть примерно вот так:



Теперь, когда мы будем вызывать метод `news.categories.append(category)` и передавать туда объект типа `Category`, у нас будет создаваться запись именно в промежуточной таблице.

Чтобы удалить категорию у новости, достаточно сделать:

```
news.categories.remove(category)
```

5. Заключение

На этом мы завершаем рассмотрение функциональности SQLalchemy, хотя в следующих темах будем использовать полученные знания (и еще вернемся к ней, когда будем рассматривать механизм миграций), а теперь нас ждет создание своего собственного API.

Справка