Урок Flask-restful

REST-API. Flask-RESTful

Аннотация

Сегодня мы посмотрим, как создавать RESTful-сервисы с помощью библиотеки flask-restful.

1. Flask-RESTful

Все то, что мы сделали ранее при создании RESTful-сервиса, можно обернуть в более лаконичную, а главное — объектно-ориентированную оболочку, забыв о множестве разбросанных функций-обработчиков. Для этого установим дополнительный модуль flask-restful с помощью pip.

```
pip install flask-restful
```

Из него нам понадобятся следующие объекты и функции:

```
from flask_restful import reqparse, abort, Api, Resource
```

Создадим вторую версию нашего REST-сервиса. После создания flask-приложения создадим объект RESTful-API:

```
app = Flask(__name__)
api = Api(app)
```

Для каждого ресурса (единица информации в REST называется **ресурсом**: новости, пользователи и т. д.) создается два класса: для одного объекта и для списка объектов. В нашем случае это классы `NewsResource` и `NewsListResource` соответственно. Создадим их в отдельном модуле, который назовем news_resources.py. Оба этих класса необходимо унаследовать от класса `Resource` из модуля flask-restful.

В классе одного объекта (`NewsResource`) определяются операции, которые мы можем сделать с **одним** объектом: получить информацию об объекте, изменить информацию, удалить объект. flask-restful подразумевает, что эти методы будут иметь имена, аналогичные соответствующим HTTP-запросам: `get`, `put`, `post`, `delete`. И все они, конечно, должны принимать в качестве аргумента идентификатор объекта.

Когда мы проектировали REST-сервис вручную, в каждой функции-обработчике мы проверяли, существует ли новость с таким идентификатором, и если нет, то отправляли ошибку. А также мы определили функцию

not_found с декоратором @app.errorhandler(404), чтобы изменить формат ответа сервера в случае ошибки. Давайте вынесем эту проверку в отдельную функцию в файле с ресурсами для записей:

```
def abort_if_news_not_found(news_id):
    session = db_session.create_session()
    news = session.query(News).get(news_id)
    if not news:
        abort(404, message=f"News {news_id} not found")
```

Функция abort генерирует HTTP-ошибку с нужным кодом и возвращает ответ в формате JSON, поэтому функция not_found нам больше не нужна. В предыдущем нашем варианте мы никак не использовали коды ответа сервера, это неверно. Наше сообщение об ошибке мог прочитать только человек, а для программы-клиента оно ничего не значит, ответ пришел со статусом OK (200). Давно знакомый вам код 404 означает, что запрашиваемый ресурс не найден. Подробнее про статусы состояния протокола HTTP можно почитать **здесь**.

Класс NewsResource с методами получения информации и удаления будет иметь вид:

```
class NewsResource(Resource):
    def get(self, news_id):
        abort_if_news_not_found(news_id)
        session = db_session.create_session()
        news = session.query(News).get(news_id)
        return jsonify({'news': news.to_dict(
            only=('title', 'content', 'user_id', 'is_private'))})

    def delete(self, news_id):
        abort_if_news_not_found(news_id)
        session = db_session.create_session()
        news = session.query(News).get(news_id)
        session.delete(news)
        session.commit()
        return jsonify({'success': 'OK'})
```

В классе списка объектов (NewsListResource) определяются операции, которые мы можем сделать с **набором** объектов: показать список объектов и добавить объект в список. В этом классе нам потребуется реализовать два метода: get и post без аргументов. Доступ к данным, переданным в теле POST-запроса, осуществляется с помощью парсера аргументов из модуля reqparse, который предварительно нужно создать и добавить в него аргументы:

```
parser = reqparse.RequestParser()
parser.add_argument('title', required=True)
parser.add_argument('content', required=True)
parser.add_argument('is_private', required=True, type=bool)
parser.add_argument('is_published', required=True, type=bool)
parser.add_argument('user_id', required=True, type=int)
```

По-прежнему считаем, что все поля новости являются обязательными. Также укажем тип идентификатора пользователя — целое число. Теперь всю проверку аргументов запроса за нас будет делать модуль reqparse.

Таким образом, класс NewsListResource будет иметь вид:

```
class NewsListResource(Resource):
    def get(self):
```

```
        session = db_session.create_session()
        news = session.query(News).all()
        return jsonify({'news': [item.to_dict(
            only=('title', 'content', 'user.name')) for item in news]})

    def post(self):
        args = parser.parse_args()
        session = db_session.create_session()
        news = News(
            title=args['title'],
            content=args['content'],
            user_id=args['user_id'],
            is_published=args['is_published'],
            is_private=args['is_private']
        )
        session.add(news)
        session.commit()
        return jsonify({'success': 'OK'})
```

После того как мы создали классы ресурсов, нам надо внести их в настройки нашего RESTful-API, указав имя класса и URL. Параметр URL также указывается в угловых скобках. Для этого в main.py перед запуском нашего приложения необходимо добавить вот такой код:

```
# для списка объектов
api.add_resource(news_resources.NewsListResource, '/api/v2/news')

# для одного объекта
api.add_resource(news_resources.NewsResource, '/api/v2/news/<int:news_id>')
```

Давайте проверим такую реализацию RESTful-сервиса с помощью запросов в нашем тестовом файле test.py. Убедитесь, что все работает, как нужно.

Документация по модулю flask-RESTful есть на **официальном сайте**.

Если ваше приложение построено по правилам REST, вы получите логичную организацию работы с ресурсами, даже если их в приложении большое количество. Клиенты используют простые и понятные URL, а новым клиентам не составит труда разобраться с интерфейсом вашего приложения. Также в мире веб-разработки стандартом становится предоставлять **RESTful API** для сторонних приложений.

Есть ли подводные камни при использовании REST? Да, и этим камнем является HTML. Дело в том, что спецификация HTML позволяет создавать формы, отправляющие только GET- или POST-запросы. Поэтому для нормальной работы с другими методами (PUT, DELETE) приходится имитировать их искусственно. Часто в этих случаях используют JavaScript-библиотеки, например, JQuery.

Как и предыдущем уроке, здесь мы не рассматривали вопрос авторизации. В видеоматериалах к уроку эта тема раскрыта.

2. Миграции

Мы уже говорили о том, что sqlalchemy при старте не вносит изменения в таблицы, если они уже созданы. Но абсолютно всегда бывает ситуация, когда после запуска первой версии нам захочется что-то поменять такого, что затронет и наши модели. Наверняка нам захочется хранить в базе дополнительную информацию. Что делать в таком случае? Не удалять же базу данных с информацией о куче пользователей, они такого

не простят. Конечно, можно написать SQL-скрипты для приведения базы данных в актуальное состояние, но в общем случае это занятие не из приятных, потому что столбцы могут изменяться, удаляться, могут меняться связи между моделями и т. д. Тут на помощь приходят инструменты для **миграции**.

Уже упоминалось, что это такая штука, вроде системы контроля версий, только для базы данных. Инструменты миграции созданы для того, чтобы поддерживать вашу базу данных в актуальном состоянии и автоматически генерировать скрипты для перехода состояния базы из состояния `n` к состоянию `n + 1` и обратно (примерно как переходы по истории коммитов).

Какие-то ORM содержат модули миграции прямо из коробки (например, в Django), для sqlalchemy нам придется устанавливать и настраивать такую библиотеку отдельно. Она называется **Alembic**.

Есть еще flask-migrate — обертка для alembic, упрощающая работу именно для flask-приложений, но мы рассмотрим более общий пример.

Установим библиотеку:

```
pip install alembic
```

После в командной строке нужно перейти в директорию с нашим проектом и выполнить команду:

```
alembic init alembic
```

Эта команда создаст директорию с именем alembic, в которой модуль будет хранить все необходимые файлы. А также alembic.ini в директории вашего проекта.

Откройте alembic.ini и измените строку с параметром `sqlalchemy.url` на путь к вашей базе данных:

```
sqlalchemy.url = sqlite:///db/blogs.sqlite?check_same_thread=False
```

После этого перейдем в папку alembic. Нас интересует файл env.py. Откроем его и найдем строчку с переменной `target_metadata`. Эта переменная должна «узнать» о всех файлах нашей модели. Напишем там следующий код:

```
import sys
sys.path.insert(0, 'Путь к папке вашего проекта')
from data.db_session import SqlAlchemyBase
import data.__all_models
target_metadata = SqlAlchemyBase.metadata
```

`sys.path.insert` — вставляет путь к каталогу в PATH. Это нужно для того, чтобы было проще писать импорты данных остальных файлов.

Затем импортируем нашу базу данных, напоминаем ей о всех моделях, после чего получаем данные о всех моделях.

Давайте попробуем. Например, мы хотим добавить в модель с новостью поле, которое показывает, опубликована запись или нет, чтобы потом показывать в списке только опубликованные записи. Идем в модель `News` и добавляем в нее новое поле:

```
is_published = sqlalchemy.Column(sqlalchemy.Boolean, default=True)
```

Теперь запускаем консоль в папке нашего проекта и выполняем команду:

```
alembic revision --autogenerate -m "добавили признак публикации"
```

Alembic посмотрит на наши модели и на базу и создаст в папке alembic/versions новую миграцию вот с таким кодом:

```
from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision = '3acec80b2659'
down_revision = None
branch_labels = None
depends_on = None


def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('news', sa.Column('is_published', sa.Boolean(),
                                    nullable=True))
    # ### end Alembic commands ###


def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_column('news', 'is_published')
    # ### end Alembic commands ###
```

Тут всего две функции. `upgrade` выполняет код для изменения состояния базы, `downgrade` — код для возврата к предыдущему состоянию. Давайте обновим нашу базу, для чего выполним в консоли команду:

```
alembic upgrade head
```

`head` означает, что мы хотим применить все миграции друг за другом для приведения базы в самое актуальное состояние. Вместо `head` можно указать номер ревизии или написать, например, `+2`, чтобы обновиться только на 2 следующие версии.

Посмотрим, что случилось с нашей базой данных. В таблице news появилось поле `is_published`, как и ожидалось, но кроме этого появилась новая таблица `alembic_version`. В этой таблице хранится номер ревизии, которой в настоящий момент соответствует база данных.

Из-за ограничений СУБД sqlite команда `alembic downgrade head`не удалит ранее добавленную в таблицу колонку.

К сожалению, alembic не всемогущ. Вот что он умеет определять.

Операции, которые Alembic умеет выявлять:

- Добавление и удаление таблиц
- Добавление и удаление колонок
- Изменения во внешних ключах
- Изменения в типах колонок
- Изменения в индексах и использованных уникальных ограничениях

А вот какие изменения не умеет:

- Изменение имени таблицы
- Изменение имени колонки

Поэтому бывают ситуации, когда в файлы миграции надо вмешиваться и дописывать в них необходимые изменения. Не бойтесь этого.