* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/165/groups/1257)

[Урок Flask-restful](https://lyceum.yandex.ru/courses/165/groups/1257/lessons/1661)

**REST-API. Flask-RESTful**

1. [Flask-RESTful](https://lyceum.yandex.ru/courses/165/groups/1257/lessons/1661/materials/4067#1)
2. [Миграции](https://lyceum.yandex.ru/courses/165/groups/1257/lessons/1661/materials/4067#2)

**Аннотация**

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

**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 ОК (200). Давно знакомый вам код 404 означает, что запрашиваемый ресурс не найден. Подробнее про статусы состояния протокола HTTP можно почитать [здесь](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP).

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](http://flask-restful.readthedocs.io/en/latest/index.html).

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](https://restfulapi.net/rest-api-design-tutorial-with-example/) 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.

**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](https://alembic.sqlalchemy.org/en/latest/).

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.

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.

[Справка](https://yandex.ru/support/lyceum-students)

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

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты