Продвинутые компоненты aiohttp-сервера

Если бы возможности современных фреймворков ограничивались только базовыми компонентами, то было бы очень сложно и долго написать действительно рабочее и многофункциональное приложение.

В этом уроке мы коснемся вещей, которые значительно упрощают жизнь разработчикам в контексте aiohttp.

Валидация

Проверка правильности введенных данных. В более широком смысле валидация также может приводить данные к единому формату.

Пример. У пользователя просят ввести год рождения, а он вводит MCMXCVIII. Или 2222. Компонент валидации должен проверить эти данные и сообщить пользователю об ошибке. А если пользователю предложат ввести дату рождения в формате дд-мм-гггг, а сервер работает с форматом гггг-мм-дд? Валидация тоже может помочь, преобразов данные из одного формата в другой и обратно.

Также валидация может быть использована, чтобы скрыть часть данных.

Пример. У вас есть объект пользователя, и вы хотите вернуть его в json-ответе. Вы достали пользователя из базы и видите три поля: *id*, *username* и *password*. Хотя вы хороший программист и держите пароль в хэшированном виде, все равно — не хотелось бы его отдавать во всеобщее обозрения. Не удалять же это поле каждый раз руками! Высока вероятность забыть сделать это в каком-то месте. Кроме того, если добавится какое-то новое конфиденциальное поле, нужно пройти по всем местам в коде и удалить его тоже.

Гораздо проще написать одну схему отдаваемых для пользователя полей и использовать ее во всех местах. Тем более, что для написания таких схем есть замечательная библиотека — *Marshmallow.*

marshmallow.**Schema**

Schema — это строго типизированная структура, которую можно использовать для валидации данных, генерации документации, но в основном — для преобразования сырых данных в python-объекты и обратно, то есть для сериализации и десериализации.

Пример 1. Так мы можем описать схему пользователя для задачи из предыдущего абзаца:

```
from marshmallow import Schema, fields

class UserSchema(Schema):
  id = fields.Int(dump_only=True)
  username = fields.Str(required=True)
  password = fields.Str(load_only=True, required=True)
```

Теперь ее можно использовать как для входящих данных, так и для исходящих:

```
# этого пользователя мы достали из базы
>>> user = {'id': 101, 'username': 'admin', 'password': '12345678'}
# и выполнили его преобразование в json с помощью UserSchema
>>> print(UserSchema().dumps(user))

'{"username": "admin", "id": 101}'
```

Как можно видеть по результату, поле *password* не было показано в итоговом json-ответе, потому что у него есть атрибут *load_only* — это поле будет использовано только при преобразовании из json-данных в python-объекты.

Пример 2. Что, если мы хотим сделать валидацию входных данных при логине пользователя? Например, мы хотим, чтобы пользователь ввел свои *username* и *password* и отправил нам в json-формате:

```
# пользователь решил проверить нас на прочность и попробовать перезаписать свой id в базе
>>> raw_json_data = '{"password": "12345678", "username": "admin", "id": 999}'
>>> print(UserSchema().loads(raw_json_data))

marshmallow.exceptions.ValidationError: {'id': ['Unknown field.']}
```

Marshmallow вернул исключение — пользователь передал лишнее поле (можно управлять этим поведением, например, включать лишние поля или исключать с помощью параметра unknown). А мы вернули пользователю ошибку и указали, в чем конкретно он был не прав.

Пример 3. Что, если пользователь забыл передать пароль? Нет проблем, параметр *required=True* напомнит пользователю о его забывчивости:

```
>>> raw_json_data = '{"username": "admin"}'
>>> print(UserSchema().loads(raw_json_data))

marshmallow.exceptions.ValidationError: {'password': ['Missing data for required field.']}
```

`fields`

Python-модуль из библиотеки marshmallow, в котором содержатся все основные типы данных, их можно увидеть [здесь](#).

Мы уже немного познакомились с fields, когда делали UserSchema. Тут вроде бы все понятно: хотим проверить, что значение поля равно числу, выбираем fields.Int или fields.Integer (они равнозначны).

Или fields.Bool, чтобы проверить значение, присланое клиентом из чек-бокса.

Пример. Что, если у нас вложенная структура, например, список пользователей? Казалось бы, надо использовать fields.List, но это поле применимо только к базовым типам данных: (fields.List(fields.Int) справится с [1, 2, 3]).

Решение — fields.Nested. Допустим, у нас есть несколько пользователей, а у каждого пользователя есть еще информация о его месте проживания. Мы хотим вернуть их в json-ответе в поле items.

Создадим схему места проживания:

```
class CitySchema(Schema):
  name = fields.Str()
  index = fields.Int()
```

Создадим схему пользователя с местом проживания:

```
class UserSchema(Schema):
  username = fields.Str()
  city = fields.Nested(CitySchema)
```

Создадим схему массива пользователей в поле items:

```
class ListUserSchema(Schema):
  items = fields.Nested(UserSchema, many=True)
```

Теперь мы спокойно можем преобразовать python-объект в json-ответ, исключив все ненужные данные.

Обратите внимание на параметр *many* для fields.Nested — если вы хотите передать в это поле массив с объектами, его нужно указывать (ListUserSchema.items), а если хотите передать единственный вложенный объект, то не нужно (UserSchema.city).

Небольшое лирическое отступление.

Цель жизни бэкенд-разработчика — писать все новые и новые методы для своего приложения (шутка, конечно, цель гораздо обширнее). А цель жизни фронтенд-разработчика — прикручивать эти новые методы к красивому клиенту.

Как только бэкенд-разработчик сделал новый метод, нужно сообщить фронту его путь, HTTP-метод запроса, какие параметры и в каком виде он принимает, какие данные он отдает... А если пришла другая команда и хочет интегрироваться? Нужно опять описывать все нужные методы. А если метод поменялся, надо сообщить всем об изменениях, пусть даже фронт и другая команда еще не начали интеграцию.

Было бы классно, если бы такое описание было в едином стиле и в общей доступности. А если бы оно еще и генерировалось за нас, было бы просто фантастично.

Насколько реальна эта фантастика — обсудим в следующем уроке.