Swagger

Инструмент для документирования API.

Позволяет описывать API в формате json и yaml и по этим описаниям генерировать красивую страницу, на которой содержится описание всех методов, ответов на них и т.д. За отображение и работу этой страницы отвечает Swagger UI. Кроме отображения документации, он также позволяет эту документацию тестировать, отправляя запросы к API прямо из браузера. Кроме того, по такой документации можно сгенерировать код веб-сервера для очень многих языков с помощью Swagger Codegen.

Пример «сырой» документации:

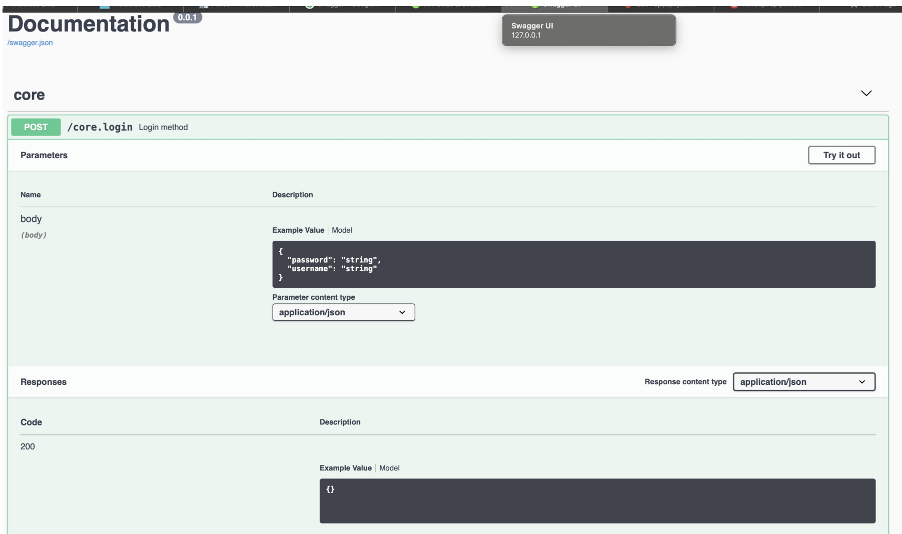
```yaml
paths:
  /tokens.add:
    post:
      tags:
        - Mobile

      summary: Создать/обновить firebase-токен
      description: Создает или обновляет firebase токен.<br/> Если токен уже существует, то он будет создан.

      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/MobileData'

      responses:
        200:
          description:  Токен успешно создан/обновлен
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/SuccessTokenResponse'
        400:
          description: Некорректный запрос
```

Пример Swagger UI:



Вы не находите, что вводные поля на последней картинке очень похожи на UserSchema, которые мы писали в прошлом уроке?

Действительно, с помощью marshmallow-схем можно генерировать Swagger-документацию, с помощью библиотеки aiohttp_apispec.

aiohttp_apispec

Библиотека, которая позволяет связать валидацию входящих/исходящих данных и генерацию/предоставление документации.

Чтобы использовать aiohttp_apispec, необходимо ее настроить и прикрепить к экземпляру aiohttp.Application:

```python
from aiohttp import web
from aiohttp_apispec import setup_aiohttp_apispec

app = web.Application()
setup_aiohttp_apispec(app, title='Documentation', swagger_path='/swagger', url='/swagger.json')
```

Стоит остановить внимание на параметре *url* — он отвечает за адрес, на котором будет показан **Swagger UI**.

```python
class CoreLoginView(View):
    @docs(tags=['core'],
          summary='Login method')
    @json_schema(LoginRequestSchema)
    @response_schema(EmptyResponseSchema)
    async def post(self):
```

Также для каждого нужного View надо применить несколько декораторов. Например, для LoginView можно применить:

- @docs — отвечает за отображение этого View в документации и его описанию. Summary - просто текстовое описание, выводящиеся в Swagger UI вместе с методом. tags - это список строк, по которым будут группироваться методы в документации. Например, у метода core.login судя на картинке выше tags=["core"].
- @json_schema — валидирует входящие json-данные
- @response_schema — показывает, какие данные в случае успешного ответа вернутся из этого View.

Чтобы aiohttp_apispec мог валидировать входящие запросы, необходимо добавить validation_middleware в общий список мидлвар. Что такое middleware, рассмотрим в следующей карточке.

Как работает aiohttp_apispec при обработке запроса

1. Запрос проходит через validation_middleware, которая ищет подходящий View и забирает оттуда Schema для входных данных (на картинке это LoginRequestSchema)
2. Дальше middleware видит, что данные должны придти в формате json и прогоняет пришедший в запрос json через Schema (условно LoginRequestSchema().loads(await request.json()))
3. Если данные невалидны, поднимается ошибка о несоответствии, которая преобразуется в ошибку HTTPUnprocessableEntity. Чтобы на фронт не уходила непонятная ошибка 422, важно ставить middleware, которая обрабатывает исключения перед validation_middleware
4. Если данные валидны, они кладутся в соответствующий ключ в запросе. Например, валидные данные для @json_schema можно найти в self.request['json'].
5. При обработке ответа @response_schema не применяется автоматически, но она нужна для генерации правильной документации. А дописать код для ее автоматического применения можно руками.

Как работает aiohttp_apispec при генерации документации

1. При навешивании любого декоратора из aiohttp_apispec (@docs, @json_schema...) этот декоратор добавляет к функции скрытый аттрибут (потому что функции тоже объекты) __apispec__, содержащий все схемы и данные о документации
2. При вызове setup_aiohttp_apispec происходит обход всех Route в Router
3. Если Route связан с View, у которого есть атрибут __apispec__, он добавляется к документации.

Главный вывод из этого сложного объяснения: сначала настройте все Route, а потом инициализируйте apispec.

Примеры

1. валидация в aiohttp-приложении