Návod na vytváranie endpointov

MonAnt Flask API is implemented using python, Flask and Apistrap. This document represents a guideline, how to conveniently and quickly create simple and safe API endpoint.

Endpoint structure

Api endpoints are usually grouped into *Blueprints*, which represent bigger parts of API and also one version. First step in order to create a new endpoint is to decide, to which version of endpoints it should belong. Each group of the same version endpoints is located in version coresspond folder. Second step is to decide, to which Blueprint it should belong. Each API endpoint accepts certain payload - body of request (usually POST or PUT). Since this API uses Schematics, you can simply define schema (items) of your request, so you can validate it. Request schemas are defined in requests.py. You can also define schema of response of your endpoint. This is done the same way in response.py file. Mapping of schemas to endpoint definitions is done via defined decorators.

Endpoint naming conventions

There are certain conventions to follow, when creating a new endpoint: - name of entity, for which there are multiple records, is in plural and as simple as possible - for exam. `articles` - if endpoint consists of multiple words, they are delimited with - sign: `data-providers` - if we search specific entities / documents according to their ID, ID is a part of url delimited with / sign: `$SERVER_ADDRESS/v1/articles/<article_id>` - if we filter entities by other attributes related to given entity, they are sent as parameters: `$SERVER_ADDRESS/v1/articles?author=shakespeare&created_before=20180101` - CRUD operations are defined by type of request, not by name: `HTTP POST $SERVER_ADDRESS/v1/articles/` - creates article, its attributes are passed as data - all letters are lowercase

Endpoint views hierarchy

Endpoint definition should be in correct version folder, such as `v1`, in views file, that is dedicated to first-level entity queried in endpoint. If we have an endpoint `$SERVER_ADDRESS/v1/articles`, it's view file, where article-related endpoint definitions are, is `./v1/article_views.py` file.

If we have an endpoint concerning multiple related entities, its views file should be chosen according to top-level entity: views file of `$SERVER_ADDRESS/v1/articles/<article_id>/photos` endpoint, which returns all photos of given article, should belong to `./v1/article_views.py` file.

Example

Let's say we want to create a new API endpoint (version 1), that adds new book and its author to database. The route of endpoint is `/v1/books`, method is POST, request scheme contains just book name and author, and endpoint responds with at least newly created book id and status code 200:

- first thing to do is to create request schema. In file `requests.py`, add following:

```
from schematics.models import Model
    from schematics.types import StringType


    class CreateBookRequest(Model):
        name: StringType = StringType(required=True)
        author: StringType = StringType(required=True)
```

- then create scheme of response (file `responses.py`):

```
from schematics.models import Model
    from schematics.types import IntType

    class CreateBookResponse(Model):
        id: IntType = IntType(required=True)
```

- then define endpoint with route in file of folder `v1`, which contains your Blueprint definition

(e.g. `book_views.py`) python from flask import Blueprint

~~~
from .swagger import swagger
    from .requests import CreateBookRequest
    from .responses import CreateBookResponse
    from api.models import Book
book_api: Blueprint = Blueprint('book_api', __name__)
@books_api.route('/books', methods=['POST'])  # define route and method
    @swagger.autodoc()  # generate documentation for endpoint (available on
<server_address>/apidocs/
    @swagger.accepts(CreateBookRequest)  # accept only correct request schema
    @swagger.responds_with(CreateBookResponse, code=200)  # respond with
defined response schema
    def create_book(payload: CreateBookRequest):  # pass Book as parameter
        """ Create book route handler """
        print(payload.author)  # access attributes in request
        book = Book(payload)   # create new book entity in database

        return CreateBookResponse(dict(
        id=book.id
    ))
* the last step, if not already done, is to reqister API blueprint in
```api.py```:
~~~

from api.views.v1.book*views.py import books*api as book*api*v1

...

```
def create_app(name: str):
    """ Creates Flask APP """
        app: Flask = Flask(name)

        # add auto-generated API doc
        swagger.init_app(app)

        # register API endpoints implementation
        app.register_blueprint(book_api_v1, url_prefix='/v1')

    return app
## API versioning

In case of creating new version of endpoints you should create new folder,
which name correspond to new version, such as
```

won't be supported in new version.

Example

Let's say we want to create `v2` version of `book_api` endpoints. First step is to create `v2` folder in `views`. Then copy `book_views.py` file from `v1` folder. In this file add import of previous version view `from api.views.v1 import book_views as previous_version`. In this case it is import of `v1` version of `book_views`. and should be used in all endpoints, where functionality is not changed, something like `previous_version.create_book(payload)`. This is example of the inheritance of endpoint versions. For all other endpoints you can add new functionality as usual. New `book_views.py` file should look like this:

```
from flask import Blueprint

    from .swagger import swagger
    from .requests import CreateBookRequest
    from .responses import CreateBookResponse
    from api.models import Book
    from api.views.v1 import book_views as previous_version


    book_api: Blueprint = Blueprint('book_api_v2', __name__)


    @books_api.route('/books', methods=['POST'])
    @swagger.autodoc()
    @swagger.accepts(CreateBookRequest)
    @swagger.responds_with(CreateBookResponse, code=200)
    def create_book(payload: CreateBookRequest):
        """ Create book route handler """
        return previous_version.create_book(payload)
```

Don't forget to reqister all API blueprint in `api.py`:

```
from api.views.v1.book_views.py import books_api as book_api_v1
from api.views.v2.book_views.py import books_api as book_api_v2

...

    def create_app(name: str):
    """ Creates Flask APP """
        app: Flask = Flask(name)

        # add auto-generated API doc
        swagger.init_app(app)

        # register API endpoints implementation
        app.register_blueprint(book_api_v1, url_prefix='/v1')
        app.register_blueprint(book_api_v2, url_prefix='/v2')

    return app
```

And that's it! Don't forget to add tests.