GeekBrains

Flask

JSON REST API, CRUD, Swagger, marshmallow, Flask-COMBO-JSONAPI, ComboJSONAPI

На этом уроке

1. Познакомимся со спецификацией JSON API.
2. Узнаем, что такое сериализация/десериализация данных.
3. Узнаем, что такое marshmallow и marshmallow-jsonapi.
4. Познакомимся со Swagger.
5. Посмотрим на обработку и выдачу связей.

Оглавление

Теория

marshmallow

Библиотека для сериализации и десериализации данных (преобразование между объектами python и JSON данными).

JSON

JSON — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом Крокфордом. Несмотря на происхождение от JavaScript (точнее, от подмножества языка стандарта ECMA-262 1999 года), формат считается независимым от языка и может использоваться практически с любым языком программирования. Для многих языков существует готовый код для создания и обработки данных в формате JSON.

API

API (программный интерфейс приложения, интерфейс прикладного программирования) (англ. application programming interface, API [эй-пи-ай]) — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.

Swagger

Swagger — это язык описания интерфейсов для описания RESTful API, выраженных с помощью JSON. Swagger используется вместе с набором программных инструментов с открытым исходным кодом для проектирования, создания, документирования и использования веб-служб RESTful.

REST

REST — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях это приводит к повышению производительности и упрощению архитектуры.

Практическое задание

Установка и настройка Flask-COMBO-JSONAPI

```
pip install Flask-COMBO-JSONAPI
```

Описываем схему тега:

```
from marshmallow_jsonapi import Schema, fields

class TagSchema(Schema):
    class Meta:
        type_ = "tag"
        self_view = "tag_detail"
        self_view_kwargs = {"id": "<id>"}
        self_view_many = "tag_list"

    id = fields.Integer(as_string=True)
    name = fields.String(allow_none=False, required=True)
```

И импортируем в blog/schemas/__init__.py:

```
from blog.schemas.tag import TagSchema

__all__ = [
    "TagSchema",
]
```

Создаём первый API ресурс

Создаём модуль blog/api, там будем создавать модули с ресурсами.

Создаём модуль blog/api/tag.py и описываем стандартные ресурсы для list и detail. Необходимо указать схему и конфигурацию прослойки для работы с данными (SQLAlchemy): сессию и модель.

```
from flask_combo_jsonapi import ResourceDetail, ResourceList

from blog.schemas import TagSchema
from blog.models.database import db
from blog.models import Tag


class TagList(ResourceList):
    schema = TagSchema
    data_layer = {
        "session": db.session,
        "model": Tag,
    }

class TagDetail(ResourceDetail):
    schema = TagSchema
    data_layer = {
        "session": db.session,
        "model": Tag,
    }
```

Инициализируем API

В blog/api/__init__.py создаём новый объект api и подключаем ресурсы тега:

```
from flask_combo_jsonapi import Api

from blog.api.tag import TagList, TagDetail


def init_api(app):
    api = Api(app)

    api.route(TagList, "tag_list", "/api/tags/")
    api.route(TagDetail, "tag_detail", "/api/tags/<int:id>/")

    return api
```

Подключаем в blog/app.py:

```
from blog.api import init_api

api = init_api(app)
```

Теперь мы можем перейти на /api/tags/ для просмотра списка тегов:

```
{
   "data":[
      {
         "type":"tag",
         "id":"1",
         "attributes":{
            "name":"flask"
         }
      },
      {
         "type":"tag",
         "id":"2",
         "attributes":{
            "name":"django"
         }
      },
      {...}
   ],
   "links":{
      "self":"http://localhost/api/tags/",
      "first":"http://localhost/api/tags/",
      "last":"http://localhost/api/tags/?page%5Bnumber%5D=5",
      "next":"http://localhost/api/tags/?page%5Bnumber%5D=2"
   },
   "meta":{
      "count":5
   },
   "jsonapi":{
      "version":"1.0"
   }
}
```

А также на /api/tags/<int:id>/ для просмотра выбранного тега:

```
{
   "data":{
      "type":"tag",
      "id":"3",
      "attributes":{
         "name":"python"
      }
   },
   "jsonapi":{
      "version":"1.0"
   }
}
```

Устанавливаем ComboJSONAPI, ApiSpec и PyYAML

Необходимо установить следующие зависимости, в результате в requirements.txt будет

```
ComboJSONAPI==1.0.5
apispec==2.0.2
PyYAML==5.3.1
```

ComboJSONAPI предоставляет набор плагинов. Сейчас нас интересует плагин ApiSpec, при помощи которого мы получим автоматически сгенерированную Swagger документацию.

И конфигурируем плагины в blog/api/__init__.py:

```
from combojsonapi.spec import ApiSpecPlugin

def create_api_spec_plugin(app):
     api_spec_plugin = ApiSpecPlugin(
         app=app,
         # Declaring tags list with their descriptions,
         # so API gets organized into groups. it's optional.
         tags={
             "Tag": "Tag API",
         }
     )
     return api_spec_plugin


def init_api(app):
     api_spec_plugin = create_api_spec_plugin(app)
     api = Api(
         app,
         plugins=[
             api_spec_plugin,
         ],
     )


     api.route(TagList, "tag_list", "/api/tags/", tag="Tag")
     api.route(TagDetail, "tag_detail", "/api/tags/<int:id>/", tag="Tag")
```

Также необходимо добавить в конфигурацию blog/configs.py следующее:

```
OPENAPI_URL_PREFIX = '/api/swagger'
OPENAPI_SWAGGER_UI_PATH = '/'
OPENAPI_SWAGGER_UI_VERSION = '3.22.0'
```

Создаём схемы для оставшихся моделей

Обязательно указываем все связи при помощи объекта Relationship. Таким образом мы сможем подтягивать все зависимости в одном запросе.

blog/schemas/article.py:

```
from combojsonapi.utils import Relationship
from marshmallow_jsonapi import Schema, fields


class ArticleSchema(Schema):
   class Meta:
       type_ = "article"
       self_view = "article_detail"
       self_view_kwargs = {"id": "<id>"}
       self_view_many = "article_list"

   id = fields.Integer(as_string=True)
   title = fields.String(allow_none=False)
   body = fields.String(allow_none=False)
   dt_created = fields.DateTime(allow_none=False)
   dt_updated = fields.DateTime(allow_none=False)

   author = Relationship(
       nested="AuthorSchema",
       attribute="author",
       related_view="author_detail",
       related_view_kwargs={"id": "<id>"},
       schema="AuthorSchema",
       type_="author",
       many=False,
   )
   tags = Relationship(
       nested="TagSchema",
       attribute="tags",
       related_view="tag_detail",
       related_view_kwargs={"id": "<id>"},
       schema="TagSchema",
       type_="tag",
       many=True,
   )
```

blog/schemas/author.py:

```
from combojsonapi.utils import Relationship
from marshmallow_jsonapi import Schema, fields


class AuthorSchema(Schema):
   class Meta:
       type_ = "author"
       self_view = "author_detail"
       self_view_kwargs = {"id": "<id>"}
       self_view_many = "author_list"

   id = fields.Integer(as_string=True)

   user = Relationship(
       nested="UserSchema",
       attribute="user",
       related_view="user_detail",
       related_view_kwargs={"id": "<id>"},
       schema="UserSchema",
       type_="user",
       many=False,
   )

   articles = Relationship(
       nested="ArticleSchema",
       attribute="articles",
       related_view="article_detail",
       related_view_kwargs={"id": "<id>"},
       schema="ArticleSchema",
       type_="article",
       many=True,
   )
```

blog/schemas/user.py:

```
from combojsonapi.utils import Relationship
from marshmallow_jsonapi import Schema, fields


class UserSchema(Schema):
    class Meta:
        type_ = "user"
        self_view = "user_detail"
        self_view_kwargs = {"id": "<id>"}
        self_view_many = "user_list"

    id = fields.Integer(as_string=True)
    first_name = fields.String(allow_none=False)
    last_name = fields.String(allow_none=False)
    username = fields.String(allow_none=False)
    email = fields.String(allow_none=False)
    is_staff = fields.Boolean(allow_none=False)

    author = Relationship(
        nested="AuthorSchema",
        attribute="author",
        related_view="author_detail",
        related_view_kwargs={"id": "<id>"},
        schema="AuthorSchema",
        type_="author",
        many=False,
    )
```

Добавляем все схемы в blog/schemas/__init__.py:

```
from blog.schemas.tag import TagSchema
from blog.schemas.user import UserSchema
from blog.schemas.author import AuthorSchema
from blog.schemas.article import ArticleSchema

__all__ = [
    "TagSchema",
    "UserSchema",
    "AuthorSchema",
    "ArticleSchema",
]
```

Создаём все ресурсы для пользователя, автора и статей

Файл blog/api/article.py:

```
from flask_combo_jsonapi import ResourceDetail, ResourceList

from blog.schemas import ArticleSchema
from blog.models.database import db
from blog.models import Article


class ArticleList(ResourceList):
    schema = ArticleSchema
    data_layer = {
        "session": db.session,
        "model": Article,
    }


class ArticleDetail(ResourceDetail):
    schema = ArticleSchema
    data_layer = {
        "session": db.session,
        "model": Article,
    }
```

Файл blog/api/author.py:

```
from flask_combo_jsonapi import ResourceDetail, ResourceList

from blog.schemas import AuthorSchema
from blog.models.database import db
from blog.models import Author


class AuthorList(ResourceList):
    schema = AuthorSchema
    data_layer = {
        "session": db.session,
        "model": Author,
    }


class AuthorDetail(ResourceDetail):
    schema = AuthorSchema
    data_layer = {
        "session": db.session,
        "model": Author,
    }
```

Файл blog/api/user.py:

```
from flask_combo_jsonapi import ResourceDetail, ResourceList

from blog.schemas import UserSchema
from blog.models.database import db
from blog.models import User


class UserList(ResourceList):
    schema = UserSchema
    data_layer = {
        "session": db.session,
        "model": User,
    }


class UserDetail(ResourceDetail):
    schema = UserSchema
    data_layer = {
        "session": db.session,
        "model": User,
    }
```

Регистрируем все новые views

Добавляем новые views в файл blog/api/__init__.py.

Также добавляем теги и описание к ним.

```
from blog.api.user import UserList, UserDetail
from blog.api.author import AuthorList, AuthorDetail
from blog.api.article import ArticleList, ArticleDetail
...

def create_api_spec_plugin(app):
    api_spec_plugin = ApiSpecPlugin(
        ...
        tags={
            "Tag": "Tag API",
            "User": "User API",
            "Author": "Author API",
            "Article": "Article API",
        }
        ...

...


def init_api(app):
    ...

    api.route(UserList, "user_list", "/api/users/", tag="User")
    api.route(UserDetail, "user_detail", "/api/users/<int:id>/", tag="User")
    api.route(AuthorList, "author_list", "/api/authors/", tag="Author")
    api.route(AuthorDetail, "author_detail", "/api/authors/<int:id>/", tag="Author")
```

Теперь мы можем посетить /api/swagger/ и увидеть полный набор вызовов методов для всех моделей. Мы можем выполнять все CRUD действия, подтягивать зависимости, выполнять фильтрацию и так далее.

blog.app 1

/api/swagger/openapi.json

Tag Tag API

GET	/api/tags/
POST	/api/tags/
GET	/api/tags/{id}/
PATCH	/api/tags/{id}/
DELETE	/api/tags/{id}/

User User API

GET	/api/users/	
POST	/api/users/	
POST	/api/users/{id}/event_update_avatar/	Update user's avatar
GET	/api/users/{id}/	
PATCH	/api/users/{id}/	

Author Author API

GET	/api/authors/
POST	/api/authors/
GET	/api/authors/{id}/event_get_articles_count/
GET	/api/authors/{id}/
PATCH	/api/authors/{id}/
DELETE	/api/authors/{id}/

Article Article API

GET	/api/articles/event_get_count/
GET	/api/articles/
POST	/api/articles/
GET	/api/articles/{id}/
PATCH	/api/articles/{id}/
DELETE	/api/articles/{id}/

Итоги

На занятии мы добавили Flask-COMBO-JSONAPI в проект, создали схемы для моделей, ресурсы для моделей, подключили плагин ApiSpec и опробовали автоматически сгенерированные ресурсы.

Практическое задание

1. Добавить Flask-COMBO-JSONAPI в свой проект.
2. Создать схему, ResourceList и ResourceDetail для моделей.
3. Подключить плагин ApiSpec и опробовать автоматически сгенерированные ресурсы.

Дополнительные материалы

1. https://jsonapi.org/.
2. https://flask-combo-jsonapi.readthedocs.io/en/latest/.
3. https://marshmallow.readthedocs.io/en/latest/.
4. https://marshmallow-jsonapi.readthedocs.io/en/latest/.
5. https://github.com/AdCombo/combojsonapi/blob/master/docs/en/api_spec_plugin.rst.