OpenAPI para DRF com drf-spectacular

Meetup Grupy-RN - 10/09/2022

Vinicius Mendes

Senior Backend Engineer na Loadsmart

Anteriormente

- Solucione Sistemas (2008-2010)
- Globo.com (2010-2013)
- Dataprev (2013-2021)
- UFRN (2021)


loadsmart


MOVE
MORE
WITH
LESS

Agenda

- ➔ Por que Open API
- ➔ Visualizando a documentação
- ➔ Django Rest Framework
- ➔ Generate Schema
- ➔ drf-spectacular
- ➔ Customizações
- ➔ Geração de client
- ➔ Referências

Como você comunica o funcionamento da sua API para os seus clientes?

Clientes

- Time de frontend
- Time de mobile
- Outra equipe que depende do seu serviço
- Uma integração externa com um parceiro
- Cliente final

Algumas formas

Explico o funcionamento sempre que alguém precisa

Coloco no meu processo uma tarefa pra sempre atualizar a documentação

Adiciono um passo no meu CI para gerar a documentação automaticamente

Algumas formas

Adiciono um passo no meu CI para gerar a documentação automaticamente

Open API!

Define uma interface padrão para APIs REST, agnóstica a linguagens, para permitir que tanto humanos quanto computadores descubram e compreendam as capacidades do serviço sem precisar acessar código, documentação ou monitorar o tráfego de rede.

https://swagger.io/specification/

Open API!

Uma definição de OpenAPI pode ser utilizada para geração de documentação, clientes, servidores, ferramentas de testes e muitos outros casos de uso.

https://swagger.io/specification/

Show me the code!

https://editor-next.swagger.io/

OpenAPI Specification

Version 3.0.3

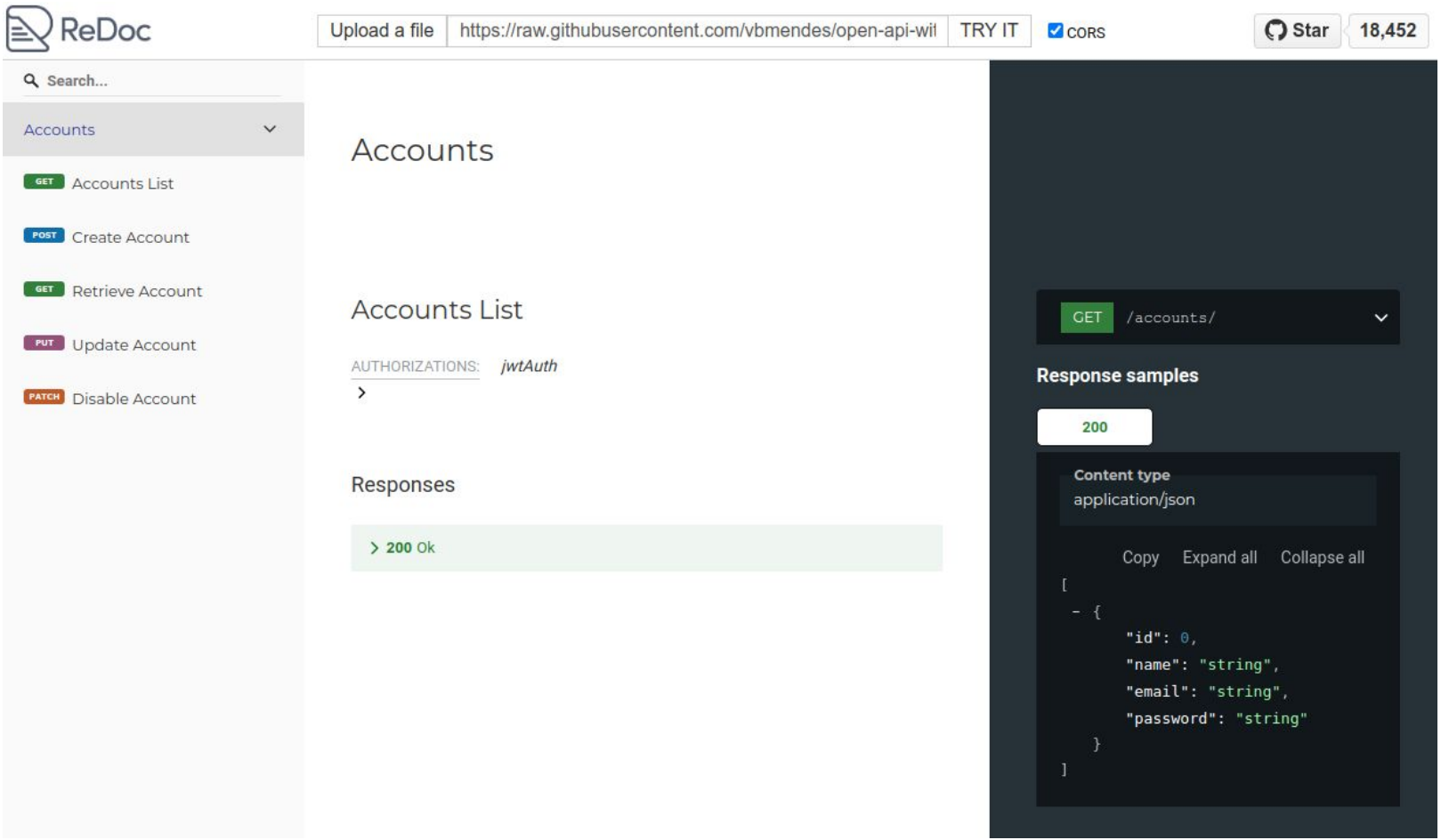
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 RFC2119 RFC8174 when, and only when, they appear in all capitals, as shown here.

Introduction

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.

https://swagger.io/specification/



https://redocly.github.io/redoc/?url=https://raw.githubusercontent.com/vbmendes/open-api-with-drf-spectacular/main/openapi.yaml

django REST framework

```python
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES


class SnippetSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    title = serializers.CharField(required=False, allow_blank=True, max_length=100)
    code = serializers.CharField(style={'base_template': 'textarea.html'})
    linenos = serializers.BooleanField(required=False)
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES, default='python')
    style = serializers.ChoiceField(choices=STYLE_CHOICES, default='friendly')

    def create(self, validated_data):
        """
        Create and return a new `Snippet` instance, given the validated data.
        """
        return Snippet.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        Update and return an existing `Snippet` instance, given the validated data.
        """
        instance.title = validated_data.get('title', instance.title)
        instance.code = validated_data.get('code', instance.code)
        instance.linenos = validated_data.get('linenos', instance.linenos)
        instance.language = validated_data.get('language', instance.language)
        instance.style = validated_data.get('style', instance.style)
        instance.save()
        return instance
```

schemas

Schema

> *A machine-readable [schema] describes what resources are available via the API, what their URLs are, how they are represented and what operations they support.*
>
> — Heroku, *JSON Schema for the Heroku Platform API*

API schemas are a useful tool that allow for a range of use cases, including generating reference documentation, or driving dynamic client libraries that can interact with your API.

Django REST Framework provides support for automatic generation of OpenAPI schemas.

Overview

Schema generation has several moving parts. It's worth having an overview:

- `SchemaGenerator` is a top-level class that is responsible for walking your configured URL patterns, finding `APIView` subclasses, enquiring for their schema representation, and compiling the final schema object.
- `AutoSchema` encapsulates all the details necessary for per-view schema introspection. Is attached to each view via the `schema` attribute. You subclass `AutoSchema` in order to customize your schema.
- The `generateschema` management command allows you to generate a static schema offline.
- Alternatively, you can route `SchemaView` to dynamically generate and serve your schema.
- `settings.DEFAULT_SCHEMA_CLASS` allows you to specify an `AutoSchema` subclass to serve as your project's default.

The following sections explain more.

Generating an OpenAPI Schema

Generating a static schema with the `generateschema` management command

If your schema is static, you can use the `generateschema` management command:

```
./manage.py generateschema --file openapi-schema.yml
```

Show me the code!

https://editor-next.swagger.io/

drf-spectacular

Ao utilizar drf-spectacular com DRF, seu schema e portanto sua documentação e clientes estarão sempre próximos à sua API.

https://swagger.io/specification/

drf-spectacular

Funciona bem out of the box, mas também disponibiliza várias formas de customizar o schema OpenAPI 3 gerado.

Desenvolvido para funcionar bem com SwaggerUI, ReDoc e geração de clientes automáticos.

https://swagger.io/specification/

Installation

Install using `pip` ...

```
$ pip install drf-spectacular
```

then add drf-spectacular to installed apps in `settings.py`

```
INSTALLED_APPS = [
    # ALL YOUR APPS
    'drf_spectacular',
]
```

and finally register our spectacular AutoSchema with DRF.

```
REST_FRAMEWORK = {
    # YOUR SETTINGS
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
}
```

drf-spectacular ships with sane default settings that should work reasonably well out of the box. It is not necessary to specify any settings, but we recommend to specify at least some metadata.

```
SPECTACULAR_SETTINGS = {
    'TITLE': 'Your Project API',
    'DESCRIPTION': 'Your project description',
    'VERSION': '1.0.0',
    'SERVE_INCLUDE_SCHEMA': False,
    # OTHER SETTINGS
}
```

Take it for a spin

Generate your schema with the CLI:

```
$ ./manage.py spectacular --file schema.yml
$ docker run -p 80:8080 -e SWAGGER_JSON=/schema.yml -v ${PWD}/schema.yml:/schema.yml swaggerapi
```

If you also want to validate your schema add the `--validate` flag. Or serve your schema directly from your API. We also provide convenience wrappers for `swagger-ui` or `redoc`.

```
from drf_spectacular.views import SpectacularAPIView, SpectacularRedocView, SpectacularSwaggerV
urlpatterns = [
    # YOUR PATTERNS
    path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
    # Optional UI:
    path('api/schema/swagger-ui/', SpectacularSwaggerView.as_view(url_name='schema'), name='swa
    path('api/schema/redoc/', SpectacularRedocView.as_view(url_name='schema'), name='redoc'),
]
```

Customizações

- ➔ queryset e serializer_class
- ➔ @extend_schema
- ➔ @extend_schema_field e type hints
- ➔ @extend_schema_serailizer
- ➔ Extensões
- ➔ Postprocessing hooks
- ➔ Preprocessing hooks

@extend_schema

- operation_id
- parameters
- responses
- request
- auth
- description
- summary
- deprecated
- tags
- filters
- exclude

- operation (sobrescrita manual)
- methods
- versions
- examples
- extensions
- callbacks
- external_docs

```python
@extend_schema(
    # extra parameters added to the schema
    parameters=[
        OpenApiParameter(name='artist', description='Filter by artist', required=False, type=st
        OpenApiParameter(
            name='release',
            type=OpenApiTypes.DATE,
            location=OpenApiParameter.QUERY,
            description='Filter by release date',
            examples=[
                OpenApiExample(
                    'Example 1',
                    summary='short optional summary',
                    description='longer description',
                    value='1993-08-23'
                ),
                ...
            ],
        ),
    ],
    # override default docstring extraction
    description='More descriptive text',
    # provide Authentication class that deviates from the views default
    auth=None,
    # change the auto-generated operation name
    operation_id=None,
    # or even completely override what AutoSchema would generate. Provide raw Open API spec as
    operation=None,
    # attach request/response examples to the operation.
    examples=[
        OpenApiExample(
            'Example 1',
            description='longer description',
            value=...
        ),
        ...
    ],
)
def list(self, request):
    # your non-standard behaviour
    return super().list(request)
```

Step 3: `@extend_schema_field` and type hints

A custom `SerializerField` might not get picked up properly. You can inform *drf-spectacular* on what is to be expected with the `@extend_schema_field` decorator. It takes either basic types or a `Serializer` as argument. In case of basic types (e.g. `str`, `int`, etc.) a type hint is already sufficient.

```
@extend_schema_field(OpenApiTypes.BYTE)  # also takes basic python types
class CustomField(serializers.Field):
    def to_representation(self, value):
        return urlsafe_base64_encode(b'\xf0\xf1\xf2')
```

You can apply it also to the method of a `SerializerMethodField`.

```
class ErrorDetailSerializer(serializers.Serializer):
    field_custom = serializers.SerializerMethodField()

    @extend_schema_field(OpenApiTypes.DATETIME)
    def get_field_custom(self, object):
        return '2020-03-06 20:54:00.104248'
```

Step 4: `@extend_schema_serializer`

You may also decorate your serializer with `@extend_schema_serializer`. Mainly used for excluding specific fields from the schema or attaching request/response examples. On rare occasions (e.g. envelope serializers), overriding list detection with `many=False` may come in handy.

```
@extend_schema_serializer(
    exclude_fields=('single',), # schema ignore these fields
    examples = [
         OpenApiExample(
            'Valid example 1',
            summary='short summary',
            description='longer description',
            value={
                'songs': {'top10': True},
                'single': {'top10': True}
            },
            request_only=True, # signal that example only applies to requests
            response_only=False, # signal that example only applies to responses
        ),
    ]
)
class AlbumSerializer(serializers.ModelSerializer):
    songs = SongSerializer(many=True)
    single = SongSerializer(read_only=True)

    class Meta:
        fields = '__all__'
        model = Album
```

Extensões

- Possibilita customizações em códigos de terceiros.
- OpenApiViewExtension
- OpenApiAuthenticationExtension
- OpenApiSerializerFieldExtension
- OpenApiSerializerExtension
- OpenApiFilterExtension

```python
class Fix4(OpenApiViewExtension):
    target_class = 'oscarapi.views.checkout.UserAddressDetail'

    def view_replacement(self):
        from oscar.apps.address.models import UserAddress

        class Fixed(self.target_class):
            queryset = UserAddress.objects.none()
        return Fixed
```

Step 6: Postprocessing hooks

The generated schema is still not to your liking? You are no easy customer, but there is one more thing you can do. Postprocessing hooks run at the very end of schema generation. This is how the choice `Enum` are consolidated into component objects. You can register additional hooks with the `POSTPROCESSING_HOOKS` setting.

```
def custom_postprocessing_hook(result, generator, request, public):
    # your modifications to the schema in parameter result
    return result
```

Step 7: Preprocessing hooks

Preprocessing hooks are applied shortly after collecting all API operations and before the actual schema generation starts. They provide an easy mechanism to alter which operations should be represented in your schema. You can exclude specific operations, prefix paths, introduce or hardcode path parameters or modify view initiation. additional hooks with the `PREPROCESSING_HOOKS` setting.

```
def custom_preprocessing_hook(endpoints):
    # your modifications to the list of operations that are exposed in the schema
    for (path, path_regex, method, callback) in endpoints:
        pass
    return endpoints
```

Gerando clientes

- Evite warnings
- COMPONENT_SPLIT_REQUEST
- ENUM_ADD_EXPLICIT_BLANK_NULL_CHOICE
- Escolha o gerador adequado para a stack do seu cliente
  - openapi-python-client
  - react-openapi-client
  - ng-openapi-gen
  - Android? Flutter? Swift? ...

Referências

- https://spec.openapis.org/oas/v3.1.0
- https://swagger.io/specification/
- https://www.django-rest-framework.org/
- https://drf-spectacular.readthedocs.io/
- https://github.com/Redocly/redoc

Obrigado!

https://twitter.com/vbmendes

https://www.linkedin.com/in/viniciusmendes/

https://github.com/vbmendes