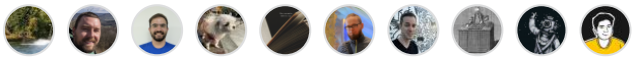



# drf-spectacular


Sane and flexible OpenAPI 3.0 schema generation for Django REST framework.

***This project has 3 goals:***

1. Extract as much schema information from DRF as possible.
2. Provide flexibility to make the schema usable in the real world (not only toy examples).
3. Generate a schema that works well with the most popular client generators.

The code is a heavily modified fork of the DRF OpenAPI generator, which is/was lacking all of the below listed features.

***Features***

- Serializers modelled as components. (arbitrary nesting and recursion supported)
- ***@extend_schema decorator for customization of APIView, Viewsets, function-based views, and `@action`***
  - additional parameters
  - request/response serializer override (with status codes)

    - polymorphic responses either manually with `PolymorphicProxySerializer` helper or via `rest_polymorphic` 's PolymorphicSerializer)
    - ... and more customization options

- Authentication support (DRF natives included, easily extendable)
- Custom serializer class support (easily extendable)
- `SerializerMethodField()` type via type hinting or `@extend_schema_field`
- i18n support
- Tags extraction
- Request/response/parameter examples
- Description extraction from `docstrings`
- Vendor specification extensions ( `x-*` ) in info, operations, parameters, components, and security schemes
- Sane fallbacks
- Sane `operation_id` naming (based on path)
- Schema serving with `SpectacularAPIView` (Redoc and Swagger-UI views are also available)
- Optional input/output serializer component split
- ***Included support for:***
    - django-polymorphic / django-rest-polymorphic
    - SimpleJWT
    - DjangoOAuthToolkit
    - djangorestframework-jwt (tested fork drf-jwt)
    - dj-rest-auth (maintained fork of django-rest-auth)
    - djangorestframework-camel-case (via postprocessing hook `camelize_serializer_fields` )
    - django-filter
    - drf-nested-routers

For more information visit the documentation.

## License

Provided by T. Franzel, Cashlink Technologies GmbH. Licensed under 3-Clause BSD.

## Requirements

- Python >= 3.6
- Django (2.2, 3.1, 3.2)
- Django REST Framework (3.10, 3.11, 3.12)

## 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',
    # OTHER SETTINGS
}
```

### Self-contained UI installation

Certain environments have no direct access to the internet and as such are unable to retrieve Swagger UI or Redoc from CDNs. drf-spectacular-sidecar provides these static files as a separate optional package. Usage is as follows:

```
$ pip install drf-spectacular[sidecar]
```

```
INSTALLED_APPS = [
    # ALL YOUR APPS
    'drf_spectacular',
    'drf_spectacular_sidecar',  # required for Django collectstatic discovery
]
SPECTACULAR_SETTINGS = {
    'SWAGGER_UI_DIST': 'SIDECAR',  # shorthand to use the sidecar instead
    'SWAGGER_UI_FAVICON_HREF': 'SIDECAR',
    'REDOC_DIST': 'SIDECAR',
    # OTHER SETTINGS
}
```

### Release management

drf-spectacular deliberately stays below version `1.x.x` to signal that every new version may potentially break you. For production we strongly recommend pinning the version and inspecting a schema diff on update.

With that said, we aim to be extremely defensive w.r.t. breaking API changes. However, we also acknowledge the fact that even slight schema changes may break your toolchain, as any existing bug may somehow also be used as a feature.

We define version increments with the following semantics. y-stream increments may contain potentially breaking changes to both API and schema. z-stream increments will never break the API and may only contain schema changes that should have a low chance of breaking you.

## 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:/sch
```

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, S
urlpatterns = [
    # YOUR PATTERNS
    path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
    # Optional UI:
    path('api/schema/swagger-ui/', SpectacularSwaggerView.as_view(url_name='s
    path('api/schema/redoc/', SpectacularRedocView.as_view(url_name='schema')
]
```

## Usage

drf-spectacular works pretty well out of the box. You might also want to set some metadata for your API. Just create a `SPECTACULAR_SETTINGS` dictionary in your `settings.py` and override the defaults. Have a look at the available settings.

The toy examples do not cover your cases? No problem, you can heavily customize how your schema will be rendered.

### Customization by using `@extend_schema`

Most customization cases should be covered by the `extend_schema` decorator. We usually get pretty far with specifying `OpenApiParameter` and splitting request/response serializers, but the sky is the limit.

```
from drf_spectacular.utils import extend_schema, OpenApiParameter, OpenApiExa
from drf_spectacular.types import OpenApiTypes

class AlbumViewset(viewset.ModelViewset)
    serializer_class = AlbumSerializer

    @extend_schema(
        request=AlbumCreationSerializer
        responses={201: AlbumSerializer},
    )
    def create(self, request):
        # your non-standard behaviour
        return super().create(request)

    @extend_schema(
        # extra parameters added to the schema
        parameters=[
            OpenApiParameter(name='artist', description='Filter by artist', r
```

```
                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
            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)

        @extend_schema(
            request=AlbumLikeSerializer
            responses={204: None},
            methods=["POST"]
        )
        @extend_schema(description='Override a specific method', methods=["GET"])
        @action(detail=True, methods=['post', 'get'])
        def set_password(self, request, pk=None):
            # your action behaviour
```

## More customization

Still not satisifed? You want more! We still got you covered. Visit customization for more information.

## Testing

Install testing requirements.

```
$ pip install -r requirements.txt
```

Run with runtests.

```
$ ./runtests.py
```

You can also use the excellent tox testing tool to run the tests against all supported versions of Python and Django. Install tox globally, and then simply run:

```
$ tox
```