openapi-core

Release 0.16.6

Artur Maciag

Mar 02, 2023

CONTENTS

Openapi-core is a Python library that adds client-side and server-side support for the OpenAPI v3.0 and OpenAPI v3.1 specification.

CHAPTER

ONE

KEY FEATURES

- **Validation** of requests and responses
- Schema **casting** and **unmarshalling**
- Media type and parameters **deserialization**
- **Security** providers (API keys, Cookie, Basic and Bearer HTTP authentications)
- Custom **deserializers** and **formats**
- **Integration** with libraries and frameworks

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Installation

Recommended way (via pip):

```
$ pip install openapi-core
```

Alternatively you can download the code and install from the repository:

```
$ pip install -e git+https://github.com/p1c2u/openapi-core.git#egg=openapi_core
```

2.2 Usage

Firstly create your specification object. By default, OpenAPI spec version is detected:

```
from json import load
from openapi_core import Spec

with open('openapi.json', 'r') as spec_file:
   spec_dict = load(spec_file)

spec = Spec.create(spec_dict)
```

2.2.1 Request

Now you can use it to validate against requests

```
from openapi_core import openapi_request_validator

result = openapi_request_validator.validate(spec, request)

# raise errors if request invalid
result.raise_for_errors()

# get list of errors
errors = result.errors
```

and unmarshal request data from validation result

```
# get parameters object with path, query, cookies and headers parameters
validated_params = result.parameters
# or specific location parameters
validated_path_params = result.parameters.path

# get body
validated_body = result.body

# get security data
validated_security = result.security
```

Request object should implement OpenAPI Request protocol (See *Integrations*).

2.2.2 Response

You can also validate against responses

```
from openapi_core import openapi_response_validator

result = openapi_response_validator.validate(spec, request, response)

# raise errors if response invalid
result.raise_for_errors()

# get list of errors
errors = result.errors
```

and unmarshal response data from validation result

```
# get headers
validated_headers = result.headers

# get data
validated_data = result.data
```

Response object should implement OpenAPI Response protocol (See *Integrations*).

2.2.3 Security

openapi-core supports security for authentication and authorization process. Security data for security schemas are accessible from *security* attribute of *RequestValidationResult* object.

For given security specification:

```
security:
  - BasicAuth: []
  - ApiKeyAuth: []
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
```

(continues on next page)

(continued from previous page)

```
    ApiKeyAuth:
      type: apiKey
      in: header
      name: X-API-Key
```

you can access your security data the following:

```
result = validator.validate(request)

# get basic auth decoded credentials
result.security['BasicAuth']

# get api key
result.security['ApiKeyAuth']
```

Supported security types:

- http – for Basic and Bearer HTTP authentications schemes
- apiKey – for API keys and cookie authentication

2.3 Integrations

2.3.1 Bottle

See bottle-openapi-3 project.

2.3.2 Django

This section describes integration with Django web framework. The integration supports Django from version 3.0 and above.

Middleware

Django can be integrated by middleware. Add `DjangoOpenAPIMiddleware` to your `MIDDLEWARE` list and define `OPENAPI_SPEC`.

```
# settings.py
from openapi_core import Spec

MIDDLEWARE = [
    # ...
    'openapi_core.contrib.django.middlewares.DjangoOpenAPIMiddleware',
]

OPENAPI_SPEC = Spec.create(spec_dict)
```

After that you have access to validation result object with all validated request data from Django view through request object.

```
from django.views import View

class MyView(View):
    def get(self, req):
        # get parameters object with path, query, cookies and headers parameters
        validated_params = req.openapi.parameters
        # or specific location parameters
        validated_path_params = req.openapi.parameters.path

        # get body
        validated_body = req.openapi.body

        # get security data
        validated_security = req.openapi.security
```

Low level

You can use `DjangoOpenAPIRequest` as a Django request factory:

```
from openapi_core import openapi_request_validator
from openapi_core.contrib.django import DjangoOpenAPIRequest

openapi_request = DjangoOpenAPIRequest(django_request)
result = openapi_request_validator.validate(spec, openapi_request)
```

You can use `DjangoOpenAPIResponse` as a Django response factory:

```
from openapi_core import openapi_response_validator
from openapi_core.contrib.django import DjangoOpenAPIResponse

openapi_response = DjangoOpenAPIResponse(django_response)
result = openapi_response_validator.validate(spec, openapi_request, openapi_response)
```

2.3.3 Falcon

This section describes integration with Falcon web framework. The integration supports Falcon from version 3.0 and above.

Middleware

The Falcon API can be integrated by `FalconOpenAPIMiddleware` middleware.

```
from openapi_core.contrib.falcon.middlewares import FalconOpenAPIMiddleware

openapi_middleware = FalconOpenAPIMiddleware.from_spec(spec)
app = falcon.App(middleware=[openapi_middleware])
```

After that you will have access to validation result object with all validated request data from Falcon view through request context.

```
class ThingsResource:
    def on_get(self, req, resp):
        # get parameters object with path, query, cookies and headers parameters
        validated_params = req.context.openapi.parameters
        # or specific location parameters
        validated_path_params = req.context.openapi.parameters.path

        # get body
        validated_body = req.context.openapi.body

        # get security data
        validated_security = req.context.openapi.security
```

Low level

You can use `FalconOpenAPIRequest` as a Falcon request factory:

```
from openapi_core import openapi_request_validator
from openapi_core.contrib.falcon import FalconOpenAPIRequest

openapi_request = FalconOpenAPIRequest(falcon_request)
result = openapi_request_validator.validate(spec, openapi_request)
```

You can use `FalconOpenAPIResponse` as a Falcon response factory:

```
from openapi_core import openapi_response_validator
from openapi_core.contrib.falcon import FalconOpenAPIResponse

openapi_response = FalconOpenAPIResponse(falcon_response)
result = openapi_response_validator.validate(spec, openapi_request, openapi_response)
```

2.3.4 Flask

This section describes integration with Flask web framework.

Decorator

Flask views can be integrated by `FlaskOpenAPIViewDecorator` decorator.

```
from openapi_core.contrib.flask.decorators import FlaskOpenAPIViewDecorator

openapi = FlaskOpenAPIViewDecorator.from_spec(spec)

@app.route('/home')
@openapi
def home():
    pass
```

If you want to decorate class based view you can use the decorators attribute:

```
class MyView(View):
    decorators = [openapi]
```

View

As an alternative to the decorator-based integration, a Flask method based views can be integrated by inheritance from `FlaskOpenAPIView` class.

```
from openapi_core.contrib.flask.views import FlaskOpenAPIView

class MyView(FlaskOpenAPIView):
    pass

app.add_url_rule('/home', view_func=MyView.as_view('home', spec))
```

Request parameters

In Flask, all unmarshalled request data are provided as Flask request object's `openapi.parameters` attribute

```
from flask.globals import request

@app.route('/browse/<id>/')
@openapi
def home():
    browse_id = request.openapi.parameters.path['id']
    page = request.openapi.parameters.query.get('page', 1)
```

Low level

You can use `FlaskOpenAPIRequest` as a Flask request factory:

```
from openapi_core import openapi_request_validator
from openapi_core.contrib.flask import FlaskOpenAPIRequest

openapi_request = FlaskOpenAPIRequest(flask_request)
result = openapi_request_validator.validate(spec, openapi_request)
```

For response factory see *Werkzeug* integration.

2.3.5 Pyramid

See pyramid_openapi3 project.

2.3.6 Requests

This section describes integration with Requests library.

Low level

You can use `RequestsOpenAPIRequest` as a Requests request factory:

```
from openapi_core import openapi_request_validator
from openapi_core.contrib.requests import RequestsOpenAPIRequest

openapi_request = RequestsOpenAPIRequest(requests_request)
result = openapi_request_validator.validate(spec, openapi_request)
```

You can use `RequestsOpenAPIResponse` as a Requests response factory:

```
from openapi_core import openapi_response_validator
from openapi_core.contrib.requests import RequestsOpenAPIResponse

openapi_response = RequestsOpenAPIResponse(requests_response)
result = openapi_response_validator.validate(spec, openapi_request, openapi_response)
```

2.3.7 Starlette

This section describes integration with Starlette ASGI framework.

Low level

You can use `StarletteOpenAPIRequest` as a Starlette request factory:

```
from openapi_core import openapi_request_validator
from openapi_core.contrib.starlette import StarletteOpenAPIRequest

openapi_request = StarletteOpenAPIRequest(starlette_request)
result = openapi_request_validator.validate(spec, openapi_request)
```

You can use `StarletteOpenAPIResponse` as a Starlette response factory:

```
from openapi_core import openapi_response_validator
from openapi_core.contrib.starlette import StarletteOpenAPIResponse

openapi_response = StarletteOpenAPIResponse(starlette_response)
result = openapi_response_validator.validate(spec, openapi_request, openapi_response)
```

2.3.8 Tornado

See tornado-openapi3 project.

2.3.9 Werkzeug

This section describes integration with Werkzeug a WSGI web application library.

Low level

You can use `WerkzeugOpenAPIRequest` as a Werkzeug request factory:

```
from openapi_core import openapi_request_validator
from openapi_core.contrib.werkzeug import WerkzeugOpenAPIRequest

openapi_request = WerkzeugOpenAPIRequest(werkzeug_request)
result = openapi_request_validator.validate(spec, openapi_request)
```

You can use `WerkzeugOpenAPIResponse` as a Werkzeug response factory:

```
from openapi_core import openapi_response_validator
from openapi_core.contrib.werkzeug import WerkzeugOpenAPIResponse

openapi_response = WerkzeugOpenAPIResponse(werkzeug_response)
result = openapi_response_validator.validate(spec, openapi_request, openapi_response)
```

2.4 Customizations

2.4.1 Spec validation

By default, spec dict is validated on spec creation time. Disabling the validator can improve the performance.

```
from openapi_core import Spec

spec = Spec.create(spec_dict, validator=None)
```

2.4.2 Deserializers

Pass custom defined media type deserializers dictionary with supported mimetypes as a key to *MediaTypeDeserializersFactory* and then pass it to *RequestValidator* or *ResponseValidator* constructor:

```
from openapi_core.deserializing.media_types.factories import␣
↪MediaTypeDeserializersFactory
from openapi_core.unmarshalling.schemas import oas30_response_schema_unmarshallers_
↪factory

def protobuf_deserializer(message):
    feature = route_guide_pb2.Feature()
```

(continues on next page)

(continued from previous page)

```
    feature.ParseFromString(message)
    return feature

custom_media_type_deserializers = {
    'application/protobuf': protobuf_deserializer,
}
media_type_deserializers_factory = MediaTypeDeserializersFactory(
    custom_deserializers=custom_media_type_deserializers,
)

validator = ResponseValidator(
    oas30_response_schema_unmarshallers_factory,
    media_type_deserializers_factory=media_type_deserializers_factory,
)

result = validator.validate(spec, request, response)
```

2.4.3 Formats

OpenAPI defines a `format` keyword that hints at how a value should be interpreted, e.g. a `string` with the type `date` should conform to the RFC 3339 date format.

Openapi-core comes with a set of built-in formatters, but it's also possible to add custom formatters in *SchemaUnmarshallersFactory* and pass it to *RequestValidator* or *ResponseValidator*.

Here's how you could add support for a `usdate` format that handles dates of the form MM/DD/YYYY:

```
from openapi_core.unmarshalling.schemas.factories import SchemaUnmarshallersFactory
from openapi_schema_validator import OAS30Validator
from datetime import datetime
import re

class USDateFormatter:
    def validate(self, value) -> bool:
        return bool(re.match(r"^\d{1,2}/\d{1,2}/\d{4}$", value))

    def format(self, value):
        return datetime.strptime(value, "%m/%d/%y").date


custom_formatters = {
    'usdate': USDateFormatter(),
}
schema_unmarshallers_factory = SchemaUnmarshallersFactory(
    OAS30Validator,
    custom_formatters=custom_formatters,
    context=UnmarshalContext.RESPONSE,
)
validator = ResponseValidator(schema_unmarshallers_factory)

result = validator.validate(spec, request, response)
```

2.5 Extensions

2.5.1 x-model

By default, objects are unmarshalled to dictionaries. You can use dynamically created dataclasses.

```
...
components:
  schemas:
    Coordinates:
      x-model: Coordinates
      type: object
      required:
        - lat
        - lon
      properties:
        lat:
          type: number
        lon:
          type: number
```

You can use your own dataclasses, pydantic models or models generated by third party generators (i.e. datamodel-code-generator) by providing `x-model-path` property inside schema definition with location of your class.

```
...
components:
  schemas:
    Coordinates:
      x-model-path: foo.bar.Coordinates
      type: object
      required:
        - lat
        - lon
      properties:
        lat:
          type: number
        lon:
          type: number
```

```
# foo/bar.py
from dataclasses import dataclass

@dataclass
class Coordinates:
    lat: float
    lon: float
```

CHAPTER

THREE

RELATED PROJECTS

- **openapi-spec-validator**
  Python library that validates OpenAPI Specs against the OpenAPI 2.0 (aka Swagger) and OpenAPI 3.0.0 specification. The validator aims to check for full compliance with the Specification.
- **openapi-schema-validator**
  Python library that validates schema against the OpenAPI Schema Specification v3.0 which is an extended subset of the JSON Schema Specification Wright Draft 00.