# openapi-core

***Release 0.18.0***

**Artur Maciag**

**Jul 10, 2023**

# CONTENTS

CHAPTER

ONE

# UNMARSHALLING

Unmarshalling is the process of converting a primitive schema type of value into a higher-level object based on a `format` keyword. All request/response data, that can be described by a schema in OpenAPI specification, can be unmarshalled.

Unmarshallers firstly validate data against the provided schema (See *Validation*).

Openapi-core comes with a set of built-in format unmarshallers:

- `date` - converts string into a date object,
- `date-time` - converts string into a datetime object,
- `binary` - converts string into a byte object,
- `uuid` - converts string into an UUID object,
- `byte` - decodes Base64-encoded string.

You can also define your own format unmarshallers (See *Customizations*).

## 1.1 Request unmarshalling

Use `unmarshal_request` function to validate and unmarshal request data against a given spec. By default, OpenAPI spec version is detected:

```
from openapi_core import unmarshal_request

# raises error if request is invalid
result = unmarshal_request(request, spec=spec)
```

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

**Note**

Webhooks feature is part of OpenAPI v3.1 only

Use the same function to validate and unmarshal webhook request data against a given spec.

```
# raises error if request is invalid
result = unmarshal_request(webhook_request, spec=spec)
```

Webhook request object should implement OpenAPI WebhookRequest protocol (See *Integrations*).

Retrieve validated and unmarshalled request data

```
# get parameters
path_params = result.parameters.path
query_params = result.parameters.query
cookies_params = result.parameters.cookies
headers_params = result.parameters.headers
# get body
body = result.body
# get security data
security = result.security
```

In order to explicitly validate and unmarshal a:

- OpenAPI 3.0 spec, import `V30RequestUnmarshaller`
- OpenAPI 3.1 spec, import `V31RequestUnmarshaller` or `V31WebhookRequestUnmarshaller`

```
from openapi_core import V31RequestUnmarshaller

result = unmarshal_request(
    request, response,
    spec=spec,
    cls=V31RequestUnmarshaller,
)
```

You can also explicitly import `V3RequestUnmarshaller` which is a shortcut to the latest OpenAPI v3 version.

## 1.2 Response unmarshalling

Use `unmarshal_response` function to validate and unmarshal response data against a given spec. By default, OpenAPI spec version is detected:

```
from openapi_core import unmarshal_response

# raises error if response is invalid
result = unmarshal_response(request, response, spec=spec)
```

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

---

**Note**

Webhooks feature is part of OpenAPI v3.1 only

---

Use the same function to validate and unmarshal response data from webhook request against a given spec.

```
# raises error if request is invalid
result = unmarshal_response(webhook_request, response, spec=spec)
```

Retrieve validated and unmarshalled response data

```
# get headers
headers = result.headers
# get data
data = result.data
```

In order to explicitly validate and unmarshal a:

- OpenAPI 3.0 spec, import `V30ResponseUnmarshaller`
- OpenAPI 3.1 spec, import `V31ResponseUnmarshaller` or `V31WebhookResponseUnmarshaller`

```
from openapi_core import V31ResponseUnmarshaller

result = unmarshal_response(
    request, response,
    spec=spec,
    cls=V31ResponseUnmarshaller,
)
```

You can also explicitly import `V3ResponseUnmarshaller` which is a shortcut to the latest OpenAPI v3 version.

CHAPTER

TWO

# VALIDATION

Validation is a process to validate request/response data under a given schema defined in OpenAPI specification.

Additionally, openapi-core uses the `format` keyword to check if primitive types conform to defined formats.

Such valid formats can be forther unmarshalled (See *Unmarshalling*).

Depends on the OpenAPI version, openapi-core comes with a set of built-in format validators such as: `date`, `date-time`, `binary`, `uuid` or `byte`.

You can also define your own format validators (See *Customizations*).

## 2.1 Request validation

Use `validate_request` function to validate request data against a given spec. By default, OpenAPI spec version is detected:

```
from openapi_core import validate_request

# raises error if request is invalid
validate_request(request, spec=spec)
```

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

**Note**

Webhooks feature is part of OpenAPI v3.1 only

Use the same function to validate webhook request data against a given spec.

```
# raises error if request is invalid
validate_request(webhook_request, spec=spec)
```

Webhook request object should implement OpenAPI WebhookRequest protocol (See *Integrations*).

In order to explicitly validate and unmarshal a:

- OpenAPI 3.0 spec, import `V30RequestValidator`
- OpenAPI 3.1 spec, import `V31RequestValidator` or `V31WebhookRequestValidator`

```
from openapi_core import V31RequestValidator

validate_request(
    request, response,
    spec=spec,
    cls=V31RequestValidator,
)
```

You can also explicitly import `V3RequestValidator` which is a shortcut to the latest OpenAPI v3 version.

## 2.2 Response validation

Use `validate_response` function to validate response data against a given spec. By default, OpenAPI spec version is detected:

```
from openapi_core import validate_response

# raises error if response is invalid
validate_response(request, response, spec=spec)
```

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

**Note**

Webhooks feature is part of OpenAPI v3.1 only

Use the same function to validate response data from webhook request against a given spec.

```
# raises error if request is invalid
validate_response(webhook_request, response, spec=spec)
```

In order to explicitly validate a:

- OpenAPI 3.0 spec, import `V30ResponseValidator`
- OpenAPI 3.1 spec, import `V31ResponseValidator` or `V31WebhookResponseValidator`

```
from openapi_core import V31ResponseValidator

validate_response(
    request, response,
    spec=spec,
    cls=V31ResponseValidator,
)
```

You can also explicitly import `V3ResponseValidator` which is a shortcut to the latest OpenAPI v3 version.

CHAPTER

THREE

# INTEGRATIONS

Openapi-core integrates with your popular libraries and frameworks. Each integration offers different levels of integration that help validate and unmarshal your request and response data.

## 3.1 aiohttp.web

This section describes integration with aiohttp.web framework.

### 3.1.1 Low level

You can use `AIOHTTPOpenAPIWebRequest` as an aiohttp request factory:

```
from openapi_core import unmarshal_request
from openapi_core.contrib.aiohttp import AIOHTTPOpenAPIWebRequest

request_body = await aiohttp_request.text()
openapi_request = AIOHTTPOpenAPIWebRequest(aiohttp_request, body=request_body)
result = unmarshal_request(openapi_request, spec=spec)
```

You can use `AIOHTTPOpenAPIWebRequest` as an aiohttp response factory:

```
from openapi_core import unmarshal_response
from openapi_core.contrib.starlette import AIOHTTPOpenAPIWebRequest

openapi_response = StarletteOpenAPIResponse(aiohttp_response)
result = unmarshal_response(openapi_request, openapi_response, spec=spec)
```

## 3.2 Bottle

See bottle-openapi-3 project.

## 3.3 Django

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

### 3.3.1 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.from_dict(spec_dict)
```

After that you have access to unmarshal 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
```

### 3.3.2 Low level

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

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

openapi_request = DjangoOpenAPIRequest(django_request)
result = unmarshal_request(openapi_request, spec=spec)
```

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

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

openapi_response = DjangoOpenAPIResponse(django_response)
result = unmarshal_response(openapi_request, openapi_response, spec=spec)
```

## 3.4 Falcon

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

### 3.4.1 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
```

### 3.4.2 Low level

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

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

openapi_request = FalconOpenAPIRequest(falcon_request)
result = unmarshal_request(openapi_request, spec=spec)
```

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

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

openapi_response = FalconOpenAPIResponse(falcon_response)
result = unmarshal_response(openapi_request, openapi_response, spec=spec)
```

## 3.5 Flask

This section describes integration with Flask web framework.

### 3.5.1 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():
    return "Welcome home"
```

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

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

    def dispatch_request(self):
        return "Welcome home"

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

### 3.5.2 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):
    def get(self):
        return "Welcome home"

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

### 3.5.3 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 browse(id):
    browse_id = request.openapi.parameters.path['id']
    page = request.openapi.parameters.query.get('page', 1)

    return f"Browse {browse_id}, page {page}"
```

### 3.5.4 Low level

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

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

openapi_request = FlaskOpenAPIRequest(flask_request)
result = unmarshal_request(openapi_request, spec=spec)
```

For response factory see *Werkzeug* integration.

## 3.6 Pyramid

See pyramid_openapi3 project.

## 3.7 Requests

This section describes integration with Requests library.

### 3.7.1 Low level

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

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

openapi_request = RequestsOpenAPIRequest(requests_request)
result = unmarshal_request(openapi_request, spec=spec)
```

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

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

openapi_response = RequestsOpenAPIResponse(requests_response)
result = unmarshal_response(openapi_request, openapi_response, spec=spec)
```

You can use `RequestsOpenAPIWebhookRequest` as a Requests webhook request factory:

```
from openapi_core import unmarshal_request
from openapi_core.contrib.requests import RequestsOpenAPIWebhookRequest

openapi_webhook_request = RequestsOpenAPIWebhookRequest(requests_request, "my_webhook")
result = unmarshal_request(openapi_webhook_request, spec=spec)
```

## 3.8 Starlette

This section describes integration with Starlette ASGI framework.

### 3.8.1 Low level

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

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

openapi_request = StarletteOpenAPIRequest(starlette_request)
result = unmarshal_request(openapi_request, spec=spec)
```

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

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

openapi_response = StarletteOpenAPIResponse(starlette_response)
result = unmarshal_response(openapi_request, openapi_response, spec=spec)
```

## 3.9 Tornado

See tornado-openapi3 project.

## 3.10 Werkzeug

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

### 3.10.1 Low level

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

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

openapi_request = WerkzeugOpenAPIRequest(werkzeug_request)
result = unmarshal_request(openapi_request, spec=spec)
```

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

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

openapi_response = WerkzeugOpenAPIResponse(werkzeug_response)
result = unmarshal_response(openapi_request, openapi_response, spec=spec)
```

CHAPTER

FOUR

# CUSTOMIZATIONS

## 4.1 Specification validation

By default, the provided specification is validated on `Spec` object creation time.

If you know you have a valid specification already, disabling the validator can improve the performance.

```
from openapi_core import Spec

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

## 4.2 Media type deserializers

Pass custom defined media type deserializers dictionary with supported mimetypes as a key to *unmarshal_response* function:

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

extra_media_type_deserializers = {
    'application/protobuf': protobuf_deserializer,
}

result = unmarshal_response(
    request, response,
    spec=spec,
    extra_media_type_deserializers=extra_media_type_deserializers,
)
```

## 4.3 Format validators

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 comes with a set of built-in format validators, but it's also possible to add custom ones.

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

```
import re

def validate_usdate(value):
    return bool(re.match(r"^\d{1,2}/\d{1,2}/\d{4}$", value))

extra_format_validators = {
    'usdate': validate_usdate,
}

validate_response(
    request, response,
    spec=spec,
    extra_format_validators=extra_format_validators,
)
```

## 4.4 Format unmarshallers

Based on `format` keyword, openapi-core can also unmarshal values to specific formats.

Openapi-core comes with a set of built-in format unmarshallers, but it's also possible to add custom ones.

Here's an example with the `usdate` format that converts a value to date object:

```
from datetime import datetime

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

extra_format_unmarshallers = {
    'usdate': unmarshal_usdate,
}

result = unmarshal_response(
    request, response,
    spec=spec,
    extra_format_unmarshallers=extra_format_unmarshallers,
)
```

CHAPTER

FIVE

# SECURITY

Openapi-core provides you easy access to security data for authentication and authorization process.

Supported security schemas:

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

Here's an example with scheme `BasicAuth` and `ApiKeyAuth` security schemes:

```yaml
security:
  - BasicAuth: []
  - ApiKeyAuth: []
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
    ApiKeyAuth:
      type: apiKey
      in: header
      name: X-API-Key
```

Security schemes data are accessible from *security* attribute of *RequestUnmarshalResult* object.

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

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

CHAPTER

SIX

# EXTENSIONS

## 6.1 x-model

By default, objects are unmarshalled to dictionaries. You can use dynamically created dataclasses by providing `x-model-path` property inside schema definition with name of the model.

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

As a result of unmarshalling process, you will get `Coordinates` class instance with `lat` and `lon` attributes.

## 6.2 x-model-path

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:
```

(continues on next page)

(continued from previous page)

```
          type: number
        lon:
          type: number
```

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

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

As a result of unmarshalling process, you will get instance of your own dataclasses or model.

CHAPTER

SEVEN

# CONTRIBUTING

Firstly, thank you all for taking the time to contribute.

The following section describes how you can contribute to the openapi-core project on GitHub.

## 7.1 Reporting bugs

### 7.1.1 Before you report

- Check whether your issue does not already exist in the Issue tracker.
- Make sure it is not a support request or question better suited for Discussion board.

### 7.1.2 How to submit a report

- Include clear title.
- Describe your runtime environment with exact versions you use.
- Describe the exact steps which reproduce the problem, including minimal code snippets.
- Describe the behavior you observed after following the steps, pasting console outputs.
- Describe expected behavior to see and why, including links to documentations.

## 7.2 Code contribution

### 7.2.1 Prerequisites

Install Poetry by following the official installation instructions. Optionally (but recommended), configure Poetry to create a virtual environment in a folder named `.venv` within the root directory of the project:

```
poetry config virtualenvs.in-project true
```

### 7.2.2 Setup

To create a development environment and install the runtime and development dependencies, run:

```
poetry install
```

Then enter the virtual environment created by Poetry:

```
poetry shell
```

### 7.2.3 Static checks

The project uses static checks using fantastic pre-commit. Every change is checked on CI and if it does not pass the tests it cannot be accepted. If you want to check locally then run following command to install pre-commit.

To turn on pre-commit checks for commit operations in git, enter:

```
pre-commit install
```

To run all checks on your staged files, enter:

```
pre-commit run
```

To run all checks on all files, enter:

```
pre-commit run --all-files
```

Pre-commit check results are also attached to your PR through integration with Github Action.

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

EIGHT

# KEY FEATURES

- *Validation* and *unmarshalling* of request and response data (including webhooks)
- *Integrations* with popular libraries (Requests, Werkzeug) and frameworks (Django, Falcon, Flask, Starlette)
- *Customization* with **media type deserializers** and **format unmarshallers**
- *Security* data providers (API keys, Cookie, Basic and Bearer HTTP authentications)

CHAPTER

NINE

# INSTALLATION

### Pip + PyPI (recommented)

```
pip install openapi-core
```

### Pip + the source

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

CHAPTER

# TEN

# FIRST STEPS

Firstly create your specification object.

```
from openapi_core import Spec

spec = Spec.from_file_path('openapi.json')
```

Now you can use it to validate and unmarshal your requests and/or responses.

```
from openapi_core import unmarshal_request

# raises error if request is invalid
result = unmarshal_request(request, spec=spec)
```

Retrieve validated and unmarshalled request data

```
# get parameters
path_params = result.parameters.path
query_params = result.parameters.query
cookies_params = result.parameters.cookies
headers_params = result.parameters.headers
# get body
body = result.body
# get security data
security = result.security
```

Request object should implement OpenAPI Request protocol. Check *Integrations* to find oficially supported implementations.

For more details read about *Unmarshalling* process.

If you just want to validate your request/response data without unmarshalling, read about *Validation* instead.

CHAPTER

# ELEVEN

# RELATED PROJECTS

- **openapi-spec-validator**
  Python library that validates OpenAPI Specs against the OpenAPI 2.0 (aka Swagger), OpenAPI 3.0 and OpenAPI 3.1 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 and OpenAPI Schema Specification v3.1.

CHAPTER

TWELVE

# LICENSE

The project is under the terms of BSD 3-Clause License.