openapi-core

Release 0.19.0

Artur Maciag

Feb 14, 2024

CONTENTS

1 Unmarshalling **1**
1.1 Request unmarshalling . . . 1
1.2 Response unmarshalling . . . 2

2 Validation **3**
2.1 Request validation . . . 3
2.2 Response validation . . . 4

3 Integrations **5**
3.1 aiohttp.web . . . 5
3.1.1 Low level . . . 5
3.2 Bottle . . . 6
3.3 Django . . . 6
3.3.1 Middleware . . . 6
3.3.2 Low level . . . 7
3.4 Falcon . . . 8
3.4.1 Middleware . . . 8
3.4.2 Low level . . . 9
3.5 FastAPI . . . 9
3.5.1 Middleware . . . 9
3.5.2 Low level . . . 10
3.6 Flask . . . 10
3.6.1 View decorator . . . 10
3.6.2 View . . . 11
3.6.3 Request parameters . . . 12
3.6.4 Low level . . . 12
3.7 Pyramid . . . 12
3.8 Requests . . . 12
3.8.1 Low level . . . 12
3.9 Starlette . . . 13
3.9.1 Middleware . . . 13
3.9.2 Low level . . . 14
3.10 Tornado . . . 15
3.11 Werkzeug . . . 15
3.11.1 Low level . . . 15

4 Customizations **17**
4.1 Specification validation . . . 17
4.2 Request validator . . . 17
4.3 Response validator . . . 18

4.4 Request unmarshaller 18
4.5 Response unmarshaller 18
4.6 Media type deserializers 19
4.7 Format validators 19
4.8 Format unmarshallers 20

5 Security **21**

6 Extensions **23**
6.1 x-model 23
6.2 x-model-path 23

7 Contributing **25**
7.1 Reporting bugs 25
7.1.1 Before you report 25
7.1.2 How to submit a report 25
7.2 Code contribution 25
7.2.1 Prerequisites 25
7.2.2 Setup 26
7.2.3 Static checks 26

8 Key features **27**

9 Installation **29**

10 First steps **31**

11 Related projects **33**

12 License **35**

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 *Format unmarshallers*).

1.1 Request unmarshalling

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

```
# raises error if request is invalid
result = openapi.unmarshal_request(request)
```

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

Note

Webhooks feature is part of OpenAPI v3.1 only

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

```
# raises error if request is invalid
result = openapi.unmarshal_request(webhook_request)
```

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
```

You can also define your own request unmarshaller (See *Request unmarshaller*).

1.2 Response unmarshalling

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

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

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

Note

Webhooks feature is part of OpenAPI v3.1 only

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

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

Retrieve validated and unmarshalled response data

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

You can also define your own response unmarshaller (See *Response unmarshaller*).

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 *Format validators*).

2.1 Request validation

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

```
# raises error if request is invalid
openapi.validate_request(request)
```

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

Note

Webhooks feature is part of OpenAPI v3.1 only

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

```
# raises error if request is invalid
openapi.validate_request(webhook_request)
```

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

You can also define your own request validator (See *Request validator*).

2.2 Response validation

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

```python
from openapi_core import validate_response

# raises error if response is invalid
openapi.validate_response(request, response)
```

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.

```python
# raises error if request is invalid
openapi.validate_response(webhook_request, response)
```

You can also define your own response validator (See *Response validator*).

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

The integration defines classes useful for low level integration.

Request

Use `AIOHTTPOpenAPIWebRequest` to create OpenAPI request from aiohttp.web request:

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

async def hello(request):
    request_body = await request.text()
    openapi_request = AIOHTTPOpenAPIWebRequest(request, body=request_body)
    openapi.validate_request(openapi_request)
    return web.Response(text="Hello, world")
```

Response

Use `AIOHTTPOpenAPIWebResponse` to create OpenAPI response from aiohttp.web response:

```python
from openapi_core.contrib.starlette import AIOHTTPOpenAPIWebResponse

async def hello(request):
    request_body = await request.text()
    response = web.Response(text="Hello, world")
    openapi_request = AIOHTTPOpenAPIWebRequest(request, body=request_body)
    openapi_response = AIOHTTPOpenAPIWebResponse(response)
    result = openapi.unmarshal_response(openapi_request, openapi_response)
    return response
```

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 to apply OpenAPI validation to your entire application.

Add `DjangoOpenAPIMiddleware` to your `MIDDLEWARE` list and define `OPENAPI`.

```python
# settings.py
from openapi_core import OpenAPI

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

OPENAPI = OpenAPI.from_dict(spec_dict)
```

After that all your requests and responses will be validated.

Also you have access to unmarshal result object with all unmarshalled request data through `openapi` attribute of request object.

```python
from django.views import View

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

        # get body
        unmarshalled_body = request.openapi.body

        # get security data
        unmarshalled_security = request.openapi.security
```

Response validation

You can skip response validation process: by setting `OPENAPI_RESPONSE_CLS` to `None`

```
  # settings.py
  from openapi_core import OpenAPI

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

  OPENAPI = OpenAPI.from_dict(spec_dict)
  OPENAPI_RESPONSE_CLS = None
```

3.3.2 Low level

The integration defines classes useful for low level integration.

Request

Use `DjangoOpenAPIRequest` to create OpenAPI request from Django request:

```
from openapi_core.contrib.django import DjangoOpenAPIRequest

class MyView(View):
   def get(self, request):
       openapi_request = DjangoOpenAPIRequest(request)
       openapi.validate_request(openapi_request)
```

Response

Use `DjangoOpenAPIResponse` to create OpenAPI response from Django response:

```
from openapi_core.contrib.django import DjangoOpenAPIResponse

class MyView(View):
   def get(self, request):
       response = JsonResponse({'hello': 'world'})
       openapi_request = DjangoOpenAPIRequest(request)
       openapi_response = DjangoOpenAPIResponse(response)
       openapi.validate_response(openapi_request, openapi_response)
       return response
```

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],
)
```

Additional customization parameters can be passed to the middleware.

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

openapi_middleware = FalconOpenAPIMiddleware.from_spec(
    spec,
    extra_format_validators=extra_format_validators,
)

app = falcon.App(
    # ...
    middleware=[openapi_middleware],
)
```

You can skip response validation process: by setting `response_cls` to `None`

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

openapi_middleware = FalconOpenAPIMiddleware.from_spec(
    spec,
    response_cls=None,
)

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
```

(continues on next page)

(continued from previous page)

```
        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.contrib.falcon import FalconOpenAPIRequest

openapi_request = FalconOpenAPIRequest(falcon_request)
result = openapi.unmarshal_request(openapi_request)
```

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

```
from openapi_core.contrib.falcon import FalconOpenAPIResponse

openapi_response = FalconOpenAPIResponse(falcon_response)
result = openapi.unmarshal_response(openapi_request, openapi_response)
```

3.5 FastAPI

This section describes integration with FastAPI ASGI framework.

Note

FastAPI also provides OpenAPI support. The main difference is that, unlike FastAPI's code-first approach, OpenAPI-core allows you to laverage your existing specification that alligns with API-First approach. You can read more about API-first vs. code-first in the Guide to API-first.

3.5.1 Middleware

FastAPI can be integrated by middleware to apply OpenAPI validation to your entire application.

Add `FastAPIOpenAPIMiddleware` with OpenAPI object to your `middleware` list.

```
from fastapi import FastAPI
from openapi_core.contrib.fastapi.middlewares import FastAPIOpenAPIMiddleware

app = FastAPI()
app.add_middleware(FastAPIOpenAPIMiddleware, openapi=openapi)
```

After that all your requests and responses will be validated.

Also you have access to unmarshal result object with all unmarshalled request data through `openapi` scope of request object.

```python
async def homepage(request):
    # get parameters object with path, query, cookies and headers parameters
    unmarshalled_params = request.scope["openapi"].parameters
    # or specific location parameters
    unmarshalled_path_params = request.scope["openapi"].parameters.path

    # get body
    unmarshalled_body = request.scope["openapi"].body

    # get security data
    unmarshalled_security = request.scope["openapi"].security
```

Response validation

You can skip response validation process: by setting `response_cls` to `None`

```python
app = FastAPI()
app.add_middleware(
    FastAPIOpenAPIMiddleware,
    openapi=openapi,
    response_cls=None,
)
```

3.5.2 Low level

For low level integration see Starlette integration.

3.6 Flask

This section describes integration with Flask web framework.

3.6.1 View decorator

Flask can be integrated by view decorator to apply OpenAPI validation to your application's specific views.

Use `FlaskOpenAPIViewDecorator` with OpenAPI object to create the decorator.

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

openapi_validated = FlaskOpenAPIViewDecorator(openapi)

@app.route('/home')
@openapi_validated
def home():
    return "Welcome home"
```

You can skip response validation process: by setting `response_cls` to `None`

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

openapi_validated = FlaskOpenAPIViewDecorator(
    openapi,
    response_cls=None,
)
```

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

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

    def dispatch_request(self):
        return "Welcome home"

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

3.6.2 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),
)
```

Additional customization parameters can be passed to the view.

```
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,
        extra_format_validators=extra_format_validators,
    ),
)
```

3.6.3 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}"
```

3.6.4 Low level

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

```
from openapi_core.contrib.flask import FlaskOpenAPIRequest

openapi_request = FlaskOpenAPIRequest(flask_request)
result = openapi.unmarshal_request(openapi_request)
```

For response factory see Werkzeug integration.

3.7 Pyramid

See pyramid_openapi3 project.

3.8 Requests

This section describes integration with Requests library.

3.8.1 Low level

The integration defines classes useful for low level integration.

Request

Use `RequestsOpenAPIRequest` to create OpenAPI request from Requests request:

```
from openapi_core.contrib.requests import RequestsOpenAPIRequest

request = Request('POST', url, data=data, headers=headers)
openapi_request = RequestsOpenAPIRequest(request)
openapi.validate_request(openapi_request)
```

Webhook request

Use `RequestsOpenAPIWebhookRequest` to create OpenAPI webhook request from Requests request:

```
from openapi_core.contrib.requests import RequestsOpenAPIWebhookRequest

request = Request('POST', url, data=data, headers=headers)
openapi_webhook_request = RequestsOpenAPIWebhookRequest(request, "my_webhook")
openapi.validate_request(openapi_webhook_request)
```

Response

Use `RequestsOpenAPIResponse` to create OpenAPI response from Requests response:

```
from openapi_core.contrib.requests import RequestsOpenAPIResponse

session = Session()
request = Request('POST', url, data=data, headers=headers)
prepped = session.prepare_request(req)
response = session,send(prepped)
openapi_request = RequestsOpenAPIRequest(request)
openapi_response = RequestsOpenAPIResponse(response)
openapi.validate_response(openapi_request, openapi_response)
```

3.9 Starlette

This section describes integration with Starlette ASGI framework.

3.9.1 Middleware

Starlette can be integrated by middleware to apply OpenAPI validation to your entire application.

Add `StarletteOpenAPIMiddleware` with OpenAPI object to your `middleware` list.

```
from openapi_core.contrib.starlette.middlewares import StarletteOpenAPIMiddleware
from starlette.applications import Starlette
from starlette.middleware import Middleware

middleware = [
    Middleware(StarletteOpenAPIMiddleware, openapi=openapi),
]

app = Starlette(
    # ...
    middleware=middleware,
)
```

After that all your requests and responses will be validated.

Also you have access to unmarshal result object with all unmarshalled request data through `openapi` scope of request object.

```
async def homepage(request):
   # get parameters object with path, query, cookies and headers parameters
   unmarshalled_params = request.scope["openapi"].parameters
   # or specific location parameters
   unmarshalled_path_params = request.scope["openapi"].parameters.path

   # get body
   unmarshalled_body = request.scope["openapi"].body

   # get security data
   unmarshalled_security = request.scope["openapi"].security
```

Response validation

You can skip response validation process: by setting `response_cls` to `None`

```
middleware = [
    Middleware(StarletteOpenAPIMiddleware, openapi=openapi, response_cls=None),
]

app = Starlette(
    # ...
    middleware=middleware,
)
```

3.9.2 Low level

The integration defines classes useful for low level integration.

Request

Use `StarletteOpenAPIRequest` to create OpenAPI request from Starlette request:

```
from openapi_core.contrib.starlette import StarletteOpenAPIRequest

async def homepage(request):
    openapi_request = StarletteOpenAPIRequest(request)
    result = openapi.unmarshal_request(openapi_request)
    return JSONResponse({'hello': 'world'})
```

Response

Use StarletteOpenAPIResponse to create OpenAPI response from Starlette response:

```
from openapi_core.contrib.starlette import StarletteOpenAPIResponse

async def homepage(request):
    response = JSONResponse({'hello': 'world'})
    openapi_request = StarletteOpenAPIRequest(request)
    openapi_response = StarletteOpenAPIResponse(response)
    openapi.validate_response(openapi_request, openapi_response)
    return response
```

3.10 Tornado

See tornado-openapi3 project.

3.11 Werkzeug

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

3.11.1 Low level

The integration defines classes useful for low level integration.

Request

Use WerkzeugOpenAPIRequest to create OpenAPI request from Werkzeug request:

```
from openapi_core.contrib.werkzeug import WerkzeugOpenAPIRequest

def application(environ, start_response):
    request = Request(environ)
    openapi_request = WerkzeugOpenAPIRequest(request)
    openapi.validate_request(openapi_request)
    response = Response("Hello world", mimetype='text/plain')
    return response(environ, start_response)
```

Response

Use WerkzeugOpenAPIResponse to create OpenAPI response from Werkzeug response:

```
from openapi_core.contrib.werkzeug import WerkzeugOpenAPIResponse

def application(environ, start_response):
    request = Request(environ)
    response = Response("Hello world", mimetype='text/plain')
```

(continues on next page)

(continued from previous page)

```
    openapi_request = WerkzeugOpenAPIRequest(request)
    openapi_response = WerkzeugOpenAPIResponse(response)
    openapi.validate_response(openapi_request, openapi_response)
    return response(environ, start_response)
```

CHAPTER FOUR

CUSTOMIZATIONS

OpenAPI accepts `Config` object that allows users to customize the behavior validation and unmarshalling processes.

4.1 Specification validation

By default, on OpenAPI creation time, the provided specification is also validated.

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

```python
from openapi_core import Config

config = Config(
    spec_validator_cls=None,
)
openapi = OpenAPI.from_file_path('openapi.json', config=config)
```

4.2 Request validator

By default, request validator is selected based on detected specification version.

In order to explicitly validate a:

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

```python
from openapi_core import V31RequestValidator

config = Config(
    request_validator_cls=V31RequestValidator,
)
openapi = OpenAPI.from_file_path('openapi.json', config=config)
openapi.validate_request(request)
```

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

4.3 Response validator

By default, response validator is selected based on detected specification version.

In order to explicitly validate a:

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

```python
from openapi_core import V31ResponseValidator

config = Config(
   response_validator_cls=V31ResponseValidator,
)
openapi = OpenAPI.from_file_path('openapi.json', config=config)
openapi.validate_response(request, response)
```

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

4.4 Request unmarshaller

By default, request unmarshaller is selected based on detected specification version.

In order to explicitly validate and unmarshal a:

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

```python
from openapi_core import V31RequestUnmarshaller

config = Config(
   request_unmarshaller_cls=V31RequestUnmarshaller,
)
openapi = OpenAPI.from_file_path('openapi.json', config=config)
result = openapi.unmarshal_request(request)
```

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

4.5 Response unmarshaller

In order to explicitly validate and unmarshal a:

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

```python
from openapi_core import V31ResponseUnmarshaller

config = Config(
   response_unmarshaller_cls=V31ResponseUnmarshaller,
)
```

(continues on next page)

(continued from previous page)

```
openapi = OpenAPI.from_file_path('openapi.json', config=config)
result = openapi.unmarshal_response(request, response)
```

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

4.6 Media type deserializers

OpenAPI comes with a set of built-in media type deserializers such as: `application/json`, `application/xml`, `application/x-www-form-urlencoded` or `multipart/form-data`.

You can also define your own ones. 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,
}

config = Config(
    extra_media_type_deserializers=extra_media_type_deserializers,
)
openapi = OpenAPI.from_file_path('openapi.json', config=config)

result = openapi.unmarshal_response(request, response)
```

4.7 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,
}

config = Config(
    extra_format_validators=extra_format_validators,
)
```

(continues on next page)

(continued from previous page)

```
openapi = OpenAPI.from_file_path('openapi.json', config=config)

openapi.validate_response(request, response)
```

4.8 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,
}

config = Config(
   extra_format_unmarshallers=extra_format_unmarshallers,
)
openapi = OpenAPI.from_file_path('openapi.json', config=config)

result = openapi.unmarshal_response(request, response)
```

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:

```
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.

```
# 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.

```yaml
# ...
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.

```yaml
...
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 OpenAPI object.

```python
from openapi_core import OpenAPI

openapi = OpenAPI.from_file_path('openapi.json')
```

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

```python
# raises error if request is invalid
result = openapi.unmarshal_request(request)
```

Retrieve validated and unmarshalled request data

```python
# 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.