Cornice Swagger

Release 0.3.0

Jul 30, 2018

Contents

Cornice Swagger is an extension package for Cornice that allows generating an OpenAPI/Swagger specification from Cornice service definitions.

CHAPTER 1

What does it do?

Cornice swagger builds a valid OpenAPI document with basically these things:

1. Path listing and parameters from service path definitions.
2. Methods defined from each service view.
3. Descriptions from view docstrings.
4. Parameters from request schemas when using cornice/colander validators on the view definition.
5. Produced content-types when using colander renderers.
6. Accepted content types when they are provided on view definitions.
7. Allow user defined tags and responses on view definitions.

CHAPTER 2

Documentation content

2.1 Quickstart

2.1.1 Installing

You may install us with pip:

```
$ pip install cornice_swagger
```

From an existing Cornice application, you may add this extension to your Pyramid configurator after including cornice:

```
from pyramid.config import Configurator

def setup():
    config = Configurator()
    config.include('cornice')
    config.include('cornice_swagger')
```

You can than create your OpenAPI/Swagger JSON using:

```
from cornice_swagger import CorniceSwagger
from cornice.service import get_services

my_generator = CorniceSwagger(get_services())
my_spec = my_generator('MyAPI', '1.0.0')
```

Alternatively you can use a directive to set up OpenAPI/Swagger JSON and serve API explorer on your application:

```
config = Configurator()
config.include('cornice')
config.include('cornice_swagger')
config.cornice_enable_openapi_view(
    api_path='/api-explorer/swagger.json',
```

(continues on next page)

(continued from previous page)

```
    title='MyAPI',
    description="OpenAPI documentation",
    version='1.0.0'
)
config.cornice_enable_openapi_explorer(
    api_explorer_path='/api-explorer')
```

Then you will be able to access Swagger UI API explorer on url:

http://localhost:8000/api-explorer (in the example above)

2.1.2 Using a scaffold

If you want to start a new project, there is a cookiecutter scaffold that can be used:

```
$ cookiecutter https://github.com/delijati/cookiecutter-cornice_swagger.git
$ cd demo
$ pip install -e .
$ cd demo/static
$ bower install
```

2.1.3 Show me a minimalist useful example

```
import colander
from cornice import Service
from cornice.validators import colander_body_validator
from wsgiref.simple_server import make_server
from pyramid.config import Configurator


_VALUES = {}


# Create a simple service that will store and retrieve values
values = Service(name='foo',
                 path='/values/{key}',
                 description="Cornice Demo")


# Create a body schema for our requests
class BodySchema(colander.MappingSchema):
    value = colander.SchemaNode(colander.String(),
                                description='My precious value')


# Create a response schema for our 200 responses
class OkResponseSchema(colander.MappingSchema):
    body = BodySchema()


# Aggregate the response schemas for get requests
response_schemas = {
    '200': OkResponseSchema(description='Return value')
}
```

(continues on next page)

(continued from previous page)

```
# Create our cornice service views
class MyValueApi(object):
    """My precious API."""

    @values.get(tags=['values'], response_schemas=response_schemas)
    def get_value(request):
        """Returns the value."""
        key = request.matchdict['key']
        return _VALUES.get(key)

    @values.put(tags=['values'], validators=(colander_body_validator, ),
                schema=BodySchema(), response_schemas=response_schemas)
    def set_value(request):
        """Set the value and returns *True* or *False*."""

        key = request.matchdict['key']
        _VALUES[key] = request.json_body
        return _VALUES.get(key)


# Setup and run our app
def setup():
    config = Configurator()
    config.include('cornice')
    config.include('cornice_swagger')
    # Create views to serve our OpenAPI spec
    config.cornice_enable_openapi_view(
        api_path='/__api__',
        title='MyAPI',
        description="OpenAPI documentation",
        version='1.0.0'
    )
    # Create views to serve OpenAPI spec UI explorer
    config.cornice_enable_openapi_explorer(api_explorer_path='/api-explorer')
    config.scan()
    app = config.make_wsgi_app()
    return app


if __name__ == '__main__':
    app = setup()
    server = make_server('127.0.0.1', 8000, app)
    print('Visit me on http://127.0.0.1:8000')
    print('''You can see the API explorer here:
    http://127.0.0.1:8000/api-explorer''')
    server.serve_forever()
```

The resulting *swagger.json* at *http://localhost:8000/__api__* is:

```
{
    "swagger": "2.0",
    "info": {
        "version": "1.0.0",
        "title": "MyAPI"
    },
```

(continues on next page)

(continued from previous page)


```
    "basePath": "/",
    "tags": [
        {
            "name": "values"
        }
    ]
    "paths": {
        "/values/{key}": {
            "parameters": [
                {
                    "name": "value",
                    "in": "path",
                    "required": true,
                    "type": "string"
                }
            ],
            "get": {
                "tags": [
                    "values"
                ],
                "responses": {
                    "200": {
                        "description": "Return value",
                        "schema": {
                            "required": [
                                "value"
                            ],
                            "type": "object",
                            "properties": {
                                "value": {
                                    "type": "string",
                                    "description": "My precious value",
                                    "title": "Value"
                                }
                            },
                            "title": "BodySchema"
                        }
                    }

                },
                "produces": [
                    "application/json"
                ]
            },
            "put": {
                "tags": [
                    "values"
                ],
                "parameters": [
                    {
                        "name": "PutBodySchema",
                        "in": "body",
                        "schema": {
                            "required": [
                                "value"
                            ],
                            "type": "object",
```


(continues on next page)

(continued from previous page)


```
                            "properties": {
                                "value": {
                                    "type": "string",
                                    "description": "My precious value",
                                    "title": "Value"
                                }
                            },
                            "title": "PutBodySchema"
                        },
                        "required": true
                    }
                ],
                "produces": [
                    "application/json"
                ],
                "responses": {
                    "200": {
                        "description": "Return value",
                        "schema": {
                            "required": [
                                "value"
                            ],
                            "type": "object",
                            "properties": {
                                "value": {
                                    "type": "string",
                                    "description": "My precious value",
                                    "title": "Value"
                                }
                            },
                            "title": "BodySchema"
                        }
                    }
                }
            }
        },
        "/__api__": {
            "get": {
                "responses": {
                    "default": {
                        "description": "UNDOCUMENTED RESPONSE"
                    }
                },
                "produces": [
                    "application/json"
                ]
            }
        }
    }
}
```


2.2 Full Tutorial

Here you may find the general aspects used by Colander Swagger to generate the swagger documentation. Most examples presented on this section refer to the example on *quickstart*.

2.2.1 Using the Pyramid Hook

In order to enable response documentation, you must add this extension to your Pyramid config. For that you may use:

```
from pyramid.config import Configurator

def setup():
    config = Configurator()
    config.include('cornice')
    config.include('cornice_swagger')
    config.cornice_enable_openapi_view(
        title='MyAPI',
        description="OpenAPI documentation",
        version='1.0.0'
    )
    config.cornice_enable_openapi_explorer()
```

If you don't know what this is about or need more information, please check the Pyramid documentation

By default API explorer will be served under */api-explorer* path in your application. You can easily configure the paths, required permissions and Pyramid route factory.

Additional `kwargs` passed to this directive will be passed to `CorniceSwagger.generate` method.

2.2.2 Extracting path parameters

Path parameters may be automatically extracted from the service definition, you may overwrite then with schemas on the view if you wish to add more view-specific information.

Another example:

```
values = Service(name='foo',
                 path='/values/{value}')
```

```
{
    "paths": {
        "/values/{value}": {
            "parameters": [
                {
                    "name": "value",
                    "in": "path",
                    "required": true,
                    "type": "string"
                }
            ]
        }
    }
}
```

2.2.3 Extracting parameters from cornice schemas

When using colander validators such as `colader_validator` or `colander_body_validator`, we can extract the operation parameters from the request schema. The schemas should comply with Cornice 2.0 colander schemas.

```
from cornice.validators import colander_body_validator

values = Service(name='foo',
                 path='/values/{value}')

class PutBodySchema(colander.MappingSchema):
    value = colander.SchemaNode(colander.String(),
                                description='My precious value')


@values.put(validators=(colander_body_validator, ),
            schema=PutBodySchema())
def set_value(request):
    """Set the value and returns *True* or *False*."""
```

```
{
    "paths": {
        "/values/{value}": {
            "put": {
                "parameters": [
                    {
                        "name": "PutBodySchema",
                        "in": "body",
                        "required": true,
                        "schema": {
                            "title": "PutBodySchema",
                            "type": "object",
                            "properties": {
                                "value": {
                                    "type": "string",
                                    "description": "My precious value",
                                    "title": "Value"
                                }
                            },
                            "required": [
                                "value"
                            ]
                        }
                    }
                ]
            }
        }
    }
}
```

When using *colander_validator*, the request should have fields corresponding the parameters locations as follows:

```
from cornice.validators import colander_validator

class BodySchema(colander.MappingSchema):
    value = colander.SchemaNode(colander.String(),
                                description='My precious value')


class QuerySchema(colander.MappingSchema):
    foo = colander.SchemaNode(colander.String(), missing=colander.drop)
```

(continues on next page)

(continued from previous page)

```
class HeaderSchema(colander.MappingSchema):
    bar = colander.SchemaNode(colander.String(), default='blah')


class PutRequestSchema(colander.MappingSchema):
    body = BodySchema()
    querystring = QuerySchema()
    header = HeaderSchema()


@values.put(validators=(colander_validator, ),
            schema=PutRequestSchema())
def set_value(request):
    """Set the value and returns *True* or *False*."""
    pass
```

When using custom validators, you can pass a method that transforms your custom schema into a regular Cornice schema that can be validated with `colander_validator`, so Cornice Swagger knows how to convert it to swagger parameters.

```
from cornice.validators import colander_body_validator

MY_IDS = [1, 2, 42]

def my_custom_validator(request, **kwargs):

    schema = kwargs.get('schema')

    # Peforms random additional validation
    if schema and schema['id'] not in MY_IDS:
        request.errors.add("body", "id", "Invalid id.")

    return colander_body_validator(request, **kwargs)

@values.put(validators=(my_custom_validator, ),
            schema=PutRequestSchema())
def set_value(request):
    pass
```

```
from cornice.service import get_services
from cornice_swagger import CorniceSwagger
from cornice_swagger.utils import body_schema_converter

def my_custom_schema_converter(schema, args):
    validators = args.get('validators', [])
    if my_custom_validator in validators:
        return body_schema_converter(schema, args)

swagger = CorniceSwagger(get_services())
swagger.schema_transformers.append(my_custom_schema_converter)
print(swagger.generate())
```

2.2.4 Extracting produced types from renderers

The produced content-type field is filled by the cornice renderer you are using. We currently support *json*, *simplejson* and *xml* renderers. Cornice uses *simplejson* renderer by default, so if you don't specify a renderer you may expect to find *application/json* on your operation produce fields.

```
values = Service(name='foo',
                 path='/values/{value}')

@values.put(renderer='xml')
def set_value(request):
    """Set the value and returns *True* or *False*."""
```

```
{
    "paths": {
        "/values/{value}": {
            "put": {
                "produces": [
                    "text/xml"
                ]
            }
        }
    }
}
```

2.2.5 Extracting accepted types from content_type field

On cornice you can defined the accepted content-types for your view through the *content_type* field. And we use it to generate the Swagger *consumes* types.

```
values = Service(name='foo',
                 path='/values/{value}')

@values.put(content_type=('application/json', 'text/xml'))
def set_value(request):
    """Set the value and returns *True* or *False*."""
```

```
{
    "paths": {
        "/values/{value}": {
            "put": {
                "consumes": [
                    "application/json",
                    "text/xml"
                ]
            }
        }
    }
}
```

2.2.6 Documenting responses

Unfortunately, on Cornice we don't have a way to provide response schemas, so this part must be provided separately and handled by Cornice Swagger.

For that you must provide a Response Colander Schema that follows the pattern:

```
class ResponseSchema(colander.MappingSchema):
    body = BodySchema()
    headers = HeaderSchema()


get_response_schemas = {
    '200': ResponseSchema(description='Return my OK response'),
    '404': ResponseSchema(description='Return my not found response')
}
```

Notice that the `ResponseSchema` class follows the same pattern as the Cornice requests using `cornice.validators.colander_validator` (except for querystrings, since obviously we don't have querystrings on responses).

A response schema mapping, as the `get_response_schemas` dict should aggregate response schemas as the one defined as `ResponseSchema` with keys matching the response status code of for each entry. All schema entries should contain descriptions. You may also provide a `default` response schema to be used if the response doesn't match any of the status codes provided.

From our minimalist example:

```
values = Service(name='foo',
                 path='/values/{value}')

# Create a body schema for our requests
class BodySchema(colander.MappingSchema):
    value = colander.SchemaNode(colander.String(),
                                description='My precious value')


# Create a response schema for our 200 responses
class OkResponseSchema(colander.MappingSchema):
    body = BodySchema()


# Aggregate the response schemas for get requests
response_schemas = {
    '200': OkResponseSchema(description='Return value')
}


@values.put(response_schemas=response_schemas)
def set_value(request):
    """Set the value and returns *True* or *False*."""
```

```
{
    "paths": {
        "/values/{value}": {
            "put": {
                "responses": {
                    "200": {
                        "description": "Return value",
                        "schema": {
                            "required": [
                                "value"
                            ],
```

(continues on next page)

(continued from previous page)

```
                            "type": "object",
                            "properties": {
                                "value": {
                                    "type": "string",
                                    "description": "My precious value",
                                    "title": "Value"
                                }
                            },
                            "title": "BodySchema"
                        }
                    }
                }
            }
        }
    }
}
```

2.2.7 Documenting tags

Cornice Swagger supports two ways of documenting operation tags. You can either provide a list of tags on the view decorator or have a `default_tags` attribute when calling the generator.

```
values = Service(name='foo',
                 path='/values/{value}')

@values.put(tags=['value'])
def set_value(request):
    """Set the value and returns *True* or *False*."""
```

```
{
    "tags": [
        {
            "name": "values"
        }
    ],
    "paths": {
        "/values/{value}": {
            "get": {
                "tags": [
                    "values"
                ]
            }
        }
    }
}
```

When using the `default_tags` attribute, you can either use a raw list of tags or a callable that takes a cornice service and returns a list of tags.

```
def default_tag_callable(service):
    return [service.path.split('/')[1]]

swagger = CorniceSwagger(get_services())
swagger.default_tags = default_tag_callable
spec = swagger.generate('IceCreamAPI', '4.2')
```

```
from cornice.service import get_services
from cornice_swagger import CorniceSwagger

swagger = CorniceSwagger(get_services())
swagger.default_tags = ['IceCream']
spec = swagger.generate('IceCreamAPI', '4.2')
```

2.2.8 Generating summaries with view docstrings

You may use view docstrings to create operation summaries. You may enable this by passing `summary_docstrings=True` when calling the generator. For example, the following view definition docstring will correspond to the following swagger summary:

```
values = Service(name='foo',
                 path='/values')

@values.get()
def get_value(request):
    """Returns the value."""

swagger = CorniceSwagger(get_services())
swagger.summary_docstrings = True
spec = swagger.generate('IceCreamAPI', '4.2')
```

```
{
    "paths": {
        "/values": {
            "get": {
                "summary": "Returns the value."
            }
        }
    }
}
```

2.2.9 Custom Swagger UI <script> bootstrap

By default standard Swagger UI (https://swagger.io/swagger-ui/) config is used, but you can customize the generated script tag by providing your own callable path in config.

The default one is:

`cornice_swagger.swagger_ui_script_generator = cornice_swagger.views:swagger_ui_script_template`

It points to the following callable that accepts a request object:

```
def swagger_ui_script_template(request, **kwargs):
    """
    :param request:
    :return:

    Generates the <script> code that bootstraps Swagger UI, it will be injected
    into index template
    """
```

(continues on next page)

(continued from previous page)

```
    swagger_spec_url = request.route_url('cornice_swagger.open_api_path')
    template = pkg_resources.resource_string(
        'cornice_swagger',
        'templates/index_script_template.html'
    ).decode('utf8')
    return Template(template).safe_substitute(
        swagger_spec_url=swagger_spec_url,
    )
```

2.3 Cornice Swagger API

Here you may find information about the Cornice Swagger internals and methods that may be overwritten by applications.

2.3.1 Basic Generator

class cornice_swagger.swagger.**CorniceSwagger**(*services=None*, *def_ref_depth=0*, *param_ref=False*, *resp_ref=False*, *pyramid_registry=None*)

Handles the creation of a swagger document from a cornice application.

schema_transformers = [<function body_schema_transformer>]
List of request schema transformers that should be applied to a request schema to make it comply with a cornice default request schema.

type_converter
alias of `cornice_swagger.converters.schema.TypeConversionDispatcher`

parameter_converter
alias of `cornice_swagger.converters.parameters.ParameterConversionDispatcher`

custom_type_converters = {}
Mapping for supporting custom types conversion on the default TypeConverter. Should map *colander.TypeSchema* to *cornice_swagger.converters.schema.TypeConverter* callables.

default_type_converter = None
Supplies a default type converter matching the interface of *cornice_swagger.converters.schema.TypeConverter* to be used with unknown types.

default_tags = None
Provide a default list of tags or a callable that takes a cornice service and the method name (e.g GET) and returns a list of Swagger tags to be used if not provided by the view.

default_op_ids = None
Provide a callable that takes a cornice service and the method name (e.g. GET) and returns an operation Id that is used if an operation Id is not provided. Each operation Id should be unique.

default_security = None
Provide a default list or a callable that takes a cornice service and the method name (e.g. GET) and returns a list of OpenAPI security policies.

summary_docstrings = False
Enable extracting operation summaries from view docstrings.

ignore_methods = ['HEAD', 'OPTIONS']
List of service methods that should NOT be presented on the documentation. You may use this to remove methods that are not essential on the API documentation. Default ignores HEAD and OPTIONS.

ignore_ctypes = []
List of service content-types that should NOT be presented on the documentation. You may use this when a Cornice service definition has multiple view definitions for a same method, which is not supported on OpenAPI 2.0.

api_title = ''
Title of the OpenAPI document.

api_version = ''
Version of the OpenAPI document.

base_path = '/'
Base path of the documented API. Default is "/".

swagger = {'info': {}}
Base OpenAPI document that should be merged with the extracted info from the generate call.

services = []
List of cornice services to document. You may use *cornice.service.get_services()* to get it.

definitions
Default *cornice_swagger.swagger.DefinitionHandler* class to use when handling OpenAPI schema definitions from cornice payload schemas.

alias of *DefinitionHandler*

parameters
Default *cornice_swagger.swagger.ParameterHandler* class to use when handling OpenAPI operation parameters from cornice request schemas.

alias of *ParameterHandler*

responses
Default *cornice_swagger.swagger.ResponseHandler* class to use when handling OpenAPI responses from cornice_swagger defined responses.

alias of *ResponseHandler*

generate(*title=None*, *version=None*, *base_path=None*, *info=None*, *swagger=None*, ***kwargs*)
Generate a Swagger 2.0 documentation. Keyword arguments may be used to provide additional information to build methods as such ignores.

Parameters

- **title** – The name presented on the swagger document.
- **version** – The version of the API presented on the swagger document.
- **base_path** – The path that all requests to the API must refer to.
- **info** – Swagger info field.
- **swagger** – Extra fields that should be provided on the swagger documentation.

Return type dict

Returns Full OpenAPI/Swagger compliant specification for the application.

2.3.2 cornice_enable_openapi_view directive

cornice_swagger.**cornice_enable_openapi_view**(*config*, *api_path='/api-explorer/swagger.json'*, *permission='__no_permission_required__'*, *route_factory=None*, ***kwargs*)

Parameters

- **config** – Pyramid configurator object
- **api_path** – where to expose swagger JSON definition view
- **permission** – pyramid permission for those views
- **route_factory** – factory for context object for those routes
- **kwargs** – kwargs that will be passed to CorniceSwagger's *generate()*

This registers and configures the view that serves api definitions

2.3.3 cornice_enable_openapi_explorer directive

cornice_swagger.**cornice_enable_openapi_explorer**(*config*, *api_explorer_path='/api-explorer'*, *permission='__no_permission_required__'*, *route_factory=None*, ***kwargs*)

Parameters

- **config** – Pyramid configurator object
- **api_explorer_path** – where to expose Swagger UI interface view
- **permission** – pyramid permission for those views
- **route_factory** – factory for context object for those routes

This registers and configures the view that serves api explorer

2.3.4 Generator Internals

CorniceSwagger.**_build_paths**()
Build the Swagger "paths" and "tags" attributes from cornice service definitions.

CorniceSwagger.**_extract_path_from_service**(*service*)
Extract path object and its parameters from service definitions.

Parameters **service** – Cornice service to extract information from.

Return type dict

Returns Path definition.

CorniceSwagger.**_extract_operation_from_view**(*view*, *args*)
Extract swagger operation details from colander view definitions.

Parameters

- **view** – View to extract information from.
- **args** – Arguments from the view decorator.

Return type dict

Returns Operation definition.

2.3.5 Section Handlers

Swagger definitions and parameters are handled in separate classes. You may overwrite those if you want to change the converters behaviour.

class cornice_swagger.swagger.**DefinitionHandler**(*ref=0*, *type_converter=<cornice_swagger.converters.schema.Ty*
object>)

Handles Swagger object definitions provided by cornice as colander schemas.

DefinitionHandler.**__init__**(*ref=0*, *type_converter=<cornice_swagger.converters.schema.TypeConversionDispatcher*
object>)

Parameters **ref** – The depth that should be used by self.ref when calling self.from_schema.

DefinitionHandler.**from_schema**(*schema_node*, *base_name=None*)

Creates a Swagger definition from a colander schema.

Parameters

- **schema_node** – Colander schema to be transformed into a Swagger definition.
- **base_name** – Schema alternative title.

Return type dict

Returns Swagger schema.

DefinitionHandler.**_ref_recursive**(*schema*, *depth*, *base_name=None*)

Dismantle nested swagger schemas into several definitions using JSON pointers. Note: This can be dangerous since definition titles must be unique.

Parameters

- **schema** – Base swagger schema.
- **depth** – How many levels of the swagger object schemas should be split into swaggger definitions with JSON pointers. Default (0) is no split. You may use negative values to split everything.
- **base_name** – If schema doesn't have a name, the caller may provide it to be used as reference.

Return type dict

Returns JSON pointer to the root definition schema, or the original definition if depth is zero.

class cornice_swagger.swagger.**ParameterHandler**(*definition_handler=<cornice_swagger.swagger.DefinitionHandle*
object>, *ref=False*,
type_converter=<cornice_swagger.converters.schema.TypeConv
object>, *parame-*
ter_converter=<cornice_swagger.converters.parameters.Parame
object>)

Handles swagger parameter definitions.

ParameterHandler.**__init__**(*definition_handler=<cornice_swagger.swagger.DefinitionHandler ob-*
ject>, *ref=False*, *type_converter=<cornice_swagger.converters.schema.TypeConversionDispatch*
object>, *parameter_converter=<cornice_swagger.converters.parameters.ParameterConversion*
object>)

Parameters

- **definition_handler** – Callable that handles swagger definition schemas.
- **ref** – Specifies the ref value when calling from_xxx methods.

ParameterHandler.**from_schema**(*schema_node*)
Creates a list of Swagger params from a colander request schema.

Parameters

- **schema_node** – Request schema to be transformed into Swagger.
- **validators** – Validators used in colander with the schema.

Return type list

Returns List of Swagger parameters.

ParameterHandler.**from_path**(*path*)
Create a list of Swagger path params from a cornice service path.

Return type list

ParameterHandler.**_ref**(*param*, *base_name=None*)
Store a parameter schema and return a reference to it.

Parameters

- **schema** – Swagger parameter definition.
- **base_name** – Name that should be used for the reference.

Return type dict

Returns JSON pointer to the original parameter definition.

class cornice_swagger.swagger.**ResponseHandler**(*definition_handler=<cornice_swagger.swagger.DefinitionHandler object>*, *type_converter=<cornice_swagger.converters.schema.Ty object>*, *ref=False*)
Handles swagger response definitions.

ResponseHandler.**__init__**(*definition_handler=<cornice_swagger.swagger.DefinitionHandler object>*, *type_converter=<cornice_swagger.converters.schema.TypeConversionDispatcher object>*, *ref=False*)

Parameters

- **definition_handler** – Callable that handles swagger definition schemas.
- **ref** – Specifies the ref value when calling from_xxx methods.

ResponseHandler.**from_schema_mapping**(*schema_mapping*)
Creates a Swagger response object from a dict of response schemas.

Parameters **schema_mapping** – Dict with entries matching `{status_code: response_schema}`.

Return type dict

Returns Response schema.

ResponseHandler.**_ref**(*resp*, *base_name=None*)
Store a response schema and return a reference to it.

Parameters

- **schema** – Swagger response definition.
- **base_name** – Name that should be used for the reference.

Return type dict

Returns JSON pointer to the original response definition.

2.3.6 Colander converters

You may use the `cornice_swagger.converters` submodule to access the colander to swagger request and schema converters. These may be also used without `cornice_swagger` generators.

This module handles the conversion between colander object schemas and swagger object schemas.

`cornice_swagger.converters.`**`convert_schema`**(*schema_node*)

`cornice_swagger.converters.`**`convert_parameter`**(*location*, *schema_node*, *definition_handler=<function convert_schema>*)

2.4 Frequently Asked Questions (FAQ)

Here is a list of frequently asked questions related to Cornice Swagger.

2.4.1 How to make a schema parameter not required?

You may use `colader.drop` as it's missing field:

```
field = colader.SchemaNode(colander.String(), missing=colader.drop)
```

2.4.2 How to define a schema with additionalAttributes?

You can use `Mapping.unknown` attribute

```
class Query(colander.MappingSchema):
    unknown='preserve'
```

2.4.3 How do I integrate Swagger UI?

The fastest way to enable Swagger UI is to use directives `cornice_enable_openapi_view` together with `cornice_enable_openapi_explorer` they will provide a special view in your application that will server the OpenAPI specification along with API explorer.

2.4.4 How do I work with colander schemas that require bound properties?

A common scenario is to have schemas that have optional fields that substitute default values when fields are missing from request data. This is normally solved by calling *bind()* on colander schemas, one thing that is important to remember is that if the value needs to be updated per request (like a date or UUID), you need to bind the schema per request. At the same time for `cornice_swagger` to get proper values when inspecting schema you also need to bind it on decorator level. Here is an example how to solve this problem:

```
@colander.deferred
def deferred_conn_id(node, kw):
    return kw.get('conn_id') or str(uuid.uuid4())


class SomeSchema(colander.MappingSchema):
    username = colander.SchemaNode(colander.String())
    conn_id = colander.SchemaNode(
        colander.String(), missing=deferred_conn_id)


def rebind_schema(schema):
    """
    Ensure we bind schema per request
    """
    def deferred_validator(request, **kwargs):
        # we need to regenerate the schema here
        kwargs['schema'] = schema().bind(request=request)
        return colander_validator(request, **kwargs)
    return deferred_validator


@legacy_connect_api.post(
    schema=SomeSchema().bind(), validators=(rebind_schema(SomeSchema),))
def connect(request):
    ...
```

2.5 Changelog

2.5.1 CHANGES

0.7.0 (2018-07-29)

- Support swagger `example` field on colander `SchemaNode` custom kwarg:

  ```
  def SomeSchema(colander.MappingSchema):
      name = colander.SchemaNode(colander.String(), example='Mr. IceCream')
  ```

  The `example` field is returned in the swagger spec accordingly.

0.6.0 (2018-03-28)

- Add `cornice_enable_openapi_view()` `cornice_enable_openapi_explorer()` Pyramid directives to serve the API Explorer and the spec information (#79)

0.5.5 (2018-03-19)

- Prevent failure with non-colander schemas (#78, thanks @ergo!)

0.5.4 (2018-02-26)

Internals

- Fix return types in docstrings (#77)

0.5.3 (2018-02-14)

Pyramid compliance

- Handle callables for `cornice.service.Service.content_type` argument. For more details, see: http://cornice.readthedocs.io/en/latest/api.html#cornice.service.Service.

0.5.2 (2017-11-07)

Internals

- Add classifiers to the Python package.

0.5.1 (2017-04-10)

Pyramid compliance

- Support subpaths and regex when parsing paths (#68).

Api

- `_extract_path_from_service`, now returns the path name along with the path swagger object (#68).

0.5.0 (2017-02-14)

Api

- Allow implementing a custom generator by subclassing the `CorniceSwagger` class (#63).
- Introduced a new method `CorniceSwagger.generate` to generate the spec (#63).
- Deprecated `CorniceSwagger` call method. You should now use `generate` (#63).
- Removed deprecated `generate_swagger_spec` call. (#64).
- Allow defining custom type converters on the `CorniceSwagger` class. (#65)

Internals

- Fixed coveralls repeated messages on PRs. (#62).

0.4.0 (2017-01-25)

Api

- Summaries from docstrings are now not included by default. You can enable them by passing `summary_docstrings = True` to the generator.
- Trying to document multiple views on same method now raises an exception. You should ignore the unwanted ones by content type.
- Raw `swagger` items are now recursively merged (instead of replaced) with the extracted fields.

- Add support for documenting operation ids via an `operation_id` argument on the view or by passing a `default_op_ids` callable to the generator.
- Add a shortcut to the generator on `cornice_swagger.CorniceSwagger`.
- Support Cornice schema synonyms (headers and GET are the same as header and querystring).
- Add support for documenting security properties via a `api_security` list on the view or by passing a `default_security` list or callable to the generator.

OpenAPI compliance

- Remove invalid `title` field from response headers and request parameters.
- Support conversion of parameter validators.

Internals

- Fix default tag generator.
- Fix references when using declarative schemas.
- Simplify parameter converter by properly isolating `body`.

0.3.0 (2017-01-17)

Api

- Use `cornice_swagger.swagger.CorniceSwagger` class to generate the swagger document rather then `generate_swagger_spec`.
- Allow overriding extractors in the application.
- Schemas are now broken into JSON pointers only if specified.
- Allow documenting responses via `response_schemas` view attribute.
- Allow documenting tags via `tags` view attribute or using a `default_tags` parameter when calling the generator.

Internals

- Decouples converters from path generators.
- Make considerable changes in the package organisation.
- Reach 100% coverage on tests.

Documentation

- Create a Sphinx documentation hosted on https://cornices.github.io/cornice.ext.swagger.

0.2.1 (2016-12-10)

- Check if schema is not instantiated.
- Add support for query parameter description. [ridha]

0.2 (2016-11-08)

- Pypi release.
- Point scaffold doc to right url.

0.1 (2016-11-05)

- First release for new cornice 2.0

2.6 Contributors

Alphabetically-ordered List of the people who contributed to Cornice Swagger:

- Gabriela Surita <gsurita@mozilla.com>
- Jason Haury
- Josip Delic
- Marcin Lulek
- Simone Marzola <marzolasimone@gmail.com>

CHAPTER 3

Contribution & Feedback

You can find us at Github or the Slack chat.

- GitHub Repository: https://github.com/Cornices/cornice.ext.swagger
- Slack chat: https://corniceswagger.herokuapp.com

You may also try the Cornice Mailing List:

- Cornice Developers Mailing List: https://mail.mozilla.org/listinfo/services-dev

Python Module Index

C

Index

Symbols

A

B

C

D

F

G

I

P

R

S

T