Sanic-OpenAPI

Release 21.3.1

May 11, 2021

Contents

Sanic-OpenAPI is an extension of Sanic web framework to easily document your Sanic APIs with Swagger UI.

CHAPTER 1

Installation

To install `sanic_openapi`, you can install from PyPI:

```
pip install sanic-openapi
```

Or, use master banch from GitHub with latest features:

```
pip install git+https://github.com/sanic-org/sanic-openapi.git
```

CHAPTER 2

Getting started

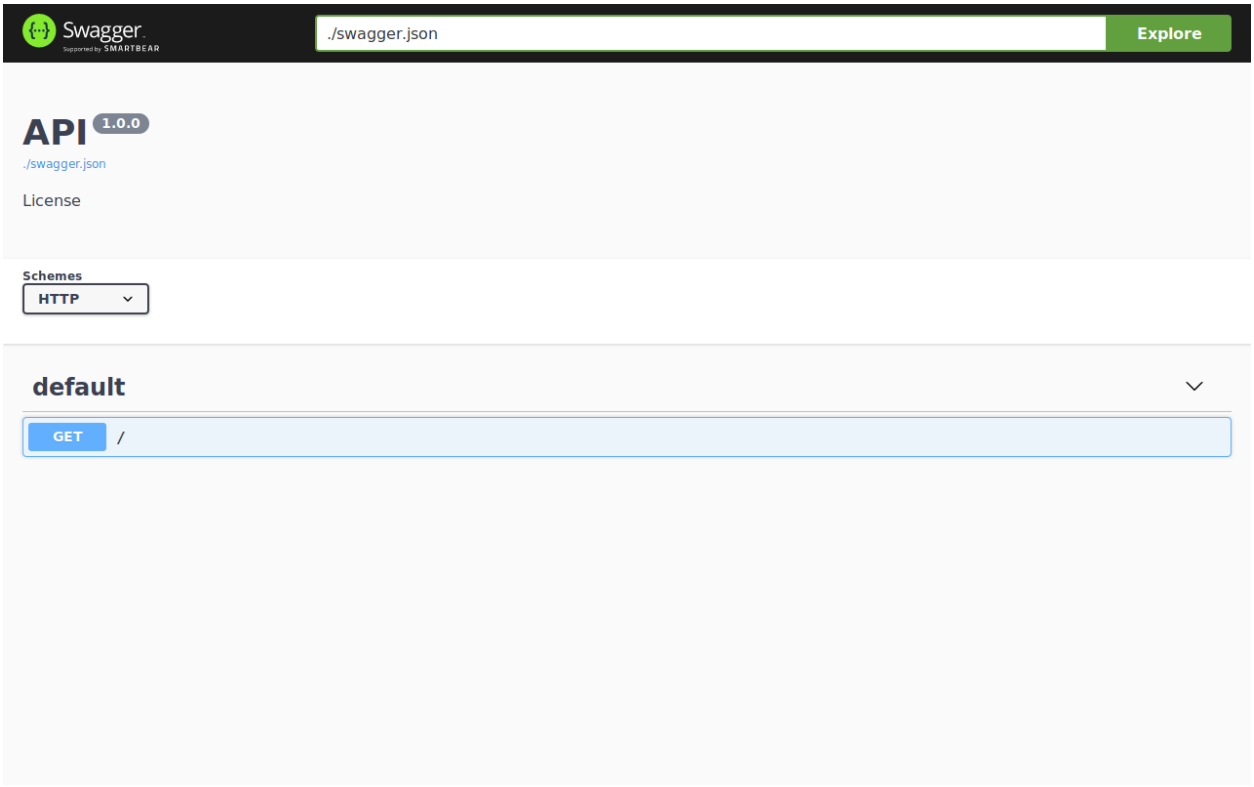
2.1 Sanic OpenAPI 2

2.1.1 Getting started

Here is an example to use Sanic-OpenAPI 2:

```
from sanic import Sanic
from sanic.response import json
from sanic_openapi import openapi2_blueprint

app = Sanic("Hello world")
app.blueprint(openapi2_blueprint)


@app.route("/")
async def test(request):
    return json({"hello": "world"})


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

And you can get your Swagger document at http://localhost:8000/swagger like this:



2.1.2 Contents

Document Routes

Sanic-OpenAPI support different ways to document APIs includes:

- routes of `Sanic` instance
- routes of `Blueprint` instance
- routes of `HTTPMethodView` under `Sanic` instance
- routes of `HTTPMethodView` under `Bluebprint` instance
- routes of `CompositionView` under `Sanic` instance

But with some exceptions:

- Sanic-OpenAPI does not support routes of `CompositionView` under `Bluebprint` instance now.
- Sanic-OpenAPI does not document routes with `OPTIONS` method.
- Sanic-OpenAPI does not document routes which registered by `static()`.

This section will explain how to document routes with above cases.

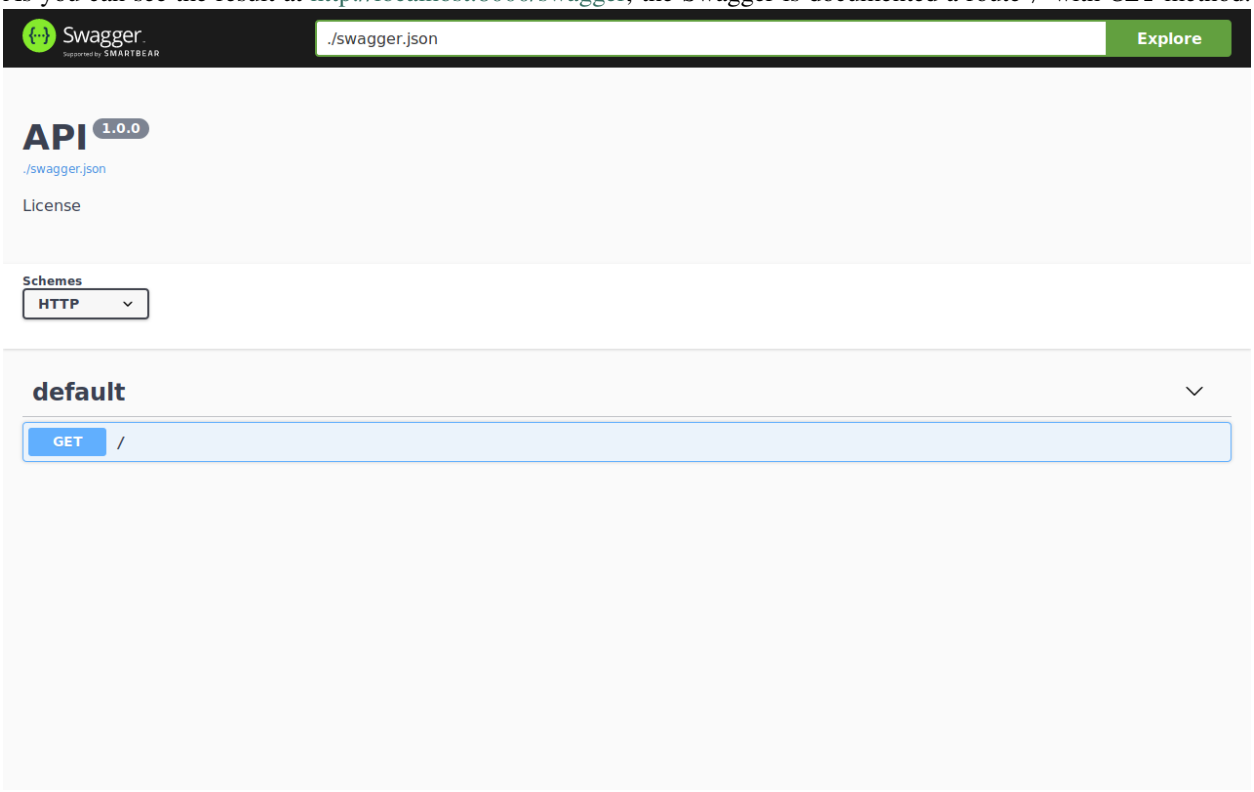
Basic Routes

To use Sanic-OpenAPI with basic routes, you only have to register `openapi2_blueprint` and it will be all set.

For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.route("/")
async def test(request):
    return json({"hello": "world"})


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

As you can see the result at http://localhost:8000/swagger, the Swagger is documented a route `/` with `GET` method.



If you want to add some additional information to this route, you can use other decorators like `summary()`, `description()`, and etc.

Blueprint Routes

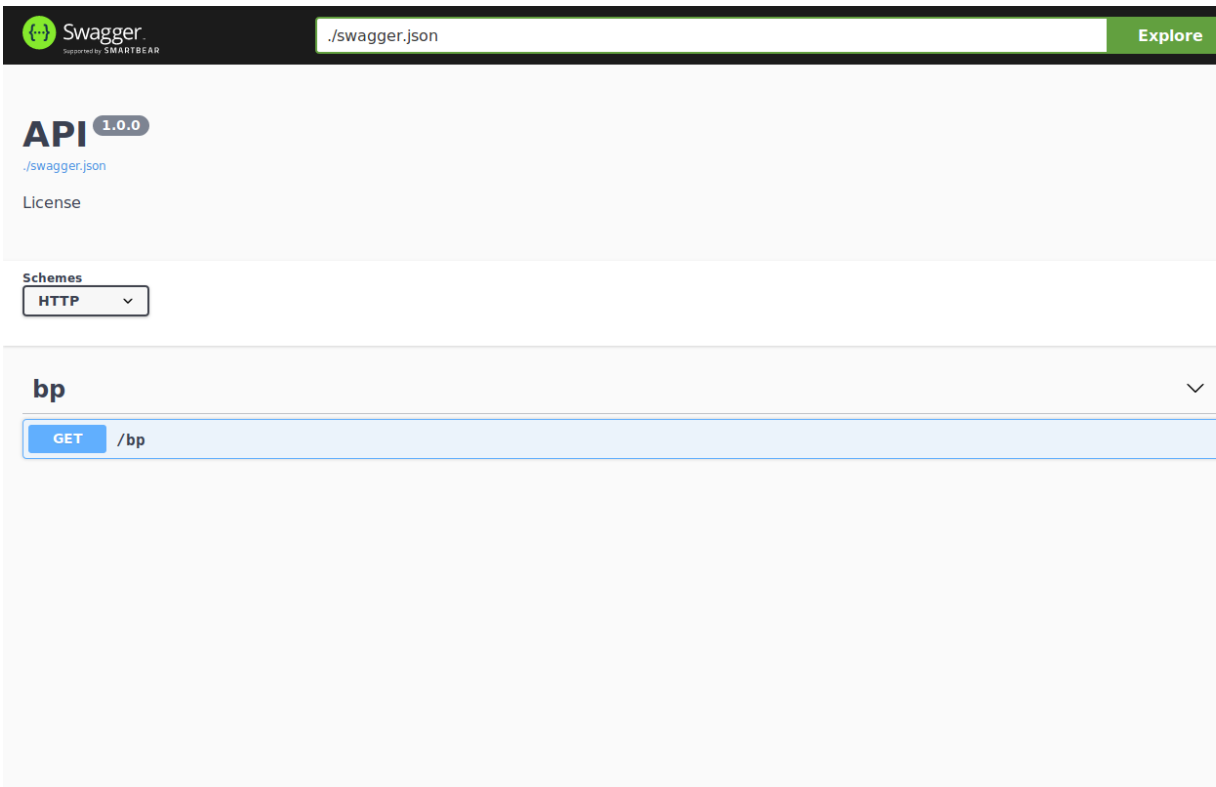
You can aldo document routes under any `Blueprint` like this:

```
from sanic import Blueprint, Sanic
from sanic.response import json
```

(continues on next page)

(continued from previous page)

```
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)

bp = Blueprint("bp", url_prefix="/bp")


@bp.route("/")
async def test(request):
    return json({"hello": "world"})


app.blueprint(bp)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```



The result looks like:

When you document routes under `Blueprint` instance, they will be document with `tags` which using the `Blueprint`'s name.

Class-Based Views Routes

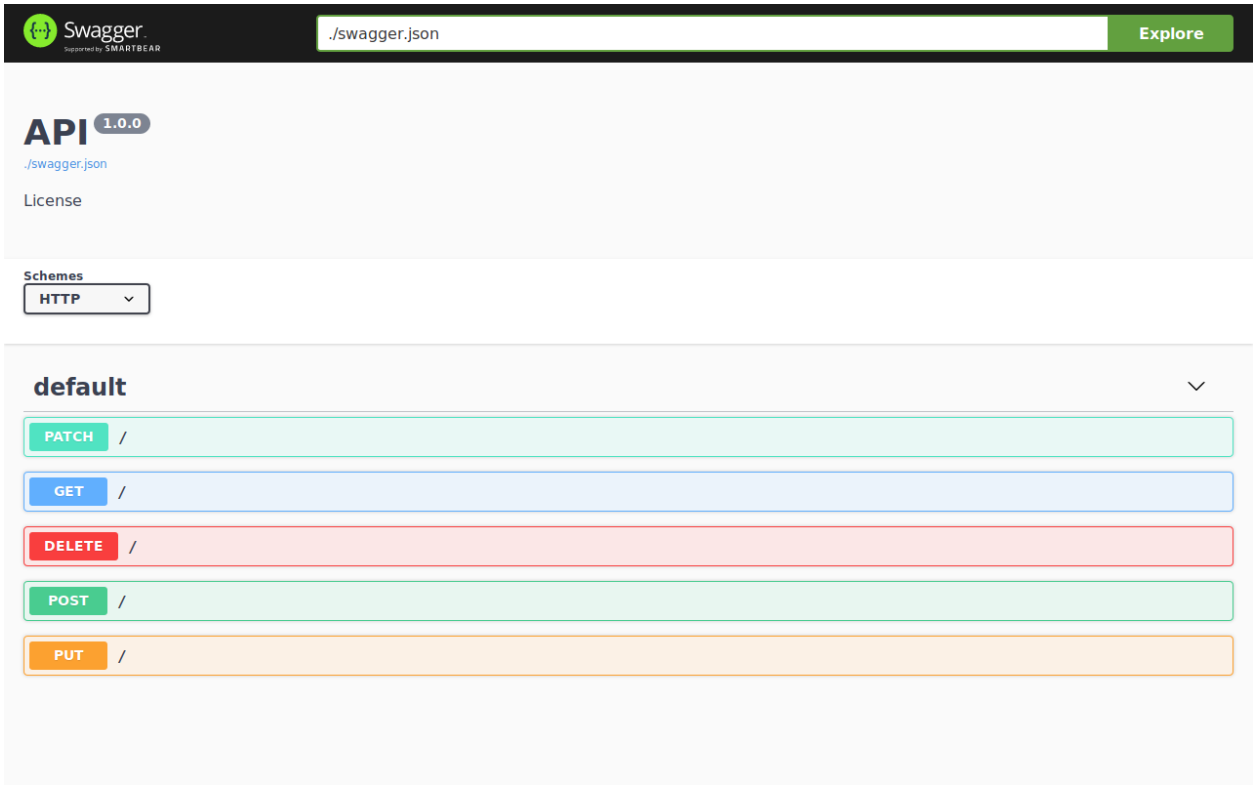
In Sanic, it provides a class-based views named `HTTPMethodView`. You can document routes under `HTTPMethodView` like:

```
from sanic import Sanic
from sanic.response import text
```

(continues on next page)

(continued from previous page)

```
from sanic.views import HTTPMethodView

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class SimpleView(HTTPMethodView):
    def get(self, request):
        return text("I am get method")

    def post(self, request):
        return text("I am post method")

    def put(self, request):
        return text("I am put method")

    def patch(self, request):
        return text("I am patch method")

    def delete(self, request):
        return text("I am delete method")

    def options(self, request): # This will not be documented.
        return text("I am options method")


app.add_route(SimpleView.as_view(), "/")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```



And the result:

Please note that Sanic-OpenAPI will not document any routes with `OPTIONS` method.

The `HTTPMethodView` can also be registered under `Blueprint`:

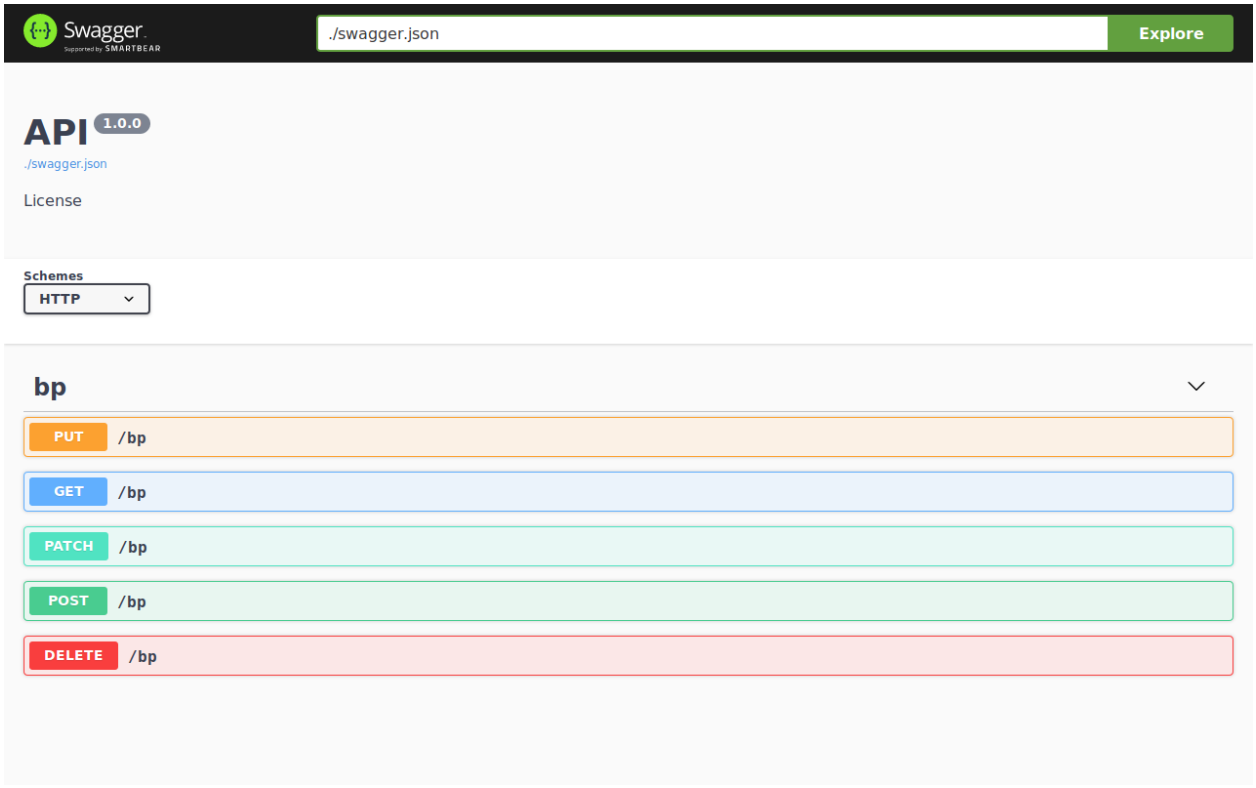
```
from sanic import Blueprint, Sanic
from sanic.response import text
from sanic.views import HTTPMethodView

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)

bp = Blueprint("bp", url_prefix="/bp")


class SimpleView(HTTPMethodView):
    def get(self, request):
        return text("I am get method")

    def post(self, request):
        return text("I am post method")

    def put(self, request):
        return text("I am put method")

    def patch(self, request):
        return text("I am patch method")

    def delete(self, request):
        return text("I am delete method")
```

(continues on next page)

(continued from previous page)

```
    def options(self, request):  # This will not be documented.
        return text("I am options method")


bp.add_route(SimpleView.as_view(), "/")
app.blueprint(bp)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

The result:



CompositionView Routes

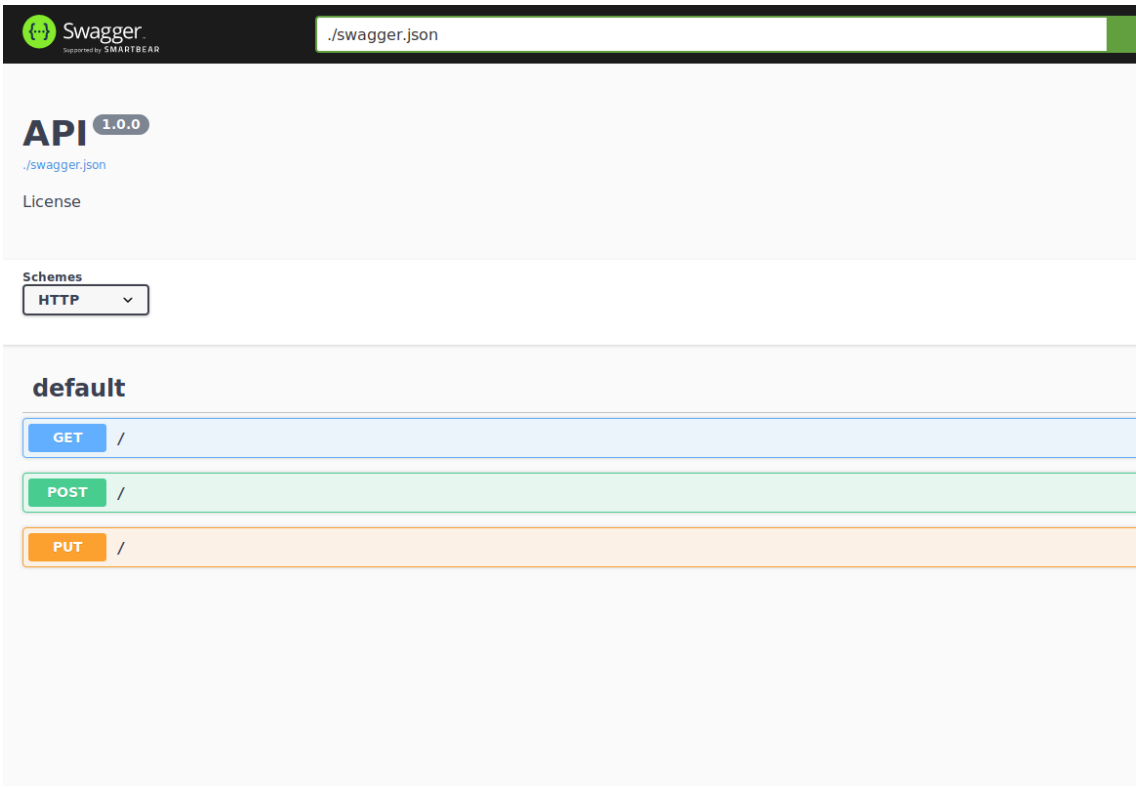
There is another class-based view named `CompositionView`. Sanic-OpenAPI also support to document routes under class-based view.

```
from sanic import Sanic
from sanic.response import text
from sanic.views import CompositionView

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


def get_handler(request):
```

(continues on next page)

(continued from previous page)

```
    return text("I am a get method")


view = CompositionView()
view.add(["GET"], get_handler)
view.add(["POST", "PUT"], lambda request: text("I am a post/put method"))

# Use the new view to handle requests to the base URL
app.add_route(view, "/")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```



The Swagger will looks like:

Note: Sanic-OpenAPI does not support routes of *CompositionView* under *Bluebprint* instance now.

Configurations

Sanic-OpenAPI provides following configurable items:

- API Server
- API information
- Authentication(Security Definitions)
- URI filter
- Swagger UI configurations

API Server

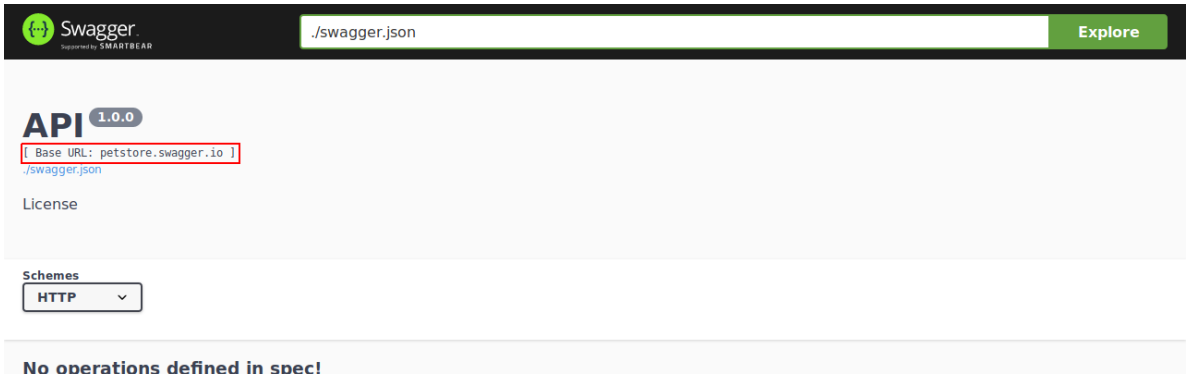
By default, Swagger will use exactly the same host which served itself as the API server. But you can still override this by setting following configurations. For more information, please check document at here.

API_HOST

- Key: `API_HOST`
- Type: `str` of IP, or hostname
- Default: `None`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_HOST"] = "petstore.swagger.io"
```

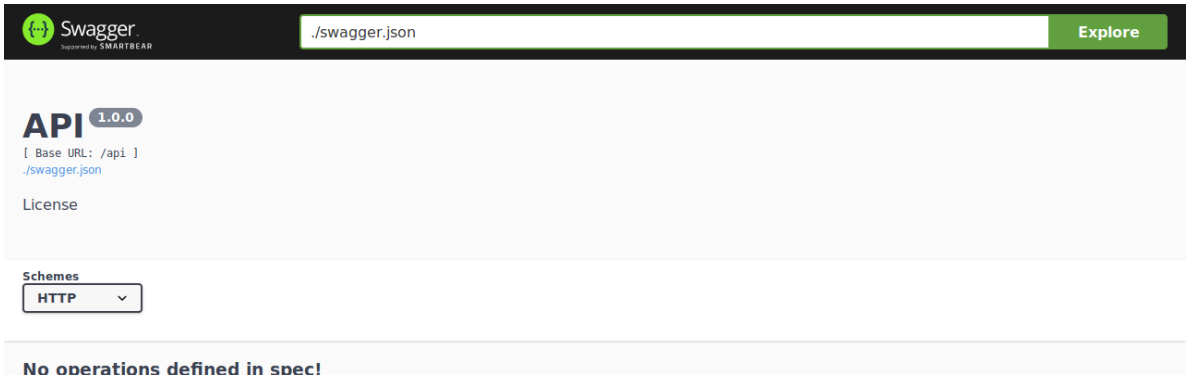
- Result:



API_BASEPATH

- Key: `API_BASEPATH`
- Type: `str`
- Default: `None`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_BASEPATH"] = "/api"
```

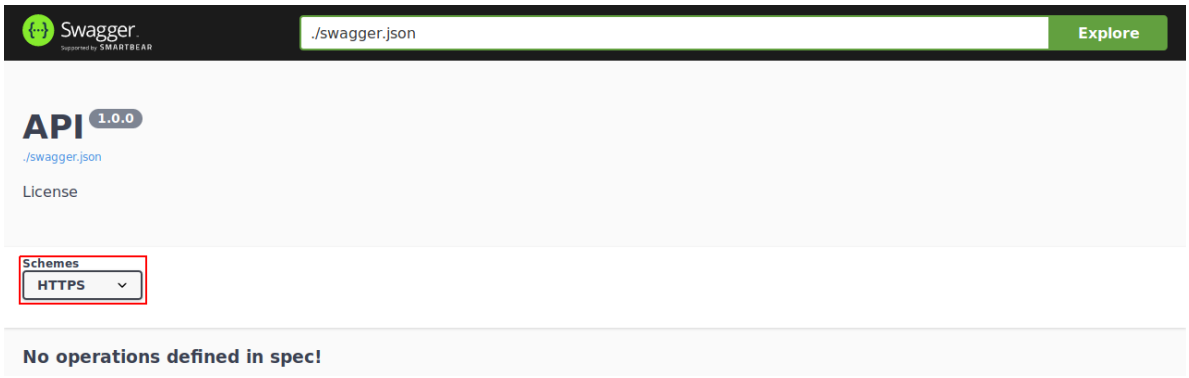


- Result: No operations defined in spec!

API_SCHEMES

- Key: `API_SCHEMES`
- Type: `list` of schemes
- Default: `["http"]`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_SCHEMES"] = ["https"]
```



- Result: No operations defined in spec!

API information

You can provide some additional information of your APIs by using Sanic-OpenAPI configurations. For more detail of those additional information, please check the document from Swagger.

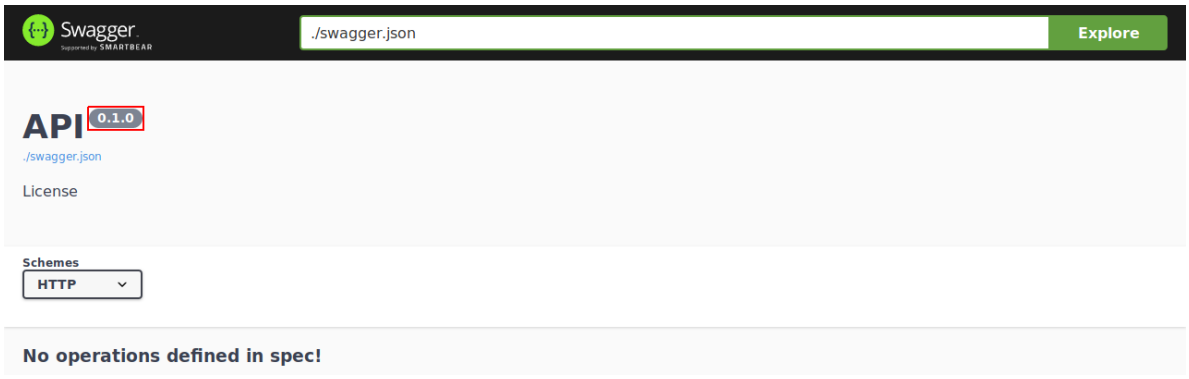
API_VERSION

- Key: `API_VERSION`
- Type: `str`
- Default: `1.0.0`

- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_VERSION"] = "0.1.0"
```

- Result:

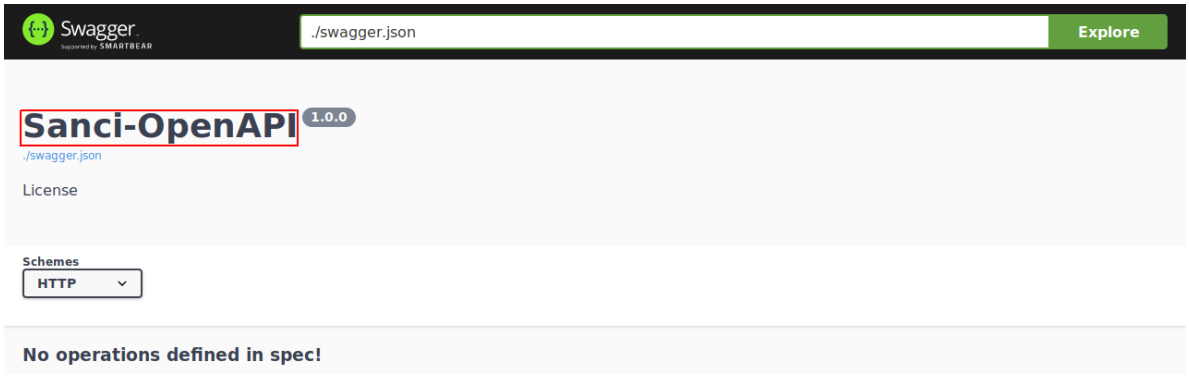


API_TITLE

- Key: `API_TITLE`
- Type: `str`
- Default: `API`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_TITLE"] = "Sanic-OpenAPI"
```

- Result:



API_DESCRIPTION

- Key: `API_DESCRIPTION`

- Type: `str`
- Deafult: `""`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_DESCRIPTION"] = "An example Swagger from Sanic-OpenAPI"
```

- Result:

API_TERMS_OF_SERVICE

- Key: `API_TERMS_OF_SERVICE`
- Type: `str` of a URL
- Deafult: `""`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_TERMS_OF_SERVICE"] = "https://github.com/sanic-org/sanic-
↪openapi/blob/master/README.md"
```

- Result:

API_CONTACT_EMAIL

- Key: `API_CONTACT_EMAIL`
- Type: `str` of email address
- Deafult: `None"`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_CONTACT_EMAIL"] = "foo@bar.com"
```

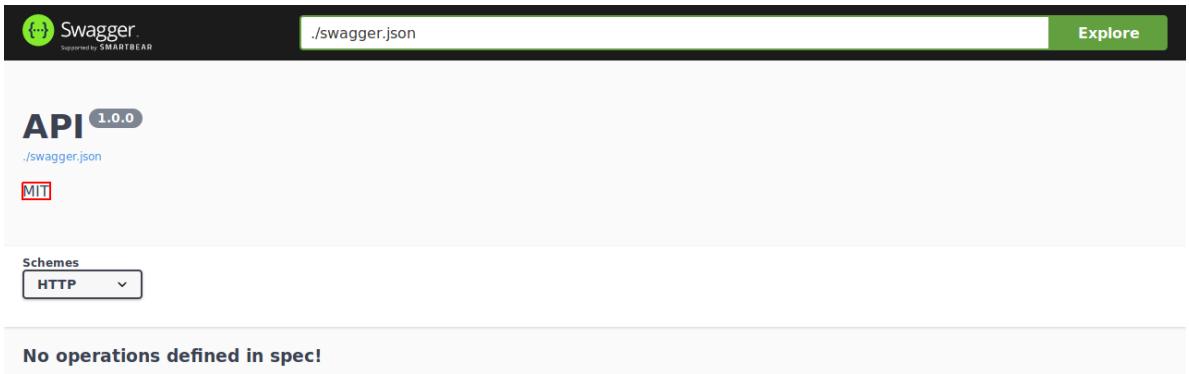
- Result:

API_LICENSE_NAME

- Key: `API_LICENSE_NAME`
- Type: `str`
- Default: `None`
- Usage:

  python from sanic import Sanic from sanic_openapi import openapi2_blueprint

  app = Sanic() app.blueprint(openapi2_blueprint) app.config[”API_LICENSE_NAME”] = “MIT”



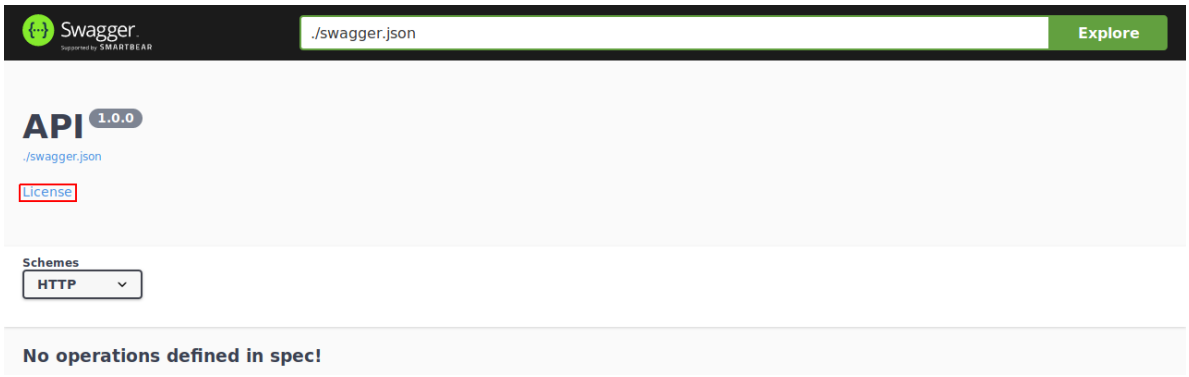
- Result:

API_LICENSE_URL

- Key: `API_LICENSE_URL`
- Type: `str` of URL
- Default: `None`
- Usgae:

```
from sanic import Sanic
from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_LICENSE_URL"] = "https://github.com/sanic-org/sanic-openapi/blob/
↪master/LICENSE"
```

- Result:



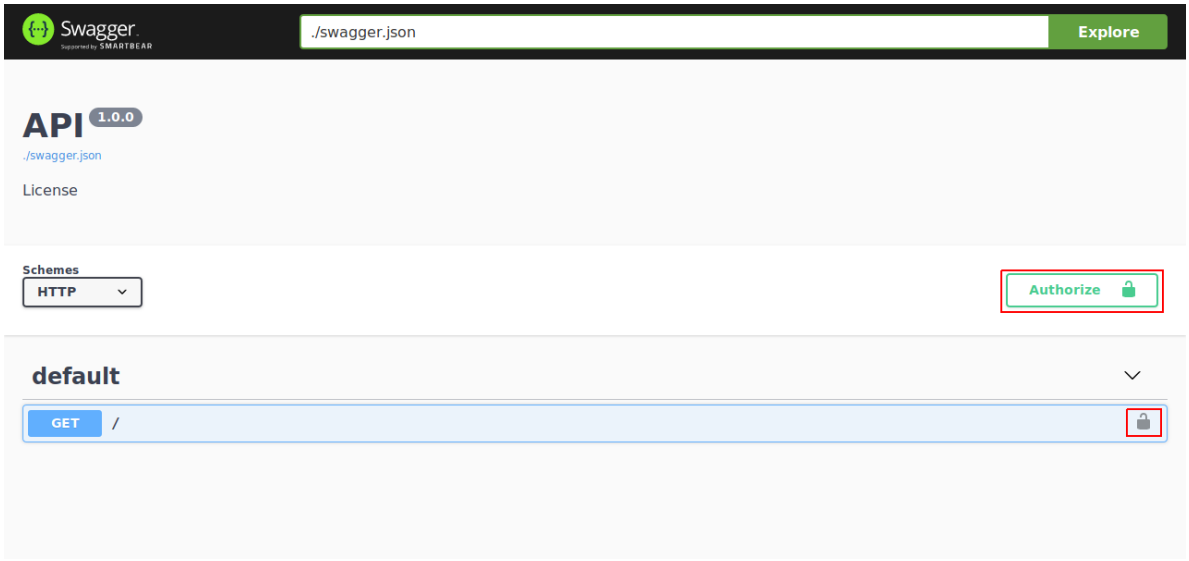
Authentication

If your API have to access with authentication, Swagger can provide related configuration as you need. For more information, check here.

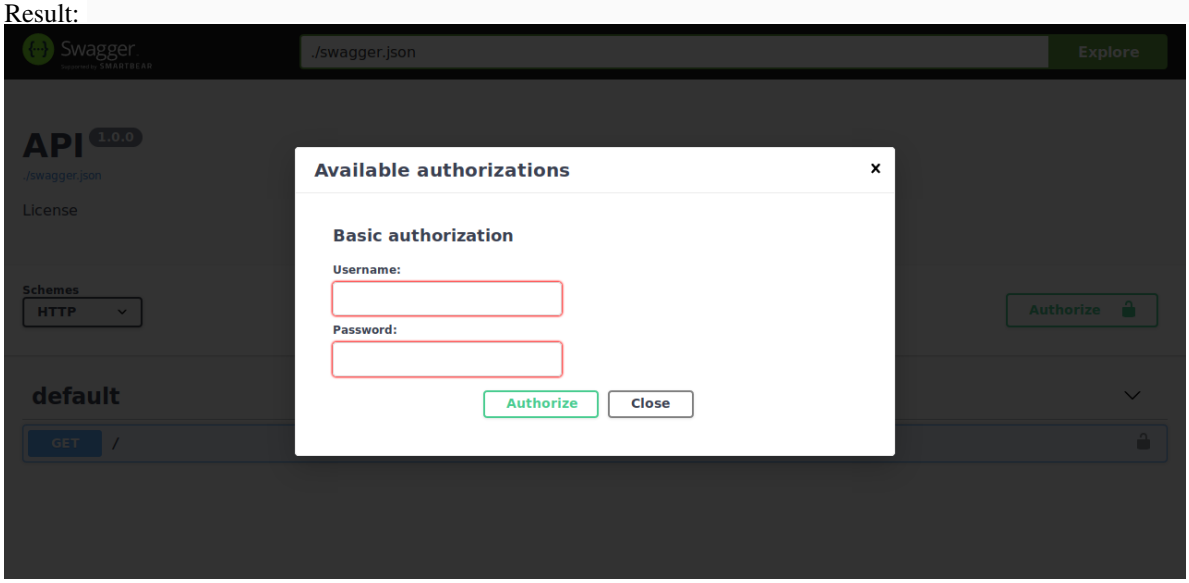
Basic Authentication

- Usage:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_SECURITY"] = [{"BasicAuth": []}]
app.config["API_SECURITY_DEFINITIONS"] = {"BasicAuth": {"type": "basic"}}


@app.get("/")
async def test(request):
    return json({"token": request.token})
```



- Result:



API Key

In Header

- Usage:

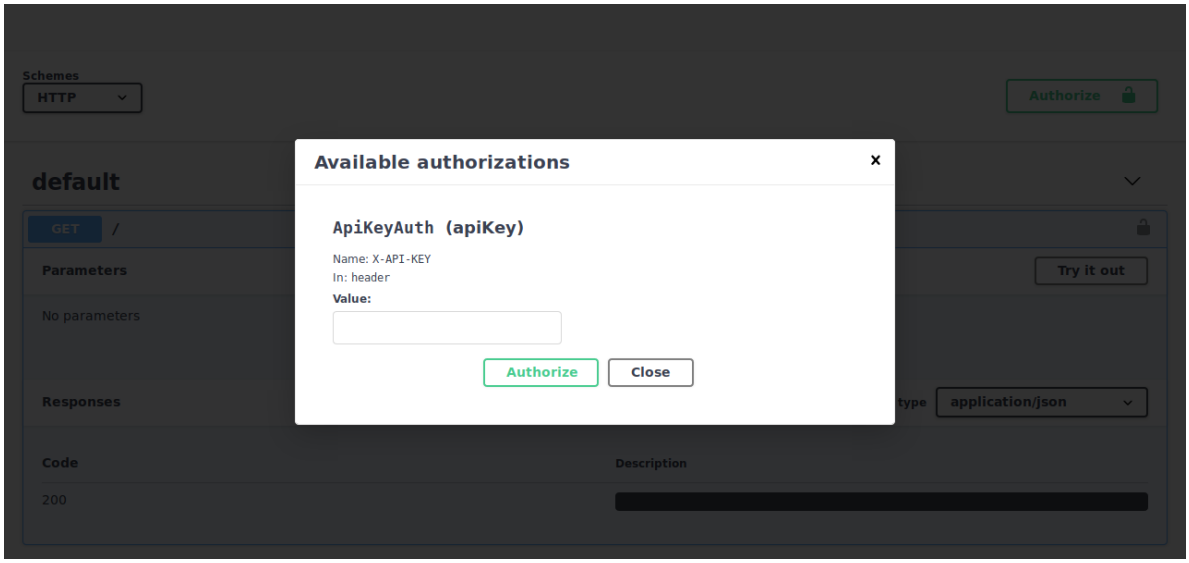
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_SECURITY"] = [{"ApiKeyAuth": []}]
app.config["API_SECURITY_DEFINITIONS"] = {
    "ApiKeyAuth": {"type": "apiKey", "in": "header", "name": "X-API-KEY"}
}
```

(continues on next page)

(continued from previous page)

```
@app.get("/")
async def test(request):
    api_key = request.headers.get("X-API-KEY")
    return json({"api_key": api_key})
```

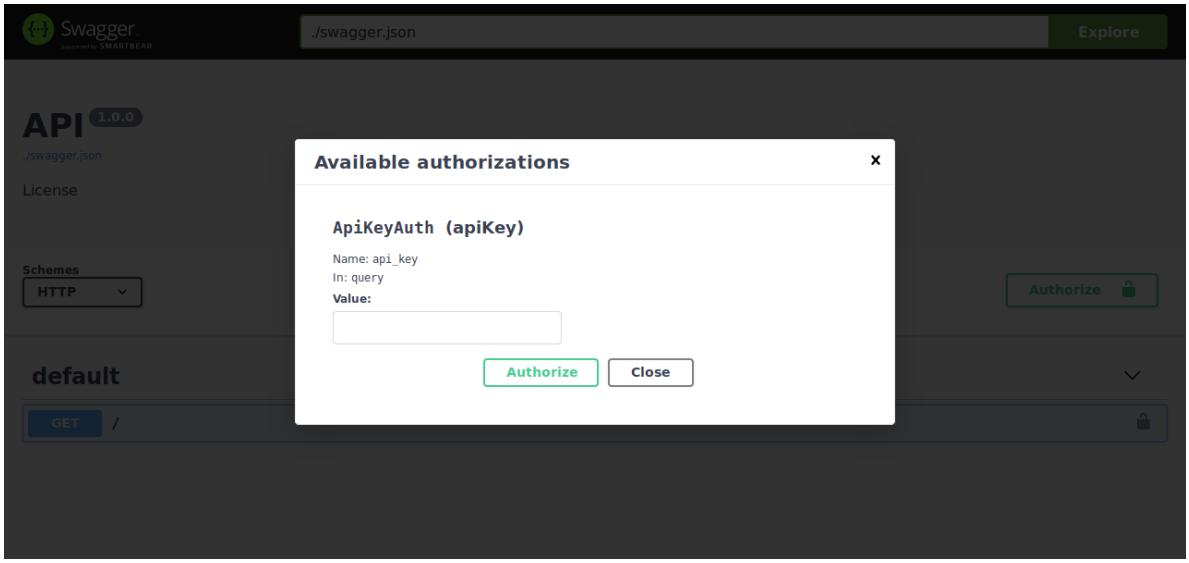
- Result:



In Query

- Usage:

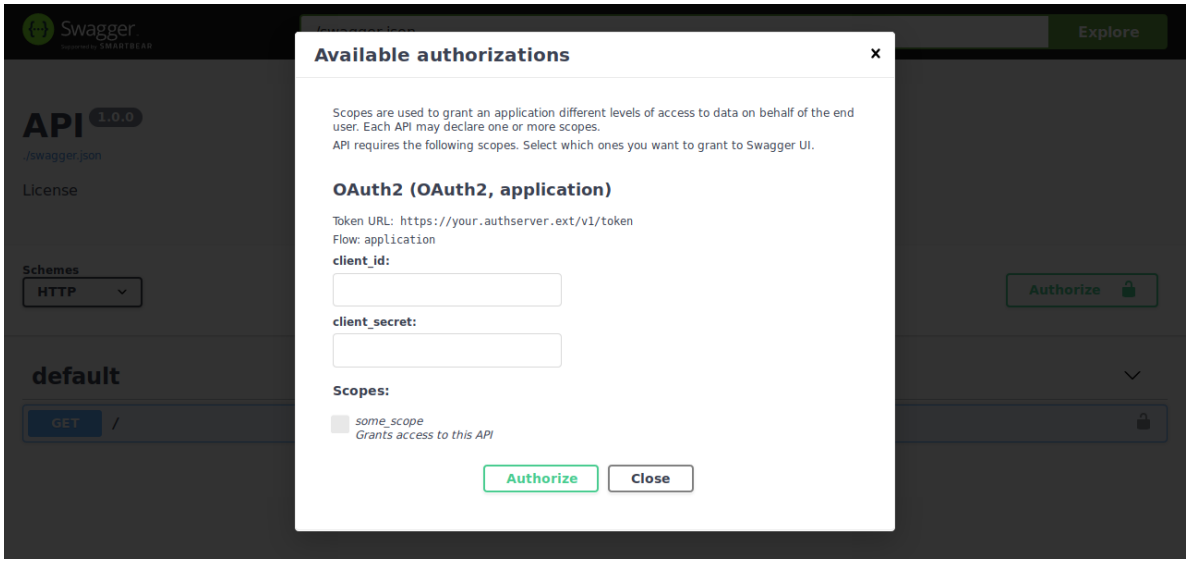
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_SECURITY"] = [{"ApiKeyAuth": []}]
app.config["API_SECURITY_DEFINITIONS"] = {
    "ApiKeyAuth": {"type": "apiKey", "in": "query", "name": "api_key"}
}


@app.get("/")
async def test(request):
    api_key = request.args.get("api_key")
    return json({"api_key": api_key})
```

- Result: 

OAuth2

- Usage:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_SECURITY"] = [{"OAuth2": []}]
app.config["API_SECURITY_DEFINITIONS"] = {
    "OAuth2": {
        "type": "oauth2",
        "flow": "application",
        "tokenUrl": "https://your.authserver.ext/v1/token",
        "scopes": {"some_scope": "Grants access to this API"},
    }
}
```

- Result: 

URI filter

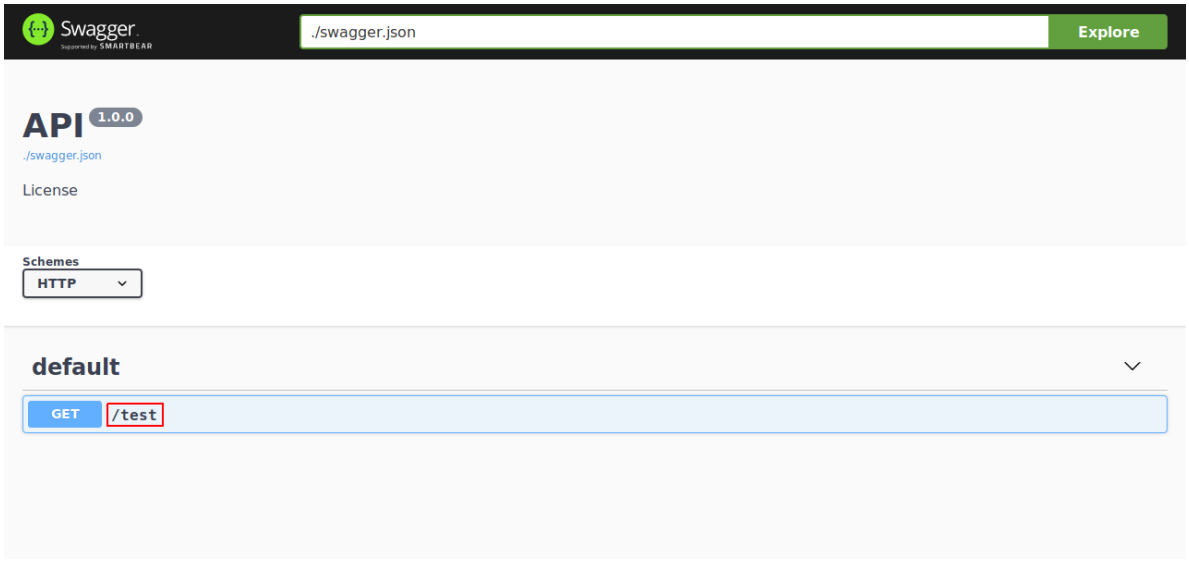
By default, Sanic registers URIs both with and without a trailing `/`. You may specify the type of the shown URIs by setting `app.config.API_URI_FILTER` to one of the following values:

- `all`: Include both types of URIs.
- `slash`: Only include URIs with a trailing `/`.
- All other values (and default): Only include URIs without a trailing `/`.

Non-Slash

- Usage:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
async def test(request):
    return json({"Hello": "World"})
```

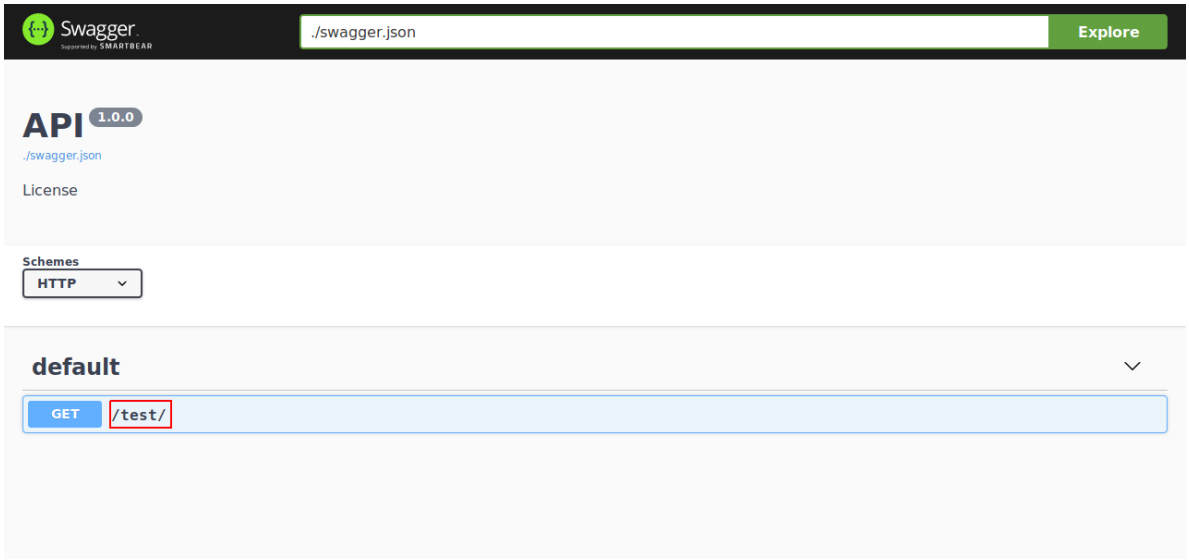


- Result:

Slash

- Usage:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_URI_FILTER"] = "slash"


@app.get("/test")
async def test(request):
    return json({"Hello": "World"})
```



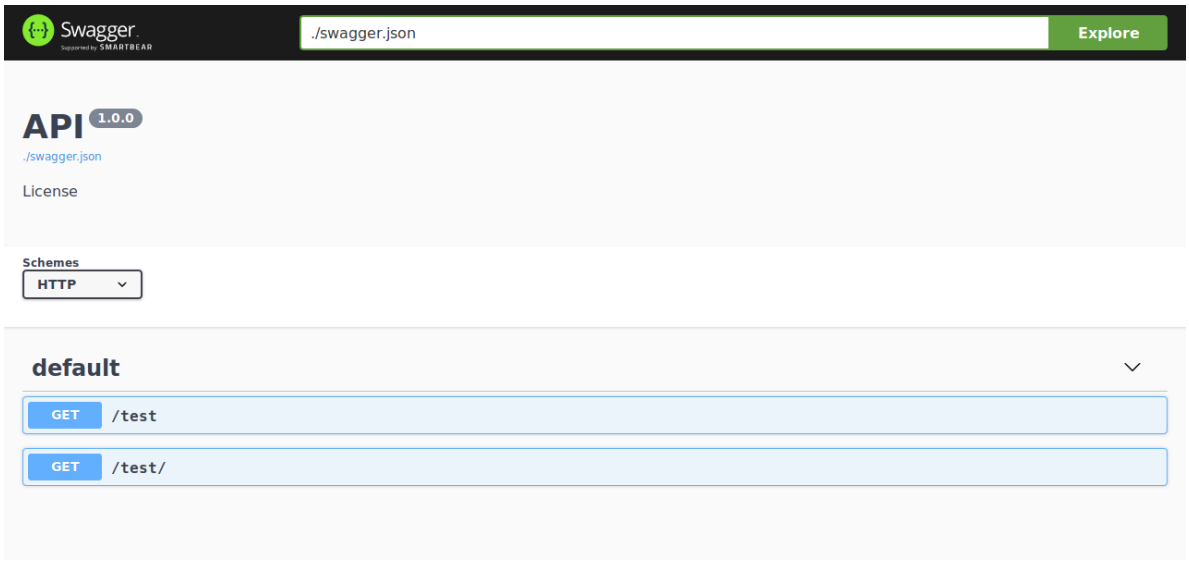
- Result:

All

- Usage:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)
app.config["API_URI_FILTER"] = "all"


@app.get("/test")
async def test(request):
    return json({"Hello": "World"})
```

- Result:



Swagger UI configurations

Here you can set any configuration described in the Swagger UI documentation.

```
app.config.SWAGGER_UI_CONFIGURATION = {
    'validatorUrl': None, # Disable Swagger validator
    'displayRequestDuration': True,
    'docExpansion': 'full'
}
```

Decorators

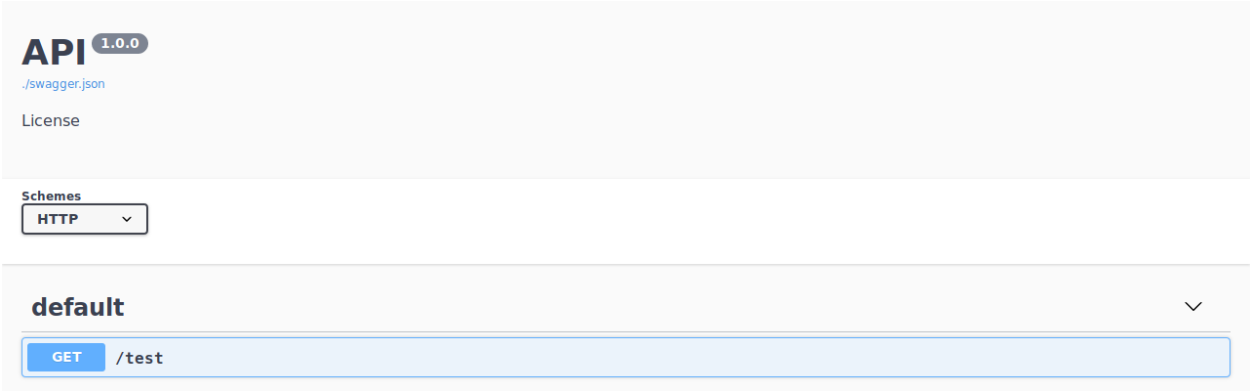
Sanic-OpenAPI provides different **decorator** can help you document your API routes.

Exclude

When you don't want to document some route in Swagger, you can use `exclude(True)` decorator to exclude route from swagger.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
async def test(request):
    return json({"Hello": "World"})


@app.get("/config")
@doc.exclude(True)
async def get_config(request):
    return json(request.app.config)
```

Once you add the `exclude()` decorator, the route will not be document at swagger.



Summary

You can add a short summary to your route by using `summary()` decorator. It is helpful to point out the purpose of your API route.

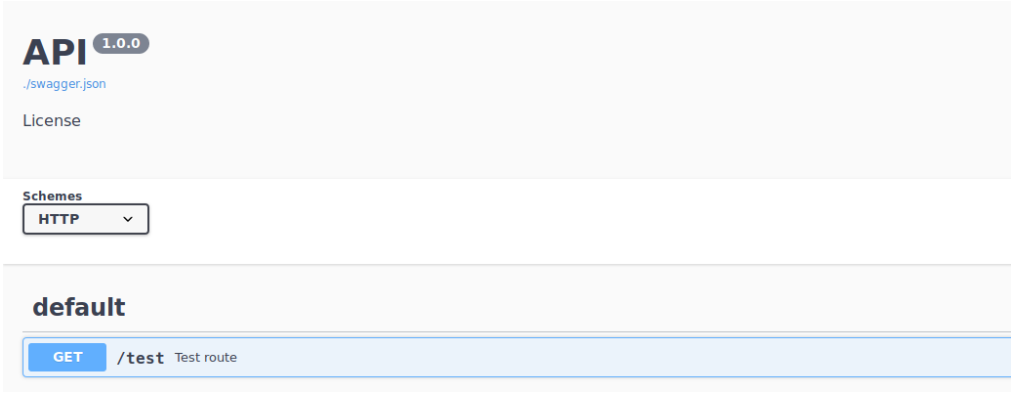
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.summary("Test route")
```

(continues on next page)

(continued from previous page)

```
async def test(request):
    return json({"Hello": "World"})
```

The summary will show behind the path:

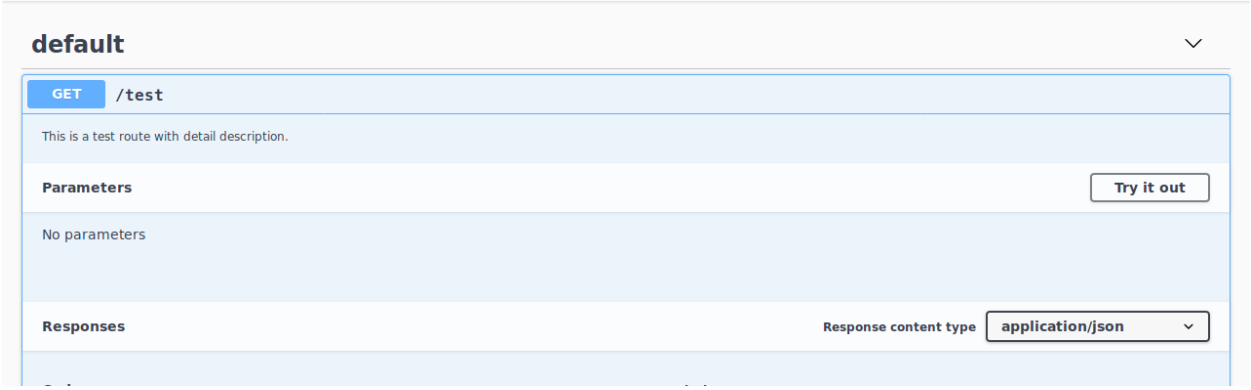


Description

Not only short summary, but also long description of your API route can be addressed by using `description()` decorator.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.description('This is a test route with detail description.')
async def test(request):
    return json({"Hello": "World"})
```

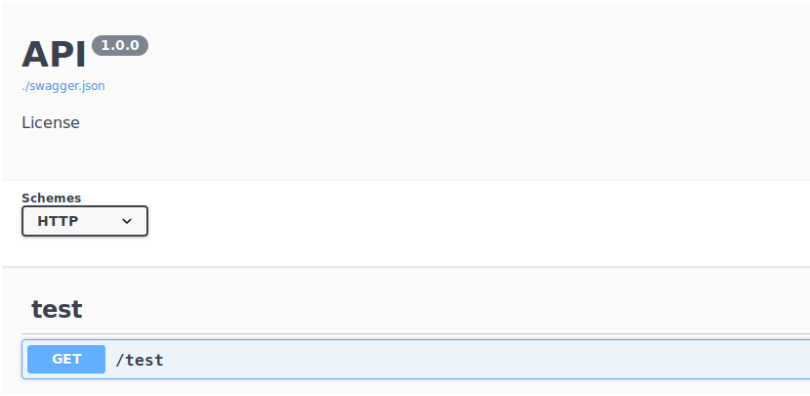
To see the description, you have to expand the content of route and it would looks like:



Tag

If you want to group your API routes, you can use `tag()` decorator to accomplish your need.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.tag("test")
async def test(request):
    return json({"Hello": "World"})
```



And you can see the tag is change from `default` to `test`:

By default, all routes register under Sanic will be tag with `default`. And all routes under Blueprint will be tag with the blueprint name.

Operation

Sanic-OpenAPI will use route(function) name as the default `operationId`. You can override the `operationId` by using `operation()` decorator. The `operation()` decorator would be useful when your routes have duplicate name in some cases.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.operation('test1')
async def test(request):
    return json({"Hello": "World"})
```

Consumes

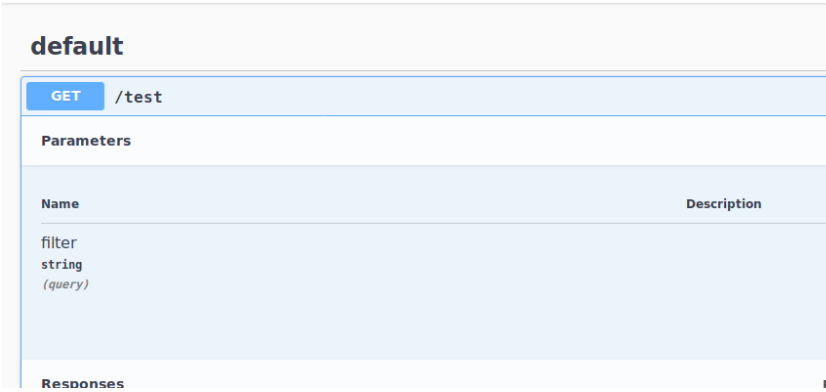
The `consumes()` decorator is the most common used decorator in Sanic-OpenAPI. It is used to document the parameter usages in swagger. You can use built-in classes like `str`, `int`, `dict` or use different fields which provides by Sanic-OpenAPI to document your parameters.

There are three kinds of parameter usages:

Query

To document the parameter in query string, you can use `location="query"` in `consumes()` decorator. This is also the default to `consumes()` decorator.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.consumes(doc.String(name="filter"), location="query")
async def test(request):
    return json({"Hello": "World"})
```



You can expand the contents of route and it will looks like:

When using `consumes()` with `location="query"`, it only support simple types like `str`, `int` but no complex types like `dict`.

Header

For doucument parameters in header, you can set `location="header"` with simple types just like `location="query"`.

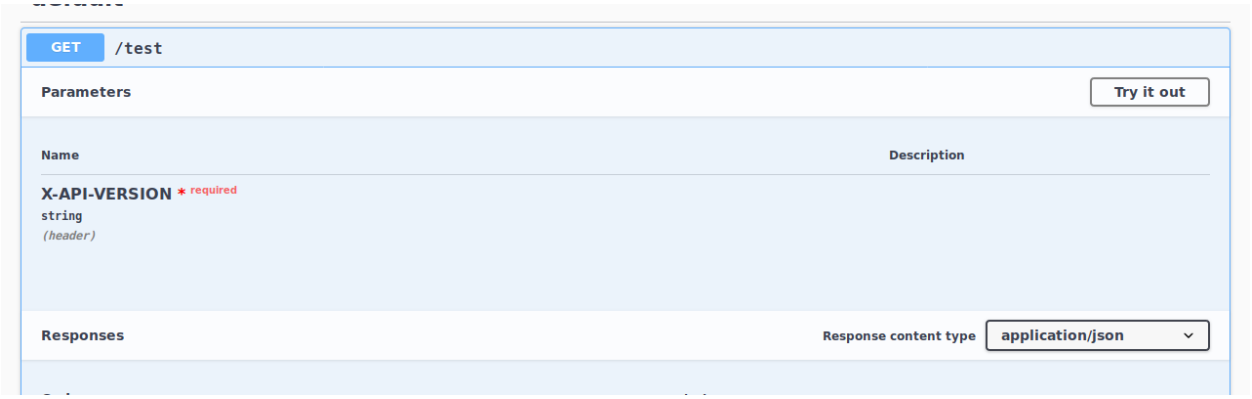
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

```

(continues on next page)

(continued from previous page)

```
app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.consumes(doc.String(name="X-API-VERSION"), location="header", required=True)
async def test(request):
    return json({"Hello": "World"})
```

It will looks like:

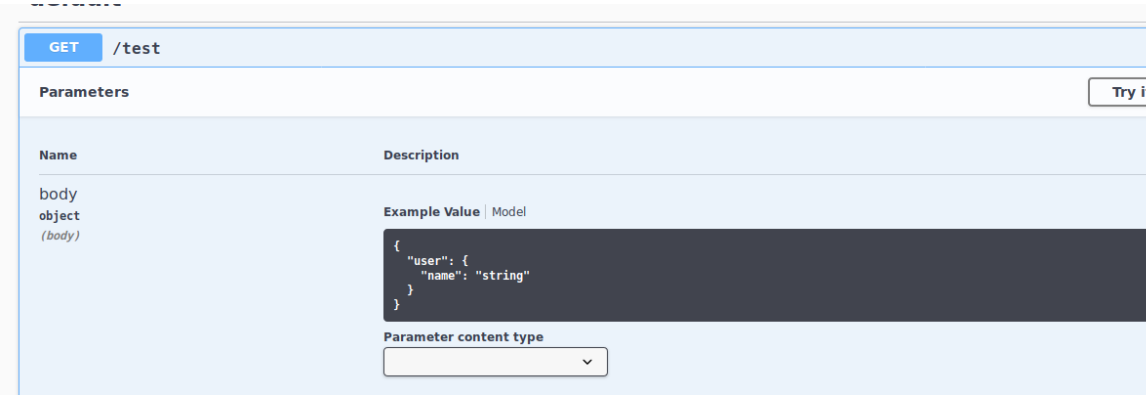


Request Body

In most cases, your APIs might contains lots of parameter in your request body. In Sanic-OpenAPI, you can define them in Python class or use fields which provides by Sanic-OpenAPI to simplify your works.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class User:
    name = str


class Test:
    user = doc.Object(User)


@app.get("/test")
@doc.consumes(Test, location="body")
async def test(request):
    return json({"Hello": "World"})
```

This will be document like:

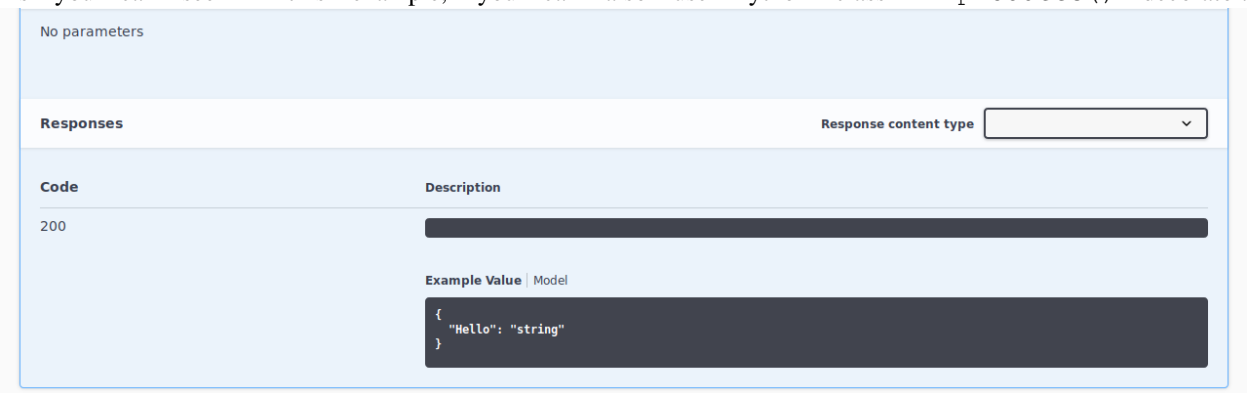


Produces

The `produces()` decorator is used to document the default response(with status 200).

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class Test:
    Hello = doc.String(description='World')

@app.get("/test")
@doc.produces(Test)
async def test(request):
    return json({"Hello": "World"})
```

As you can see in this example, you can also use Python class in `produces()` decorator.



Response

To document responses not with status `200`, you can use `response()` decorator. For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.response(401, {"message": str}, description="Unauthorized")
async def test(request):
    return json({"Hello": "World"})
```

And the responses will be:

Please note that when you use `response()` and `produces()` decorators together, the `response()` decorator with status 200 will have no effect.

Fields

In Sanic-OpenAPI, there are lots of fields can be used to document your APIs. Those fields can represent different data type in your API request and response.

Currently, Sanic-OpenAPI provides following fileds:

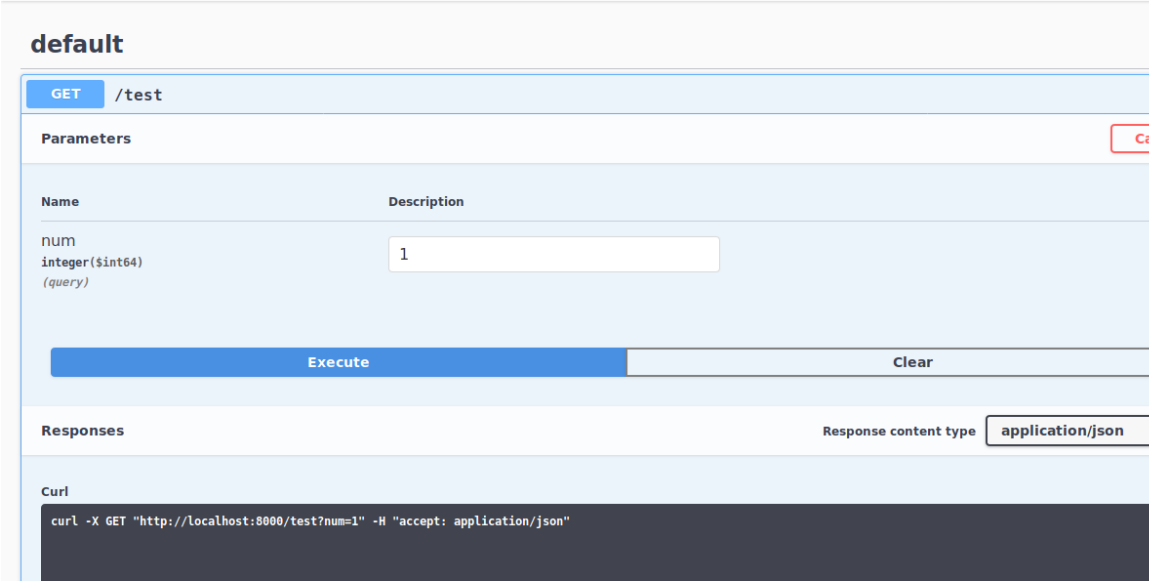
- *Integer*
- *Float*
- *String*
- *Boolean*
- *Tuple*
- *Date*
- *DateTime*
- *File*
- *Dictionary*
- *JsonBody*
- *List*
- *Object*

Integer

To document your API with integer data type, you can use `int` or `doc.Integer` with your handler function. For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.consumes(doc.Integer(name="num"), location="query")
async def test(request):
    return json({"Hello": "World"})
```

And the swagger would be:



Float

Using the `float` or `doc.Float` is quite similar with `doc.integer`:

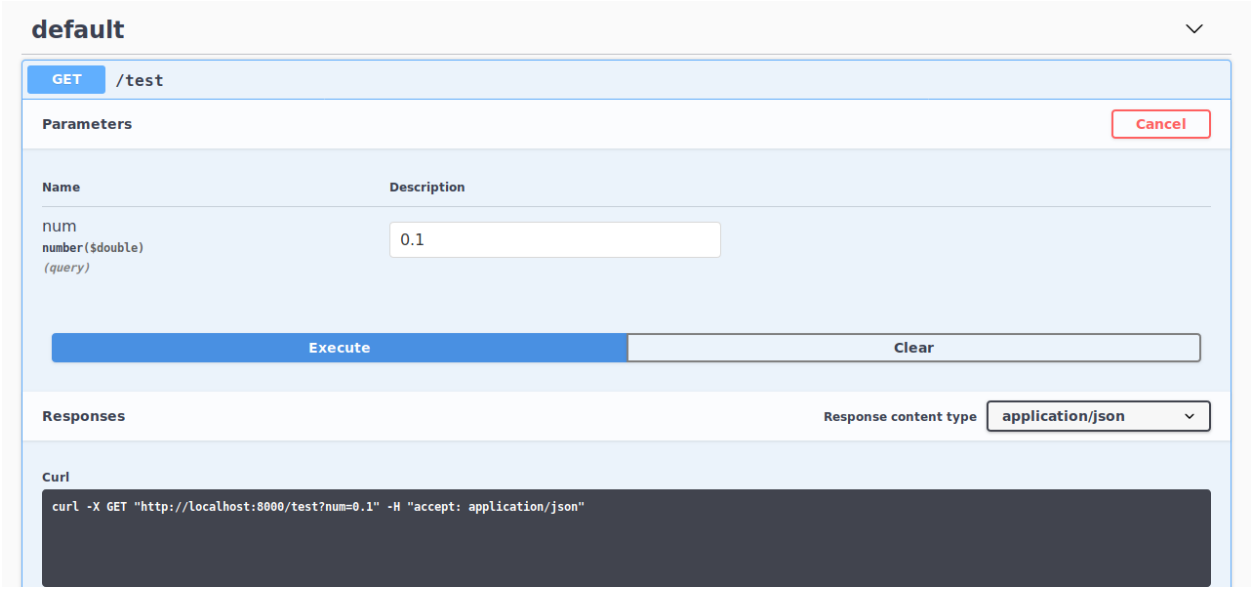
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.consumes(doc.Float(name="num"), location="query")
```

(continues on next page)

(continued from previous page)

```
async def test(request):
    return json({"Hello": "World"})
```

The swagger:

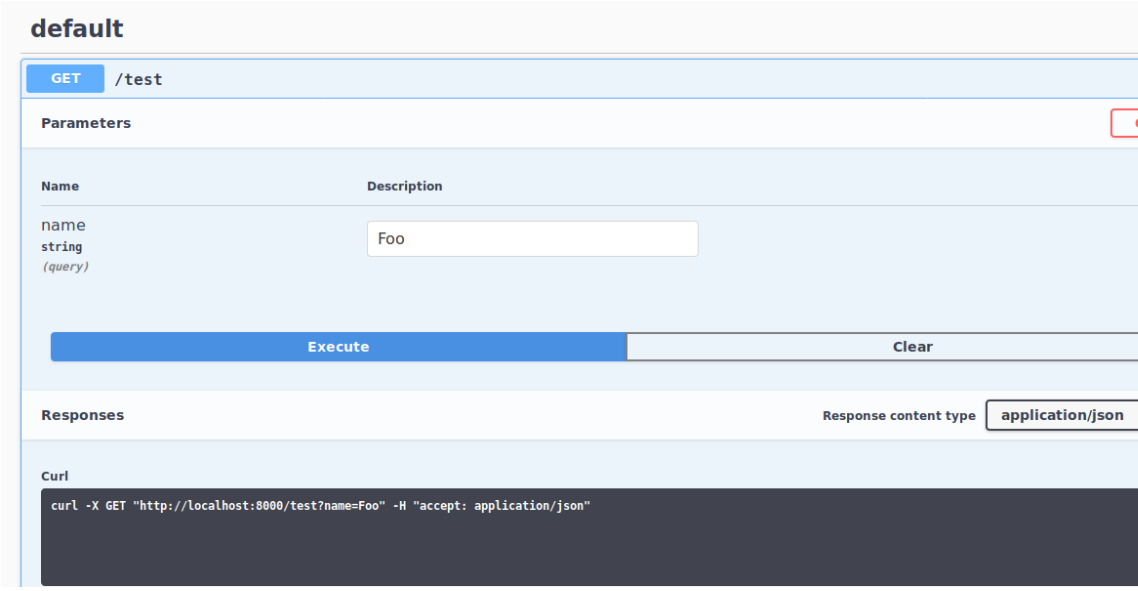


String

The `doc.String` might be the most common filed in API documents. You can use it like this:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.consumes(doc.String(name="name"), location="query")
async def test(request):
    return json({"Hello": "World"})
```

The swagger will looks like:



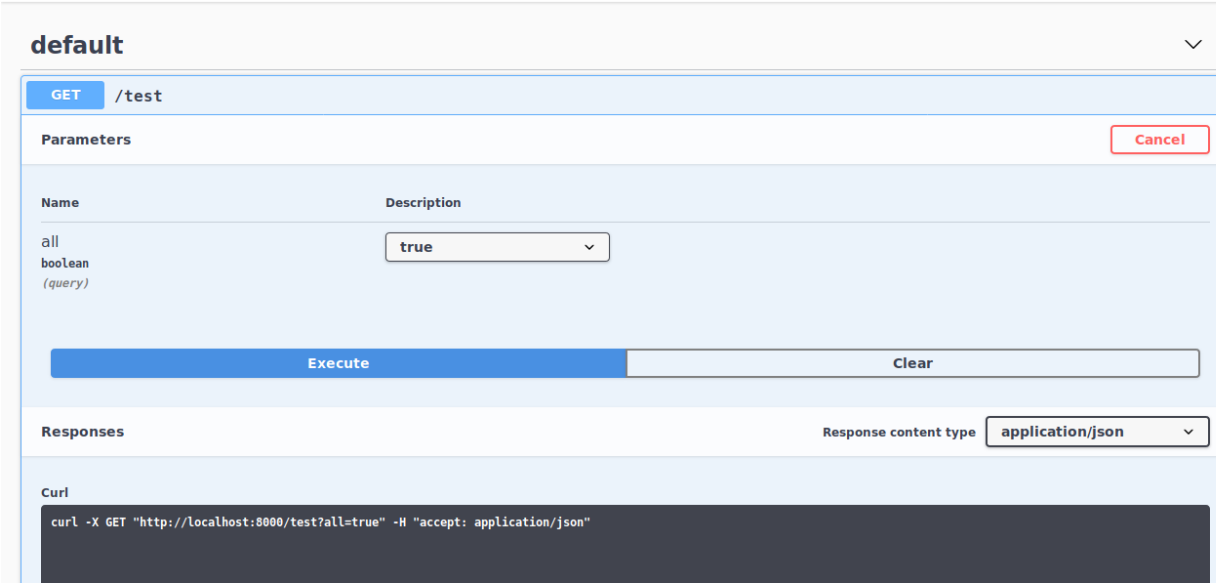
Boolean

If you want to provide an `true` or `false` options in your API document, the `doc.Boolean` is what you need. When using `doc.Boolean` or `bool`, it will be convert in to a dropdown list with `true` and `false` options in swagger.

For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.consumes(doc.Boolean(name="all"), location="query")
async def test(request):
    return json({"Hello": "World"})
```

The swagger will be:



Tuple

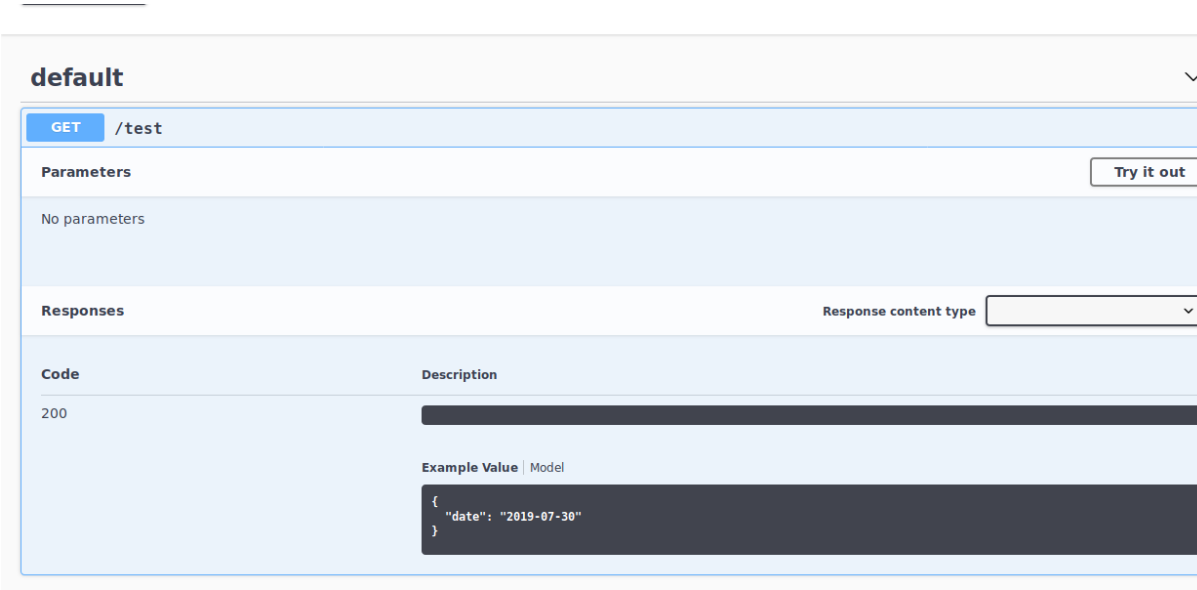
To be done.

Date

To repersent the date data type, Sanic-OpenAPI also provides `doc.Date` to you. When you put `doc.Date` in `doc.produces()`, it will use the local date as value.

```
from datetime import datetime

from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.produces({"date": doc.Date()})
async def test(request):
    return json({"date": datetime.utcnow().date().isoformat()})
```

The example swagger:

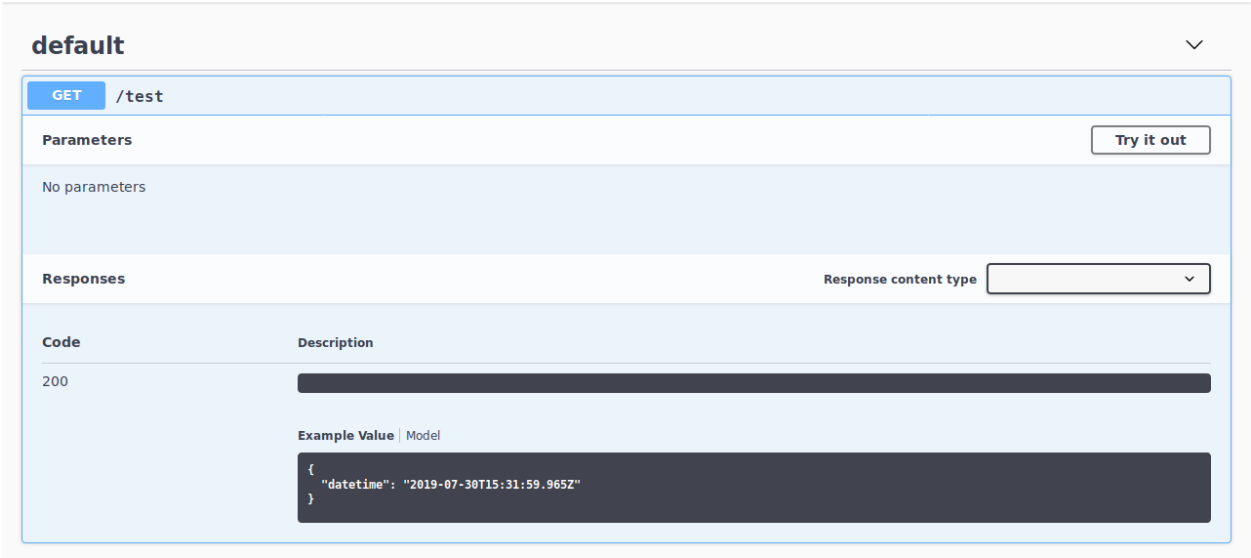


DateTime

Just like `doc.Date`, you can also use the `doc.DateTime` like this:

```
from datetime import datetime

from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.get("/test")
@doc.produces({"datetime": doc.DateTime()})
async def test(request):
    return json({"datetime": datetime.utcnow().isoformat()})
```

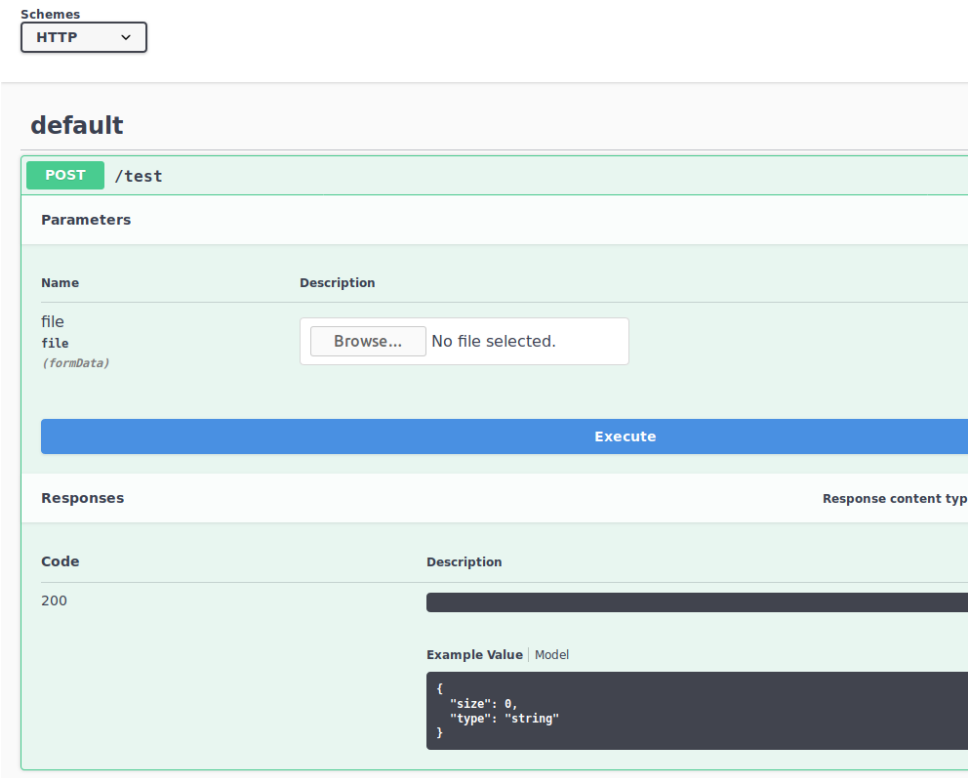
And the swagger:



File

Sanic-OpenAPI also support file field now. You can use this field to upload file through the swagger. For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.post("/test")
@doc.consumes(
    doc.File(name="file"), location="formData", content_type="multipart/form-data"
)
@doc.produces({"size": doc.Integer(), "type": doc.String()})
async def test(request):
    file = request.files.get("file")
    size = len(file.body)
    return json({"size": size, "type": file.type})
```



And it would be a upload button on swagger:

Dictionary

To be done.

JsonBody

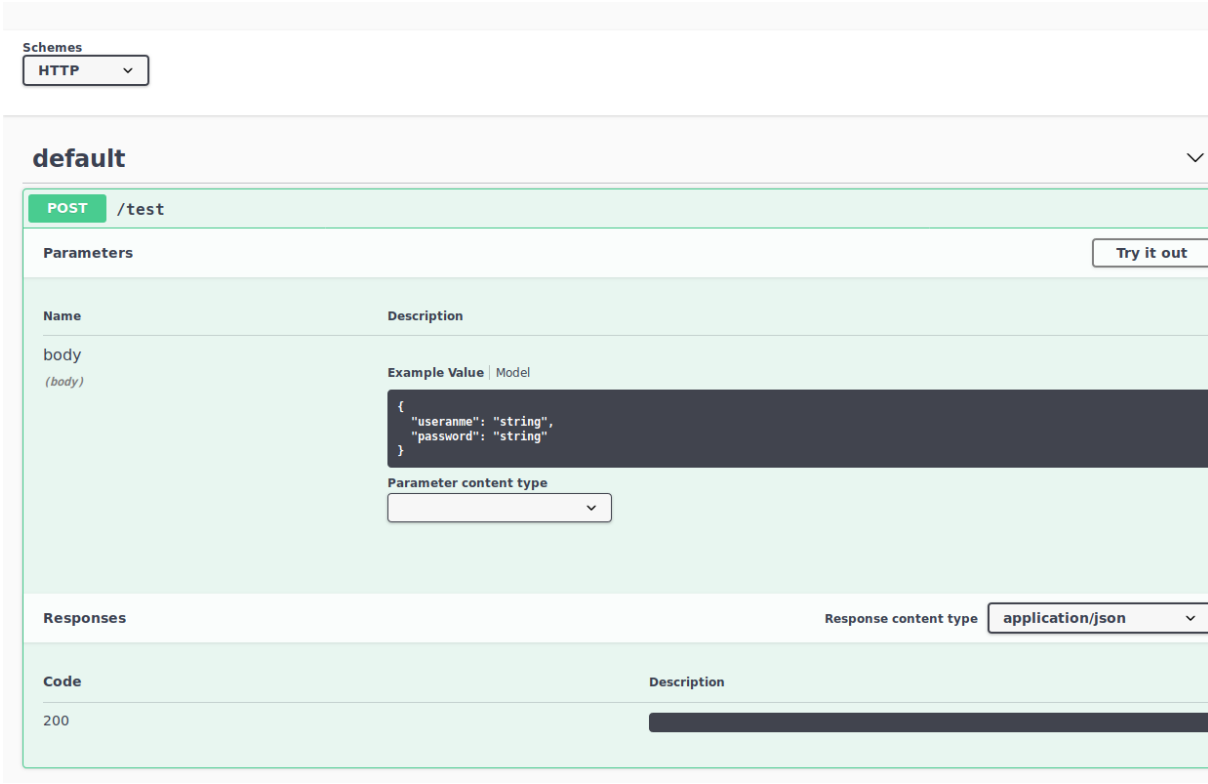
To document you request or response body, the `doc.JsonBody` is the best choice. You can put a `dict` into `doc.JsonBody` like this:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


@app.post("/test")
@doc.consumes(
    doc.JsonBody(
        {
            "useranme": doc.String("The name of your user account."),
            "password": doc.String("The password of your user account."),
        }
    ),
    location="body",
```

(continues on next page)

(continued from previous page)

```
)
async def test(request):
    return json({})
```



And it will convert to:

Note: The *doc.JsonBody* only support *dict* input. If you want to put a python *class* in body, please use *doc.Object*.

List

When design a RESTful with list resources API, the `doc.List` can help you document this API.

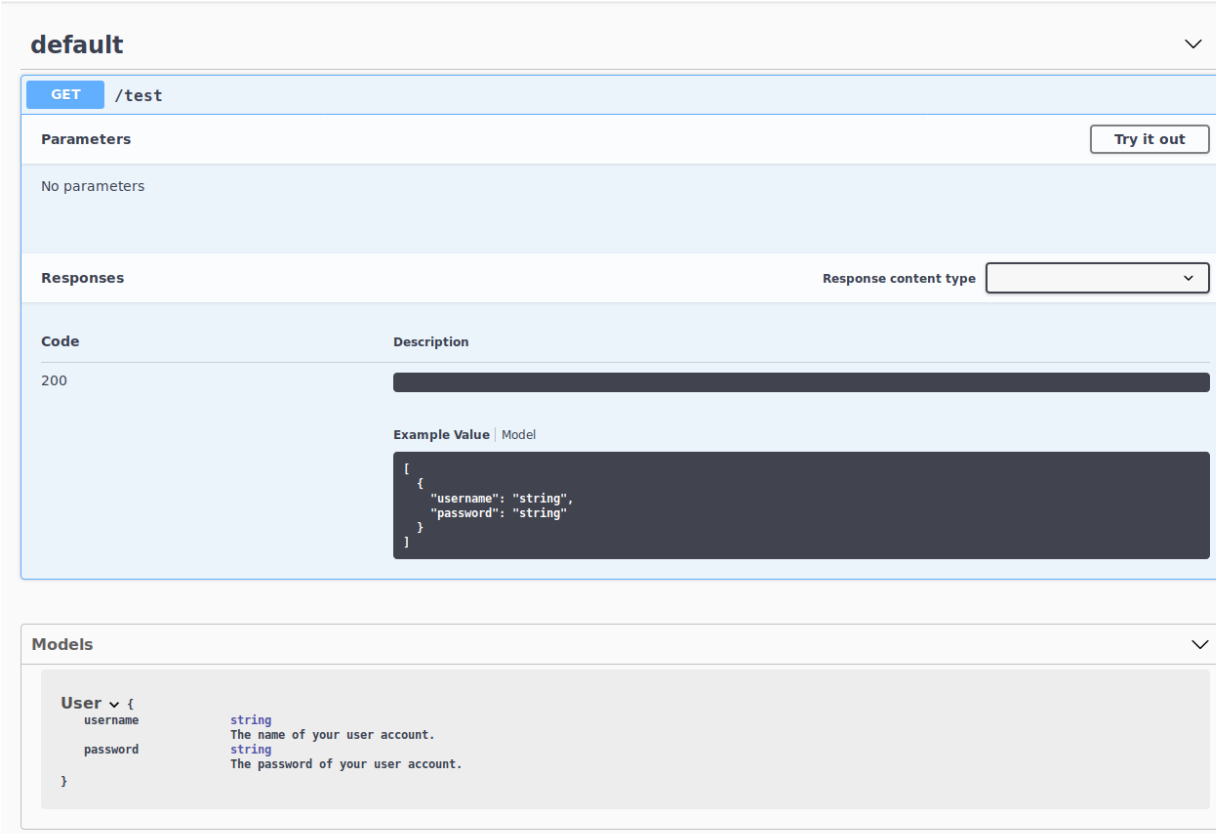
For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class User:
    username = doc.String("The name of your user account.")
    password = doc.String("The password of your user account.")
```

(continues on next page)

(continued from previous page)

```
@app.get("/test")
@doc.produces(doc.List(User))
async def test(request):
    return json([])
```



The swagger will be:

Note: When using a Python *class* to model your data, Sanic-OpenAPI will put it at model definitions.

Object

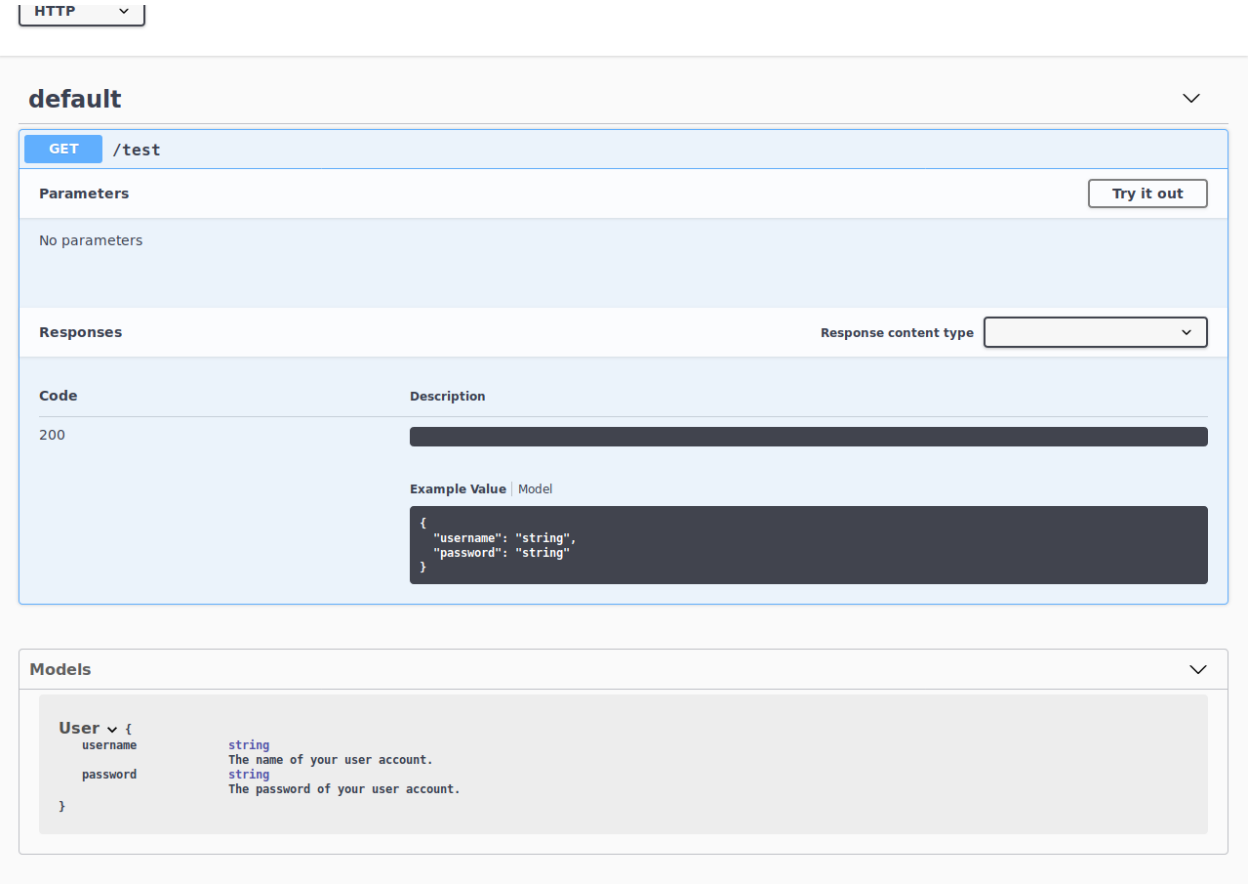
In Sanic-OpenAPI, you can document your data as a Python `class` and it wil be convert to `doc.Object` automaticlly. After the conversion, you can find your model definitions at the bottom of swagger.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class User:
```

(continues on next page)

(continued from previous page)

```
    username = doc.String("The name of your user account.")
    password = doc.String("The password of your user account.")


@app.get("/test")
@doc.produces(User)
async def test(request):
    return json({})
```



And the result:

Inheritance is also supported.

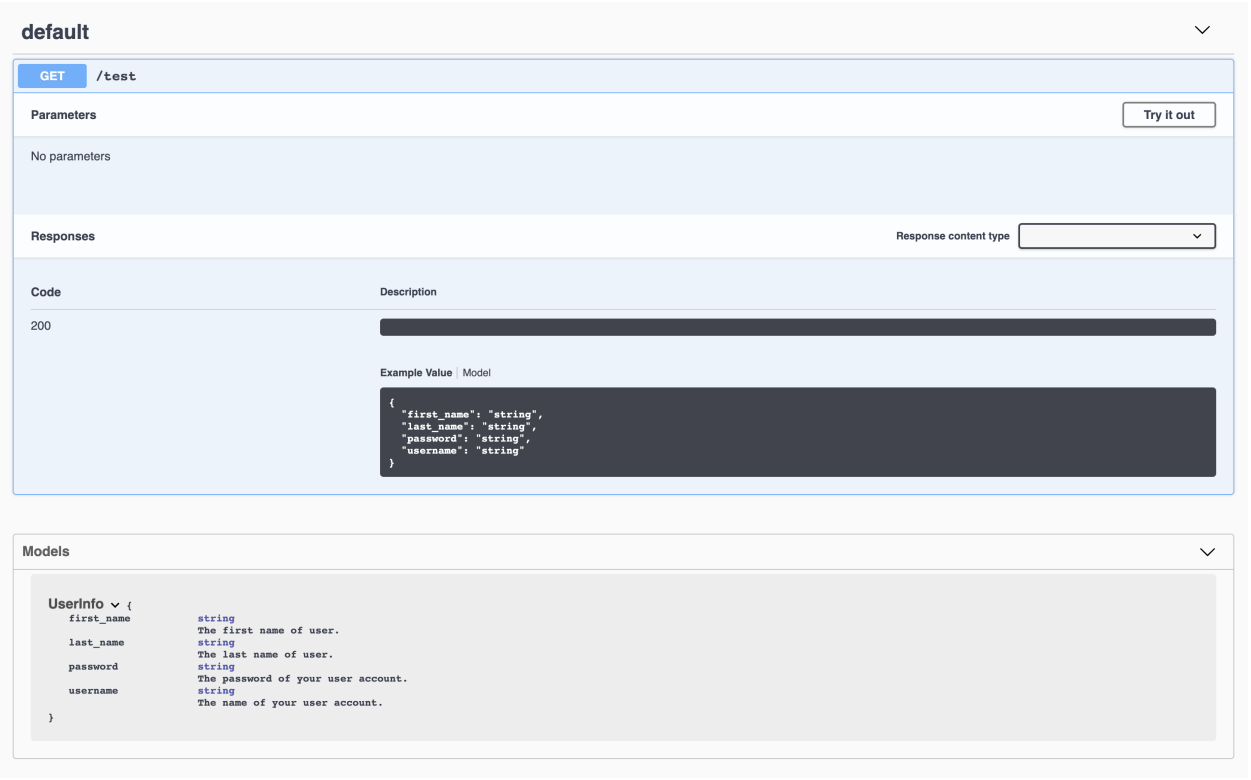
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class User:
    username = doc.String("The name of your user account.")
    password = doc.String("The password of your user account.")


class UserInfo(User):
```

(continues on next page)

(continued from previous page)

```
    first_name = doc.String("The first name of user.")
    last_name = doc.String("The last name of user.")


@app.get("/test")
@doc.produces(UserInfo)
async def test(request):
    return json({})

app.run(host="0.0.0.0", debug=True)
```

And the result:



PEP484's type hinting

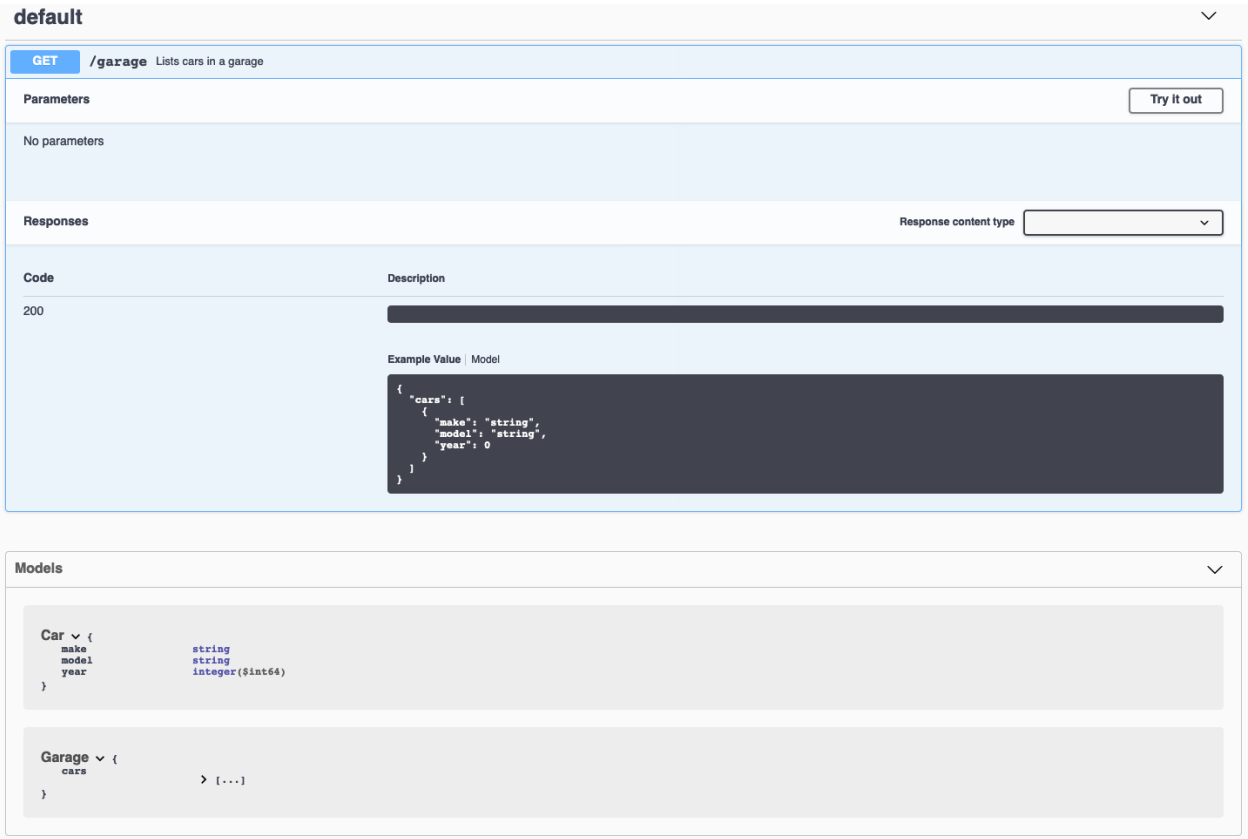
Provisional support, as discussed at #128

```
from typing import List

from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class Car:
    make: str
```

(continues on next page)

(continued from previous page)

```
    model: str
    year: int

class Garage:
    cars: List[Car]


@app.get("/garage")
@doc.summary("Lists cars in a garage")
@doc.produces(Garage)
async def get_garage(request):
    return json([{
                "make": "Nissan",
                "model": "370Z",
                "year": "2006",
    }])
```



And the result:

TypedDicts are also supported. In the previous example, `Car` could be defined as:

```
class Car(TypedDict):
    make: str
    model: str
    year: int
```

Descriptive Field

As the object example, you can use python class to document your request or response body. To make it more descriptive, you can add the descriptoin to every fields if you need. For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import doc, openapi2_blueprint

app = Sanic()
app.blueprint(openapi2_blueprint)


class Car:
    make = doc.String("Who made the car")
    model = doc.String("Type of car.  This will vary by make")
    year = doc.Integer("4-digit year of the car", required=False)

class Garage:
    spaces = doc.Integer("How many cars can fit in the garage")
    cars = doc.List(Car, description="All cars in the garage")

@app.get("/test")
@doc.produces(Garage)
async def test(request):
    return json({})
```

And you can get this model definitions on swagger:

Examples

API Reference

sanic_openapi.doc

class sanic_openapi.openapi2.doc.**Boolean**(*description=None*, *required=None*, *name=None*, *choices=None*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**Date**(*description=None*, *required=None*, *name=None*, *choices=None*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**DateTime**(*description=None*, *required=None*, *name=None*, *choices=None*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**Dictionary**(*fields=None*, ***kwargs*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**Field**(*description=None*, *required=None*, *name=None*, *choices=None*)
Bases: object

serialize()

class sanic_openapi.openapi2.doc.**File**(*description=None*, *required=None*, *name=None*, *choices=None*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**Float**(*description=None*, *required=None*, *name=None*, *choices=None*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**Integer**(*description=None*, *required=None*, *name=None*, *choices=None*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**JsonBody**(*fields=None*, ***kwargs*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**List**(*items=None*, **args*, ***kwargs*)
Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**Object**(*cls*, **args*, *object_name=None*, ***kwargs*)
Bases: *sanic_openapi.openapi2.doc.Field*

definition

serialize()

class sanic_openapi.openapi2.doc.**RouteField**(*field*, *location=None*, *required=False*, *description=None*)

Bases: object

description = None

field = None

location = None

required = None

class sanic_openapi.openapi2.doc.**RouteSpec**

Bases: object

blueprint = None

consumes = None

consumes_content_type = None

description = None

exclude = None

operation = None

produces = None

produces_content_type = None

response = None

summary = None

tags = None

class sanic_openapi.openapi2.doc.**String**(*description=None*, *required=None*, *name=None*, *choices=None*)

Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

class sanic_openapi.openapi2.doc.**Tuple**(*description=None*, *required=None*, *name=None*, *choices=None*)

Bases: *sanic_openapi.openapi2.doc.Field*

class sanic_openapi.openapi2.doc.**UUID**(*description=None*, *required=None*, *name=None*, *choices=None*)

Bases: *sanic_openapi.openapi2.doc.Field*

serialize()

sanic_openapi.openapi2.doc.**consumes**(**args*, *content_type='application/json'*, *location='query'*, *required=False*)

sanic_openapi.openapi2.doc.**description**(*text*)

sanic_openapi.openapi2.doc.**exclude**(*boolean*)

sanic_openapi.openapi2.doc.**operation**(*name*)

sanic_openapi.openapi2.doc.**produces**(**args*, *description=None*, *content_type=None*)

sanic_openapi.openapi2.doc.**response**(**args*, *description=None*)

sanic_openapi.openapi2.doc.**route**(*summary=None*, *description=None*, *consumes=None*, *produces=None*, *consumes_content_type=None*, *produces_content_type=None*, *exclude=None*, *response=None*)

sanic_openapi.openapi2.doc.**serialize_schema**(*schema*)

sanic_openapi.openapi2.doc.**summary**(*text*)

sanic_openapi.openapi2.doc.**tag**(*name*)

sanic_openapi.api

class sanic_openapi.openapi2.api.**API**
Bases: object

Decorator factory class for documenting routes using *sanic_openapi* and optionally registering them in a *sanic* application or blueprint.

Supported class attribute names match the corresponding *sanic_openapi.doc* decorator's name and attribute values work exactly as if they were passed to the given decorator unless explicitly documented otherwise. The supported class attributes (all of which are optional) are as follows:

- ***summary*: Its value should be the short summary of the route. If neither *summary*** nor *description* is specified, then the first paragraph of the API class' documentation will be used instead. You may also set it to *None* to disable automatic *summary* and *description* generation.
- ***description*: A longer description of the route. If neither *summary* nor** *description* is specified, then the API class' documentation will be used except its first paragraph that serves as the default summary. You may also set it to *None* to disable automatic *summary* and *description* generation.
- *exclude*: Whether to exclude the route (and related models) from the API documentation.
- ***consumes*: The model of the data the API route consumes. If *consumes* is a class** that has a docstring, then the docstring will be used as the description of th data.
- *consumes_content_type*: The content type of the data the API route consumes.
- *consumes_location*: The location where the data is expected (*query* or *body*).
- *consumes_required*: Whether the consumed data is required.
- *produces*: The model of the data the API route produces.
- *produces_content_type*: The content type of the data the API route produces.
- ***produces_description*: The description of the data the API route produces. If** not specified but *produces* is a class that has a docstring, then the docstring will be used as the default description.
- ***response*: A *Response* instance or a sequence of *Response* instances that describe** the route's response for different HTTP status codes. The value of the *produces* attribute corresponds to HTTP 200, you don't have to specify that here.
- *tag*: The tags/groups the API route belongs to.

Example:

‘‘‘Python class JSONConsumerAPI(API):

consumes_content_type = “application/json” consumes_location = “body” consumes_required = True

class JSONProducerAPI(API): produces_content_type = “application/json”

class MyAPI(JSONConsumerAPI, JSONProducerAPI): “”” Route *summary* in first paragraph.

First paragraph of route *description*.

Second paragraph of route *description*. “””

class consumes: foo = str bar = str

class produces: result = bool

Document and register the route at once. @MyAPI.post(app, “/my_route”) def my_route(request: Request):

return {“result”: True}

Or simply document a route. @app.post(“/my_route”) @MyAPI def my_route(request: Request):

return {“result”: True}

“‘

Additionally, you may specify a *decorators* class attribute, whose value must be a sequence of decorators to apply on the decorated routes. These decorators will be applied *before* the *sanic_openapi* decorators - and the *sanic* routing decorators if the routing decorators provided by this class are used - in *reverse* order. It means that the following cases are equivalent:

“‘Python class Data(API):

class consumes: stg = str

class DecoratedData(Data): decorators = (first, second)

@DecoratedData.get(app, “/data”) def data_all_in_one(request: Request):

return “data”

@app.get(“/data”) @DecoratedData def data_doc_and_decorators_in_one(request: Request):

return “data”

@Data.get(app, “/data”) @first @second def data_routing_and_doc_in_one(request: Request):

return “data”

@app.get(“/data”) @Data @first @second def data(request: Request):

return “data”

“‘

It is possible to override all the described class attributes on a per decorator basis simply by passing the desired custom value to the decorator as a keyword argument:

“‘Python class JSONConsumerAPI(API):

consumes_content_type = “application/json” consumes_location = “body” consumes_required = True

class consumes: foo = str bar = str

The consumed data is required. @JSONConsumerAPI.post(app, “/data”) def data(request: Request):

return “data”

The consumed data is optional. @app.post(“/data_optional”) @JSONConsumer-API(consumes_required=False) def data_consumed_not_required(request: Request):

return “data”

“‘

classmethod delete(*app*, *uri*, ***kwargs*)
Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

The decorated method will be registered for *DELETE* requests.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *delete()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

classmethod get(*app*, *uri*, ***kwargs*)
Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

The decorated method will be registered for *GET* requests.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *get()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

classmethod head(*app*, *uri*, ***kwargs*)
Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

The decorated method will be registered for *HEAD* requests.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *head()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

classmethod options(*app*, *uri*, ***kwargs*)
Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

The decorated method will be registered for *OPTIONS* requests.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *options()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

classmethod patch(*app*, *uri*, ***kwargs*)
Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

The decorated method will be registered for *PATCH* requests.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *patch()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

classmethod post(*app*, *uri*, ***kwargs*)
Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

The decorated method will be registered for *POST* requests.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *post()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

classmethod put(*app*, *uri*, ***kwargs*)

Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

The decorated method will be registered for *PUT* requests.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *put()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

classmethod route(*app*, *uri*, *, *methods*, ***kwargs*)

Decorator that registers the decorated route in the given *sanic* application or blueprint with the given URI, and also documents its API using *sanic_openapi*.

Keyword arguments that are not listed in arguments section will be passed on to the *sanic* application's or blueprint's *route()* method as they are.

Arguments: app: The *sanic* application or blueprint where the route should be registered. uri: The URI the route should be accessible at.

class sanic_openapi.openapi2.api.**Response**

Bases: *sanic_openapi.openapi2.api.Response*

HTTP status code - returned object model pair with optional description.

If *model* is a class that has a docstring, the its docstring will be used as description if *description* is not set.

2.2 Sanic OpenAPI 3

Sanic OpenAPI 3 support is experimental. This feature may be subject to change in future releases.

2.2.1 Getting started

Here is an example to use Sanic-OpenAPI 2:

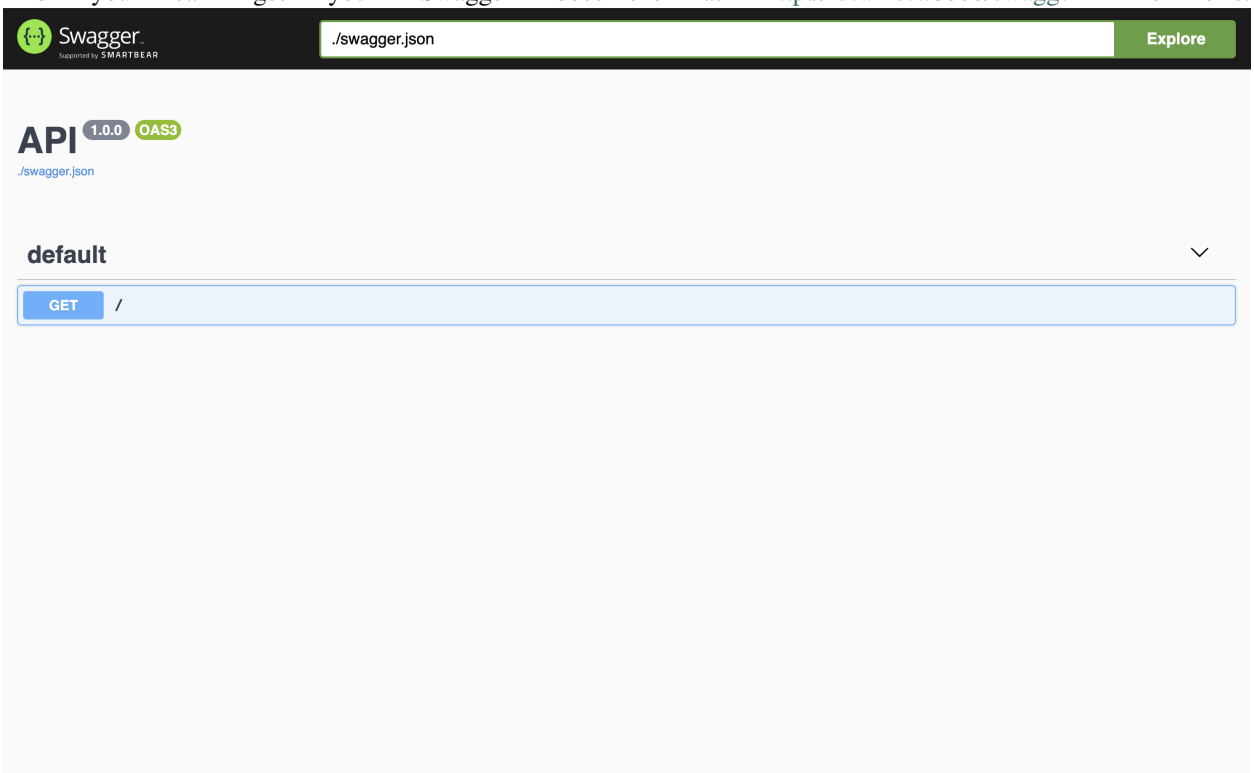
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi3_blueprint

app = Sanic("Hello world")
app.blueprint(openapi3_blueprint)


@app.route("/")
async def test(request):
    return json({"hello": "world"})
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

And you can get your Swagger document at http://localhost:8000/swagger like this:



2.2.2 Contents

Document Routes

Sanic-OpenAPI support different ways to document APIs includes:

- routes of `Sanic` instance
- routes of `Blueprint` instance
- routes of `HTTPMethodView` under `Sanic` instance
- routes of `HTTPMethodView` under `Bluebprint` instance
- routes of `CompositionView` under `Sanic` instance

But with some exceptions:

- Sanic-OpenAPI does not support routes of `CompositionView` under `Bluebprint` instance now.
- Sanic-OpenAPI does not document routes with `OPTIONS` method.
- Sanic-OpenAPI does not document routes which registered by `static()`.

This section will explain how to document routes with above cases.

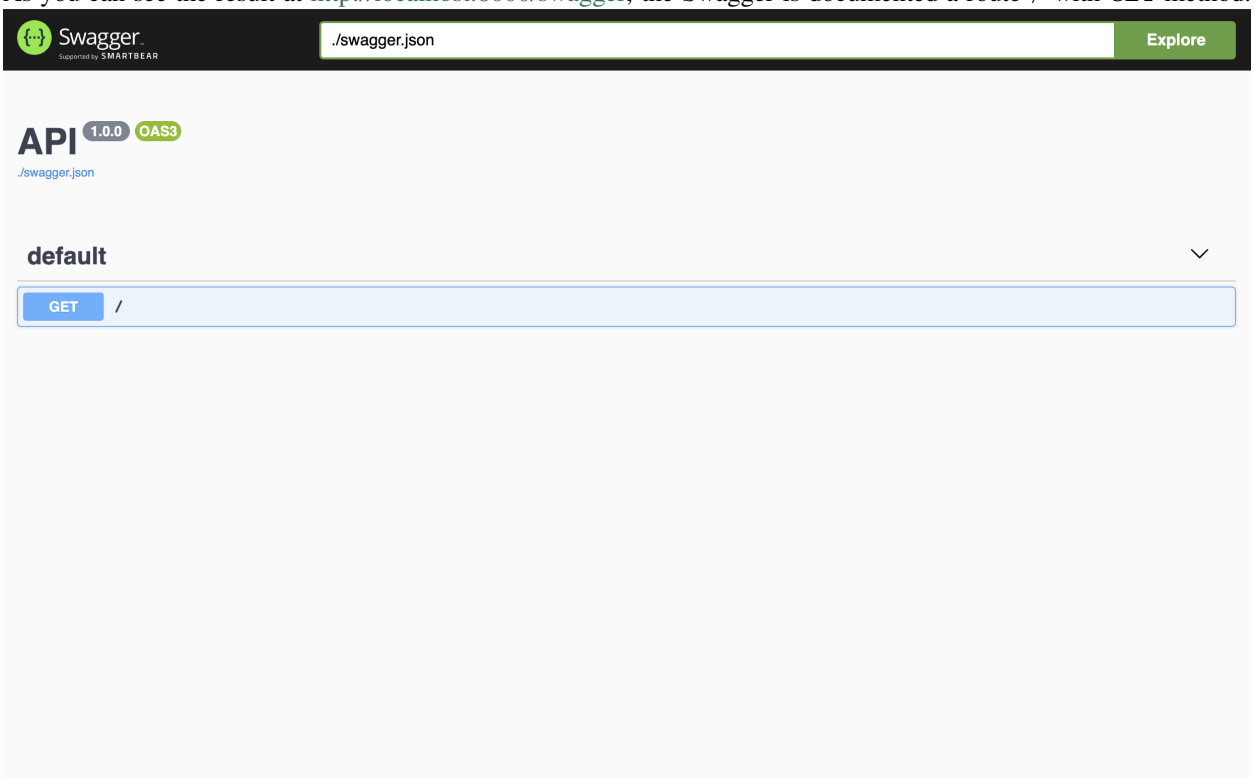
Basic Routes

To use Sanic-OpenAPI with basic routes, you only have to register `openapi3_blueprint` and it will be all set.

For example:

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi3_blueprint

app = Sanic("Hello world")
app.blueprint(openapi3_blueprint)


@app.route("/")
async def test(request):
    return json({"hello": "world"})


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

As you can see the result at http://localhost:8000/swagger, the Swagger is documented a route `/` with `GET` method.

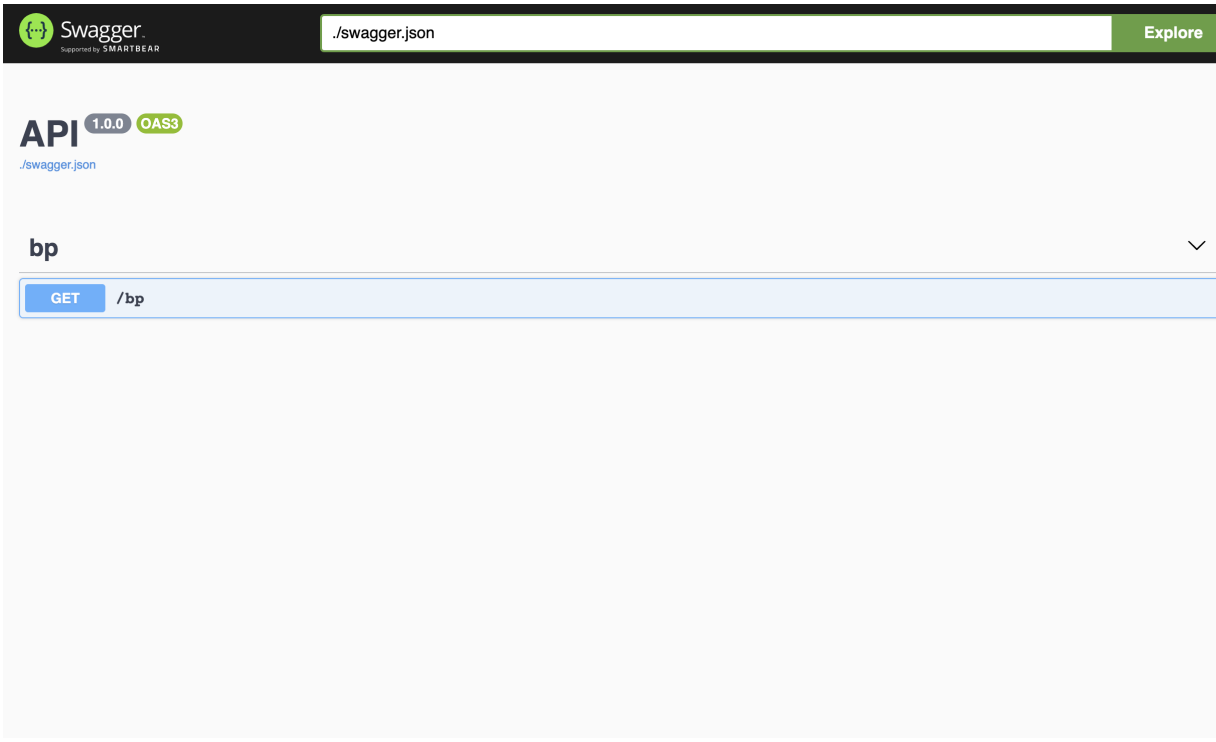


If you want to add some additional information to this route, you can use other decorators like `summary()`, `description()`, and etc.

Blueprint Routes

You can aldo document routes under any `Blueprint` like this:

```python
from sanic import Blueprint, Sanic
from sanic.response import json

from sanic_openapi import openapi3_blueprint

app = Sanic("Hello world")
app.blueprint(openapi3_blueprint)

bp = Blueprint("bp", url_prefix="/bp")


@bp.route("/")
async def test(request):
    return json({"hello": "world"})


app.blueprint(bp)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```



The result looks like:

When you document routes under `Blueprint` instance, they will be document with `tags` which using the `Blueprint`'s name.

Class-Based Views Routes

In Sanic, it provides a class-based views named `HTTPMethodView`. You can document routes under `HTTPMethodView` like:

```
from sanic import Sanic
from sanic.response import text
from sanic.views import HTTPMethodView

from sanic_openapi import openapi3_blueprint

app = Sanic("Hello world")
app.blueprint(openapi3_blueprint)


class SimpleView(HTTPMethodView):
    def get(self, request):
        return text("I am get method")

    def post(self, request):
        return text("I am post method")

    def put(self, request):
        return text("I am put method")

    def patch(self, request):
        return text("I am patch method")

    def delete(self, request):
        return text("I am delete method")

    def options(self, request): # This will not be documented.
        return text("I am options method")


app.add_route(SimpleView.as_view(), "/")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

And the result:

Please note that Sanic-OpenAPI will not document any routes with `OPTIONS` method.

The `HTTPMethodView` can also be registered under `Blueprint`:

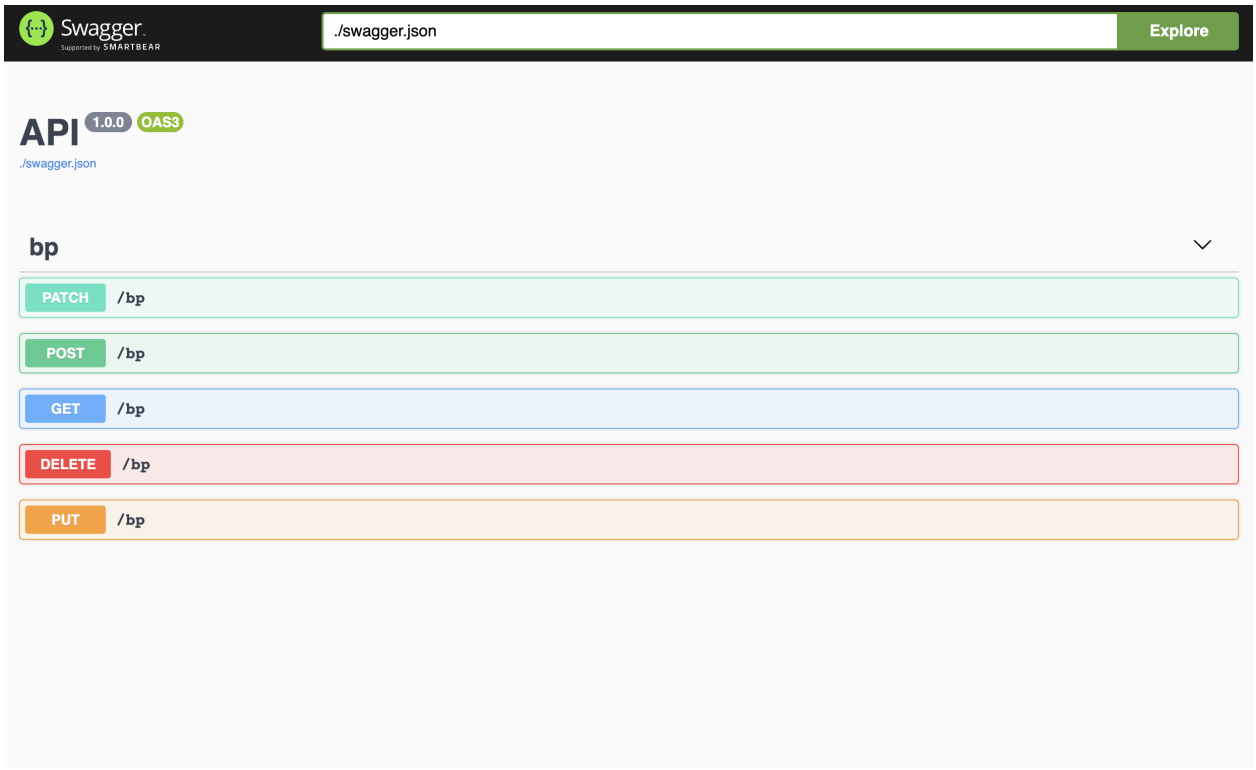
```
from sanic import Blueprint, Sanic
from sanic.response import text
from sanic.views import HTTPMethodView

from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)

bp = Blueprint("bp", url_prefix="/bp")


class SimpleView(HTTPMethodView):
    def get(self, request):
        return text("I am get method")

    def post(self, request):
        return text("I am post method")

    def put(self, request):
        return text("I am put method")

    def patch(self, request):
        return text("I am patch method")

    def delete(self, request):
        return text("I am delete method")

    def options(self, request):  # This will not be documented.
```

(continues on next page)

(continued from previous page)

```
        return text("I am options method")


bp.add_route(SimpleView.as_view(), "/")
app.blueprint(bp)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

The result:



CompositionView Routes

There is another class-based view named `CompositionView`. Sanic-OpenAPI also support to document routes under class-based view.

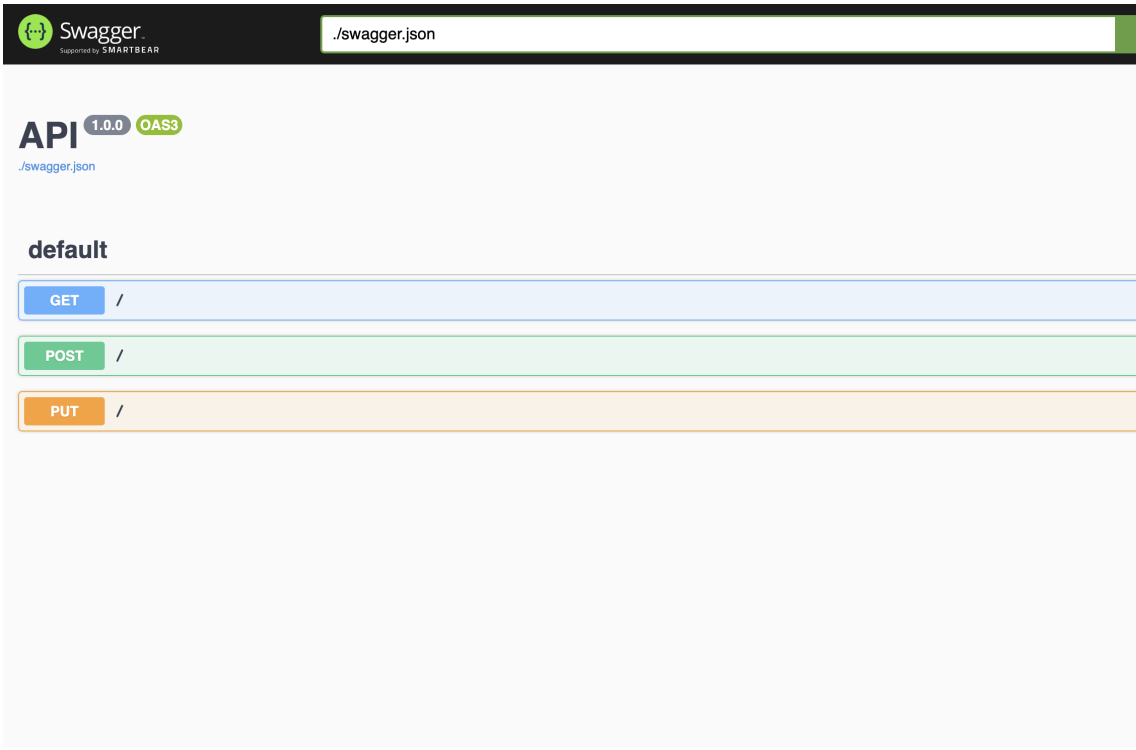
```
from sanic import Sanic
from sanic.response import text
from sanic.views import CompositionView

from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


def get_handler(request):
    return text("I am a get method")
```

(continues on next page)

(continued from previous page)

```
view = CompositionView()
view.add(["GET"], get_handler)
view.add(["POST", "PUT"], lambda request: text("I am a post/put method"))

# Use the new view to handle requests to the base URL
app.add_route(view, "/")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

The Swagger will looks like:



Note: Sanic-OpenAPI does not support routes of *CompositionView* under *Bluebprint* instance now.

Configurations

Sanic-OpenAPI provides following configurable items:

- API Server
- API information
- Authentication(Security Definitions)
- URI filter
- Swagger UI configurations

API Server

By default, Swagger will use exactly the same host which served itself as the API server. But you can still override this by setting following configurations. For more information, please check document at here.

API_HOST

- Key: `API_HOST`
- Type: `str` of IP, or hostname
- Default: `None`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_HOST = "petstore.swagger.io"
```

- Result:

API_BASEPATH

- Key: `API_BASEPATH`
- Type: `str`
- Default: `None`
- Usage:

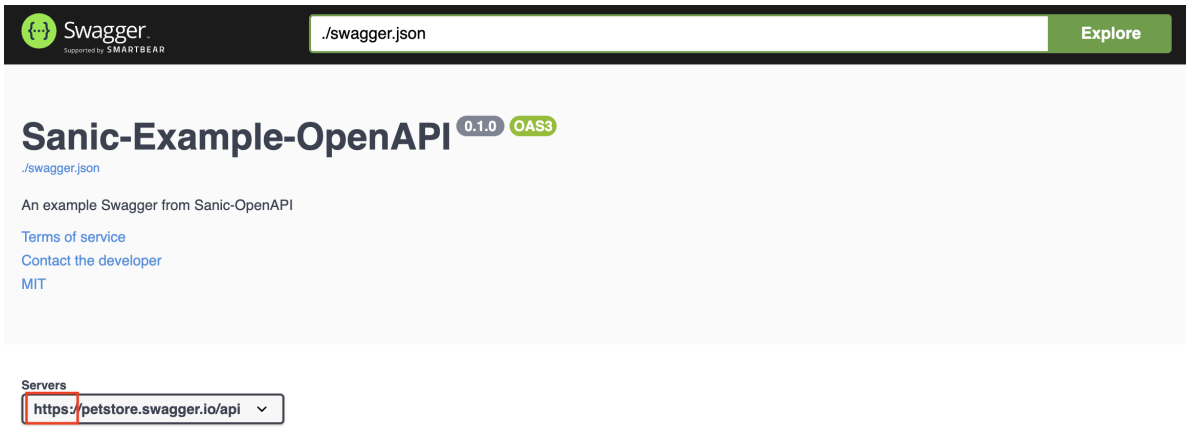
```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_BASEPATH = "/api"
```

- Result:

API_SCHEMES

- Key: `API_SCHEMES`
- Type: `list` of schemes
- Default: `["http"]`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_SCHEMES = ["https"]
```



- Result:

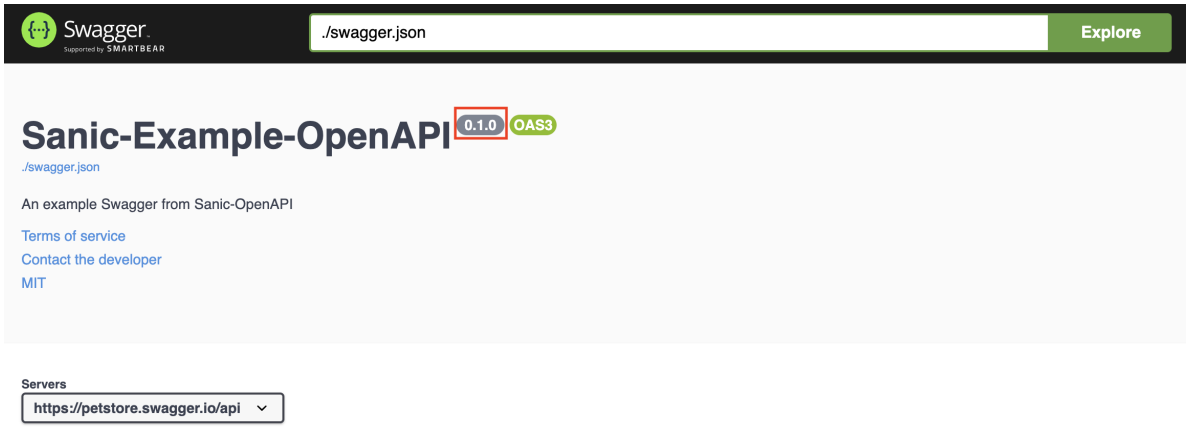
API information

You can provide some additional information of your APIs by using Sanic-OpenAPI configurations. For more detail of those additional information, please check the document from Swagger.

API_VERSION

- Key: `API_VERSION`
- Type: `str`
- Default: `1.0.0`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_VERSION = "0.1.0"
```



- Result:

API_TITLE

- Key: `API_TITLE`
- Type: `str`
- Default: `API`
- Usage:

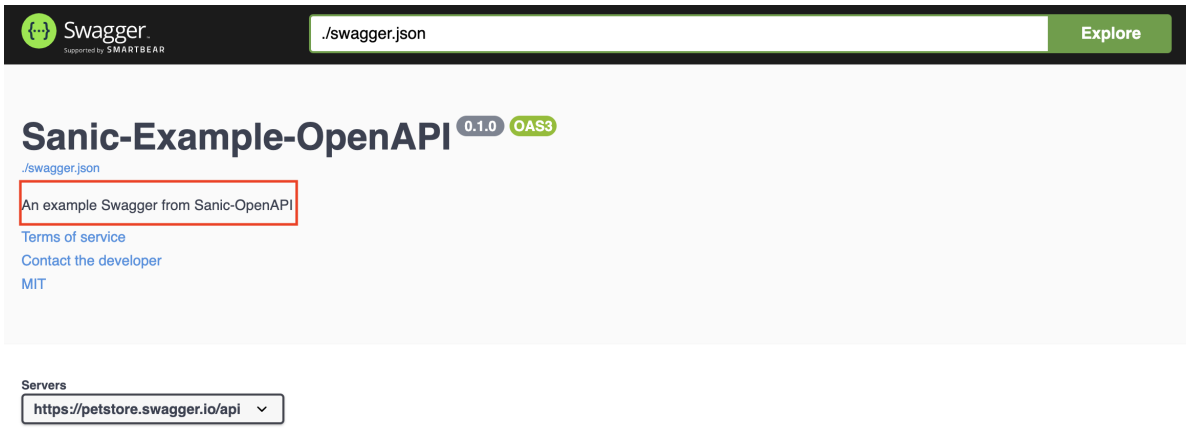
```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_TITLE = "Sanic-Example-OpenAPI"
```

- Result:

API_DESCRIPTION

- Key: `API_DESCRIPTION`
- Type: `str`
- Deafult: `""`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_DESCRIPTION = "An example Swagger from Sanic-OpenAPI"
```



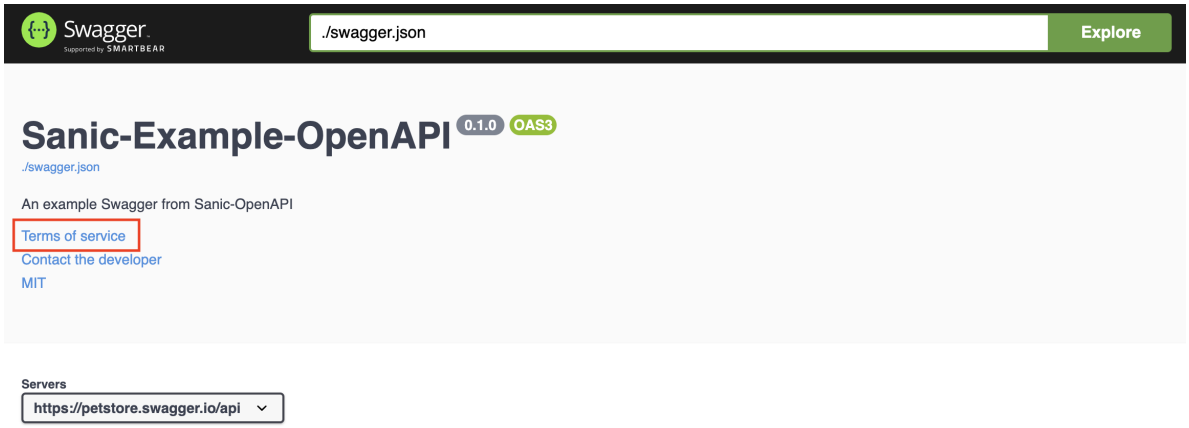
- Result:

API_TERMS_OF_SERVICE

- Key: `API_TERMS_OF_SERVICE`
- Type: `str` of a URL
- Deafult: `""`

- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_TERMS_OF_SERVICE = "https://github.com/sanic-org/sanic-openapi/
↪blob/master/README.md"
```

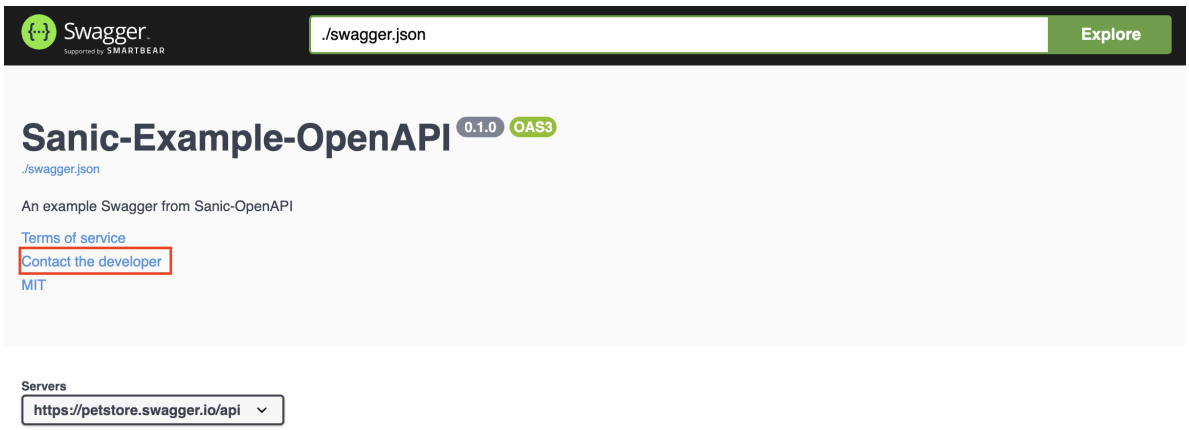


- Result:

API_CONTACT_EMAIL

- Key: `API_CONTACT_EMAIL`
- Type: `str` of email address
- Deafult: `None"`
- Usage:

```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_CONTACT_EMAIL = "foo@bar.com"
```



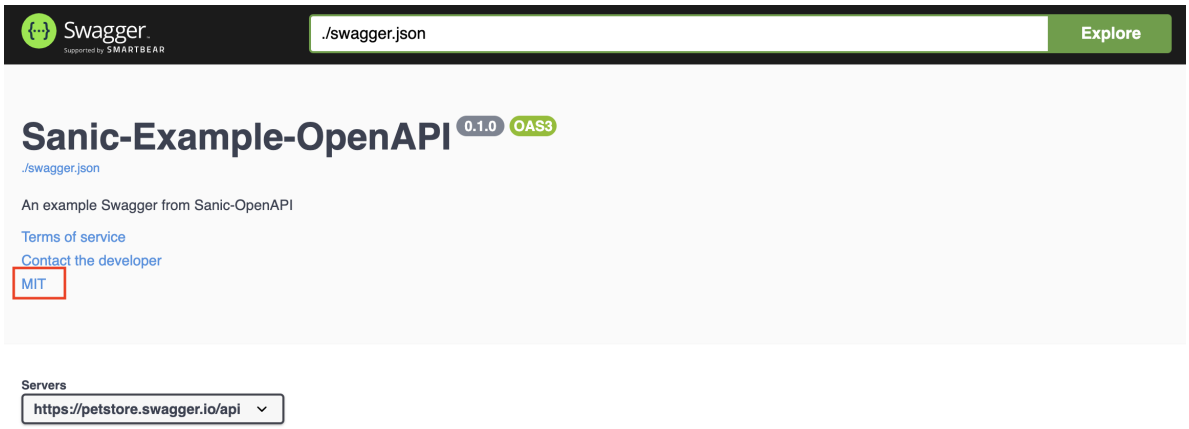
- Result:

API_LICENSE_NAME

- Key: `API_LICENSE_NAME`
- Type: `str`
- Default: `None`
- Usage:

  python from sanic import Sanic from sanic_openapi import openapi3_blueprint

  app = Sanic() app.blueprint(openapi3_blueprint) app.config.API_LICENSE_NAME = “MIT”

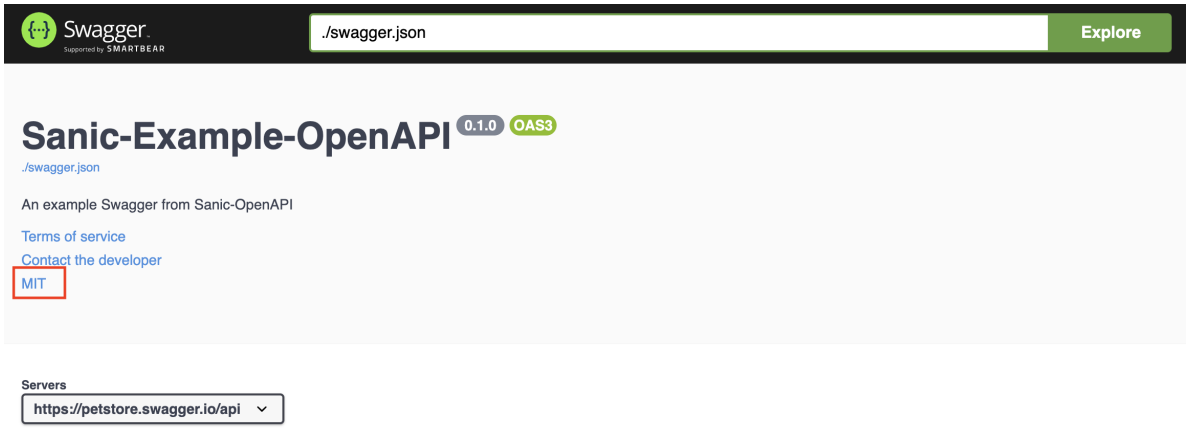


- Result:

API_LICENSE_URL

- Key: `API_LICENSE_URL`
- Type: `str` of URL
- Default: `None`
- Usgae:

```
from sanic import Sanic
from sanic_openapi import openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)
app.config.API_LICENSE_URL = "https://github.com/sanic-org/sanic-openapi/blob/
↪master/LICENSE"
```



- Result:

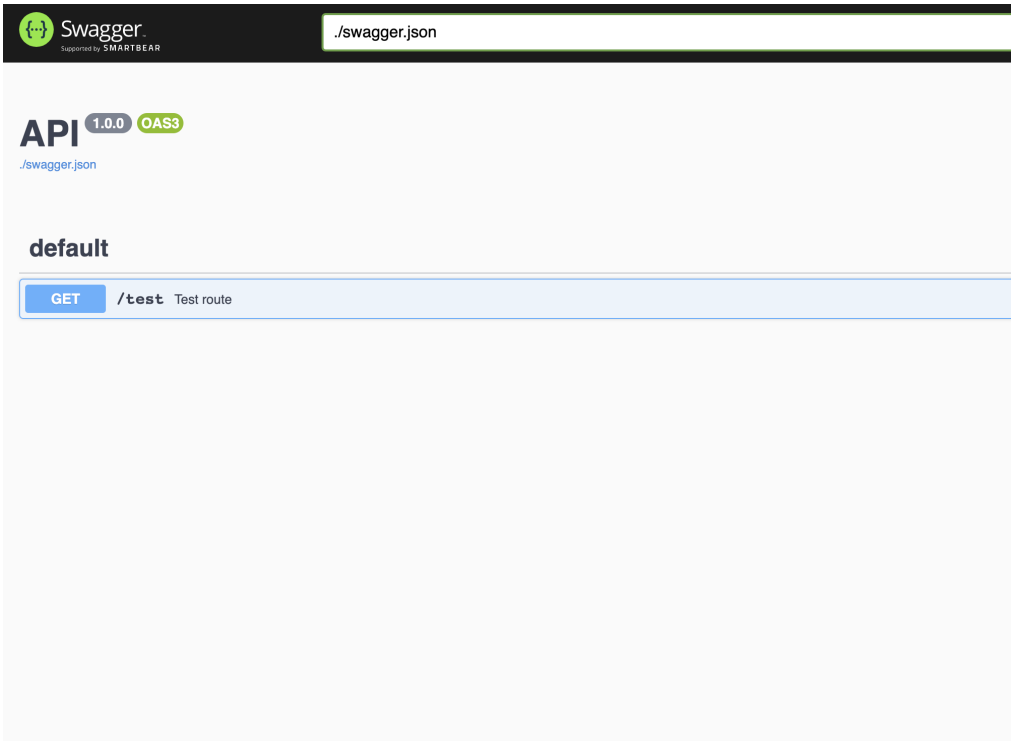
Decorators

Sanic-OpenAPI provides different **decorator** can help you document your API routes.

Summary

You can add a short summary to your route by using `summary()` decorator. It is helpful to point out the purpose of your API route.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic("Hello world")
app.blueprint(openapi3_blueprint)


@app.get("/test")
@openapi.summary("Test route")
async def test(request):
    return json({"Hello": "World"})
```



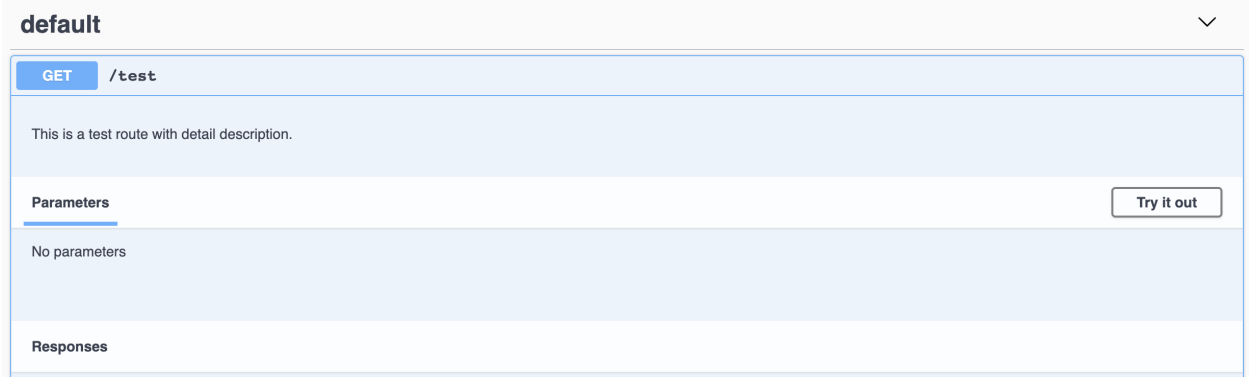
The summary will show behind the path:

Description

Not only short summary, but also long description of your API route can be addressed by using `description()` decorator.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


@app.get("/test")
@openapi.description('This is a test route with detail description.')
async def test(request):
    return json({"Hello": "World"})
```

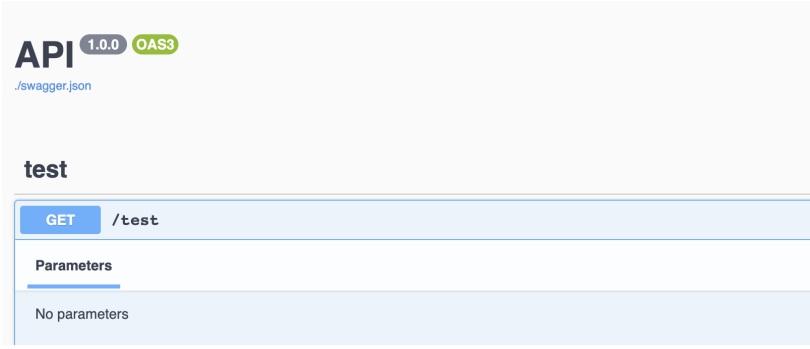
To see the description, you have to expand the content of route and it would looks like:



Tag

If you want to group your API routes, you can use `tag()` decorator to accomplish your need.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


@app.get("/test")
@openapi.tag("test")
async def test(request):
    return json({"Hello": "World"})
```



And you can see the tag is change from `default` to `test`:

By default, all routes register under Sanic will be tag with `default`. And all routes under Blueprint will be tag with the blueprint name.

Operation

Sanic-OpenAPI will use route(function) name as the default `operationId`. You can override the `operationId` by using `operation()` decorator. The `operation()` decorator would be useful when your routes have duplicate name in some cases.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


@app.get("/test")
@openapi.operation('test1')
async def test(request):
    return json({"Hello": "World"})
```

Consumes

The `consumes()` decorator is the most common used decorator in Sanic-OpenAPI. It is used to document the parameter usages in swagger. You can use built-in classes like `str`, `int`, `dict` or use different fields which provides by Sanic-OpenAPI to document your parameters.

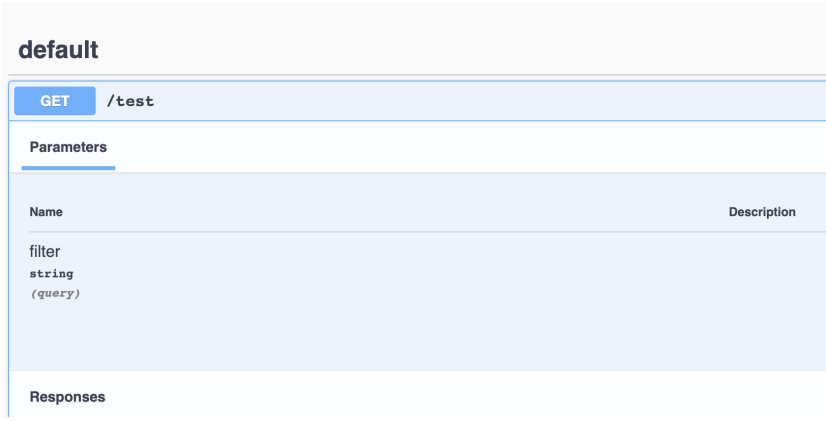
There are three kinds of parameter usages:

Query

To document the parameter in query string, you can use `location="query"` in `parameter()` decorator. This is also the default to `parameter()` decorator.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


@app.get("/test")
@openapi.parameter("filter", str, location="query")
async def test(request):
    return json({"Hello": "World"})
```

You can expand the contents of route and it will looks like:



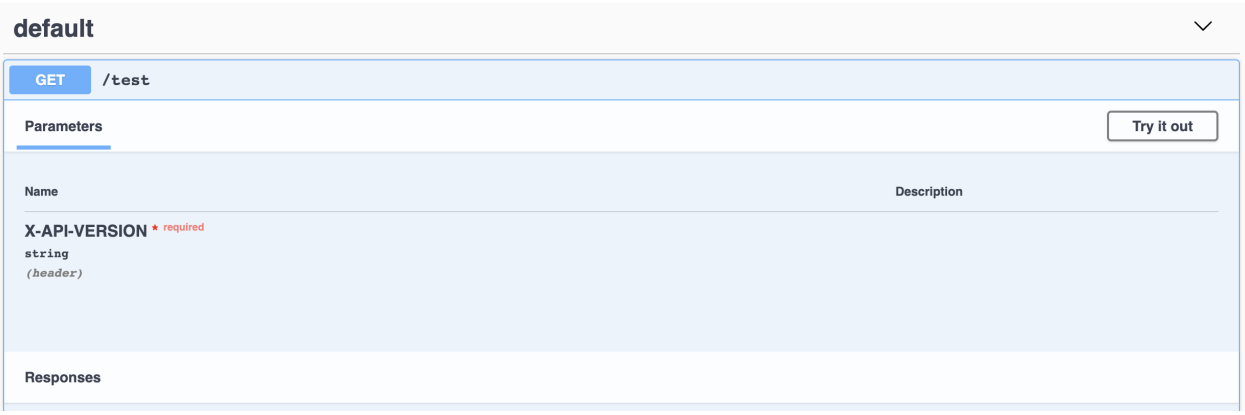
When using `parameter()` with `location="query"`, it only support simple types like `str`, `int` but no complex types like `dict`.

Header

For doucument parameters in header, you can set `location="header"` with simple types just like `location="query"`.

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


@app.get("/test")
@openapi.parameter("X-API-VERSION", str, location="header", required=True)
async def test(request):
    return json({"Hello": "World"})
```

It will looks like:



Request Body

In most cases, your APIs might contains lots of parameter in your request body. In Sanic-OpenAPI, you can define them in Python class or use fields which provides by Sanic-OpenAPI to simplify your works.

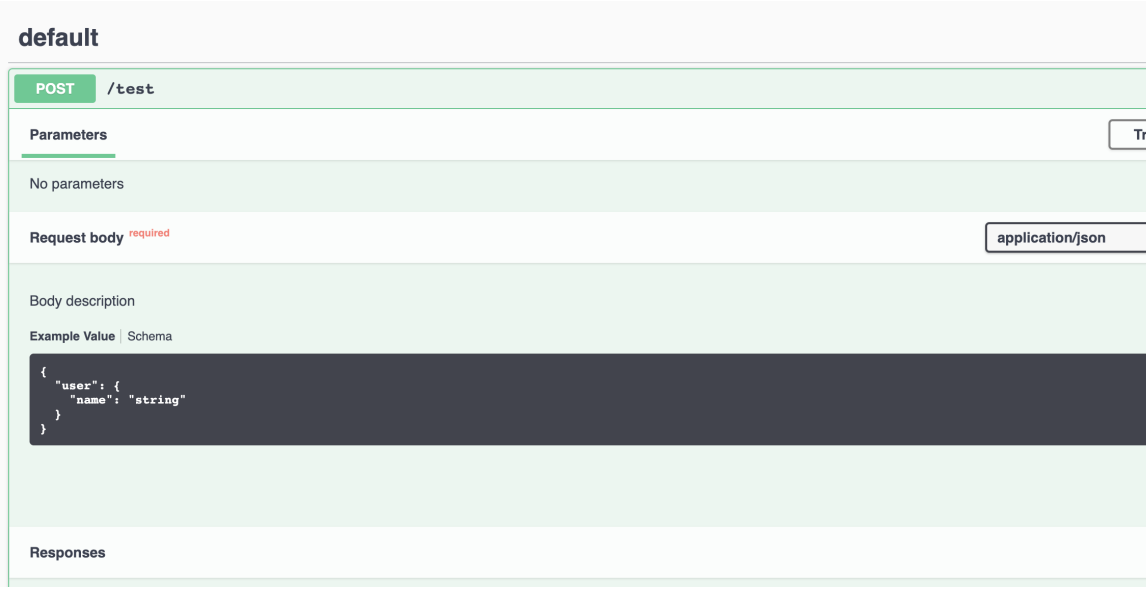
```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


class User:
    name = str
```

(continues on next page)

(continued from previous page)

```
class Test:
    user = User


@app.post("/test")
@openapi.body(
    { "application/json" : Test },
    description="Body description",
    required=True,
)
async def test(request):
    return json({"Hello": "World"})
```

This will be document like:

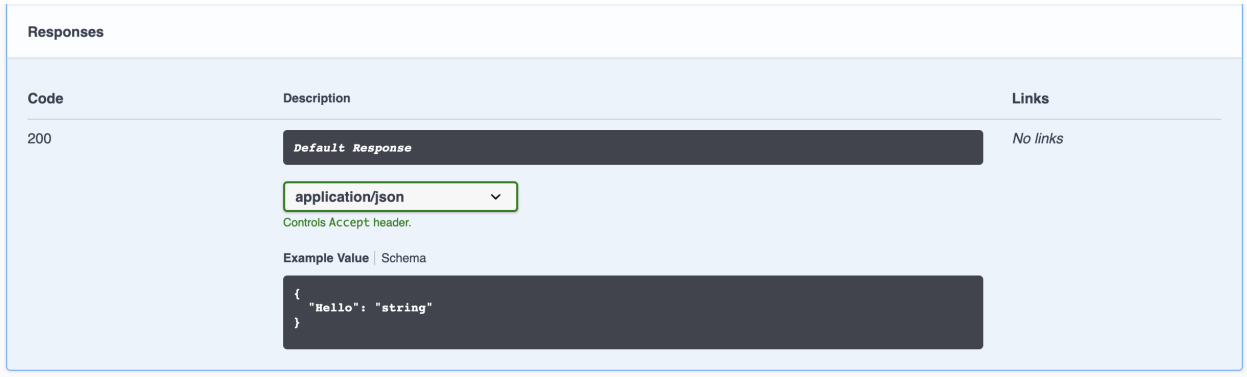


Produces

The `response()` decorator is used to document the default response(with status 200).

```
from sanic import Sanic
from sanic.response import json

from sanic_openapi import openapi, openapi3_blueprint

app = Sanic()
app.blueprint(openapi3_blueprint)


class Test:
    Hello = openapi.String(description='World')

@app.get("/test")
@openapi.response(200, {"application/json" : Test})
async def test(request):
    return json({"Hello": "World"})
```

As you can see in this example, you can also use Python class in `produces()` decorator.


Responses
Code
Description
Links
200
Default Response
No links
application/json
Controls Accept header.
Example Value
Schema
{
"Hello": "string"
}

Examples

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

R

S

T

U