Connexion Documentation

Release 0.5

Zalando SE

Sep 17, 2018

Contents

Connexion is a framework on top of Flask that automagically handles HTTP requests based on OpenAPI 2.0 Specification (formerly known as Swagger Spec) of your API described in YAML format. Connexion allows you to write a Swagger specification and then maps the endpoints to your Python functions. This is what makes it unique from other tools that generate the specification based on your Python code. You are free to describe your REST API with as much detail as you want and then Connexion guarantees that it will work as you specified. We built Connexion this way in order to:

- Simplify the development process
- Reduce misinterpretation about what an API is going to look like

Contents:

CHAPTER 1

Quickstart

1.1 Prerequisites

Python 2.7 or Python 3.4+

1.2 Installing It

In your command line, type this:

```
$ pip install connexion
```

1.3 Running It

Put your API YAML inside a folder in the root path of your application (e.g `swagger\`) and then do

```
import connexion

app = connexion.FlaskApp(__name__, specification_dir='swagger/')
app.add_api('my_api.yaml')
app.run(port=8080)
```

1.4 Dynamic Rendering of Your Specification

Connexion uses Jinja2 to allow specification parameterization through *arguments* parameter. You can either define specification arguments globally for the application in the *connexion.App* constructor, or for each specific API in the *connexion.App#add_api* method:

```
app = connexion.FlaskApp(__name__, specification_dir='swagger/',
                         arguments={'global': 'global_value'})
app.add_api('my_api.yaml', arguments={'api_local': 'local_value'})
app.run(port=8080)
```

When a value is provided both globally and on the API, the API value will take precedence.

1.5 The Swagger UI Console

The Swagger UI for an API is available, by default, in `{base_path}/ui/` where `base_path` is the base path of the API.

You can disable the Swagger UI at the application level:

```
app = connexion.FlaskApp(__name__, specification_dir='swagger/',
                         swagger_ui=False)
app.add_api('my_api.yaml')
```

You can also disable it at the API level:

```
app = connexion.FlaskApp(__name__, specification_dir='swagger/')
app.add_api('my_api.yaml', swagger_ui=False)
```

1.6 Server Backend

By default connexion uses the default flask server but you can also use Tornado or gevent as the HTTP server, to do so set server to `tornado` or `gevent`:

```
import connexion

app = connexion.FlaskApp(__name__, port = 8080, specification_dir='swagger/', server=
↪'tornado')
```

Connexion has the `aiohttp` framework as server backend too:

```
import connexion

app = connexion.AioHttpApp(__name__, port = 8080, specification_dir='swagger/')
```

CHAPTER 2

Command-Line Interface

For convenience Connexion provides a command-line interface (CLI). This interface aims to be a starting point in developing or testing OpenAPI specifications with Connexion.

The available commands are:

- `connexion run`

All commands can run with -h or –help to list more information.

2.1 Running an OpenAPI specification

The subcommand `run` of Connexion's CLI makes it easy to run OpenAPI specifications directly even before any operation handler function gets implemented. This allows you to verify and inspect how your API will work with Connexion.

To run your specification, execute in your shell:

```
$ connexion run your_api.yaml --stub --debug
```

This command will tell Connexion to run the `your_api.yaml` specification file attaching a stub operation (`--stub`) to the unavailable operations/functions of your API and in debug mode (`--debug`).

The basic usage of this command is:

```
$ connexion run [OPTIONS] SPEC_FILE [BASE_MODULE_PATH]
```

Where:

- SPEC_FILE: Your OpenAPI specification file in YAML format.
- BASE_MODULE_PATH (optional): filesystem path where the API endpoints handlers are going to be imported from. In short, where your Python code is saved.

There are more options available for the `run` command, for a full list run:

```
$ connexion run --help
```

2.2 Running a mock server

You can run a simple server which returns example responses on every request. The example responses must be defined in the `examples` response property of the OpenAPI specification. Your API specification file is not required to have any `operationId`.

```
$ connexion run your_api.yaml --mock=all -v
```

CHAPTER 3

Routing

3.1 Endpoint Routing to Your Python Views

Connexion uses the `operationId` from each Operation Object to identify which Python function should handle each URL.

Explicit Routing:

```
paths:
  /hello_world:
    post:
      operationId: myapp.api.hello_world
```

If you provided this path in your specification POST requests to `http://MYHOST/hello_world`, it would be handled by the function `hello_world` in `myapp.api` module. Optionally, you can include `x-swagger-router-controller` in your operation definition, making `operationId` relative:

```
paths:
  /hello_world:
    post:
      x-swagger-router-controller: myapp.api
      operationId: hello_world
```

Keep in mind that Connexion follows how HTTP methods work in Flask and therefore HEAD requests will be handled by the `operationId` specified under GET in the specification. If both methods are supported, `connexion.request.method` can be used to determine which request was made.

3.2 Automatic Routing

To customize this behavior, Connexion can use alternative `Resolvers`—for example, `RestyResolver`. The `RestyResolver` will compose an `operationId` based on the path and HTTP method of the endpoints in your specification:

```
from connexion.resolver import RestyResolver

app = connexion.FlaskApp(__name__)
app.add_api('swagger.yaml', resolver=RestyResolver('api'))
```

```
paths:
  /:
    get:
      # Implied operationId: api.get
  /foo:
    get:
      # Implied operationId: api.foo.search
    post:
      # Implied operationId: api.foo.post

  '/foo/{id}':
    get:
      # Implied operationId: api.foo.get
    put:
      # Implied operationId: api.foo.put
    copy:
      # Implied operationId: api.foo.copy
    delete:
      # Implied operationId: api.foo.delete
```

`RestyResolver` will give precedence to any `operationId` encountered in the specification. It will also respect `x-swagger-router-controller`. You may import and extend `connexion.resolver.Resolver` to implement your own `operationId` (and function) resolution algorithm.

3.3 Parameter Name Sanitation

The names of query and form parameters, as well as the name of the body parameter are sanitized by removing characters that are not allowed in Python symbols. I.e. all characters that are not letters, digits or the underscore are removed, and finally characters are removed from the front until a letter or an under-score is encountered. As an example:

```
>>> re.sub('^[^a-zA-Z_]+', '', re.sub('[^0-9a-zA-Z_]', '', '$top'))
'top'
```

Without this sanitation it would e.g. be impossible to implement an OData API.

3.4 Parameter Variable Converters

Connexion supports Flask's `int`, `float`, and `path` route parameter variable converters. Specify a route parameter's type as `integer` or `number` or its type as `string` and its format as `path` to use these converters. For example:

```
paths:
  /greeting/{name}:
    # ...
    parameters:
      - name: name
        in: path
```

(continues on next page)

(continued from previous page)

```
        required: true
        type: string
        format: path
```

will create an equivalent Flask route `/greeting/<path:name>`, allowing requests to include forward slashes in the `name` url variable.

3.5 API Versioning and basePath

You can also define a `basePath` on the top level of the API specification. This is useful for versioned APIs. To serve the previous endpoint from `http://MYHOST/1.0/hello_world`, type:

```
basePath: /1.0

paths:
  /hello_world:
    post:
      operationId: myapp.api.hello_world
```

If you don't want to include the base path in your specification, you can just provide it when adding the API to your application:

```
app.add_api('my_api.yaml', base_path='/1.0')
```

3.6 Swagger JSON

Connexion makes the OpenAPI/Swagger specification in JSON format available from `swagger.json` in the base path of the API.

You can disable the Swagger JSON at the application level:

```
app = connexion.FlaskApp(__name__, specification_dir='swagger/',
                         swagger_json=False)
app.add_api('my_api.yaml')
```

You can also disable it at the API level:

```
app = connexion.FlaskApp(__name__, specification_dir='swagger/')
app.add_api('my_api.yaml', swagger_json=False)
```

CHAPTER 4

Request Handling

Connexion validates incoming requests for conformance with the schemas described in swagger specification.

Request parameters will be provided to the handler functions as keyword arguments if they are included in the function's signature, otherwise body parameters can be accessed from `connexion.request.json` and query parameters can be accessed from `connexion.request.args`.

4.1 Request Validation

Both the request body and parameters are validated against the specification, using jsonschema.

If the request doesn't match the specification connexion will return a 400 error.

4.2 Automatic Parameter Handling

Connexion automatically maps the parameters defined in your endpoint specification to arguments of your Python views as named parameters and with value casting whenever possible. All you need to do is define the endpoint's parameters with matching names with your views arguments.

As example you have an endpoint specified as:

```
paths:
  /foo:
    get:
      operationId: api.foo_get
      parameters:
        - name: message
          description: Some message.
          in: query
          type: string
          required: true
```

And the view function:

```
# api.py file

def foo_get(message):
    # do something
    return 'You send the message: {}'.format(message), 200
```

In this example Connexion will automatically identify that your view function expects an argument named *message* and will assign the value of the endpoint parameter *message* to your view function.

Connexion will also use default values if they are provided.

Warning: Please note that when you have a parameter defined as *not* required at your endpoint and your Python view have a non-named argument, when you call this endpoint WITHOUT the parameter you will get an exception of missing positional argument.

4.2.1 Type casting

Whenever possible Connexion will try to parse your argument values and do type casting to related Python natives values. The current available type castings are:

Swagger Type	Python Type
integer	int
string	str
number	float
boolean	bool
array	list
object	dict

In the Swagger definition if the *array* type is used you can define the *collectionFormat* that it should be recognized. Connexion currently supports collection formats "pipes" and "csv". The default format is "csv".

Note: For more details about *collectionFormat*'s please check the official Swagger/OpenAPI specification.

4.2.2 Parameter validation

Connexion can apply strict parameter validation for query and form data parameters. When this is enabled, requests that include parameters not defined in the swagger spec return a 400 error. You can enable it when adding the API to your application:

```
app.add_api('my_apy.yaml', strict_validation=True)
```

4.2.3 Nullable parameters

Sometimes your API should explicitly accept nullable parameters. However OpenAPI specification currently does not support officially a way to serve this use case, Connexion adds the *x-nullable* vendor extension to parameter definitions. Its usage would be:

```
/countries/cities:
   parameters:
     - name: name
       in: query
       type: string
       x-nullable: true
       required: true
```

It is supported by Connexion in all parameter types: *body*, *query*, *formData*, and *path*. Nullable values are the strings *null* and *None*.

Warning: Be careful on nullable parameters for sensitive data where the strings "null" or "None" can be valid values.

Note: This extension will be removed as soon as OpenAPI/Swagger Specification provide an official way of supporting nullable values.

4.3 Header Parameters

Currently, header parameters are not passed to the handler functions as parameters. But they can be accessed through the underlying `connexion.request.headers` object which aliases the `flask.request.headers` object.

```
def index():
    page_number = connexion.request.headers['Page-Number']
```

4.4 Custom Validators

By default, body and parameters contents are validated against OpenAPI schema via `connexion.decorators.validation.RequestBodyValidator` or `connexion.decorators.validation.ParameterValidator`, if you want to change the validation, you can override the defaults with:

```
validator_map = {
    'body': CustomRequestBodyValidator,
    'parameter': CustomParameterValidator
}
app = connexion.FlaskApp(__name__, ..., validator_map=validator_map)
```

See custom validator example in `examples/enforcedefaults`.

CHAPTER 5

Response Handling

5.1 Response Serialization

If the endpoint returns a *Response* object this response will be used as is.

Otherwise, and by default and if the specification defines that an endpoint produces only JSON, connexion will automatically serialize the return value for you and set the right content type in the HTTP header.

If the endpoint produces a single non-JSON mimetype then Connexion will automatically set the right content type in the HTTP header.

5.1.1 Customizing JSON encoder

Connexion allows you to customize the *JSONEncoder* class in the Flask app instance *json_encoder* (*connexion.App:app*). If you wanna reuse the Connexion's date-time serialization, inherit your custom encoder from *connexion.apps.flask_app.FlaskJSONEncoder*.

5.2 Returning status codes

There are two ways of returning a specific status code.

One way is to return a *Response* object that will be used unchanged.

The other is returning it as a second return value in the response. For example

```
def my_endpoint():
    return 'Not Found', 404
```

5.3 Returning Headers

There are two ways to return headers from your endpoints.

One way is to return a *Response* object that will be used unchanged.

The other is returning a dict with the header values as the third return value in the response:

For example

```
def my_endpoint():
    return 'Not Found', 404, {'x-error': 'not found'}
```

5.4 Response Validation

While, by default Connexion doesn't validate the responses it's possible to do so by opting in when adding the API:

```
import connexion

app = connexion.FlaskApp(__name__, specification_dir='swagger/')
app.add_api('my_api.yaml', validate_responses=True)
app.run(port=8080)
```

This will validate all the responses using *jsonschema* and is specially useful during development.

5.5 Custom Validator

By default, response body contents are validated against OpenAPI schema via `connexion.decorators.response.ResponseValidator`, if you want to change the validation, you can override the default class with:

```
validator_map = {
    'response': CustomResponseValidator
}
app = connexion.FlaskApp(__name__, ..., validator_map=validator_map)
```

5.6 Error Handling

By default connexion error messages are JSON serialized according to Problem Details for HTTP APIs

Application can return errors using `connexion.problem`.

CHAPTER 6

Security

6.1 OAuth 2 Authentication and Authorization

Connexion supports one of the three OAuth 2 handling methods. (See "TODO" below.) With Connexion, the API security definition **must** include a 'x-tokenInfoUrl' or 'x-tokenInfoFunc (or set `TOKENINFO_URL` or `TOKENINFO_FUNC` env var respectively).

If 'x-tokenInfoFunc' is used, it must contain a reference to a function used to obtain the token info. This reference should be a string using the same syntax that is used to connect an `operationId` to a Python function when routing. For example, an `x-tokenInfoFunc` of `auth.verifyToken` would pass the user's token string to the function `verifyToken` in the module `auth.py`. The referenced function should return a dict containing a `scope` or `scopes` field that is either a space-separated list or an array of scopes belonging to the supplied token. This list of scopes will be validated against the scopes required by the API security definition to determine if the user is authorized.

If 'x-tokenInfoUrl' is used, it must contain a URL to validate and get the token information which complies with RFC 6749.

When both 'x-tokenInfoUrl' and 'x-tokenInfoFunc' are used, Connexion will prioritize the function method. Connexion expects to receive the OAuth token in the `Authorization` header field in the format described in RFC 6750 section 2.1. This aspect represents a significant difference from the usual OAuth flow.

The `uid` property (username) of the Token Info response will be passed in the `user` argument to the handler function.

You can find a minimal OAuth example application in Connexion's "examples" folder.

6.2 HTTPS Support

When specifying HTTPS as the scheme in the API YAML file, all the URIs in the served Swagger UI are HTTPS endpoints. The problem: The default server that runs is a "normal" HTTP server. This means that the Swagger UI cannot be used to play with the API. What is the correct way to start a HTTPS server when using Connexion?

CHAPTER 7

Connexion Cookbook

This section aims to be a cookbook of possible solutions for specific use cases of Connexion.

7.1 Custom type format

It is possible to define custom type formats that are going to be used by the Connexion payload validation on request parameters and response payloads of your API.

Let's say your API deals with Products and you want to define a field *price_label* that have a "money" format value. You can create a format checker function and register that to be used to validate values of "money" format.

Example of a possible schema of Product having an attribute with "money" format that would be defined in your OpenAPI specification:

```
type: object
properties:
  title:
    type: string
  price_label:
    type: string
    format: money
```

Then we create a format checker function for that type of value:

```
import re

MONEY_RE = re.compile('^\$\s*\d+(\.\d\d)?')

def is_money(val):
    if not isinstance(val, str):
        return True
    return MONEY_RE.match(val)
```

The format checker function is expected to return *True* when the value matches the expected format and return *False* when it doesn't. Also is important to verify if the type of the value you are trying to validate is compatible with the format. In our example we check if the *val* is of type "string" before performing any further checking.

The final step to make it work is registering our *is_money* function to the format "money" in json_schema library. For that, we can use the draft4 format checker decorator.

```
from jsonschema import draft4_format_checker

@draft4_format_checker.checks('money')
def is_money(val):
    ...
```

This is all you need to have validation for that format in your Connexion application. Keep in mind that the format checkers should be defined and registered before you run your application server. A full example can be found at https://gist.github.com/rafaelcaricio/6e67286a522f747405a7299e6843cd93

7.2 CORS Support

CORS (Cross-origin resource sharing) is not built into Connexion, but you can use the flask-cors library to set CORS headers:

```
import connexion
from flask_cors import CORS

app = connexion.FlaskApp(__name__)
app.add_api('swagger.yaml')

# add CORS support
CORS(app.app)

app.run(port=8080)
```

CHAPTER 8

Exception Handling

8.1 Rendering Exceptions through the Flask Handler

Flask by default contains an exception handler, which connexion's app can proxy to with the `add_error_handler` method. You can hook either on status codes or on a specific exception type.

Connexion is moving from returning flask responses on errors to throwing exceptions that are a subclass of `connexion.problem`. So far exceptions thrown in the OAuth decorator have been converted.

8.2 Default Exception Handling

By default connexion exceptions are JSON serialized according to Problem Details for HTTP APIs

Application can return errors using `connexion.problem` or exceptions that inherit from both `connexion.ProblemException` and a `werkzeug.exceptions.HttpException` subclass (for example `werkzeug.exceptions.Forbidden`). An example of this is the `connexion.exceptions.OAuthProblem` exception

```
class OAuthProblem(ProblemException, Unauthorized):
    def __init__(self, title=None, **kwargs):
        super(OAuthProblem, self).__init__(title=title, **kwargs)
```

8.3 Examples of Custom Rendering Exceptions

To custom render an exception when you boot your connexion application you can hook into a custom exception and render it in some sort of custom format. For example

```
import connexion
from connexion.exceptions import OAuthResponseProblem

def render_unauthorized(exception):
```

(continues on next page)

(continued from previous page)

```
    return Response(response=json.dumps({'error': 'There is an error in the oAuth␣
↪token supplied'}), status=401, mimetype="application/json")

app = connexion.FlaskApp(__name__, specification_dir='./../swagger/', debug=False,␣
↪swagger_ui=False)
app = app.add_error_handler(OAuthResponseProblem, render_unauthorized)
```

8.4 Custom Exceptions

There are several exception types in connexion that contain extra information to help you render appropriate messages to your user beyond the default description and status code:

8.4.1 OAuthProblem

This exception is thrown when there is some sort of validation issue with the Authorisation Header

8.4.2 OAuthResponseProblem

This exception is thrown when there is a validation issue from your OAuth 2 Server. It contains a `token_response` property which contains the full http response from the OAuth 2 Server

8.4.3 OAuthScopeProblem

This scope indicates the OAuth 2 Server did not generate a token with all the scopes required. This contains 3 properties - `required_scopes` - The scopes that were required for this endpoint - `token_scopes` - The scopes that were granted for this endpoint - `missing_scopes` - The scopes that were not given by the OAuth 2 server that are required to access this endpoint