Connexion Documentation

Release 0.5

Zalando SE

March 31, 2016

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

You can also disable it at the API level:

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

CHAPTER 2

Routing

2.1 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
```

2.2 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.App(__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.post
    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-router-controller`. You may import and extend `connexion.resolver.Resolver` to implement your own `operationId` (and function) resolution algorithm.

2.3 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')
```

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

You can also disable it at the API level:

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

CHAPTER 3

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`.

3.1 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.

3.2 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 a 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.

3.2.1 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".

3.2.2 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. It's 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 paramenters 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 a official way of supporting nullable values.

3.3 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']
```

CHAPTER 4

Response Handling

4.1 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.

4.1.1 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 sezialization, inherit your custom encoder from *connexion.decorators.produces.JSONEncoder*.

4.2 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 second return value in the response. For example

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

4.3 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'}
```

4.4 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.App(__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.

4.5 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 5

Security

5.1 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 set `TOKENINFO_URL` env var) with the URL to validate and get the token information. 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.

5.2 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?