MetaCPAN, Mojolicious and OpenAPI

Overview of how OpenAPI was integrated into MetaCPAN with Mojolicious.

- During meta::hack 3 work with Joel Berger on integrating/documenting OpenAPI
 with the MetaCPAN API via Mojolicious.

What is OpenAPI?

OpenAPI is a specification for designing, documenting, validating and driving your RESTful API.

- The OpenAPI Specification originated as the Swagger specification
- renamed to separate the API description format (OpenAPI) from the open source tooling (Swagger).
- used to provide documentation to an existing API, or when creating a new one.
- For MetaCPAN API, we set out to provide documentation to the existing API, but through discussion, validation and driving the API calls as well.

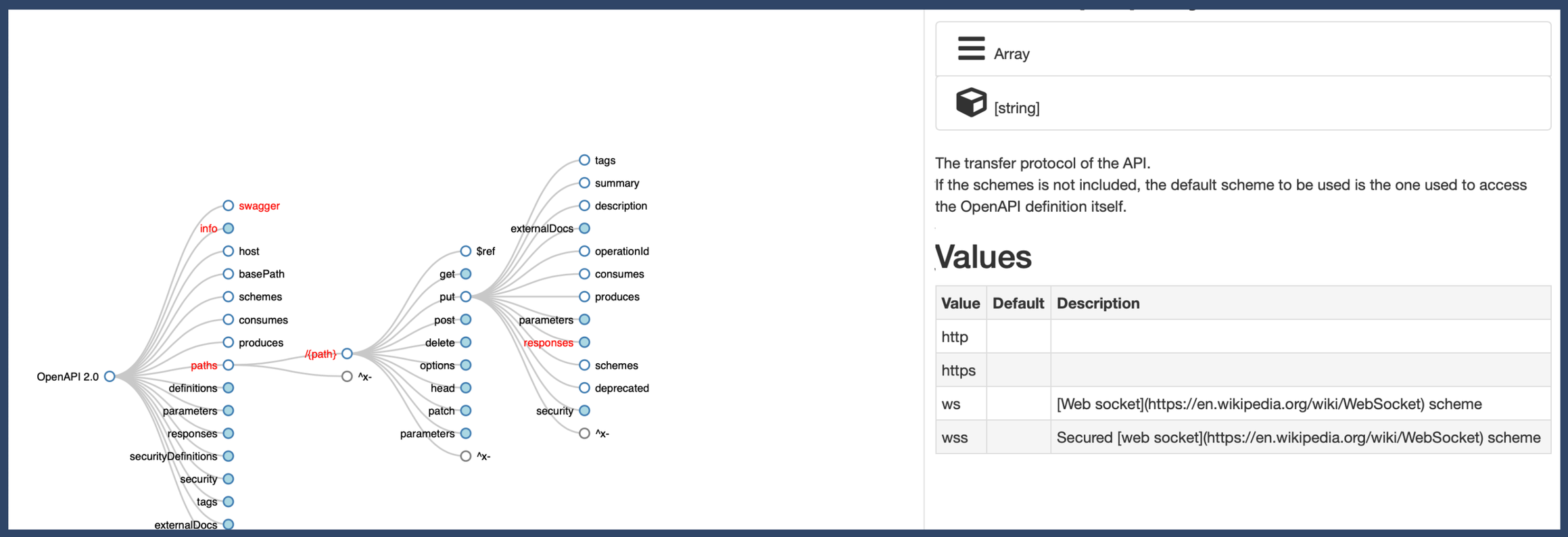
One Source of Truth

- Most importantly, the OpenAPI spec becomes the one source of truth for the API
  - documentation
  - routing
  - validation
  - responses

The Tools

- There are discovery tools that will assist in writing the specification.
- We chose to write the definition by hand (in vim of course) and use tools to
 generate the documentation and to integrate the specification into MetaCPAN.

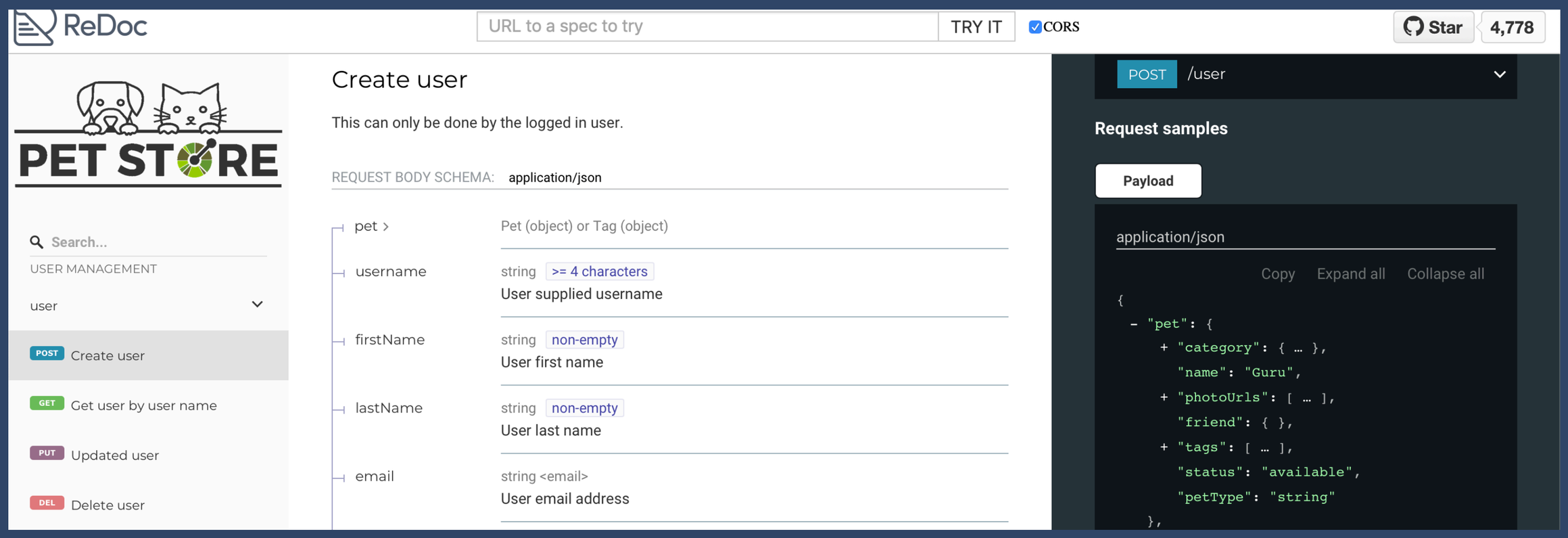
OpenAPI Map



The OpenAPI Map is an interactive site to aid in working with the OpenAPI Specification.

- Interactive map that shows each layers options
- it includes acceptable values and documentation

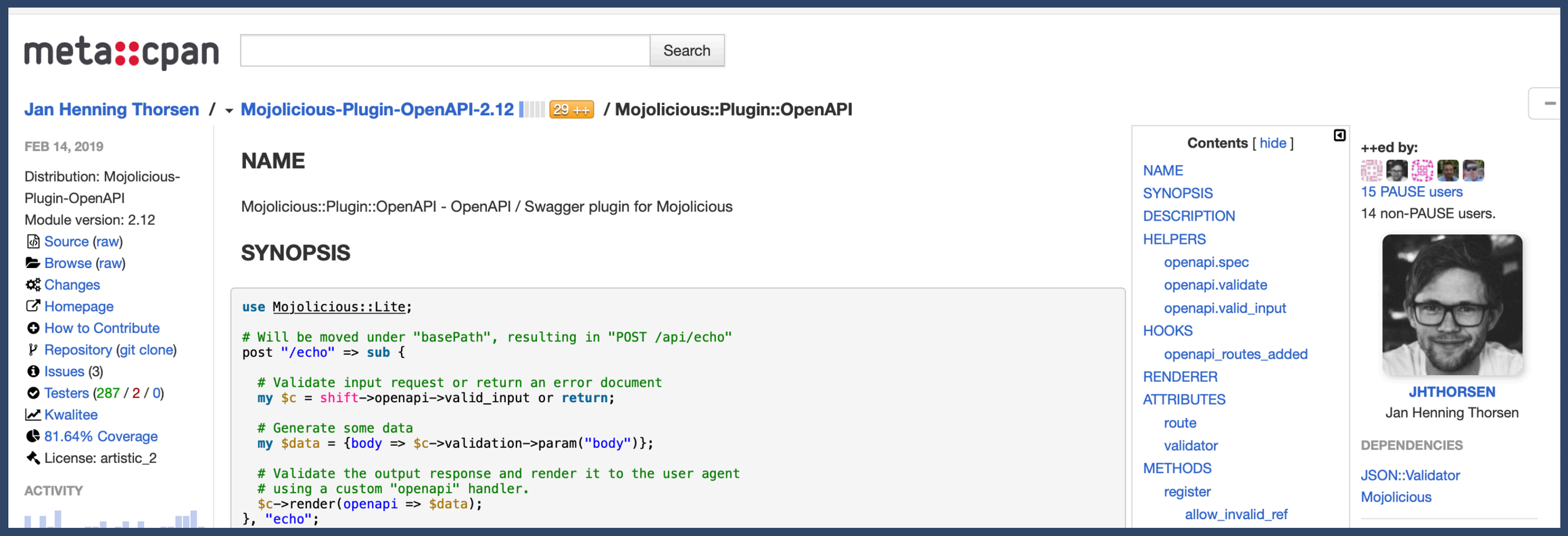
ReDoc



ReDoc – OpenAPI/Swagger-generated API Reference Documentation

- ReDoc creates an interactive page providing documentation and examples based on the details provided in the OpenAPI specification file.
- this is what creates the documentation for your API

Mojolicious::Plugin::OpenAPI

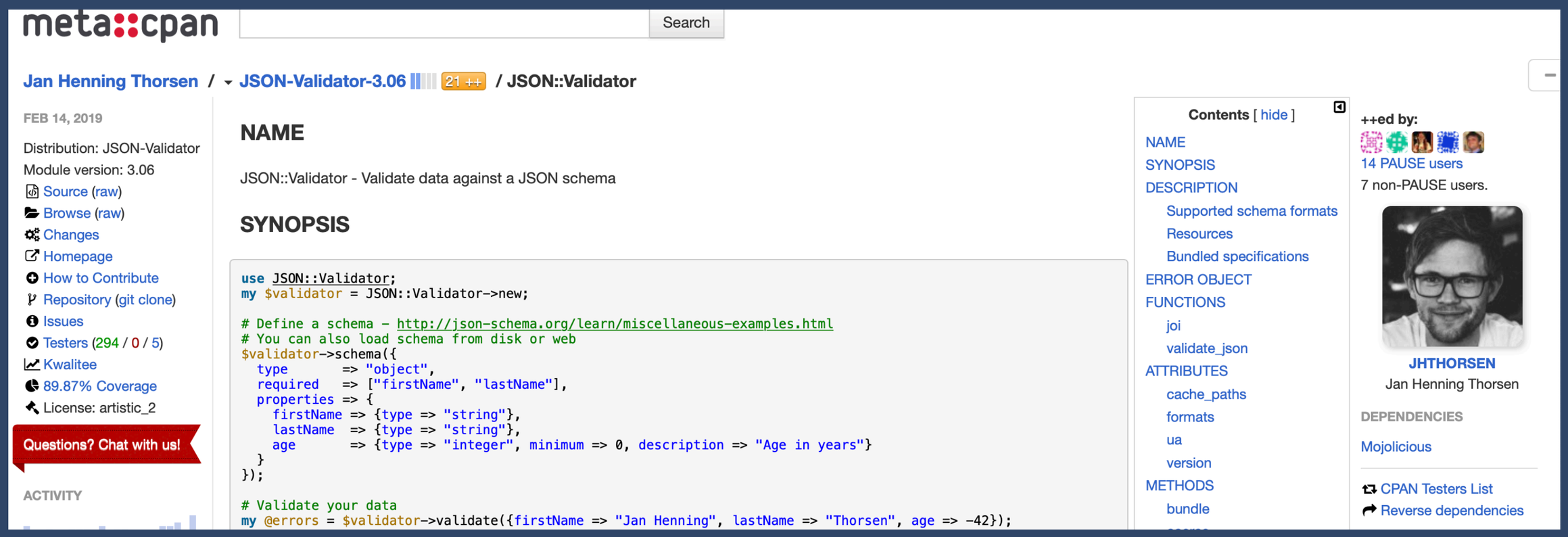


Mojolicious::Plugin::OpenAPI – OpenAPI / Swagger plugin for Mojolicious

— OpenAPI / Swagger plugin for Mojolicious

- Author: Jan Henning Thorsen
- Reads the OpenAPI specification file and adds the appropriate routes and validations for your Mojolicious based application.

JSON::Validator



JSON::Validator – Validate data against a JSON schema

- Author: Jan Henning Thorsen
- Integrated into the Mojolicious::Plugin::OpenAPI module
- provides the input and output validation
- providing validation for the specification file itself.

Getting Started

ReDoc

ReDoc includes a HTML template to be served as a static file for customizing how the documentation is displayed.

```
<!DOCTYPE html>
<html>
  <head>
    <title>MetaCPAN API</title>
    <!-- needed for adaptive design -->
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://fonts.googleapis.com/css?family=Montserrat:300,400,700|Roboto:300,400,700" rel="stylesheet">

    <!--
    ReDoc doesn't change outer page styles
    -->
    <style>
      body {
        margin: 0;
        padding: 0;
      }
    </style>
  </head>
  <body>
    <redoc spec-url="./v1.yml" lazy-rendering="1" required-props-first="1" path-in-middle-panel="1"></redoc>
    <script src="https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js"> </script>
  </body>
</html>
```


- going to start with setting up ReDoc as it is quite simple to configure

ReDoc

ReDoc includes a HTML template to be served as a static file for customizing how the documentation is displayed.

```html
<!DOCTYPE html>
<html>
  <head>
    <title>MetaCPAN API</title>
    <!-- needed for adaptive design -->
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="https://fonts.googleapis.com/css?family=Montserrat:300,400,700|Roboto:300,400,700" rel="stylesheet">

    <!--
    ReDoc doesn't change outer page styles
    -->
    <style>
      body {
        margin: 0;
        padding: 0;
      }
    </style>
  </head>
  <body>
    <redoc spec-url="./v1.yml" lazy-rendering="1" required-props-first="1" path-in-middle-panel="1"></redoc>
    <script src="https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js"> </script>
  </body>
</html>
```


- The import part here is the `redoc` tag and the `spec-url`
- the html and spec file must be in a location that the server will serve static files from

The OpenAPI Specification File

The specification file is `v1.yml`

```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: "MetaCPAN API"
basePath: "/v1"
```


- specification files can be quite long, breaking it down for demonstration purposes
- later I'll show how to organize better
- specification can be in either JSON or YAML
- YAML supports multiline attribute values making it much easier to read and write with less formatting

The OpenAPI Specification File

The specification file is `v1.yml`

```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: "MetaCPAN API"
basePath: "/v1"
```


- Define the version of the OpenAPI spec to use.
- Version 2.0 still uses swagger as the key

The OpenAPI Specification File

The specification file is `v1.yml`

```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: "MetaCPAN API"
basePath: "/v1"
```


- General information about the API

The OpenAPI Specification File

The specification file is `v1.yml`

```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: "MetaCPAN API"
basePath: "/v1"
```


- common path shared throughout the API

Defining an Endpoint

Each of the paths available to the API are defined within the paths object.

```
paths:
  /search/web:
    get:
      operationId: search_web
      x-mojo-to: Search#web
      summary: Perform API search in the same fashion as the Web UI
```


- configuration of API paths are contained in sections named after the URL path to access them

Defining an Endpoint

Each of the paths available to the API are defined within the paths object.

```
paths:
  /search/web:
    get:
      operationId: search_web
      x-mojo-to: Search#web
      summary: Perform API search in the same fashion as the Web UI
```


- the URL path to access this portion of the API
- the `basePath` is automatically prefixed to the URL on consumption

Defining an Endpoint

Each of the paths available to the API are defined within the paths object.

```
paths:
  /search/web:
    get:
      operationId: search_web
      x-mojo-to: Search#web
      summary: Perform API search in the same fashion as the Web UI
```


- The HTTP methods that the endpoint accepts

Defining an Endpoint

Each of the paths available to the API are defined within the paths object.

```
paths:
  /search/web:
    get:
      operationId: search_web
      x-mojo-to: Search#web
      summary: Perform API search in the same fashion as the Web UI
```


- A unique identifier for the method

Defining an Endpoint

Each of the paths available to the API are defined within the paths object.

```
paths:
  /search/web:
    get:
      operationId: search_web
      x-mojo-to: Search#web
      summary: Perform API search in the same fashion as the Web UI
```


- an extension attribute used by Mojolicious
- This attribute points to the name of the class in the application and the method to call separated by #

Defining an Endpoint

Each of the paths available to the API are defined within the paths object.

```
paths:
  /search/web:
    get:
      operationId: search_web
      x-mojo-to: Search#web
      summary: Perform API search in the same fashion as the Web UI
```


- A description of the API Endpoint

Defining Parameters

Each method can define its own parameters.

```yaml
parameters:
  - name: q
    in: query
    description: |
      The query search term. If the search term contains a term with the
      tags `dist:` or `module:` results will be in expanded form, otherwise
      collapsed form.

      See also `collapsed`
    type: string
    # Define the attribute as required
    required: true
  - name: from
    in: query
    description: The offset to use in the result set
    type: integer
    default: 0
  - name: size
    in: query
    description: Number of results per page
    type: integer
    default: 20
  - name: collapsed
    in: query
    description: |
        Force a collapsed even when searching for a particular
        distribution or module name.
    type: boolean
```


- The parameters that the HTTP method accepts

Defining Parameters

Each method can define its own parameters.

```yaml
parameters:
  - name: q
    in: query
    description: |
      The query search term. If the search term contains a term with the
      tags `dist:` or `module:` results will be in expanded form, otherwise
      collapsed form.

      See also `collapsed`
    type: string
    # Define the attribute as required
    required: true
```


- The `parameter` attribute identifies the parameters and accepts an array of objects.

Defining Parameters

Each method can define its own parameters.

```yaml
parameters:
  - name: q
    in: query
    description: |
      The query search term. If the search term contains a term with the
      tags `dist:` or `module:` results will be in expanded form, otherwise
      collapsed form.

      See also `collapsed`
    type: string
    required: true
```


- In this instance the parameter name is *q*
- The location to parse the parameter from is from the query
- parameters are accessible from within Mojolicious by name (I'll show the calls later)
- `in` defines the location of the parameter:
  - this instance is `query`
  - `header`
  - `path`
  - `formData`
  - `body`

Defining Parameters

Each method can define its own parameters.

```
parameters:
  - name: q
    in: query
    description: |
      The query search term. If the search term contains a term with the
      tags `dist:` or `module:` results will be in expanded form, otherwise
      collapsed form.

      See also `collapsed`
    type: string
    required: true
```


- Document what the parameter is.
- This example uses the YAML HEREDOC syntax to make the description easier to read and write.
- using the HEREDOC also allows for paragraph separation in the documentation

Each method can define its own parameters.

```
parameters:
  - name: q
    in: query
    description: |
      The query search term. If the search term contains a term with the
      tags `dist:` or `module:` results will be in expanded form, otherwise
      collapsed form.

      See also `collapsed`
    type: string
    required: true
```


- The type of the value that the API accepts:
  - `array`
  - `boolean`
  - `integer`
  - `number`
  - `null`
  - `object`
  - `string`
- `required` does what it says.

Defining the Response

The OpenAPI specification allows you to define each response to a method call, this includes both specific and generic error handling.

```yaml
responses:
  200:
    description: Search response
    schema:
      type: object
      properties:
        total:
          type: integer
        took:
          type: number
        collapsed:
          type: boolean
        results:
          title: Results
          type: array
          items:
            type: object
```

Defining the Response

The OpenAPI specification allows you to define each response to a method call, this includes both specific and generic error handling.

```yaml
responses:
  200:
    description: Search response
    schema:
      type: object
      properties:
        total:
          type: integer
        took:
          type: number
        collapsed:
          type: boolean
        results:
          title: Results
          type: array
          items:
            type: object
```


- the `responses` attribute contains objects keyed on the HTTP response code or the keyword `default`

Defining the Response

The OpenAPI specification allows you to define each response to a method call, this includes both specific and generic error handling.

```yaml
responses:
  200:
    description: Search response
    schema:
      type: object
      properties:
        total:
          type: integer
        took:
          type: number
        collapsed:
          type: boolean
        results:
          title: Results
          type: array
          items:
            type: object
```


- response codes are defined by HTTP status numbers

Defining the Response

The OpenAPI specification allows you to define each response to a method call, this includes both specific and generic error handling.

```yaml
responses:
  200:
    description: Search response
    schema:
      type: object
      properties:
        total:
          type: integer
        took:
          type: number
        collapsed:
          type: boolean
        results:
          title: Results
          type: array
          items:
            type: object
```


- The *schema* defines what the result will look like

Defining the Response

The OpenAPI specification allows you to define each response to a method call, this includes both specific and generic error handling.

```yaml
responses:
  200:
    description: Search response
    schema:
      type: object
      properties:
        total:
          type: integer
        took:
          type: number
        collapsed:
          type: boolean
        results:
          title: Results
          type: array
          items:
            type: object
```


- The schema defines what the result will look like
  - objects
  - primitives
  - arrays

Defining the Response

The OpenAPI specification allows you to define each response to a method call, this includes both specific and generic error handling.

```yaml
responses:
  200:
    description: Search response
    schema:
      type: object
      properties:
        total:
          type: integer
        took:
          type: number
        collapsed:
          type: boolean
        results:
          title: Results
          type: array
          items:
            type: object
```


- While items can be further broken into properties per item,
- type `object` is a catch all

Error Definitions

Error messages can be defined by HTTP status codes or `default`.

```yaml
default:
  description: "unexpected error"
  schema:
    type: object
    required:
      - code
      - message
    properties:
      code:
        type: integer
        format: int32
      message:
        type: string
```


- I'll admit, this is taken right from the error definitions from the OpenAPI repository
- https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v2.0/yaml/petstore.yaml#L92-L101

Hooking in Mojolicious

- Up until now all this work is defining the API spec
- now to use it

Mojolicious Application

```
package MetaCPAN::API;
use Mojo::Base 'Mojolicious';

sub startup {
    $self->plugin(
        'OpenAPI' => { url => $self->home->rel_file('root/static/v1.yml') } );
}

1;
```


- Additions to the Mojolicious application consist of enabling the plugin

Mojolicious Application

```
package MetaCPAN::API;
use Mojo::Base 'Mojolicious';

sub startup {
    $self->plugin(
        'OpenAPI' => { url => $self->home->rel_file('root/static/v1.yml') } );
}

1;
```


- configures the API plugin
- url points to the OpenAPI specification file

Mojolicious Controller

Configure the controller to use the OpenAPI plugin to validate and extract parameters.

```
package MetaCPAN::API::Controller::Search;

use Mojo::Base 'Mojolicious::Controller';

sub web {
    my $c = shift;
    return unless $c->openapi->valid_input;
    my $args = $c->validation->output;

    my @search = ( @{$args}{qw/q from size/} );
    push @search, $args->{collapsed} if exists $args->{collapsed};
    my $results = $c->model->search->search_web(@search);

    return $c->render( json => $results );
}

1;
```


- this is an example of the Search web interface from MetaCPAN
- as you can see there's not a lot of code to the method

Mojolicious Controller

Configure the controller to use the OpenAPI plugin to validate and extract parameters.

```
package MetaCPAN::API::Controller::Search;

use Mojo::Base 'Mojolicious::Controller';

sub web {
    my $c = shift;
    return unless $c->openapi->valid_input;
    my $args = $c->validation->output;

    my @search = ( @{$args}{qw/q from size/} );
    push @search, $args->{collapsed} if exists $args->{collapsed};
    my $results = $c->model->search->search_web(@search);

    return $c->render( json => $results );
}

1;
```


- validate the parameters against the specification
- return `undef` triggers an error, resulting in a message specifying the validation issue

Mojolicious Controller

Configure the controller to use the OpenAPI plugin to validate and extract parameters.

```
package MetaCPAN::API::Controller::Search;

use Mojo::Base 'Mojolicious::Controller';

sub web {
    my $c = shift;
    return unless $c->openapi->valid_input;
    my $args = $c->validation->output;

    my @search = ( @{$args}{qw/q from size/} );
    push @search, $args->{collapsed} if exists $args->{collapsed};
    my $results = $c->model->search->search_web(@search);

    return $c->render( json => $results );
}

1;
```


- extract the validated parameters into a hashref for use

Mojolicious Controller

Configure the controller to use the OpenAPI plugin to validate and extract parameters.

```
package MetaCPAN::API::Controller::Search;

use Mojo::Base 'Mojolicious::Controller';

sub web {
    my $c = shift;
    return unless $c->openapi->valid_input;
    my $args = $c->validation->output;

    my @search = ( @{$args}{qw/q from size/} );
    push @search, $args->{collapsed} if exists $args->{collapsed};
    my $results = $c->model->search->search_web(@search);

    return $c->render( json => $results );
}

1;
```


- now that the parameters are available, do the work

Advanced Definitions

Reusing definitions through references

Use `$ref` as a relative pointer to a file to include.

```
results:
  title: Results
  type: array
  items:
    $ref: "../definitions/results.yml#/search_result_items"
```

Reusing definitions through references

Use `$ref` as a relative pointer to a file to include.

```
results:
  title: Results
  type: array
  items:
    $ref: "../definitions/results.yml#/search_result_items"
```


- The specification allows for reuse by means of JSON references.
- The `$ref` attribute is a relative pointer to the file
- and the section of the file to find it in (again separated by #)

Reusing definitions through references

Reusing error definitions.

```
responses:
  200:
    description: Release response
    schema:
      type: object
      properties:
        name:
          type: string
  default:
    description: "unexpected error"
    schema:
      $ref: "../definitions/common.yml#/ErrorModel"
```


- excellent use for references is error definitions

Reusing definitions through references

Reusing error definitions.

```yaml
responses:
  200:
    description: Release response
    schema:
      type: object
      properties:
        name:
          type: string
  default:
    description: "unexpected error"
    schema:
      $ref: "../definitions/common.yml#/ErrorModel"
```


- meanings of error codes are fairly static
- use references to reduce repitition

Reusing definitions through references

Some gotchas

- The v2.0 specification does have restrictions on where references can be use, which does cause repetition in the specification file.
- The v3.0 specification has corrected these issues, and also allows for `http` references.

Might be `null`

```yaml
favorites:
  type:
    - integer
    - null
```


- There are times that a property of an object might be `null`.
- In the MetaCPAN API the favourite count may either be an integer representing how many people have favourited the distribution, or `null`.
- Using a list for the property `type` allows the property to contain both.

Words of Advice

The entire specification doesn’t need to be complete in order to get OpenAPI up and running. When documenting an existing API, it’s possible to begin with one portion of the API.

- start small
- spec doesn't have to include everything

Words of Advice

Using an `object` as a response value while developing can help while the full specification is being written. Add `properties` to the response as time permits.

- With MetaCPAN we started with the search endpoints.

Words of Advice

Refactoring the specification to reuse, organize and consolidate is possible after publishing.

- The import part is to get started
- organization will likely change as the specification is fleshed out
- the API may be "static", but the specification contents can evolve

Further Reading

- MetaCPAN spec file
  https://github.com/metacpan/metacpan-api/blob/master/root/static/v1.yml
- MetaCPAN API documentation here
  https://fastapi.metacpan.org/static/index.html
- OpenAPI Specification repository
  https://github.com/OAI/OpenAPI-Specification
  includes full documentation and many examples of varying levels of details.
- ReDoc
  https://github.com/Rebilly/ReDoc

DEMO?

Demo URLs

- MetaCPAN ReDoc API Documentation
- Search Web
- Autocomplete Suggest for Moose
- Validation Failure