# Swagger API documentation for Orphanet – User Guide

API descriptions act as contracts to help users understand how to best work with an API. The description is language-agnostic, and readable by both humans and machines.

With Swagger we adopt a design first approach – the API contract is created before writing any code. The code is then built from the specifications in the contract. This allows better API design.

The contract, or documentation, can be written in JSON or YAML.

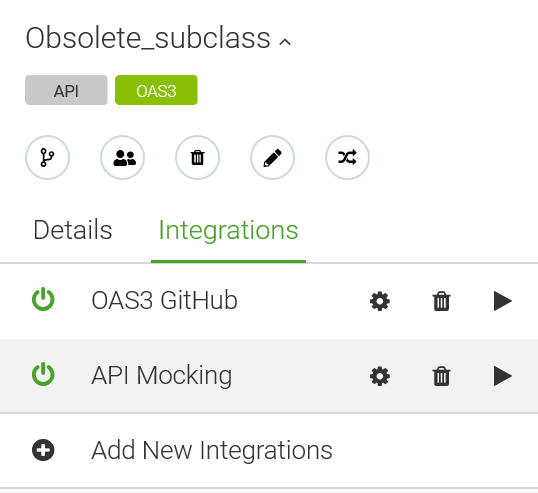
Swagger: <https://app.swaggerhub.com/apis/orphanet/Obsolete_subclass/2-oas3#/>

GitHub: <https://github.com/murphydn/Orphanet_API>

## Swagger integrations

Before starting. It’s a good idea to be aware of some of the features that Swagger offers. Two of the features that I make use of are:

1. GitHub Sync – synchronizes the API definition and code with a specified repository whenever saving the API
2. API Auto Mocking - Automatically creates a mock for the API at [https://virtserver.swaggerhub.com/<owner>/<api>/<version](https://virtserver.swaggerhub.com/%3cowner%3e/%3capi%3e/%3cversion)>



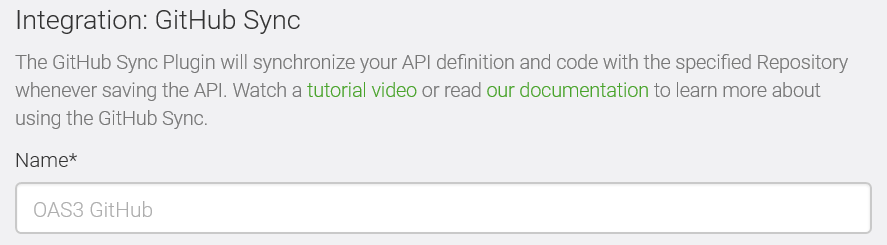
The different integrations can be created, deleted and accessed in the top left part of the Swagger page. They are configured by clicking the cog, deleted by clicking the bin icon, and are put into effect by clicking the play icon.

### GitHub Sync

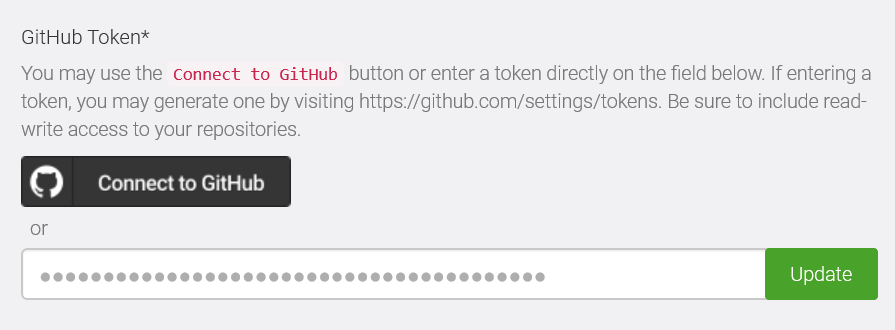
I’ve experienced a number of issues with the GitHub integration. Writing the documentation in YAML rather than JSON seemed to get rid of some of them, although I have managed to get it working with both now. The error messages tend not to be helpful, and sometimes there is no error message at all even when the integration clearly has not worked.

When setting up the GitHub Sync integration there are a number of variables to enter and depending on what you enter it may or may not work. The following steps outline what I did to get a working integration.

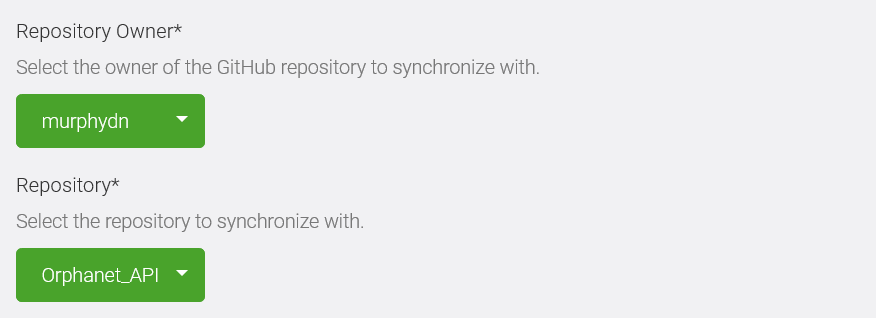
Once you click on the cog icon you can fill in the various entries. The name is arbitrary and can be of your choosing.



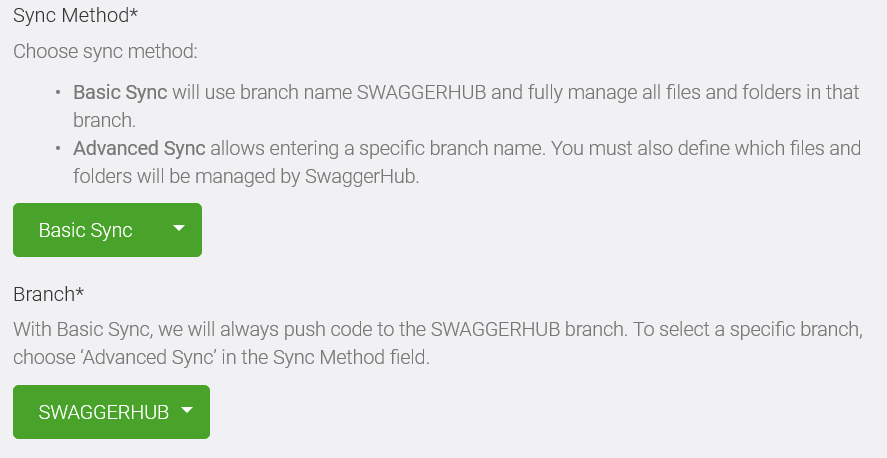
For the GitHub token it’s easiest to log in to your GitHub account using the ‘Connect to GitHub’ button.



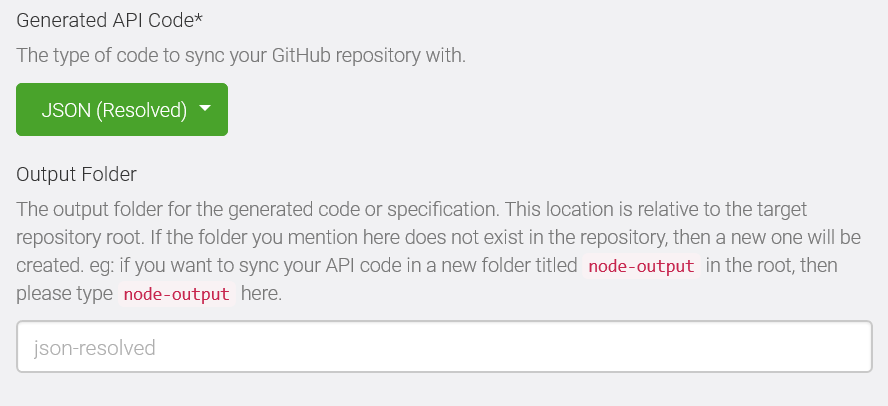
Select the owner of the repository and the repository that you will be syncing with.



For the Sync Method the best option is to just choose ‘Basic Sync’, I couldn’t get ‘Advanced Sync’ to work. The basic sync will also automatically create a branch called SWAGGERHUB.



And then depending on whether your documentation is in JSON or YAML you would select ‘JSON (Resolved)’ or ‘YAML (Resolved)’, respectively. The ‘Output folder’ will also autofill depending on the type of code you select to sync with.

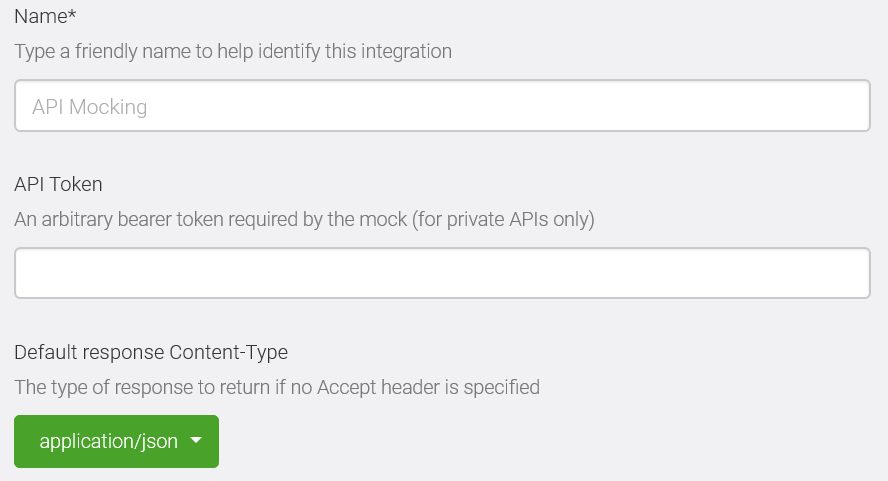


At the bottom of the menu you fill in the filename for the documentation, but for some reason this doesn’t seem to currently have any effect, the file will always just be called openapi.yaml or openapi.json.

With the above details the documentation file I synced has the following location:  
Orphanet\_API/yaml-resolved/openapi.yaml

### API mocking

API mocking creates a virtual server and is very useful for testing and improving the API description. A suitable name is required, the ‘API token’ field is optional. And the response type can be selected as json, yaml or xml.



## API testing

There are two simple ways to test the API documentation:

1. Using the Swagger Virtserver
2. Setting up a local json server

I used the Swagger Virtserver first to check the basic components of the documentation are correct. I then tested the requests using a simple API server I created with json-server.

The servers are specified in the following way:

"servers": [  
 {  
 "url": "https://virtserver.swaggerhub.com/orphanet/Obsolete\_subclass/2-oas3/",  
 "description": "Remote testing host"  
 },  
 {  
 "url": "http://localhost:3000/",  
 "description": "local testing host"  
 }  
]

You select which of them you wish to use on the right of the Swagger page:



Postman (<https://www.getpostman.com/>) can be used for testing your API documentation as well. So far, however, I have found json-server to be more useful as it allows me to create and modify a working server, and to develop the API documentation as if there is a working server to test.

## Setting up a server using json-server

The server is set up using json-server (<https://github.com/typicode/json-server>).

In order to start you need to install cURL, node.js, and json-server. I did this using the Ubuntu app for Windows 10 (<https://tutorials.ubuntu.com/tutorial/tutorial-ubuntu-on-windows#0>):

$ sudo apt install curl

$ curl -sL https://deb.nodesource.com/setup\_8.x | sudo bash -

$ sudo apt-get install -y nodejs

$ npm –v

$ sudo npm install -g json-server

Once installed, you will be able to run json-server with a json file that will represent your database. This file contains the information that you will GET data from and it will be modified when you PUT, POST or DELETE data. The json file (orpha\_db.json) will look something like the following:

{   
 "status": [  
 {  
 "id": 1,  
 "ORPHApacketID": 1000,  
 "status": "retired"  
 },  
 {  
 "id": 2,  
 "ORPHApacketID": 100,  
 "status": "retired"  
 },  
 {  
 "id": 3,  
 "ORPHApacketID": 10,  
 "status": "deprecated"  
 },  
 {  
 "id": 5,  
 "ORPHApacketID": "1000",  
 "status": "retired"  
 }  
 ]  
}

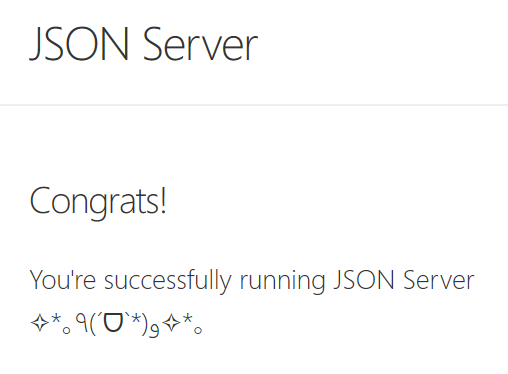
To run json-server with the above file:

$ json-server --watch orpha\_db.json

When json-server is running, if you go to the following URL in your browser:

<http://localhost:3000/>

You should see the following message:



And if you go to this URL:

<http://localhost:3000/status/1>

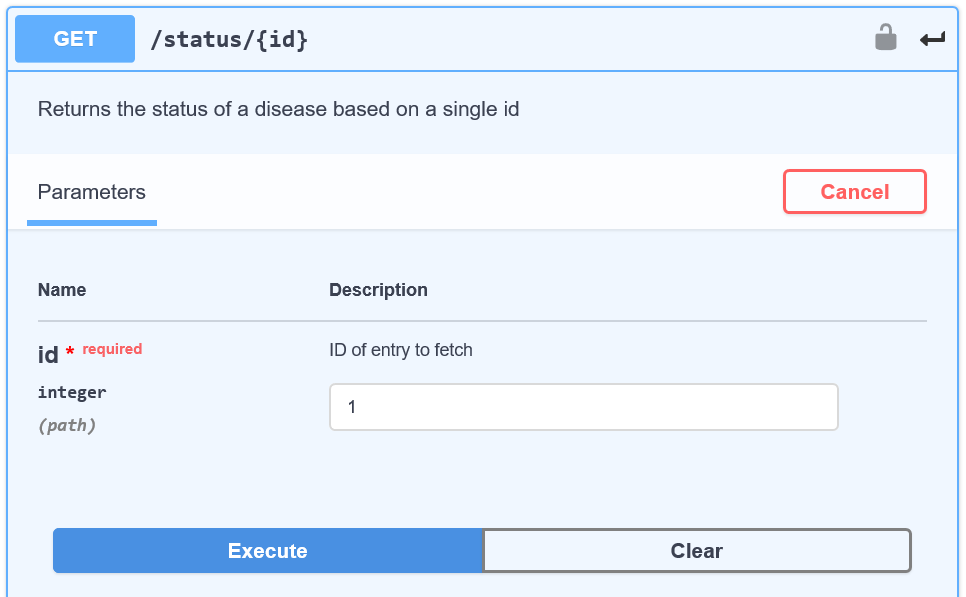
You will get this:

{  
 "id": 1,  
 "ORPHApacketID": 1000,  
 "status": "retired"   
}

## JSON server requests from the command line

Example values have already been specified in the Swagger documentation and you can try out querying with certain values yourself.

To get the Status of the entry with ID equal to 1, for example, you enter the number 1 under parameters for GET/status/{id} and click ‘Execute’.



Swagger then produces a curl request and a request URL. The request URL will be:

http://localhost:3000/status/1

The curl request it returns can be used on the command line and it will return the same data as when you go to the request URL in your browser (as long as json-server is running):

curl -X GET "http://localhost:3000/status/1" -H "accept: application/json"

To POST a new entry, Swagger will return something like the following curl request:

curl -X POST "http://localhost:3000/status" -H "accept: application/json" -H "Content-Type: application/json" -d {"id":200,"ORPHApacketID":5000,"status":"retired"}

PUT will return something similar. You can copy and paste this into the command line and it will modify the orpha\_db.json file to now have the following information:

{  
 "id":200,  
 "ORPHApacketID":5000,  
 "status":"retired"  
}

It’s important to note that the Ubuntu terminal requires quotations, commas and spaces within the new data to be escaped with back slashes. So the curl request should be modified to be:

curl -X POST "http://localhost:3000/status" -H "accept: application/json" -H "Content-Type: application/json" -d {\"id\":200\,\"ORPHApacketID\":5000\,\"status\":\"retired\"}

## Specific queries and query strings

By default the “id” value is used to identify the entry. If you would prefer to query according to a different key, such as ORPHApacketID, then the GET request must adopt the query structure using a ? such as:

curl -X GET "http://localhost:3000/status?ORPHApacketID=1000" -H "accept: application/json"

If you want to retrieve a certain number of entries then you specify this using ‘\_limit’. To retrieve 2 entries:

curl -X GET "http://localhost:3000/status?\_limit=2" -H "accept: application/json"

You can combine queries using &:

curl -X GET "http://localhost:3000/status?status=retired&\_limit=1" -H "accept: application/json"

[  
 {  
 "id": 1,  
 "ORPHApacketID": 1000,  
 "status": "retired"  
 }  
]

## Nested objects

Object structures will rarely be as simple as the ORPHApacket status objects we’ve looked so far. You could have an object that looks like this:

[  
 {  
 "id": 1,  
 "ORPHApacketID": 1000,  
 "status": "retired",  
 "Name": {  
 "label": "ALS",  
 "lang": "eng"  
 }  
 }  
]

And you may want to search for it using the label of “ALS”. To do this you would simply reflect the nested structure in the query:

curl -X GET "http://localhost:3000/status?Name/label=ALS" -H "accept: application/jon"

However, these types of nested queries don’t seem to work with json-server and so cannot be currently tested.

Similarly, json-server does not seem to allow for slashes in the object text in the orpha\_db.json file, which may be necessary in the database endpoint. So the following object in the orpha\_db.json file will give a json-server error when you attempt to run json-server:

"status/findByORPHApacketID": [  
 {  
 "id": 1,  
 "ORPHApacketID": 1000,  
 "status": "active"  
 },  
 {  
 "id": 2,  
 "ORPHApacketID": 100,  
 "status": "retired"  
 },  
 {  
 "id": 3,  
 "ORPHApacketID": 10,  
 "status": "deprecated"  
 }  
]

However, if we are to remove the slash and replace it with something like an underscore like so:

status\_findByORPHApacketID

This will obviously work fine when an underscore is used in the query:

curl -X GET "http://localhost:3000/status\_findByORPHApacketID" -H "accept: application/json"

However, we would like to create our documentation as the API will be used in practice and not have to go back and change it later. Remember that we are only using json-server for testing. We want our API documentation to contain the following query:

curl -X GET "http://localhost:3000/status/findByORPHApacketID" -H "accept: application/json"

But this won’t return anything as the object in the orpha\_db.json file is called status\_findByORPHApacketID

We can get around this issue with json-server using special routes that are specified in the routes.json file. The file essentially contains aliases, such as:

{  
 "/orphapacket/findByORPHApacketID": "/orphapacket\_findByORPHApacketID",  
 "/status/listByStatus": "/status\_listByStatus",  
 "/status/findByORPHApacketID": "/status\_findByORPHApacketID"  
}

This means that while the orpha\_db.json file might contain the object orphapacket\_findByORPHApacketID we can call it using orphapacket/findByORPHApacketID

In order to use those aliases you need to run json\_server while specifying the routes file:

To run json-server with the above file:

$ json-server --watch orpha\_db.json --routes routes.json

So the following will work:

curl -X GET "http://localhost:3000/status/findByORPHApacketID" -H "accept: application/json"

Even though the orpha\_db.json file contains the object status\_findByORPHApacketID

For cases where you want to get entries by a key that isn’t ‘Id’ you need to pass json-server an attribute that states the correct key. For example, if you want the ORPHAnumber to be used as the ID then you would enter the following when starting json-server:

$ json-server --watch orpha\_db.json --routes routes.json --id ORPHAnumber

## Authentication

OpenAPI uses the term security scheme for authentication and authorization schemes. Security is described using the securitySchemes and security keywords. You use securitySchemes to define all security schemes your API supports, then use security to apply specific schemes to the whole API or individual operations.

Security schemes are defined in the components section:

"components": {  
 "securitySchemes": {  
 "api\_key": {  
 "type": "apiKey",  
 "name": "api\_key",  
 "in": "header"  
 },  
 "basic\_auth": {  
 "type": "http",  
 "scheme": "basic"  
 }  
 }

After you have defined the security schemes in the securitySchemes section, you can apply them to the whole API or individual operations by adding the security section on the root level or operation level, respectively.

"security": [  
 {  
 "api\_key": [],  
 "basic\_auth": []  
 }  
 ]