Documenting RESTful APIs Using Swagger and the Open API Specification

Workbook

Jan 26, 2018

Peter Gruenbaum

SDK BRIDGE

Schedule for Jan 26, 2018

10:00 Introduction and YAML
11:00 Break
11:15 API Definition
11:30 OAS Basics
12:15 Lunch
12:45 Schemas
1:30 OAS Continued
2:15 Documentation
3:00 Break
3:15 SwaggerHub
3:45 Final lectures
4:30 Q&A

Exercise 1: YAML

For this exercise, you'll need a simple text editor to create a YAML file. On Windows, you can use NotePad and on Mac OSX, you can use TextEdit. We won't do anything with the file.

Let's say you were writing a YAML file to control a choose-your-own-adventure game. (A choose-your-own-adventure game is where you read part of a story and then you get to make choices as to what happens next in the story.)

Add title and author information

Start with two simple key/value pairs, for title and author. It will look like this:

```
title: Wizard's Choice
author: Delight Games
```

Add a section

Let's now create a list of sections. To do this, create a key/value pair called "sections".

```
sections:
```

Under that, create the first item in sections. It will be indented with a dash. Each section will have an id and a content section.

Note: You can use any number of spaces for indentation, as long as you are consistent throughout the file. Open API specification files tend to have a lot of levels, so I recommend 2 spaces.

```
sections:
  - id: intro
    content:
```

Next, create the content section, which will be a list of paragraphs. Use > so that the first paragraph can span more than one line.

```
    content:
      - >
        You are a young wizard seeking treasure and glory. You
        are walking along a path in the forest. Night has just
        fallen and you're thinking about how it might be a good
        idea to find a campsite. After all, you are in goblin
        territory, and it is dangerous to travel in the dark.
      - Suddenly you smell something awful. What do you do?
```

Finally, add three choices. Add a choice key/value pair, where the value is a list of choices. Each choice will have a description and the id of the section to go to if you make that choice.

```
    choices:
      - description: Dive flat on your face
        id: dive

      - description: Hide
        id: hide

      - description: Stop and listen
        id: stop
```

Add a second section

For practice, add another section. This section will be the section the user would see if they chose "Hide" after reading the first section. It will have:

- id of "hide"
- Two choices:
  - Description "Fight the goblins" leading to section with id "fight"
  - Description "Run away" leading to section with id of "run"

Content:

> When in doubt, hiding is a fine strategy. And the forest offers plenty of cover.
>
> Now inside the brush, you can see green, glowing eyes staring at you from behind a tree several paces away. You hear a snort as several green-skinned goblins charge out of hiding toward you.
>
> What do you do?

Solution

If you get stuck, you can look at my version of the YAML file:
http://sdkbridge.com/swagger/Exercise1Answer.yaml.

Exercise 2: OAS Basics

For this exercise, you'll use the Swagger editor to document some simple requests. Let's say you were writing an OAS file to define an API for a music service. Let's define two requests: one to retrieve a list of playlists, and another to delete a playlist.

To start with, you can refer to the OAS file I created for the photo album requests. Note that the response sections are very basic, just saying that they return 200 and a description that indicates that it's a successful response. Also, remember that lines that start with # are just comments and are ignored, so you aren't required to have those, but they are recommended.

Note: REST resources are sometimes plural and sometimes singular. For example, the URL below could end with **albums** instead of **album**. For this workshop, I've chosen to use the singular, but it's actually more common to use the plural.

The URL for the endpoint is: **https://api.example.com/photo/album**

```
# Every Open API file needs this
swagger: '2.0'

# Document metadata
info:
  version: "0.0.1"
  title: Example Photo Service

# URL data
host: api.example.com
basePath: /photo
schemes:
  - https

# Endpoints
paths:
  # Photo albums
  /album:
    # Get one or more albums
    get:
      # Query parameters
      parameters:
        # Starting date
        - name: start
          in: query
          required: false
          type: string

        # Ending date
        - name: end
          in: query
          required: false
          type: string

      # Incomplete response (to finish later)
      responses:
        # Response code
        200:
          description: Successful response

  # Photo album
  /album/{id}:
    # Get an album
    get:
      # Query parameters
      parameters:
        # Album id
        - name: id
          in: path
          required: true
          type: integer

        # Customer level
        - name: Access-level
          in: header
          required: false
          type: string

       # Incomplete response (to finish later)
      responses:
        # Response code
        200:
          description: Successful response
```

Playlist API

Now it's your turn. You'll create a definition for the API in general, and then two requests: one to retrieve information on one or more playlists, and one that deletes a playlist.

General Information

Open the Swagger editor at: http://editor2.swagger.io

Select everything on the left side and delete it. You'll start from scratch.

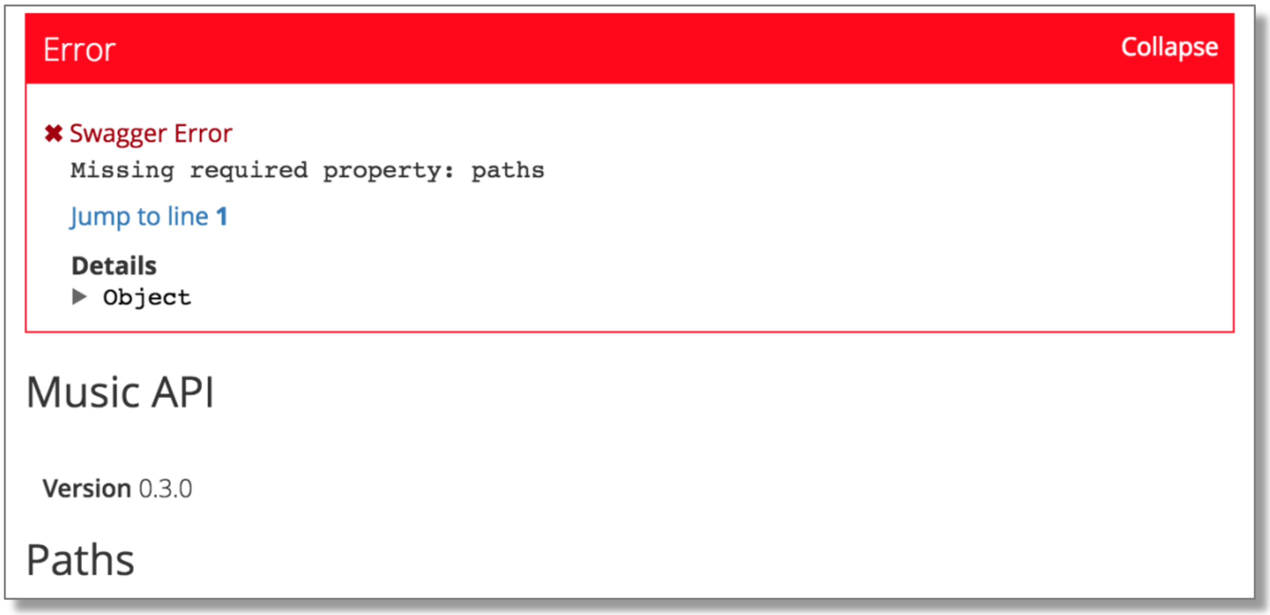
The company whose API you are documenting has the domain muzicplayz.com. (It's not real.) This is version 3 of their API, so the base URL is:

```
        https://api.muzicplayz.com/v3
```

Add these initial pieces into the Swagger file. It's good practice to add comments, but not required. Refer to the sample YAML above if you need help.

1. Always required: a **swagger** key with a value of **'2.0'**
2. An **info** key with two keys: **version** and **title**. Give it a version of 0.3.0 (needs to be in quotes) and a title of "Music API" (shouldn't be in quotes).
3. The host, basePath, and schemes.

If you look on the right side, you should see your API documented so far. You'll see the title and the version.


Error
Collapse

✖ Swagger Error

```
Missing required property: paths
```

Jump to line 1

Details

▶ Object

Music API

Version 0.3.0

Paths

Note that the paths are missing, so we are seeing an error.

Paths and GET Request

The first request to define will return one or more playlists and uses the GET method. Here's a sample request:

```
GET https://api.muzicplayz.com/v3/playlist?limit=10&offset=20&search=jazz
```

The path has this URL:

```
        https://api.muzicplayz.com/v3/playlist
```

Add this to the YAML file:

1. Add a **paths** key.

2. Then add the path as the next key. It's the part of the URL after the base path. It should start with a /
3. Add the HTTP method as the next key. It should be lowercase.
4. Add the **parameters** key

This request will have three query parameters:

Query parameter	Required	Definition
limit	Optional	Number of playlists to return
offset	Optional	Index of the first playlist to return. (0=start at the beginning, 10 = skip the first 10, etc.)
search	Optional	Return playlists whose name contains this string

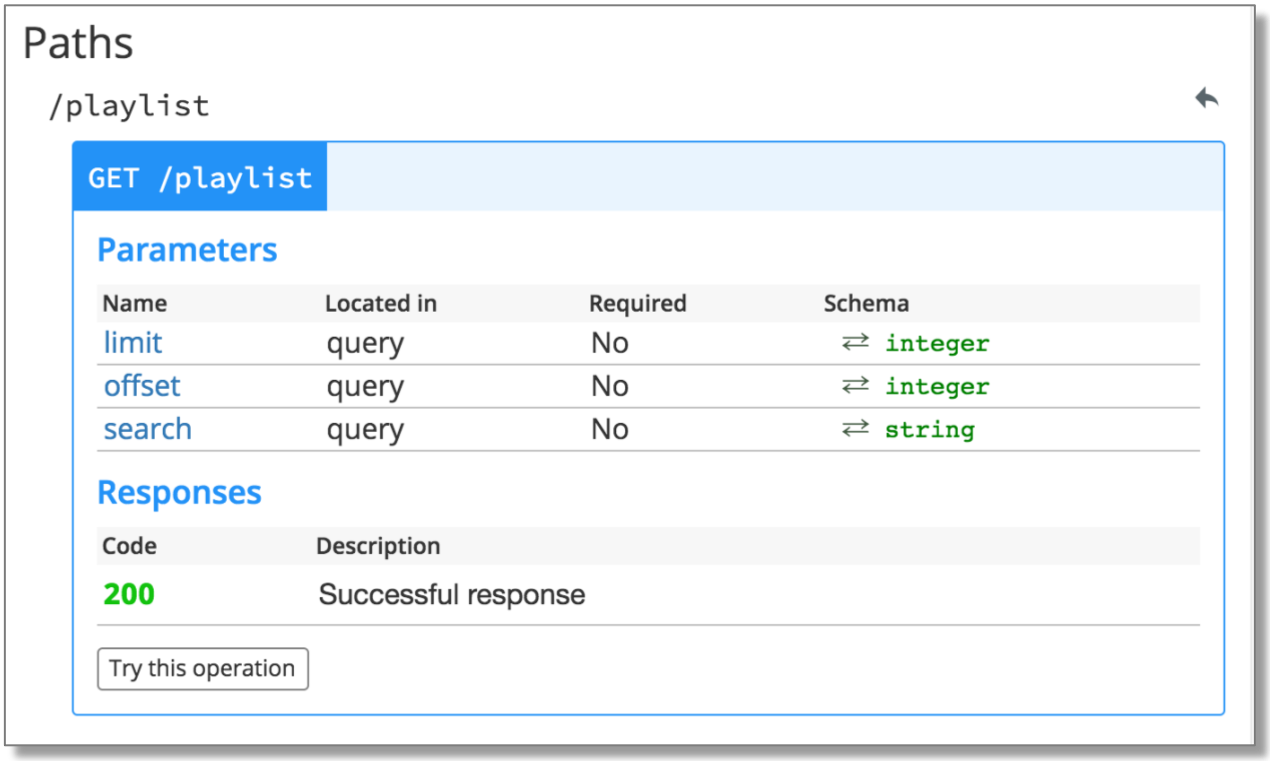
Add list items for each of the query parameters. You will need to have keys for **name**, **in**, **required**, and **type**. See if you can figure those out from the table above and the sample OAS document at the top of this page. Don't forget that you need a dash at the beginning of each list item.

Finally, add a basic response like this:

```
responses:
  # Response code
  200:
    description: Successful response
```

The **responses** key should be at the same indentation as the **parameters** key. We will add more response information in the next exercise.

When done, on the right side, you should see the documentation on the right side:


Paths

/playlist

GET /playlist

Parameters

Name	Located in	Required	Schema
limit	query	No	⇄ integer
offset	query	No	⇄ integer
search	query	No	⇄ string

Responses

Code	Description
200	Successful response

DELETE Request

The second request to define will delete a playlist using the DELETE method. Here is a sample request:

```
DELETE https://api.muzicplayz.com/v3/playlist/playlist333
```

The path has this URL:

```
https://api.muzicplayz.com/v3/playlist/playlist333
```

where `playlist333` is the playlist ID. Note that this ID is a string because it contains both text and numbers.

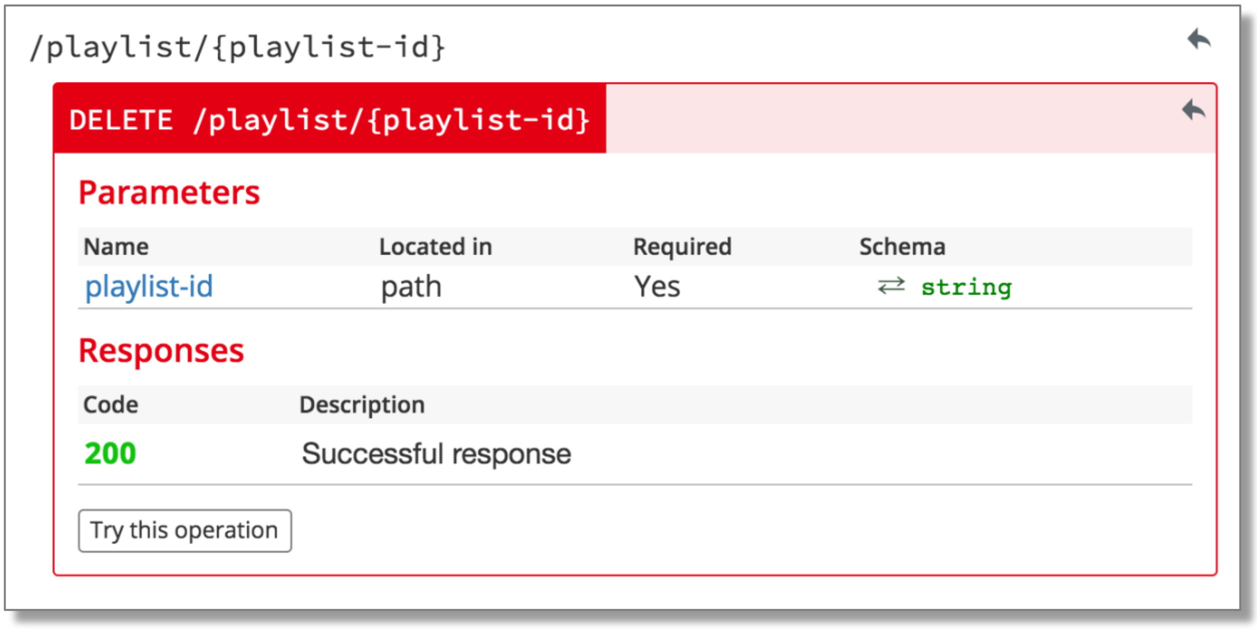
Continue the YAML file:

1. Add the path as the next key. It's the part of the URL after the base path. It should start with a / and contain `{playlist-id}` to indicate a path parameter.
2. Add the HTTP method as the next key.
3. Add the **parameters** key.
4. Add the path parameter as a list item. You will need to have keys for **name**, **in**, **required**, and **type**. See if you can figure those out from the sample OAS document at the top of this page. Don't forget that you need a dash at the beginning because even though there's only one parameter, it's still considered a list item.

Finally, add a basic response like this:

```
      responses:
        # Response code
        200:
          description: Successful response
```

Now your documentation on the right should show this information:



/playlist/{playlist-id}

DELETE /playlist/{playlist-id}

Parameters

Name	Located in	Required	Schema
playlist-id	path	Yes	⇄ string

Responses

Code	Description
200	Successful response

Try this operation

Note that there's a **Try this operation** button on the right side. This will provide information to developers on what information they need to actually make a call to this request.

Click on the **Try this operation** button for the DELETE request. It will open up a box where you can input the playlist ID. You can then click the **Send Request** button. This won't actually work because the API is fictional and there is no server at muzicplayz.com. But if it were real, then clicking that button would actually make a call to the API and show you the results.

Try clicking it anyway. You will get a message: ERROR Server not found or an error occurred.

Save

It's a good idea to save your YAML file after each exercise. From the **File** menu, choose **Download YAML**. Save this somewhere where you can easily get to it for the next exercise.

Solution

If you get stuck, you can look at my version of the OAS file: http://sdkbridge.com/swagger/Exercise2Answer.yaml.

Exercise 3: Schemas

Now you'll add to the YAML file: a POST, PUT, and some responses.

You'll probably want to refer to the OAS file I created for the photo album example and used in the lesson you just watched. You can find it at http://sdkbridge.com/swagger/schemas.yaml.

Add a new schema

Open the Swagger editor at: http://editor2.swagger.io You should see the YAML from the last exercise, but if not, then you can import it from the file you saved.

You are going to add information about the request body for a POST request to create a new playlist, with a name and a list of IDs for each of the songs in the playlist. Here's an example of JSON in the request body:

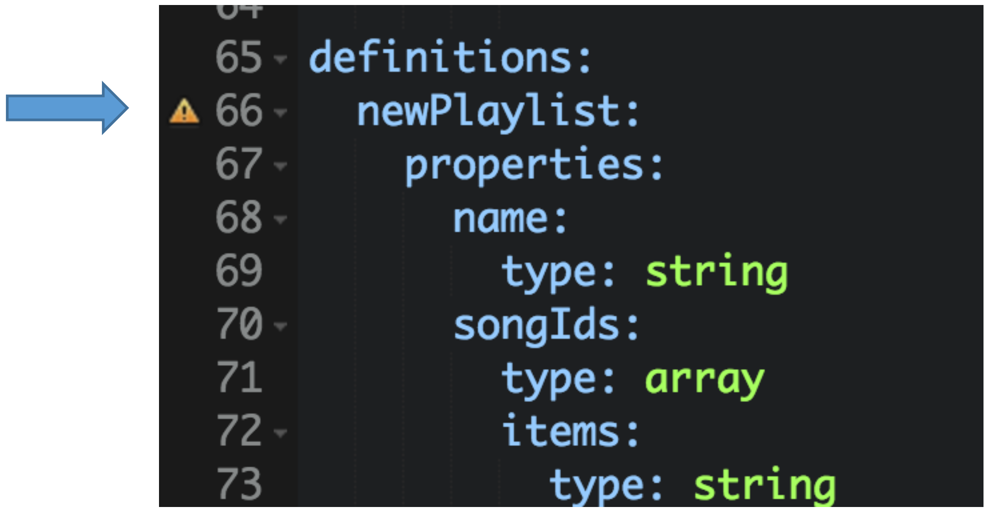
```
{
    "name": "Mellow jazz",
    "songIds": [183, 13, 435, 98, 689]
}
```

Add a **definitions** key at the bottom of the file. Under that, add a **newPlayList** key (indented) and then add a string property for the name and an array of integers, which are the song IDs in the playlist.

```
definitions:
  newPlaylist:
    properties:
      name:
        type: string
      songIds:
        type: array
        items:
          type: integer
```

Now add a **required** key and make the name required, but leave the **songIds** as optional.

At this stage, you should see no errors, but you should see a little warning saying that the definition is not used. Remember, if you are seeing errors, try a refresh on the page.


```
definitions:
  newPlaylist:
    properties:
      name:
        type: string
      songIds:
        type: array
        items:
          type: string
```

Add a POST Request

Add a POST Request to create a new playlist. Here's a sample request

```
POST https://api.muzicplayz.com/v3/playlist

{
    "name": "Mellow jazz",
    "songIds": [183, 13, 435, 98, 689]
}
```

The URL is:

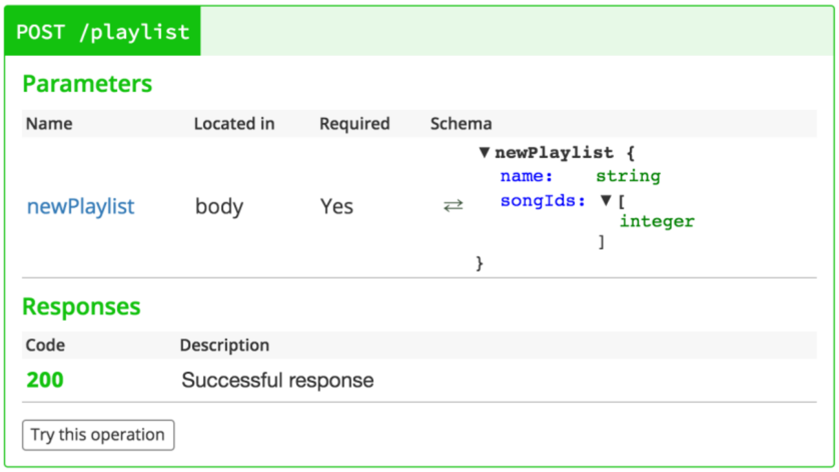
```
https://api.muzicplayz.com/v3/playlist
```

This means that you don't need a new path. You can add the request to the existing /`playlist` path.

1. Below the **get** section, add a similar section called **post**. (Same indentation.)
2. Add a **parameters** key, just like in the **get** section.
3. For **name**, use **newPlaylist**
4. For **in**, use **body**
5. For **required**, use **true**
6. For **schema**, refer to the **newPlaylist** object you've created in the **definitions** section.
7. Copy the simple **responses** section from **get** into **post:**

```
responses:
  # Response code
  200:
    description: Successful response
```

Notice that the warning has gone away, and also that your new POST request appears in the documentation on the left. There is a Schema section that will show your schema once you expand it all: the name as a string, and songIds as an array of integers.



Name	Located in	Required	Schema
newPlaylist	body	Yes	▼newPlaylist { name: string songIds: ▼[integer] }

Code	Description
200	Successful response

Add a GET Response

Let's add GET request to the **/playlist/{playlist-id}** path that returns a playlist.

Here's a sample request:

```
GET https://api.muzicplayz.com/v3/playlist/playlist333
```

Here's a sample response:

```
{
   "id": "playlist333",
   "name": "Mellow jazz",
   "songs":
     [
        {"id": 183, "title": "String of Pearls"},
        {"id": 13, "title": "Stella by Starlight"},
        ...
     ]
}
```

Unlike the **newPlaylist** schema, for the **get**, we don't just want the song IDs. We want information about the songs in the playlist to be returned that we can display it. So we'll create a different schema for returning playlist info.

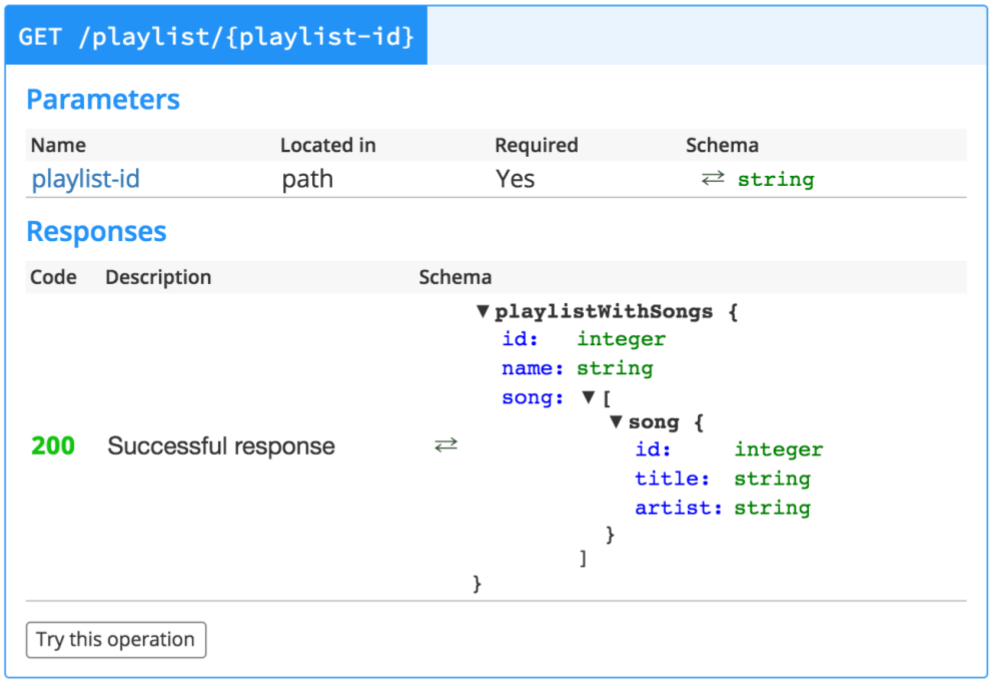
1. Copy the delete section and paste it under itself (including the **responses** section). It should have identical indentation.
2. Change **delete** to **get**
3. The path parameter is the same, so we don't need to modify that.
4. For the 200 response, add a reference to a schema called **playlistWithSongs** that we'll put in the **definitions** section.
5. At the bottom of the file, create a new section for **playlistWithSongs**. It should have the following properties (you won't use the description just yet):

Property	Type	Description
id	integer	ID of the playlist
name	string	Name of the playlist
songs	array	Array of type "song" object. Use the $ref key for this

6. You're going to need to add another section called **song**. It will have these properties:

Property	Type	Description
id	integer	ID of the song
title	string	Name of the song

On the right side, the documentation will look like this. You can see the response schema in the right column, and if you open it all out, you can see down to the song schema.


GET /playlist/{playlist-id}

Parameters

Name	Located in	Required	Schema
playlist-id	path	Yes	⇄ string

Responses

Code	Description	Schema
200	Successful response	⇄

```
▼playlistWithSongs {
   id:   integer
   name: string
   song: ▼[
           ▼song {
              id:     integer
              title:  string
              artist: string
           }
         ]
}
```

Save

It's a good idea to save your YAML file after each exercise. From the **File** menu, choose **Download YAML**. Save this somewhere where you can easily get to it for the next exercise.

Solution

If you get stuck, you can look at my version of the OAS file:
http://sdkbridge.com/swagger/Exercise3Answer.yaml.

Exercise 4: OAS Continued

Let's add some security, errors, content type, and operation IDs to your example.

You'll probably want to refer to the OAS file I created for the photo album example and used in the lesson you just watched. You can find it at http://sdkbridge.com/swagger/openApiCont.yaml.

Add security

Let's start by adding basic authentication as a security method to one of the operations. In a real example, you would probably add the same security to all operations. The only exception is OAuth, where different operations might require different scopes.

1. Open the Swagger editor at: http://editor2.swagger.io You should see the YAML from the last exercise, but if not, then you can import it from the file you saved.
2. You can put security anywhere in the operation, but a logical place would be right above **responses**. It should be the same indentation level as **responses**, and it's very simple:

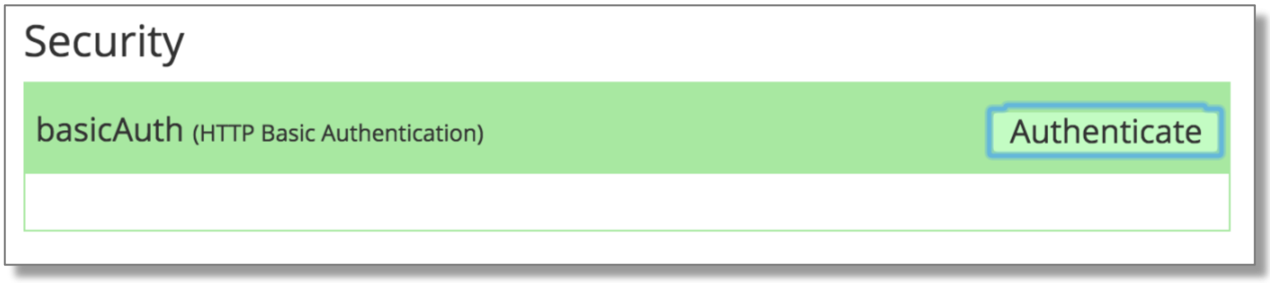
```
security:
  - basicAuth: [ ]
```

I used **basicAuth** as a name for the security type, but you could call it anything, as long as you use the same name in the **securityDefinitions**.

Next, add a **securityDefinitions** key at the bottom of the file. It is at the top level, so has no indentation. Add an item called **basicAuth**, which has a **type** key with value **basic**. Again, it doesn't have to be "basicAuth", but it needs to match what you put in the **security** list.

```
securityDefinitions:
  basicAuth:
    type: basic
```

Look at the right side of the editor, and at the top you'll see a security section. It has the name and it explains that it's using HTTP Basic Authentication. If you click on **Authenticate**, then it will bring up a dialog to ask for username and password. If this were a real API, you would need to do this before clicking on **Try this operation** so that it would have the username and password to send with its requests.



Add an error

Let’s say our API returned JSON in this format when an error occurs:

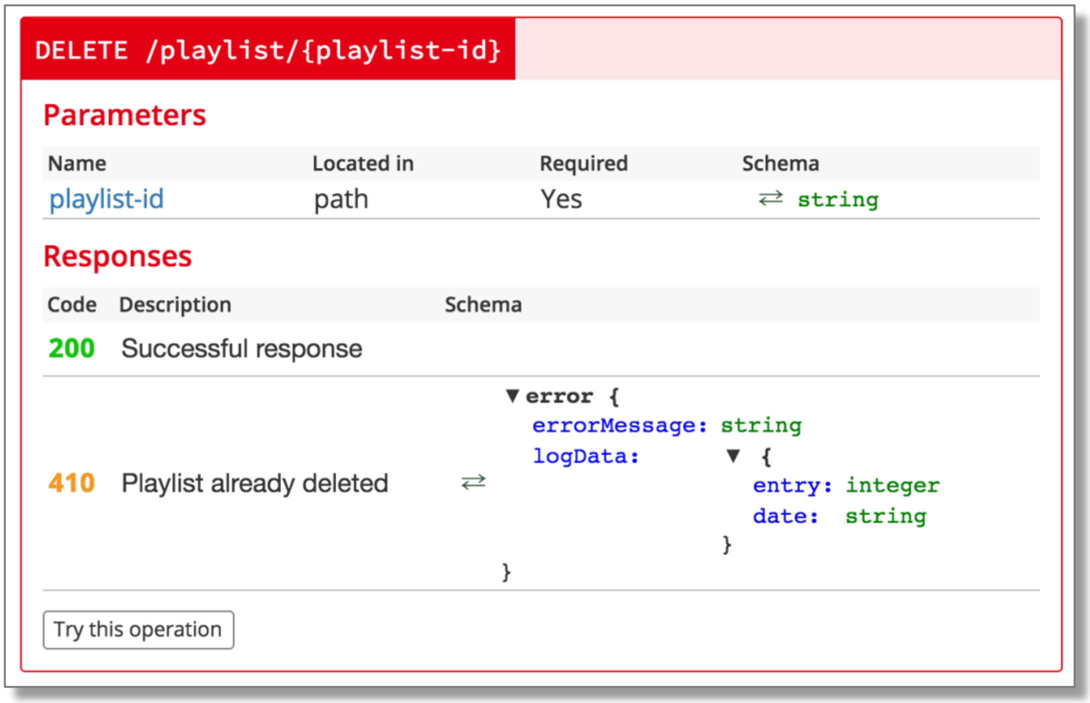
```
{
  "errorMessage": "Playlist with that name already exists",
  "logData": {
    "entry": 3548,
    "date": "2017-10-09 09:40:34"
  }
}
```

In addition to the message, there is data for the server log. A developer could give this information to technical support, and they could find the exact entry in the server log to help debug what is going on.

In the **responses** section, create a new key called **error** and give it properties that reflect the JSON above. Hint: in addition to the **errorMessage** key of type **string**, you will need a **logData** key of type **object**, and then another properties section for that.

Next, go to the **delete** operation and add a response for 410 (Gone). This will handle the case where someone tries to delete a playlist that has already been deleted. Have it refer to the error definition. If the editor is showing you errors, just refresh the page and they should disappear (if you’ve written everything correctly).

Now on the right, you should see your 410 response:


DELETE /playlist/{playlist-id}

Parameters

Name	Located in	Required	Schema
playlist-id	path	Yes	⇄ string

Responses

Code	Description	Schema
200	Successful response	
410	Playlist already deleted	⇄ ▼error { errorMessage: string logData: ▼ { entry: integer date: string } }

Add content type

For the most part, this API will send and receive data in JSON format, so after the **schemes** section, add keys for **consumes** and **produces** with values of **application/json**.

```
consumes:
  - application/json
produces:
  - application/json
```

Let's imagine that our API can generate a PNG image that it generates for the playlist, using album art from the songs in the playlist. It's going to be a new path:

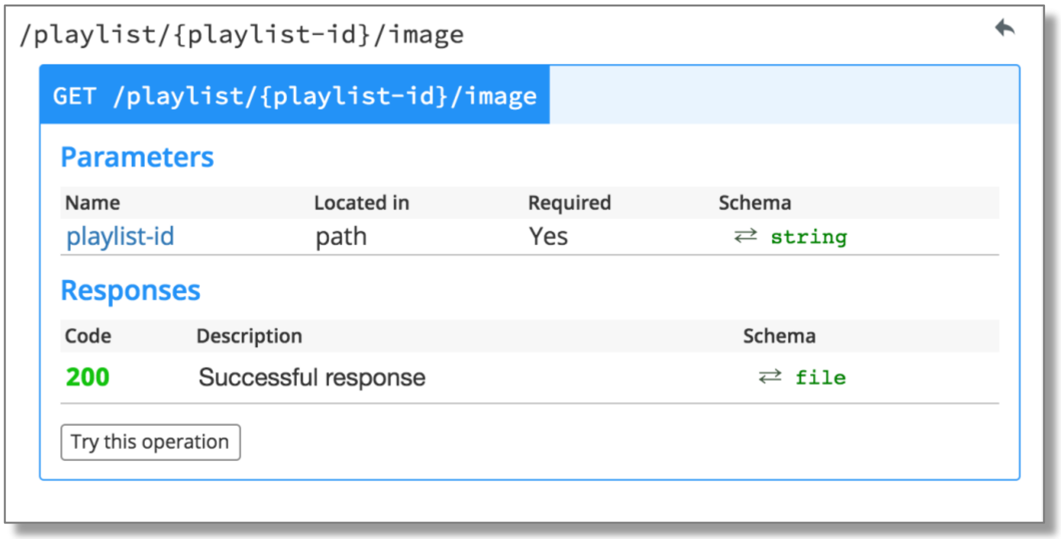
```
GET /playlists/{playlist-id}/image
Response: Image in PNG format
```

1. Add the new path
2. Add a **get** method
3. Copy the parameters from the previous **GET /playlists/{playlist-id}** operation, since this operation has the same parameters (just the path parameter playlist-id).
4. Add a **responses** section that handles the 200 code. Since it's returning a bunch of digital data rather than JSON, the schema type is called **file**.

```
responses:
   200:
    description: Successful response
    schema:
      type: file
```

5. Lastly, add a **produces** section at the same indent level as **responses**. The list should have one value, which is the MIME-type for PNG.
```
produces:
  - image/png
```

When you've done it correctly, the documentation on the right side for the new path should look like this, with "file" listed for the schema:


/playlist/{playlist-id}/image

GET /playlist/{playlist-id}/image

Parameters

Name	Located in	Required	Schema
playlist-id	path	Yes	⇄ string

Responses

Code	Description	Schema
200	Successful response	⇄ file

Add operation ID

Let's add an operation ID to the new image operation, just so you can see what that's like. Under the **get:** add a new line and indent. Then put:

```
operationId: getImage
```

You won't see any changes on the right, but if you've done it correctly, there will be no errors in the editor.

Save

It's a good idea to save your YAML file after each exercise. From the **File** menu, choose **Download YAML**. Save this somewhere where you can easily get to it for the next exercise.

Solution

If you get stuck, you can look at my version of the OAS file:
http://sdkbridge.com/swagger/Exercise4Answer.yaml.

Exercise 5: Documentation

Let's add some documentation to your exercise file.

You may want to refer to the OAS file I created for the photo album example and used in the lesson you just watched. You can find it at http://sdkbridge.com/swagger/photo3.yaml.

Even though it's tedious, I recommend adding description files wherever possible. This is good practice for creating OAS files with excellent documentation. Here are some guidelines and suggestions:

1. To the info section, add a description "Music API with playlists"
2. For each operation, add a description in the present tense, ending with a period (since it's practically a complete sentence). For **get** operations, start with "Returns". For example, the first **get** can be "Returns one or more playlists."
3. Add a description for each parameter. Have it be a short sentence fragment, with no period at the end. For example, the **limit** parameter can be "Number of playlists to return"
4. Each response should have a description. Although most of them do already.
5. Each definition should have a description. For example, the first one could be "New playlist".
6. Each property should have a description. For example, the **name** description in **newPlaylist** could be "Playlist name".
7. Finally, the basicAuth security definition should have a description. For example, "Username and password"

On the right side, you should be able to see each of your descriptions. To see the schema parameter descriptions, you may have to open up the triangles.

Finally, play around with Markdown. Add some *italic* (*italic*), **bold** (**bold**), and `monospace` (`monospace`) words in your descriptions.

Save

Save your YAML file because you will use it in the next exercise. From the **File** menu, choose **Download YAML**. Save this somewhere where you can easily get to it for the next exercise.

Solution

If you get stuck, you can look at my version of the OAS file:
http://sdkbridge.com/swagger/Exercise5Answer.yaml.

Exercise 6: SwaggerHub

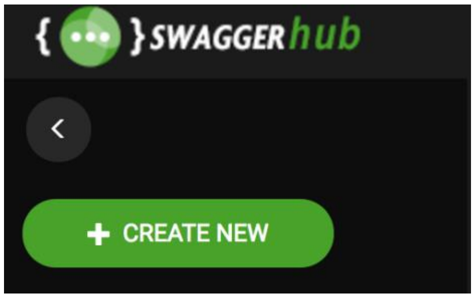
As of the writing of this exercise, SwaggerHub lets you create an account for one use for free. Let's try it out.

Create a SwaggerHub account

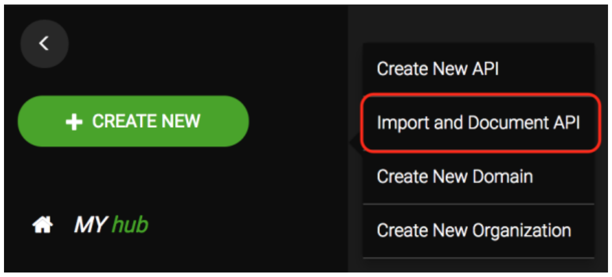
1. Open a browser tab and navigate to https://swaggerhub.com/
2. Click on **Sign Up for Free**.
3. Click on **Start Free** under the **Free** section.
4. Sign up, either through GitHub or using your email
5. When it asks for an organization, click **Skip and Continue with Free Account**.
6. Click **Done**.

Import your OAS file

1. In the top left corner, click on **+ Create New**.



2. Click on **Import and Document API**.

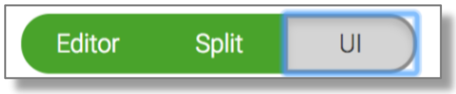


3. Click **Browse** and navigate to the YAML file that you saved from the documentation exercise.
4. Click **Upload File**. You should get the message, "The resource is valid YAML".
5. Click **Import Swagger**.

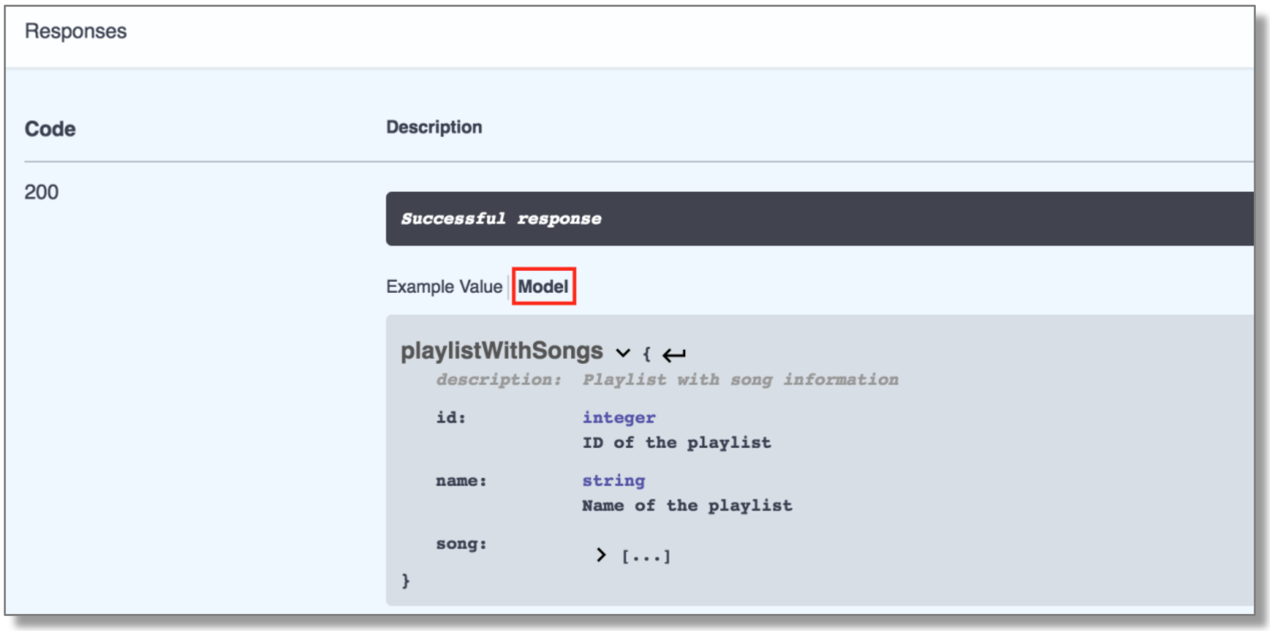
View Documentation

At this point you should see the Swagger editor. It looks like the one you've been using, but it's got some buttons on top that show you just the editor, just the documentation, or a split view of both.

1. Click **UI** to see the full documentation.



2. Note that the documentation is similar to the Swagger editor's right side, but a little nicer.
3. Click on a line (such as **GET**), it will expand and show you the full documentation.
4. Click on the line **GET /playlist/{playlist-id}**. Scroll down to responses and you will now see an example response in JSON. If you click on **Model**, you will see the model with its descriptions.



```
playlistWithSongs ∨ {
    description:  Playlist with song information
    id:           integer
                  ID of the playlist
    name:         string
                  Name of the playlist
    song:          > [...]
}
```

5. Scroll to the bottom to see all of the schemas, which are called Models. Click on the > to open up any model to see more details.

Publish

The API is public, so you can click on the share button at the upper right to share it with anyone: This will give you a URL that anyone can see.

When you feel your API definition is ready for production use, you can publish it. Publishing is a way to show that the API is in a stable state and its endpoints can be reliably called from other applications.

Publishing makes the API definition read-only, so any changes you make after that point will be saved as a different version of the API.

1. Click on the menu with three dots in the upper right:
2. Choose **Publish**.
3. Click **Publish Version**

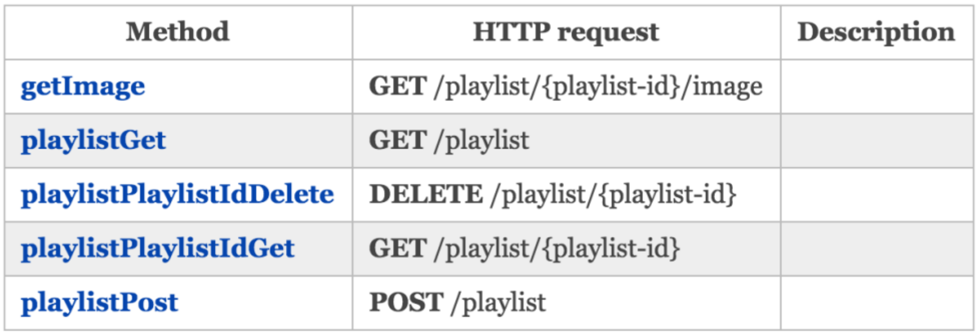
Generate Code

Let's say you wanted to have a JavaScript SDK to make it easy to call your API. SwaggerHub will generate it automatically for you.

1. Click the download button at the upper right:
2. Select **Client**, then **JavaScript**.
3. A file called **javascript-client-generated.zip** will be downloaded.

That's it! Your SDK is ready to go.

If you unzip and look in the **docs** folder, you'll find some Markdown files. You can read these in a Markdown editor. Here's a table that shows some of the methods that were created to make the calls to your API:



Method	HTTP request	Description
getImage	**GET** /playlist/{playlist-id}/image	
playlistGet	**GET** /playlist	
playlistPlaylistIdDelete	**DELETE** /playlist/{playlist-id}	
playlistPlaylistIdGet	**GET** /playlist/{playlist-id}	
playlistPost	**POST** /playlist	

Each of these methods has autogenerated sample code on how to use them.

So that gives you an idea of what you can do on SwaggerHub.

Exercise 7: Final Project

Give yourself an hour or more to create an OAS file from scratch. This time, I will not hold your hand very much. I'll give you bunch of sample requests and responses and related information you might get from a developer's team. This is not a trivial exercise. I suggest having the previous exercise OAS file handy that you can refer to and search this document to see what the values are. You may need to refer back to previous lessons to figure some of this out. Feel free to download the PowerPoint presentations, if that's helpful.

Imagine you are working for a company called MemeMeister. They have an API where you give it an image and some text, and it superimposes the image on the text and returns a new image. This is very useful in creating memes.

There are four APIs: create a meme, get a list of memes and captions, get a meme by ID, and delete a meme. For security, you are using Facebook's OAuth URL, so that you can log in with your Facebook credentials. (In reality, using Facebook to authenticate is more complicated than that.)

Include **description** keys wherever possible so that the documentation is complete.

General Information

The version is 0.1.0. For the **title**, use "Meme Meister". For description, say "API to create memes."

In general, the API consumes and produces JSON.

The API only has responses if successful (200). Those are the only responses you need to document.

Security

Security is using OAuth, so you will need to create a security definition. You may want to review how to do OAuth security.

Call the security definition **oauthFacebook**. For the **authorizationUrl**, put **https://dev.facebook.com/oauth/authenticate** (not a real Facebook URL!). The **flow** is **implicit** (meaning that users go to a separate site for authentication). There are two scopes – one for reading memes and one for writing memes:

```
scopes:
  write:memes: Modify memes in your account
  read:memes: Read memes in your account
```

Be sure to write a description that mentions Facebook.

Creating a new meme

To create a new meme, you make a POST request with a query parameter that contains the captions as query parameters. The body of the post will contain the image. The software will combine these to create a new image, which it returns in the response body.

Request:

Sample request:

```
POST
https://dev.mememeister.com/v1/meme?topcaption=excellent&bottomcaption=epic%20fail
```

The caption query parameters are both required. In your description for these query parameters, be sure to mention that the string need to be URL-encoded. (This means that spaces should be written as %20, etc.)

The POST body is of type **file** and is an image. The MIME type it consumes are:

- image/jpeg
- image/gif
- image/png

Hint: To have the body be a file, use these properties for the parameters:

```
in: formData
required: true
type: file
```

Note that you use **formData** instead of **body** for **in**. In addition to these properties, you should also have a **name** and **description** property. The value of the **name** property isn't important.

The security section should use your security definition and have both scopes (write and read).

Response:

The response produces MIME type image/jpeg, which means the schema is of type **file**. (You don't need a schema definition for this. See how you did it on the last exercise for the **get** that ended in /image.)

Listing all memes

This GET request will return a list of meme IDs and captions. No images are returned. If you want to get an image, then you use the ID to make a call to the other GET request.

Request:

```
GET https://dev.mememeister.com/v1/meme?q=grass
```

The **q** parameter is a search term and is optional. If used, it will filter the responses based on the search. Mention in the description that it’s URL-encoded.

Security is OAuth with the read scope only.

Response:

The 200 code response returns JSON like this:

```
[
  {
    "id": 115,
    "topCaption": "THIS GRASS",
    "bottomCaption": "I'D LIKE TO BURY YOU UNDER IT"
  },
  {
    "id": 543,
    "topCaption": "THE GRASS IS NEVER GREENER",
    "bottomCaption": "ON MARS"
  }
]
```

Create a **definitions** section with a schema for this. Note that what’s returned is a JSON object, not a JSON array. This means that when you reference the schema, you need to mention that it’s of type array, like this:

```
          schema:
            type: array
            items:
              $ref: '#/definitions/memeInfo'
```

Getting a meme

This GET request will return a JPEG given a meme ID.

Request:

```
GET https://dev.mememeister.com/v1/meme/543
```

In this example, 543 is the meme ID, which is a required path parameter.

Security is OAuth with the read scope only.

The response is identical to the POST response: it returns a JPEG file.

Deleting a meme

This DELETE request will delete a meme with the specified ID.

Request:

```
DELETE https://dev.mememeister.com/v1/meme/543
```

In this example, 543 is the meme ID, which is a required path parameter.

If successful, returns a 204 response, meaning no content.

Final words

This is a tough exercise. Don't let yourself give up easily! If you get totally confused, you can check my version at http://sdkbridge.com/swagger/Exercise7Answer.yaml.

Discounts for Online Classes

SDK Bridge offers five online courses to learn API Documentation. This workshop covers part of the first and second courses. For taking this workshop, we are offering discounts on all five courses using coupon TCCAMP.

Learn Swagger and the Open API Specification
~~$35~~ $19.99 with coupon

https://www.udemy.com/learn-swagger-and-the-open-api-specification/?couponCode=TCCAMP

Learn API Technical Writing: JSON and XML
~~$25~~ $13 with coupon

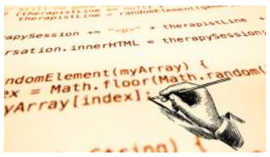
https://www.udemy.com/api-documentation-1-json-and-xml/?couponCode=TCCAMP

Learn API Technical Writing 2: REST
~~$40~~ $20 with coupon

https://www.udemy.com/learn-api-technical-writing-2-rest-for-writers/?couponCode=TCCAMP

The Art of API Documentation
~~$25~~ $13 with coupon

https://www.udemy.com/the-art-of-api-documentation/?couponCode=TCCAMP



Coding for Writers 1: Basic Programming
~~$45~~ $22 with coupon

https://www.udemy.com/coding-for-writers-1-basic-programming/?couponCode=TCCAMP