Swagger™

Supported by SMARTBEAR

Swagger

Pronoide

Version 1.0.1 2022-05-10

Contenidos

Chapter 1. Introducción

Swagger es una herramienta que nos permite describir, crear, consumir y mostrar nuestras APIs.

El proyecto Swagger se compone de distintas herramientas:

- Swagger Editor: es un editor web que nos permite diseñar y editar documentos de OpenAPI. Usa Yaml para describir como tienen que ser los servicios dentro de una API.
- Swagger UI: se encarga de generar la documentación de las APIs que hemos diseñado con el editor.
- Swagger Inspector: es una herramienta que nos permite probar las APIs (similar a Postman) enviando peticiones y obteniendo las respuestas.
- Swagger Codegen: esta herramienta se encarga de generar código en distintos lenguajes, a partir de un documento OpenAPI.
- SwaggerHub: nos permite usar las demás herramientas en la nube de tal forma que:
  - Podemos publicar las APIs (de forma pública o privada).
  - Hacer pública la documentación de las APIs generada con Swagger UI.
  - Permite la colaboración de distintas personas en nuestras APIs.
  - ...

Chapter 2. Swagger Editor

Swagger Editor es una herramienta que nos permite crear la documentación de nuestras APIs utilizando la especificación de Swagger o OpenAPI 3.

A parte de permitirnos rellenar la documentación de la API, nos ayuda durante la definición de esta, mostrándonos los posibles errores y como corregirlos, y sugiriéndonos los distintos tipos de objetos que tenemos que ir rellenando con sus atributos.

2.1. Instalación

Crearemos nuestra documentación utilizando la herramienta Swagger Editor. Para disponer de ella localmente debemos instalar el siguiente software:

- Node.js
- NPM (incluido en la instalación de Node.js)

Para instalar Node.js debemos descargarlo de https://nodejs.org/es/, recomendándose la versión LTS. Durante la instalación aceptaremos las opciones marcadas por defecto.

Una vez instalado Node procederemos a obtener el código de Swagger Editor (no es un software que se instale).

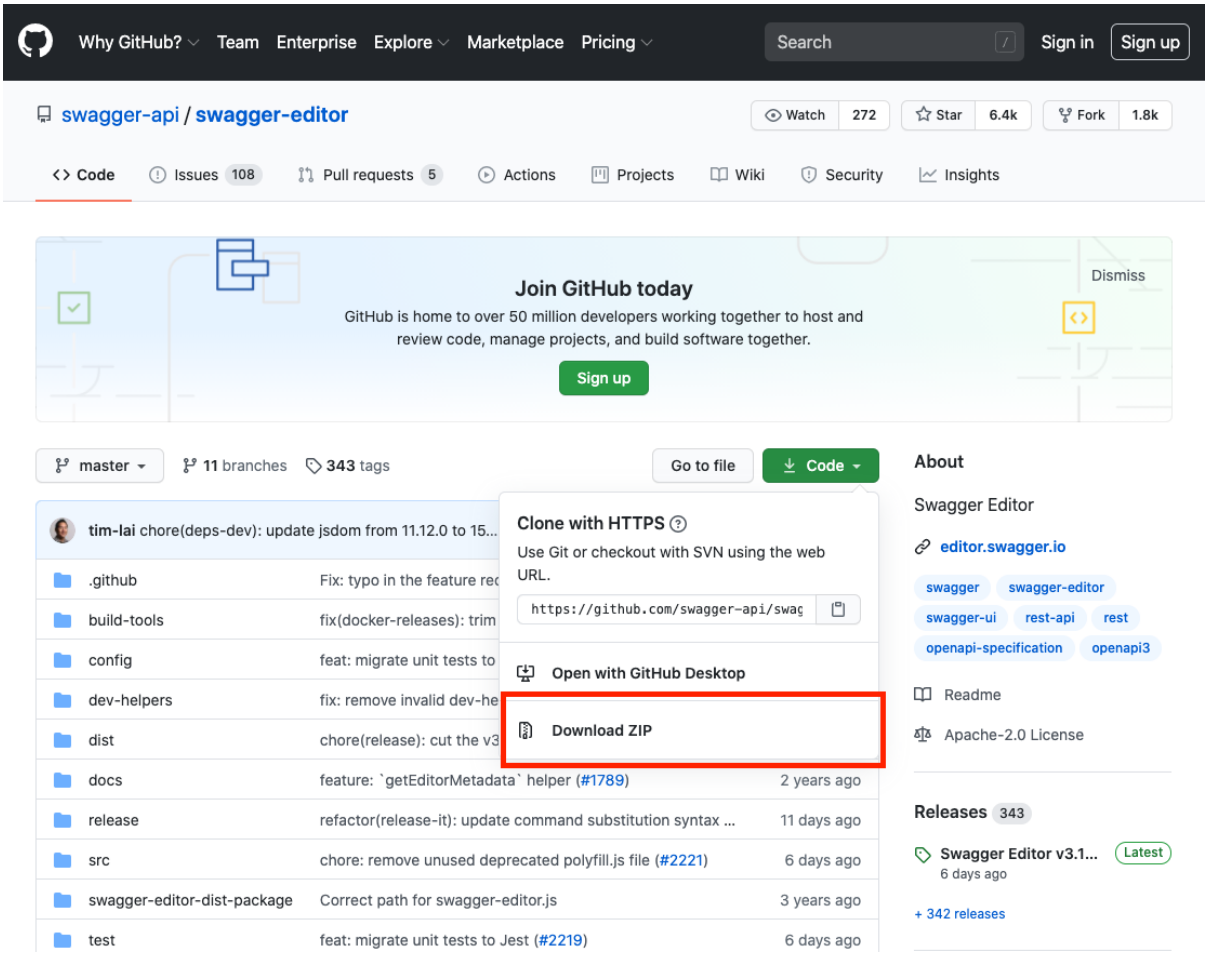
- https://github.com/swagger-api/swagger-editor

Podemos clonar el repositorio (debemos tener instalado git):

- Abrimos el enlace de github que contiene el proyecto de **swagger-editor**.
- Pulsamos sobre el botón **Code** y copiamos la url que aparece en la sección de **Clone with HTTPS**.
- Clonamos el repositorio en el lugar que queramos de nuestro dispositivo lanzando el siguiente comando:

```
$ git clone <URL_COPIADA>
$ git clone https://github.com/swagger-api/swagger-editor.git
```

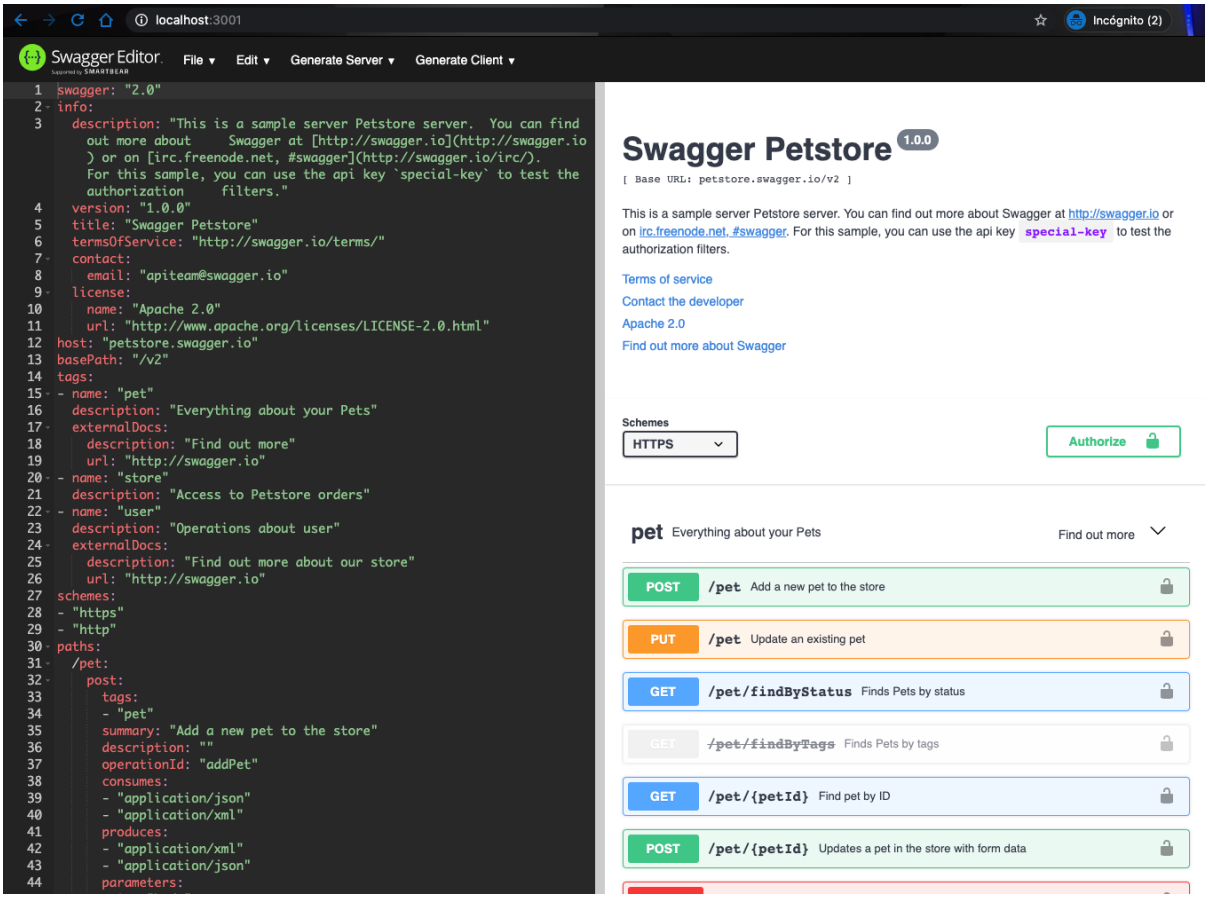
También podemos descargar el repositorio como un zip para descomprimirlo en un directorio a nuestra elección.



A continuación abrimos una consola, navegamos hasta el directorio donde está el código y ejecutamos el siguiente comando:

```
$ npm start
```

Tras unos segundos podemos acceder a la url http://localhost:3001 para comprobar que Swagger Editor está disponible (se trata de una aplicación web).


localhost:3001
Incógnito (2)
Swagger Editor
File
Edit
Generate Server
Generate Client
swagger: "2.0"
info:
description: "This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on [irc.freenode.net, #swagger](http://swagger.io/irc/). For this sample, you can use the api key `special-key` to test the authorization filters."
version: "1.0.0"
title: "Swagger Petstore"
termsOfService: "http://swagger.io/terms/"
contact:
email: "apiteam@swagger.io"
license:
name: "Apache 2.0"
url: "http://www.apache.org/licenses/LICENSE-2.0.html"
host: "petstore.swagger.io"
basePath: "/v2"
tags:
- name: "pet"
description: "Everything about your Pets"
externalDocs:
description: "Find out more"
url: "http://swagger.io"
- name: "store"
description: "Access to Petstore orders"
- name: "user"
description: "Operations about user"
externalDocs:
description: "Find out more about our store"
url: "http://swagger.io"
schemes:
- "https"
- "http"
paths:
/pet:
post:
tags:
- "pet"
summary: "Add a new pet to the store"
description: ""
operationId: "addPet"
consumes:
- "application/json"
- "application/xml"
produces:
- "application/xml"
- "application/json"
parameters:
Swagger Petstore
1.0.0
[Base URL: petstore.swagger.io/v2]
This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key special-key to test the authorization filters.
Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger
Schemes
HTTPS
Authorize
pet Everything about your Pets
Find out more
POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status
GET /pet/findByTags Finds Pets by tags
GET /pet/{petId} Find pet by ID
POST /pet/{petId} Updates a pet in the store with form data

Chapter 3. OpenApi

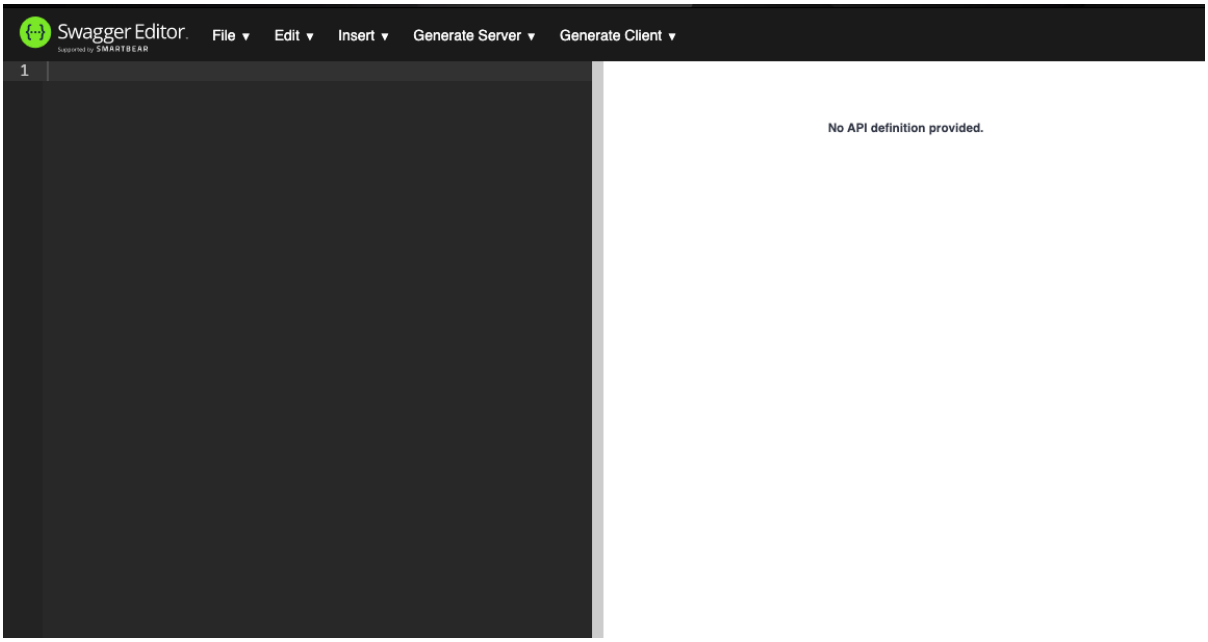
La especificación OpenAPI (OAS) es un estándar para describir de una manera entendible las APIs REST, de tal forma que todo quede bien documentado para que sea mucho más fácil para los desarrolladores de aplicaciones externas o internas, usar nuestros servicios.

Antiguamente era conocida como 'Swagger Specification'.

Utilizando OAS podremos describir con todo detalle una API REST, indicando que URIs incluye, que deben enviar los clientes y qué recibirán como respuesta. Se pueden definir en ficheros JSON o YAML, siendo este último formato el mas recomendable al ser más legible, conciso y mantenible.

Para iniciar la creación de un documento primero debemos limpiar el editor de Swagger Editor (por defecto nos muestra una API de ejemplo). Para ello debemos:

- Seleccionar **File** en el menú.
- Pulsar **Clear editor**.



3.1. Cabecera del documento

El primer campo que se añade dentro del documento es el de la versión de OpenAPI o Swagger que queremos usar para definir nuestra API. Aquí nos encontramos con dos posibles atributos:

- **swagger**: cuando se usa la versión 2 de OpenAPI.
- **openapi**: cuando se usa la versión 3 de OpenAPI.

```
swagger: '2.0'
openapi: '3.0.3'
```

El siguiente campo es el de la información, **info** (obligatorio), perteneciente a la API que vamos a definir. El valor de este atributo es un objeto donde vamos a añadir toda la información

correspondiente a la API, como:

- **title**: nombre de la API.
- **description**: descripción de la API.
- **version**: versión de la API.
- **termsOfService**: url a los términos.
- **contact**: es un objeto que contiene la información de la persona responsable de la API. Los campos de este objeto son:
  - **name**: nombre del responsable.
  - **email**: email del responsable.
  - **url**: url del responsable.
- **license**: es un objeto con los datos de la licencia que se le da a la API. Los campos que contiene el objeto son:
  - **name**: nombre de la licencia.
  - **url**: url a la licencia.

```
info:
  title: Mi API
  description: Descripción de mi API
  version: '1.0.0'
  termsOfService: https://url-a-los-terminos.org
  contact:
    name: Charly Falco
    url: http://www.charlyfal.co
    email: cfalco@gmail.com
  license:
    name: MIT
    url: https://opensource.org/licenses/MIT
```

3.2. Servers

El atributo **servers** contiene una lista de objetos con la información necesaria para conectarse a los servidores donde tenemos nuestra API.

Cada uno de los objetos server llevan los siguientes atributos:

- **url**: la URL destino del servidor.
- **description**: una descripción sobre el servidor.
- **variables**: es una lista de objetos con variables que podemos usar en el valor de la url. Estas variables tienen como clave el nombre de ella, y dentro del objeto lleva los atributos:
  - **description**: una descripción de la variable.
  - **default**: el valor por defecto.

    - **enum**: una lista de valores posibles.

```yaml
servers:
  - url: https://url-produccion
    description: Servidor de producción
  - url: http://url-integracion
    description: Servidor de integración
  - url: https://localhost:{puerto}
    description: Servidor local
    variables:
      puerto:
        description: Número de puerto
        enum:
          - '8080'
          - '3000'
        default: '8080'
```

3.3. Paths

El atributo **paths** (obligatorio) es aquel en el que vamos a incluir todas las operaciones que se van a implementar en la API. Y dentro de cada uno de los paths, pondremos también los tipos de petición como claves, los cuales llevarán más atributos que veremos después:

- **get**
- **post**
- **put**
- **delete**
- **patch**
- **head**
- ...

```yaml
paths:
  /recurso1:
    get:
    post:
    put:
    delete:
```

Además, a parte de los distintos métodos, podemos utilizar también los siguientes atributos:

- **servers**: configuración de servidores que aplica solo a este recurso, sobrescribiendo a la configuración global definida en la raíz del documento. Se muestra al intentar lanzar la petición.
- **parameters**: lista de parámetros que se aplican a todas operaciones que se definen para el

recurso.

- **$ref**: hace referencia a una definición de este objeto que se encuentra en otro lugar.

3.3.1. Operaciones

Dentro de cada uno de los tipos de petición del recurso al que pertenecen tenemos que añadir más atributos con sus valores.

Los atributos que nos encontramos son:

- **summary**: breve descripción de la operación.
- **description**: descripción más larga que la del summary en la que vamos a poner como funciona y cuando usarla, además de los parámetros o información que puede ser importante para alguien que va a utilizar esta operación.
- **operationId**: si decidimos usar un generador de código, el valor de este atributo será el nombre del método encargado de ejecutar el código correspondiente a esta operación.
- **tags**: es una lista de strings que nos ayuda a organizar las operaciones por las palabras clave.
- **deprecated**: marca el path como en desuso (true).
- **externalDocs**: documentación externa adicional para la operación. Son objetos con los atributos:
  - **url**: url donde podemos encontrar la documentación externa.
  - **description**: descripción de la documentación externa.
- **servers**: configuración de servidores que aplica solo a este recurso, sobrescribiendo a la configuración global definida en la raíz del documento. Se muestra al intentar lanzar la petición.
- **parameters**: lista de parámetros que se aplican a la operación.
- **responses**: es una lista con los posibles códigos de estado de las respuestas que se van a devolver al cliente y los datos asociados a estas. Es un atributo obligatorio.
- **requestBody**: es un objeto que define como son los datos que se envían en las peticiones que lo necesitan como POST, PUT...
- **security**: objeto en el que se define el mecanismo de seguridad usado para la operación.

```
/recurso1:
    get:
      tags:
        - tag1
        - tag2
      summary: Obtiene los datos del recurso1
      description: "Esta operación se encarga de obtener los datos del recurso1 en formato JSON o XML. Tenemos que tener
en cuenta que necesitamos estar autenticados..."
      operationId: getRecurso1
      parameters:
        - ...
      deprecated: false
      responses:
        ...
```

Respuestas

Con el atributo **responses** vamos a poder definir todas las respuestas posibles que puede enviarle nuestro servidor al cliente que le haya lanzado una petición.

Este atributo es un objeto donde pondremos como clave de cada respuesta el código de error (200, 404, 500, ...) que se va a mandar.

```
responses:
  200:
    ...
  404:
    ...
```

Y cada una de estas claves puede contener los siguientes atributos:

- **description**: descripción de la respuesta.
- **headers**: es un objeto donde la clave es el nombre de la cabecera y su valor es un objeto que puede tener los siguientes atributos:
  - **description**: descripción de la cabecera.
  - **schema**: define el modelo de datos para las cabeceras.
  - **required**: indica que la cabecera es obligatoria.
  - **deprecated**: indica que la cabecera se ha deprecado.
- **content**: objeto con los distintos formatos (MIME Types) de respuesta que se pueden obtener, como valor de estas claves se le asigna otro objeto con los siguientes atributos:
  - **schema**: con el se define el modelo de datos de los distintos tipos de respuesta.
  - **examples**: nos permite definir diferentes ejemplos para el contenido de la respuesta. El valor es un objeto donde las claves son los nombres dados a los ejemplos, y cada uno de ellos tiene como valor un objeto con los siguientes atributos:
    - **summary**: una descripción corta de lo que se está haciendo en el ejemplo.
    - **description**: una descripción larga de lo que se está haciendo en el ejemplo.
    - **value**: el valor a mostrar como ejemplo.
    - **externalValue**: una url que apunta al ejemplo.
- **links**: permite establecer una relación de una respuesta con la entrada a otra operación, por ejemplo, al crear un usuario, podemos usar su identificador dentro de la petición que elimina el usuario.

```
responses:
  200:
    description: Petición procesada correctamente
    headers:
      mi-cabecera:
        description: Una cabecera nueva
```

```
        schema:
          ...
    content:
      application/json:
        schema:
          ...
        examples:
          ejemplo-1:
            summary: Ejemplo 1
            description: Un ejemplo de la respuesta
            value: '[
              {
                "prop1": "Lorem ipsum",
                "prop2": 2
              }
            ]'
```

Body de las peticiones

Para definir como van a ser los cuerpos de las peticiones que nos manda el cliente, se usa **requestBody** el cual es un objeto en el que podemos usar los siguientes atributos:

- **description**: una descripción que indica que datos van en la petición.
- **required**: indica si es obligatorio o no mandar los datos en el cuerpo de la petición.
- **content**: aquí se indica como van a ser los datos que se mandan en la petición. Podemos indicar como serán estos datos según el formato de datos que se va a mandar, los cuales se ponen como claves de este atributo. Los atributos que se le ponen a cada uno de los tipos de las peticiones son:
  - **schema**: el esquema que define el modelo de datos que se envían en la petición.
  - **examples**: ejemplos de los datos que se envían en las peticiones. Es un objeto cuya clave es un identificador que se le da, y el valor otro objeto en el que tendremos que poner los siguientes atributos:
    - **summary**
    - **description**
    - **value**
    - **externalValue**

```
post:
  summary: Crea un recurso1
  description: "Esta operación se encarga de crear un recurso1 en formato JSON. Tenemos que tener en cuenta que
necesitamos estar autenticados..."
  operationId: createRecurso1
  requestBody:
    description: Datos del recurso1 que se va a crear
    content:
      application/json:
        schema:
          ...
```

```
  responses:
    201:
      ...
```

3.3.2. Parámetros

Dentro de nuestras operaciones, podemos definir los parámetros que se van a mandar en las peticiones HTTP usando el atributo **parameters**.

Este atributo tiene como valor una lista de objetos, los cuales llevan los siguientes atributos:

- **name**: nombre del parámetro. Es obligatorio indicarlo.
- **description**: descripción del parámetro.
- **required**: indica si el parámetro es obligatorio (true) u opcional (false).
- **deprecated**: indica si el parámetro es se ha deprecado.
- **in**: el lugar donde se manda el parámetro, es un campo obligatorio. El valor puede ser:
  - **query**: van detrás del **?** (/recurso1?param1=true).
  - **path**: van en la propia URI (/recurso1/param1). El nombre del parámetro tiene que coincidir con el que se haya puesto en el recurso entre llaves.
  - **header**: van en la cabecera de la petición. Si la cabecera es **Accept** o **Content-Type** no se indica en este objeto, sino con el **content** en las respuestas o el cuerpo de la petición.
- **schema**: nos permite definir el modelo de objeto para el parámetro.

```
/recurso1/{path-param}:
  get:
    parameters:
      - in: query
        name: param1
        description: El param1 sirve para ...
        required: false
        schema:
          type: integer
      - in: path
        name: path-param
        description: El parámetro donde obtenemos ...
        required: true
        schema:
          type: string
    ...
```

3.4. Schema

El atributo **schema** es con el que vamos a poder definir que forma tienen los datos con los que vamos a trabajar, es decir, los parámetros, el contenido de las respuestas, el cuerpo de las peticiones...

El valor de este atributo es un objeto, dentro del cual podemos utilizar:

- **type**: el tipo de dato a usar. Puede ser:
  - **string**
  - **integer**
  - **number**
  - **boolean**
  - **array**: si se le asigna este valor, tenemos que añadir obligatoriamente la clave **items**.
  - **object**: si se le asigna este valor, podemos definir cada una de las claves con su forma dentro de **properties**.
- **items**: con ella definimos la estructura de los items dentro del array. Los atributos que podemos poner son los mismos que con el **schema**.
- **properties**: con este atributo podemos definir como va a ser cada una de las claves dentro del objeto, donde los atributos serán el nombre de la propiedad de dicho objeto y el valor de este será otro objeto con:
  - **type**: el tipo de la propiedad.
  - **description**: descripción de la propiedad del objeto.
  - **example**: ejemplo para mostrar en la documentación.
- **example**: ejemplo para mostrar en la documentación.
- **minimum** o **maximum**: para indicar un valor mínimo o máximo para los números.
- **minLength** o **maxLength**: para indicar una longitud mínima o máxima para el parámetro.
- **default**: el valor que se va a usar por defecto si no se le manda ningún valor.
- **pattern**: string con una expresión regular que debería de cumplir el valor del parámetro para que sea válido.
- **format**: permite indicar el formato que tiene que tener el dato que se está definiendo. Por ejemplo, que un string tenga la forma de un email o una fecha. Los valores posibles son:
  - Para los **integer** puede usar **int32** o **int64**.
  - Para los **number** puede usar **float** o **double**.
  - Para los **string** puede usar **byte** (caracteres codificados en base64), **binary**, **date** (2000-03-24), **date-time** (2000-03-24T23:59:03) o **password** (indicamos que se tiene que ocultar en la vista).

```
content:
  application/json:
    schema:
      type: array
      items:
        type: object
        properties:
          prop1:
```

```
            type: string
            description: Descripción de la prop1.
          prop2:
            type: number
            description: Descripción de la prop2.
```

3.5. Links

Los **links** permiten establecer una relación de datos entre varias operaciones. Por ejemplo, en una respuesta obtenemos un valor que podemos usar en la petición de otra operación.

Este atributo recibe como valor un objeto, donde las claves serán el nombre que le asignamos a cada uno de los links que vamos a generar, los cuales reciben como valor otro objeto con los siguientes atributos:

- **operationId**: hace referencia a la operación que tiene el mismo valor en el atributo **operationId**.
- **description**: una descripción para el link.
- **parameters**: aquí definimos los valores que se pueden mandar a la otra operación. El valor es un objeto donde la clave es el nombre del parámetro que vamos a enlazar y el valor es una referencia a un valor de la operación en la que se define el link.
- **requestBody**: lo mismo que se hace con los parámetros, pero enlazando un valor al cuerpo de la petición de otra operación.

Por tanto teniendo el siguiente recurso:

```
/recurso1/{id}:
  delete:
    operationId: deleteRecurso1
    summary: Elimina un recurso1 dado su id
    description: "Esta operación se encarga de eliminar un recurso1 dado su id como parámetro de path. Tenemos que tener
en cuenta que necesitamos estar autenticados..."
    parameters:
      - in: path
        name: id
        description: El id sirve para ...
        required: true
        schema:
          type: string
    responses:
      200:
        description: Recurso1 eliminado
        content:
          application/json:
            schema:
              type: string
```

Podríamos indicarle que el parámetro **id** lo puede obtener de la respuesta de la siguiente operación:

```
/recurso1:
```

```
  post:
    summary: Crea un recurso1
    description: "Esta operación se encarga de crear un recurso1 en formato JSON. Tenemos que tener en cuenta que
necesitamos estar autenticados..."
    operationId: createRecurso1
    requestBody:
      ...
    responses:
      201:
        description: Recurso1 creado
        content:
          application/json:
            schema:
              type: object
              properties:
                prop1:
                  type: string
                  description: Descripción de la prop1.
        links:
          deleteRecurso1ById:
            operationId: deleteRecurso1
            description: "La prop1 que se recibe en la respuesta se puede usar como id para eliminar el recurso"
            parameters:
              id: $response.body#/prop1
```

3.6. Componentes

Es muy probable que distintos paths utilicen los mismos parámetros, o se utilicen los mismos objetos JSON en las peticiones y las respuestas de la API.

Podemos definir estos elementos de manera global para su reutilización en la sección de **components**.

Dentro de este atributo podemos definir los siguientes campos:

- **schemas**
- **responses**
- **parameters**
- **examples**
- **requestBodies**
- **headers**
- **securitySchemes**
- **links**

```
components:
  schemas:
    ListaResource1:
      type: array
      items:
        type: object
        properties:
```

```
          prop1:
            type: string
            description: Descripción de la prop1.
          prop2:
            type: number
            description: Descripción de la prop2.
```

Una vez creados los distintos componentes, se puede llamar a estos trozos desde distintos lugares de nuestro documento de definición utilizando el atributo **$ref** al que le tenemos que asignar como valor la ruta hasta llegar al componente que queremos utilizar.

```
200:
  ...
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/ListaResource1'
```

3.7. Security

Con swagger también podemos definir como va a ser la seguridad de nuestras APIs.

Dentro de los componentes (**components**) hay que definir las distintas medidas de seguridad que vamos a implementar en la API con el atributo **securitySchemes** que espera recibir como valor un objeto donde las claves serán un nombre para cada medida de seguridad. Cada uno de estos nombres recibe como atributos:

- **type**: el tipo de esquema de seguridad. Es un campo obligatorio que podemos rellenar con:
  - **apiKey**: con este tipo tenemos que añadir los siguientes atributos también:
    - **name**: nombre de la cabecera o parámetro de query donde se va a mandar el token. Es obligatorio.
    - **in**: aquí indicamos como se va a mandar el token de seguridad, donde los valores posibles son **query**, **header** o **cookie**. Es obligatorio.
  - **http**: para este tipo hay que añadir los atributos:
    - **scheme**: es el nombre del esquema de seguridad HTTP que se va a usar. Es un campo obligatorio.
    - **bearerFormat**: una pista para mostrar cual es el formato del token bearer. Este atributo solo se pone si el tipo de autenticación es **bearer**.
  - **oauth2**: para este tipo tenemos que añadir el atributo:
    - **flows**: es un objeto que contiene la información de la configuración de los distintos flujos. Las claves de dicho objeto pueden ser **implicit**, **password**, **clientCredentials** y **authorizationCode**. Y los objetos que se asignan a estas claves son:
      - **refreshUrl**: la url que se usa para obtener nuevos tokens.

        - **scopes**: es un objeto con los scopes (la clave) para el esquema de seguridad de OAuth2. El valor es una pequeña descripción del scope. Este es un campo obligatorio.
        - **tokenUrl**: la token URL usada para los flujos con **password**, **clientCredentials** y **authorizationCode**. Este es un campo obligatorio.
        - **authorizationUrl**: la URL de autorización usada para el flujo de **implicit** y **authorizationCode**. Este es un campo obligatorio.
    - **openIdConnect**: con este tipo hay que añadir el atributo:
        - **openIdConnectUrl**: una url donde se nos indica los valores de configuración de OAuth2.
- **description**: descripción breve del esquema de seguridad que se va a usar.

```
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
    ApiKeyAuth:
      type: apiKey
      name: Token
      in: header
```

Una vez definidos, utilizamos en las operaciones el atributo **security**, el cual recibe una lista con los objetos que indican los esquemas de seguridad que se pueden aplicar, donde la clave será el nombre que pusimos al definirlos en los **components**, y el valor una lista de scopes (si aplica al tipo de seguridad) o un **[]** (si para el tipo de seguridad a usar no se usan scopes).

```
/recurso1:
  post:
    security:
      - ApiKeyAuth: []
    ...
/recurso1/{id}:
  delete:
    security:
      - BearerAuth: []
    ...
```

3.8. Lab: Diseñar una API con OpenAPI 3

En este laboratorio vamos a definir una API con OpenAPI 3. Dicha API se describe a continuación:

- Es una API para gestionar ofertas de trabajo.
- Las operaciones que se van a incluir en esta primera versión son:
    - Crear ofertas de trabajo.
    - Listar todas las que haya creadas.

    - Mostrar una en concreto.
- Una oferta de trabajo se compone de los siguientes campos:
    - **name**: nombre de la oferta de trabajo.
    - **description**: descripción del puesto de trabajo.
    - **salary**: salario a cobrar, será un entero.
    - **city**: ciudad donde se encuentran las oficinas.
    - **company**: nombre de la empresa que ha publicado la oferta de trabajo.
- Se tiene que mostrar la información de contacto del responsable de la API.
- Los datos enviados tanto con las peticiones como las respuestas van en formato JSON.
- Al guardar una oferta de trabajo nueva:
    - Obtenemos un objeto con solo el identificador que le ha asignado la BBDD como valor de un atributo **name**.
    - La oferta de trabajo se guarda con el siguiente formato (cuidado con la respuesta que se obtiene al listar todas las ofertas, ya que no es un array):

```
{
  "idOferta1": {
    "name": "Nombre 1",
    "description": "Descripción 1",
    "salary": 1,
    "city": "Ciudad 1",
    "company": "Empresa 1",
  },
  "idOferta2": {
    "name": "Nombre 2",
    "description": "Descripción 2",
    "salary": 2,
    "city": "Ciudad 2",
    "company": "Empresa 2",
  }
}
```

Empezamos añadiendo la información de la cabecera, es decir, la versión de OpenAPI a usar y la información de la API que estamos definiendo.

/swagger-openapi-lab/fjobs.yaml

```
openapi: '3.0.3'

info:
  title: FJobs API
  description: API donde vamos a gestionar ofertas de trabajo
  version: 1.0.0
  contact:
```

```
    name: Charly Falco
    email: cfalco@fjobs.com
    url: http://www.fjobs.com
```

Ahora viene la parte de los servidores, donde añadiremos dos, uno de producción y otro de testing.

/swagger-openapi-lab/fjobs.yaml

```
servers:
  - url: https://fjobs.com/v1
    description: Servidor de producción
  - url: https://test.fjobs.com/v1
    description: Servidor de testing
```

El siguiente paso es definir los paths que vamos a tener, y como se ha comentado en la descripción, necesitamos 3 operaciones:

- Una para crear nuevas ofertas y listar todas las creadas que pueden compartir el path: **/jobs**.
- Otra para mostrar la información de una oferta en la que necesitaremos indicarle que oferta hay que mostrar de alguna forma. Por ello vamos a utilizar un parámetro de path dejando el path como: **/jobs/{jobId}**.

Por tanto, los paths y operaciones que vamos a tener son:

/swagger-openapi-lab/fjobs.yaml

```
paths:
  /jobs:
    get:
    post:
  /jobs/{jobId}:
    get:
```

Vamos a empezar rellenando el **GET /jobs/{jobId}**, y le añadiremos los atributo **summary** y **description** para indicar de que se encarga esta operación.

/swagger-openapi-lab/fjobs.yaml

```
/jobs/{jobId}:
  get:
    summary: Obtiene una oferta de trabajo
    description: Operación GET para obtener una oferta de trabajo dado su identificador. No es necesario estar
autenticado. Hay que mandar el identificador de la oferta pedida como parámetro.
```

El siguiente paso es definir el parámetro **jobId** el cual será un string, por tanto quedaría de la siguiente forma:

/swagger-openapi-lab/fjobs.yaml

```
/jobs/{jobId}:
```

```
  get:
    summary: Obtiene una oferta de trabajo
    description: Operación GET para obtener una oferta de trabajo dado su identificador. No es necesario estar
autenticado. Hay que mandar el identificador de la oferta pedida como parámetro.
    parameters:
      - in: path
        name: jobId
        description: Identificador de la oferta de trabajo pedida
        required: true
        schema:
          $ref: '#/components/schemas/OfertaId'
        example:
          $ref: '#/components/examples/EjOfertaId/value'
```

Y si os fijáis, el esquema y el ejemplo tienen como valores una referencia. Esto se debe a que posiblemente se puedan reutilizar estos modelos de datos en alguna otra parte de la API, por lo tanto sería mejor extraerlos a un componente.

El siguiente paso es definir esas dos referencias, por lo que añadimos en la raíz el atributo **components** que va a contener esquemas y ejemplos.

/swagger-openapi-lab/fjobs.yaml

```yaml
components:
  schemas:
    OfertaId:
      type: string
  examples:
    EjOfertaId:
      summary: Identificador Oferta
      description: Ejemplo de un identificador de una oferta de trabajo
      value: -LOpEeXtROeJKi0bfoQZ
```

Una vez que tenemos los parámetros ya completos, el siguiente paso es rellenar la parte de la respuesta, donde vamos a necesitar añadir otros dos componentes, uno para la respuesta de esta operación, y otro para el ejemplo de la respuesta, por lo que antes de meternos a definir en la operación la respuesta, vamos a diseñar el modelo de datos de una oferta de trabajo.

Como vimos en la descripción de la API al principio del laboratorio, una oferta tiene que tener este aspecto:

```
{
  "name": "Front-end developer",
  "description": "As a front-end developer you will be joining the development team to work along with the product and
data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
  "city": "Madrid",
  "salary": 25000,
  "company": "E-Corp SL"
}
```

Por lo tanto podemos utilizar esa oferta como ejemplo, y como definir el esquema donde es un objeto que tiene 5 campos, de los cuales uno es un integer y el resto son strings.

/swagger-openapi-lab/fjobs.yaml

```
components:
  schemas:
    OfertaId:
      type: string
    OfertaTrabajo:
      type: object
      properties:
        name:
          type: string
        description:
          type: string
        city:
          type: string
        salary:
          type: integer
        company:
          type: string
  examples:
    EjOfertaId:
      summary: Identificador Oferta
      description: Ejemplo de un identificador de una oferta de trabajo
      value: -LOpEeXtROeJKi0bfoQZ
    EjOfertaTrabajoFront:
      summary: Oferta Frontend
      description: Ejemplo de una oferta de trabajo de Frontend
      value: '{
        "name": "Front-end developer",
        "description": "As a front-end developer you will be joining the development team to work along with the product
and data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
        "city": "Madrid",
        "salary": 25000,
        "company": "E-Corp SL"
      }'
```

Una vez definidos estos modelos, vamos a la operación a terminar de completarla.

Tendremos una respuesta con código 200 en la que vamos a recibir los datos en formato JSON, por tanto, nuestra operación quedaría de la siguiente forma:

/swagger-openapi-lab/fjobs.yaml

```
/jobs/{jobId}:
  get:
    summary: Obtiene una oferta de trabajo
    description: Operación GET para obtener una oferta de trabajo dado su identificador. No es necesario estar
autenticado. Hay que mandar el identificador de la oferta pedida como parámetro.
    parameters:
      - in: path
        name: jobId
        description: Identificador de la oferta de trabajo pedida
        required: true
        schema:
          $ref: '#/components/schemas/OfertaId'
        example:
          $ref: '#/components/examples/EjOfertaId/value'
    responses:
      200:
        description: Petición procesada correctamente
        content:
          application/json:
```

```
            schema:
              $ref: '#/components/schemas/OfertaTrabajo'
            examples:
              oferta-front:
                $ref: '#/components/examples/EjOfertaTrabajoFront'
```

Ahora vamos a rellenar la petición POST que se encarga de crear ofertas de trabajo. En este caso necesitamos añadir a parte de la respuesta el cuerpo de la petición donde se mandan los datos.

Empezamos añadiendo los datos básico de información.

/swagger-openapi-lab/fjobs.yaml

```
/jobs:
  post:
    summary: Crea una oferta de trabajo
    description: Operación POST para crear una nueva oferta de trabajo en la BBDD. No es necesario estar autenticado para realizar la operación. Es necesario mandar junto a la petición los datos de la oferta a crear. Esta petición devuelve el identificador de la oferta de trabajo guardada.
```

Como hemos dicho, necesitamos enviar en el cuerpo de la petición la oferta a crear, y por tanto aquí podemos reutilizar el componente de OfertaTrabajo que ya habíamos definido, además de poder reutilizar también el ejemplo. Quedando así la operación:

/swagger-openapi-lab/fjobs.yaml

```
/jobs:
  post:
    summary: Crea una oferta de trabajo
    description: Operación POST para crear una nueva oferta de trabajo en la BBDD. No es necesario estar autenticado para realizar la operación. Es necesario mandar junto a la petición los datos de la oferta a crear. Esta petición devuelve el identificador de la oferta de trabajo guardada.
    requestBody:
      description: Datos de la oferta de trabajo que se va a guardar
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OfertaTrabajo'
          examples:
            oferta-a-guardar:
              $ref: '#/components/examples/EjOfertaTrabajoFront'
```

Para terminar con esta petición, necesitamos saber como va a ser la respuesta y como se ha comentado en la descripción del laboratorio, como respuesta obtendremos un objeto como el siguiente:

```
{
  "name": "-LOpEeXtROeJKi0bfoQZ"
}
```

Vamos a crear otro modelo y un ejemplo (que nos puede servir este de arriba) dentro del apartado de componentes:

/swagger-openapi-lab/fjobs.yaml

```
components:
  schemas:
    ResponseOfertaId:
      type: object
      properties:
        name:
          $ref: '#/components/schemas/OfertaId'
    OfertaId:
      type: string
    OfertaTrabajo:
      type: object
      properties:
        name:
          type: string
        description:
          type: string
        city:
          type: string
        salary:
          type: integer
        company:
          type: string
  examples:
    EjResponseOfertaId:
      summary: Respuesta del POST
      description: Ejemplo de una respuesta de la petición POST que trae el identificador de una oferta de trabajo
      value: {"name": "-LOpEeXtROeJKi0bfoQZ"}
    EjOfertaId:
      summary: Identificador Oferta
      description: Ejemplo de un identificador de una oferta de trabajo
      value: -LOpEeXtROeJKi0bfoQZ
    EjOfertaTrabajoFront:
      summary: Oferta Frontend
      description: Ejemplo de una oferta de trabajo de Frontend
      value: '{
        "name": "Front-end developer",
        "description": "As a front-end developer you will be joining the development team to work along with the product
and data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
        "city": "Madrid",
        "salary": 25000,
        "company": "E-Corp SL"
      }'
```

Una vez tenemos los componentes, ahora ya podemos añadir en la respuesta (en formato JSON) la referencia a este modelo de esquema y al ejemplo.

/swagger-openapi-lab/fjobs.yaml

```
/jobs:
  post:
    summary: Crea una oferta de trabajo
    description: Operación POST para crear una nueva oferta de trabajo en la BBDD. No es necesario estar autenticado para
realizar la operación. Es necesario mandar junto a la petición los datos de la oferta a crear. Esta petición devuelve el
identificador de la oferta de trabajo guardada.
    requestBody:
      description: Datos de la oferta de trabajo que se va a guardar
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OfertaTrabajo'
```

```
          examples:
            oferta-a-guardar:
              $ref: '#/components/examples/EjOfertaTrabajoFront'
    responses:
      200:
        description: Oferta de trabajo creada correctamente
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ResponseOfertaId'
            examples:
              respuesta:
                $ref: '#/components/examples/EjResponseOfertaId'
```

Por último, nos queda definir la petición GET que lista todas las ofertas de trabajo, y para ello, vamos a necesitar crear otro modelo y otro ejemplo.

Nuestra BBDD es un poco especial, y como hemos puesto al principio en la descripción del laboratorio, no obtenemos un array con todas las ofertas, sino que obtendremos un objeto con las ofertas dentro. El ejemplo que vamos a usar puede ser el que se muestra a continuación:

```
{
  "-LOpEeXtROeJKi0bfoQZ": {
    "name": "Front-end developer",
    "description": "As a front-end developer you will be joining the development team to work along with the product and
data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
    "city": "Madrid",
    "salary": 25000,
    "company": "E-Corp SL"
  },
  "-LNpPeXuW7eLZi2bfoDT": {
    "name": "Ruby on Rails developer",
    "description": "As a RoR developer you will be joining the development team to work along with the product and data
science teams.",
    "city": "Madrid",
    "salary": 45000,
    "company": "E-Corp SL"
  }
}
```

Una vez que tenemos el ejemplo y sabemos como va a ser la estructura, vamos a añadirlos en la sección de componentes:

Como las claves dentro del objeto de ofertas pueden ser valores dinámicos, vamos a utilizar la propiedad **additionalProperties** que indica que la clave puede ser cualquier valor, y le asignaremos como valor la referencia a la oferta de trabajo.

/swagger-openapi-lab/fjobs.yaml

```
components:
  schemas:
    ResponseOfertaId:
      type: object
      properties:
        name:
          $ref: '#/components/schemas/OfertaId'
    OfertaId:
```

```
      type: string
    OfertasTrabajo:
      type: object
      additionalProperties:
        $ref: '#/components/schemas/OfertaTrabajo'
    OfertaTrabajo:
      type: object
      properties:
        name:
          type: string
        description:
          type: string
        city:
          type: string
        salary:
          type: integer
        company:
          type: string
  examples:
    EjResponseOfertaId:
      summary: Respuesta del POST
      description: Ejemplo de una respuesta de la petición POST que trae el identificador de una oferta de trabajo
      value: {"name": "-LOpEeXtROeJKi0bfoQZ"}
    EjOfertaId:
      summary: Identificador Oferta
      description: Ejemplo de un identificador de una oferta de trabajo
      value: -LOpEeXtROeJKi0bfoQZ
    EjOfertaTrabajoFront:
      summary: Oferta Frontend
      description: Ejemplo de una oferta de trabajo de Frontend
      value: '{
        "name": "Front-end developer",
        "description": "As a front-end developer you will be joining the development team to work along with the product
and data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
        "city": "Madrid",
        "salary": 25000,
        "company": "E-Corp SL"
      }'
    EjOfertasTrabajo:
      summary: Ofertas de trabajo
      description: Ejemplo de las ofertas de trabajo guardadas en la BBDD
      value: '{
        "-LOpEeXtROeJKi0bfoQZ": {
          "name": "Front-end developer",
          "description": "As a front-end developer you will be joining the development team to work along with the
product and data science teams. All of our products use the React library which allows us to build scalable and complex
user interfaces in an efficient way.",
          "city": "Madrid",
          "salary": 25000,
          "company": "E-Corp SL"
        },
        "-LNpPeXuW7eLZi2bfoDT": {
          "name": "Ruby on Rails developer",
          "description": "As a RoR developer you will be joining the development team to work along with the product and
data science teams.",
          "city": "Madrid",
          "salary": 45000,
          "company": "E-Corp SL"
        }
      }'
```

El siguiente paso es rellenar la petición GET con los datos necesarios. En este caso, a parte de la información, necesitaremos añadir la respuesta (de código 200 en formato JSON) que apuntará al

esquema y ejemplo que acabamos de añadir en los componentes.

/swagger-openapi-lab/fjobs.yaml

```
/jobs:
  get:
    summary: Obtiene las ofertas de trabajo
    description: Operación GET para obtener todas las ofertas de trabajo que se encuentran en la BBDD. No es necesario
estar autenticado para realizar esta operación. Tampoco se necesita mandar ningún parámetro.
    responses:
      200:
        description: Petición procesada correctamente
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OfertasTrabajo'
            examples:
              ofertas:
                $ref: '#/components/examples/EjOfertasTrabajo'
```

Y con esto ya habríamos definido la primera versión de la API siguiendo las indicaciones que se han puesto en la descripción del laboratorio.

Finalmente el documento completo de OpenAPI quedaría así:

/swagger-openapi-lab/fjobs.yaml

```
openapi: '3.0.3'
info:
  title: FJobs API
  description: API donde vamos a gestionar ofertas de trabajo
  version: 1.0.0
  contact:
    name: Charly Falco
    email: cfalco@fjobs.com
    url: http://www.fjobs.com
servers:
  - url: https://fjobs.com/v1
    description: Servidor de producción
  - url: https://test.fjobs.com/v1
    description: Servidor de testing

paths:
  /jobs:
    get:
      summary: Obtiene las ofertas de trabajo
      description: Operación GET para obtener todas las ofertas de trabajo que se encuentran en la BBDD. No es necesario
estar autenticado para realizar esta operación. Tampoco se necesita mandar ningún parámetro.
      responses:
        200:
          description: Petición procesada correctamente
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/OfertasTrabajo'
              examples:
                ofertas:
                  $ref: '#/components/examples/EjOfertasTrabajo'
    post:
      summary: Crea una oferta de trabajo
      description: Operación POST para crear una nueva oferta de trabajo en la BBDD. No es necesario estar autenticado
para realizar la operación. Es necesario mandar junto a la petición los datos de la oferta a crear. Esta petición
```

```
devuelve el identificador de la oferta de trabajo guardada.
      requestBody:
        description: Datos de la oferta de trabajo que se va a guardar
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OfertaTrabajo'
            examples:
              oferta-a-guardar:
                $ref: '#/components/examples/EjOfertaTrabajoFront'
      responses:
        200:
          description: Oferta de trabajo creada correctamente
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ResponseOfertaId'
              examples:
                respuesta:
                  $ref: '#/components/examples/EjResponseOfertaId'
  /jobs/{jobId}:
    get:
      summary: Obtiene una oferta de trabajo
      description: Operación GET para obtener una oferta de trabajo dado su identificador. No es necesario estar
autenticado. Hay que mandar el identificador de la oferta pedida como parámetro.
      parameters:
        - in: path
          name: jobId
          description: Identificador de la oferta de trabajo pedida
          required: true
          schema:
            $ref: '#/components/schemas/OfertaId'
          example:
            $ref: '#/components/examples/EjOfertaId/value'
      responses:
        200:
          description: Petición procesada correctamente
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/OfertaTrabajo'
              examples:
                oferta-front:
                  $ref: '#/components/examples/EjOfertaTrabajoFront'
components:
  schemas:
    ResponseOfertaId:
      type: object
      properties:
        name:
          $ref: '#/components/schemas/OfertaId'
    OfertaId:
      type: string
    OfertasTrabajo:
      type: object
      additionalProperties:
        $ref: '#/components/schemas/OfertaTrabajo'
    OfertaTrabajo:
      type: object
      properties:
        name:
          type: string
        description:
          type: string
        city:
          type: string
```

```
        salary:
          type: integer
        company:
          type: string
  examples:
    EjResponseOfertaId:
      summary: Respuesta del POST
      description: Ejemplo de una respuesta de la petición POST que trae el identificador de una oferta de trabajo
      value: {"name": "-LOpEeXtROeJKi0bfoQZ"}
    EjOfertaId:
      summary: Identificador Oferta
      description: Ejemplo de un identificador de una oferta de trabajo
      value: -LOpEeXtROeJKi0bfoQZ
    EjOfertaTrabajoFront:
      summary: Oferta Frontend
      description: Ejemplo de una oferta de trabajo de Frontend
      value: '{
        "name": "Front-end developer",
        "description": "As a front-end developer you will be joining the development team to work along with the product
and data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
        "city": "Madrid",
        "salary": 25000,
        "company": "E-Corp SL"
      }'
    EjOfertasTrabajo:
      summary: Ofertas de trabajo
      description: Ejemplo de las ofertas de trabajo guardadas en la BBDD
      value: '{
        "-LOpEeXtROeJKi0bfoQZ": {
          "name": "Front-end developer",
          "description": "As a front-end developer you will be joining the development team to work along with the
product and data science teams. All of our products use the React library which allows us to build scalable and complex
user interfaces in an efficient way.",
          "city": "Madrid",
          "salary": 25000,
          "company": "E-Corp SL"
        },
        "-LNpPeXuW7eLZi2bfoDT": {
          "name": "Ruby on Rails developer",
          "description": "As a RoR developer you will be joining the development team to work along with the product and
data science teams.",
          "city": "Madrid",
          "salary": 45000,
          "company": "E-Corp SL"
        }
      }'
```

Chapter 4. Swagger UI

Es una interfaz en la que se muestra la documentación de nuestra API, generada a partir del documento de OpenAPI que hemos creado.

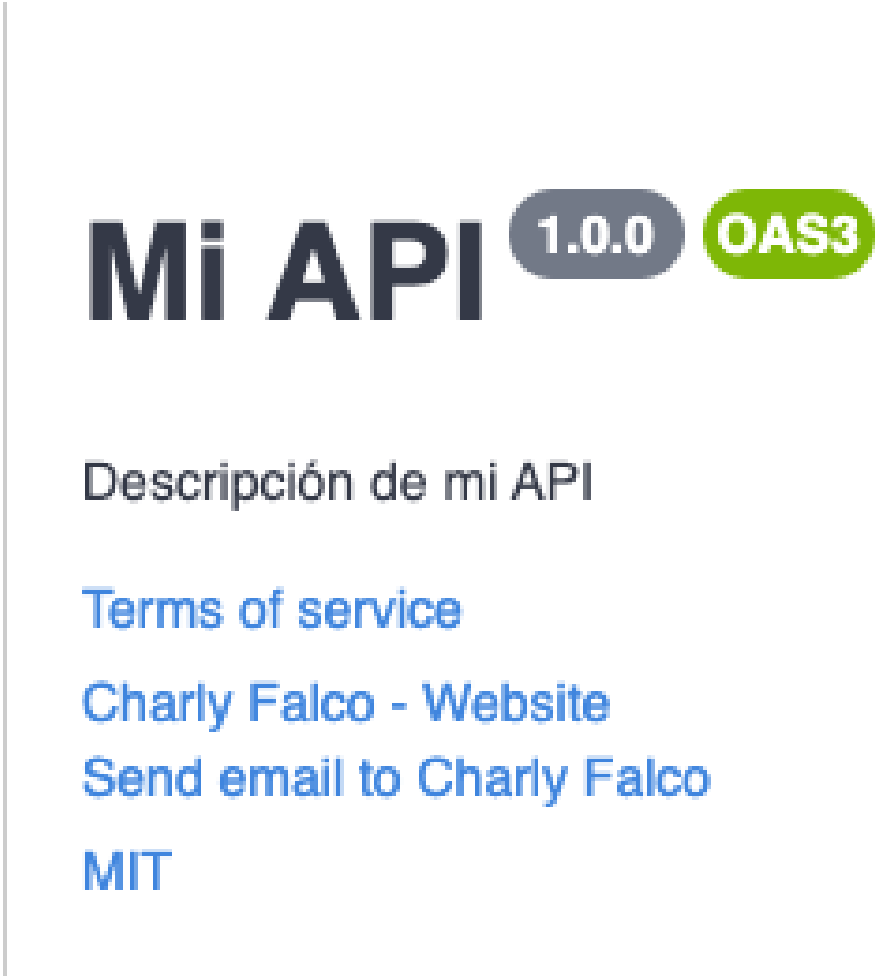
Con esta herramienta conseguimos que la documentación se encuentre organizada y podamos navegar por ella de una forma sencilla, de tal forma que será mucho más sencillo leerla desde esta herramienta que leer el documento de OpenAPI.

Esta herramienta nos la encontramos junto al Swagger Editor, pues es la pantalla de la derecha donde se va mostrando la documentación generada.

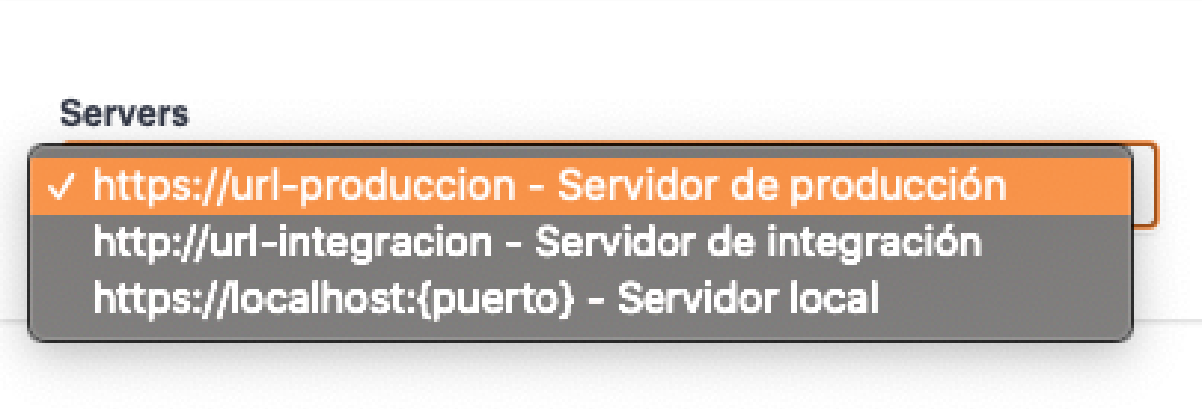
4.1. Organización

Como hemos dicho antes, esta herramienta se encarga de organizar la documentación, por tanto todo aquello que hemos puesto en el documento lo va a ir mostrando en un orden determinado.

Al principio del documento vamos a ver aquella información de la API que se ha puesto en **info**, es decir, el nombre de la API, la versión, la información de contacto…



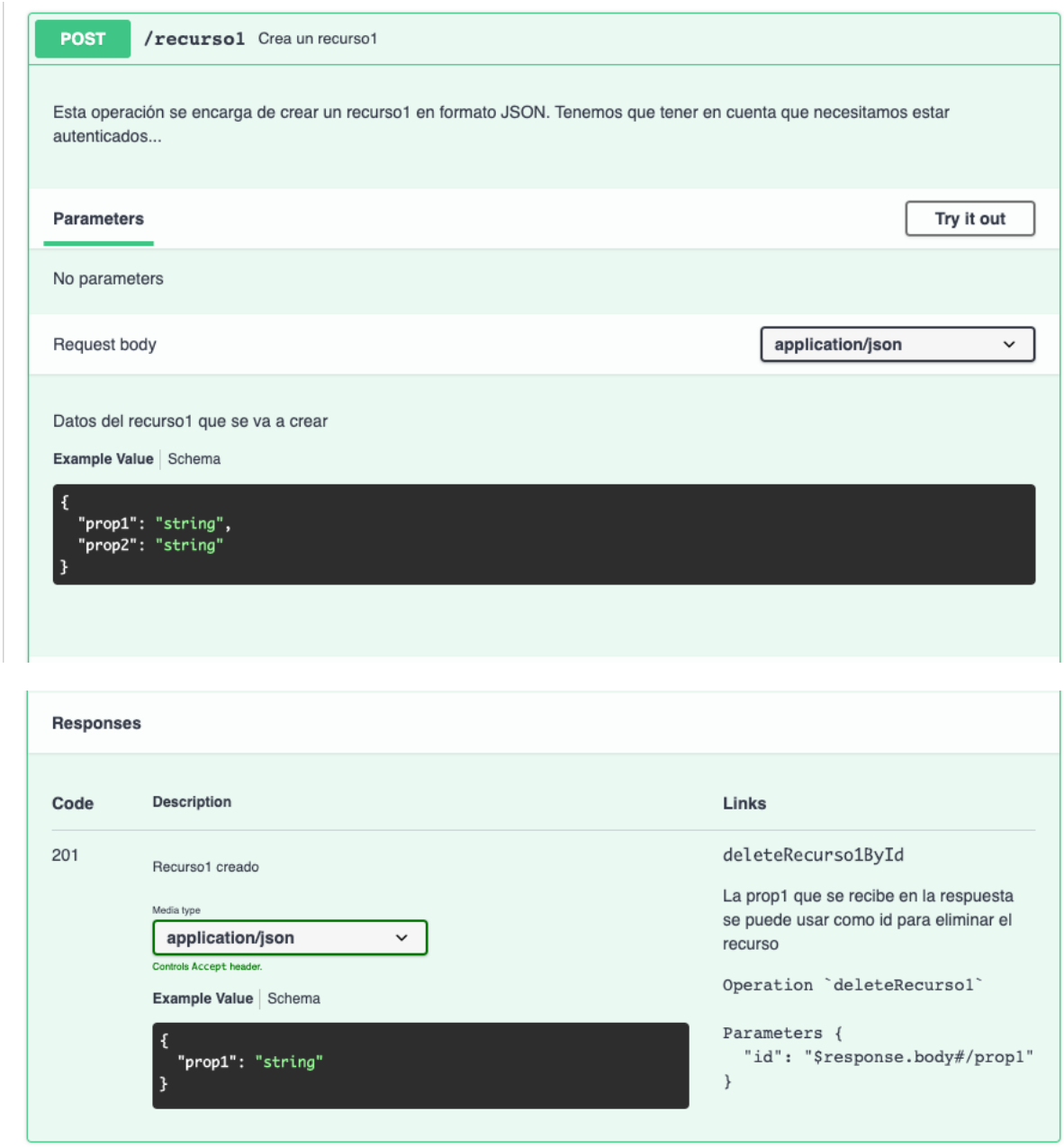
Justo después se muestra la información relativa a los servidores, donde podremos seleccionar uno de ellos para ver como quedarían las peticiones al servidor seleccionado.



Después de los servidores, nos encontramos con todas las peticiones que hemos definido en la API organizadas por las etiquetas (**tags**) que les hayamos puesto.

Y al final del todo los componentes que hemos definido para reutilizar en las distintas operaciones.

Por cada una de las operaciones, se muestra al principio su información, seguida de los parámetros y el cuerpo de las peticiones, y finalmente las distintas respuestas.


POST
/recurso1
Crea un recurso1
Esta operación se encarga de crear un recurso1 en formato JSON. Tenemos que tener en cuenta que necesitamos estar autenticados...
Parameters
Try it out
No parameters
Request body
application/json
Datos del recurso1 que se va a crear
Example Value
Schema
{
"prop1": "string",
"prop2": "string"
}
Responses
Code
Description
Links
201
Recurso1 creado
Media type
application/json
Controls Accept header.
Example Value
Schema
{
"prop1": "string"
}
deleteRecurso1ById
La prop1 que se recibe en la respuesta se puede usar como id para eliminar el recurso
Operation `deleteRecurso1`
Parameters {
"id": "$response.body#/prop1"
}

Chapter 5. SwaggerHub

SwaggerHub es una herramienta en la nube que incluye a todas las que componen Swagger, es decir, Swagger Editor, Swagger UI… pero además añade:

- La capacidad de **colaboración** entre distintas personas en la misma API.
- **Reutilizar** recursos de una forma sencilla entre APIs.
- **Publicar la documentación**.
- Añadir **roles** a las personas que colaboran con las APIs.
- Añadir **reglas de estandarización** para que todas las APIs tengan el mismo diseño.
- …

5.1. Crear cuenta de SwaggerHub

Para empezar a usar las funcionalidades de SwaggerHub necesitamos crearnos una cuenta.

Empezamos por ir a https://app.swaggerhub.com/signup y rellenar los datos que nos piden:

- Username
- Email
- Password

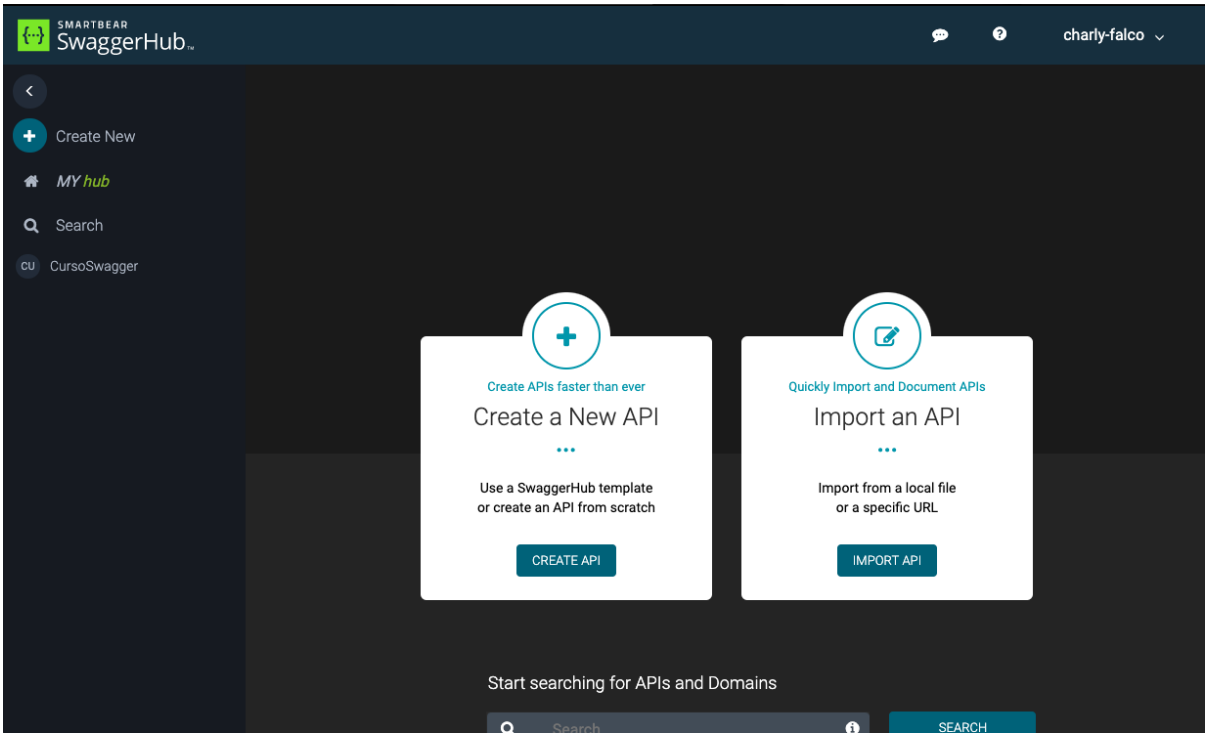
O podemos utilizar nuestra cuenta de Github.

Al darle al botón de **Sign Up** vamos a la pantalla de crear una organización donde nos piden:

- Nombre de la organización
- Teléfono

En la siguiente pantalla podemos invitar a más miembros a la organización que acabamos de crear. Solo tenemos que poner los emails de estas personas.

Después de invitar o no a los miembros de la empresa, llegamos a la pantalla inicial de SwaggerHub.



5.2. Importar API

SwaggerHub nos permite importar una API a partir de un enlace o un archivo.

Desde la pantalla inicial de SwaggerHub, hay que darle a **Create New** > **Import and document API** que se encuentra en el menú lateral.

Ahora se abre un modal en el que tenemos que rellenar:

- Path or URL
- Owner
- Project
- Visibility

Una vez rellenados estos campos le damos a **Upload File** donde nos pide que rellenemos los siguientes datos:

- Nombre de la API
- Versión inicial

Le damos a **Import API** y se abre el editor con el documento que hemos cargado.

5.3. Comentar APIs

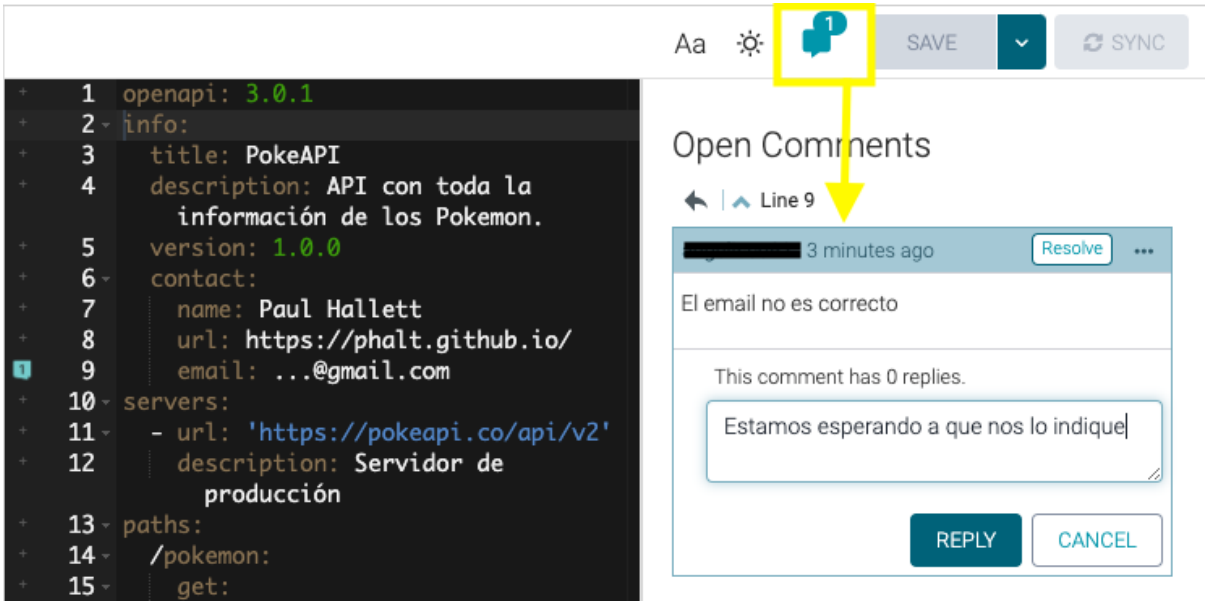
Los miembros que pueden entrar a las APIs pueden haber obtenido permisos para poner comentarios en estas de tal forma que entre todos los participantes en el proceso de definición de la API puedan proponer mejoras, detectar fallos antes de tiempo...

Para poner comentarios, lo único que necesitamos es entrar a la pantalla de edición/visualización

de la API a la que se accede pulsando sobre el nombre de esta y pulsar sobre los botones + que se encuentran al lado de la columna con los números de filas.

Una vez puesto un comentario, se puede contestar a la persona desde el mismo comentario, y una vez resuelto el problema, se puede marcar como resuelto.

Todo esto se puede hacer pulsando sobre el icono de mensajes que aparece en la barra superior.

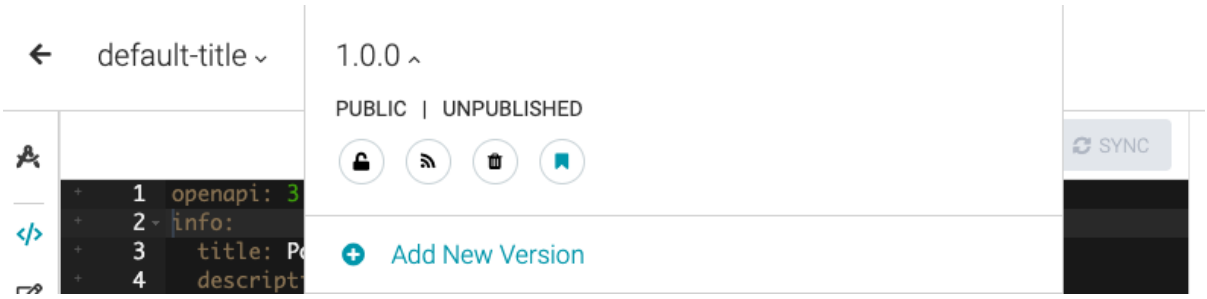


5.4. Gestionar versiones de la API

Una vez que tenemos la API publicada es probable que vayamos a seguir mejorandola añadiendo nuevas operaciones, modificando y mejorando las que ya existen por ejemplo añadiendo nuevos parámetros o marcando los existentes como obligatorios...

Todas estas modificaciones deberíamos de hacerlas en una nueva versión de la API para que aquellos consumidores de nuestra API no tengan que cambiar sus desarrollos, sino que puedan seguir usando la versión antigua hasta que se decida deprecarla, momento en el que deberían de estar adaptándose a la nueva versión, y eliminarla, aquí ya no podrán usarla, así que mejor que hayan adaptado su desarrollo a la nueva versión.

En SwaggerHub podemos llevar la gestión de las versiones de la API desde el editor. Una vez dentro, se puede pulsar sobre el número de versión que aparece en la barra superior y pulsar sobre el botón de **Add new version** donde pondremos la nueva versión.



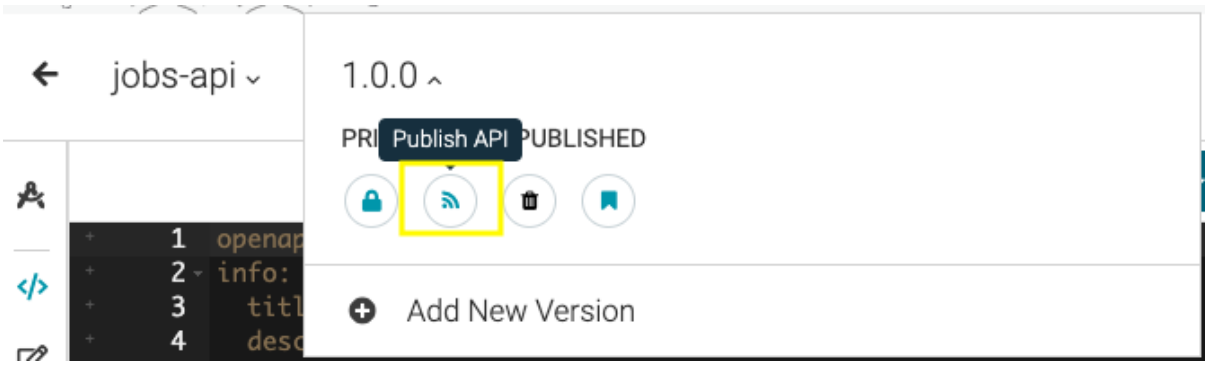
5.5. Publicar la API

SwaggerHub nos permite coger las APIs que tenemos definidas y publicarlas o quitar la publicación.

Esta opción nos permite marcar una API como que ya está bien definida, se ha revisado y se ha llegado a la conclusión de que está lista para producción, de tal forma que al publicarse nadie podrá realizar modificaciones sobre ella.

Cualquier modificación que necesitemos realizar sobre esta API tendrá que hacerse en una nueva versión.

Para publicar una API, tenemos que entrar al editor de esta, y una vez dentro, solo tenemos que pulsar sobre el desplegable de las versiones y pulsar sobre el icono de RSS.



Una vez publicada ya no se puede modificar el archivo con su definición. En el caso de que necesitemos realizar algún cambio, podríamos quitar la publicación pulsando sobre el mismo botón.

Publicar o quitar la publicación de una API solo lo pueden realizar los propietarios de esta, es decir, aquellos usuarios que tengan el rol de **Owner**.

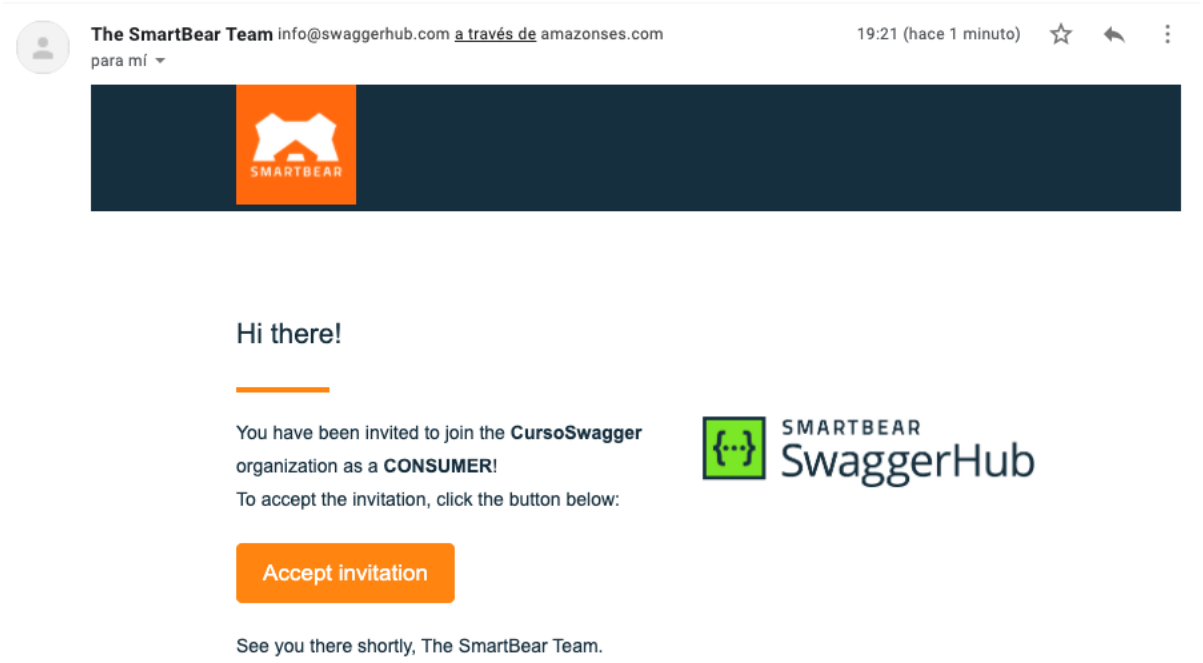
5.6. Roles

Durante la definición de la API pueden participar distintas personas, algunas que se encargan de desarrollar la definición, otras que gestionan el equipo que está trabajando en la API, otras que solo pueden consultarla…

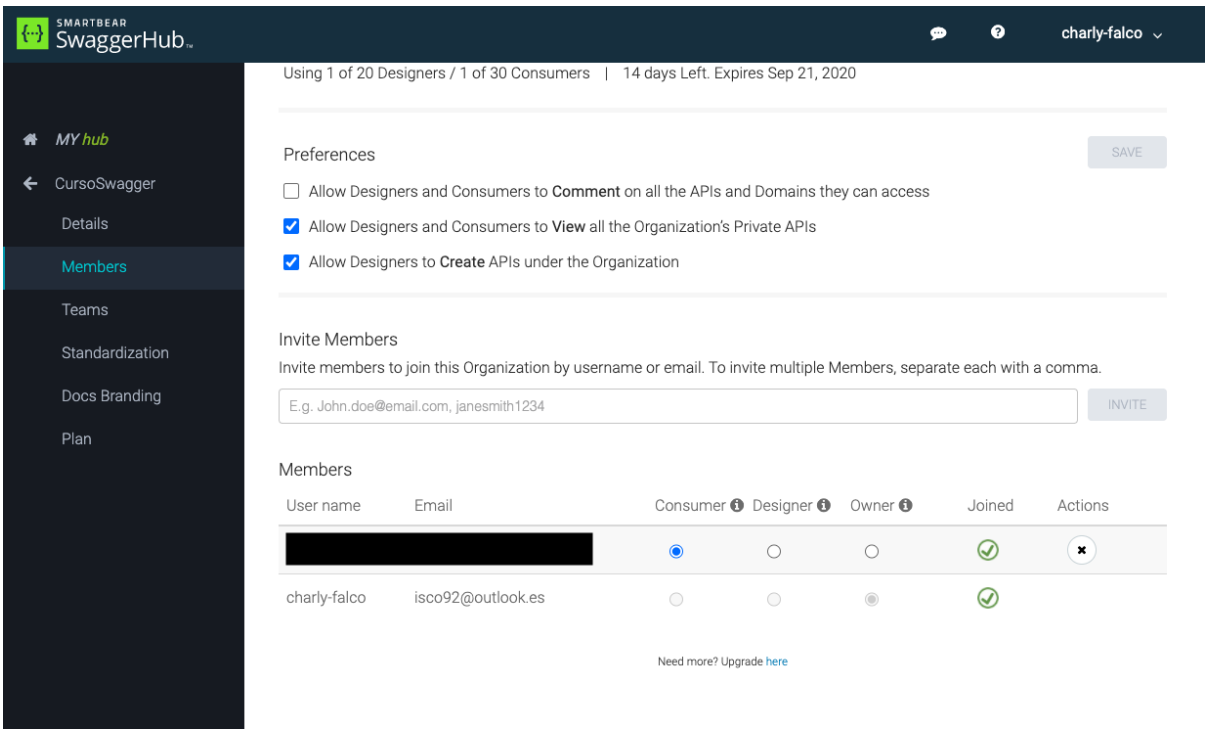
Para evitar que aquellas personas que no deben, modifiquen la API sin querer o queriendo, el propietario de la API puede darles permisos a todos los participantes.

Para añadir a una persona a la organización y darle permisos, primero hay que ir a **Menú lateral** > **Organización** > **Orgarnization Settings** > **Members** y una vez dentro, poner el email o el nombre de usuario en el campo de **Invite Members**.

Al darle a invitar, le llegará un email al invitado donde puede aceptar unirse a la organización.



Cuando ocurre esto, se marca el checkbox de **Joined** que aparece en el panel con todos los miembros.



En este panel es donde podremos darle el rol adecuado a esta persona. Los roles disponibles son:

- **Consumer**: puede ver la documentación de las APIs.
- **Designer**: puede ver, crear y colaborar en el diseño de las APIs.
- **Owner**: es el propietario de la API, y puede verlas, crearlas, editarlas y añadir a más miembros a la organización.

5.7. Crear dominios

Los **dominios** de SwaggerHub nos permiten separar aquellos componentes que hemos definido y se pueden reutilizar entre las distintas APIs de la organización, de tal forma que no haya que mantenerlas en distintos sitios, sino que solo habría que mantenerlas en el dominio.

Los dominios contienen básicamente componentes.

Para crear un nuevo dominio, hay que pulsar sobre **Create New** > **Create New Domain**. Una vez pulsada sobre esta opción se abre un modal con los siguientes campos que tenemos que rellenar:

- **OpenAPI Version**: versión del documento OpenAPI.
- **Name**: nombre del dominio.
- **Version**: versión del dominio.
- **Owner**: el propietario del dominio.
- **Project**: el proyecto donde se va a guardar.
- **Visibility**: visibilidad del dominio.

Al rellenar estos datos hay que pulsar sobre **Create domain** y se abre el editor de swagger donde rellenaremos con aquellas partes de la definición de las APIs que se pueden compartir entre distintas APIs de nuestra organización.

Una vez completado este documento, hay que guardarlo y al ir al listado de APIs se pueden ver también los dominios creados.

Como se han extraído aquellos componentes que se pueden reutilizar entre distintas APIs de estas, entonces hay que cambiar las referencias a estos componentes.

Para realizar estos cambios, lo más fácil es eliminar el valor de los **$ref**, y utilizar el autocompletado para que nos lo complete con la opción que queramos, que en este caso será la que se ha definido en el dominio.

Estos valores serán algo parecido a:

```
$ref: 'https://api.swaggerhub.com/domains/nombre-de-la-organizacion/nombre-del-dominio/version-del-
dominio#/components/schemas/nombre-del-modelo'
```

5.8. Estandarización de las APIs

En **SwaggerHub** podemos aplicar ciertas reglas predefinidas o personalizadas para hacer que todas las APIs tengan el mismo diseño a la hora de definir como van a ser las operaciones.

Para aplicar las reglas de estandarización a nuestras APIs tenemos que ir a **Menú lateral** > **Organización** > **Orgarnization Settings** > **Standardization** donde nos aparecen todas las reglas predefinidas que podemos aplicar.

En esta página nos encontramos las reglas separadas en secciones, donde cada una de ellas se corresponde con alguna sección dentro del documento OpenAPI:

- **Options**:
  - La API tiene que pasar las reglas de estandarización para poder publicarse.
- **API Info**
  - La licencia de la API tiene que estar presente y no puede ser un string vacío.
  - La descripción de la API tiene que estar presente y no puede ser un string vacío.
  - La información de contacto de la API tiene que estar presente y no puede ser un string vacío.
  - El título de la API tiene que estar presente y no puede ser un string vacío.
- **Operations**
  - El summary debe empezar en mayúscula y terminar con un punto.
  - La operación debe de tener una respuesta por defecto.
  - El campo operationId debe estar presente y no puede ser un string vacío.
  - El campo description debe estar presente y no puede ser un string vacío.
  - Las operaciones deben de tener un tag y no puede ser un string vacío.
  - El campo summary debe estar presente y no puede ser un string vacío.
  - Las operaciones deben de tener solo un tag.
  - Las operaciones deben de tener al menos una respuesta con código 2xx.
- **Models**
  - Todos los modelos deben de tener ejemplos.
  - Las APIs no deben de tener definiciones locales (solo se permiten $ref).

Y al final de las secciones con reglas predefinidas, tenemos un botón que nos permite crear nuestras propias reglas.

En el modal que se abre al pulsar el botón **Add custom rule**, tendremos que configurar nuestras reglas rellenando los campos:

- **Name**: nombre que le damos a la regla.
- **OpenAPI Version**: a que versión de OpenAPI queremos permitir que se aplique.

- **Path Required**: el JSONPath al elemento objetivo.
  - **Path Exists**: checkbox para indicar que la ruta tiene que existir.
- **Regex**: expresión regular que se comprueba con el campo anterior.
  - **Must Match**: checkbox que indica que el campo tiene que cumplir con la expresión regular.
- **Rule Severity**: tipo de severidad del error. Puede ser un **warning** o un **error**.
- **Error Message**: error a mostrar cuando no se cumpla la regla.
- **Try It Out**: aquí podemos importar una de nuestras APIs para comprobar si pasa la regla.

5.8.1. Lab: Estandarización de las APIs

En este laboratorio vamos a crear una regla que compruebe que nuestro documento de OpenAPI está usando la versión '3.0.3'.

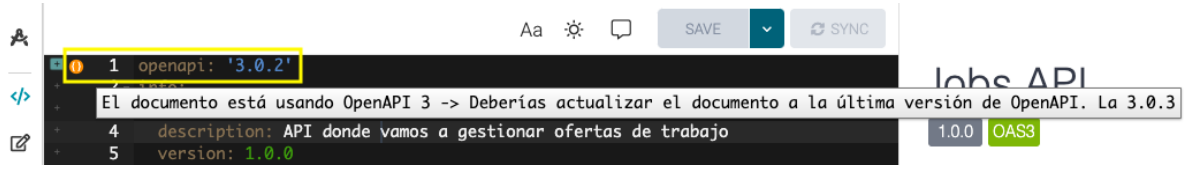
Lo primero que hacemos es ir hasta la pantalla de **Menú lateral** > **Organización** > **Orgarnization Settings** > **Standardization**, y una vez allí pulsamos sobre el botón de **Add custom rule**.

En el modal que se abre vamos a rellenar los campos con los siguientes datos:

- **Name**: El documento está usando OpenAPI 3
- **OpenAPI Version**: All
- **Path Required**: $.openapi
  - **Path Exists**: false
- **Regex**: 3\.0\.3
  - **Must Match**: true
- **Rule Severity**: warning
- **Error Message**: Deberías actualizar el documento a la última versión de OpenAPI. La 3.0.3

Y una vez que tenemos los campos rellenos, la guardamos y pulsamos sobre el botón de **Save** que aparece arriba a la derecha.

Para probarla con nuestras APIs, podemos dejar el checkbox de esta regla marcado e ir a cualquiera de ellas y comprobar que si ponemos la propiedad **openapi: 3.0.2** nos sale al lado un warning con el mensaje que hemos puesto al configurar la regla.



Chapter 6. Swagger Inspector

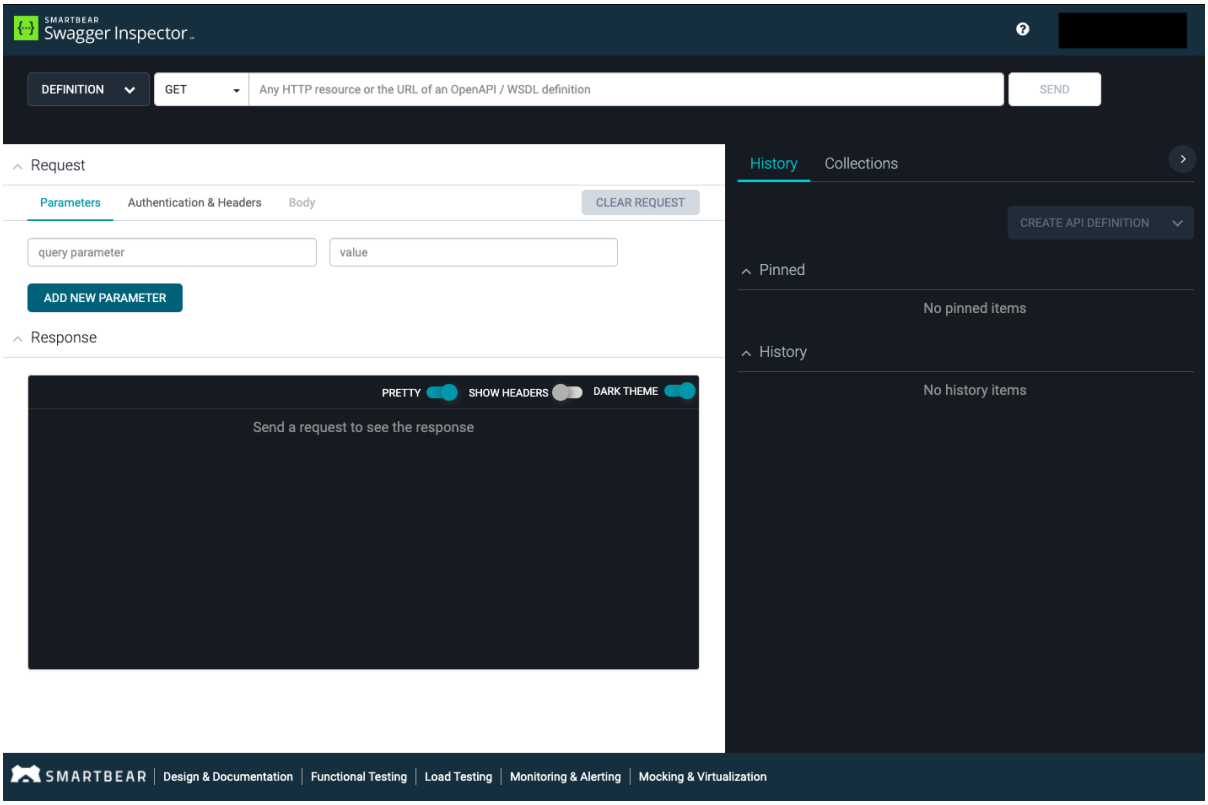
Swagger Inspector es una herramienta que lee los documentos de OpenAPI y nos permite testear de una forma rápida las operaciones que se han definido en estos documentos.

Podemos encontrar la versión gratuita en https://inspector.swagger.io/ que nos permite usar las funcionalidades más básicas, que serían:

- Leer los documentos OpenAPI.
- Realizar peticiones para probar las APIs.

Si nos logueamos con nuestra cuenta de Swagger en esta página, podremos utilizar funcionalidades extra como:

- Crear colecciones de peticiones, de tal forma que cuando tengamos muchas operaciones nos resulte fácil localizar grupos de peticiones que se relacionan entre ellas.
- Crear el documento OpenAPI a partir de las peticiones realizadas.



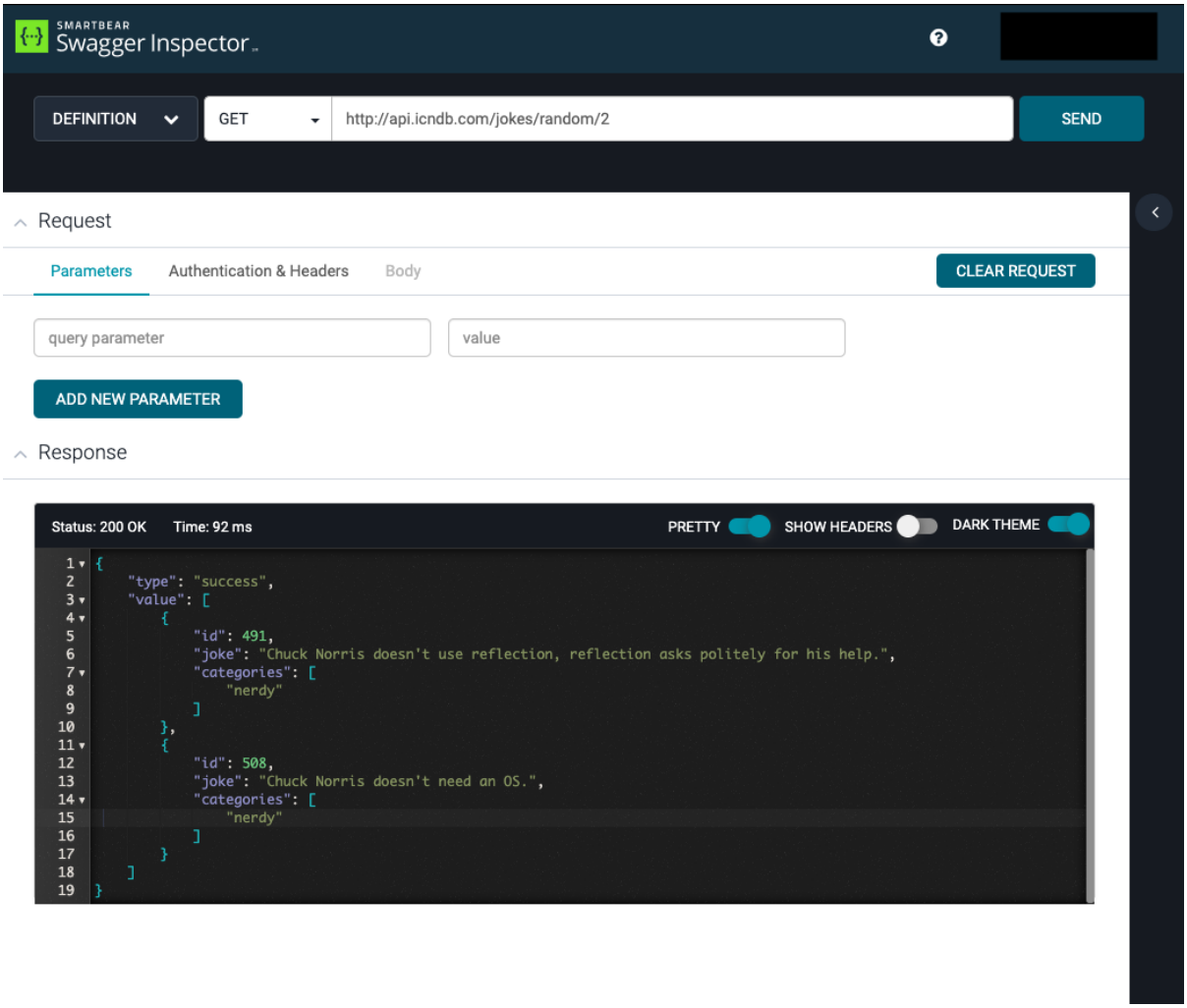
6.1. Lab: Realizar petición en Inspector

En este laboratorio vamos a ver como realizar una petición HTTP a la API de http://www.icndb.com/api/ desde **Swagger Inspector**.

Para empezar, necesitamos saber a que endpoint vamos a realizar la petición. Por ejemplo queremos obtener dos frases al azar sobre Chuck Norris, por tanto la petición sera:

- GET http://api.icndb.com/jokes/random/2

Si pulsamos sobre el botón **Send**, podremos ver debajo la respuesta a la petición.



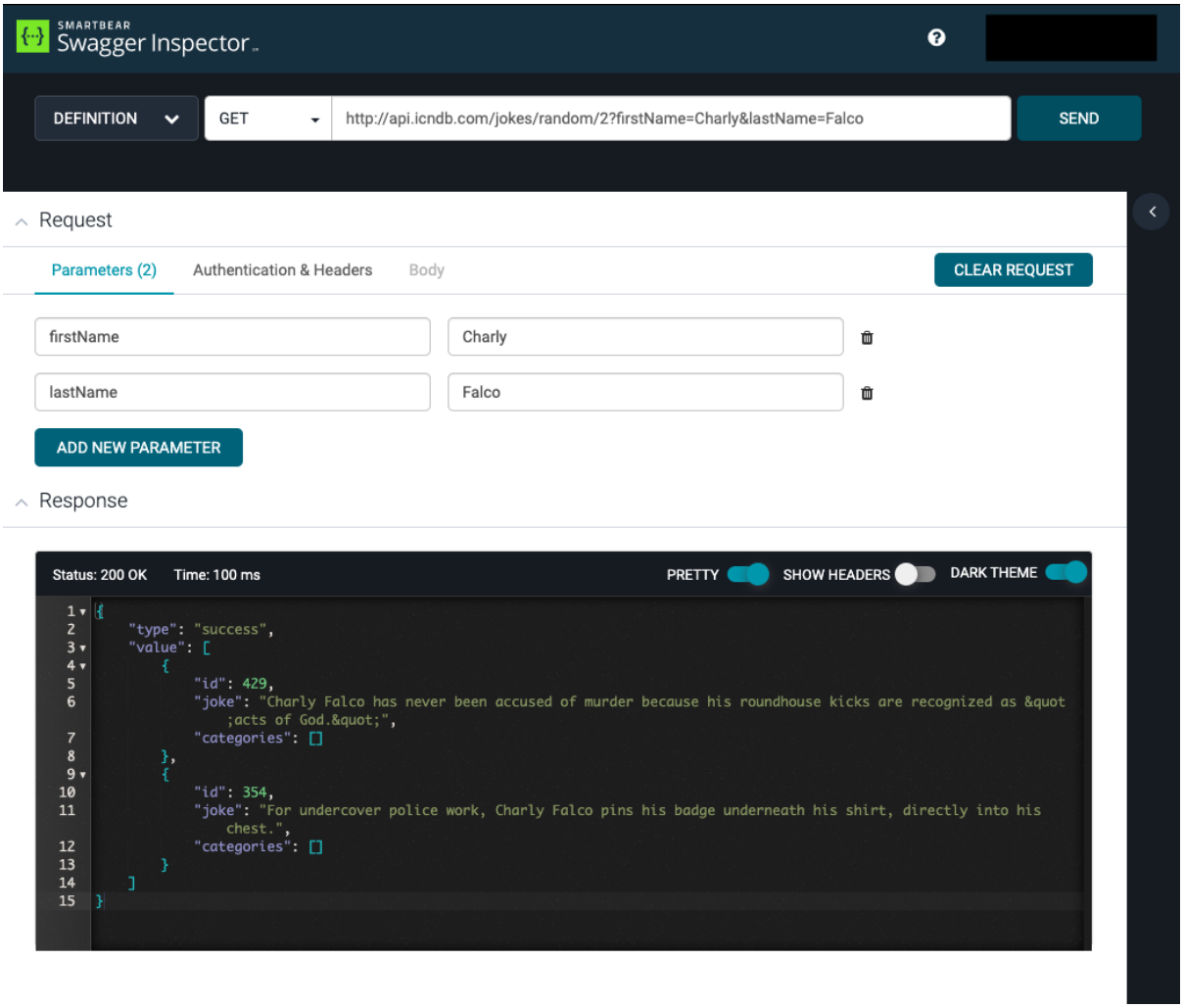
Además, ahora queremos que en estas frases sustituya el nombre y apellido por los nuestros. Si miramos la documentación de la API, veremos que hay que añadir dos parámetros de consulta:

- firstName
- lastName

Por tanto, en la herramienta, tenemos que ir a la pestaña de **Parameters** y añadir estos dos parámetros con sus valores.

- firstName: Charly
- lastName: Falco

Volvemos a pulsar sobre el botón de **Send** y obtenemos de nuevo la respuesta.



6.2. Lab: Crear colección en Inspector

En este laboratorio vamos a ver como crear una colección con dos operaciones de una API, que se define a continuación, para tenerlas agrupadas y que nos resulten más fácil de encontrar.

Vamos a agrupar las operaciones que se definen en el siguiente documento de OpenAPI:

```
openapi: '3.0.3'
info:
  title: Jobs API
  description: API donde vamos a gestionar ofertas de trabajo
  version: 1.0.0
  contact:
    name: Charly Falco
    email: cfalco@fjobs.com
    url: http://www.fjobs.com
servers:
  - url: https://fluentjobs-fa22e.firebaseio.com/v1
    description: Servidor de producción

paths:
  /jobs:
    get:
      summary: Obtiene las ofertas de trabajo
      description: Operación GET para obtener todas las ofertas de trabajo que se encuentran en la BBDD. No es necesario
estar autenticado para realizar esta operación. Tampoco se necesita mandar ningún parámetro.
      responses:
        200:
```

```
          description: Petición procesada correctamente
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/OfertasTrabajo'
              examples:
                ofertas:
                  $ref: '#/components/examples/EjOfertasTrabajo'
    post:
      summary: Crea una oferta de trabajo
      description: Operación POST para crear una nueva oferta de trabajo en la BBDD. No es necesario estar autenticado
para realizar la operación. Es necesario mandar junto a la petición los datos de la oferta a crear. Esta petición
devuelve el identificador de la oferta de trabajo guardada.
      requestBody:
        description: Datos de la oferta de trabajo que se va a guardar
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/OfertaTrabajo'
            examples:
              oferta-a-guardar:
                $ref: '#/components/examples/EjOfertaTrabajoFront'
      responses:
        201:
          description: Oferta de trabajo creada correctamente
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ResponseOfertaId'
              examples:
                respuesta:
                  $ref: '#/components/examples/EjResponseOfertaId'
  /jobs/{jobId}:
    get:
      summary: Obtiene una oferta de trabajo
      description: Operación GET para obtener una oferta de trabajo dado su identificador. No es necesario estar
autenticado. Hay que mandar el identificador de la oferta pedida como parámetro.
      parameters:
        - in: path
          name: jobId
          description: Identificador de la oferta de trabajo pedida
          required: true
          schema:
            $ref: '#/components/schemas/OfertaId'
          example:
            $ref: '#/components/examples/EjOfertaId/value'
      responses:
        200:
          description: Petición procesada correctamente
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/OfertaTrabajo'
              examples:
                oferta-front:
                  $ref: '#/components/examples/EjOfertaTrabajoFront'
components:
  schemas:
    ResponseOfertaId:
      type: object
      properties:
        name:
          $ref: '#/components/schemas/OfertaId'
    OfertaId:
      type: string
    OfertasTrabajo:
```

```
      type: object
      additionalProperties:
          $ref: '#/components/schemas/OfertaTrabajo'
    OfertaTrabajo:
      type: object
      properties:
        name:
          type: string
        description:
          type: string
        city:
          type: string
        salary:
          type: integer
        company:
          type: string
  examples:
    EjResponseOfertaId:
      summary: Respuesta del POST
      description: Ejemplo de una respuesta de la petición POST que trae el identificador de una oferta de trabajo
      value: {name: -LOpEeXtROeJKi0bfoQZ}
    EjOfertaId:
      summary: Identificador Oferta
      description: Ejemplo de un identificador de una oferta de trabajo
      value: -LOpEeXtROeJKi0bfoQZ
    EjOfertaTrabajoFront:
      summary: Oferta Frontend
      description: Ejemplo de una oferta de trabajo de Frontend
      value: '{
        "name": "Front-end developer",
        "description": "As a front-end developer you will be joining the development team to work along with the product
and data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
        "city": "Madrid",
        "salary": 25000,
        "company": "E-Corp SL"
      }'
    EjOfertasTrabajo:
      summary: Ofertas de trabajo
      description: Ejemplo de las ofertas de trabajo guardadas en la BBDD
      value: '{
        "-LOpEeXtROeJKi0bfoQZ": {
          "name": "Front-end developer",
          "description": "As a front-end developer you will be joining the development team to work along with the
product and data science teams. All of our products use the React library which allows us to build scalable and complex
user interfaces in an efficient way.",
          "city": "Madrid",
          "salary": 25000,
          "company": "E-Corp SL"
        },
        "-LNpPeXuW7eLZi2bfoDT": {
          "name": "Ruby on Rails developer",
          "description": "As a RoR developer you will be joining the development team to work along with the product and
data science teams.",
          "city": "Madrid",
          "salary": 45000,
          "company": "E-Corp SL"
        }
      }'
```

Empezamos por realizar la operación POST que se ha definido, para lo que vamos a necesitar poner los siguientes datos en la barra de peticiones:

- Método: POST

- URL: https://fluentjobs-fa22e.firebaseio.com/v1/jobs.json
- Body:

```
{
    "name": "Front-end developer",
    "description": "As a front-end developer you will be joining the development team to work along with the product and
data science teams. All of our products use the React library which allows us to build scalable and complex user
interfaces in an efficient way.",
    "city": "Madrid",
    "salary": 25000,
    "company": "E-Corp SL"
}
```

Ahora pulsamos sobre el botón de **Send** y comprobamos que nos devuelve el identificador de la oferta que se ha guardado.

El siguiente paso es realizar la petición GET que pide la oferta de trabajo que acabamos de crear, para lo que rellenamos la barra de peticiones con los datos siguientes:

- Método: GET
- URL: https://fluentjobs-fa22e.firebaseio.com/v1/jobs/-TGcYs3IEa8R6z0NrvC9.json

Hay que cambiar en la URL el {jobId} con un identificador válido, que puede ser el que hemos obtenido en la petición anterior.

Una vez tenemos la petición lista para lanzar, pulsamos el botón de **Send** y se nos debería de mostrar en la respuesta los datos de la oferta de trabajo que hemos pedido.

La última petición que vamos a lanzar es la que obtiene la lista de ofertas de trabajo que hay guardadas, para lo que vamos a rellenar la barra de peticiones con:

- Método: GET
- URL: https://fluentjobs-fa22e.firebaseio.com/v1/jobs.json

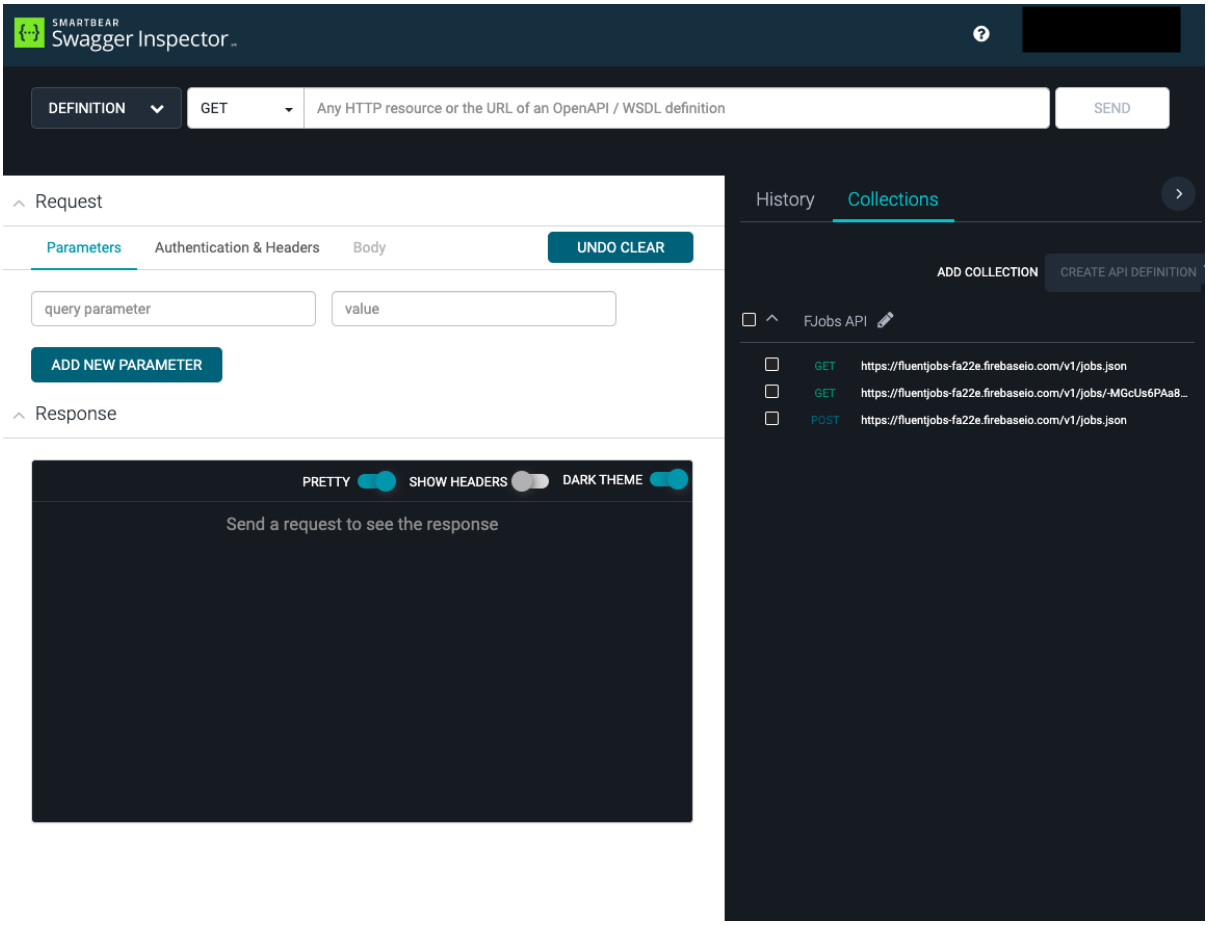
Lanzamos la petición, y deberíamos de ver todas las ofertas que hay guardadas en ese momento en la BBDD.

Una vez que hemos lanzado las peticiones anteriores, las podemos ver en la sección del historial (**History**) que aparece en el panel de la derecha.

Para crear una colección con estas peticiones, tenemos que seleccionar aquellos checkboxes que se encuentran junto a las peticiones a guardar, y una vez seleccionados pulsamos sobre el botón **Add to collection**.

En el modal que se abre pulsamos sobre el botón de **Create new collection** donde pondremos el nombre de la colección (**FJobs API**) y pulsaremos el botón de **Add to collection** para crearla con las peticiones seleccionadas.

Ahora podemos comprobar que se ha generado la colección con las 3 peticiones si vamos a la pestaña de **Collections**.



Una forma sencilla de haber realizado las peticiones sin tener que configurarlas nosotros a mano, hubiera sido cargando el documento de OpenAPI y desde ahí seleccionando las peticiones una a una.

6.3. Lab: Generar documento OpenAPI en Inspector

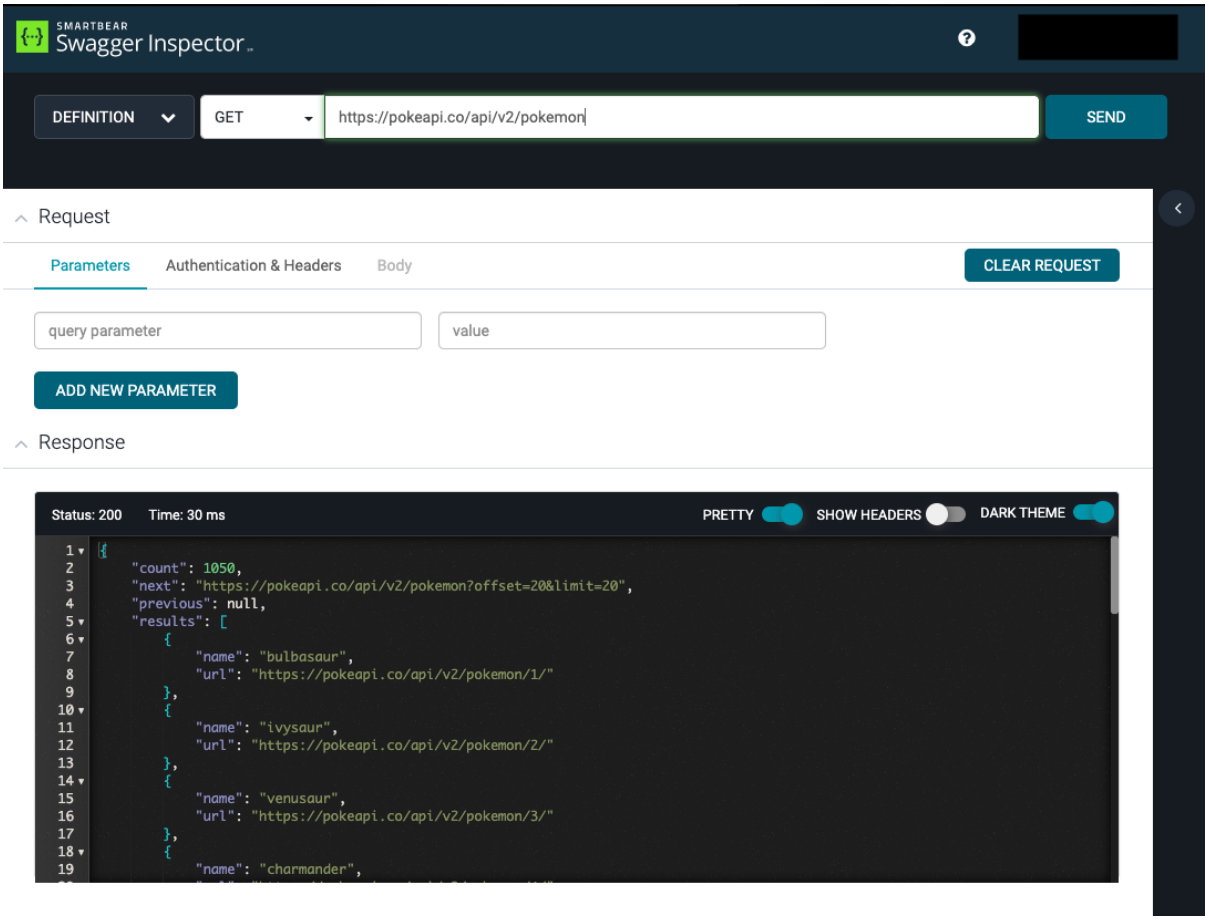
En este laboratorio vamos a ver como generar un documento de OpenAPI de la API de Pokemon (https://pokeapi.co/) a partir de varias peticiones que vamos a realizar con el Inspector.

Para poder realizar este laboratorio, es necesario tener una cuenta de Swagger ya que esta funcionalidad es una funcionalidad de la versión PRO.

Lo primero que tenemos que hacer es loguearnos en Swagger Inspector, y una vez hecho esto, vamos a ir a la API para coger 3 endpoints:

- https://pokeapi.co/api/v2/pokemon/ditto
- https://pokeapi.co/api/v2/pokemon/1
- https://pokeapi.co/api/v2/pokemon

Una que hemos cogido los endpoints, hay que realizar las peticiones a ellos desde el Inspector, poniendo esas URLs en la barra de peticiones, donde además seleccionaremos como método HTTP el **GET**.



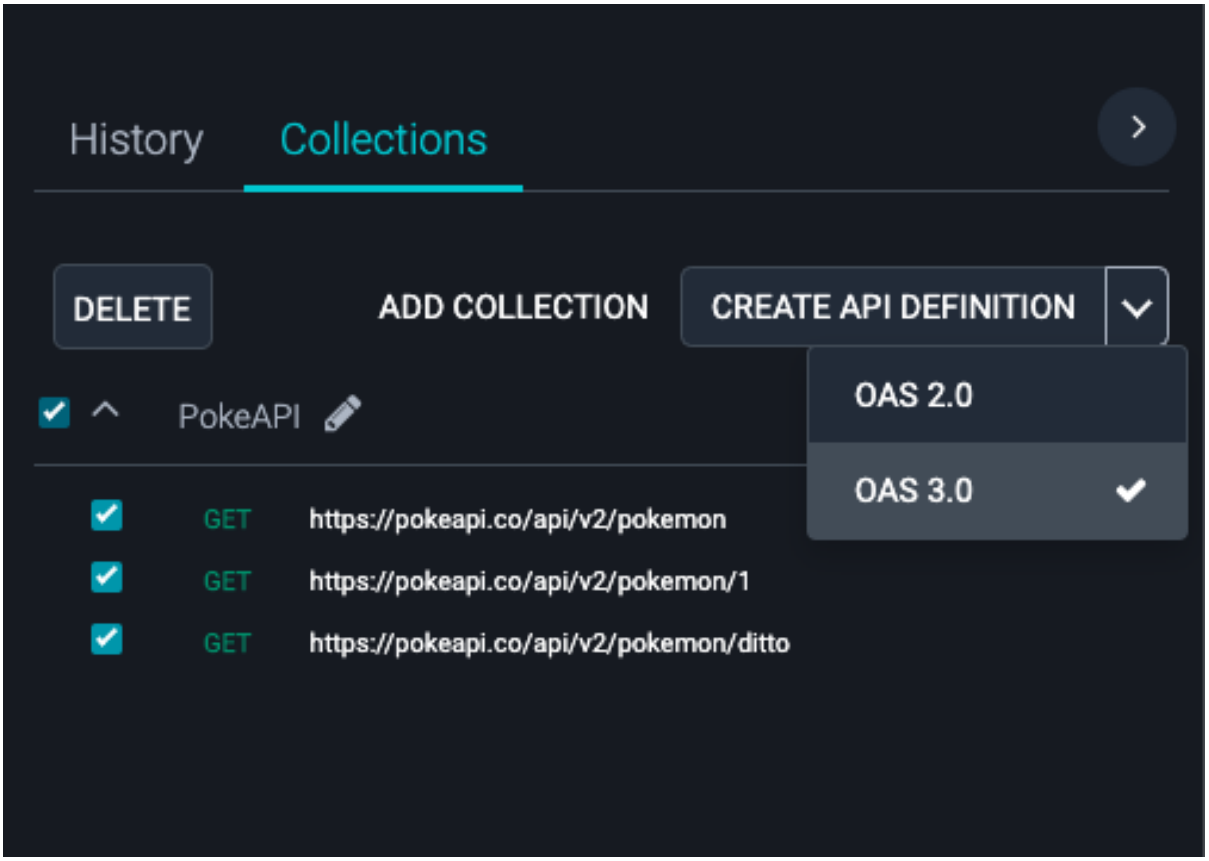
Una vez realizadas las 3 peticiones, nos deberían de salir en el historial de peticiones que aparece en el panel de la derecha.

El siguiente paso es agruparlas en una colección, por lo tanto seleccionamos los checkboxes de las 3 peticiones en el historial, y después le damos al botón de **Add to collection** que aparece un poco más arriba.

En el modal que aparece le damos a **Create new collection**, le ponemos como nombre **PokeAPI** y pulsamos sobre **Add to collection**.

Si cambiamos al panel de **Collections**, nos encontramos con la colección que acabamos de crear y las 3 peticiones dentro de ella.

Ahora tenemos que pulsar sobre el checkbox de PokeAPI para habilitar el botón de **Create API definition**, pero antes de pulsar sobre el para generar el documento OpenAPI, seleccionamos la versión de OpenAPI en la que queremos generar dicho documento pulsando sobre el desplegable que hay al lado del botón. En nuestro caso seleccionaremos **OAS3**.

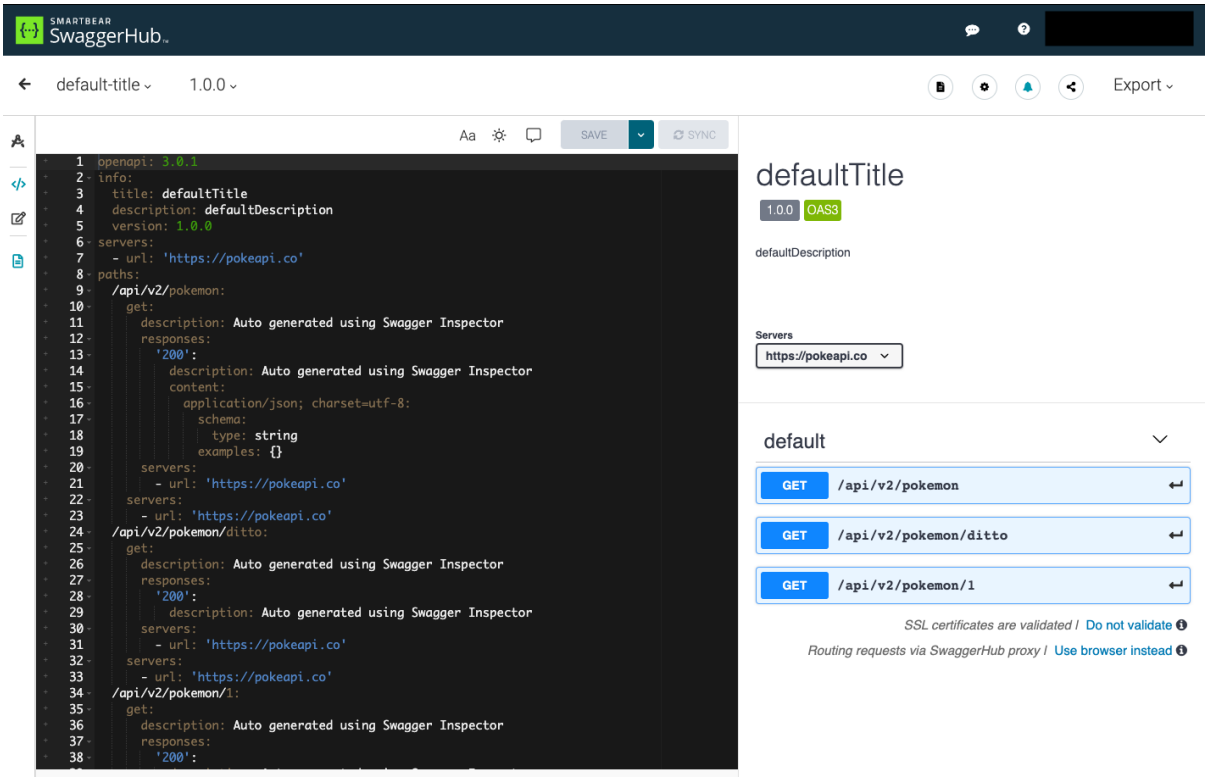


Una vez seleccionada la versión, pulsamos sobre el botón **Create API definition** el cual nos manda a **SwaggerHub** donde encontraremos la definición de la API y donde podremos modificar el documento.

Lo primero que nos encontramos al abrir esta nueva página es un modal donde nos pregunta:

- **Name**: PokeAPI
- **Version**: 1.0.0
- **Visibility**: Public
- **Owner**: team1
- **Project**: None

Le damos a **Import OpenAPI** y nos muestra el **Swagger Editor** integrado en SwaggerHub para poder modificar el documento OpenAPI y rellenar con los campos que faltan.



Ahora que ya tenemos la estructura básica del documento OpenAPI nos resultará más sencillo avanzar y completar esta documentación.

Para empezar, podemos cambiar el título y la descripción de la información, además de añadir la información de contacto.

```
info:
  title: PokeAPI
  description: API con toda la información de los Pokemon.
  version: 1.0.0
  contact:
    name: Paul Hallett
    url: https://phalt.github.io/
    email: ...@gmail.com
```

Los paths tienen en común el **/api/v2**, por lo tanto esto lo podemos considerar parte de la url del servidor y si tuviéramos distintas versiones, incluso podríamos ponerlas para seleccionarlas en un desplegable.

```
servers:
  - url: 'https://pokeapi.co/api/v2'
    description: Servidor de producción
```

Ahora tenemos que cambiar todos los paths y quitar el trozo del path que hemos añadido en la URL del servidor.

```
paths:
```

```
  /pokemon:
    ...
  /pokemon/ditto:
    ...
  /pokemon/1:
    ...
```

Como podemos observar, los dos últimos paths tienen un parámetro de path, por tanto podemos quitar uno de ellos, y poner el segundo segmento como un parámetro.

```
/pokemon/{pokemon}:
    get:
      parameters:
        - name: pokemon
          in: path
          description: id o nombre del pokemon
          required: true
          schema:
            oneOf:
              - type: string
              - type: integer
      description: Nos devuelve la información de un pokemon dado su id o su nombre.
      responses:
        ...
```