Designing and Documenting RESTful APIs with OpenAPI Specification Language

Software Engineering - Lab

Marco Robol - marco.robol@unitn.it

OpenAPI Specification Language

> *Online documentation:* https://swagger.io/docs/specification/v3_0/about

OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs. An OpenAPI file allows you to describe your entire API, including: endpoints and operations, input and output parameters, authentication.

Use the following tools to document and test your APIs:

- https://editor.swagger.io/
- https://app.apiary.io/ (No support for v3.0!)
- https://www.postman.com/

Basic Structure

A sample OpenAPI 3.0 definition written in YAML looks like:

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9

servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing

paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        "200": # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

All keyword names are **case-sensitive**.

Metadata

The OpenAPI version defines the overall structure of an API definition – what you can document and how you document it.

```
openapi: 3.0.0
```

The `info` section contains API information: `title`, `description` (optional), `version`:

```
info:
  title: Sample API
  description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9
```

Servers

The `servers` section specifies the API server and base URL. You can define one or several servers, such as production and sandbox.

```yaml
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
```

All API paths are relative to the server URL. In the example above, `/users` means `http://api.example.com/v1/users` or `http://staging-api.example.com/users`, depending on the server used. For more information, see API Server and Base Path.

Paths

API paths are defined in the global `paths` section of the API specification.

```yaml
paths:
  /ping: ...
  /users: ...
  /users/{id}:
    summary: Represents a user
    description: >
      This resource represents an individual user in the system.
      Each user is identified by a numeric `id`.
    get: ...
    delete: ...
```

All paths are relative to the API server URL. The full request URL is constructed as `<server-url>/path`. Paths may have an optional short `summary` and a longer `description` for documentation purposes. `description` can be multi-line and supports Markdown (CommonMark) for rich text representation.

Operations

The `paths` section defines individual endpoints (paths) in your API, and the **HTTP methods** (operations) supported by these endpoints. A single path can support multiple operations, for example `GET /users` to get a list of users and `POST /users` to add a new user. For example, `GET /users`:

```yaml
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML
      responses:
        "200":
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Operation Parameters

OpenAPI 3.0 supports operation parameters passed via **path**, **query string**, **headers**, and **cookies**. You can also define the request body for operations that transmit data to the server, such as POST, PUT and PATCH. For details, see Describing Parameters and Describing Request Body.

Path Templating

You can use curly braces `{}` to mark parts of an URL as path parameters:

```
/users/{id}
/organizations/{orgId}/members/{memberId}
/report.{format}
```

The API client needs to provide appropriate parameter values when making an API call, such as `/users/5` or `/users/12`.

Query String in Paths

Query string parameters **must not** be included in paths. They should be defined as query parameters instead.

```yaml
# Incorrect
paths:
  /users?role={role}:
```

```yaml
# Correct
paths:
  /users:
    get:
      parameters:
        - in: query
          name: role
          schema:
            type: string
            enum: [user, poweruser, admin]
          required: true
```

Describing Request Body

`requestBody` consists of the `content` object, an optional `description`, and an optional `required` flag (`false` by default). `content` lists the media types consumed by the operation (such as `application/json`) and specifies the `schema` for each media type. **Request bodies are optional by default**.

```yaml
paths:
  /pets:
    post:
      summary: Add a new pet
      requestBody:
        description: Optional description in *Markdown*
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Pet"
      responses:
        "201":
          description: Created
```

Components Section

You can define global common definitions and reference them using $ref.

```yaml
paths:
  /users/{userId}:
    get:
      summary: Get a user by ID
      parameters: ...
      responses:
        "200":
          description: A single user.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/User"
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
```

> Continue reading at https://swagger.io/docs/specification/v3_0/about/

Designing your RESTful APIs 1/4 - 10 minutes

Setup a **Swagger project** and synch with branch *swagger* in folder `swagger/aos3.yaml`

```yaml
# https://github.com/unitn-software-engineering/EasyLib/oas3.yaml
openapi: 3.0.0
info:
  version: '1.0'
  title: "EasyLib OpenAPI 3.0"
  description: API for managing book lendings.
  license:
    name: MIT
servers:
  - url: http://localhost:8000/api/v1
    description: Localhost
```

> Check out EasyLib APIs documentation at https://easylib.docs.apiary.io/# or
>
> https://app.swaggerhub.com/apis/IS-unitn/EasyLib/1

Now, starting from your user stories, design your RESTful APIs.

Designing your RESTful APIs 2/4 - 15 minutes

1. Identify at least **3 resources** and define the schemas

```yaml
# https://github.com/unitn-software-engineering/EasyLib/oas3.yaml
components:
  schemas:
    Booklending:
      type: object
      required:
      - student
      - book
      properties:
        user:
          type: string
          description: 'Link to the user'
        book:
          type: integer
          description: 'Link to the book'
```

Designing your RESTful APIs 3/4 - 15 minutes

2. Define your **root paths** to your resources and the supported methods

```yaml
# https://github.com/unitn-software-engineering/EasyLib/oas3.yaml
paths:
  /booklendings:
    post:
      description: >-
        Creates a new booklending.
      summary: Borrow a book
      responses:
        '201':
          description: 'Booklending created. Link in the Location header'
          headers:
            'Location':
              schema:
                type: string
              description: Link to the newly created booklending.
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Booklending'
```

Designing your RESTful APIs 4/4 - 15 minutes

3. Refine your APIs considering at least **1 sub resource** and at least **1 query parameter**

```
paths:
  /users:
    get:
      description: It is possible to show users by their role /users?role={role}
      parameters:
        - in: query
          name: role
          schema:
            type: string
            enum: [user, poweruser, admin]
          required: true
  /users/{userId}/books: ...
  /users/{userId}/books/{bookId}: ...
```

APIs Versioning

While you keep refining your APIs when considering more user stories, try not to modify previous APIs so to not break other parts of your application.
If at some point you will need to introduce some breaking changes, consider releasing a new version of your APIs. However, ensuring back-compatibility with older APIs is not simple, so at some point you may decide to drop the support to your old APIs.

```yaml
servers:
  - url: http://api.example.com/v1
```

```yaml
servers:
  - url: http://api.example.com/v2
```

Questions?

marco.robol@unitn.it