**5 Documenting REST APIs with OpenAPI**

This chapter covers

* Using JSON Schema to create validation models for JSON documents
* Describing REST APIs with the OpenAPI documentation standard
* Modeling the payloads for API requests and responses
* Creating reusable schemas in OpenAPI specifications

In this chapter, you’ll learn to document APIs using OpenAPI: the most popular standard for describing RESTful APIs, with a rich ecosystem of tools for testing, validating, and visualizing APIs. Most programming languages have libraries that support OpenAPI specifications, and in chapter 6 you’ll learn to use OpenAPI-compatible libraries from the Python ecosystem.

OpenAPI uses JSON Schema to describe an API’s structure and models, so we start by providing an overview of how JSON Schema works. JSON Schema is a specification for defining the structure of a JSON document, including the types and formats of the values within the document.

After learning about JSON Schema, we study how an OpenAPI document is structured, what its properties are, and how we use it to provide informative API specifications for our API consumers. API endpoints constitute the core of the specification, so we pay particular attention to them. We break down the process of defining the endpoints and schemas for the payloads of the API’s requests and responses, step by step. For the examples in this chapter, we work with the API of CoffeeMesh’s orders service. As we mentioned in chapter 1, CoffeeMesh is a fictional on-demand coffee-delivery platform, and the orders service is the component that allows customers to place and manage their orders. The full specification for the orders API is available under ch05/oas.yaml in the GitHub repository for this book.

**5.1 Using JSON Schema to model data**

This section introduces the specification standard for JSON Schema and explains how we leverage it to produce API specifications. OpenAPI uses an extended subset of the JSON Schema specification for defining the structure of JSON documents and the types and formats of its properties. It’s useful for documenting interfaces that use JSON to represent data and to validate that the data being exchanged is correct. The JSON Schema specification is under active development, with the latest version being 2020-12.[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/05.htm#pgfId-1304948)

**DEFINITION** *JSON Schema* is a specification standard for defining the structure of a JSON document and the types and formats of its properties. OpenAPI uses JSON Schema to describe the properties of an API.

A JSON Schema specification usually defines an object with certain attributes or properties. A JSON Schema object is represented by an associative array of key-value pairs. A JSON Schema specification usually looks like this:

{

"status": { ①

"type": "string" ②

}

}

① Each property in a JSON Schema specification comes as a key whose values are the descriptors of the property.

② The minimum descriptor necessary for a property is the type. In this case, we specify that the status property is a string.

In this example, we define the schema of an object with one attribute named status, whose type is string.

JSON Schema allows us to be very explicit with respect to the data types and formats that both the server and the client should expect from a payload. This is fundamental for the integration between the API provider and the API consumer, since it lets us know how to parse the payloads and how to cast them into the right data types in our runtime.

JSON Schema supports the following basic data types:

* string for character values
* number for integer and decimal values
* object for associative arrays (i.e., dictionaries in Python)
* array for collections of other data types (i.e., lists in Python)
* boolean for true or false values
* null for uninitialized data

To define an object using JSON Schema, we declare its type as object, and we list its properties and their types. The following shows how we define an object named order, which is one of the core models of the orders API.

Listing 5.1 Defining the schema of an object with JSON Schema

{

"order": {

"type": "object", ①

"properties": { ②

"product": {

"type": "string"

},

"size": {

"type": "string"

},

"quantity": {

"type": "integer"

}

}

}

}

① We can declare the schema as an object.

② We describe the object’s properties under the properties keyword.

Since order is an object, the order attribute also has properties, defined under the properties attribute. Each property has its own type. A JSON document that complies with the specification in listing 5.1 is the following:

{

"order": {

"product": "coffee",

"size": "big",

"quantity": 1

}

}

As you can see, each of the properties described in the specification is used in this document, and each of them has the expected type.

A property can also represent an array of items. In the following code, the order object represents an array of objects. As you can see, we use the items keyword to define the elements within the array.

Listing 5.2 Defining an array of objects with JSON Schema

{

"order": {

"type": "array",

"items": { ①

"type": "object",

"properties": {

"product": {

"type": "string"

},

"size": {

"type": "string"

},

"quantity": {

"type": "integer"

}

}

}

}

}

① We define the elements within the array using the items keyword.

In this case, the order property is an array. Array types require an additional property in their schema, which is the items property that defines the type of each of the elements contained in the array. In this case, each of the elements in the array is an object that represents an item in the order.

An object can have any number of nested objects. However, when too many objects are nested, indentation grows large and makes the specification difficult to read. To avoid this problem, JSON Schema allows us to define each object separately and to use JSON pointers to reference them. A *JSON pointer* is a special syntax that allows us to point to another object definition within the same specification.

As you can see in the following code, we can extract the definition of each item within the order array as a model called OrderItemSchema and use a JSON pointer to reference OrderItemSchema using the special $ref keyword.

Listing 5.3 Using JSON pointers to reference other schemas

{

"OrderItemSchema": {

"type": "object",

"properties": {

"product": {

"type": "string"

},

"size": {

"type": "string"

},

"quantity": {

"type": "integer"

}

}

},

"Order": {

"status": {

"type": "string"

},

"order": {

"type": "array",

"items": {

"$ref": '#/OrderItemSchema' ①

}

}

}

}

① We can specify the type of the array’s items using a JSON pointer.

JSON pointers use the special keyword $ref and JSONPath syntax to point to another definition within the schema. In JSONPath syntax, the root of the document is represented by the hashtag symbol (#), and the relationship of nested properties is represented by forward slashes (/). For example, if we wanted to create a pointer to the size property of the OrderItemSchema model, we would use the following syntax: '#/OrderItemSchema/size'.

**DEFINITION** A *JSON pointer* is a special syntax in JSON Schema that allows us to point to another definition within the same specification. We use the special keyword $ref to declare a JSON pointer. To build the path to another schema, we use JSONPath syntax. For example, to point to a schema called OrderItemSchema, defined at the top level of the document, we use the following syntax: {"$ref": "#/OrderItemSchema"}.

We can refactor our specification using JSON pointers by extracting common schema objects into reusable models, and we can reference them using JSON pointers. This helps us avoid duplication and keep the specification clean and succinct.

In addition to being able to specify the type of a property, JSON Schema also allows us to specify the format of the property. We can develop our own custom formats or use JSON Schema’s built-in formats. For example, for a property representing a date, we can use the date format—a built-in format supported by JSON Schema that represents an ISO date (e.g., 2025-05-21). Here’s an example:

{

"created": {

"type": "string",

"format": "date"

}

}

In this section, we’ve worked with examples in JSON format. However, JSON Schema documents don’t need to be written in JSON. In fact, it’s more common to write them in YAML format, as it’s more readable and easier to understand. OpenAPI specifications are also commonly served in YAML format, and for the remainder of this chapter, we’ll use YAML to develop the specification of the orders API.

**5.2 Anatomy of an OpenAPI specification**

In this section, we introduce the OpenAPI standard, and we learn to structure an API specification. OpenAPI’s latest version is 3.1; however, this version still has little support in the current ecosystem, so we’ll document the API using OpenAPI 3.0. There’s not much difference between the two versions, and nearly everything you learn about OpenAPI 3.0 applies to 3.1.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/05.htm#pgfId-1305088)

OpenAPI is a standard specification format for documenting RESTful APIs (figure 5.1). OpenAPI allows us to describe in detail every element of an API, including its endpoints, the format of its request and response payloads, its security schemes, and so on. OpenAPI was created in 2010 under the name of Swagger as an open source specification format for describing RESTful web APIs. Over time, this framework grew in popularity, and in 2015 the Linux Foundation and a consortium of major companies sponsored the creation of the OpenAPI initiative, a project aimed at improving the protocols and standards for building RESTful APIs. Today, OpenAPI is by far the most popular specification format used to document RESTful APIs,[**3**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/05.htm#pgfId-1305093) and it benefits from a rich ecosystem of tools for API visualization, testing, and validation.

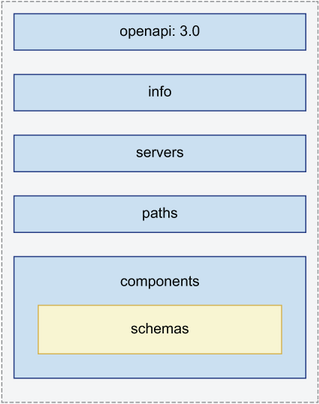


Figure 5.1 An OpenAPI specification contains five sections. For example, the paths section describes the API endpoints, while the components section contains reusable schemas referenced across the document.

An OpenAPI specification contains everything that the consumer of the API needs to know to be able to interact with the API. As you can see in figure 5.1, an OpenAPI is structured around five sections:

* openapi—Indicates the version of OpenAPI that we used to produce the specification.
* info—Contains general information, such as the title and version of the API.
* servers—Contains a list of URLs where the API is available. You can list more than one URL for different environments, such as the production and staging environments.
* paths—Describes the endpoints exposed by the API, including the expected payloads, the allowed parameters, and the format of the responses. This is the most important part of the specification, as it represents the API interface, and it’s the section that consumers will be looking for to learn how to integrate with the API.
* components—Defines reusable elements that are referenced across the specification, such as schemas, parameters, security schemes, request bodies, and responses.[**4**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/05.htm#pgfId-1305109) A *schema* is a definition of the expected attributes and types in your request and response objects. OpenAPI schemas are defined using JSON Schema syntax.

Now that we know how to structure an OpenAPI specification, let’s move on to documenting the endpoints of the orders API.

**5.3 Documenting the API endpoints**

In this section, we declare the endpoints of the orders API. As we mentioned in section 5.2, the paths section of an OpenAPI specification describes the interface of your API. It lists the URL paths exposed by the API, with the HTTP methods they implement, the types of requests they expect, and the responses they return, including the status codes. Each path is an object whose attributes are the HTTP methods it supports. In this section, we’ll focus specifically on documenting the URL paths and the HTTP methods. In chapter 4, we established that the orders API contains the following endpoints:

* POST /orders—Places an order. It requires a payload with the details of the order.
* GET /orders—Returns a list of orders. It accepts URL query parameters, which allow us to filter the results.
* GET /orders/{order\_id}—Returns the details of a specific order.
* PUT /orders/{order\_id}—Updates the details of an order. Since this is a PUT endpoint, it requires a full representation of the order.
* DELETE /orders/{order\_id}—Deletes an order.
* POST /orders/{order\_id}/pay—Pays for an order.
* POST /orders/{order\_id}/cancel—Cancels the order.

The following shows the high-level definitions of the orders API endpoints. We declare the URLs and the HTTP implemented by each URL, and we add an operation ID to each endpoint so that we can reference them in other sections of the document.

Listing 5.4 High-level definition of the orders API endpoints

paths:

/orders: ①

get: ②

operationId: getOrders

post: # creates a new order

operationId: createOrder

/orders/{order\_id}:

get:

operationId: getOrder

put:

operationId: updateOrder

delete:

operationId: deleteOrder

/orders/{order\_id}/pay:

post:

operationId: payOrder

/orders/{order\_id}/cancel:

post:

operationId: cancelOrder

① We declare a URL path.

② An HTTP method supported by the /orders URL path

Now that we have the endpoints, we need to fill in the details. For the GET /orders endpoint, we need to describe the parameters that the endpoint accepts, and for the POST and PUT endpoints, we need to describe the request payloads. We also need to describe the responses for each endpoint. In the following sections, we’ll learn to build specifications for different elements of the API, starting with the URL query parameters.

**5.4 Documenting URL query parameters**

As we learned in chapter 4, URL query parameters allow us to filter and sort the results of a GET endpoint. In this section, we learn to define URL query parameters using OpenAPI. The GET /orders endpoint allows us to filter orders using the following parameters:

* cancelled—Whether the order was cancelled. This value will be a Boolean.
* limit—Specifies the maximum number of orders that should be returned to the user. The value for this parameter will be a number.

Both cancelled and limit can be combined within the same request to filter the results:

GET /orders?cancelled=true&limit=5

This request asks the server for a list of five orders that have been cancelled. Listing 5.5 shows the specification for the GET /orders endpoint’s query parameters. The definition of a parameter requires a name, which is the value we use to refer to it in the actual URL. We also specify what type of parameter it is. OpenAPI 3.1 distinguishes four types of parameters: path parameters, query parameters, header parameters, and cookie parameters. Header parameters are parameters that go in an HTTP header field, while cookie parameters go into a cookie payload. Path parameters are part of the URL path and are typically used to identify a resource. For example, in /orders/ {order\_id}, order\_id is a path parameter that identifies a specific order. Query parameters are optional parameters that allow us to filter and sort the results of an endpoint. We define the parameter’s type using the schema keyword (Boolean in the case of cancelled, and a number in the case of limit), and, when relevant, we specify the format of the parameter as well.[**5**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/05.htm#pgfId-1305171)

Listing 5.5 Specification for the GET /orders endpoint’s query parameters

paths:

/orders:

get:

parameters: ①

- name: cancelled ②

in: query ③

required: false ④

schema: ⑤

type: boolean

- name: limit

in: query

required: false

schema:

type: integer

① We describe URL query parameters under the parameters property.

② The parameter’s name

③ We use the in descriptor to specify that the parameter goes in the URL path.

④ We specify whether the parameter is required.

⑤ We specify the parameter’s type under schema.

Now that we know how to describe URL query parameters, in the next section we’ll tackle something more complex: documenting request payloads.

**5.5 Documenting request payloads**

In chapter 4, we learned that a request represents the data sent by a client to the server through a POST or a PUT request. In this section, we learn to document the request payloads of the orders API endpoints. Let’s start with the POST /orders method. In section 5.1, we established that the payload for the POST /orders endpoint looks like this:

{

"order": [

{

"product": "cappuccino",

"size": "big",

"quantity": 1

}

]

}

This payload contains an attribute order, which represents an array of items. Each item is defined by the following three attributes and constraints:

* product—The type of product the user is ordering.
* size—The size of the product. It can be one of the three following choices: small, medium, and big.
* quantity—The amount of the product. It can be any integer number equal to or greater than 1.

Listing 5.6 shows how we define the schema for this payload. We define request payloads under the content property of the method’s requestBody property. We can specify payloads in different formats. In this case, we allow data only in JSON format, which has a media type definition of application/json. The schema for our payload is an object with one property: order, whose type is array. The items in the array are objects with three properties: the product property, with type string; the size property, with type string; and the quantity property, with type integer. In addition, we define an enumeration for the size property, which constrains the accepted values to small, medium, and big. Finally, we also provide a default value of 1 for the quantity property, since it’s the only nonrequired field in the payload. Whenever a user sends a request containing an item without the quantity property, we assume that they want to order only one unit of that item.

Listing 5.6 Specification for the POST /orders endpoint

paths:

/orders:

post:

operationId: createOrder

requestBody: ①

required: true ②

content: ③

application/json:

schema: ④

type: object

properties:

order:

type: array

items:

type: object

properties:

product:

type: string

size:

type: string

enum: ⑤

- small

- medium

- big

quantity:

type: integer

required: false

default: 1 ⑥

required:

- product

- size

① We describe request payloads under requestBody.

② We specify whether the payload is required.

③ We specify the payload’s content type.

④ We define the payload’s schema.

⑤ We can constrain the property’s values using an enumeration.

⑥ We specify a default value.

Embedding payload schemas within the endpoints’ definitions, as in listing 5.6, can make our specification more difficult to read and understand. In the next section, we learn to refactor payload schemas for reusability and for readability.

**5.6 Refactoring schema definitions to avoid repetition**

In this section, we learn strategies for refactoring schemas to keep the API specification clean and readable. The definition of the POST /orders endpoint in listing 5.6 is long and contains several layers of indentation. As a result, it’s difficult to read, and that means in the future it’ll become difficult to extend and to maintain. We can do better by moving the payload’s schema to a different section of the API specification: the components section. As we explained in section 5.2, the components section is used to declare schemas that are referenced across the specification. Every schema is an object where the key is the name of the schema, and the values are the properties that describe it.

Listing 5.7 Specification for the POST /orders endpoint using a JSON pointer

paths:

/orders:

post:

operationId: createOrder

requestBody:

required: true

content:

application/json:

schema:

$ref: '#/components/schemas/CreateOrderSchema' ①

components: ②

schemas:

CreateOrderSchema: ③

type: object

properties:

order:

type: array

items:

type: object

properties:

product:

type: string

size:

type: string

enum:

- small

- medium

- big

quantity:

type: integer

required: false

default: 1

required:

- product

- size

① We use a JSON pointer to reference a schema defined somewhere else in the document.

② Schema definitions go under components.

③ Every schema is an object, where the key is the name and the values are the properties that describe it.

Moving the schema for the POST /orders request payload under the components section of the API makes the document more readable. It allows us to keep the paths section of the document clean and focused on the higher-level details of the endpoint. We simply need to refer to the CreateOrderSchema schema using a JSON pointer:

#/components/schemas/CreateOrderSchema

The specification is looking good, but it can get better. CreateOrderSchema is a tad long, and it contains several layers of nested definitions. If CreateOrderSchema grows in complexity, over time it’ll become difficult to read and maintain. We can make it more readable by refactoring the definition of the order item in the array in the following code. This strategy allows us to reuse the schema for the order’s item in other parts of the API.

Listing 5.8 Schema definitions for OrderItemSchema and Order

components:

schemas:

OrderItemSchema: ①

type: object

properties:

product:

type: string

size:

type: string

enum:

- small

- medium

- big

quantity:

type: integer

default: 1

CreateOrderSchema:

type: object

properties:

order:

type: array

items:

$ref: '#/OrderItemSchema' ②

① We introduce the OrderItemSchema.

② We use a JSON pointer to point to OrderItemSchema.

Our schemas are looking good! The CreateOrderSchema schema can be used to create an order or to update it, so we can reuse it in the PUT /orders/{order\_id} endpoint, as you can see in listing 5.9. As we learned in chapter 4, the /orders/{order\_id} URL path represents a singleton resource, and therefore the URL contains a path parameter, which is the order’s ID. In OpenAPI, path parameters are represented between curly braces. We specify that the order\_id parameter is a string with a UUID format (a long, random string often used as an ID).[**6**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/05.htm#pgfId-1305342) We define the URL path parameter directly under the URL path to make sure it applies to all HTTP methods.

Listing 5.9 Specification for the PUT /orders/{order\_id} endpoint

paths:

/orders:

get:

...

/orders/{order\_id}: ①

parameters: ②

- in: path ③

name: order\_id ④

required: true ⑤

schema:

type: string

format: uuid ⑥

put: ⑦

operationId: updateOrder

requestBody: ⑧

required: true

content:

application/json:

schema:

$ref: '#/components/schemas/CreateOrderSchema'

① We declare the order’s resource URL.

② We define the URL path parameter.

③ The order\_id parameter is part of the URL path.

④ The name of the parameter

⑤ The order\_id parameter is required.

⑥ We specify the parameter’s format (UUID).

⑦ We define the HTTP method PUT for the current URL path.

⑧ We document the request body of the PUT endpoint.

Now that we understand how to define the schemas for our request payloads, let’s turn our attention to the responses.

**5.7 Documenting API responses**

In this section, we learn to document API responses. We start by defining the payload for the GET /orders/{order\_id} endpoint. The response of the GET /orders/ {order\_id} endpoint looks like this:

{

"id": "924721eb-a1a1-4f13-b384-37e89c0e0875",

"status": "progress",

"created": "2022-05-01",

"order": [

{

"product": "cappuccino",

"size": "small",

"quantity": 1

},

{

"product": "croissant",

"size": "medium",

"quantity": 2

}

]

}

This payload shows the products ordered by the user, when the order was placed, and the status of the order. This payload is similar to the request payload we defined in section 5.6 for the POST and PUT endpoints, so we can reuse our previous schemas.

Listing 5.10 Definition of the GetOrderSchema schema

components:

schemas:

OrderItemSchema:

...

GetOrderSchema: ①

type: object

properties:

status:

type: string

enum: ②

- created

- paid

- progress

- cancelled

- dispatched

- delivered

created:

type: string

format: date-time ③

order:

type: array

items:

$ref: '#/components/schemas/OrderItemSchema' ④

① We define the GetOrderSchema schema.

② We constrain the values of the status property with an enumeration.

③ A string with date-time format

④ We reference the OrderItemSchema schema using a JSON pointer.

In listing 5.10, we use a JSON pointer to point to GetOrderSchema. An alternative way to reuse the existing schemas is to use inheritance. In OpenAPI, we can inherit and extend a schema using a strategy called *model composition*, which allows us to combine the properties of different schemas in a single object definition. The special keyword allOf is used in these cases to indicate that the object requires all the properties in the listed schemas.

**DEFINITION** *Model composition* is a strategy in JSON Schema that allows us to combine the properties of different schemas into a single object. It is useful when a schema contains properties that have already been defined elsewhere, and therefore allows us to avoid repetition.

The following code shows an alternative definition of GetOrderSchema using the allOf keyword. In this case, GetOrderSchema is the composition of two other schemas: CreateOrderSchema and an anonymous schema with two keys—status and created.

Listing 5.11 Alternative implementation of GetOrderSchema using the allOf keyword

components:

schemas:

OrderItemSchema:

...

GetOrderSchema:

allOf: ①

- $ref: '#/components/schemas/CreateOrderSchema' ②

- type: object ③

properties:

status:

type: string

enum:

- created

- paid

- progress

- cancelled

- dispatched

- delivered

created:

type: string

format: date-time

① We use the allOf keyword to inherit properties from other schemas.

② We use a JSON pointer to reference another schema.

③ We define a new object to include properties that are specific to GetOrderSchema.

Model composition results in a cleaner and more succinct specification, but it only works if the schemas are strictly compatible. If we decide to extend CreateOrderSchema with new properties, then this schema may no longer be transferable to the GetOrderSchema model. In that sense, it’s sometimes better to look for common elements among different schemas and refactor their definitions into standalone schemas.

Now that we have the schema for the GET /orders/{order\_id} endpoint’s response payload, we can complete the endpoint’s specification. We define the endpoint’s responses as objects in which the key is the response’s status code, such as 200. We also describe the response’s content type and its schema, GetOrderSchema.

Listing 5.12 Specification for the GET /orders/{order\_id} endpoint

paths:

/orders:

get:

...

/orders/{order\_id}:

parameters:

- in: path

name: order\_id

required: true

schema:

type: string

format: uuid

put:

...

get: ①

summary: Returns the details of a specific order ②

operationId: getOrder

responses: ③

'200': ④

description: OK ⑤

content: ⑥

application/json:

schema:

$ref: '#/components/schemas/GetOrderSchema' ⑦

① We define the GET endpoint of the /orders/{order\_id} URL path.

② We provide a summary description of this endpoint.

③ We define this endpoint’s responses.

④ Each response is an object where the key is the status code.

⑤ A brief description of the response

⑥ We describe the content types of the response.

⑦ We use a JSON pointer to reference GetOrderSchema.

As you can see, we define response schemas within the responses section of the endpoint. In this case, we only provide the specification for the 200 (OK) successful response, but we can also document other status codes, such as error responses. The next section explains how we create generic responses we can reuse across our endpoints.

**5.8 Creating generic responses**

In this section, we learn to add error responses to our API endpoints. As we mentioned in chapter 4, error responses are more generic, so we can use the components section of the API specification to provide generic definitions of those responses, and then reuse them in our endpoints.

We define generic responses within the responses header of the API’s components section. The following shows a generic definition for a 404 response named NotFound. As with any other response, we also document the content’s payload, which in this case is defined by the Error schema.

Listing 5.13 Generic 404 status code response definition

components:

responses: ①

NotFound: ②

description: The specified resource was not found. ③

content: ④

application/json:

schema:

$ref: '#/components/schemas/Error' ⑤

schemas:

OrderItemSchema:

...

Error: ⑥

type: object

properties:

detail:

type: string

required:

- detail

① Generic responses go under responses in the components section.

② We name the response.

③ We describe the response.

④ We define the response’s content.

⑤ We reference the Error schema.

⑥ We define the schema for the Error payload.

This specification for the 404 response can be reused in the specification of all our endpoints under the /orders/{order\_id} URL path, since all of those endpoints are specifically designed to target a specific resource.

**NOTE** You may be wondering, if certain responses are common to all the endpoints of a URL path, why can’t we define the responses directly under the URL path and avoid repetition? The answer is this isn’t possible as of now. The responses keyword is not allowed directly under a URL path, so we must document all the responses for every endpoint individually. There’s a request in the OpenAPI GitHub repository to allow including common responses directly under the URL path, but it hasn’t been implemented (<http://mng.bz/097p>).

We can use the generic 404 response from listing 5.13 under the GET /orders/ {order\_id} endpoint.

Listing 5.14 Using the 404 response schema under GET /orders/{order\_id}

paths:

...

/orders/{order\_id}:

parameters:

- in: path

name: order\_id

required: true

schema:

type: string

"format": uuid

get:

summary: Returns the details of a specific order

operationId: getOrder

responses:

'200':

description: OK

content:

application/json:

schema:

$ref: '#/components/schemas/GetOrderSchema'

**'404':**  ①

**$ref: '#/components/responses/NotFound'**  ②

① We define a 404 response.

② We reference the NotFound response using a JSON pointer.

The orders API specification in the GitHub repository for this book also contains a generic definition for 422 responses and an expanded definition of the Error component that accounts for the different error payloads we get from FastAPI.

We’re nearly done. The only remaining endpoint is GET /orders, which returns a list of orders. The endpoint’s payload reuses GetOrderSchema to define the items in the orders array.

Listing 5.15 Specification for the GET /orders endpoint

paths:

/orders:

get: ①

operationId: getOrders

responses:

'200':

description: A JSON array of orders

content:

application/json:

schema:

type: object

properties:

orders:

type: array ②

items:

$ref: '#/components/schemas/GetOrderSchema' ③

required:

- order

post:

...

/orders/{order\_id}:

parameters:

...

① We define the new GET method of the /orders URL path.

② orders is an array.

③ Each item in the array is defined by GetOrderSchema.

Our API’s endpoints are now fully documented! You can use many more elements within the definitions of your endpoints, such as tags and externalDocs. These attributes are not strictly necessary, but they can help to provide more structure to your API or make it easier to group the endpoints. For example, you can use tags to create groups of endpoints that logically belong together or share common features.

Before we finish this chapter, there’s one more topic we need to address: documenting the authentication scheme of our API. That’s the topic of the next section!

**5.9 Defining the authentication scheme of the API**

If our API is protected, the API specification must describe how users need to authenticate and authorize their requests. This section explains how we document our API’s security schemes. The security definitions of the API go within the components section of the specification, under the securitySchemes header.

With OpenAPI, we can describe different security schemes, such as HTTP-based authentication, key-based authentication, Open Authorization 2 (OAuth2), and OpenID Connect.[**7**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/05.htm#pgfId-1305609) In chapter 11, we’ll implement authentication and authorization using the OpenID Connect and OAuth2 protocols, so let’s go ahead and add definitions for these schemes. Listing 5.16 shows the changes we need to make to our API specification to document the security schemes.

We describe three security schemes: one for OpenID Connect, another one for OAuth2, and another for bearer authorization. We’ll use OpenID Connect to authorize user access through a frontend application, and for direct API integrations, we’ll offer OAuth’s client credentials flow. We’ll explain how each protocol and each authorization flow works in detail in chapter 11. For OpenID Connect, we must provide a configuration URL that describes how our backend authentication works under the openIdConnectUrl property. For OAuth2, we must describe the authorization flows available, together with a URL that clients must use to obtain their authorization tokens and the available scopes. The bearer authorization tells users that they must include a JSON Web Token (JWT) in the Authorization header to authorize their requests.

Listing 5.16 Documenting an API’s security scheme

components:

responses:

...

schemas:

...

**securitySchemes:**  ①

**openId:**  ②

**type: openIdConnect**  ③

**openIdConnectUrl: https://coffeemesh-dev.eu.auth0.com/.well-**

➥ **known/openid-configuration**  ④

**oauth2:**  ⑤

**type: oauth2**  ⑥

**flows:**  ⑦

**clientCredentials:**  ⑧

**tokenUrl: https://coffeemesh-dev.eu.auth0.com/oauth/token** ⑨

**scopes: {}**  ⑩

**bearerAuth:**

**type: http**

**scheme: bearer**

**bearerFormat: JWT**  ⑪

**...**

**security:**

**- oauth2:**

**- getOrders**

**- createOrder**

**- getOrder**

**- updateOrder**

**- deleteOrder**

**- payOrder**

**- cancelOrder**

**- bearerAuth:**

**- getOrders**

**- createOrder**

**- getOrder**

**- updateOrder**

**- deleteOrder**

**- payOrder**

**- cancelOrder**

① The security schemes under the securitySchemes header of the API’s components section

② We provide a name for the security scheme (it can be any name).

③ The type of security scheme

④ The URL that describes the OpenID Connect configuration in our backend

⑤ The name of another security scheme

⑥ The type of the security scheme

⑦ The authorization flows available under this security scheme

⑧ A description of the client credentials flow

⑨ The URL where users can request authorization tokens

⑩ The available scopes when requesting an authorization token

⑪ The bearer token has a JSON Web Token (JWT) format.

This concludes our journey through documenting REST APIs with OpenAPI. And what a ride! You’ve learned how to use JSON Schema; how OpenAPI works; how to structure an API specification; how to break down the process of documenting your API into small, progressive steps; and how to produce a full API specification. The next time you work on an API, you’ll be well positioned to document its design using these standard technologies.

**Summary**

* JSON Schema is a specification for defining the types and formats of the properties of a JSON document. JSON Schema is useful for defining data validation models in a language-agnostic manner.
* OpenAPI is a standard documentation format for describing REST APIs and uses JSON Schema to describe the properties of the API. By using OpenAPI, you’re able to leverage the whole ecosystem of tools and frameworks built around the standard, which makes it easier to build API integrations.
* A JSON pointer allows you to reference a schema using the $ref keyword. Using JSON pointers, we can create reusable schema definitions that can be used in different parts of an API specification, keeping the API specification clean and easy to understand.
* An OpenAPI specification contains the following sections:
  + openapi—Specifies the version of OpenAPI used to document the API
  + info—Contains information about the API, such as its title and version
  + servers—Documents the URLs under which the API is available
  + paths—Describes the endpoints exposed by the API, including the schemas for the API requests and responses and any relevant URL path or query parameters
  + components—Describes reusable components of the API, such as payload schemas, generic responses, and authentication schemes

**1** A. Wright, H. Andrews, B. Hutton, “JSON Schema: A Media Type for Describing JSON Documents” (December 8, 2020); <https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00>. You can follow the development of JSON Schema and contribute to its improvement by participating in its repository in GitHub: <https://github.com/json-schema-org/json-schema-spec>. Also see the website for the project: <https://json-schema.org/>.

**2** For a detailed analysis of the differences between OpenAPI 3.0 and 3.1, check out OpenAPI’s migration from 3.0 to 3.1 guide: <https://www.openapis.org/blog/2021/02/16/migrating-from-openapi-3-0-to-3-1-0>.

**3** According to the 2022 “State of the API” report by Postman (<https://www.postman.com/state-of-api/api-technologies/#api-technologies>).

**4** See <https://swagger.io/docs/specification/components/> for a full list of reusable elements that can be defined in the components section of the API specification.

**5** To learn more about the date types and formats available in OpenAPI 3.1 see <http://spec.openapis.org/oas/v3.1.0#data-types>.

**6** P. Leach, M. Mealling, and R. Salz, “A Universally Unique Identifier (UUID) URN Namespace,” RFC 4112 (<https://datatracker.ietf.org/doc/html/rfc4122>).

**7** For a complete reference of all the security schemas available in OpenAPI, see <https://swagger.io/docs/specification/authentication/>.