

# Designing APIs

with Swagger and OpenAPI

Josh Ponelat  
Lukas Rosenstock



MANNING





**MEAP Edition**  
**Manning Early Access Program**  
**Designing APIs with Swagger and OpenAPI**  
**Version 8**

Copyright 2021 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](https://manning.com)

# welcome

---

Thank you for purchasing the MEAP of *Designing APIs with Swagger and OpenAPI!*

My name is Josh Ponelat ( Pah-neh-lat ) and I have the privilege of introducing you to the world of OpenAPI and Swagger. The eco-systems that surround RESTful APIs and the sheer joy of leveraging and automating those systems.

The goal of this book is to teach the fundamentals of describing APIs using OpenAPI, and how those descriptions can be used to drive parts of your work flows and pipelines with tooling, such as those prefixed with Swagger.

Within you'll find out how to read and write OpenAPI definitions, where writing consists of describing existing or new RESTful APIs. We'll disambiguate as much as we can, aiming to answer questions as to "why" something is the way it is. After literacy of OpenAPI, we'll dive into the design process. How to create new APIs cheaply and iterate on their design. And finally we'll look at some lovely advanced topics, including my favourite.. guidelines to building your own OpenAPI tooling.

OpenAPI and Swagger are both Open Source, and there are resources on the subjects online, but few books to capture just how they work and why they're so sorely needed today. While I've endeavoured to do just that, I'd like to ask for your help.

Please reach out in the [liveBook's discussion forum](#) and comments, the biggest help can come from silly questions, such as asking for clarity on a subject or even just asking why something happens to be that way.

Ultimately by learning the fundamentals of OpenAPI and Swagger, my hope is that this book will make your life a little easier and hopefully even inspire you to automate even more things!

—Joshua S. Ponelat

# *brief contents*

---

## **PART 1**

- 1 Introducing APIs and OpenAPI*
- 2 Getting set up to make API requests*
- 3 Our first taste of OpenAPI definitions*
- 4 Using SwaggerEditor to write OpenAPI definitions*
- 5 Describing API responses*
- 6 Creating resources*
- 7 Adding Authentication and Authorization*
- 8 Preparing and hosting API documentation*

## **PART 2**

- 9 Designing a web application*
- 10 Creating an API design using OpenAPI*
- 11 Building a change workflow around API Design First*
- 12 Implementing frontend code and reacting to changes*
- 13 Building a Backend with Node.js and Swagger Codegen*
- 14 Integrating and releasing the web application*
- 15 The API Design First approach*

## **PART 3**

- 16 Designing the next API iteration*
- 17 Versioning an API and handling breaking changes*
- 18 Scaling collection endpoints with filters and pagination*
- 19 Supporting the unhappy path: error handling with problem+json*
- 20 Improving input validation with advanced JSON Schema*
- 21 Beyond the application: what's next for our API?*

## APPENDIXES

*A OpenAPI keywords*

*B Differences in specification versions (swagger 2.0 vs openapi 3.x)*

# *Introducing APIs and OpenAPI*



## This chapter covers

- Describing an API Ecosystem
- What OpenAPI and Swagger are
- When to use OpenAPI
- An overview of this book

In this chapter we'll take a look at the world of APIs and OpenAPI, so that we can get comfortable with the topics of this book. We'll start by taking a look at the benefits of describing an API, how it forms part of an API ecosystem and where OpenAPI fits in. We'll look at an example of an OpenAPI document and when to use OpenAPI in practice.

Lets get started...

## **1.1 What is an API Ecosystem?**

I like the word ecosystem. It describes the interactions and relationships between living and non-living within a fully functioning environment. I always picture a wetland pond with frogs, wild grasses and stones for some reason, but you may imagine something a little different - either way the principle of an interactive symbiotic system remains.

If we were to borrow (**cough** maybe steal ) this principle from biology we could use it to describe the world of APIs within a team or organization.

The living, changing variables would represent elements that we have control over. These are the things we make such as our services, stacks or code. The fixed, non living components would

then be the useful things we can benefit from but cannot easily change. These are the libraries and external services we use. And of course there is the environment. It could be the Internet, an internal network or a tiny device stuck on the roof of our house. Perhaps even all of them!

All of these pieces put together forms a complete ecosystem. When these parts are moving in harmony then our system is healthy and our developers/consumers/users are all happy.

APIs tie together these disparate services, forming the foundation of the ecosystem. When we assume the role of an "API designer", then our job is to create contracts for services, incorporate feedback from consumers and ensure changes are communicated ahead of time.

Why *API Ecosystem* and not *Service Ecosystem* or perhaps even just *Ecosystem* you might ask? That's a reasonable question, and the answer would depend on which we choose to focus on. In this book we're interested in APIs so naturally we focus on that aspect. Since APIs are the contracts that hold together the ecosystem it is not an unreasonable focal point. This is in fact a very important part of the ecosystem, without which our services are all isolated. Understanding them gives us a holistic perspective.

This book is going to focus on the APIs and the contracts that define, or describe them.

## 1.2 Describing things

When we look at our ecosystem as individual aspects of a multitude of connections, we might wonder at what each aspect truly means, what it's comprised of, and how each one fits into this design. The *how* part is where all the interesting bits are.

When those services change without updating all of their dependencies the ecosystem loses functionality and in some cases can completely break.

Let's take a look at this story, for example.

### 1.2.1 Bridget's Task

Bridget has been tasked to manage a medium sized web stack. Her stack (or ecosystem) is made up of services that talk to, and depend on each other. The stack also makes use of external services which are beyond her control.

Every now and then one of the APIs will change in such a way as to negatively impact, and sometimes break the services that rely on it. This disrupts the ecosystem, bringing down parts of her stack and ultimately causing failures.

Bridget will need to effectively solve this problem, therefore when an API changes she needs to tell the affected developers beforehand and keep the ecosystem running smoothly.

Bridget takes a moment to think about how this ecosystem works, by breaking it down into tangible steps. She knows that each service has an API, and that each of those APIs are made up of smaller operations. Each operation expects a certain input and generates an equal resulting output. When an operation changes so that it requires different inputs, any service that doesn't evolve and adapt along with it will result in a systemic failure.

In the same way, when an operation produces a different output, it will cause other dependent services to break unless they update to address those changes.

Bridget knows that if an API changes its operations to expect different inputs or produce different outputs, then her ecosystem will suffer.

She concludes that tracking those changes is an important part of keeping functionality up, but how will she know when an API has changed?

Bridget decides she needs a way of describing APIs, so that she can compare an old API with a new API, in order to see if the new one has any breaking changes. She writes a program that takes a description of an API and compares the older version with a newer version - generating a report. The report is simple and tells her if the new API has any breaking changes from the old.

Happy with her plan, she instructs the developers to describe their APIs in her format so that she can continue to compare old with new. Aware that the external services aren't under control, she keeps an eye on any developments and describes it herself- she is prepared for when those external services change.

### 1.2.2 *The potential of Bridget's solution*

Bridget's solution to her problem is centered around the idea that you can describe your APIs in such a way that people can write them and software can understand them.

While she only used that approach to solve one specific problem, there is now much more potential for growth with those descriptions. They can serve as the basis for generating more than just reports. For example she could generate documentation, test out changes before building them, reduce the overhead of boilerplate code and much more.

Given the title of this book, the punchline may be a little ruined. But let us take a look at how Bridget's solution is used in the real world. Let's take a look at how OpenAPI works...

## 1.3 *What is OpenAPI?*

OpenAPI is a description of HTTP-based APIs which are typically, RESTful APIs. It comes in the form of a YAML file or definition, that describes the inputs and outputs of an API. It can also include information such as where the API is hosted, what authorization is required to access it, and other details needed for consumers and producers (such as web-developers).

Definitions can be written by hand, by tools, or even generated from code. Once an API has been written down we say it has been *described*. Once API has been described into a definition, it becomes a platform for tools and humans to make use of. A typical example of making use of API definitions is to generate human readable documentation from it.

### 1.3.1 Example OpenAPI Definition

There is a fun little API for dog breeds and their images on the Internet hosted at [dog.ceo](https://dog.ceo). To give you an example of what an OpenAPI definition looks like, I've described a single operation from this dog API, along with some other basic details of the API. Here is that definition ( as a YAML file )...

#### Listing 1.1 Example OpenAPI Document/Definition

```
openapi: 3.0.0
info:
  title: Dog API
  version: 1.0.0
servers:
- url: https://dog.ceo/api
paths:
  /breed/{breedName}/images:
    get:
      description: Get images of dog breeds
      parameters:
        - in: path
          name: breedName
          schema:
            type: string
            example: hound
            required: true
      responses:
        '200':
          description: A list of dog images
          content:
            application/json:
              schema:
                type: object
                properties:
                  status:
                    type: string
                    example: success
                  message:
                    type: array
                    items:
                      type: string
                      example: https://images.dog.ceo/breeds/hound-afghan/n02088094_1003.jpg
```

It can be a little verbose at first glance, but you will find some exceptionally useful information contained within.

From it, we can gather a few snippets about the single operation it describes and how to consume it.

- The API is hosted at [dog.ceo/api](https://dog.ceo/api)
- There is a GET operation: GET /breed/{breedName}/images

- Where the `breedName` in the URI, is a string
- We see that a successful response will give us a JSON array where each item is an object containing the `message` and `status` fields.
- The `message` fields is an array of strings which are URLs of dog images.

That is usable information. Developers can build clients to consume the API, product managers can determine if the API suits their needs and meets their standards, and documentation teams can use it as the basis for showing human readable documentation.

As an example of using an OpenAPI definition, we can load this definition into a tool called SwaggerUI (we'll go into details of that later in the book). It will render documentation based on the definition and provide other small niceties. It would look something like the following...

**Figure 1.1 SwaggerUI with Dogs API**

SwaggerUI can consume the API definition file and from it render a more human friendly version of it.

## 1.4 Where do OpenAPI definitions fit in ?

Once we have an API definition we use tools to leverage them, build bigger abstractions and more automated work flows.

Here is a diagram showing how OpenAPI definitions could fit into an organization's work flows.

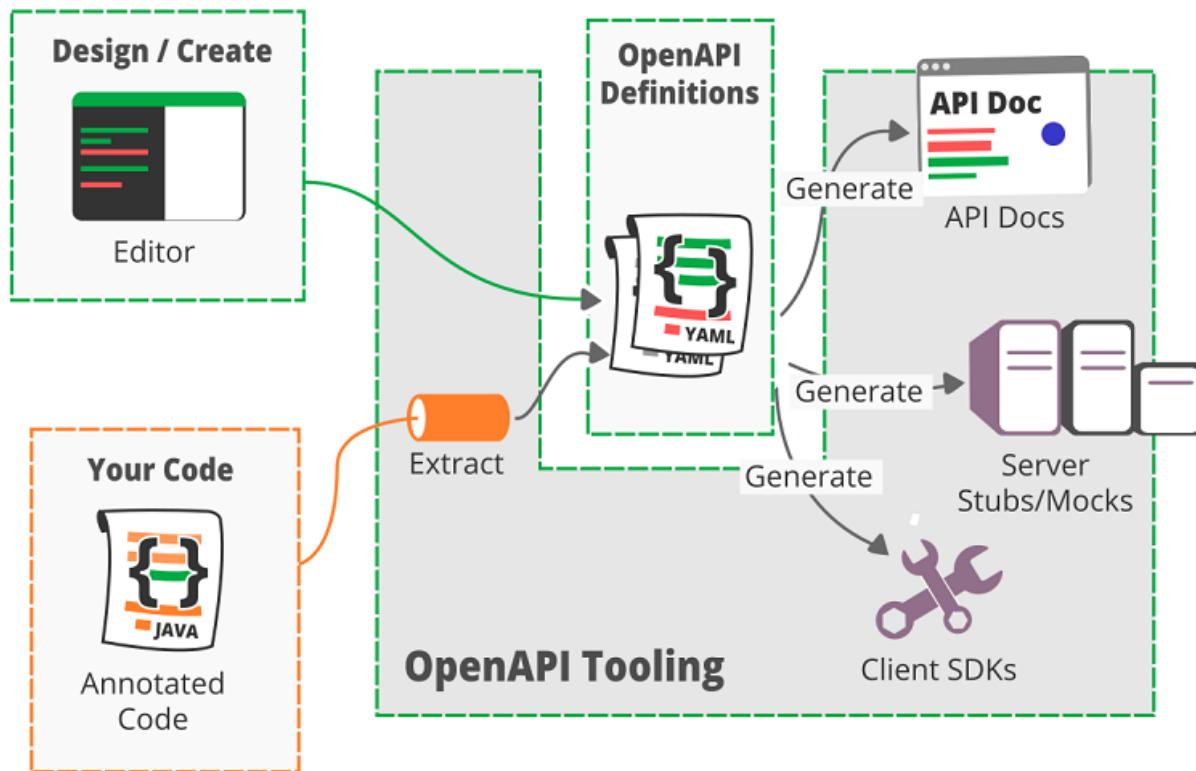


Figure 1.2 OpenAPI

The diagram shows how to leverage OpenAPI definitions, that are described by tools or by extracting annotations from code. They are then transformed into API documentation, server stubs / routers and client SDKs. There are certainly more work flows that could be diagrammed out, ones that provide more specific value depending on the business cases.

Other work flows could include...

#### Example Workflows:

- Automating parts of testing (APIs)
- Getting early feedback on the design of the API
- Ensuring API consistency
- Comparing API changes across versions, etc.

The beauty of OpenAPI is that, once you have an OpenAPI definition, the rest (pun!) is simply a matter of leveraging it for your needs.

## 1.5 And what is Swagger?

In the beginning there was SwaggerUI and a rough guide for writing YAML files that described HTTP APIs. Later, more tools were built that relied on this guide that soon became a specification and a standard. The tools and this specification were collectively known as "Swagger". The term referred to all of it. The specification grew more mature and was released as open source encouraging the community to create even more tools. They soon began to contribute features into the specification, which finally began to be adopted by large companies and household names.

Then in 2015 it was adopted by SmartBear who then donated the specification part to the Linux Foundation<sup>1</sup>. During that transfer the specification part was subsequently renamed to the OpenAPI Specification and SmartBear retained the copyright for the term Swagger.

Today you'll find the terms used interchangeably as a result of this historical quirk. Going forward it's encouraged to use the term OpenAPI to refer the heart of this ecosystem: the specification. And to use Swagger to refer to the specific set of tools managed by SmartBear ( which includes SwaggerUI, SwaggerEditor, SwaggerParser and at least a dozen more ).

## 1.6 What about REST?

REST or REpresentational State Transfer is a collection of ideas around how to design networked systems (in particular server/client systems), and while it is not restricted to HTTP based APIs they are both closely linked in practice.

The principles of REST were outlined by Roy Fielding in his dissertation on networked systems which was released in the year 2000. RESTful APIs now drive the majority of web servers on the Internet and what makes an API RESTful is determined by how closely it adopts the ideas ( or constraints ) of that dissertation.<sup>2</sup> and can be considered a little subjective. Out in the wild HTTP-based APIs have to make trade offs between what they require, and how standard or RESTful they wish to be. It is a balancing act for all API producers to juggle.

The ideas in REST aim to be simple, and to decouple the API from the underlying services that serve the API. It has a request/response model and is stateless, as all the information necessary to do something is contained within the request.

One of the key ideas behind REST is the idea of a resource. I like to think of them as boxes that you can put things into. Things such as *user accounts*, *billing reminders* or even the *weather in San Francisco*, all of these things are resources. Resources are identified by a URI. For a user's account we might have `/users/123` where 123 is the identifier for that user.

Given a resource, consumers will want to be able to *do things* to and with them. Think of these actions as verbs. HTTP has a set of well defined ones, such as

POST,GET,PUT,DELETE,PATCH and a few less common ones. These are HTTP methods but they are derived from the ideas in REST. As an example of a HTTP method, if you wish to fetch data related to a resource you would use the GET method. If you wish to create a new resource you could use the POST method.

Where REST starts and HTTP ends is a tricky one to answer, but the rule of thumb is HTTP is the actual protocol and REST is a way of designing APIs. HTTP has incorporated many of the ideas of REST into its protocol, which is why they are so closely related.

Typically we'll more often note when a HTTP API is *not* RESTful, by that we mean it doesn't conform to the design patterns outlined by REST.

OpenAPI was designed to describe as many HTTP based API as possible, but not all of them. Its major constraint ( and a huge benefit of OpenAPI ) is to still allow tools to generate usable code from the definitions, this means that some API nuances may not be describable in OpenAPI.

#### NOTE

#### A quick note regarding Hypermedia.

In Roy's REST paper he mentions the idea of Hypermedia. A system of returning context aware links, in the form of URIs. For example, If you were to execute a GET on a `/users/{userId}` resource it could return a link ( URI ) to execute a password reset. Another link might be to the login operation for that user. The links are related to resource ( at that point in time ) and decouple clients from knowing those URIs outside of the response. This is a crude description of a very powerful model.

Many REST purists are eager to point out that this is a sorely missed component of RESTful APIs. In OpenAPI ( particularly v3.0.0+ ) support was added to help document these hypermedia *links*, but their semantics are out of scope for the OpenAPI specification.

OpenAPI is sufficient to describe what is required by Hypermedia APIs but not the semantics of what each link should do. There are specifications that attempt to tackle describing those details. Here is an incomplete list that I know of:

- HATEOAS: [restfulapi.net/hateoas/](http://restfulapi.net/hateoas/)
- Siren: [github.com/kevinswiber/siren](https://github.com/kevinswiber/siren)
- Hydra: [www.hydra-cg.com/](http://www.hydra-cg.com/)

## 1.7 When to use OpenAPI

Always.

I hope that statement triggered the picture of a grinning author. I couldn't resist. But let's take a look at when using OpenAPI makes sense.

OpenAPI describes HTTP based APIs (including RESTful APIs) so when you're tasked with designing, managing and in some cases consuming the API, then OpenAPI makes sense. Of course if you're dealing with other API technologies that don't leverage HTTP, such as gRPC or GraphQL... then OpenAPI doesn't make sense at all.

For this section, when we refer to APIs we're referring to HTTP based APIs. It's a little easier to communicate with that assumption.

### **1.7.1 For API consumers**

If you're required to consume some API you may need an SDK, and if you have OpenAPI you could generate SDKs for many different languages. The benefit is in having familiarity with a single type of SDK or even customizing the SDK to suit your needs. Each and every API described with OpenAPI can then be turned into an SDK of your choosing. Although typically the SDKs provided by tools such as SwaggerCodegen or OpenapiGenerator are sufficient and give a good head start into developing clients.

More importantly is the ability to know what to build a client against, and to even develop against mock servers, generated by the same OpenAPI definition.

### **1.7.2 For API producers**

Building APIs can be quite fun, particularly when you have a contract to develop against (and you agree with that design! Perhaps a different conversation). Building out the boilerplate of an HTTP server is less fun once you've done it umpteen times. Generating boilerplate code and stubs from an OpenAPI definition gives you speed and consistency ( since you can customize the templates to your needs ).

Although there are even more exciting methods of developing APIs, using OpenAPI definitions during runtime to act as a router (have API operations map to classes/methods in code) and as a validation layer (incoming requests will fail validation unless they conform to the OpenAPI definition's schema). Such practices are becoming more common in microservice-oriented architecture where services are being built-out at a faster rate.

### **1.7.3 For API designers**

Designing APIs has also been given a new focus of late ( at least at the time of writing ) and its importance cannot be understated. While I'm a huge fan of Agile practices ( short feedback cycles ) and the art of failing fast ( ie: bringing products to market quicker to validate success/failure )... APIs should still be designed with longevity in mind as changing them, means changing the consumers ( typically beyond your control ). No-one likes getting stuck maintaining an old API!

OpenAPI is a medium to communicate to both consumers and producers, allowing designers to get feedback early in the process and to iterate based on that feedback.

Design becomes even more interesting when it comes to managing more than one API. In those cases consistency also plays an important role.

Standardizing all your APIs to have consistent patterns becomes possible when you can measure those patterns. OpenAPI definitions is such a measurement.

## 1.8 *This book*

This book aims to help you understand how OpenAPI works, how it and the tooling can be used to design APIs, and how you can create advanced and very specific work flows for your team and organization.

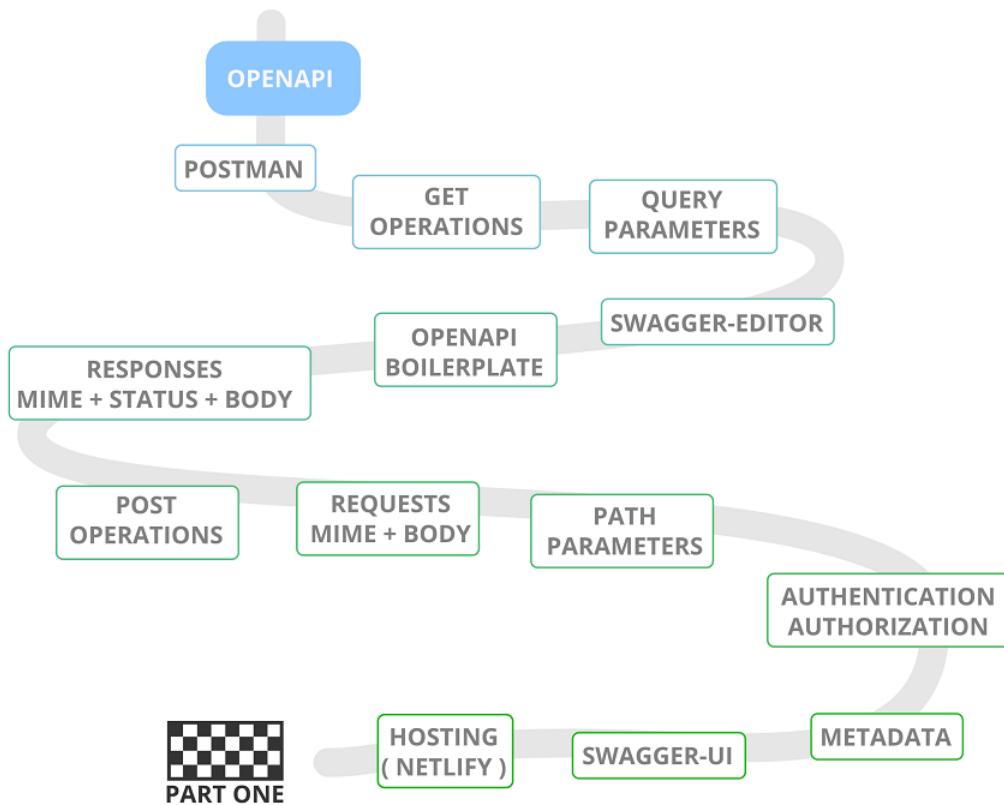
In my opinion, all efforts in OpenAPI are aimed at automating parts of your work flow thus freeing your team to accomplish more. The small upfront cost of describing APIs with OpenAPI is greatly offset by the power you can wield leveraging it, and the new opportunities it presents.

This book is broken down into three parts...

- Part one deals with OpenAPI literacy and introduces you to the syntax and structure of OpenAPI definitions. Giving you the ability to describe APIs. Throughout this part we document a contrived FarmStall API that is hosted online and is simple enough to easily grok ( ie: understand without knowing the details ).
- Part two deals in the design phase and how we'll use the tools to create a new API and iterate its design. We'll be extending our contrived API into a new one.
- Part three is a deep dive into some of more specific tools and work flows, including a look at how to build your own OpenAPI tooling and features.

## 1.9 *Part One*

Part one will have the following diagram to indicate where we are in the scheme of things...



**Figure 1.3 Where we are**

We'll progressively highlight those sections as we're about to learn them.

Onward!

## 1.10 Summary

- OpenAPI is a specification for describing HTTP-based APIs, most notably RESTful APIs.
- Swagger is a trademarked set of tools by SmartBear.
- Describing APIs into a definition (ie: YAML file) allows you to leverage tools to help automate a lot of API related processes.
- OpenAPI is useful for consumers, producers and API designers. Each can benefit from knowing and utilizing tools that consume OpenAPI definitions.
- This book will give you further understanding and a knowledge base of how to work with OpenAPI. To ultimately incorporate it into your team and organization work flows.

# Getting set up to make API Requests



## This chapter covers

- Introducing the FarmStall API and some of its business logic
- Introducing a tool to make HTTP requests — Postman
- Executing API requests and inspecting the responses

Our task in part one is to describe facets of an API called FarmStall.

FarmStall is an API designed specifically for this book, and was made to be as simple as possible. For those interested, the server was written in Go and you are more than welcome to inspect the source code at [github.com/ponelat/farmstall](https://github.com/ponelat/farmstall). The API is hosted online at [farmstall.ponelat.com/v1/](https://farmstall.ponelat.com/v1/) (v1 is specifically for part one).

Before we can describe this API, we'll need to understand how it works and to be able to make HTTP requests and inspect the responses.

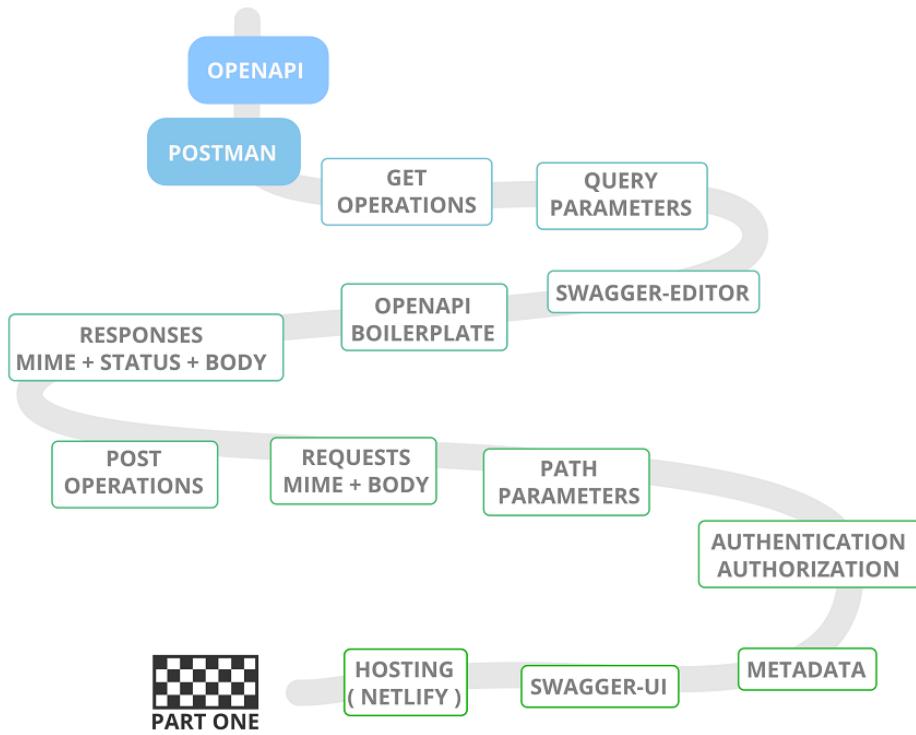
In this chapter we'll be looking at how to use a tool called Postman, to make HTTP requests against our API. We'll be verifying that we get "decent looking" responses without concerning ourselves too much with the details of those responses.

We also want to learn a bit about the business domain of the API so that we have an understanding of what we're doing. This will make it easier to describe later on.

What we'll be touching on

- Postman — [getpostman.com](https://getpostman.com)
- FarmStall API landing page — [farmstall.ponelat.com/](https://farmstall.ponelat.com/)
- FarmStall API — [farmstall.ponelat.com/v1/reviews](https://farmstall.ponelat.com/v1/reviews)

- (Source code) — [github.com/ponelat/farmstall](https://github.com/ponelat/farmstall)



**Figure 2.1 Where we are**

## 2.1 The problem

Our problem in this chapter is to discover and learn more about the FarmStall API. In particular we want to interact with it and confirm some basics, such as access and the ability to create requests and inspect the responses.

First an overview of the API itself...

### SIDE BAR

#### FarmStall API Overview

The FarmStall API is hosted on [farmstall.ponelat.com/v1](https://farmstall.ponelat.com/v1). The API's primary focus is to allow patrons of the Farmer's Market to write up reviews and give their feedback on their experiences. Users can submit anonymous reviews, which include a message and a rating (1 to 5 inclusive). Users can also sign up to create reviews that will then be associated with them. Once signed up, they can get a "UserToken" to create reviews with their user ID.

### 2.1.1 First two operations of FarmStall API

The FarmStall API has several operations and we're going to try the following two...

- To get a list of public reviews you can use `GET /reviews`, you can filter reviews by their rating using the query parameter `maxRating`.

- To submit a new review you can use `POST /reviews`, the body of this request will include `message` and `rating` fields.

**Table 2.1 API Operations Table**

Method	URI	Query Params	Body	Response
GET	/reviews	maxRating (1-5)	N/A	?
POST	/reviews	N/A	{message: "Was good.", rating: 5}	?

From the above information we can gather enough detailed information to create our first two requests, including where the API is hosted ([farmstall.ponelat.com/v1](http://farmstall.ponelat.com/v1)) and the details of each operation. You will note in the above table that the Response column has question marks, our task in this chapter is to fill out those question marks by personally verifying what responses come back from each operation.

So how do we make these HTTP requests? Fortunately for API folks there are numerous ways that we can make these HTTP requests. From the brave who'd try their hand using `telnet`, the practical ones who would use `curl`, to the individuals who have whole software suites with bells, whistles and bunches of utilities.

**NOTE**

**For the brave**

While no-one really writes HTTP requests by hand, I encourage you to do so. It is actually quite satisfying when you form an HTTP request completely from scratch and get a response. I'll stick a section at the end of this chapter (not a biggie) to craft a request using the low level tools: `telnet` (for HTTP) and `openssl` (for HTTPS). Watch out for that!

There is no hard requirement for you to use any particular tool for making HTTP requests, and we've tried to structure this book in a way as to steer clear where ever possible from requiring those tools. However I'd still encourage you to try out the suggested tools as-is, to more closely follow along. Perhaps there are features that you can incorporate back into your own arsenal!

Without further ado let's learn about Postman.

## 2.2 Getting set up with Postman

Postman is a general HTTP tool that has a pleasant user interface and is suitable for beginners and professionals alike. It can be a bit overwhelming if we look at all the text boxes, but it has got good bones.

It was chosen as a tool for this book predominately because of its popularity (so that you're not stuck using an esoteric tool like some that I use) and because of the many features it provides. Some you'll find useful and others you might find inspirational.

Lets go and get it!

### 2.2.1 Installing Postman

In order to use Postman we need to install it. So go ahead and download then install it from [getpostman.com](http://getpostman.com) (or directly from [www.getpostman.com/downloads/](http://www.getpostman.com/downloads/)). It has support for Microsoft Windows, macOS and most Linux distributions.

At the time of writing Postman was on version 6.7.2. Your version may look and act a little differently depending on how much the authors of Postman change it in the interim. The UI has been pretty stable so it should look similar to the screen-shots in this chapter.

Also, during the period when this was being written you did not need to create an account with Postman in order to use it, although they will encourage you to do so. There are free and paid-for plans, as well as the option to NOT create any account at all. For this chapter we'll assume you *didn't* create an account so that we'll only use the features that are available to non registered users, which should be ample for our purposes.

Go ahead and install Postman, I'll wait :)

## 2.3 FarmStall API

We're going to make requests against our FarmStall API, but before we do a quick note...

The data in the FarmStall API will persist but only for a day or two, so that its doesn't start to overflow with too much data (its only a little service!).

It was designed specifically for this book to help guide us in describing an existing API, and it wasn't designed to be robust enough to handle production level data. If you add a review one day and don't see it the next, you're not going crazy — the API is just cleaning up!

To get to grips with the basics of creating an HTTP request we're going to execute two of them. A GET request with query parameters, and a POST request with a header parameter. We'll progressively examine the details of these operations as we go along, for now we're going to focus more on the practical side of making requests and less on what the operations are actually doing.

Lets start by getting a list of reviews from the API.

## 2.4 Our first request!

From the two operations described in the beginning of this chapter we're going to start by getting the list of reviews. It has the following details...

**Table 2.2 GET /reviews**

Method	URI	Query Params	Body	Response
GET	/reviews	maxRating (1-5)	N/A	?

From it we know that it is a `GET` method and that it has at least one *query* parameter called `maxRating`. This parameter accepts a number from 1 to 5 inclusive.

Operations are often described relative to where the server is hosted and this API has a base URL of [farmstall.ponelat.com/v1](http://farmstall.ponelat.com/v1) so the URL for `GET /reviews` becomes:

[farmstall.ponelat.com/v1/reviews](http://farmstall.ponelat.com/v1/reviews)

If we add in the query parameter it'll form our final URL of:

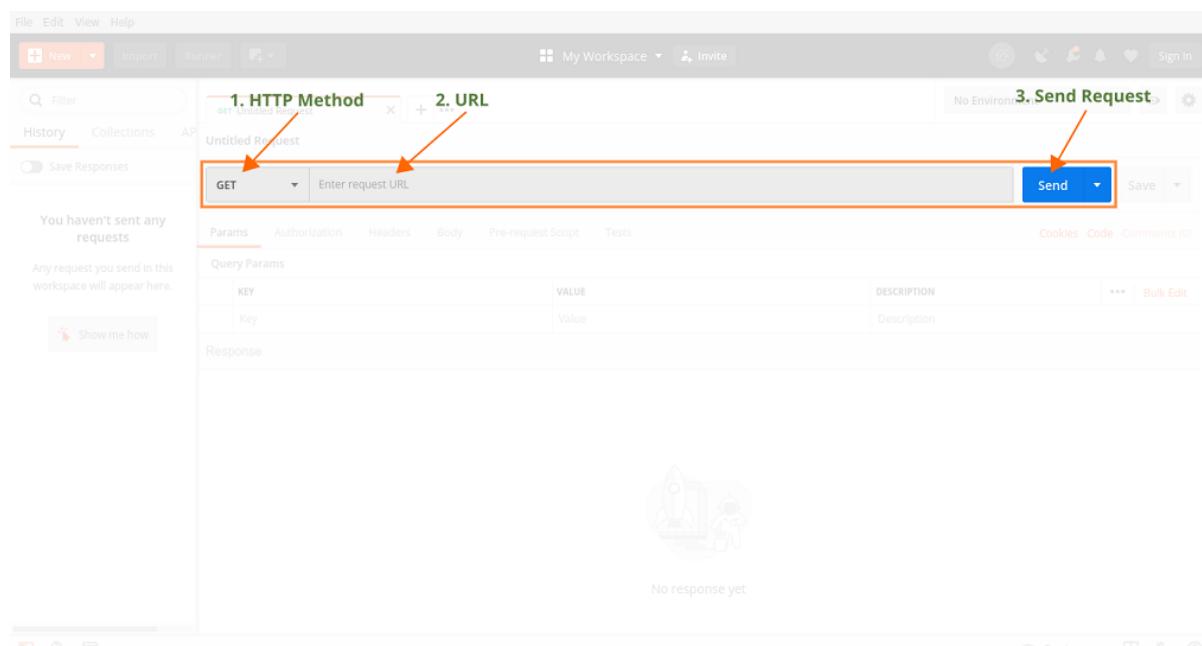
[farmstall.ponelat.com/v1/reviews?maxRating=5](http://farmstall.ponelat.com/v1/reviews?maxRating=5)

Armed with a URL and a method we have enough to execute this particular request — time to use Postman.

#### 2.4.1 Forming the request in Postman

If you haven't done so already, go ahead and start up Postman. We want to get to the main page so if you're confronted by a welcome modal feel free to dismiss it. You can generally dismiss modals by clicking on the small "x" at the top right side of the modal.

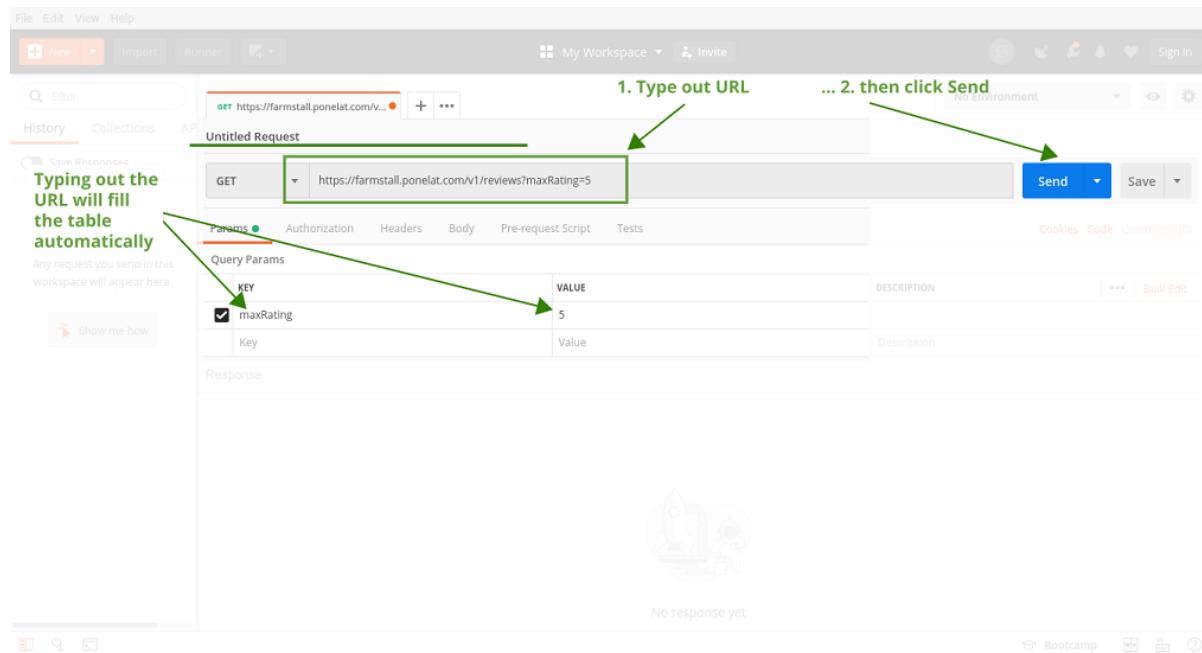
Here are the key areas in the main page that we are interested in (for our `GET` request):



**Figure 2.2 Postman - the key areas**

1. HTTP method drop-down This selects the method to use. The default is likely GET but you can select it again if it isn't already.
2. URL input box This is where we will put the URL of the endpoint we want to make the request against.
3. Send request button The button that executes the request.

To create a request against our endpoint we need to add the URL into the URL input box and hit the send button after that, so go ahead and type it out. You should end up with the following...



**Figure 2.3 Postman - GET /reviews**

After hitting send, you should see a chunk of JSON data in the Response Body section. This is the result of us executing the request. If you see this JSON data — congratulations, you've successfully executed a request!

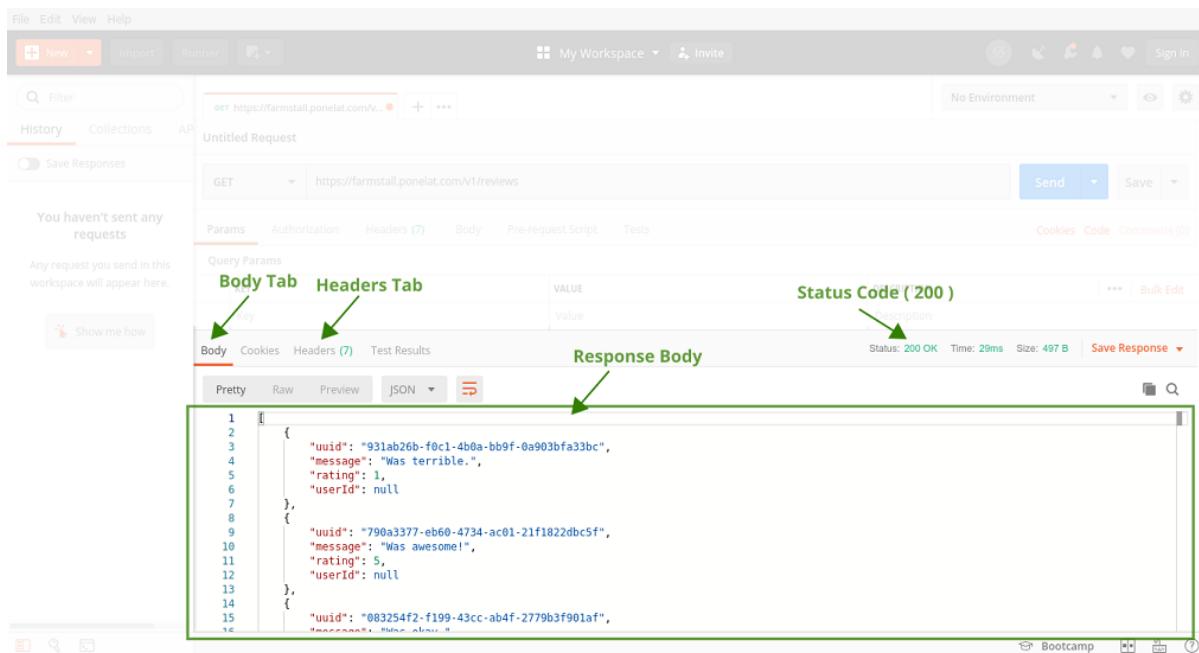


Figure 2.4 Postman - GET /reviews Response

**NOTE**

If for some reason you encountered an error in the response, that's okay. It could be that there is a typo, that the server is down/misbehaving or some other unforeseen reason. If you're happy you wrote the request correctly, that is enough for now. We'll have more examples later on in the chapter that we can test against.

## 2.4.2 Verification

Wonderful... we should now see some response data from our request. Confirming that our API works and that we can reach it. Our response data should look similar to the following:

### Listing 2.1 Data received from GET /reviews

```
[
  {
    "message": "Was awesome.",
    "rating": 5,
    "uuid": "16f5e7e1-b581-4ca4-8af2-8dead5894869",
    "userId": null
  },
  {
    "message": "Was aweful",
    ...
  }
]
```

Now we can fill in the question mark in our operation table, as we know what the response data is! Later on we'll need to describe this data, for now we're just satisfied that the operation works and does indeed return some data.

Now let's try creating a new review by executing a POST request.

## 2.5 Adding a Review to the FarmStall API

Looking at the FarmStall documentation table we can see that the operation we're after is `POST /reviews`. It takes no query parameters but it does require a body. In this case the response isn't the most interesting part of the operation, what is more interesting is that we are adding data into the API.

**Table 2.3 GET /reviews**

Method	URI	Query Params	Body	Response
POST	/reviews	N/A	{message: "Was good.", rating: 5}	?

One of the key differences between POST and GET is the request body. One could conceivably send data as query parameters, but they impose too many limitations. From size limitations of query parameters, to the fact that they cannot contain binary data. A request body doesn't have these limitations, the size is limited only by practicality and the body can contain binary data.

**NOTE**

Another interesting benefit of sending data in the body is security. Query parameters are part of the URL, and as such, often get logged by servers and proxies. If you were to send secret data as query parameters, there is a good chance it will be recorded somewhere between your client and the server. Whereas bodies are most often not processed by proxies, nor are they typically logged.

In the `POST /reviews` operation, the body is required to be in JSON format. In this format we can see it's an object that has two fields.

They are `message` and `rating`.

- `message` will be a string, and is the feedback for the Market.
- `rating` will be a number, between 1 and 5 inclusive. Which will indicate our general experience where 1 is the worst and 5 is the best.

Let us build the request...

**Listing 2.2 JSON Request body for POST /review**

```
{
  "message": "Was pretty good.",
  "rating": 4
}
```

As we previously did with the `GET /reviews` operation, we need to combine the URI with the base URL of the server to form the following complete URL:

[farmstall.ponleat.com/v1/reviews](http://farmstall.ponleat.com/v1/reviews)

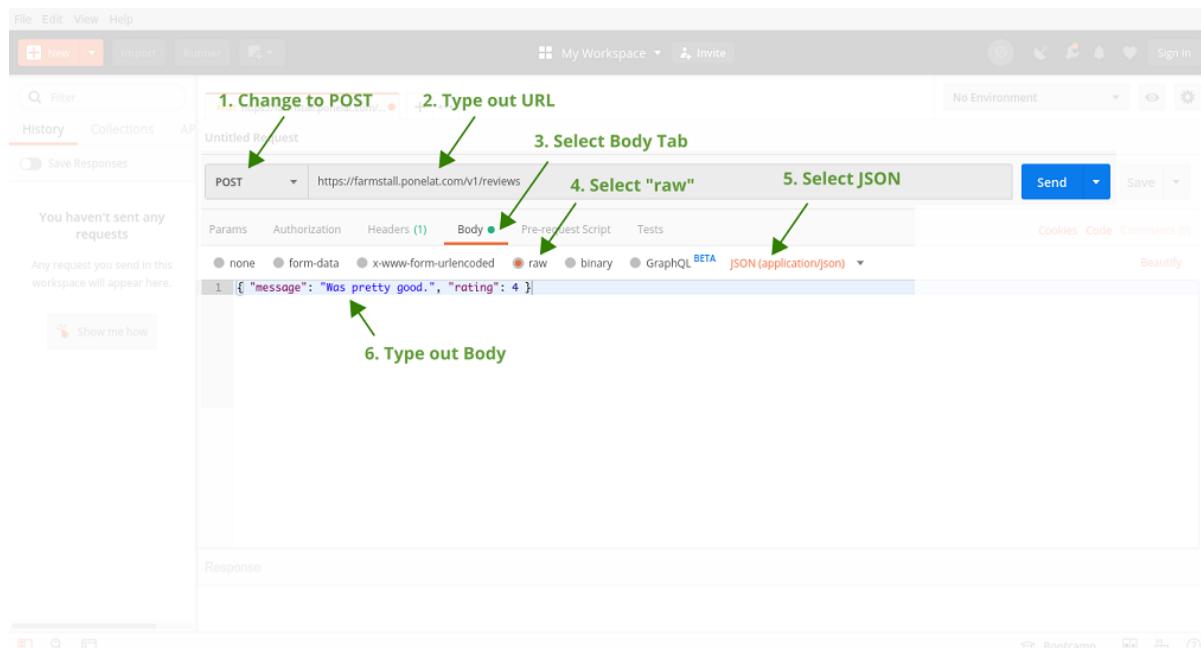
Good stuff, now let's go through what we need to do in Postman in order to execute this request.

We'll need to do the following:

1. Change the method to be **POST**
2. Type out the URL to be **farmstall.ponelat.com/v1/reviews** (same as the GET /review)
3. Select the Body Tab, so that we can type out the body
4. Type out the JSON body that includes the **message** and **rating** fields
5. Ensure that the content type is set to **JSON** (or **application/json**). Depends on the UI (User Interface)

Number (5) will set a special header called `Content-Type` which indicates to the server which media type our data is in (more on that later). Since the UI of Postman could change, we'll need to double check that it is set, and we will also see that other headers can be easily added to future requests.

First go ahead and make the changes in Postman so that you see the following (allowing for minor differences):



**Figure 2.5 Postman - POST /reviews Was Good.**

Now confirm that the `Content-Type` header was set and that its value is `application/json` (the official media type for JSON data).

There is a tab in the Postman main page specifically for headers. When you typed out your body there was also a dropdown for selecting the content type of the data written. Postman will create the header for us based on this value, but we need to be sure. So go ahead, click on the header tab and ensure it looks similar to the following. `Content-Type: application/json`

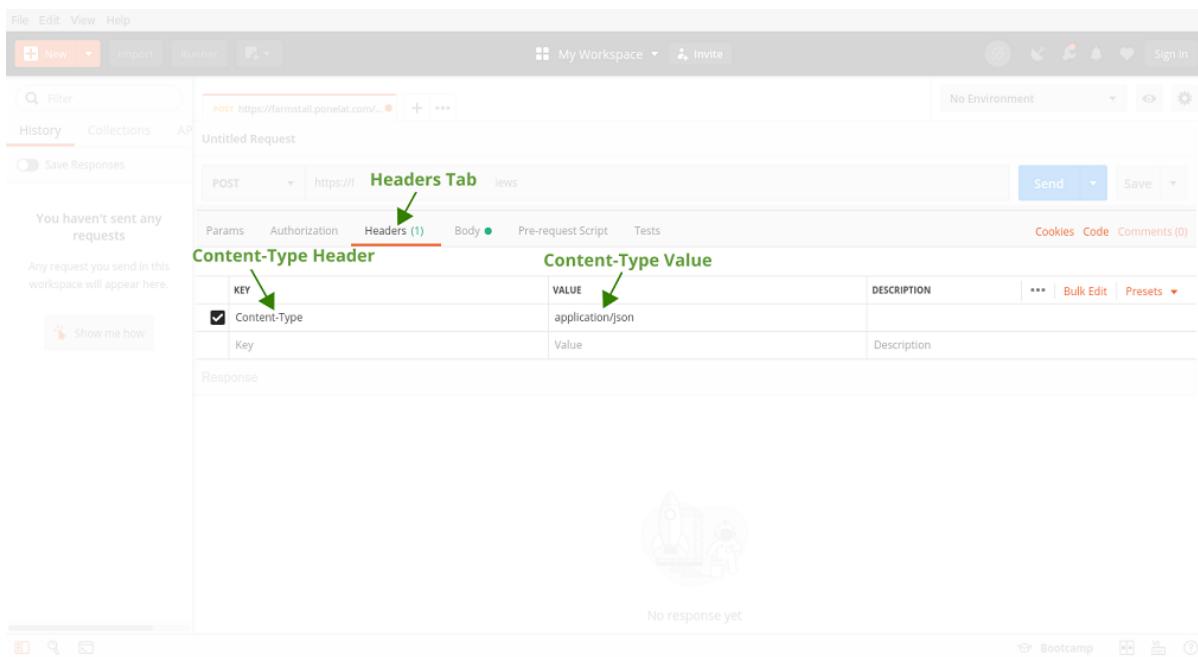


Figure 2.6 Postman - POST /reviews Headers Tab

**NOTE**

**Executing Requests**

Sending, executing, calling, requesting. There are a lot of ways to term creating a request and executing it. All of these have the same meaning and I just wanted to illustrate how you might often find these words used interchangeably. It'll depend on what feels more natural to write down, if in doubt, use the term

Great! Our request is ready to send. After clicking on the Send button the new review should be created. To confirm that it was created you should see the following in the Response Body section of Postman, as well as the Status code 201 (which is for "Resource Created" or just "Created").

**Listing 2.4 JSON response body from POST /review**

```
{
  "message": "Was pretty good.",
  "rating": 4,
  "uuid": "16f5e7e1-b581-4ca4-8af2-8dead5894869", ①
  "userId": null
}
```

① Note that this value will be different (its random)

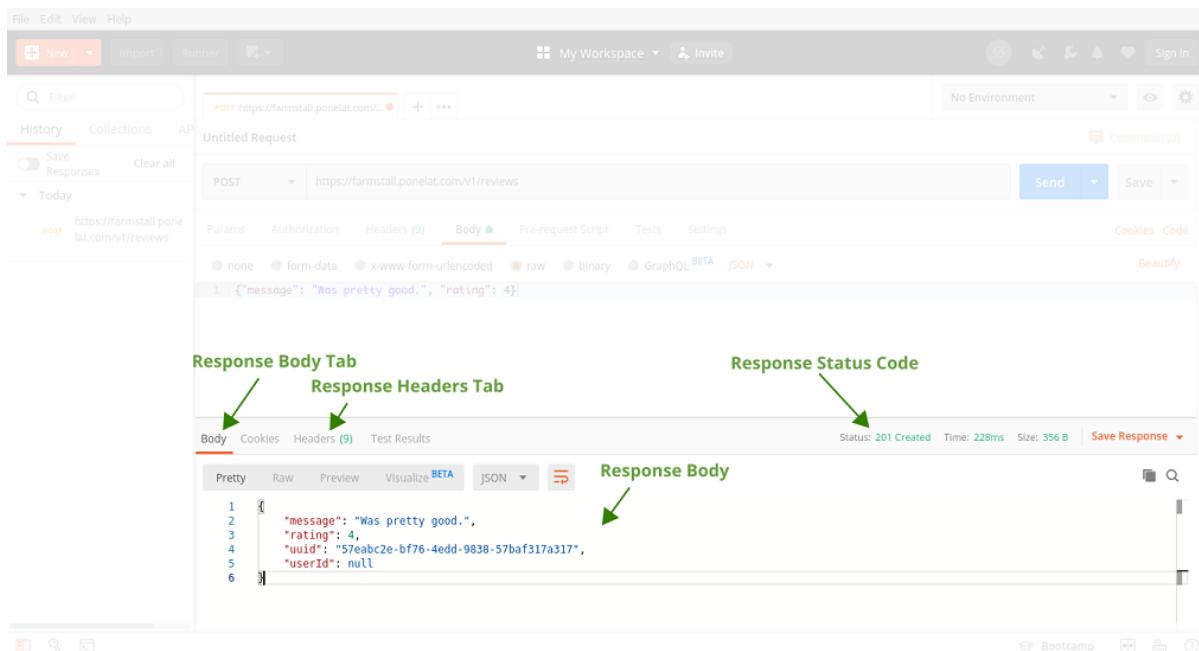


Figure 2.7 Postman - POST /review Response

### 2.5.1 Verification

What have we accomplished so far? We've successfully executed two requests. One for getting the list of reviews, and another for creating a new review. Our operations details table has now been verified — and is telling the truth! For now, just seeing reasonable data is enough. Soon we're going to describe these operations in a way that clarifies what is possible, without actually executing requests and making assumptions about the data.

Now that we're able to make basic requests with Postman, we can have a little fun and practice with more APIs!

## 2.6 Practice

Now for a bit of practice! The following HTTP requests are a short list of APIs that have fun, interesting or perhaps even useful responses. Given the nature of the world, it's entirely possible that some (hopefully not all) of these APIs may become unavailable, or worse, change their interfaces so that these requests will fail! The latter is something we hope to avoid when designing our own APIs.

The ones chosen here were considered to be stable enough at time of print but we hope you understand that they might change. We've put the expected results at the end of this chapter so that you can compare with your own.

Without further ado here are the requests...

## 2.6.1 Fun API requests to make

A little fun with cat (and other animal) facts.

### Listing 2.5 Cat Facts API

```
Documentation: https://alexwohlbruck.github.io/cat-facts/docs/
Server: https://cat-fact.herokuapp.com

GET /facts?animal_type=cat,horse
GET /facts/random

Example response:
{
  "_id": "58e008780aac31001185ed05",
  "user": "58e007480aac31001185ecef",
  "text": "Owning a cat can reduce the risk of stroke and heart attack by a third.",
  "__v": 0,
  "updatedAt": "2019-01-19T21:20:01.811Z",
  "createdAt": "2018-03-29T20:20:03.844Z",
  "deleted": false,
  "type": "cat",
  "source": "user",
  "used": false
}
```

For those times you need a random image to put onto a website prototype.

### Listing 2.6 LoremPixel API

```
Documentation: http://lorempixel.com/
Server: http://lorempixel.com

GET /{width}/{height}/{category}/{text}

Example: GET /400/200/sports/Dummy-Text
Example response:
<See image>
```



A look at DuckDuckGo's search engine API.

## Listing 2.7 DuckDuckGo API

```
Documentation: https://api.duckduckgo.com/api
Server: https://api.duckduckgo.com

GET /?q={query}&format=json&pretty=1

Example: /?q=cats&format=json&pretty=1
Example response:
{
  "Abstract" : "",
  "ImageWidth" : 0,
  "AbstractSource" : "Wikipedia",
  "meta" : {
    "src_domain" : "en.wikipedia.org",
    "blockgroup" : null,
    "is_stackexchange" : null,
    "dev_milestone" : "live",
    ...
  }
  <a bit too large to print>
```

And because the world needs more "Pirate speak", someone went and made an API for that too!

## Listing 2.8 Pirate Talk API

```
Documentation: https://funtranslations.com/api/pirate
API: https://api.funtranslations.com

POST /translate/pirate.json?text={text}
Example:

POST /translate/pirate.json?text=Hello%20Good%20Sir
Note: %20 is URL encoding for spaces

Example response:
{
  "success": {
    "total": 1
  },
  "contents": {
    "translated": "Ahoy Good matey",
    "text": "Hello Good Sir",
    "translation": "pirate"
  }
}
```

## 2.7 HTTP For the brave

Bonus section! Only proceed if you feel brave...

As promised here is a section on how to craft an HTTP request completely from scratch. If you're feeling less than brave feel free to give this section a skip. You may spontaneously start sporting a neck beard if you continue... you've been warned!

*PS: There is a bit of low level jargon in this section.*

There are two utilities you can use to open a TCP connection (a pipe you can read and write data into) suitable for HTTP requests. The first is `telnet` which is available on most systems, and

openssl which is typically found on 'nix (Linux, macOS, etc) systems. openssl can be used to open a TCP connection over SSL/TLS, which is necessary for HTTPS only servers.

*I'm going to assume a 'nix system here as I haven't tried these commands on Windows*

To open the connection...

### **Listing 2.9 Opening a TCP connection**

```
$ telnet farmstall.ponelat.com 80
# Or for HTTPS sites...
$ openssl s_client -quiet -connect farmstall.ponelat.com:443
```

After running either of those commands, your terminal will pause and wait for you to enter in text to send to the server. By typing the following we can get a list of reviews...

### **Listing 2.10 GET /v1/reviews over TCP**

```
GET /v1/reviews HTTP/1.1 <enter> ①
Host: farmstall.ponelat.com <enter> ②
<enter> ③
```

- ① This is the status line, it includes the method, the URI, and the version of HTTP protocol.
- ② The host header is important as a lot of servers use a reverse proxy, and can determine which server you are asking for accordingly.
- ③ A blank line to separate the headers from the body section.

Congratulations! You should get back a response including headers and a body.

Here is an openssl example with response...

## Listing 2.11 OpenSSL connect with response

```
$ openssl s_client -quiet -connect farmstall.ponelat.com:443 ①
depth=0 CN = letsencrypt-nginx-proxy-companion ②
verify error:num=18:self signed certificate
verify return:1
depth=0 CN = letsencrypt-nginx-proxy-companion
verify return:1
GET /v1/reviews HTTP/1.1 ③
Host: farmstall.ponelat.com

HTTP/1.1 200 OK ④
Server: nginx/1.17.5
Date: Thu, 14 Nov 2019 09:24:50 GMT
Content-Type: application/json
Content-Length: 465
Connection: keep-alive
Vary: Origin
X-Ratelimit-Limit: 36
X-Ratelimit-Remaining: 35
X-Ratelimit-Reset: 1573723550
[{"uuid": "16f5e7e1-b581-4ca4-8af2-8dead5894869", "message": "Was okay.", "rating": 3, "userId": ""}, {"uuid": "92dalefe-a0ab-40a5-bbb9-466e7c32e96d", "message": "Was terrible.", "rating": 1, "userId": ""}, {"uuid": "5ca80db6-82f7-41a6-8c54-19fb7db77a31", "message": "hello", "rating": 5, "userId": ""}, {"uuid": "13151e0e-f3e7-4f33-ad5b-d4bda9adf496", "message": "hello", "rating": 5, "userId": ""}, {"uuid": "e4d99a5c-5883-43e7-8133-bb05bf34d0d9", "message": "Was awesome!", "rating": 5, "userId": ""}]
```

- ① The openssl command to open up the connection
- ② Some connection details (not typed)
- ③ Typing out the HTTP request...
- ④ The start of the response (not typed)

Now to create a new review (and add a body to our request). After creating a connection using `telnet` or `openssl` type out the following...

## Listing 2.12 Creating a new review over TCP

```
POST /v1/reviews HTTP/1.1 <enter> ①
Host: farmstall.ponelat.com <enter>
Content-Length: 37 <enter> ②
Content-Type: application/json <enter> ③
<enter> ④
{"message": "neckbeard", "rating": 5} <enter> ⑤
```

- ① Use the `POST` method.
- ② Need to indicate the size of the body (I counted).
- ③ The media type of our payload.
- ④ A blank line to separate header section from body.
- ⑤ Our body (all 37 characters).

As soon as you hit that last `<enter>` you should get a response. Note that if you increase the "Content-Length" to a larger number your response will only be returned after you hit enter

multiple times.

That is the HTTP protocol, of which you wrote a GET and POST request *by hand!*

That is brave.

## 2.8 Summary

- You learned a little about our example API, FarmStall API and how it works
- After installing and setting up Postman you then used it to make requests against the FarmStall API.
- Those requests proved that we can interact with the API, and will allow us to test assumptions about it.
- You were given some example APIs in order to practice this new skill.
- As a bonus you can use `telnet` for writing out HTTP requests by hand, and `openssl` for doing the same but over SSL/TLS.

# 3

## *Our first taste of OpenAPI definitions*

### This chapter covers

- Informal vs formal descriptions.
- Learning about the OpenAPI specification.
- Learning about YAML.
- Describing our first GET operation.

OpenAPI definitions are at the heart of automating our API workflows. They are the slices of bread in a sandwich shop, the fruit on a breakfast buffet, and the vanilla in vanilla muffins. Which is my way of saying that they're important.

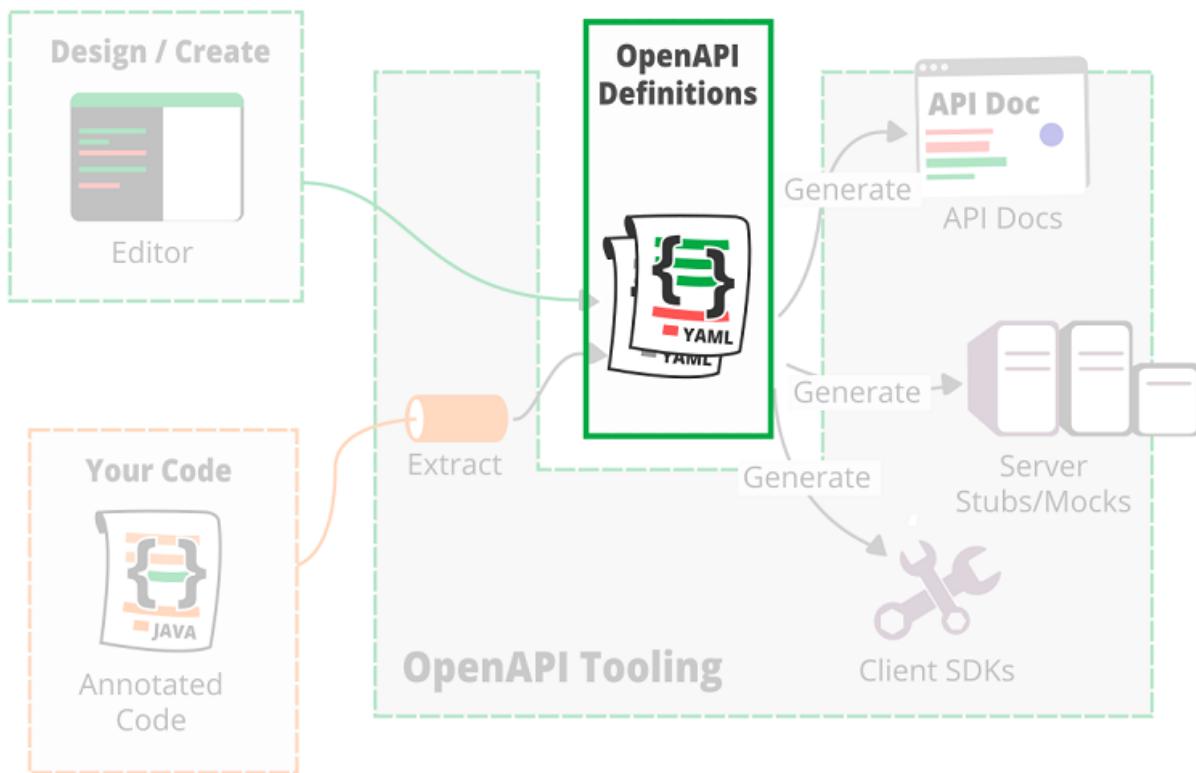
When we formally describe an API we're turning the idea of that API into some data which we call a *definition*. It differs from an informal description which has no strict rules or syntactical structure. Informal descriptions are akin to documentation found on websites, great for humans to read but hard for machines to decipher.

Once an API has been described into a definition it can be used by tools (**cough** machines), fueling different parts of the API ecosystem such as API request validation, code stubs, documentation and more.

*PS: Machines shouldn't have too much power if movies have taught us anything! However, they should be able to help us out just a little!*

**WARNING** API Definitions both excite and propel me, and as such, I will continue to wax lyrical on the merits of API definitions. You've been warned!

If we were to look at where this fits in the scale of things, we could view it as such:

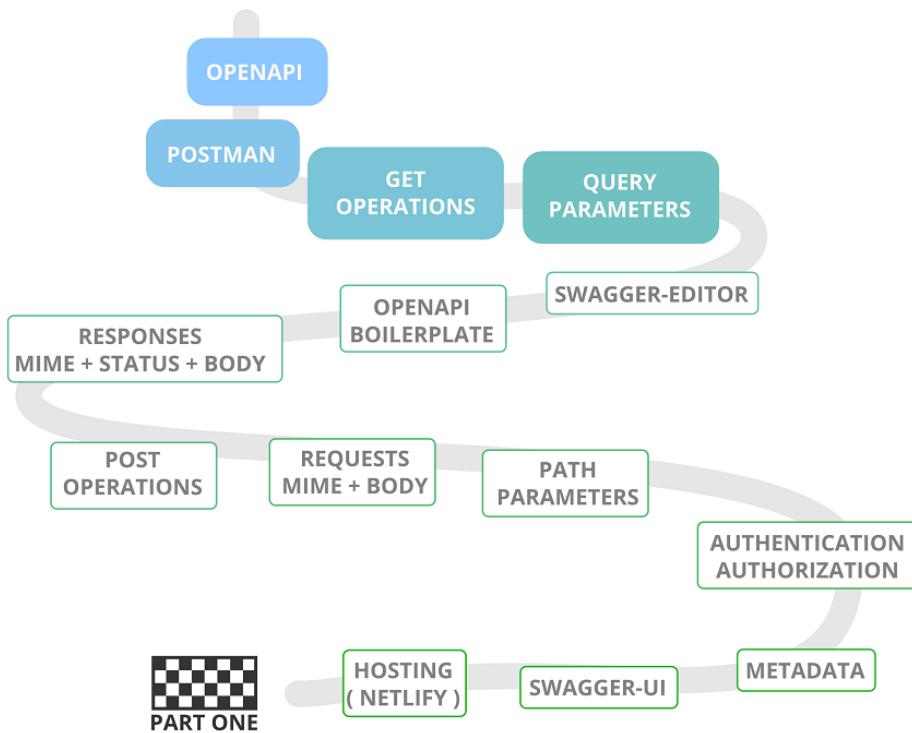


**Figure 3.1 Mental model of OpenAPI/Swagger, showing where definitions fit in.**

In this chapter we're going to write a formal definition of a single operation from the FarmStall API. To get there we'll need to understand what that operation requires, then will take a look at YAML, and finally we'll write an OpenAPI definition fragment (ie: not a complete OpenAPI definition).

What we'll be touching on

- FarmStall API - [farmstall.ponelat.com/v1](http://farmstall.ponelat.com/v1)
- YAML - [yaml.org/](http://yaml.org/)
- OpenAPI Specification - [github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md)



**Figure 3.2 Where we are**

### 3.1 The problem

In this chapter we want to formally describe a single operation from the FarmStall API, based on details of that operation. We'll supply the details and build up to that fragment. We're not going to add the boilerplate of an OpenAPI definition, so this definition won't pass validation. We'll soon look at that.

At the end of it we'll have an OpenAPI fragment that looks like this:

#### **Listing 3.1 The OpenAPI fragment we'll describe**

```
/reviews:
  get:
    description: Get a list of reviews
    parameters:
      - name: maxRating
        in: query
        schema:
          type: number
    responses:
      200:
        description: A list of reviews
```

The details of that operation are as follows...

## Listing 3.2 Summary of GET /reviews

```
`GET /reviews`
```

Returns a list of reviews in the FarmStall API.

The list can be filtered down by the `maxRating` query parameter. Each review is an object with at least

**Table 3.1 Parameters of GET /reviews**

Param	Desc.	Where	Type	Notes
maxRating	reviews below this rating	query	number	1-5 inclusive

In addition to the critical parts of this operation such as the method (GET) and the URI (/reviews), we'll also be describing a parameter `maxRating`. What we *won't* be describing is the response body.

**NOTE**

API descriptions fall on a scale of vague/useless to pedantically-precise.

## 3.2 Introducing the OpenAPI specification

Formal descriptions need a standard or specification. A source of truth for *how to describe a thing*.

The OpenAPI specification is a formal way for describing RESTful or HTTP based APIs, which is tantamount to a template. A set of rules and constraints to show you *how you could* describe an API.

The trick is, *if you do follow the template*, then software ( and people ) will be able to make use of your description using generally available tools. They'll not only understand what you're describing but will also be able to use it as part of their system with far less effort than if it were described using a bespoke specification.

Let's take a look at a preview of what a fragment ( ie: not complete ) *OpenAPI Definition* looks like...

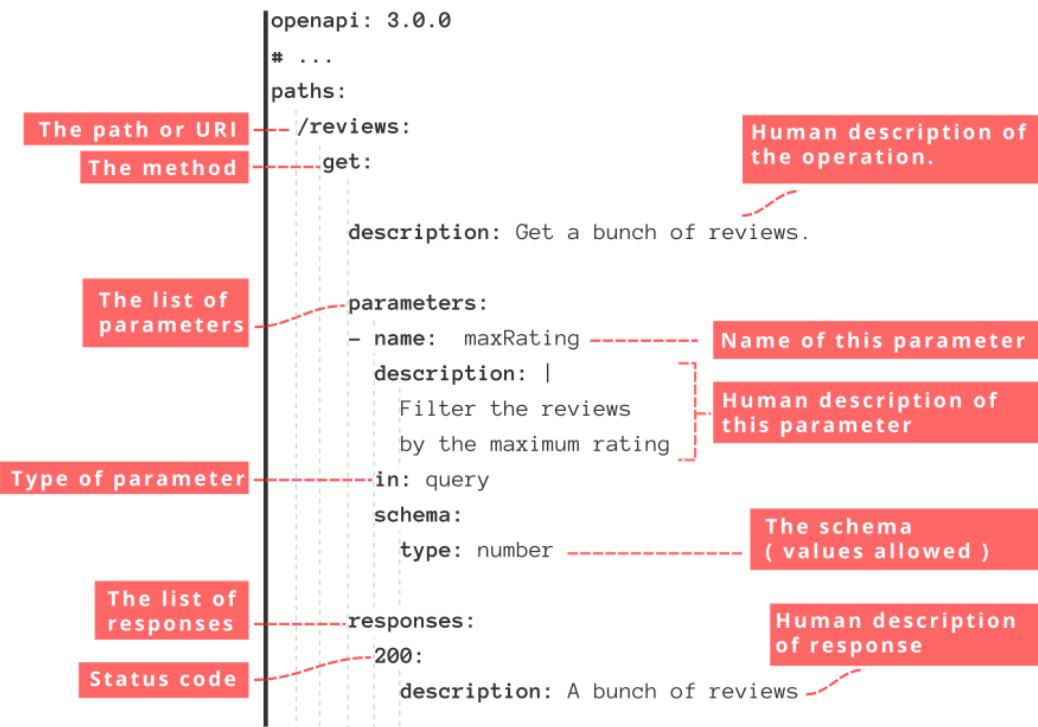


Figure 3.3 An example OpenAPI document, with some labels

First of all, what is up with the indentation? What are those dashes... and those colons?

That is YAML.

If you know JSON it'll be pretty straightforward to understand, and I promise we'll take a look at it in a bit. For now I just want you to take a look at the *gist* of an OpenAPI definition, to get a better feel for what it looks like.

**NOTE**

An OpenAPI Definition is a document(s) that conforms to the OpenAPI specification. If it breaks a rule set out by the OpenAPI specification, its said to be "invalid".

### 3.3 A quick refresher on YAML

To conform to the OpenAPI Specification we need a data format to write in. You could use JSON, but when you try to write in JSON one soon learns that... well, it can be painful.

As it is becoming a trend (kubernetes, openapi, ansible, etc), YAML is a popular alternative to JSON.

Particularly for those cases where you might be required to write pieces of it by hand. As it is a little gentler to write than JSON. YAML has far less restrictions than JSON, with several ways of expressing the same piece of data (eg: strings can be quoted or unquoted, and trailing commas

are allowed).

One of YAML's features is its support for "flow-types" which is what it calls the JSON-like objects {} and arrays []. With this support it becomes a full superset of JSON, which is awesome considering all JSON documents are legal YAML documents. Hurrah! <sup>3</sup>

JSON is arguably the standard when it comes to Web communication. It is the lowest common denominator of data types in most programming languages, it's compact and basic enough that most programmers can grok it (ie: *understand it intuitively*) pretty quickly, and the grammar of JSON was simple enough to fit on a business card!<sup>4</sup>

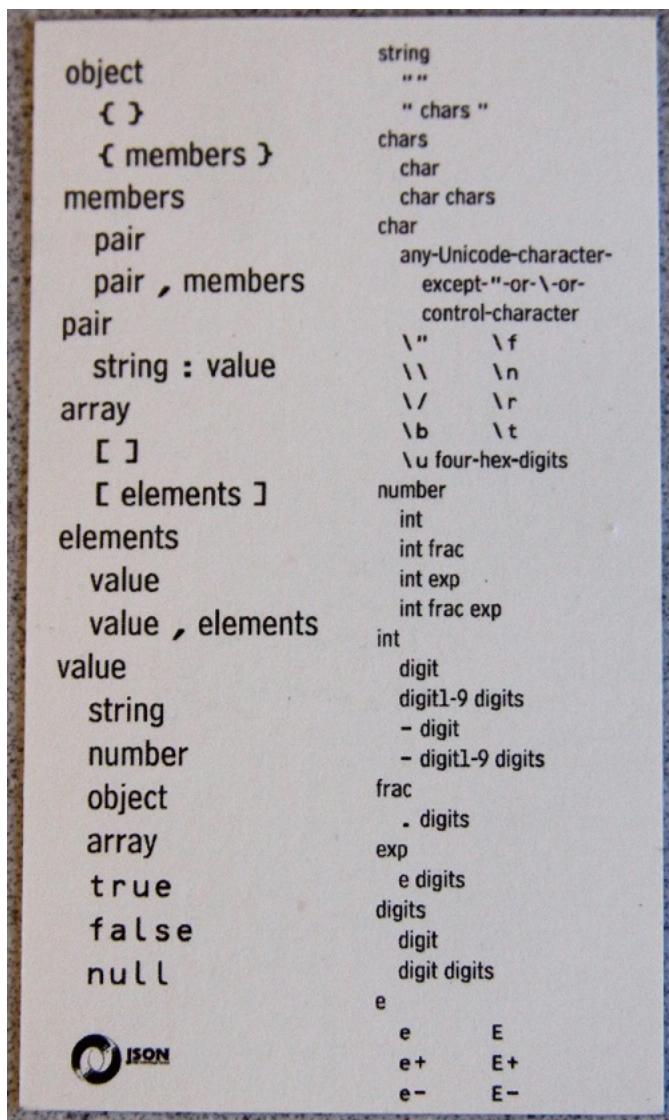


Figure 3.4 Douglas Crawford's JSON Grammar on a business card.

### 3.3.1 From JSON to YAML

YAML originally stood for Yet-Another-Markup-Language, but they changed it to YAML-Ain't-Markup-Language. I guess they really wanted a recursive acronym, although the motivation for the changed acronym was to emphasise the data aspect and less on the mark up side of it (for human-readable documents).

The YAML specification itself is quite large and comes in different flavours (or schemas). OpenAPI focuses on the bare minimum for its needs, which is the JSON Schema of YAML version 1.2. [yaml.org/spec/1.2/spec.html](https://yaml.org/spec/1.2/spec.html).

All this talk of schemas and specification can be daunting. OpenAPI documents are quite easy to work with as they're pretty much a prettier version of JSON.

*PS: OpenAPI can support JSON documents directly, but we'll only be referring to YAML.*

So what does YAML look and feel like?

#### Listing 3.3 A taste of YAML

```
SomeNumber: 1
SomeString: hello over there! ①
IsSomething: true
# Some Comment ②
SomeObject:
  SomeKey: Some string value ③
  SomeNestedObject:
    With: A nested key/value pair ( that is quite indented! ) ④
AList:
- a string
- another string
SomeOldSchoolJSONObject: { one: 1, two: 2 } ⑤
SomeOldSchoolJSONArray: [ "one", 'two', three ] ⑤
OneTypeOfMultiLineString: |
  hello over there,
  this is a multiline string!
```

- ① Strings don't need to be wrapped in quotes, but they can be.
- ② YAML supports comments, yay!
- ③ YAML uses indentation to nest object and arrays somewhat like python uses indentation. It doesn't matter how many spaces or tabs you use as indentation (I like two spaces... fight me!), as long as you're consistent then YAML parsers will be happy.
- ④ ...and you only have to be consistent *within* the scope (ie: map or sequence).
- ⑤ YAML is a superset of JSON so you can stick pieces of JSON where ever it feels natural.
- ⑥ YAML supports multi-line strings, although there are many different variants. See: [yaml-multiline.info/](https://yaml-multiline.info/) to get more familiar. Different types to support different newline handling and trimming.

For comparision, here is the same document but in JSON format...

#### **Listing 3.4 That same taste in JSON**

```
{
  "SomeNumber": 1,
  "SomeString": "hello over there!",
  "IsSomething": true,
  "SomeObject": {
    "SomeKey": "Some string value",
    "SomeNestedObject": {
      "With": "A nested key/value pair"
    }
  },
  "AList": [
    "a string",
    "another string"
  ],
  "SomeOldSchoolJSONObject": {
    "one": 1,
    "two": 2
  },
  "SomeOldSchoolJSONArray": [
    "one",
    "two",
    "three"
  ],
  "OneTypeOfMultiLineString": "hello over there,\nthis is a multiline string!\n"
}
```

As you can see its quite similar to JSON, and as OpenAPI only supports the data types that are in JSON... the two can be seen as interchangeable according to OpenAPI parsers. While YAML supports a multitude of more advanced features well beyond what JSON can do (advanced/custom data type, anchors, etc), those advanced features aren't interesting for our purposes as they don't limit the descriptive power of YAML.

If you'd like to find out more about YAML and its more flavourful features go take a look at its homepage: [yaml.org/](http://yaml.org/).

With YAML we can write data. That alone is quite a powerful concept but we're after bigger fish — OpenAPI uses YAML as its platform to describe APIs.

And we want OpenAPI...

### **3.4 Describing our first operation**

What do we want to know about our operation? In this chapter we want to know enough to make a request. With our informal description, we know the critical info.

- We know the Path... `/reviews`.
- We know the Method... `GET`.
- We know that this Operation returns a list of reviews.

Let's write some OpenAPI!

### Listing 3.5 The bare bones of our first operation

```
/reviews: ①
  get: ②
    description: Gets a bunch of reviews. ③
    responses: ④
      200: ⑤
        description: A bunch of reviews ⑥
```

Interesting... those are the core details of the operation, described according to OpenAPI's specification.

Shall we break it down a bit more?

What we have is a fragment of an OpenAPI document... not a full one yet (ie: it pass validator), we'll be inserting this fragment into a more complete OpenAPI definition in the next chapter.

At this point we need a celebratory image, something that captures this moment. Unfortunately my artistic skills are not so profound as to match the moment, so I'll provide what I can...



Too much? Too soon? "What about the query parameter?" You may be asking and you're quite right to do so. We need to add `maxRating`...

## 3.5 Extending our first Operation

Building on top of our initial OpenAPI fragment we need to describe the query parameter. The `maxRating` parameter serves the purpose of filtering the reviews by `rating` up to (and including) `maxRating`'s value.

*The business analysts at FarmStall Inc will make good use of this parameter in their reports.*

Pulling up the table we have at the beginning of this chapter...

**Table 3.2 Parameters of GET /reviews**

Param	Desc.	Where	Type	Notes
maxRating	reviews below this rating	query	number	1-5 inclusive

We can glean the following...

- We know that it's a number 1-5 (inclusive).
- We know that it appears in the query string.
- We know that its called `maxRating`.

To describe this we would have to do the following...

### **Listing 3.6 The `maxRating` query parameter**

```
name: maxRating ①
description: Filter the reviews by the maximum rating ②
in: query ③
schema: ④
type: number ⑤
```

Okay so that's a little more involved, and I dare say a little more *OpenAPI-ish*.

This is another fragment, it doesn't stand on its own and as such we need to add it to the operation we described earlier. Yeah, our celebration was a little too soon!

Doing so involves copying our fragment, as detailed above, into its rightful place...

### **Listing 3.7 Adding the `maxRating` query parameter**

```
/reviews:
  get:
    description: Get a bunch of reviews.
    parameters: ①
      - name: maxRating ②
        description: Filter the reviews by the maximum rating
        in: query
        schema:
          type: number ③
    responses:
      200:
        description: A bunch of reviews
```

Here we have our original fragment, and in it we added a `parameters` field which is an array of parameters. The astute will notice the trailing dash - before the `name` field... indicating an array item that is an object ( `name`, `description`, `in` and `schema` are fields of that object ).

## 3.6 Summary

- The difference between formal and informal descriptions, is in whether they follow strict rules or follow a specification. A formal description can be more readily consumed by software, whereas informal descriptions cannot.
- OpenAPI is a formal description of RESTful APIs, and an OpenAPI definition is a YAML file that describes some API.
- YAML is an "easy to write" data language that is the base of OpenAPI definitions. It is a superset of JSON
- OpenAPI only supports the "JSON Scheme" flavour of YAML<sup>5</sup>, which means it only supports the data types that JSON supports and not more.
- By using OpenAPI it is possible to describe operations and their parameters. This is certainly not the only parts of an API that can be described, as will be seen in the chapters to follow.

# Using SwaggerEditor to write OpenAPI definitions

Writing OpenAPI definitions has a lot of nuance that most of us are not bothered to learn right away. This is the way of the developer to jump into a new technology and try to hack it out until it looks right and hopefully works. However, we often stumble and end up reading the documentation anyway, just enough to get the job done.

A better trend is to try and minimize the amount of documentation we actually need to know off-hand, by building tools to help guide our actions.

SwaggerEditor is one such tool for writing OpenAPI definitions. It is a web application that is hosted online at [editor.swagger.io](https://editor.swagger.io) or can be self-hosted and like a lot of Swagger tools it is open source. The web application contains both a text editor, and a panel showing generated documentation. The documentation pane will show us the results of what we type, giving us immediate feedback, and a great affirmation that we typed the right things. We also get validation on the definition, which means when we type something incorrectly it'll shout at us and (hopefully) give us insight into fixing it.

In the previous chapter we looked at describing a single operation (`GET /reviews`) using OpenAPI, but it was not a complete definition, only a component.

In this chapter we're going to use SwaggerEditor to create a valid, if small, OpenAPI definition, by writing the necessary boilerplate that goes into making a fully valid OpenAPI definition. After that we will be adding in the description of `GET /reviews` from the previous chapter.

At the end of this chapter we'll have our very first OpenAPI definition for the FarmStall API!

What we'll be touching on

- Introducing SwaggerEditor - [editor.swagger.io](https://editor.swagger.io)
- Writing the smallest OpenAPI definition in SwaggerEditor
- Adding `GET /reviews` from the last chapter into our definition

- Interacting with the API in SwaggerEditor

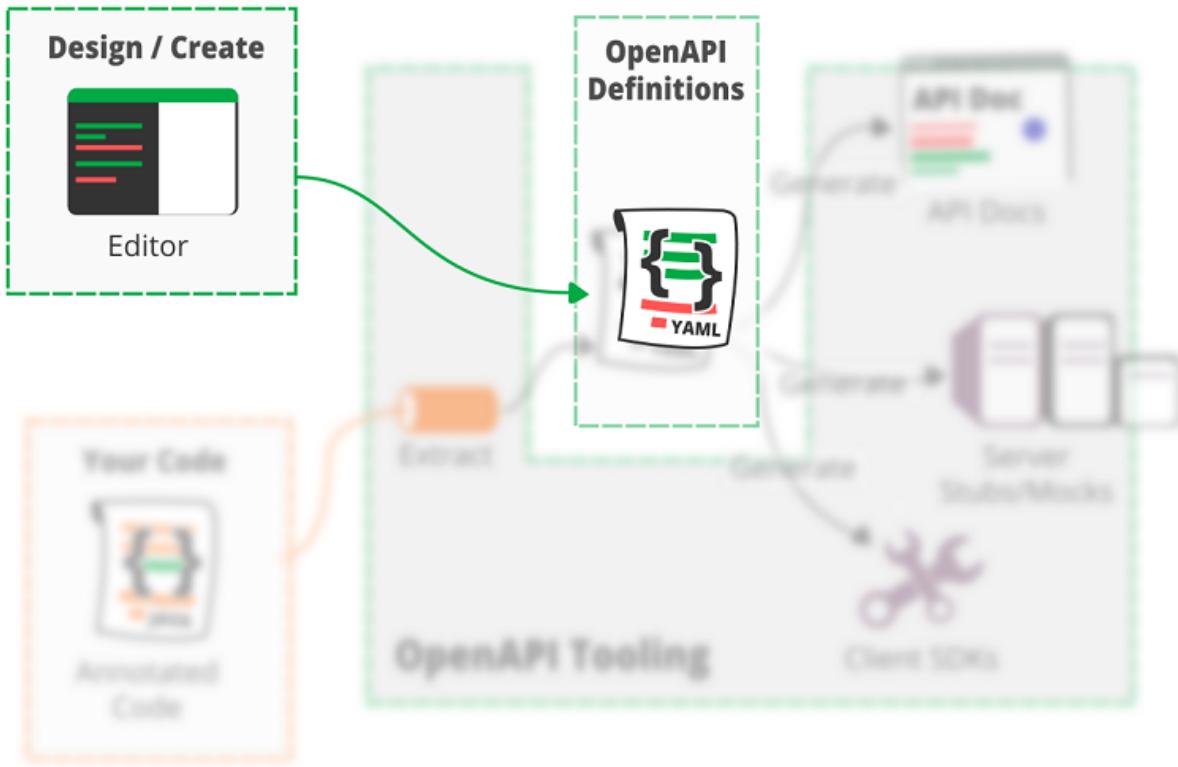


Figure 4.1 Overview of Editor Highlights

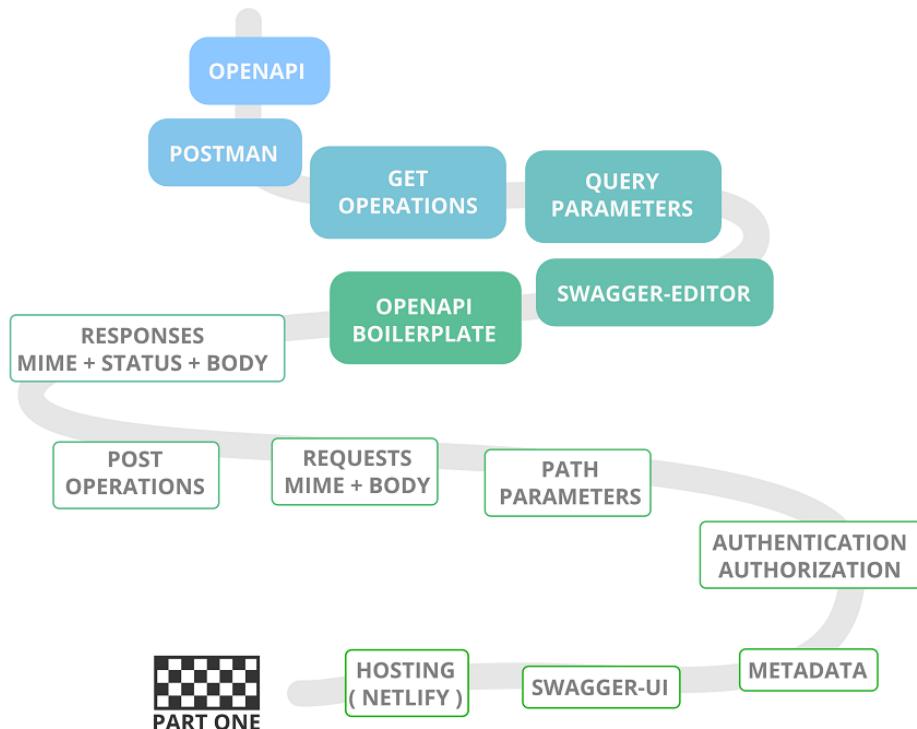


Figure 4.2 Where we are

## 4.1 Introducing SwaggerEditor

To begin our journey into SwaggerEditor let us load it up and take a glance at some of the features.

As mentioned previously, it is an open source web application. We can either use the online hosted version or host it ourselves using a web server (there is a docker version too!). For simplicity's sake we're going to stick to the online version, which should be very close to the latest version of the application.

The online version is hosted at [editor.swagger.io](https://editor.swagger.io) and at the time of writing it looked like this...

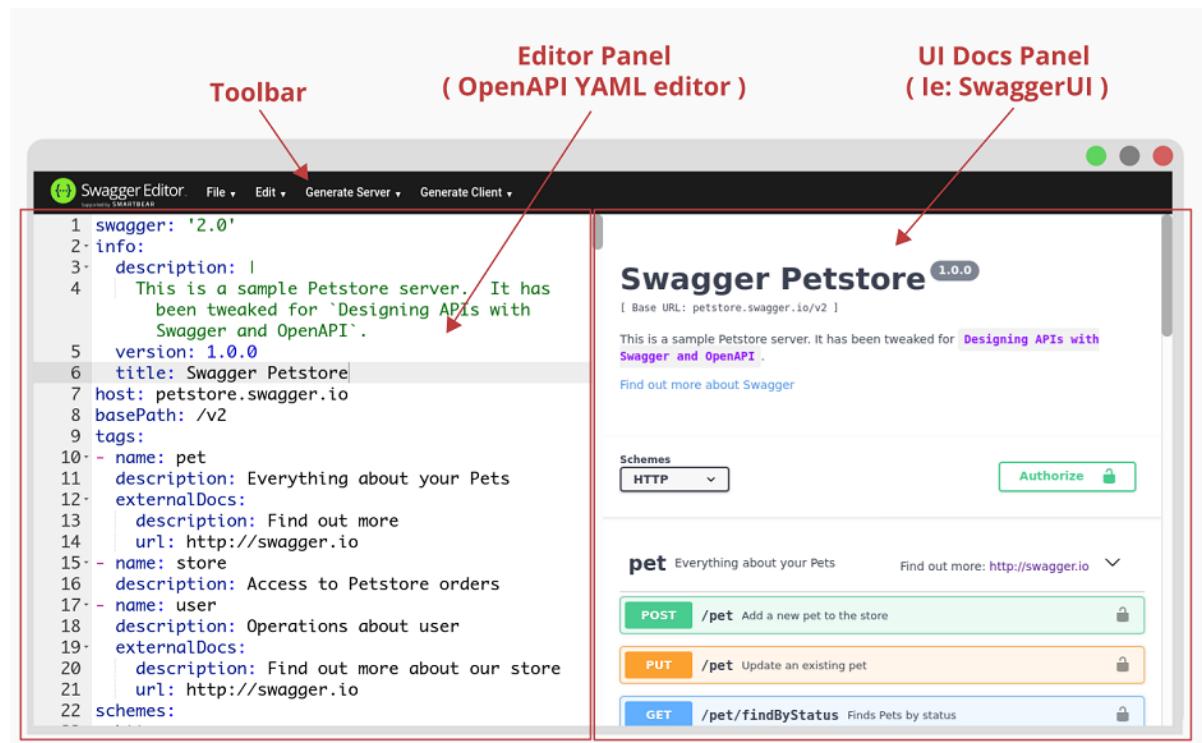


Figure 4.3 SwaggerEditor the initial page

What we can see is the Editor Panel, the Toolbar and the UI Docs Panel.

### NOTE

### The Editor Panel

The Editor Panel is the text editor where we will write our YAML for our OpenAPI definition. The content of this panel *is our OpenAPI definition*.

**NOTE****The Toolbar**

The Toolbar contains some options to import (or fetch) a definition from a URL, a menu for generating code stubs and SDKs, and includes utilities to help generate OpenAPI fragments. We won't be looking at the toolbar in this part but it is useful to know that those features are there.

**NOTE****The UI Docs Panel**

The UI Docs Panel is a reflection of what is in the Editor Panel. This is panel is an embedded version of another Swagger tool called SwaggerUI (we'll meet the standalone version later on in the book). As you type or make changes in the Editor Panel, you will see immediate feedback in the UI Docs Panel. This gives us some level of confidence in what we're writing.

**NOTE****Persistence**

The first time you visit the web app it will come pre-loaded with an example OpenAPI definition. As a convenience, any changes you make to the YAML will be stored on your browser. This means if you reload the page or visit the site at a later time, your changes should still be there. It is only meant as a convenience and isn't full proof! It is prudent to also keep a copy of your definition outside of the tool, should it be important to you.

**IMPORTANT**

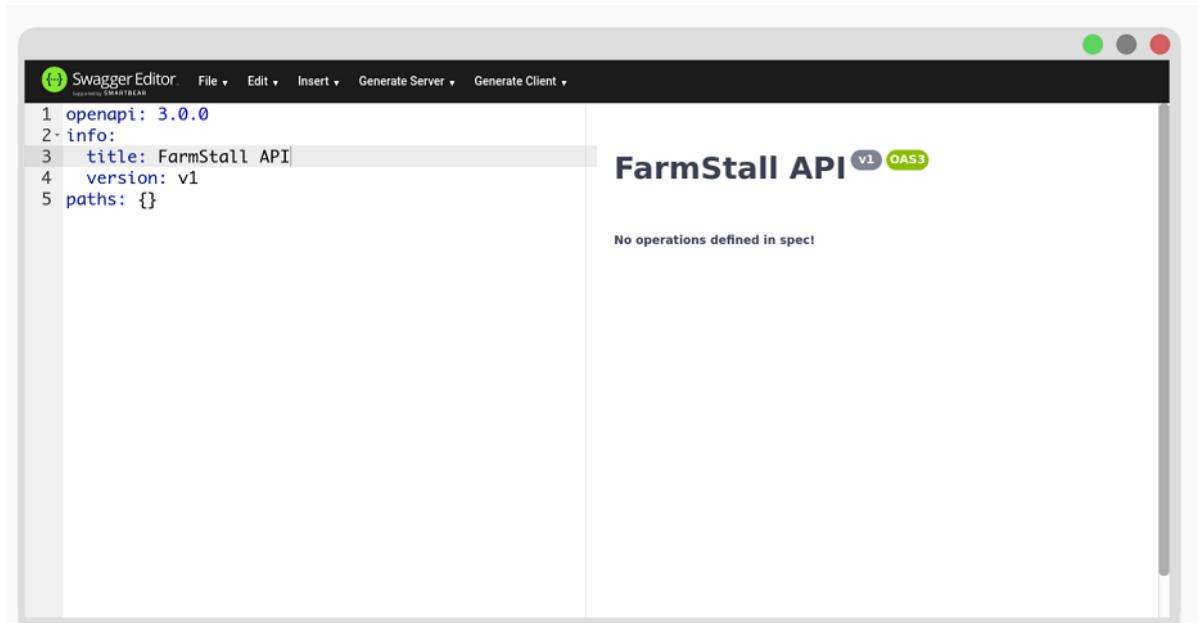
At the time of writing, SwaggerEditor will load up the Petstore API, written with the `Swagger 2.0` specification. So take care when using it as the basis for learning the specification. In this book we describe APIs using the `OpenAPI 3.0.0` specification which is the latest version and encouraged over `Swagger 2.0`. Later on in this book, we'll cover migrating from `Swagger 2.0` to `OpenAPI 3.0.0`. Right now it is important to ignore this default definition as there are a lot of differences between the two versions.

## 4.2 Writing the smallest OpenAPI definition in SwaggerEditor

Before we can start describing any operations of the FarmStall API we will need to write some boilerplate YAML to set the stage. We'll start with the smallest but still valid definition and slowly add to it.

After we have a valid definition we'll hop over to SwaggerEditor and write it out using that tool. The definition we're writing will serve as the base of the FarmStall API definition. As we go through the next few chapters we'll continue to flesh out the details and describe more areas of that API.

Our goal right now is to get the following:



**Figure 4.4 The smallest OpenAPI definition**

That's not too scary... lets go ahead make an valid definition and then write it up!

#### 4.2.1 The smallest yet valid OpenAPI definition

What goes into the smallest OpenAPI definition? Three things...

1. The OpenAPI identifier and the version of OpenAPI used
2. The Info object with the Title and Version fields
3. An empty Paths object

To identify this YAML document as being an OpenAPI definition we add in the `openapi` field, where its value is the *version of the OpenAPI specification* we're using.

##### Listing 4.1 Just the OpenAPI field

```
openapi: 3.0.0 ①
```

- ① We're using version 3.0.0 of OpenAPI

Not exactly thrilling but needed... lets move on to the metadata of the API...

We need to have a `title` and a `version` (version of the API definition and not of the specification), both of those fields fall under the `info` object. When we write it out it looks like this:

## Listing 4.2 OpenAPI field and Info object

```
openapi: 3.0.0
info:
  ①
    title: FarmStall API ②
    version: v1 ③
```

- ① The `info` object stores metadata of the API we're describing
- ② The `title` of the API, a human friendly name of the API
- ③ The `version` of the API, can be any string... we're using the old fashioned `v1`.

That's starting to look a little more interesting. To put a final touch on this we can finish it off and make it a *valid OpenAPI definition* by adding the last required field: `paths`. We'll leave the value of `paths` empty for now, later it will hold the GET operation we described in the last chapter.

## Listing 4.3 Minimal OpenAPI Definition

```
openapi: 3.0.0
info:
  title: FarmStall API
  version: v1
paths: {} ①
```

- ① The `paths` field which we've set to be an empty object.

Aha! At last we have the barest, smallest, most spartan yet valid OpenAPI definition!

We've effectively described very little but we are on the road to doing so. For our next trick we're going to write this definition inside SwaggerEditor and see what that feels like.

### 4.2.2 Writing in SwaggerEditor

The steps we need to take after visiting [editor.swagger.io](https://editor.swagger.io) are:

- Clear out the editor
- Type out our minimal OpenAPI definition
- See what happens

There are two ways to quickly clear the editor:

- Click in the editor panel, so that you see a cursor blinking. Then press `Ctrl-A` or `Cmd-A` to select all the YAML within the editor. Then delete by pressing the `delete` or `backspace` key.
- Alternatively there is an option in the Toolbar under `File > Clear editor`

We should now have no YAML in the editor and a sad looking UI Docs panel. Time to write out

our bare bones definition, so go ahead and write out the following:

#### Listing 4.4 Minimal OpenAPI definition

```
openapi: 3.0.0
info:
  title: FarmStall API
  version: v1
paths: {}
```

If we get it right it'll look something like this:

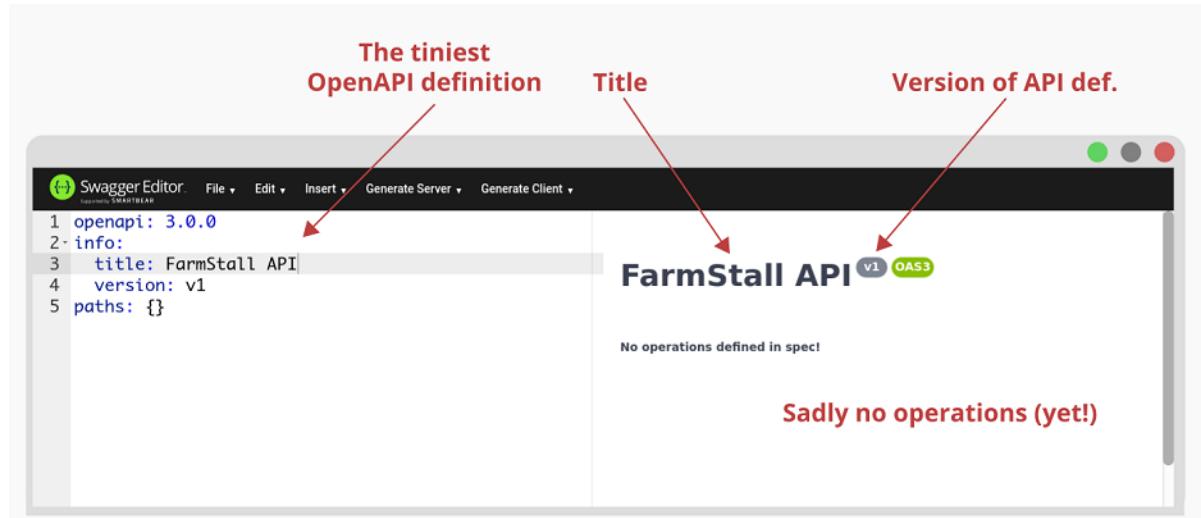


Figure 4.5 The smallest OpenAPI definition

Whoop! Much happiness abounds. We've done just enough work to form a coherent definition... now we can go home and party like it's a Friday on New Year's eve.

#### 4.2.3 A word on validation

As we write, we make mistakes. Some of them can be *happy accidents*<sup>6</sup> but most will be silly little things and typos. SwaggerEditor will try to help out, mostly by gently screaming at you from the UI Docs Panel. The validation happens as you are typing (for that instant feedback) so you can expect it to complain a bit as you type.

This is what they look like so you know what's happening...

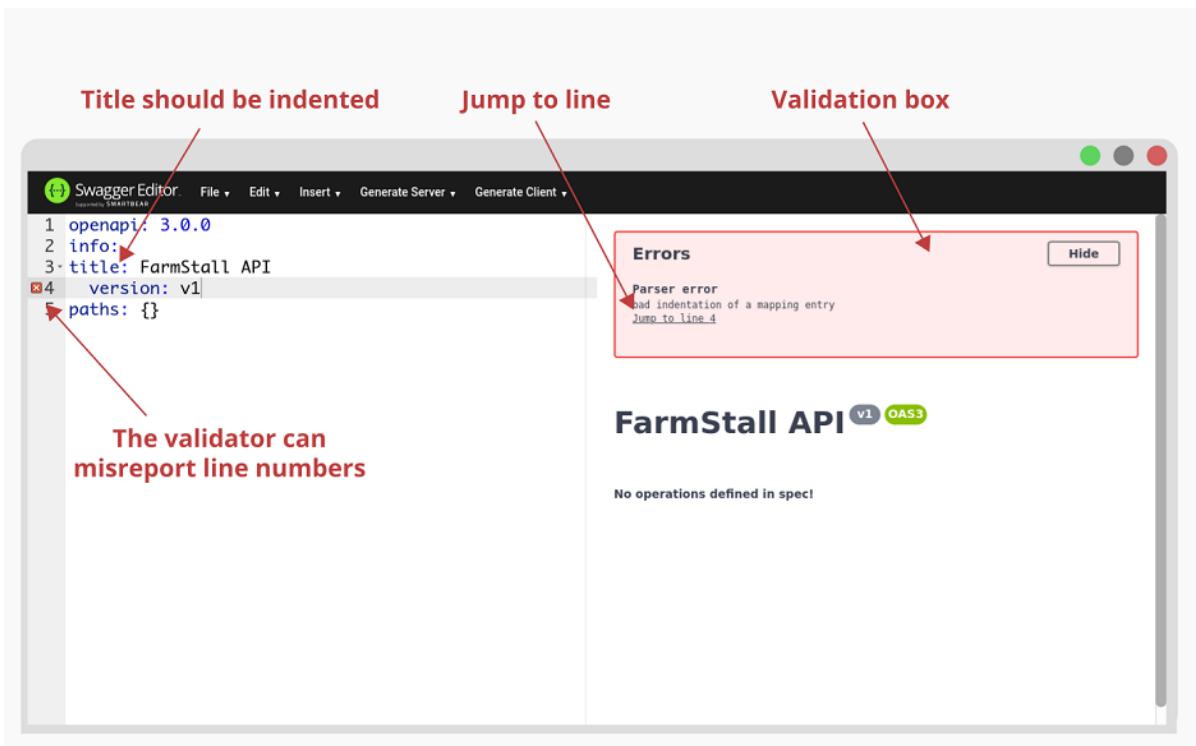


Figure 4.6 SwaggerEditor validation error

### 4.3 Adding `GET /reviews` into our definition

After our celebrating has subsided we have more demands out of the humble (although loud) SwaggerEditor. We want to add in our operation! We want our efforts of the previous chapter recorded into this new OpenAPI Definition.

We're going to add the fragment under the `paths` object, because that's where it belongs :) Together they'll look like this...

#### Listing 4.5 OpenAPI definition of FarmStall with one operation

```
openapi: 3.0.0
info:
  title: FarmStall API
  version: v1
paths: ①
  /reviews: ②
    get:
      description: Get a bunch of reviews.
      parameters:
        - name: maxRating
          description: Filter the reviews by the maximum rating
          in: query
          schema:
            type: number
      responses:
        200:
          description: A bunch of reviews
```

- ① We add our operation into the `paths` object (removing the empty `{ }`)

- ② Our GET `/reviews` operation nestled lovingly within the `paths` object.

When we add that into SwaggerEditor (I get excited thinking about seeing the results) we get the following:

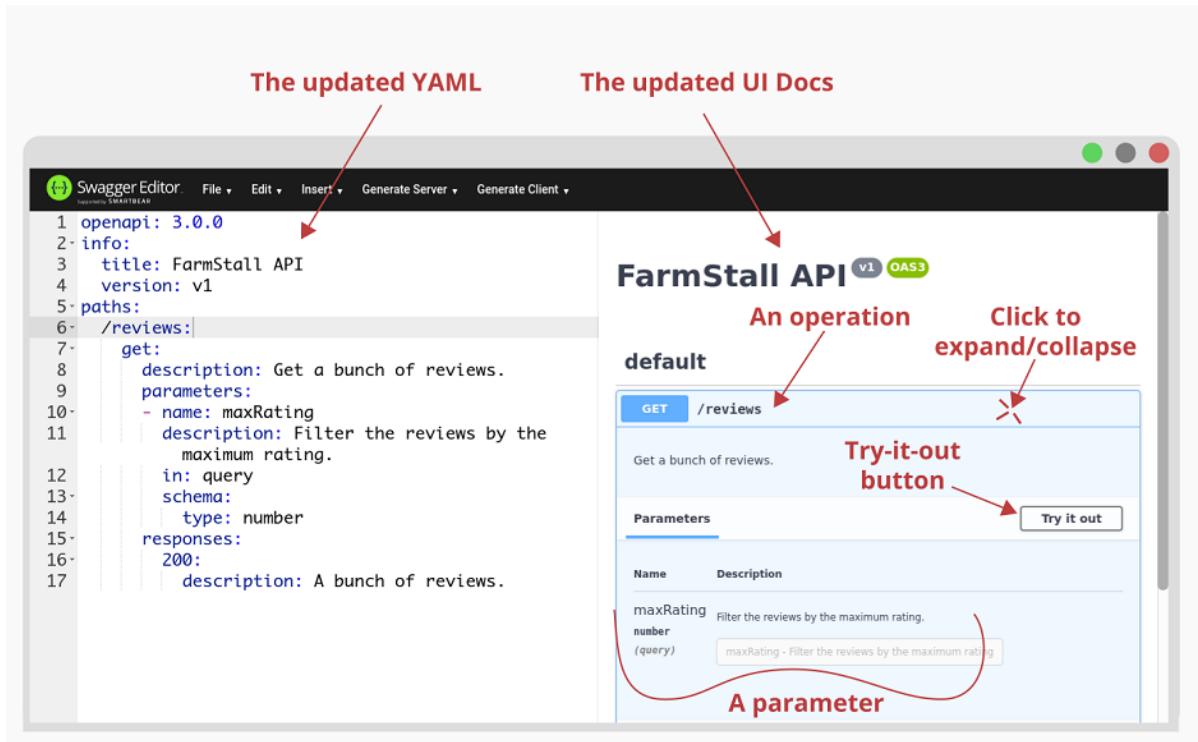


Figure 4.7 SwaggerEditor with the start of FarmStall API

In it we can see the operation (you can click on the operation to expand it and see the details within), as well as the metadata from the minimal OpenAPI definition (ie: title, version).

Awesome. At this point we might be satisfied that our definition serves a good purpose. It has automagically generated documentation but you might be wondering, is there anything more we can do? Well... we can interact with the API!

## 4.4 Interacting with our API

Having described an API, The UI Docs Panel of SwaggerEditor has a nifty little tool built-in. It has an API console, or as its known in the tool... the Try-it-out feature.

It allows you to execute API requests from within SwaggerEditor, to see if they work and what they return. This helps us a bunch when we're describing an existing API, since it allows us to confirm the operations work as we've described them.

We are missing one key element to make this work, let's try to figure out what it is. We will try to execute, and see what happens.

#### 4.4.1 Executing GET /reviews

To execute the GET /reviews operation there are several steps:

1. Expand the operation
2. Click the "Try it out" button to enable the feature
3. Fill in any parameters
4. Click on the blue "Execute" button

For steps (1) and (2) Refer to Figure Image : 4.7

For steps (3) and (4) see below:

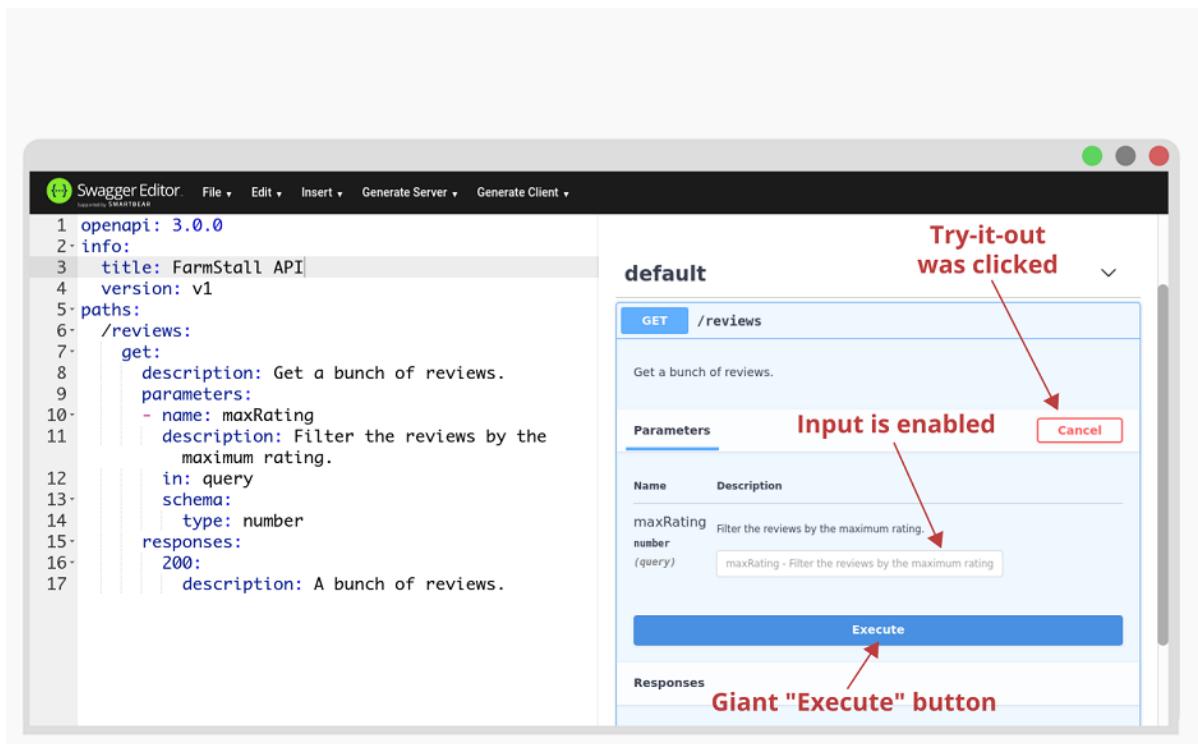


Figure 4.8 SwaggerEditor Try It Out enabled

If you follow the steps (it's encouraged) you won't find much joy. The request should fail, and perhaps you can guess at the reason... we don't know where the server/host is!

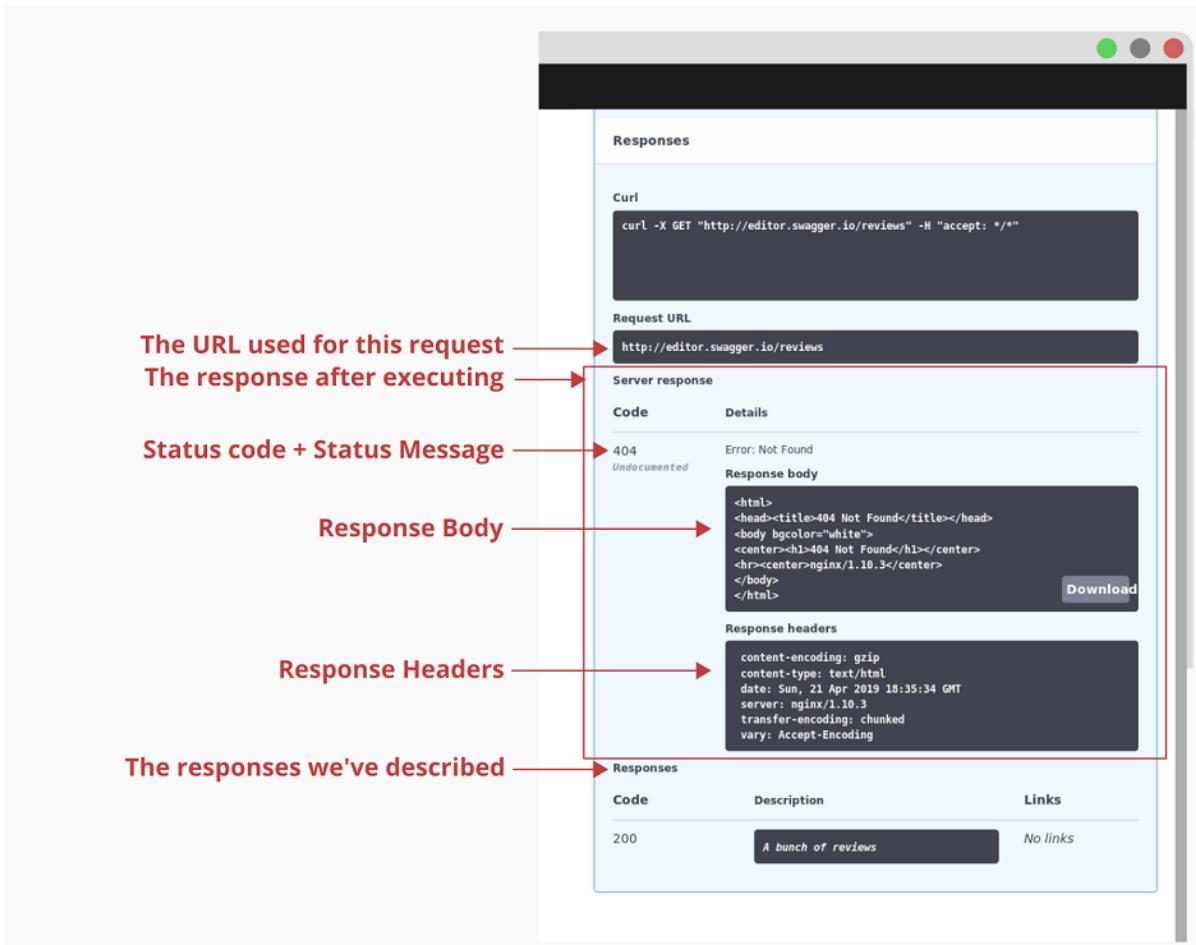


Figure 4.9 SwaggerEditor Try It Out failed!

Do you see the issue? The Try-it-out feature believes our server to be [editor.swagger.io](http://editor.swagger.io) itself! We're trying to call `GET editor.swagger.io/reviews`. Now this is expected behaviour because a common pattern is to serve a version of SwaggerEditor (or the UI Docs Panel only, ie: SwaggerUI) with the API itself, so we might find [farmstall.ponelat.com/v1/swagger-ui](https://farmstall.ponelat.com/v1/swagger-ui) (no, I didn't make it that easy, it's just an example of what a lot of folks do :P).

Our problem is that we haven't described *where the API is hosted*, fortunately there is a simple solution for that. The `servers` field.

#### 4.4.2 Adding servers to our definition

Once described our server object will look like this:

##### Listing 4.6 Servers Array

```
servers: ①
- url: https://farmstall.ponelat.com/v1 ②
```

① The servers field

- ② An array item with an object. Which has a single field `url` that points to the base URL of our server. It'll serve as the base for all paths in this definition.

We can easily add this server object into our API definition...

#### Listing 4.7 OpenAPI definition of FarmStall with servers added

```
openapi: 3.0.0
info:
  title: FarmStall API
  version: v1
servers: ①
- url: https://farmstall.ponelat.com/v1
paths:
  # ... our operation ②
```

- ① The `servers` array added to the root of our API definition  
 ② We redacted the `paths` contents for brevity

The affect of adding the `server` field should be a new dropdown in the UI Docs Panel. It'll just contain the one server and that's fine, and naturally the Try-it-out feature will pick up on this server and use it as the base URL.

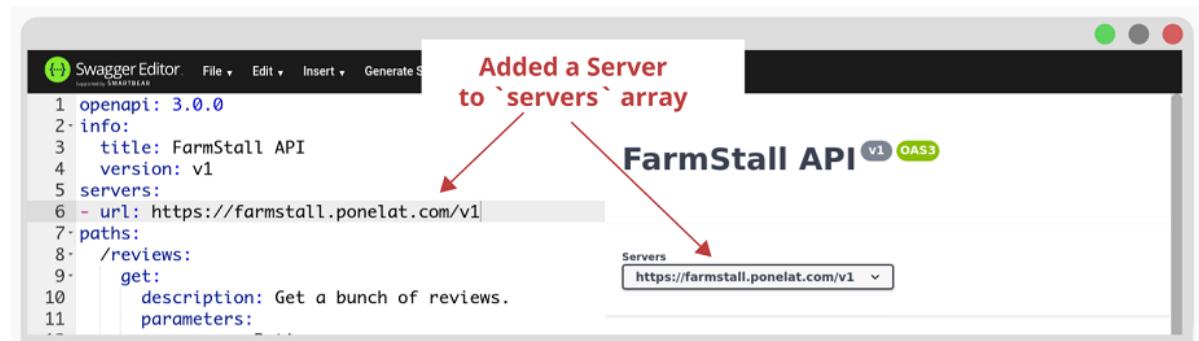


Figure 4.10 SwaggerEditor with an added server

#### 4.4.3 Executing GET /reviews (again)

Cool. Now go ahead and execute the `GET /reviews` operation again. With our server added we should get a taste of success. Something that looks dangerously like this:

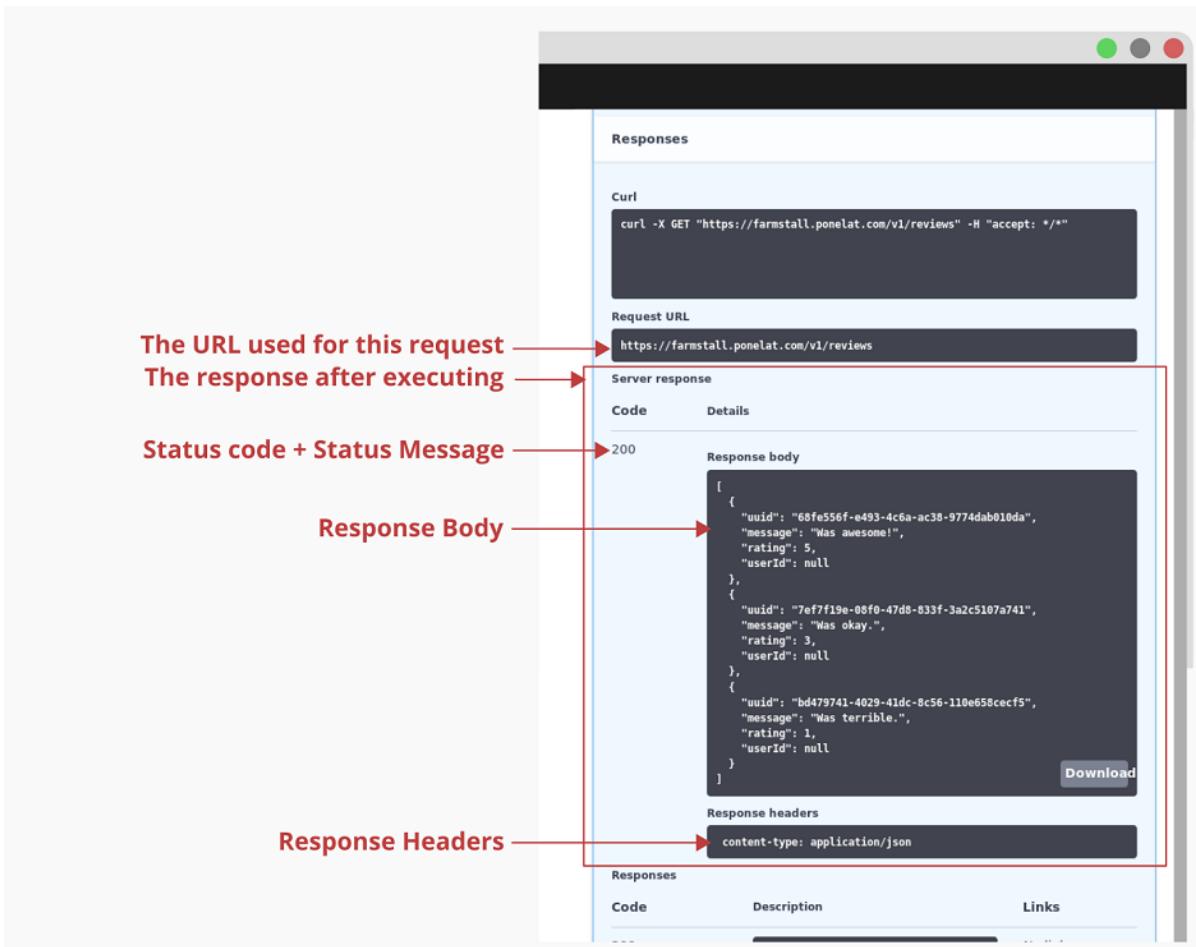


Figure 4.11 SwaggerEditor Try It Out succeeded!

Congratulations!

We've successfully described a part of our FarmStall API — one that is complete and functional! We used SwaggerEditor to help with the writing and we then got some feedback by using the Try-it-out feature. Going forward there are lots of interesting things we can continue to describe... from complex models to security features.

Slowly we'll shape and describe the FarmStall API until developers around the world would find it straight forward to implement.

## 4.5 Summary

- We introduce the SwaggerEditor, a tool where one can write OpenAPI definitions and see feedback from the UI Docs Panel.
- The smallest, valid OpenAPI definition includes the following fields: `openapi` ( version of the specification ), `info` (includes meta data of the API) and `paths` (where our operations are defined).
- The `info` field has two children fields namely `title` and `version`. Where `title` is the human friendly name of the API, and `version` is the version of the API definition (file).
- Operations can be added under the `paths` field. The direct children of these are URIs (ie: `/reviews`) and the children under the URIs are methods (ie: `get`). Finally the fields under the methods detail the operation (ie: `description`, `parameters`, `responses`, etc).
- To describe where the API is hosted (FarmStall API in this case), there is the root level `servers` field. It is an array of Server Objects where, at minimum, a `url` field is defined. For this chapter we had a single Server Object that had a `url` pointing to [farmstall.ponelat.com/v1](http://farmstall.ponelat.com/v1).
- SwaggerEditor includes a Try-it-out feature that allows you to execute requests based on the OpenAPI definition.

# Describing API responses

*Data makes the world go round ...or maybe that's money?*

In this chapter we're going to describe a simple HTTP response with OpenAPI and add it to our FarmStall API definition. Responses are the fuel for consumers and they can be large/complex. Describing them is an important part of communicating an API.

An HTTP response is made up of three things. A status code, a set of headers and an optional body. We're going to focus on the status code and body for now and leave response headers for another chapter.

A response definition in OpenAPI needs at least a status code and a description. If there is a response body it must include at least one media type (eg: application/json).

Response bodies are where it's at though. All the exciting bits. To describe the *shape of the data* OpenAPI adopted the **JSON Schema (v4)**<sup>7</sup> specification. Which was designed to describe what goes into a JSON document but with a few tweaks can describe XML and FormData as moderately well too. This format blends seamlessly with the rest of OpenAPI but there are proposals to include external specifications for describing data in newer versions of OpenAPI... watch this space!

Here is a quick glance at a sample response definition...

## Listing 5.1 Example response definition

```
responses:
  200: ①
    description: A human description ②
    content:
      application/json: ③
        schema: ④
          type: object
          properties:
            # ...
```

- ① The status code
  - ② The description
  - ③ The media type for JSON
  - ④ The schema (OpenAPI's flavour of JSON Schema)

Let's learn more about describing data!

This chapter covers

- Learning about JSON Schema and how to describe data
  - Describing the 200 response of GET /reviews
  - Adding the response definition piece into our FarmStall API

## NOTE

## What is a media type?

A **media type** or MIME (Multipurpose Internet Mail Extensions) is list of data formats. The list is standardized by the IANA ( Internet Assigned Numbers Authority) and indicates what format a request (or response) body is in. Common ones are: - application/json - application/xml - text/csv - image/png

## Learn

more:

developer.mozilla.org/en-US/docs/Web/HTTP/Basics\_of\_HTTP/MIME\_types

## What is a status code?

The status codes defined in the HTTP specification are three-digit numbers between 100 and 599 inclusive. These status codes are the high level semantics of the response. Broadly speaking we can think of these statuses as the *success* or *failure* of the request. Eg: 200 Success or 404 Not Found.

Fun references: [http.cat/](http://http.cat/) [httpstatusdogs.com/](http://httpstatusdogs.com/)

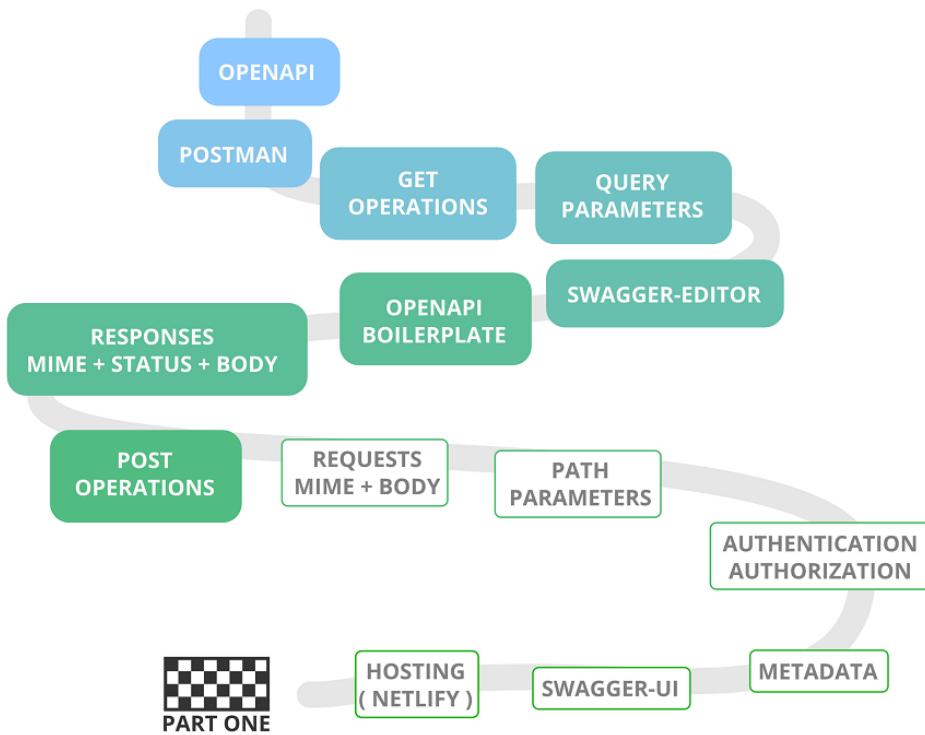


Figure 5.1 Where we are

## 5.1 The problem

Our task this chapter is describe the (successful) response of the `GET /reviews` operation and add it to our burgeoning FarmStall API definition. The response in question is moderately complex as it is an array of objects, but not complex enough to require inheritance or any other sort of composition.

We are presented with the following information about the response...

Table 5.1 Response of GET/reviews

Field	Type	Description	Limits
uuid	string	The ID of the review	UUID v4 (36 chars)
message	string	The review notes	
rating	number	The rating of how good an experience it was, higher is better.	1-5 inclusive, whole number
userId	string or null	The ID of the author	UUID v4 (36 chars), null for anonymous

We're going to take that information of the response and translate it into the OpenAPI format, using the slight variant of JSON Schema - which we will look at first.

The definition will ultimately look like the following — with sections commented out for brevity...

## Listing 5.2 GET /reviews response body

```
openapi: 3.0.0
# ...
paths:
  /reviews:
    get:
      # ...
      responses:
        '200':
          description: A bunch of reviews
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    uuid:
                      type: string
                      pattern: '^[0-9a-fA-F\-\_]{36}$'
                    message:
                      type: string
                    rating:
                      type: integer
                      minimum: 1
                      maximum: 5
                    userId:
                      type: string
                      pattern: '^[0-9a-fA-F\-\_]{36}$'
                      nullable: true
```

In it we can see some familiar terms like "string", "integer" and "object". We'll soon learn the structure of this format to describe data which can be simple and plain or quite intricate and complex.

When it comes to JSON there is an industry standard namely JSON Schema. Before we add our `GET /reviews` response into the definition we can take a look at JSON Schema on its own...

## 5.2 JSON Schema and the mind-blowing world of data schemas

I'm going to send you some data. It'll be in JSON format but I won't tell you what's in the data.

What can you do with it?

If you were bold you may have suggested some interesting ways to handle unknown data — perhaps you'd employ a data discovery algorithm.

But if you were kind enough to allow me the rhetorical question, then perhaps you'd answer with, "Not much can be done with unknown data".

And in truth most applications do require the consumer to know *what structure the data will be in*, which also includes what the structure that data *can be in*. This makes sense since we must be able to make some assumptions about the data in order to make use of it. The more we

understand it, the more we can do with it.

I like to refer to the schema of data as *its shape*. A triangle will fit into a triangular hole, a circle into a circular hole. The data needs to fit the consumer's application. Knowing the shape of the data allows us to build useful things. And that's what we're all about! I joke, of course I build useless things too!

We're going to look at how to describe the JSON data. We're going to use the rock-solid, industry standard JSON Schema. However, we must make note that OpenAPI has a *slight variation of JSON Schema*. This has some small but important differences which we'll note on when we bump into them.

### 5.2.1 JSON Schema

Have you ever heard of XML Schema? Perhaps you have. Well JSON Schema is a way to say what can and cannot be done with JSON, much like XML Schema does this, with XML.

In this section we'll start with describing an object that has the `rating` field in it. So that we can get a taste for JSON Schema.

We'll be aiming to describe the following JSON data...

#### Listing 5.3 Sample Object with a single field

```
{
  "rating": 3
}
```

The above is an object with a single field, `rating`. The field `rating` cannot be more than five nor less than one. It is also a whole number (ie: not a float such as 1.3 or 2.9). We can sum up that info in a table...

**Table 5.2 The rating field requirements**

Field	Type	Description	Limits
rating	number	The rating of how good an experience it was, higher is better.	1-5 inclusive, whole number

Taking that info we'll build up the following JSON Schema fragment...

#### Listing 5.4 JSON Schema of rating

```
type: object
properties:
  rating:
    type: integer
    minimum: 1
    maximum: 5
```

## 5.2.2 Type

We have to describe a JSON object with a single field. Where do we start with this? Well we could start by saying that the root level "type" is an object.

### Listing 5.5 Example JSON Schema for simple object

```
type: object ①
```

- ① This says an object is required.

JSON Schema talk about validation. According to a given schema, data can be valid or invalid. This is core to understanding JSON Schema. The use cases for schemas is large but at its heart it's about validating data.

With our simple schema above we can see how it validates against some data...

Simple validation tests, against the schema above.

### Table 5.3 Validating JSON against the schema

Valid	JSON	Description
valid	{"rating": 3}	Its an object with a field
invalid	"hello"	Expected object, found string
valid	{ }	Nothing wrong with an empty object

Ok, so "hello" isn't an object and {}, {"rating": 1} both are. Sounds about right. Right off the bat we can declare the type of the data, with the `type` field. This field is required for all JSON Schemas.

Let's extend our schema to include the `rating` field.

### Listing 5.6 Adding the rating field

```
type: object
properties:
  rating: ②
  type: number ③
```

- ① When `type: object`, we can declare the `properties` keyword which shows which *fields/properties* it's allowed to have.
- ② Each key under `properties` is a field, where the value is the schema for that field.
- ③ `rating` is declared to have a type of `number`

Let's validate some data against this updated schema...

**Table 5.4** Running validation on expanded schema

Valid	JSON	Description
invalid	{ "rating": "hi" }	Expected number, found string
invalid	{ "rating": true }	Expected number, found boolean
valid	{ "rating": 100 }	Valid. 100 is obviously a number.
valid	{ "rating": 100, "a": "b" }	Extra fields are fine (by default)

*Note how extra fields are fine in this schema.*

### 5.2.3 Minimum and Maximum

Now that we've increased the specificity of our schema, to not only require an object, but to require an object that has a field called `rating`. We've also specified that the field needs to have a number for a value. Not so bad!

We're still not quite there yet with our requirements. As you can tell, we're not supposed to have ratings go above 5... let alone 100!

Let's take a look at the `minimum` and `maximum` keywords we can use to limit the range of allowed numbers.

#### Listing 5.7 JSON Schema of rating

```
type: object
properties:
  rating:
    type: number
    minimum: 1 ①
    maximum: 5 ②
```

- ① Just like `properties` applies to `type: object`, there are other modifiers. For `type: number`, we can declare a `minimum` value.
- ② ...and a `maximum`. The values are inclusive ( ie: 5 is allowed ).

**Table 5.5** Running validation on the limited number schema

Valid	JSON	Description
valid	{ "rating": 1 }	Good stuff
invalid	{ "rating": -48 }	Yup... negative numbers are below 1
valid	{ "rating": 1.43 }	Uh oh... we're not supposed to have decimals

### 5.2.4 Number vs Integer

Our schema is shaping up. But we can still get a few unwanted results shapes, such as floating point numbers (ie: those with decimal points). Our requirements specifically say whole numbers. Our rating system will likely not handle those in-between numbers but we'll sort that right out...

## Listing 5.8 Limiting the rating field

```
type: array
items:
  type: object
  properties:
    rating:
      type: integer ①
      minimum: 1
      maximum: 5
```

- ① What we've done is change to the `integer` type which includes whole numbers only

JSON-Schema has two number types: `number` and `integer`. As you can see the latter is a more limited variant that only includes negative, zero and positive whole numbers. Perfect for our needs!

Let us validate our freshly limited, number type!

**Table 5.6 Validating against whole numbers**

Valid	JSON	Description
invalid	{ "rating": -48 }	Number is too low (less than 1)
invalid	{ "rating": 3.43 }	Yay! Expected an <code>integer</code> found a <code>number</code>
valid	{ "rating": 5 }	Perfect

Yay! We have a schema that correctly represents our `rating` field!

Eager as we are to add more fields, we need to take a step back and head over to OpenAPI land where we can describe the response. In particular we'll look at the status codes first. This is to give our schema a home in the OpenAPI definition before describing our response to its fullest.

## 5.3 Status Codes

A quick word on status codes.

Status codes are those three-digit codes that we find in an HTTP response. Perhaps the most well known status code is `404 Not Found` which of course means that the resource you're after isn't there, *or* you don't have access to know if it is. The HTTP specification puts status codes into five categories and each category is a range of codes, eg: `200-299` (or `2xx`) are those codes that indicate the request was successful.

Here are some example status codes and their semantics...

**Table 5.7 Status code examples:**

Status	Status text	Description
101	Switching Protocols	Typically used to upgrade to a websocket connection
200	Ok	The request was successfully executed
201	Created	A new resource was successfully created
301	Moved Permanently	Redirect to another URL, which the client can always do in future.
403	Forbidden	Elevated permissions are required
404	Not Found	The resource asked for was not found
504	Gateway Timeout	The proxy/gateway could not reach the backend server

And this is the list of those categories (using 1xx to represent 100–199 inclusive).

**Table 5.8 Status code categories**

Range	Category	Notes
1xx	Informational	Most common is when a websocket connection is upgraded
2xx	Success	This indicates some form of success like the general 200 or the 201 for <i>created</i>
3xx	Redirects	The resource has a different location/URI
4xx	Client Error	The client did something wrong like misspell a resource or provide invalid details
5xx	Server Error	The server hit an error that isn't a fault of the client

In addition to Mozilla's excellent documentation on status codes at

- [developer.mozilla.org/en-US/docs/Web/HTTP/Status](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)

There are also a few fun websites that detail the semantics, such as

- [http.cat](http://http.cat)
- [httpstatusdogs.com](http://httpstatusdogs.com)

## 5.4 Media Types (aka MIME)

The briefest word on media types.

HTTP is a multimedia protocol and can handle requests and responses in many different formats. To indicate what format the data is in it will include a header, typically `Accept` or `Content-Type` with a *media type* as its value.

A media type has a "type" and a "sub-type" (and optional parameters, see: [developer.mozilla.org/en-US/docs/Web/HTTP/Basics\\_of\\_HTTP/MIME\\_types](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types) for more info).

The type is the category, such as `text`, `audio`, `image`, `font`, etc. And the sub-type makes it concrete, eg: `text/plain`, `image/png`, etc.

Here are some common data formats...

**Table 5.9 Common MIME Types for data**

MIME	Description
text/html	The HTML you get back from a webserver
text/csv	Comma separate values
image/png	PNG encoded image
application/json	JSON data
application/xml	XML data

**NOTE**

You may find suffixes in some media types that indicate a wrapper format. For example the SVG media type used for salable images has the media type of `image/svg+xml` which has the suffix `+xml`. This indicates that the format is XML but will be compliant with the SVG schema.

A practice in API design is to sometimes create your own (or vendor) media types which can be used to version APIs. Again using the suffix to indicate the wrapper format. An example of this in the wild is `application/vnd.github.v3+json` which is for GitHub's API, version 3 and using JSON as a wrapper format.

That should be enough to keeps us abreast of media types for now. Let's lay get into the OpenAPI side of things!

## 5.5 Describing `GET /reviews` response

With the briefest overview of the moving parts in an API response, namely JSON Schema, status codes and media types. I think it's about time we combine them into our (most delicious) OpenAPI definition.

The goal of this chapter is to describe the `GET /reviews` response, in particular the 200 response. At the end we'll have a response definition that looks like the following...

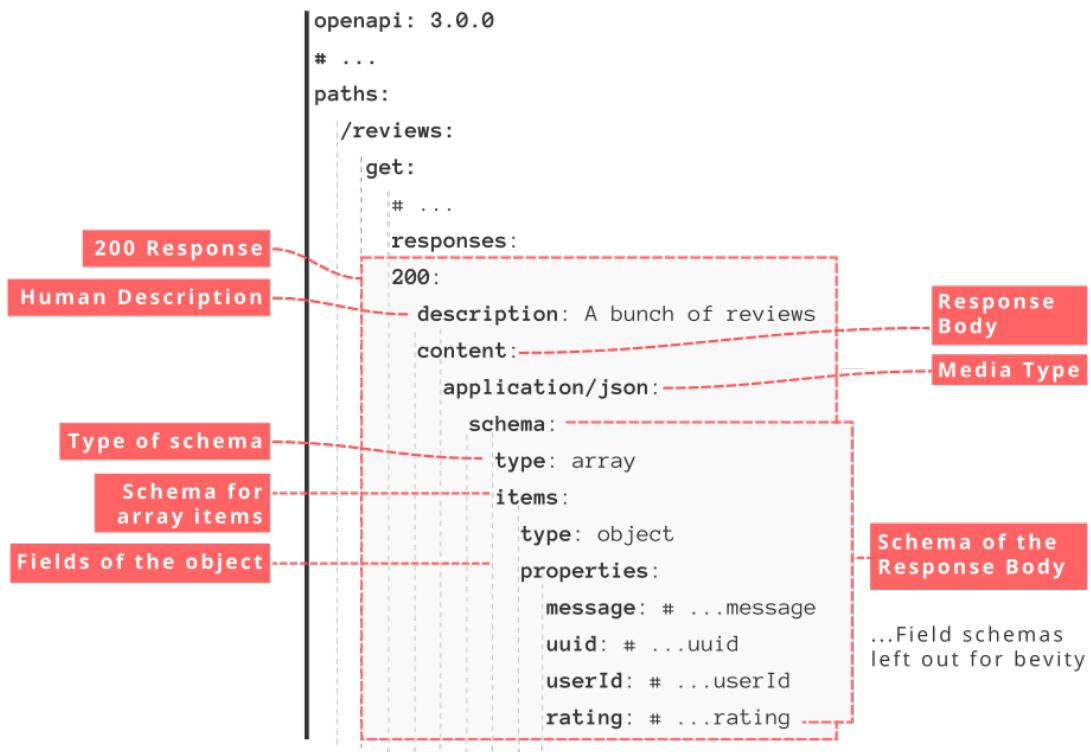


Figure 5.2 OpenAPI Response Annotated

### 5.5.1 Smallest response in OpenAPI

We've already seen the smallest (or minimum) response in a previous chapter as we needed it to form a valid OpenAPI definition. But we didn't really look at it.

Inside each operation we can describe several responses, one for each status code. The responses all go under the `responses` field. Each key in the `responses` field, is a status code and its value is a response definition object. The only required field in a response definition object is the `description` field, which is for humans to read.

#### Listing 5.9 Bare minimum response definition

```

paths:
  /reviews:
    get: ①
      responses: ②
        '200': ③
          description: A list of reviews ④

```

- ① The operation under which we're describing responses
- ② The `responses` keyword
- ③ The status code... as a string
- ④ The `description` keyword, mostly for humans

Voilà! This is the start of our `GET /reviews` - 200 response. Time to add meat to those bones...

### 5.5.2 `GET /reviews 200 response body`

So far we have the skeleton of a response but before we can add in the JSON Schema, we need to write out some boilerplate for the response body.

Such boilerplate includes the `content` ( ie: response body ) field and at least one media type (ie: `application/json`). We'll also start our schema by being valid from the get-go. We're going to declare the response body as being an array. An array of what? Well it can be an array of anything at this point. This schema, although broad, is valid for our response and that's important. We are of course, going to add more details and constraints to that schema.

Here is our response with a body definition...

#### **Listing 5.10 Response body boilerplate**

```
#...
paths:
  /reviews:
    get:
      # ...
      responses:
        '200':
          description: A bunch of reviews
          content: ①
            application/json: ②
              schema: ③
                type: array ④
```

- ① The response body goes under `content`
- ② The media type of the response body ( ie: JSON )
- ③ The `schema` field which will contain the schema (ie: the data shape)
- ④ We'll start with the broadest, but valid, schema we can muster. An array

### 5.5.3 Adding `rating` into our response body

The first field we want to add is `rating`. It'll represent the rating that the review got. Where 1 is the poorest and 5 is the most glowing rating of a review. Fortunately we've already described this field in our first taste of JSON Schema. What we'll do now is add it into our definition.

Our current response body describes an array of *anything*. Let's change that to be an array of objects, with the `rating` field in them.

## Listing 5.11 Response body with rating field

```
#...
paths:
  /reviews:
    get:
      # ...
      responses:
        '200':
          description: A bunch of reviews
          content:
            application/json:
              schema: ①
              type: array
              items: ②
                type: object ③
                properties:
                  rating:
                    type: integer
                    minimum: 1
                    maximum: 5
```

- ① For the rest of this chapter, we'll just focus on the schema underneath this keyword — the good stuff
- ② `items` is a property that only applies to `type: array`
- ③ The schema that describes an object with a rating field. Which we did earlier.

There is a fair bit of YAML there... take a moment to digest it.

As a quick sanity check let's look at a validation table based on the schema.

**Table 5.10 Validating the schema**

Valid	JSON	Description
<code>valid</code>	<code>[]</code>	Empty array is fine as we didn't specify min or max items
<code>valid</code>	<code>[{ "rating": 1 }]</code>	An array item that is valid
<code>invalid</code>	<code>[{ "rating": 5}, false ]</code>	Expected object found boolean

Nice. Our response body is described... all that's lacking is a few more fields to completely describe it according to the Farmstall v1 API.

### 5.5.4 Describing message, `uuid` and `userId`

Let's start extending our schema with something fun.. strings! As part of the `GET /reviews` endpoint there are several strings defined in the response — namely, `message`, `uuid` and `userId`

Here is the table describing those fields...

**Table 5.11 The string fields and their requirements**

Field	Type	Description	Limits
uuid	string	The ID of the review	UUID v4
message	string	The review notes	None
userId	string or null	The ID of the author	UUID v4 and null for anonymous

**Listing 5.12 Adding the message field**

```

①
type: array
items:
  type: object
  properties:
    rating: # ...
    message: ②
      type: string ③

```

- ① We're only looking at the contents under the `schema` field for bevity
- ② Our `message` field
- ③ With the type set to `string`

That was pretty easy we added a field and ensured its type was `string`.

Let's see how it validates...

**Table 5.12 Validating message**

Valid	JSON	Description
invalid	<code>[{ "rating": 5, "message": 1 }]</code>	Expected string, found number
valid	<code>[{ "rating": 5, "message": "Hello" }]</code>	All good, it has a string in it
valid	<code>[{ "rating": 5, "message": "" }]</code>	There are no limits so <code>message</code> can be an empty string too

Time for something more meaty and what is more meaty than UUIDs? Universally Unique Identifiers are a standard for generating unique IDs. The FarmStall API makes use of them for all of its IDs as they're easy to generate and are statistically guaranteed to be unique. No need to track counters in a database and we can even generate IDs from the client side!

The format is a series of random letters and numbers with some dashes in prescribed places, eg: `3b5b1707-b82c-4b1d-9078-157053902525`. To find out more about UUIDs and UUIDv4 in particular — the [Wikipedia page](https://en.wikipedia.org/wiki/Universally_unique_identifier) is a good start, [en.wikipedia.org/wiki/Universally\\_unique\\_identifier](https://en.wikipedia.org/wiki/Universally_unique_identifier)

In order to limit our schema to match this type of string we're going to add a crude regular expression... that reads "36 characters composed of letters, numbers and dashes". We don't need to be more specific than that in this case.

**NOTE** OpenAPI uses the JavaScript regular expression syntax. More precisely it uses: [www.ecma-international.org/ecma-262/5.1/#sec-7.8.5](http://www.ecma-international.org/ecma-262/5.1/#sec-7.8.5). As different languages have slightly different regular expressions it helps to know which one is used :) For playing around with regular expressions I find [regex101.com/](http://regex101.com/) to be a great resource.

### Listing 5.13 Adding uuid field

```
type: array
items:
  type: object
  properties:
    rating: # ...
    message: # ...
    uuid: ①
      type: string ②
      pattern: '^[0-9a-fA-F\\-]{36}$' ③
```

- ① Our uuid field
- ② It is also of type `string`
- ③ Ooh a regular expression pattern! This ensures that the string meets the UUID v4 spec (not 100% precise but close)

Let's create a validation table based on our growing schema...

**Table 5.13 Validating message**

Valid	JSON	Description
invalid	[ { "rating": 5, "message": "hello", "uuid": "hi" } ]	Pattern did not match for <code>uuid</code>
invalid	[ { "rating": 5, "message": "hello", "uuid": "" } ]	Pattern did not match for <code>uuid</code>
valid	[ { "rating": 5, "message": "hello", "uuid": "3b5b1707-b82c-4b1d-9078-157053902525" } ]	That's a UUID v4!

This schema stuff is easy! Let us add some more, let's add `userId`...

## USERID AND NULLABLE

Up till this point we've been using vanilla JSON Schema, however, now we come to one of the subtle (but biggish) differences between OpenAPI's flavour of JSON Schema and JSON Schema itself.

The `nullable` keyword.

In JSON Schema multiple types are allowed, eg:

## Listing 5.14 Multiple types in JSON Schema

```
# JSON Schema, NOT valid in OpenAPI
type: [ number, string, null ] ①
```

- ① The value can be either a `number`, `string` or `null`

In OpenAPI multiple types like this are **not supported** (as of 3.0.x at least). But to make allowance for the very common use case of having a schema be some value *or null*. The spec has the `nullable` keyword.

## Listing 5.15 Nullable in OpenAPI

```
type: string ①
nullable: true ②
```

- ① Only one type at a time is allowed in OpenAPI (a string in this case)
- ② This value can also be `null`

Let's go ahead and add `userId` which is a UUIDv4 (so we'll add the `pattern` property from before) but is also allowed to be `null`. As anonymous reviews are allowed and they have no associated author.

This is the schema so far with the `userId` included.

## Listing 5.16 Full schema for an array of Review objects

```
type: array
items:
  type: object
  properties:
    rating:
      type: integer
      minimum: 1
      maximum: 5
    message:
      type: string
    uuid:
      type: string
      pattern: '^[0-9a-fA-F\\-]{36}$'
    userId: ①
      type: string ②
      pattern: '^[0-9a-fA-F\\-]{36}$' ③
      nullable: true ④
```

- ① Added our `userId` field
- ② Type `string`
- ③ The UUID v4 regular expression pattern
- ④ This field is allowed to either by `"3b5b1707-b82c-4b1d-9078-157053902525"` (or similar) or `null`.

And we can see our full OpenAPI schema so far (careful, it's growing big now!)...

### Listing 5.17 Full definition for GET /reviews so far

```
openapi: 3.0.0
info:
  title: FarmStall API
  version: v1
paths:
  /reviews:
    get:
      description: Get a list of reviews
      parameters:
        - name: maxRating
          in: query
          schema:
            type: number
      responses:
        '200':
          description: A bunch of reviews
          content:
            application/json:
              schema: ①
              type: array
              items:
                type: object
                properties:
                  rating:
                    type: integer
                    minimum: 1
                    maximum: 5
                  message:
                    type: string
                  uuid:
                    type: string
                    pattern: '^[0-9a-fA-F\\-]{36}$'
                userId:
                  type: string
                  pattern: '^[0-9a-fA-F\\-]{36}$'
                  nullable: true
```

- ① Our schema for the (successful) response body of GET /reviews

## 5.5.5 *Schemas*

What we've done is describe the successful response (200) of GET /reviews. But more than that we've touched on data schemas, in particular JSON Schema. Data schemas are perhaps, *the most interesting* part of an API definition. As they describe the data we get back or need to send up.

As we'll see, the schemas learned here will work just as well in request bodies as they do in response bodies. And there are more powerful features we haven't touched upon around composition and polymorphism!

### IMPORTANT OpenAPI vs JSON Schema

JSON Schema is a specification to describe JSON data. OpenAPI is a specification to describe REST APIs which *can contain JSON data*. When OpenAPI wrestled with how they were going to describe this data they needed to figure out something that would work for XML, JSON and FormData. They chose JSON Schema as the spec used to model the data shapes. However they needed to make some tweaks to support the different formats (ie: XML, FormData). They were also very interested in generating code from the OpenAPI definitions and so further tweaked the JSON Schema to be *more deterministic*. This decision to use a variation of JSON Schema remained controversial and so a motion was made to allow for *external schemas*. At the time of writing this is slated to be in 3.1.0 version of OpenAPI. And will allow designers to use full-blown JSON Schema. For our purposes the feature set of the OpenAPI flavour (of JSON Schema) is quite sufficient.

## 5.6 Summary

- Operations in OpenAPI can describe a single response for each status code (eg: 200, 404) and within that response, it can describe a response body for each media type (eg: application/json). All responses in an operation will be described under the `responses` field
- For describing data, OpenAPI uses a flavour of JSON Schema that is around 90% the same as JSON Schema v4. The differences were to allow for a more deterministic code generation which was key to Swagger/OpenAPI's success. One such example is the `nullable` keyword (OpenAPI only) and lack of multiple-types (JSON Schema only).
- All schemas have a `type` field that describes one of the basic JSON types: `object`, `array`, `string`, `number`, `boolean`, `integer` or `null`.
- Object schemas (those with `type: object`) can have the `properties` property for describing the fields. Array schemas must have the `items` property for describing the items within the array, where the value of `items` is another schema.
- Schemas of type `string` can have `minLength` and `maxLength` fields to limit the size of the string. While schemas of type `number` can have `minimum` and `maximum` fields to limit the range of that number.
- String schemas can also use the `patterns` field to limit the string to match a regular expression. OpenAPI makes use of JavaScript variant of regular expressions (or RegExp).

# Creating resources

In previous chapters we learned a little about using Postman, and in one of those examples we learned how to create new reviews in the FarmStall API. Creating those reviews required executing a POST operation with a request body. Creating reviews happens to be a critical part of this API, what good is a review centric API without the ability to create reviews!

What we'll be doing this chapter is describing the operation to create new reviews, ie: `POST /reviews`. In addition to that, we're going to take a look at `GET /reviews/{reviewId}`. This GET operation interests us for two reasons, firstly we'd like to confirm that we did indeed create a new review by fetching the same review back again, and secondly we can see how a *path parameter* works.

Part of the charm of this approach is using the API itself to verify our work.

Similar to a response body, request bodies are described using OpenAPI's JSON Schema variant and require a media type to indicate the type of data being sent (ie: `application/json`).

What we'll be touching on

- Describe `POST /review` to create new reviews using a request body
- Create new reviews using Try-it-out in SwaggerEditor
- Describe `GET /review/{reviewId}` including its *path parameter*
- Verify that our newly created reviews are really created using SwaggerEditor's Try-it-out

The screenshot shows the Postman interface with a POST request to the endpoint `/reviews`. The request body is a JSON object with fields `message` and `rating`. The response body is a JSON object with fields `uuid`, `message`, `rating`, and `userId`. Red arrows point from the `message` field in the request body to the `message` field in the response body, and from the `rating` field in the request body to the `rating` field in the response body.

```

POST /reviews
{
  "message": "Was pretty good.",
  "rating": 4
}

Response Body (201)
{
  "uuid": "f7f680a8-d111-421f-b6b3-493ebf905078",
  "message": "Was pretty good.",
  "rating": 4,
  "userId": null
}

```

Figure 6.1 Postman creating a new review

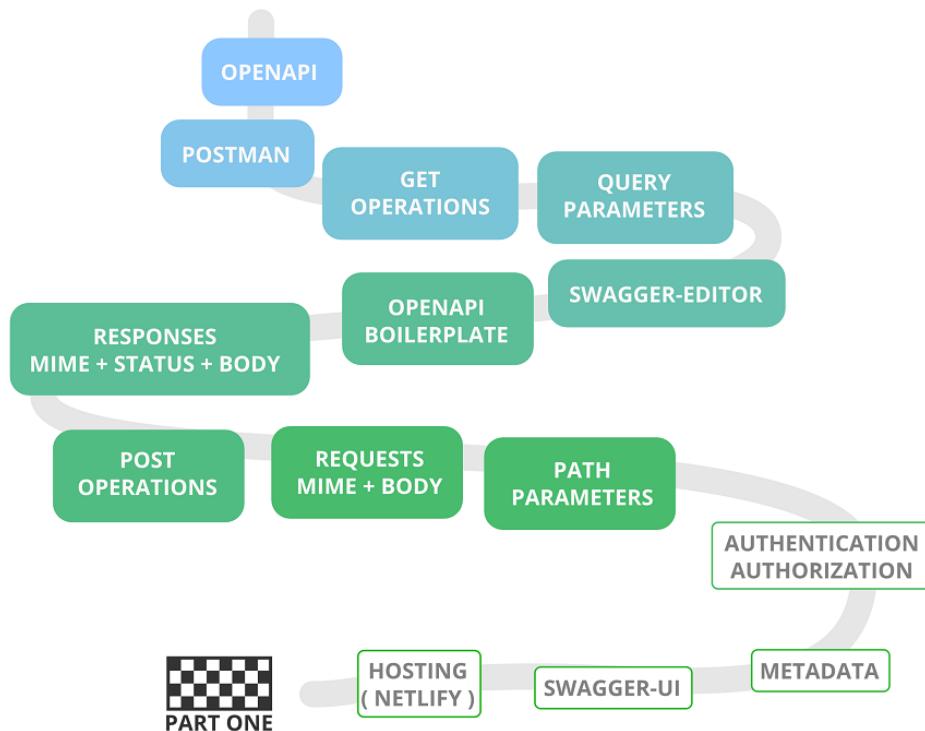


Figure 6.2 Where we are

## 6.1 The problem

Right... let's get set up to create reviews! We're going to look at `POST /reviews` first. To describe it we'll need to know the details of the operation, including both the request and response.

For `POST /reviews`, the response body's schema will need to cover the following:

An object with these fields...

**Table 6.1 POST /reviews request body**

Field	Type	Description
message	string	The message of the review
rating	number	Whole number between one and five inclusive
userId	string	The ID (UUID v4) of the author or null for anonymous

We also want to describe the (successful) response that will be returned from this operation, as it includes something of interests — it includes the server generated ID of the review.

Another point to mention is that the API could have used the response code `200 OK` but there is a more specific response code when creating new resources and that's the `201 Created` code.

Here are the details of that response...

**Table 6.2 POST /reviews responses**

Status	Body	Description
201 Created	Review	Successfully created a new review

The `Review` object is described below...

**Table 6.3 Review schema**

Field	Type	Description
message	string	The message of the review
rating	number	Whole number between one and five inclusive
userId	string	The ID (UUID v4) of the author or null for anonymous
uuid	string	The ID (UUID v4) of this review

As we can see, it's pretty much all been covered before in previous chapters, but with a few extra bits thrown in there.

Points of interest:

- The `uuid` is missing from the request body as it will be created by the server
- The response isn't `200 OK` but the more specific `201 Created`
- The response body includes the server-generated `uuid`

And to give you a mental image of where we'll be describing the body, there is the following:

## Listing 6.1 Where request body goes in OpenAPI

```
openapi: 3.0.0
# ...
paths:
  /reviews
    post: ①
      requestBody: # ... ②
```

- ① Only some methods are allowed a `requestBody`, and `POST` is one of them
- ② The request body will go here

Shall we start describing? Hell yeah...

**NOTE**

When designing APIs it's helpful to understand the semantics of each method such as `GET` and `POST`. These methods are described in the HTTP 1.1 specification, but a lighter introduction to the semantics can be found in [developer.mozilla.org/en-US/docs/Web/HTTP/Methods](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods) .

## 6.2 Describing `POST /reviews` with a request body

I love request bodies as they hint at changing the world.

Unlike `GET` requests, which just fetch some data, `POST` requests are far more active and exciting. The semantics of a `POST` request, is to create a new resource each time it is executed, ie: it *isn't* idempotent <sup>8</sup>. Due to the flexibility of what a "resource" can be, the `POST` request could do anything (and often does many different things), such as launch a rocket, sell a company's stock, sign a peace treaty (that'd be pretty cool) or... it could create a new review in the FarmStall API.

**NOTE****To give you an example...**

To give you an example of a POST operation, I'll relate the following anecdote can relate an anecdote. My friend runs a computer shop that in addition to selling parts and supplies, it also services computers and fixes them. That last part was being tracked by writing out "job cards" by hand and keeping those cards near the PCs that needed servicing. There were some problems with that, from notifying customers when a computer was ready for pickup and not least of which... was not losing the job card! To show off my API skills to my friend (I'm pretty sure most innovations in this world start with the words "Hold my beer and watch this...") I went on to code and wire up a service that would collect the "job card" data in a simple form, send that to an online tool for managing "jobs" (ie: a Trello board) and finally send a message the customer's phone (Twilio) notifying them that their job was underway and included a link to see the progress. In each of those steps a `POST` request was used. A `POST` to send the customer data into my service, a `POST` create the new Card in the online tool and a `POST` to send a message to the customer. `POST`'s are what drive the API world forward. They are the active part of an API and for that alone — they are thrilling!

Request bodies are data in the same way that response bodies are data, and as such they are described in the same way. In this section we're going to describe the `POST /reviews` request and create some reviews. The OpenAPI definition will look like this, when we're done...

## Listing 6.2 POST /reviews description

```

openapi: 3.0.0
info:
  version: v1
  title: FarmtStall API

servers:
- url: https://farmstall.ponelat.com/v1

paths:
  /reviews:
    get: #... left out for brevity
    post:
      description: Create a new Review
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                message:
                  type: string
                  example: An awesome time for the whole family.
                rating:
                  type: integer
                  minimum: 1
                  maximum: 5
                  example: 5
      responses:
        '201':
          description: Successfully created a new Review
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
                    example: An awesome time for the whole family.
                  rating:
                    type: integer
                    minimum: 1
                    maximum: 5
                    example: 5
                  userId:
                    type: string
                    nullable: true
                    pattern: '[a-zA-Z-.0-9]{36}'
                    example: f7f680a8-d111-421f-b6b3-493ebf905078
                  uuid:
                    type: string
                    pattern: '[a-zA-Z-.0-9]{36}'
                    example: f7f680a8-d111-421f-b6b3-493ebf905078
    /reviews/{reviewId}:
      get:
        description: Get a single review
        parameters:
        - name: reviewId
          in: path
          required: true
          schema:
            type: string
            minLength: 36
            maxLength: 36
            pattern: '[a-zA-Z0-9-]+'
        responses:

```

```

'200':
  description: A single review
  content:
    application/json:
      schema:
        type: object
        properties:
          message:
            type: string
            example: An awesome time for the whole family.
          rating:
            type: integer
            minimum: 1
            maximum: 5
            example: 5
          userId:
            minLength: 36
            maxLength: 36
            pattern: '^[a-zA-Z0-9-]+$'
            nullable: true
            example: f7f680a8-d111-421f-b6b3-493ebf905078
          uuid:
            minLength: 36
            maxLength: 36
            pattern: '^[a-zA-Z0-9-]+$'
            example: f7f680a8-d111-421f-b6b3-493ebf905078

```

Let's get going.

### 6.2.1 Where to find request bodies

There can only be one `requestBody` per operation, but each media type can describe its own shape and each shape can be made to fit many different bodies (we'll look into *how* that can be done in later chapters). That means you could conceivably describe two, randomly different, bodies if you so choose. That would be a poor design choice, just saying, but there is merit in describing slightly different bodies to match the media types when required.

Also worth noting is that *only some operations* are allowed to have request bodies. The notable ones that *aren't* allowed them are GET and DELETE. *Technically* you could include a request body, but the HTTP specification doesn't like it and servers that implement the specification to the letter SHOULD ignore those bodies — so just don't do it.

Request bodies are described at the root of the operation...

### Listing 6.3 Where are request bodies

```
openapi: 3.0.0
# ...
paths:
  /foo:
    post: ①
      #...
      requestBody: ②
        description: Return a bar after creating a foo
        content: ③
          application/json: ④
            schema: ⑤
              type: object
              properties:
                bar:
                  type: string
```

- ① POST methods are allowed request bodies (encouraged actually).
- ② The `requestBody` keyword.
- ③ The `content` or *data* goes here.
- ④ The media type of the data — `application/json`.
- ⑤ The schema of our data.

Excellent! The anatomy of our request body has shown itself. Given the previous chapter, adding in the schema for the review, should be a breeze to describe. Or at most... a mildly strong breeze!

It is in fact a rather close copy of work we've done previously, as we've already described the schema of a Review before. However instead of an array of reviews we're describing a single review, and in addition we're going to remove the `uuid` field, as that will be generated by the server.

This request body has the following fields, to be described... `message`, `rating` and `userId`. Do you think you can describe it from memory? Don't fret, it's not that critical that we do so from memory.

Let's add in the details for this request body schema, so we can test it out. The schema on its own will look like the following...

## Listing 6.4 A new review schema

```

type: object ①
properties:
  message: ②
    type: string
  rating: ③
    type: integer
    minimum: 1
    maximum: 5
  userId: ④
    type: string
    pattern: '^[0-9a-fA-F\\-]{36}$'
    nullable: true
  
```

- ① Describing a single object.
- ② The `message` field, a string with no limits.
- ③ The `rating` field, a whole number between one and five.
- ④ The `userId` field, a UUID that can optionally be `null`

Now we can add it into the request body section of our operation. And while we're doing that we'll sneak in the response body as well — which is quite similar (it is the same, except for including the `uuid` field)...

## Listing 6.5 Request body with schema

```

openapi: 3.0.0
info:
  version: v1
  title: FarmStall API
servers:
- url: https://farmstall.ponelat.com/v1
paths:
  /reviews:
    # get: ... ①
    post: ②
      description: Create a new review ③
      requestBody: ④
        description: A new review object
        content:
          application/json:
            schema: ⑤
              type: object
              properties:
                message:
                  type: string
                rating:
                  type: integer
                  minimum: 1
                  maximum: 5
                userId:
                  type: string
                  pattern: '^[0-9a-fA-F\\-]{36}$'
                  nullable: true
      responses:
        '201': ⑥
          description: Successfully created a new Review
          content:
            application/json:
              schema: ⑦
                type: object
                properties:
                  message:
                    type: string
                    example: An awesome time for the whole family.
                  rating:
                    type: integer
                    minimum: 1
                    maximum: 5
                  userId: ⑧
                    type: string
                    nullable: true
                    pattern: '[a-zA-Z-.0-9]{36}'
                  uuid: ⑨
                    type: string
                    pattern: '[a-zA-Z-.0-9]{36}'

```

- ① Our GET /reviews operation redacted for brevity.
- ② We're creating a POST method under the /reviews path.
- ③ All operations deserve (and require) a description for humans.
- ④ The buzz word of this chapter requestBody, where our request body will go.
- ⑤ The schema we copied and tweaked from a previous chapter.
- ⑥ We always need to describe a response, and we get to use the different and specific 201 Created status code.

- ⑦ This schema is the same as the `requestBody` except for the addition of ...
- ⑧ The `userId` field and...
- ⑨ The `uuid` field, which will have a our server-generated `uuid`.

That was quiet a mouthful getting all that into our definition. Time for a breather of theory, let's copy that into SwaggerEditor and create some reviews!

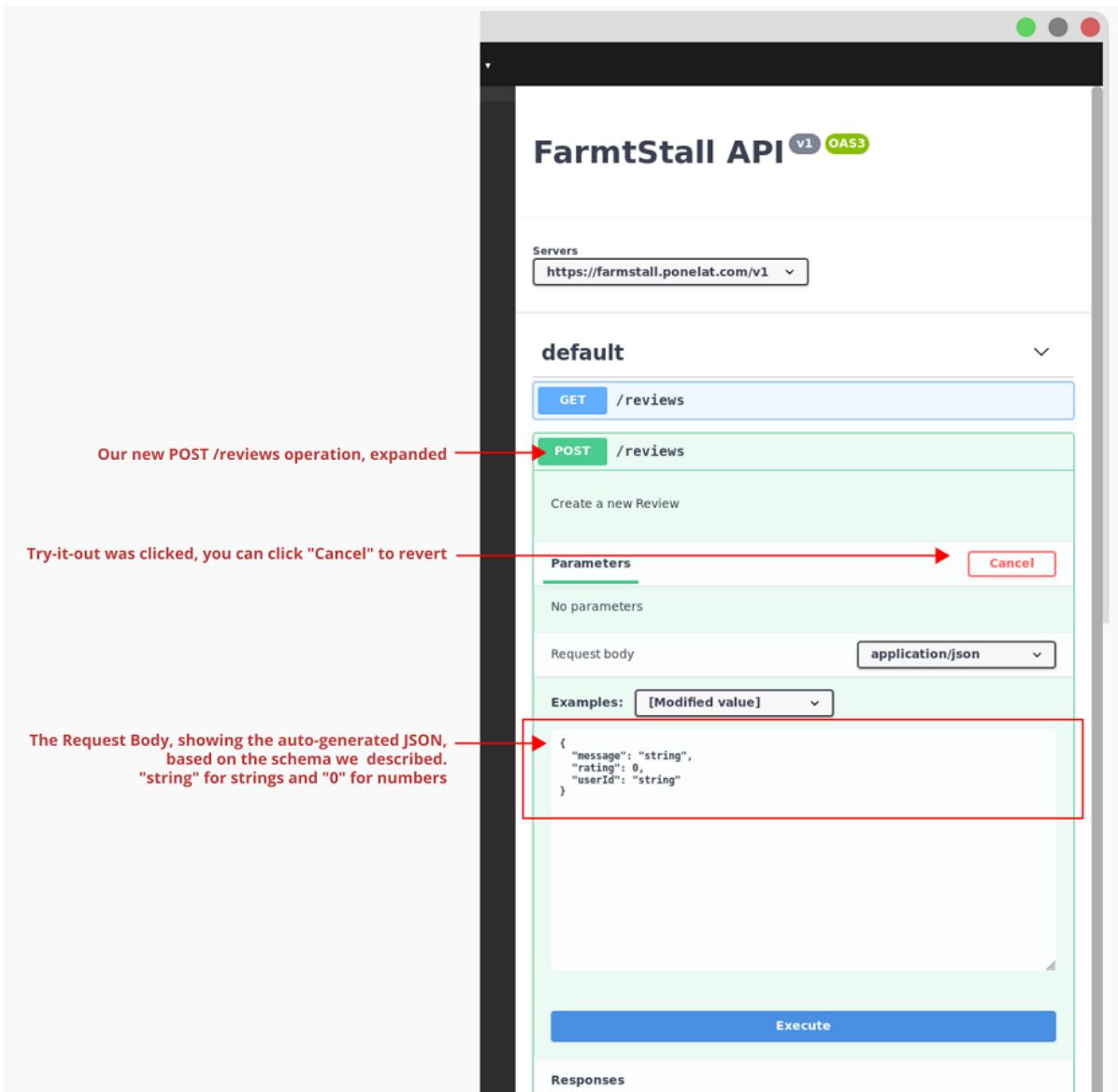
## 6.3 Executing operations with request bodies

SwaggerEditor's Try-It-Out feature supports request bodies, much like you'd expect.

The way it does it, is by generating an example JSON (or XML) string and placing that inside a Textarea. The user can then modify the Textarea (ie: the body) and execute the request. When we say "modify", we mean quite manually, as it is *just a Textarea input*. I'm sure in the near future there will be fancier ways to edit request bodies, but at the moment we simply tweak the text as we see it. We should be careful when working with the raw text, since JSON (and XML) have syntax rules that should be obeyed.

The JSON (or XML) string that SwaggerEditor generates is based on the schema we've provided. SwaggerEditor will build one based on the shape it expects. It may not be glamorous (strings will be `string` and numbers will be `0`) but it will be functional.

Let's try out, the Try-It-Out!



**Figure 6.3 Try-It-Out executing POST /reviews request**

After clicking on the "Try it" button we should see the request body become editable. Go ahead and change it to the following...

#### **Listing 6.6 Editing the request body in SwaggerEditor**

```
{
  "message": "Totally awesome",
  "rating": 5,
  "userId": null
}
```

It should then look like this...

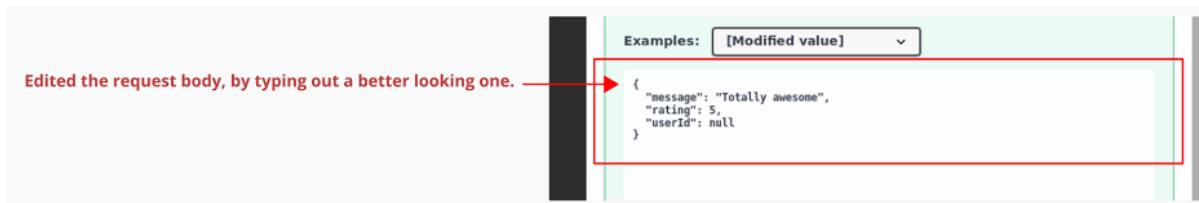


Figure 6.4 Text area of request body — edited

Go on and hit the execute button! After we press the big blue "Execute" button, the browser will execute the request and the server will receive it, internally create a new review and then send a response back to the browser (and us). This response will have the generated UUID inside of it.

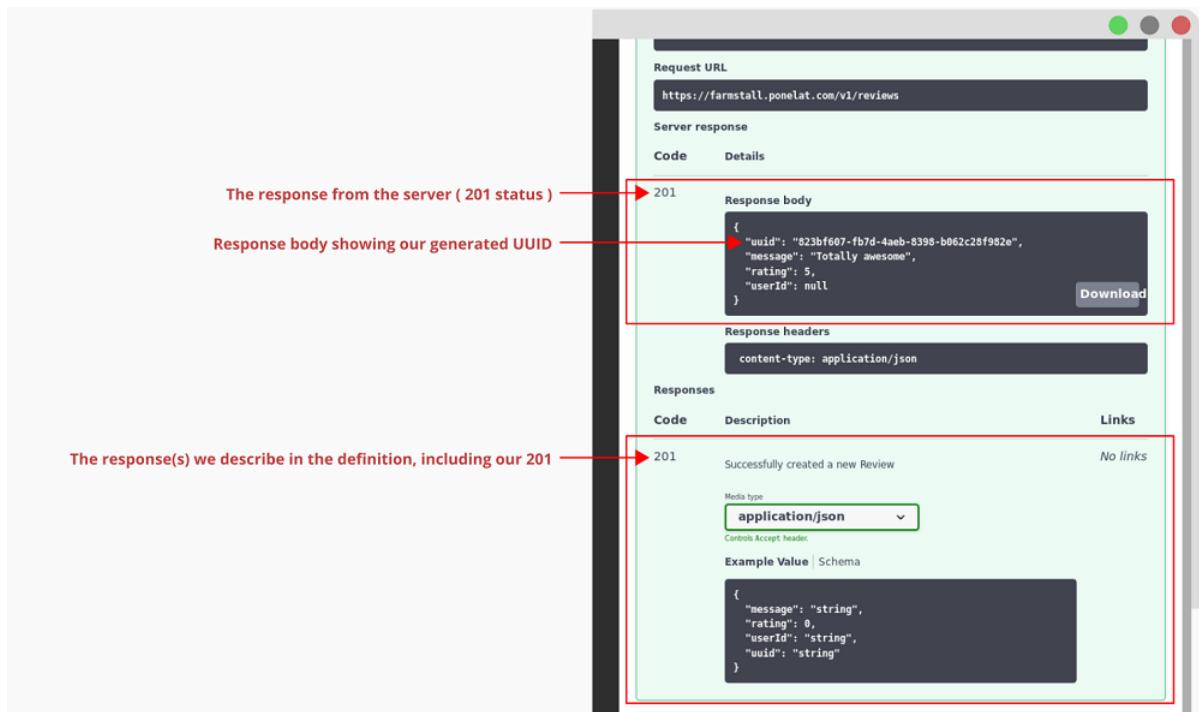


Figure 6.5 201 response from POST /reviews

Later on, we'll be using that UUID to verify that we actually created a review, by fetching it (and only it) back from the server.

**WARNING** **Trailing commas**

When editing JSON by hand we must look out for *trailing commas*. While all fields and array items are separate by a comma, the list field or item should NOT have a comma. This restriction in JSON is a bit of a pain, as JavaScript allows trailing commas. Just pointing it out to avoid some pain when working with JSON!

**Listing 6.7 Trailing comma ( INVALID )**

```
{
  "foo": true, ①
  "bar": true, ②
}
```

- ① This is valid, as fields are separate by a comma.
- ② This is invalid, the last field cannot end with a comma.

Do you know what's really missing? A little bit of developer love. We have no examples for the API consumers to draw from and OpenAPI has a way to show examples...

### 6.3.1 Adding examples to make try-it-out look pretty

Our schemas look dry. Strings, numbers, more strings. What joy are we imparting to consumers when they read this? It could be one of those Mondays, where nothing is going right, and we're not helping. Let's add some love to the definition, by showing there are humans behind these schemas.

Examples give us the gist of data much faster than reading schemas, which are precise but verbose.

Consider the following... Imagine you see the field — `name`. What can you grasp from it? Well, it could be a full name like `Josh Ponelat` or it could be a user's name like `ponelat`. An example can go that extra mile and help your consumers out by showing real data.

We have one field where we can unleash our inner creative beasts. That field is `message`, as it is a free-style string, by humans for humans. We also have `rating` but it hardly allows much creativity.

Let's create some fun examples:

## Listing 6.8 Example Review 1

```
{
  "message": "The utter worst experience of my life, I feel bad, simply recalling it.",
  "rating": 1
}

{
  "message": "My heart burns with anticipation of my next visit. It was breathtaking.",
  "rating": 5
}

{
  "message": "Completely average. Like the colour grey.",
  "rating": 3
}
```

There, that looks more interesting! Our attention to the developer experience will get noticed by others, especially on those particularly tough Mondays.

OpenAPI has places for these examples, on different levels. You can put an example on each individual field or on the whole schema itself — both can be useful.

Let's add a basic example to each individual field within our request body schema...

## Listing 6.9 Examples for requestBody

```
requestBody:
  description: A review object
  content:
    application/json:
      schema:
        type: object
        example: ①
          message: A lovely experience ②
          rating: 4 ③
        properties:
          message:
            type: string
            example: Blew my mind, life won't be the same after this. ④
          rating:
            type: integer
            minimum: 1
            maximum: 5
            example: 5 ⑤
```

- ① The example for the whole object will take precedence over the individual examples found on each individual field. The example must also be valid, according to the schema described (makes sense).
- ② Part of the example, we're showing `message` here.
- ③ Another part of the example, here we're showing what `rating` could look like.
- ④ This will help consumers who are deep down in your schema. And if you don't provide examples higher up (like on the root object) then they'll be composed into an example for you
- ⑤ ...and an example for the field `rating`

Go ahead and add some examples to your definition, it'll help when you want to use the Try-It-Out feature, as it'll generate more pleasing request bodies!

Now, getting back to the task at hand...

## 6.4 Describing `GET /reviews/{reviewId}` with a path parameter

Now to validate that we did actually create a new review on the server. It's all good and well that we think we made a dent in the number of reviews but without checking, did it really happen? To test that assumption, we're going to kill two birds with one stone.

First we're going to describe an operation that needs to be described, and second, we're going to use it (by executing it) to confirm that our reviews were indeed created.

We're talking about the humble `GET /review/{reviewId}` operation.

Right away we can see it is a little special, since the path includes some curly brackets in it! The name surrounded by curly brackets, is known as a *path parameter* in OpenAPI parlance.

The requirements of `GET /review/{reviewId}` as summarized by the following...

**Table 6.4 Parameters of `GET /review/{reviewId}`**

Parameter	In	Type	Description
reviewId	path	string (UUIDv4)	The ID of the author, required.

The operation has a response body of a single review — something we've already described before...

**Table 6.5 Getting a single review response body**

Field	Type	Description
message	string	The message of the review
rating	number	Whole number between one and five inclusive
uuid	string ( UUIDv4 )	The ID of this review
userId	string (UUIDv4) or null	The ID of the author, or null for anonymous

This is the same schema we've already described in `POST /reviews`. So it makes sense to just copy it over...

## Listing 6.10 The GET /review/{reviewId} operation

```
openapi: 3.0.0
paths:
  #...
  /reviews/{reviewId}:
    get:
      description: Get a single review object
      responses:
        '200':
          description: Review object
          content:
            application/json:
              schema: # ...Copy from POST /reviews
              type: object
              properties:
                message: # ...
                rating: # ...
                userId: # ...
                uuid: # ...
```

What remains is to describe the `reviewId` path parameter, let's go take a closer look at that...

### 6.4.1 Path parameters

Path parameters are described in the same way as as query parameters. Each parameter requires the following properties to describe it...

- `name` — name of the parameter.
- `in` — the location of the parameter (query, path, header, cookie).
- `schema` — schema of the parameter (string, number, array, boolean, null).

Parameters can also include the following properties (we'll cover some of them later on)...

- `required` — Whether or not this parameter is required (which for path parameters must **ALWAYS** be true).
- `example` — An example of the parameter's value.
- `examples` (plural) — List of examples, which is mutually exclusive with `example`.
- `deprecated` — Whether or not this parameter is marked as deprecated.
- `style` — How the value will be serialized.
- `explode` — Whether or not to create a separate value for arrays/objects.
- `allowedReserved` — Allow reserved characters (ie: / or ?). Useful for when you want a catch-all parameter.

For this parameter we're going to stick to the basics — let's get describing...

### 6.4.2 Describing the path parameter

To describe the path parameter, we follow the same pattern as describing query parameters. Looking at the path parameter in isolation we'd see the following...

## Listing 6.11 Path parameter

```
parameters:
- in: path ①
  name: reviewId
  required: true ②
  schema: ③
    type: string
    description: The review's ID
  example: 3b5b1707-b82c-4b1d-9078-157053902525 ④
```

- ① This is the critical piece. Here we say it's a "path" parameter and not a "query" parameter.
- ② This is necessary boilerplate as all "path" parameters are required.
- ③ As for all parameters we include a schema.
- ④ Hell yeah, we want examples everywhere!

Note that besides, `in` being set to `path`, the only extra field we require (compared to query parameters) is the `required` field. A path parameter cannot be optional according to the OpenAPI spec.

Adding the parameter into our definition should be straightforward now...

## Listing 6.12 Get /review/{reviewId}

```
openapi: 3.0.0
paths:
  #...
  /reviews/{reviewId}:
    get:
      description: Get a single review object
      parameters: ①
        - in: path
          name: reviewId
          required: true
          schema:
            type: string
            description: The review's ID
            example: 3b5b1707-b82c-4b1d-9078-157053902525
      responses:
        '200':
          description: Review object
          content:
            application/json:
              schema: ②
                type: object
                example: # ...
                properties:
                  message: # ...
                  rating: # ...
                  userId: # ...
                  uuid: # ...
```

- ① Our path parameter, added in.
- ② The schema for a single review, redacted for brevity.

**NOTE****You may be wondering...**

Given how similar all these schemas are, surely there is a way to reuse them? Bravo! That's an excellent question, and worth a good talking about. We'll be covering composition and polymorphism later on in the book where we'll look at `$refs`, `allOf`, etc. For now don't mind the verbosity of duplicating all those fields ( like `message`, `rating`, etc ), rest assured there are ways of reducing the duplication.

Here is the high level view of the OpenAPI definition thus far. It can be tough to see the forest for the trees...

### **Listing 6.13 High level view of definition so far**

```
openapi: 3.0.0
info:
  title: FarmStall API
  version: v1
paths:
  /reviews: ①
    get: # ...
    post: # ...
  /reviews/{reviewId}: ②
    get: #... ③
```

- ① Our reviews path with the two methods underneath (get and post).
- ② Our new path which includes a path parameter
- ③ ...and its method: get

Now to verify that our we have indeed created a new review on the system, when we execute `POST /reviews`, by utilizing `GET /reviews/{reviewId}`...

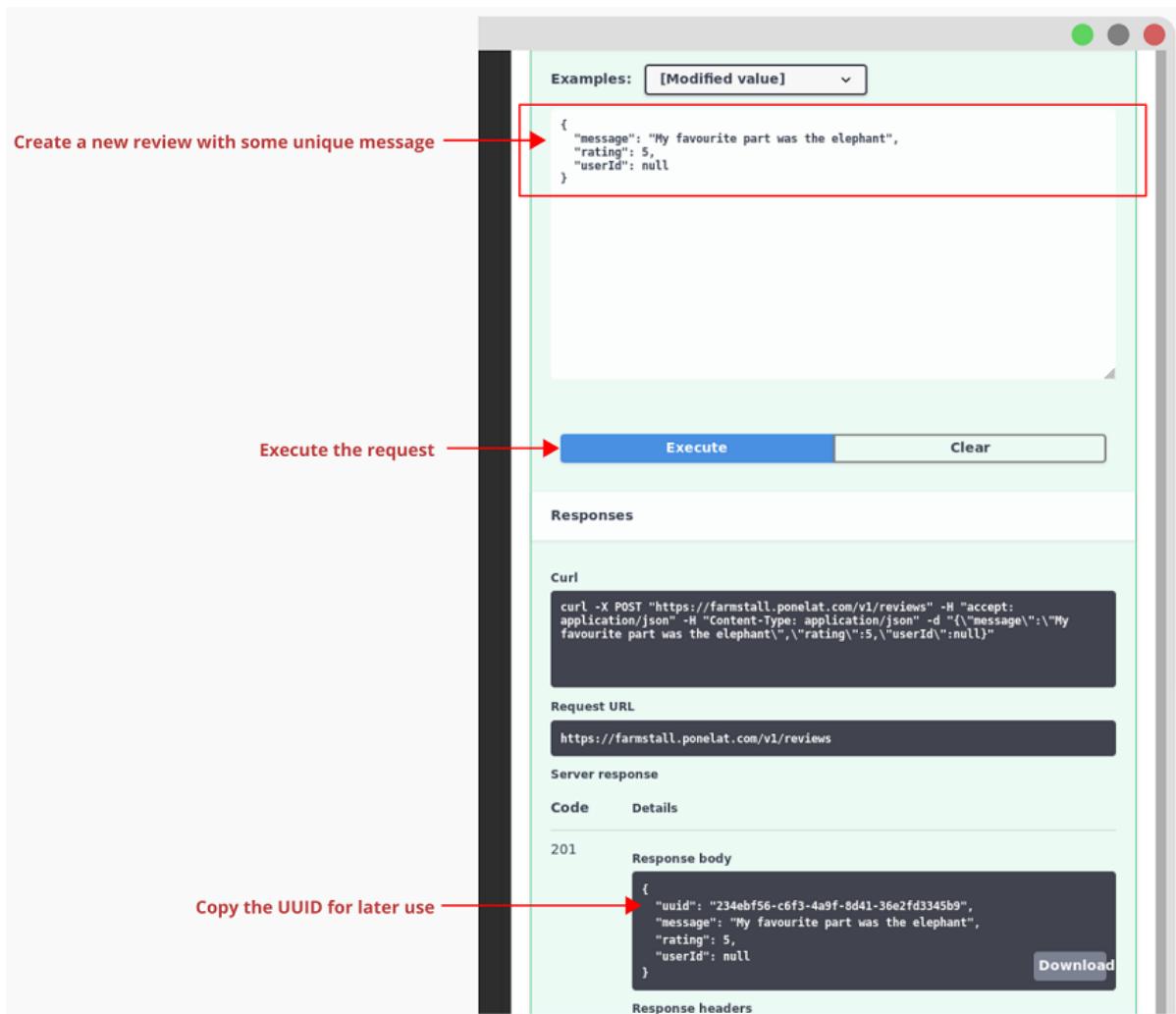
## **6.5 Verifying our reviews are getting created**

To accomplish our goal of verifying whether or not we've created a review. We want to...

1. Create a review with `POST /reviews`.
2. Copy/remember the review ID we get back in the response.
3. Execute the `GET /reviews/{reviewId}` (using the review ID we copied).
4. Ensure that the response is expected, ie: that it's the same review we just created.

Using SwaggerEditor, execute a `POST /reviews` operation. Add something memorable and unique in the `message` field, so that you know it is your own. When the operation returns a response, copy the `uuid` field and put it into notepad or some other text editor — so that we can use it again later.

Example...



**Figure 6.6 Creating a review noting the ID**

Now to fetch the same review by using `GET /review/{reviewId}`, confirming that we created a review on the system.

Using SwaggerEditor, expand out the `GET /review/{reviewId}` operation. After clicking the *Try-It* button and the previously captured `uuid` field into the `reviewId` input box. Then go ahead and execute the request.

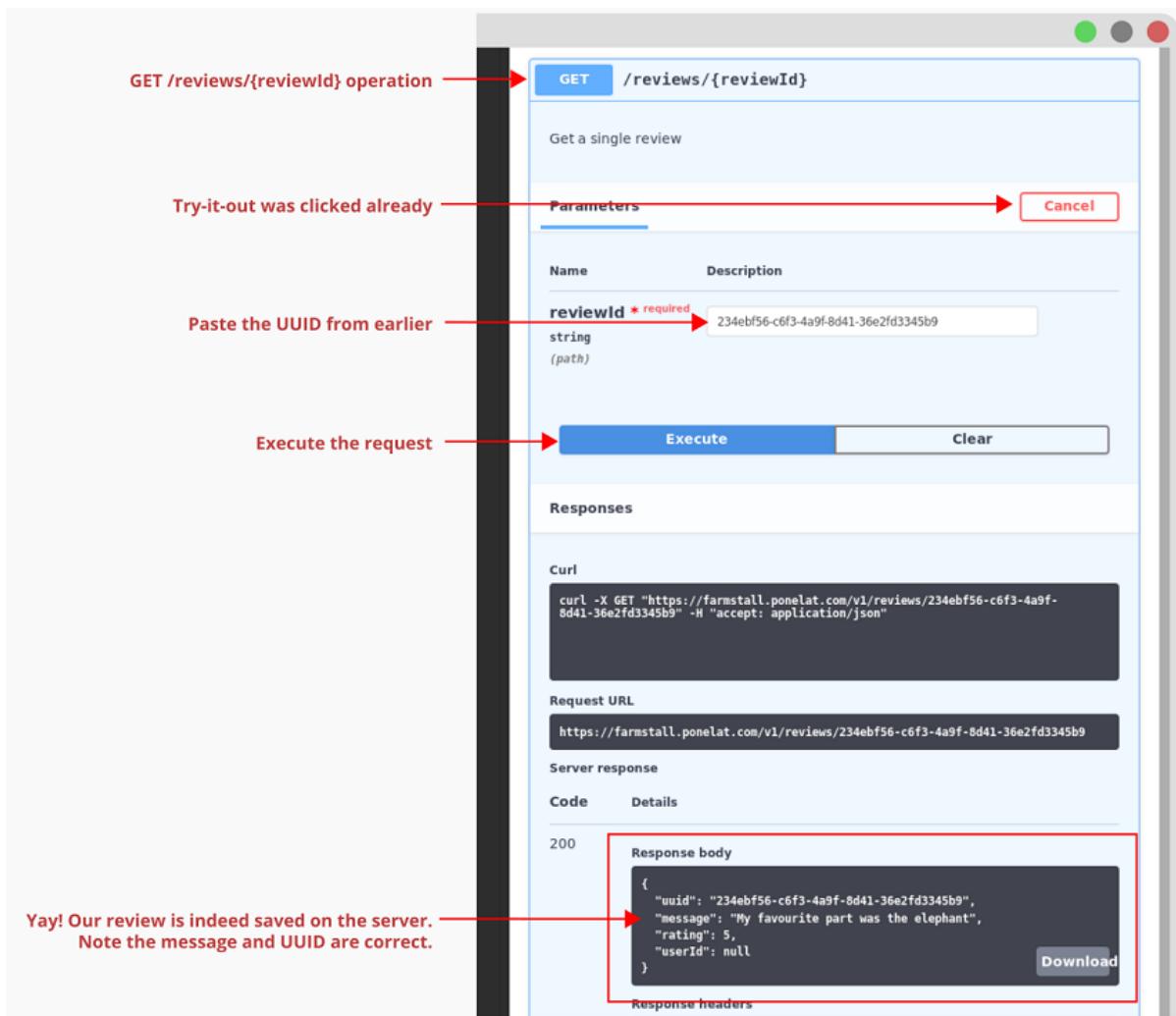


Figure 6.7 Fetching a review with by ID

Congratulations. If all went well, you should see the response of `GET /reviews/{reviewId}`, which will include the `uuid` you used as well as the other details of the review, that was recently created.

From this exercise we were able to verify that our `POST /reviews` operation did indeed create a new review. By copying the `uuid` from the response of `POST /reviews` and using it as the path parameter we were able to fetch *that review* back again.

All this while describing our FarmStall API. What fun :)

## 6.6 Summary

- POST is described in the same fashion as GET but can include the `requestBody` property.
- Request bodies are added underneath the `requestBody` field inside an operation. It has the high level fields, `description` and `content`. `content` will include fields for each media type ( eg: `application/json` ) and those media types will include the `description` and `schema` properties.
- Request bodies are described in the same way as response bodies, with media types and schemas.
- Examples help consumers understand the data more quickly and can be added to each field , where more parent examples will take precedence over child examples. Ie: A Review example will take precedence over the example inside the Review's message field.
- Path parameters are declared in the path and MUST be described by the operations under that path (ie: `get`, `post`, etc).
- Using the Try-It-Out feature is great for interacting with the API as you describe it, and can be used to verify the functionality of your API.



# *Adding Authentication and Authorization*

## This chapter covers

- Identifying the difference between authentication and authorization.
- Adding operations for creating users.
- Adding an operation for getting a user's token (authentication).
- Adding the Authorization header to POST /reviews operation (authorization).

We're going to be looking at authentication and authorization, two close friends in APIs that are often a little misunderstood. Authentication is about proving you are, who you say, you are — which could be through a username and password. While authorization is about being allowed to access to special resources or actions that are normally private. Like getting user details or creating a new review.

APIs almost always include a form of authorization (and of course a type of authentication), so naturally describing them is important. In today's world we have multiple standards dealing with authorization, each with different tradeoffs and strengths, and we should communicate to our consumers which of these standards we use.

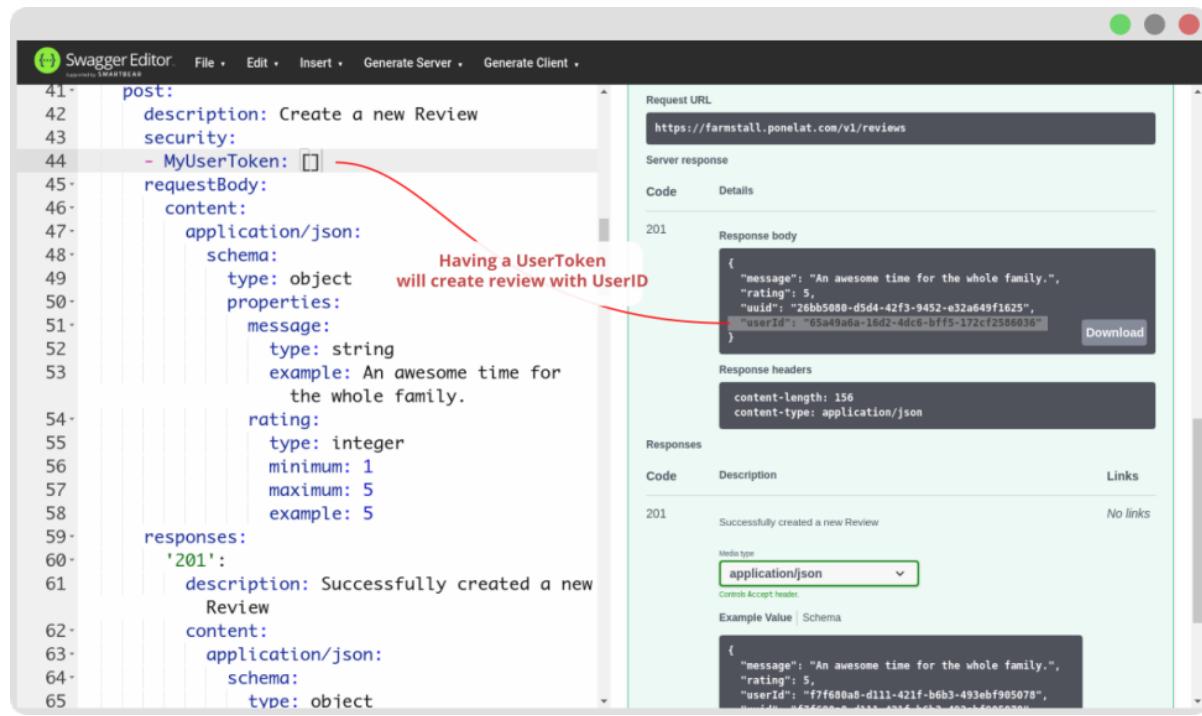
Personally, one of the biggest hurdles to using an API, is getting authorization to work. I've found myself wading through documentation just to be able to use the API!

At the end of this chapter you'll be able to describe simple security schemes for authentication/authorization and add them to operations in OpenAPI.

In our FarmStallAPI we'll be adding - POST /users - POST /tokens - The Authorization header to POST /reviews

Ultimately being able to create new reviews as a given user. Using SwaggerEditor to describe

and test it out, with results like the following...



```

41- post:
42-   description: Create a new Review
43-   security:
44-     - MyUserToken: []
45-   requestBody:
46-     content:
47-       application/json:
48-         schema:
49-           type: object
50-           properties:
51-             message:
52-               type: string
53-               example: An awesome time for
54-                 the whole family.
55-             rating:
56-               type: integer
57-               minimum: 1
58-               maximum: 5
59-               example: 5
60-   responses:
61-     '201':
62-       description: Successfully created a new
63-         Review
64-       content:
65-         application/json:
66-           schema:
67-             type: object

```

Having a UserToken will create review with UserID

```

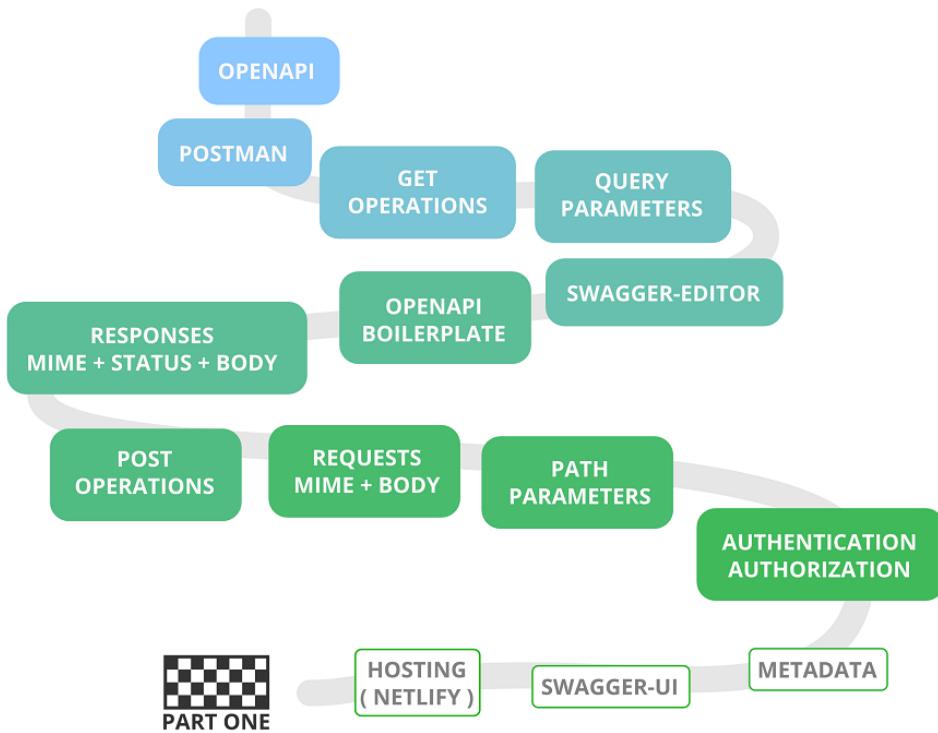
{
  "message": "An awesome time for the whole family.",
  "rating": 5,
  "uuid": "26bb5000-d5d4-42f3-9452-e32a649f1625",
  "userId": "65a49a6a-16d2-4dc8-bff5-172cf2586038"
}

```

Figure 7.1 SwaggerEditor Successfully created a review with Frank Abagnale's uuid

Above you'll note the extra `userId` field in the `POST /reviews` response. As well as the extra OpenAPI syntax for adding a security requirement for a `MyUserToken: []`. Let's make that happen.

For this chapter we'll be building on top of our existing definitions. If you don't have it handy here is a link to get a copy... [app.swaggerhub.com/apis/designing-apis/part-one/ch07-start](https://app.swaggerhub.com/apis/designing-apis/part-one/ch07-start)



**Figure 7.2 Where we are**

## 7.1 The problem

In this chapter we want to describe the authorization and authentication of the FarmStall API so that our consumers can know how to use those operations that require it.

In particular we want to describe the Authorization header in `POST /reviews` operation and the operations necessary to get the token used in that header.

### 7.1.1 The flow of `POST /reviews`

Looking at the following diagram we can learn more about the `POST /reviews` operation and how it handles Authorization...

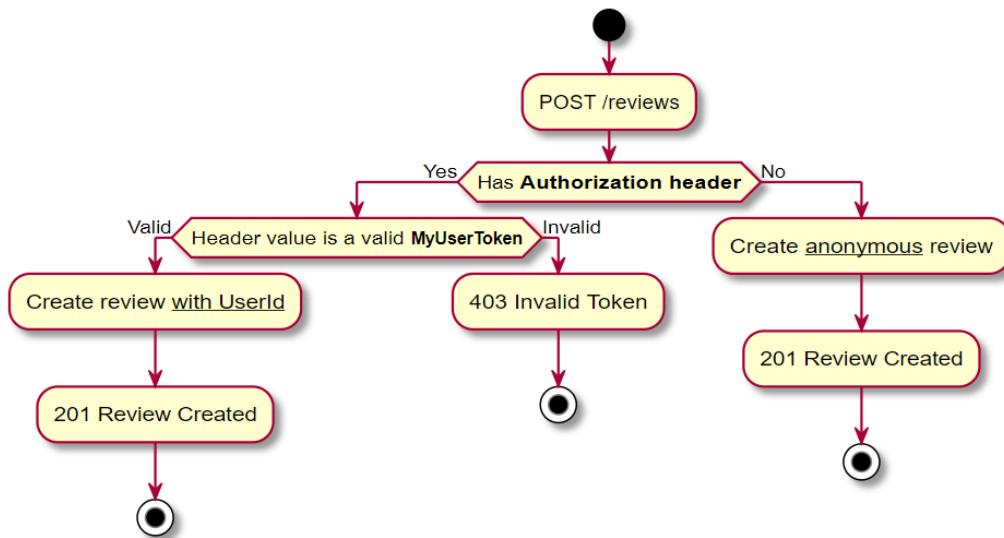


Figure 7.3 Diagram of authorization in POST /reviews

By adding a Authorization header, which has a valid "MyUserToken", the operation will create a review and populate the `userId` field. If the MyUserToken is invalid it'll return an error and finally, if we DO NOT provide an Authorization header then it'll create an anonymous review (ie: `userId` is `null`). Which can also be summed up with the following table...

Table 7.1 Authorization flow of POST /reviews

Authorization header	Valid MyUserToken	Result
Yes	Yes	201 Created Review (with <code>userId</code> )
No		201 Created Review ( <code>userId</code> is <code>null</code> )
Yes	no	403 Invalid MyUserToken

Up till now all our reviews have been anonymous, but that will change by the end of this chapter.

Today is not about errors. Today is about the happy path<sup>9</sup> of creating a review *with a UserId* using a valid **MyUserToken**. So how do we do that?

The steps we're going to take:

- Figure out what a "MyUserToken" is and how to get one.
- Describe the requirement of the Authorization header (that will be used in POST /reviews).
- Describe POST /reviews as having that requirement.
- Use SwaggerEditor to create a review as a given user.

Let's get going!

## 7.2 Getting set up for authentication

What is "MyUserToken"?

In the "Diagram of authorization in POST /reviews" above we see "MyUserToken" as something important in determining whether the `POST /reviews` operation is *authorized* to create a review as a given user or not.

This is the name we (FarmStall API) have given to the value of the `Authorization` header, to distinguish it from other values or tokens. The name is arbitrary.

In order to get a "MyUserToken", you need to first register a new user with `POST /users` and then you can call `POST /tokens` which will return a MyUserToken.

In this section we're going to look at the details of those two operations — `POST /users` and `POST /tokens` — but as the operations themselves do not introduce any new OpenAPI concepts, what we will do is leave it as an exercise for you to stretch those newly learned OpenAPI skills.

*Our ulterior motive is that we need these operations described for the next sections around authorization.*

*Also note we only need to describe the success responses and not the failure responses (ie: only the 2xx status codes, not the 4xx).*

We'll outline the requirements of these operations and near the end of the section we'll show how they should work with SwaggerEditor's try-it-out. The definition changes will be at the end of this section, but try taking a stab at describing the operations without peeking too much!

Ready for the challenge?

### 7.2.1 Challenge - Describe `POST /users` and `POST /tokens`

#### **POST /users**

This operation will create/register a new user in the FarmStall API. A constraint is that each `username` be unique to in the system. So when trying out the operations, pick something unique, and note that the FarmStall API periodically resets all its data. Have fun with the names :)

The request body of `POST /users`. See chapter 06 for how request bodies are described.

**Table 7.2 POST /users - request body**

Field	Type	Description
<code>username</code>	string	The <code>username</code> of the user.
<code>password</code>	string (format = <code>password</code> )	The <code>password</code> of the user.
<code>fullName</code>	string	The full name of the user.

The response of `POST /users`. Please note that we're only mentioning the successful response. There are also error responses if you try to create a user with a `username` that already exists. We'll cover errors in a later chapter.

**Table 7.3 POST /users - responses**

Status	Body	Description
201 Created	User	Successfully created a new user

The User response object.

**Table 7.4 User schema**

Field	Type	Description
username	string	The username of the user.
fullName	string	The full name of the user.
uuid	string	The ID of the user, as a UUIDv4.

## POST /TOKENS

This operation will create a MyUserToken for a given user. The user is identified (ie: authenticated) by a `username` and `password` combination. This is perhaps the most common example of authentication in APIs. You'll need to create a user before being able to create a MyUserToken for that user.

The request body of POST /tokens.

**Table 7.5 POST /tokens - request body**

Field	Type	Description
username	string	The username of the user.
password	string (format = password)	The password of the user.

The response of POST /tokens. Please note that we're only mentioning the successful response. There are also error responses if you try to authenticate with invalid credentials.

**Table 7.6 POST /tokens - responses**

Status	Body	Description
200 Success	Token	Successfully created a token

The Token object which is a simple object wrapper around a token string. We wrap it in an object so that its easier to extend in the future, eg: `{ "token" : "abcababcabc" }`.

**Table 7.7 Token schema**

Field	Type	Description
token	string	The token for a given user

Those are the operations you need to describe! We'll include the answers/definition down below. But have a crack at it first using the base definition and SwaggerEditor to add the changes.

Here is the base definition to work from...

[app.swaggerhub.com/apis/designing-apis/part-one/ch07-start](https://app.swaggerhub.com/apis/designing-apis/part-one/ch07-start) (copy the definition into Swagger Editor or fork it in SwaggerHub).

### 7.2.2 Solution - Definition changes

Here is the solution, a definition with the two operations described...

[app.swaggerhub.com/apis/swagger-in-action/part-one/ch07-with-users](https://app.swaggerhub.com/apis/swagger-in-action/part-one/ch07-with-users)

Or just showing the relevant changes...

## Listing 7.1 Two user-related operations

```

openapi: 3.0.0
#...
paths:
#...
/users:
  post:
    description: Create a new user
    requestBody:
      description: User details
      content:
        application/json:
          schema:
            type: object
            properties:
              username:
                type: string
                example: ponelat
              password:
                type: string
                format: password
              fullName:
                type: string
                example: Josh Ponelat
    responses:
      '201':
        description: Successfully created a new user
        content:
          application/json:
            schema:
              type: object
              properties:
                username:
                  type: string
                  example: ponelat
                uuid:
                  type: string
                  example: f7f680a8-d111-421f-b6b3-493ebf905078

/tokens:
  post:
    description: Create a new token
    requestBody:
      content:
        application/json:
          schema:
            type: object
            properties:
              username:
                type: string
                example: ponelat
              password:
                type: string
                format: password

    responses:
      '201':
        description: Create a new token for gaining authenticated access to resources
        content:
          application/json:
            schema:
              type: object
              properties:
                token:
                  type: string

```

### 7.2.3 Verifying we can create users and get a token

Let's showcase how we'll verify getting a MyUserToken. First we'll need to register a user.

*PS: You'll need to change at least the username so that it's unique to you.*

```

126
127  /users:
128    post:
129      description: Create a new user
130      requestBody:
131        description: User details
132        content:
133          application/json:
134            schema:
135              type: object
136              properties:
137                username:
138                  type: string
139                  example: ponelat
140                password:
141                  type: string
142                  format: password
143                fullName:
144                  type: string
145                  example: Josh Ponelat
146
147      responses:
148        '201':
149          description: Successfully created a new user
150          content:
151            application/json:

```

Figure 7.4 Creating a user with try-it-out

```

100
101  /tokens:
102    post:
103      description: Create a new token
104      requestBody:
105        content:
106          application/json:
107            schema:
108              type: object
109              properties:
110                username:
111                  type: string
112                  example: ponelat
113                password:
114                  type: string
115                  format: password
116
117      responses:
118        '201':
119          description: Create a new token for gaining authenticated access to resources
120          content:
121            application/json:
122              schema:
123                type: object
124                properties:

```

Figure 7.5 Create a token from username and password

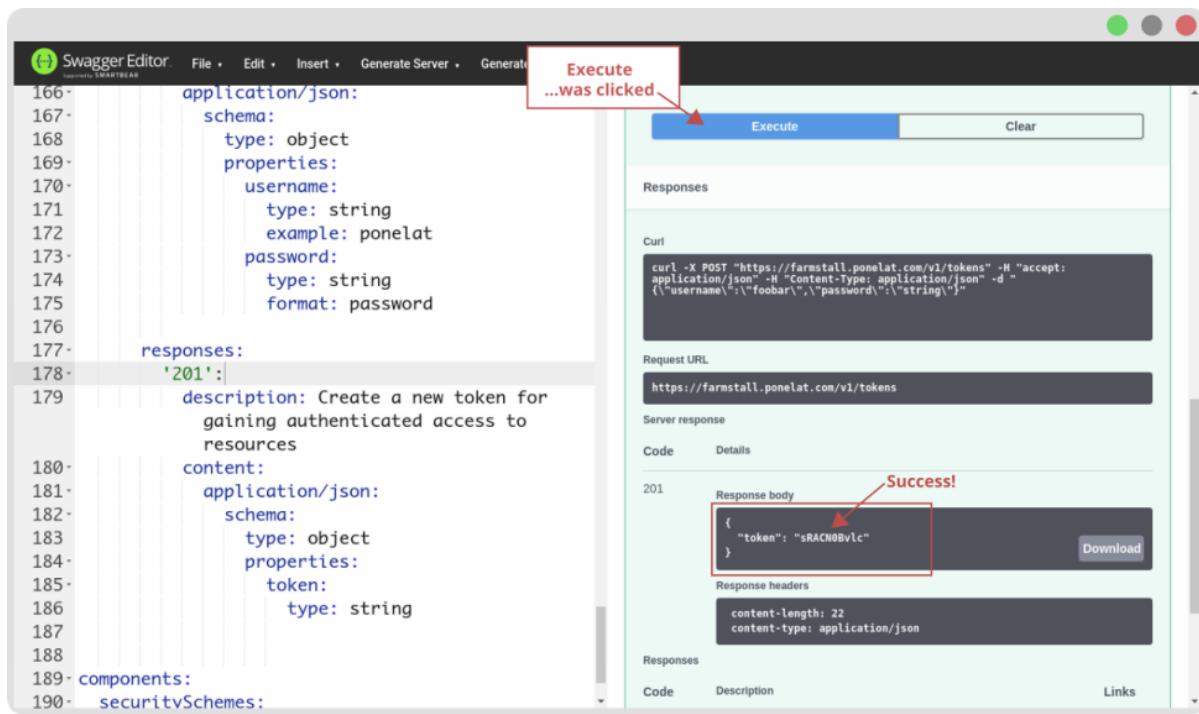


Figure 7.6 Create a token from username and password - Success

Now that we can create MyUserTokens we're ready to add authorization to our `POST /reviews` operation!

## 7.3 Adding the Authorization header

In this section we want to add an `Authorization` header to the `POST /reviews` operation, so that we can create reviews as ourselves and not anonymously. We need to first describe it and then verify we've done it correctly by executing the request.

We'll be adding to the `securitySchemes` component and adding a `security` object to `POST /reviews`, in our OpenAPI definition. Here are the changes we'll be making to our definition...

### Listing 7.2 Bones of the definition we'll be using

```
openapi: 3.0.0
# ...
paths:
  /reviews:
    post:
      # ...
      security:
        - MyUserToken: [] ①
# ...
components:
  securitySchemes:
    MyUserToken: ②
      type: apiKey
      in: header
      name: Authorization
```

- ① Note: the name of the security requirement (MyUserToken is an arbitrary name).
- ② ...which matches the key of the security scheme.

### 7.3.1 How does OpenAPI handle authorization

To describe a security requirements or authorization to an operation you need to do two things...  
 1. Add a `securityScheme` describing the *type of security/authorization*, and give that security a name. 2. Add that security name to the list of required securities in your operation, under the `security` field in the operation.

In the above example we can see that `POST /reviews` has added a security which was described under `securitySchemes` using the key as the name of that security.

A security scheme tells us the requirements of the security, we'll dig into the details shortly, but at a glance we can see the words *header* and *Authorization*. So a reasonable guess is to assume that we're specifying a header called "Authorization".

What types of securities or authorization mechanisms does OpenAPI support in describing? Let's take a look...

### 7.3.2 Types of authorization (securities) supported in OpenAPI 3.0.x

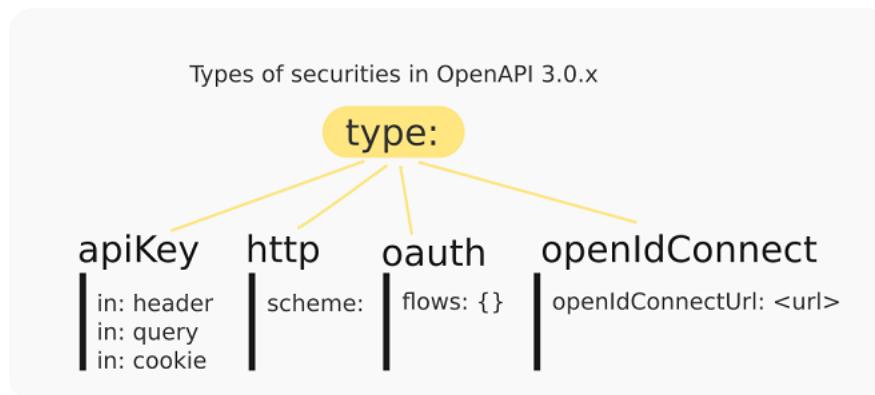


Figure 7.7 Different security scheme types in OpenAPI 3.0.0

OpenAPI 3.0.x supports four categories of security types... - `apiKey` - `http` - `oauth` - `openIdConnect`

The most basic type of security is `apiKey` which describes either a header, query parameter or cookie value as a way of authorizing the request. For our FarmStall API this will work just fine, as we need to describe a header with the name `Authorization`. When the security schema has a `type` of `apiKey`, then the following fields apply...

**Table 7.8 Fields of security scheme object when type = apiKey**

Field	Notes	Required
	type	apiKey
Yes		in
Can be either header, query or cookie.	Yes	
name	The name of the header/query/cookie.	Yes
	description	A short description of the security, can be in Markdown format.

**NOTE****What about a query/header/cookie parameters?**

Some of you may be thinking back to our query parameter. Some might even know that OpenAPI supports headers as well as cookie parameters and wondering... what's the difference between those and security types? Excellent question! While it's possible to add a header parameter underneath the `parameters` keyword in an operation (just like we did for `query parameters`) it doesn't lend itself well to the semantics of general authorization. Servers and clients are efficient at handling authorization as a special type of abstraction and telling API consumers that this isn't just a normal header parameter but the way in which APIs can allow authorization allows for servers and clients to treat them differently.

### 7.3.3 Adding the Authorization header security scheme

Time to start describing. Let's start by describing the security scheme...

#### Listing 7.3 Authorization security scheme

```
openapi: 3.0.0
# ...
components:
  securitySchemes:
    MyUserToken: ①
      type: apiKey ②
      in: header ③
      name: Authorization ④
```

- ① The name of our security scheme which will be referenced in other parts of the specification.
- ② The type of our security, `apiKey`.
- ③ Narrow down the type of `apiKey` security this is (ie: header, query or cookie).
- ④ The name of the header, query or cookie.

This is how we declare security types within OpenAPI. It doesn't mean we've described which operations require it, only that the security is declared. We've chosen the simplest type of

authorization which is what FarmStall API uses, an HTTP header named `Authorization`. The name of the security (the name that'll be referenced later) is any arbitrary string, we've chosen `MyUserToken`, although that is only for our benefit as it has nothing to do with the API itself — we could've called it `FooBar` and been just fine.

Having declared our security type we can now add it to `POST /reviews`

### 7.3.4 Adding the security requirements to `POST /reviews`

#### Listing 7.4 Added security requirement object to `POST /reviews`

```
openapi: 3.0.0
# ...
paths:
# ...
/reviews:
  post:
    # ...
    security: ①
      - MyUserToken: [] ②
```

- ① Declares that this operation has security requirements, its value is a list of security objects.
- ② Our `MyUserToken` security, the value is a list of scopes (empty and irrelevant for now).

The `security` declares which security schemes apply for this operation. The semantics of OpenAPI are a logical *OR*, as in, so long as one of the security is applied then it is considered acceptable.

The empty array value on the other side of `MyUserToken` is a list of scopes that apply within that security scheme. This is only relevant for OAuth2 security types (and not `apiKey`).

We're now "code complete" as it were, in that we've described the security requirements of our `MyUserToken` and added it to the operation we're interested in. It's time now to verify we did a good job; time to try it out!

### 7.3.5 Using the security feature of try-it-out

We're in a position to verify that we've correctly described the security needs of `POST /reviews`. Before we can do that we need to ensure we have a user and a `MyUserToken`.

If you haven't already (or if its been a while... the server may have been reset) go ahead and create a user and grab a user token by...

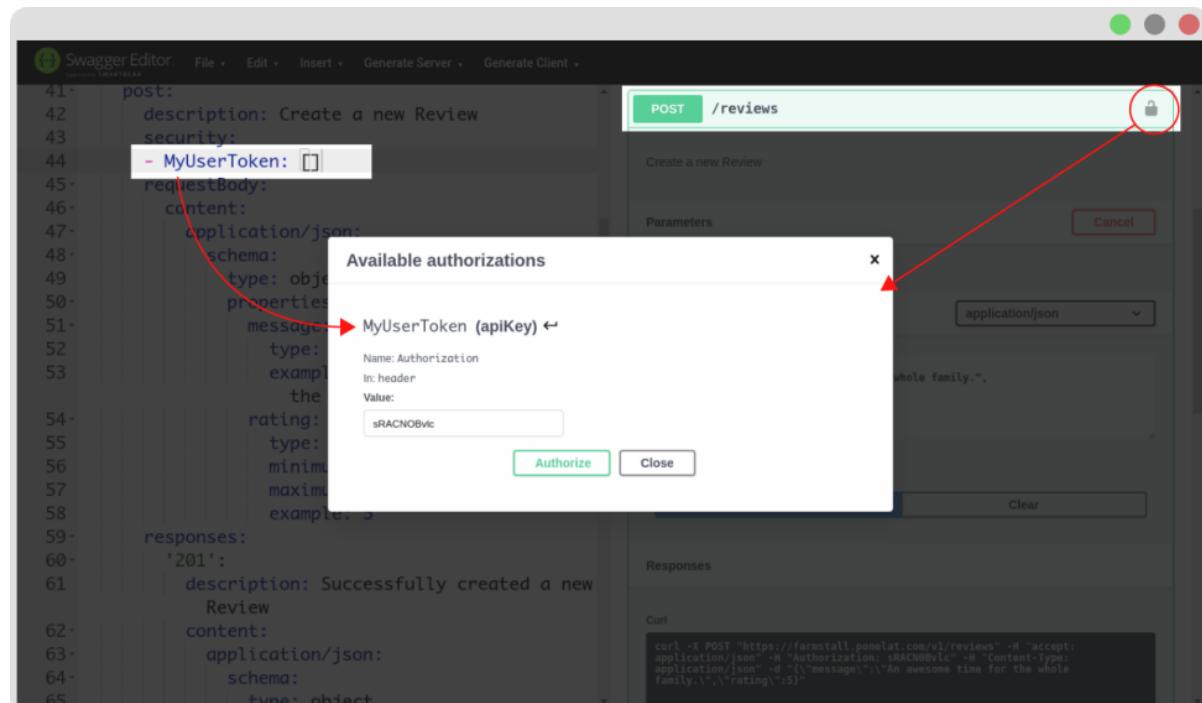
- Executing `POST /users`.
- Executing `POST /tokens` (with the username/password).

*Note: The exact steps are in the beginning sections of the this chapter, but I reckon you can wing it ;)*

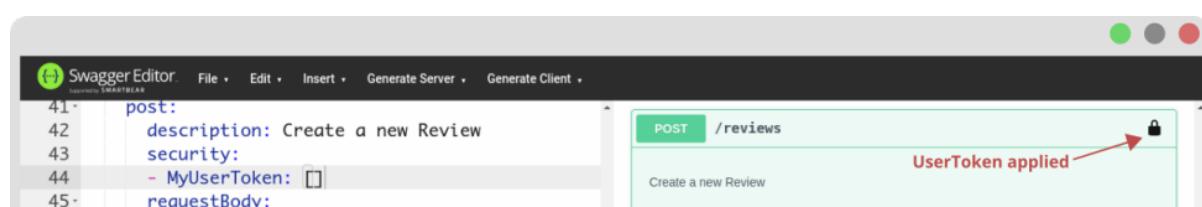
Note the MyUserToken when you grab one, as you'll need it for the next step.

We're going to...

- Add the security token.
- Execute POST /reviews.



**Figure 7.8 Adding a security value via try-it-out**



**Figure 7.9 Showing that an operation has a security applied to it**

```

41-   post:
42-     description: Create a new Review
43-     security:
44-       - MyUserToken: []
45-     requestBody:
46-       content:
47-         application/json:
48-           schema:
49-             type: object
50-             properties:
51-               message:
52-                 type: string
53-                 example: An awesome time for
54-                   the whole family.
55-               rating:
56-                 type: integer
57-                 minimum: 1
58-                 maximum: 5
59-                 example: 5
60-     responses:
61-       '201':
62-         description: Successfully created a new
63-           Review
64-         content:
65-           application/json:
66-             schema:
67-               type: object

```

Figure 7.10 Executing POST /reviews with security applied

Booya! If you managed to create a review and see that exciting `userId` in the response, then congratulations — you successfully added a security type!

If you're still stuck with something, go check out [app.swaggerhub.com/apis/designing-apis/part-one/ch07-end](https://app.swaggerhub.com/apis/designing-apis/part-one/ch07-end) which has a complete example. Compare it to your own, to see where a difference may have sneaked in.

Let's sum up the process.

## 7.4 How to add security schemes in general

In this chapter we described `POST /reviews` as having a security, which was a header named `Authorization`. We did so by first describing an `apiKey` security scheme named `MyUserToken` in the global `securitySchemes` component. We indicated it was located in the header with `in: header` and that header name was `Authorization`, with the field/value of `name: Authorization`.

The general pattern of adding authorization to operations is to first declare it under `securitySchemes` and then simply reference it from the operations that demand it, in the `security` list.

The value of the security requirement object (that empty array) only applies to OAuth scopes, so if you're using any other security type you can simply leave it as an empty array.

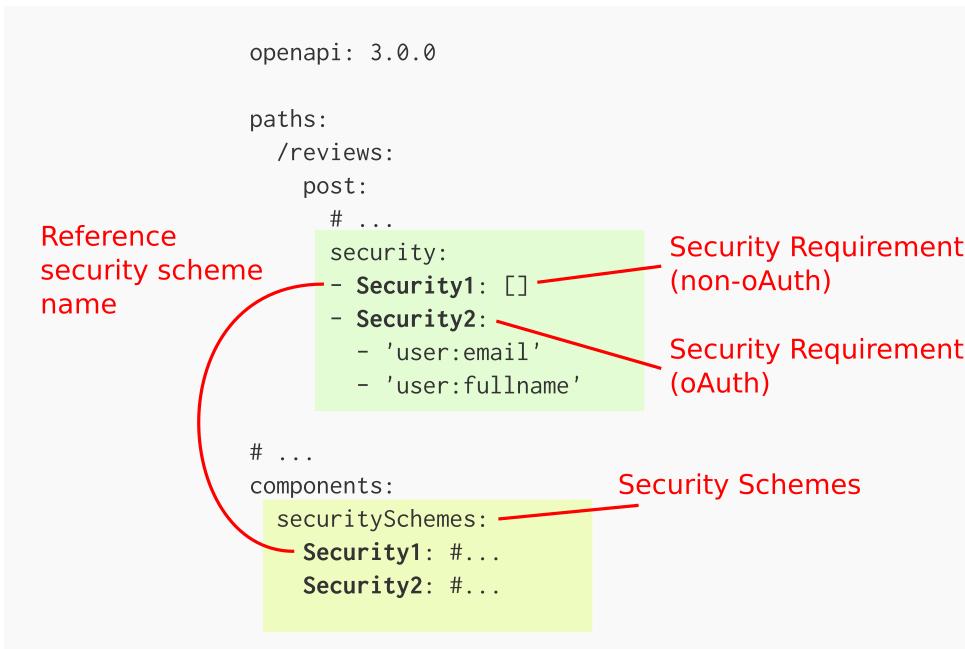


Figure 7.11 How to add security schemes in general

The details of the security schemes relate specifically to each of the different types. The most general one is `apiKey`, it can describe a primitive value of `query`, `header` or `cookie`.

The other types include, `oAuth`, `openIdConnect` and `http` (which is for Basic Auth, etc) can be tackled in a little more detail later on, or by looking at the specification directly, once you become more familiar with it!

## 7.5 Summary

- Describing POST operations is straightforward.
- `securitySchemes` which is written underneath `components` holds all the security declarations, where each is given an arbitrary name.
- `security` which is written under an operation, lists the securities that apply to the that operation. Each array item under `security` is an object with a single key. The key is the name of a security (as declared in `securitySchemes`) and the value is an empty array or an array of scopes (for `oAuth`).
- SwaggerEditor's Try-it-out feature allows you to *authorize* a request by filling in the value for a security.



# Preparing and hosting API documentation

## This chapter covers

- Adding metadata to our API definition
- Writing a description in Markdown
- Grouping operations together using tags
- Hosting our API documentation online using SwaggerUI and Netlify.com

In this chapter we're going to take our OpenAPI definition and turn it into online documentation. Before hosting it online we'll be adding some human touches such as the API metadata, rich text description and operation tags. These touches will make it a lot easier to consume by our users.

So far we've only been describing the bare essentials of the API, such as what operations exist and how to use them. This is the meat of the definition but we're lacking a softness that comes from explaining the API from one person to another. In addition we're also lacking some critical information that consumers require. Critical information such as the license of the API (are they allowed to consume it?) and contact information if they need to reach out. We'll be adding this metadata to the definition under the `info` section.

Metadata is the data *of the data*, which is such a fun term to use. In our context it's the data about the API definition, ie: all the necessary data that isn't related to the mechanics of the API.

To give our consumers the best possible introduction we're going to add a rich text description of our API using the awesome Markdown syntax that OpenAPI supports. This gives us a little freedom in how we showcase our API documentation without getting too deep into the weeds of a more formal website.

With our API definition ready to go we're then going to look at hosting it online for people to see.

There are many ways to turn an OpenAPI definition into API documentation and we're going to use one of the oldest, a tool called **SwaggerUI**. We've seen it already and some may have already noticed that. It is the UI part of SwaggerEditor, ie: the right-hand side panel. To be precise SwaggerEditor has an embedded version of SwaggerUI.

There is a myriad of ways to host static websites online. You could use an existing server if you have one, perhaps configure Apache/Nginx/etc as an HTTP server to host static files or do something silly like using `patchhub.pub` and a bash loop! We're not going to assume knowledge of any static file hosting, so I've chosen a suitable solution for our needs — [Netlify.com](https://netlify.com). Netlify.com is a static website hosting service. Which at the time of writing has a free account that is more than suitable for our needs of hosting a static website online. It's free (very important!) and is super quick to set up.

At the end of this chapter we'll have hosted our API documentation just like this...

The screenshot shows a custom SwaggerUI interface for the FarmStall API v1, hosted on <https://serene-volhard-dbc5fd.netlify.com>. The interface includes:

- Auth**: Instructions for creating reviews and getting a MyUserToken.
- Reviews**: Examples of reviews with star ratings and authors.
- Example Reviews**: A list of sample reviews.
- Servers**: A dropdown menu set to <https://farmstall.ponelat.com/v1>.
- Reviews**: A detailed view of the /reviews endpoint with methods: GET /reviews, POST /reviews, and GET /reviews/{reviewId}.
- Users**: A detailed view of the /users endpoint with methods: POST /users and POST /tokens.

Figure 8.1 A custom SwaggerUI hosted on Netlify

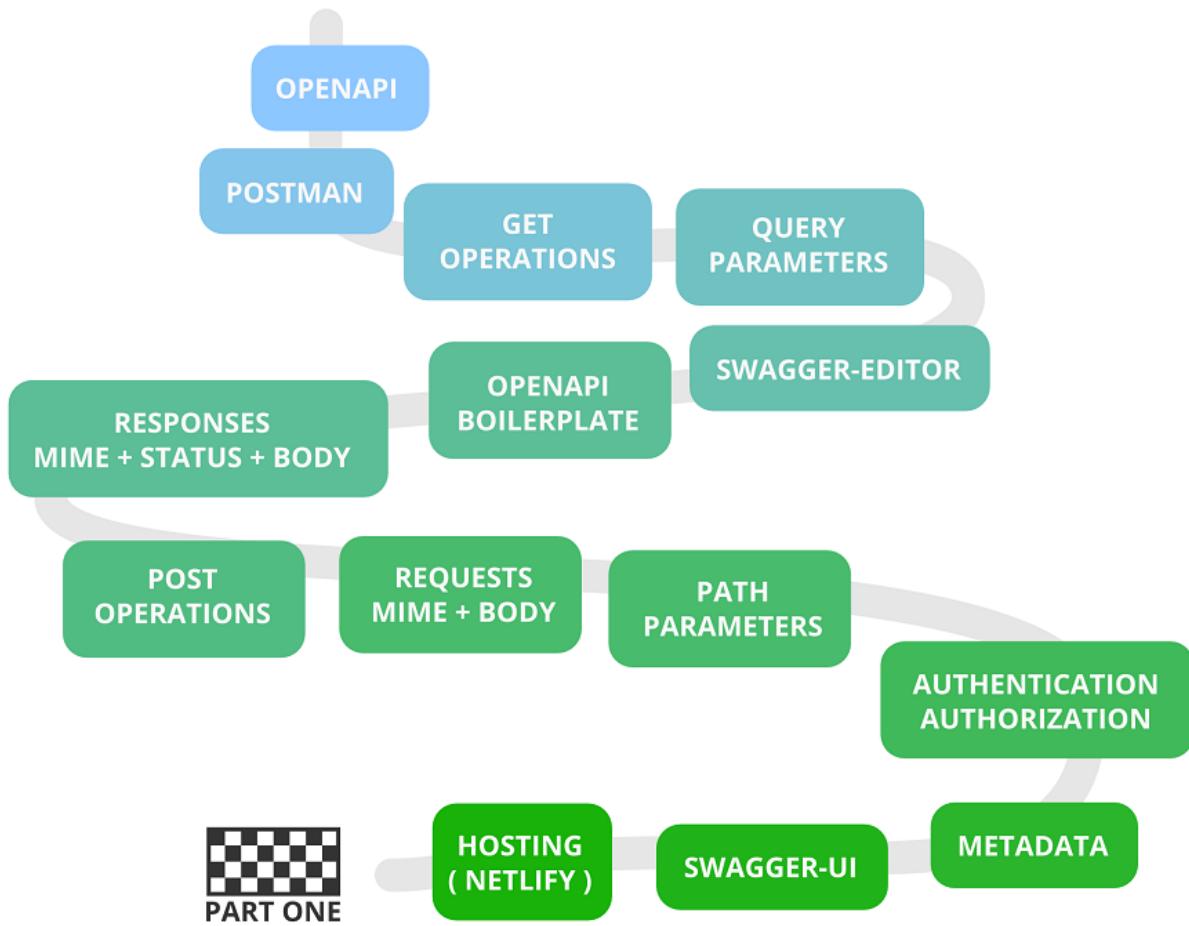


Figure 8.2 Where we are

## 8.1 The problem

We have several tasks to get through:

1. Add license info, contact info and a link to external docs
2. Add a rich text description of the API in Markdown
3. Host the API documentation online using SwaggerUI

The first two tasks involve adding to our API definition. The last task is going to be an operational one where we create an account, copy over some HTML and click a few buttons.

The metadata of our API, that we're going to add, will tell consumers some basic facts that they'll need. What information do we need to add to our API? Let's take a look...

**Table 8.1 Information to be added**

Information	Value	Notes
API Description	An API for writing reviews about your favourite ( or worst ) farm stalls.	<i>Description in markdown</i>
Contact Name	<your name>	Or John Doe if you like
Contact Email	<your email>	If you use a fake email be sure to use the @example.com domain, eg: <a href="mailto:fake@example.com">fake@example.com</a> . This domain was designed for this purpose. It avoids awkwardly sending emails to a real account. This is good practice for any/all dummy email addresses
Contact URL	<a href="http://farmstall.ponelat.com">farmstall.ponelat.com</a>	Usually a contact page on a website
License URL	<a href="http://apache.org/licenses/LICENSE-2.0">apache.org/licenses/LICENSE-2.0</a>	
License Name	Apache 2.0	It's a nice license
Link to External Docs	<a href="http://app.swaggerhub.com/apis/designing-apis/part-one">app.swaggerhub.com/apis/designing-apis/part-one</a>	A link to any documentation that isn't found within this definition. Can be anything relevant.
Description of External Docs	SwaggerHub hosted API definition	This tells the consumer what the externalDocs link points to

That will give the API consumers enough information to use the API, a place to reach out if there are issues, and the license info to see if and how they can use the API.

The description (written in Markdown) will look like this when it has been rendered by SwaggerUI...

An API for writing reviews about your favourite (or worst) farm stalls.



## Auth

To create **Reviews** without being anonymous. You need to add a **MyUserToken** to the **Authorization** header.

To get a **MyUserToken**:

1. Create a User with [POST /users](#)
2. Get a **MyUserToken** by calling [POST /tokens](#) with your User credentials.

## Reviews

Reviews are the heart of this API. Registered **Users** and anonymous users can both write reviews based on their experience at farm stalls.

Each review comes with a rating of between one and five stars inclusive.

- One star being the worst experience
- Five stars being the best

### Example Reviews

"A wonderful time!" — Bob McNally



"An awful place" — Anonymous



"A totally average place." — Jane Fair



Josh Ponelat - Website

Send email to [Josh.Ponelat@ponelat.com](mailto:Josh.Ponelat@ponelat.com)

Apache 2.0

SwaggerHub

## Figure 8.3 Markdown description

In addition to metadata we want to organize our API documentation so that it's a little easier to figure out what operations exist, at a glance. OpenAPI has the concept of tags and we're going to

use them to categorize our five operations into **Reviews** and **Users** to give a better overview of the API's operations.

**Table 8.2 Organize into Tags**

Operation	Tag
GET /reviews	Reviews
POST /reviews	Reviews
GET /reviews/{reviewId}	Reviews
POST /users	Users
POST /tokens	Users

We can also add descriptions to those tags so that we understand the categories' purpose better...

**Table 8.3 Tag descriptions**

Tag	Description
Reviews	Creating and getting reviews
Users	For creating and authenticating users

To sum up...

- Add metadata into the `info` object
- Add `description`, written in Markdown, into the `info` object
- Add tags operations and update their descriptions
- Host our API documentation online!

Go ahead and continue to extend the API definition you already have in SwaggerEditor. If it got corrupted or any other misfortune befall it, you can always grab a copy from:

[app.swaggerhub.com/apis/designing-apis/part-one/ch08-start](http://app.swaggerhub.com/apis/designing-apis/part-one/ch08-start)

## 8.2 Adding metadata information to the definition

Let's begin by writing in the metadata, the information we're going to add is...

**Table 8.4 Information to be added**

Field	Value
<code>info.description</code>	An API for writing reviews about your favourite (or worst) farm stalls.
<code>info.contact.name</code>	Josh Ponelat
<code>info.contact.email</code>	<a href="mailto:jponelat+daso@gmail.com">jponelat+daso@gmail.com</a>
<code>info.contact.url</code>	<a href="http://farmstall.ponelat.com/">farmstall.ponelat.com/</a>
<code>info.license.url</code>	<a href="http://apache.org/licenses/LICENSE-2.0">apache.org/licenses/LICENSE-2.0</a>
<code>info.license.name</code>	Apache 2.0
<code>externalDocs.url</code>	<a href="http://app.swaggerhub.com/apis/designing-apis/part-one">app.swaggerhub.com/apis/designing-apis/part-one</a>
<code>externalDocs.description</code>	SwaggerHub Hosted Docs

We'll be adding this to the `info` (and `externalDocs`) section of the definition. We have already created this section, as it was required for very basic metadata such as `title` and `version`. Without which it would be very difficult to identify the API at all! Extending it should be straightforward. We can smash those out pretty quickly. In later sections we're going to flesh out the `description` field to include a host of rich text elements, but for now, we'll just add a single line of text.

### Listing 8.1 Info section fleshed out

```
openapi: 3.0.0
info:
  version: v1
  title: FarmStall API
  description: |- ①
    An API for writing reviews about your favourite (or worst) farm stalls. ②
  contact:
    name: Josh Ponelat ③
    email: jponelat+daso@gmail.com ④
    url: https://ponelat.com/contact ⑤
  license:
    url: https://www.apache.org/licenses/LICENSE-2.0.html ⑥
    name: Apache 2.0 ⑦
  # ...end of the `info` section
  externalDocs: ⑧
    url: https://app.swaggerhub.com/apis/designing-apis/part-one ⑨
    description: SwaggerHub ⑩
  # ...rest of the definition...
```

- ① The `description` field will contain a lot of text later on. So we're making it a multi-line string. See [yaml-multiline.info](#) if you're hungry for those details again.
- ② Note that this text is indented (since `description` is as multi-line string).
- ③ Name of the person to reach out to for API related queries.
- ④ Email of the person to reach out to for API related queries.
- ⑤ URL of a website where consumers can get more contact details, possibly a contact form.
- ⑥ URL of licensing information (typically one of the well established ones) .
- ⑦ Name of the license.
- ⑧ The `externalDocs` section (same level as `info` section).
- ⑨ Link to external docs — using my copy of the definition in SwaggerHub.
- ⑩ Description of the external doc.

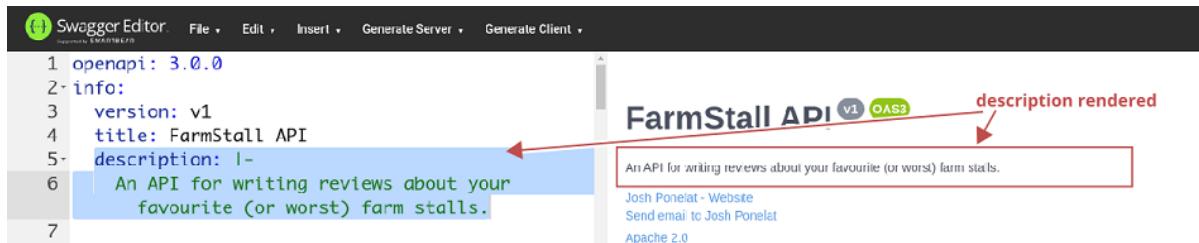
*Something small worth noting is that the contact and license information are housed under the fields `contact` and `license` respectively instead of being directly under the `info` section. For a list of popular open source licenses... [opensource.org/licenses](#)*

## 8.3 Writing the description in Markdown

In this section we're going to learn about Markdown and how to use it to create a rich text description of the FarmStall API. The rich text will go under the `info.description` field in the API definition.

Markdown is an amazing syntax that allows us to create rich text which can be rendered to HTML. It is far friendlier to write compared to HTML and is simple enough to use.

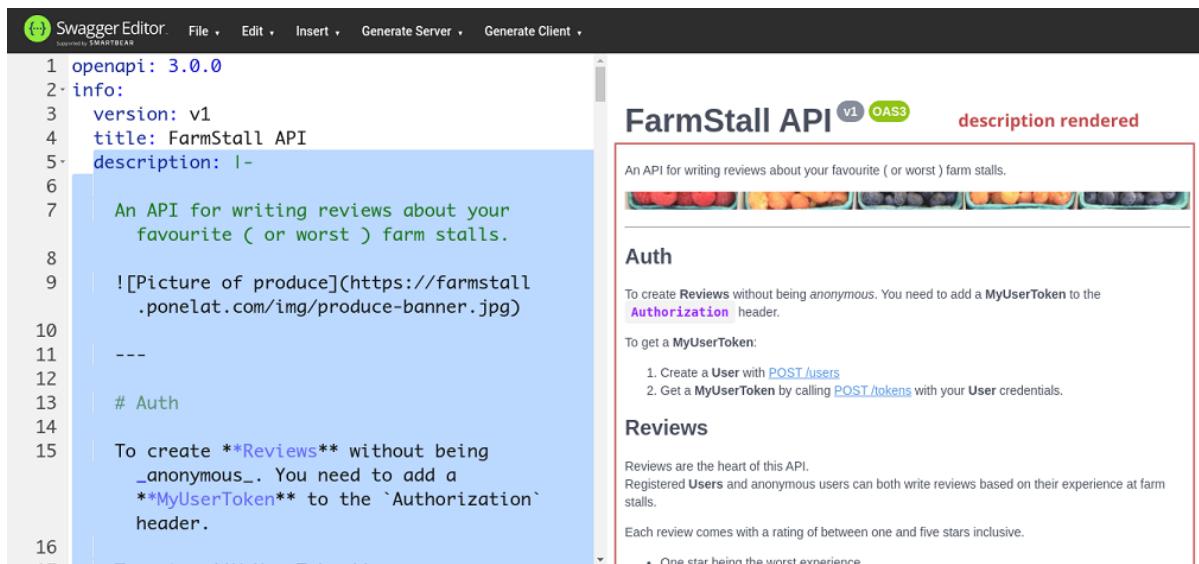
From the previous section where we added a placeholder, seen here...



The screenshot shows the Swagger Editor interface. On the left, the API definition is displayed in YAML format. Line 6 contains the placeholder `An API for writing reviews about your favourite (or worst) farm stalls.`. On the right, the rendered API description is shown. It includes the API title "FarmStall API v1", a green "OAS3" badge, and a "description rendered" label. Below the title is a box containing the text "An API for writing reviews about your favourite (or worst) farm stalls." with a red arrow pointing from the placeholder in the YAML to this box. At the bottom of the rendered description, there is contact information: "Josh Ponelat - Website", "Send email to Josh Ponelat", and "Apache 2.0".

Figure 8.4 Showing the placeholder description

We're going to flesh it out with Markdown and transform it into this...



The screenshot shows the Swagger Editor interface with the completed API definition. The placeholder text from Figure 8.4 has been replaced by a rich text description. Line 7 now contains `An API for writing reviews about your favourite ( or worst ) farm stalls.`. Line 9 contains `![Picture of produce](https://farmstall.ponelat.com/img/produce-banner.jpg)`. Line 15 contains `To create **Reviews** without being _anonymous_. You need to add a **MyUserToken** to the `Authorization` header.`. On the right, the rendered API description is shown with a red border. It includes the API title "FarmStall API v1 OAS3", a green "OAS3" badge, and a "description rendered" label. The rendered description box contains the text "An API for writing reviews about your favourite ( or worst ) farm stalls." and a small image of various fruits and vegetables. Below the rendered description, there are sections for "Auth" and "Reviews". The "Auth" section explains that reviews require a `MyUserToken` in the `Authorization` header. The "Reviews" section states that reviews are the heart of the API and that each review has a rating from one to five stars, with a note that one star is the worst experience.

Figure 8.5 Rich text description of FarmStall API

Rich text usually consists of italic, headers, lists, links, etc. Knowing how to write Markdown is becoming a *must* for developers from all walks of life! It is perhaps the most straight forward way to lift plain text into rich text, and is used in many different environments and supported by a host of platforms. You'll find it in blogging/cms platforms, wiki pages, chatting apps and most places that rich text is desired.

There are different flavours of markdown — such as the popular *GitHub Flavoured Markdown* (

*GFM* ) which is used in GitHub issues and in most *README* files. These flavours tend to offer platform (ie: GitHub) specific features and so an effort was made to create a standard Markdown specification that is more generic and open... those efforts culminated into CommonMark.

The version of Markdown that OpenAPI officially supports is **CommonMark**.

*For details on the CommonMark Spec (a little boring) [spec.commonmark.org/0.27/](https://spec.commonmark.org/0.27/) For a tutorial on CommonMark (way more fun) [commonmark.org/help/tutorial/](https://commonmark.org/help/tutorial/)*

We're going to cover the basics of Markdown (which are (mostly) applicable to all flavours of Markdown, but when in doubt the CommonMark specification is the official standard for OpenAPI). If you're already familiar with Markdown, feel free to skip the next section, although there might be some interesting snippets — so give it a glance before you do!

Things we'll be covering...

- How to format text into bold,italic,inline-code.
- Links and images.
- Lists.
- Code blocks.
- Headings and Horizontal Rules.

### 8.3.1 Markdown basics

Why use Markdown?

Back in the day if you wanted to write a blog post and have some of the text be bold you would add an HTML snippet such as `<b> Some Bold Text </b>`. This was good and well, but it soon became tedious and was error prone. From these efforts arose simple mark up languages (languages that add semantics to text) that had shortcuts for making text look more exciting, so instead of `<b> Bold </b>` you could write something like this: `**more bold text**`, which was much easier to remember and to write.

In addition to the simple markup shortcuts (ie: bold/italic), all sorts of shorthand notations began to appear and eventually you had full markup languages that covered a lot of the rich text needs of bloggers and website content writers.

The leading implementation that soon outshone the others was Markdown (a play on the words *mark up*). It's a simple syntax that gives you a lot of power in livening up plain text, but still simple enough to remember. It also has a trick up its sleeves... it allows you to nest HTML inside of it, so when you need something particularly fancy — you can defer to HTML.

Here is a simple cheat sheet that covers the Markdown basics. The best way to learn Markdown is to simply play with it. There are many online (and offline) editors that allow you to see what

the different mark ups look like when rendered, just as SwaggerEditor does for OpenAPI! In fact you can use SwaggerEditor to learn Markdown by simply typing markdown in any of the description tags.

Before you dive into playing with Markdown, cast your eyes across this cheatsheet for the basics...

Markdown is a simple syntax to turn plain text into rich text.

( Usually converted into HTML ).

**CommonMark is the standard used in OpenAPI 3.0.x**

<code>*italic*</code> or <code>_italic_</code>	<code>italic</code>
<code>**bold**</code> or <code>--bold--</code>	<code>bold</code>
<code>_Mix italic and **bold**_</code>	<code>Mix italic and bold</code>
<code>'inline code'</code>	<code>inline code</code>
> Blockquote	Blockquote
<code># Heading 1</code> <code>## Heading 2</code> <code>### Heading 3</code>	Heading 1 Heading 2 Heading 3
<code>[Link](https://example.com)</code>	<code>Link</code>
<code>![Image](https://farmstall.ponelat.com/img/rating-5.png)</code>	
<code>* list item</code> <code>* list item</code> <code>* list item</code>	<code>• list item</code> <code>• list item</code> <code>• list item</code>
<code>1. list item</code> <code>1. list item</code> <code>1. list item</code>	<code>1. list item</code> <code>2. list item</code> <code>3. list item</code>
<i>Horizontal Rule</i> ---	
---	
<code>...</code>	
<code>// Code block</code>	<code>// Code block</code>
<code>print "hello"</code>	<code>printf "hello"</code>

**Figure 8.6 Markdown description**

Let's make use of it and create a simple but rich description for our FarmStall API.

### 8.3.2 Adding a rich text description to FarmStall API definition

Let's break down the description area into sections so that we can tackle them one-by-one...

**FarmStall API** v1 OAS3  
[openapi.yaml](#)

**Header Section** An API for writing reviews about your favourite (or worst) farm stalls.  


**Auth Section** To create **Reviews** without being *anonymous*. You need to add a **MyUserToken** to the **Authorization** header.  
 To get a **MyUserToken**:  
 1. Create a **User** with [POST /users](#)  
 2. Get a **MyUserToken** by calling [POST /tokens](#) with your **User** credentials.

**Reviews Section** Reviews are the heart of this API.  
 Registered **Users** and anonymous users can both write reviews based on their experience at farm stalls.  
 Each review comes with a rating of between one and five stars inclusive.  
 • One star being the worst experience  
 • Five stars being the best

**Example Reviews Section** "A wonderful time!" — Bob McNally  
  
 "An awful place" — *Anonymous*  
  
 "A totally average place." — Jane Fair  


Josh Ponelat - Website  
[Send email to Josh Ponelat](#)  
 Apache 2.0  
[SwaggerHub](#)

**Figure 8.7 Sections of the final rich text description**

**Table 8.5 Sections of the rich text description**

Section	Notes
Header	A simple paragraph followed by a banner image and a horizontal rule.
Auth	A heading-1 with a paragraph having bold, italic and inline-code text. Followed by a numbered list, including links.
Reviews	A heading-1, paragraph using line-breaks and a bulleted list.
Example Reviews	Inline images, line breaks and HTML entities (an <i>em dash</i> ).

## HEADER SECTION

We open up with a simple paragraph that has no markup in it, followed by a banner image that we include to add a splash of color. The banner is inserted as an image, in Markdown, images have an Alt-Text (the text that displays if the image fails to load) and URL attributes. The URL pointing to the image can be relative or absolute. Finally to separate the header from the rest of the description we add a horizontal rule.

## Listing 8.2 Rich text - Header section

```
openapi: 3.0.0
info:
  #...
  description: |-
    An API for writing reviews about your favourite (or worst) farm stalls. ①
    ! [Picture of produce] (https://farmstall.ponelat.com/img/produce-banner.jpg) ②
    *** ③
# ...
```

- ① Opening paragraph that doesn't include any mark up.
- ② The banner image with an alt text of "Picture of produce". Note the beginning '!' that indicates an image markup.
- ③ A horizontal rule... which can be three or more dashes/asterisks.

## AUTH SECTION

Now for some (more) exciting markup (although I did enjoy the image from the last section)! In this section we have a *heading-1* which is the largest of the headings, we also have bold, italic and inline-code (the monospaced one). *AND* we have a numbered-list!

## Listing 8.3 Rich text - Auth section

```
openapi: 3.0.0
info:
  #...
  description: |-
    An API for writing reviews about your favourite (or worst) farm stalls.

    ! [Picture of produce] (https://farmstall.ponelat.com/img/produce-banner.jpg)

    *** ①

    # Auth ②

    ③ To create **Reviews** without being _anonymous_. You need to add a **MyUserToken** to the `Authorization` header.

    To get a **MyUserToken**:
    1. Create a **User** with [POST /users] (#Users/post_users) ④
    1. Get a **MyUserToken** by calling [POST /tokens] (#Users/post_tokens) with your **User** credential
    ⑤

# ...
```

- ① ...The section we already wrote up, we'll begin to leave it out for brevity.
- ② The Auth header, which is a heading-1 element (largest of the headings).
- ③ A paragraph containing bold ("Reviews"), italic ("anonymous") and inline-code ("Authorization").
- ④ A numbered list item (note that the number itself doesn't matter, so I always use 1.). A link to an internal local anchor (ie: #Users/post\_users).

- ⑤ Another numbered list item including bold text and a link. Again the number itself doesn't matter.

### 8.3.3 Reviews Section

This section introduces two new elements, the bullet list (without numbers) and an HTML element for line-breaks. The line-breaks can be done in Markdown (by using two newlines) but I wanted to show how you can use raw HTML.

#### Listing 8.4 Rich text - Reviews section

```
openapi: 3.0.0
info:
  ...
  description: |-  

    ... ①  

    # Reviews ②  

    Reviews are the heart of this API. <br/> ③  

    Registered **Users** and anonymous users can both write reviews based on their experience at farm st  

    Each review comes with a rating of between one and five stars inclusive.  

    - One star being the worst experience ④  

    - Five stars being the best ⑤
```

- ① Previous rich text left out for brevity.
- ② Reviews heading (heading-1).
- ③ The end of this line includes raw HTML, the `<br/>` element (for line-breaks).
- ④ A bullet list item.
- ⑤ Another bullet list item.

### 8.3.4 Example Reviews Section

To round off our rich text section we're going to display a few **Review** examples that look pretty. They'll include an inline image and an HTML entity (ie: a special character).

## Listing 8.5 Rich text - Examples section

```

openapi: 3.0.0
info:
  #...
  description: |-
    ...
    ①

    ### Example Reviews ②

    "A wonderful time!" &mdash; Bob McNally ③
    <br/> ④
    ! [5 stars] (https://farmstall.ponelat.com/img/rating-5.png) ⑤

    "An awful place" &mdash; _Anonymous_ ⑥
    <br/>
    ! [1 star] (https://farmstall.ponelat.com/img/rating-1.png)

    "A totally average place." &mdash; Jane Fair ⑦
    <br/>
    ! [3 stars] (https://farmstall.ponelat.com/img/rating-3.png)
  
```

- ① The description we've written so far has been left out for brevity.
- ② A heading-3 (smaller than heading-1 and heading-2).
- ③ A paragraph with a HTML-entity inside &mdash; which is a long dash, ie: "—".
- ④ Line break (but not a new paragraph).
- ⑤ An image with alternative-text (the alt-text is "5 stars").
- ⑥ Another review example (same structure as first one).
- ⑦ And another example.

## 8.4 Organizing operations with tags

In order to help organize operations within an API definition, OpenAPI has support for a feature called Tags. One or more tags can be added to operations to better categorize and group different operations together. In the FarmStall API we have described five operations. In this section we're going to add a tag to each operation grouping them together into *Reviews* operations and *Users* operations.

In SwaggerUI these tags will show up as sections with operations grouped underneath each section. Using tags is a great way to organize related operations together.

*Note with SwaggerUI and Tags — An operation can appear under multiple tag headings.*

Here are the operations and the tags we'll be adding to each one...

**Table 8.6 Organize into Tags**

Operation	Tag	Note
GET /reviews	Reviews	Getting reviews
POST /reviews	Reviews	Creating reviews
GET /reviews/{reviewId}	Reviews	Getting a specific review
POST /users	Users	Creating a new user
POST /tokens	Users	Authenticating a user by giving them a token

Once we add the tags and the tag descriptions we should see something like this show up in SwaggerEditor...

58  
59 **tags:**  
60   - **name:** Reviews  
61    **description:** Reviews of your favourite/worst  
62    farm stalls  
63   - **name:** Users  
64    **description:** Users and authentication  
65  
66 **servers:**  
67   - **url:** 'https://farmstall.ponelat.com/v1'  
68  
69 **paths:**  
70   **/reviews:** **tag(s) added to operation**  
71     **get:**  
72       **tags:**  
73        - **Reviews**  
74        **description:** Get a list of reviews  
75        **parameters:**  
76        - **name:** maxRating  
77        **in:** query  
78        **schema:**  
79        | **type:** number  
80        **responses:**  
81        | '200':  
82        |    **description:** A bunch of reviews  
83        |    **content:**

**Figure 8.8 Showing how tags are rendered in SwaggerEditor**

#### 8.4.1 Adding the Reviews tag to GET /reviews

Let's begin by adding the Reviews tag to the GET /reviews operation...

##### **Listing 8.6 Added the Reviews tag to GET /reviews**

```
openapi: 3.0.0
# ...
paths:
  /reviews:
    get:
      tags: ①
      - Reviews ②
      #...
```

- ① Inside the operation add the `tags` field.

- ② Add an array item and inside it, the name of the tag — ie: `Reviews`.

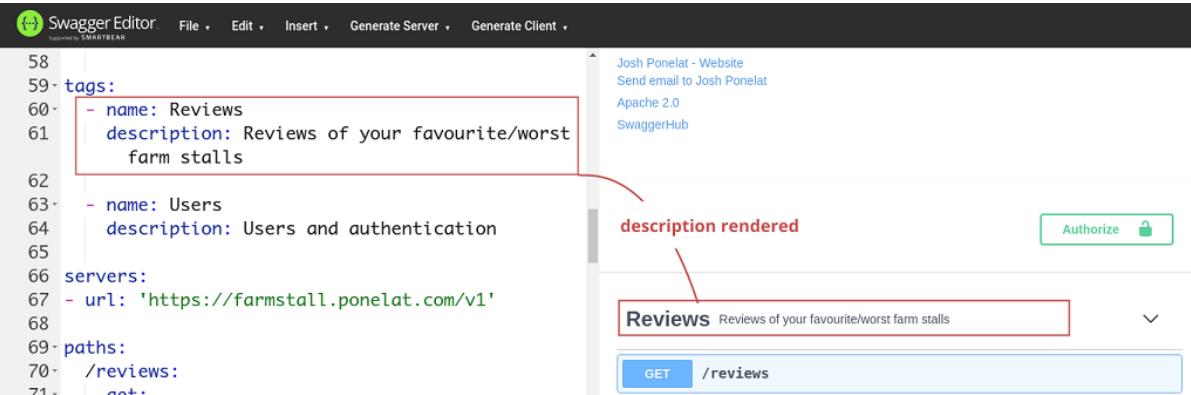
Done! That wasn't too hard. Without adding anything else this tag is created "on the fly". Go ahead and add it in SwaggerEditor.

Here is a copy of the definition so far:  
[app.swaggerhub.com/apis/designing-apis/part-one/ch08-tags-start](http://app.swaggerhub.com/apis/designing-apis/part-one/ch08-tags-start). Copy the YAML you into [editor.swagger.io](http://editor.swagger.io) and you can begin hacking it!

### 8.4.2 Adding descriptions to tags

What happened in the last section is that we added a tag to an operation. Those tags will be created on the fly and tools (such as SwaggerEditor/SwaggerUI) can render UIs based on them. But we are missing a description for the tag, something to expand on what the tag entails. The word `Reviews` may not be sufficient to tell the consumer what's going on.

After adding a description to the tags we'll see the following rendered in SwaggerEditor...



The screenshot shows the Swagger Editor interface. On the left, the YAML code for the API definition is displayed. A red box highlights the `tags` section, which contains an array of tag definitions. The first tag, `Reviews`, has a detailed description: `Reviews of your favourite/worst farm stalls`. A red box on the right highlights this description. A callout arrow points from this box to the rendered description in the UI on the right. The UI shows a card for the `Reviews` tag with the same description. Below the card is a list of operations, with the first one being a `GET /reviews` operation.

```

58
59  tags:
60    - name: Reviews
61      description: Reviews of your favourite/worst
62      farm stalls
63
64    - name: Users
65      description: Users and authentication
66
67  servers:
68    - url: 'https://farmstall.ponelat.com/v1'
69
70  paths:
71    /reviews:
72      get:

```

Figure 8.9 Showing the description of the tag

To add a description to a tag we need to create a root-level `tags` field and describe our tags in it.

#### Listing 8.7 Adding a root-level tags object with a description for the Reviews tag

```

openapi: 3.0.0
# ...
tags: ①
  - name: Reviews ②
    description: Reviews of your favourite/worst farm stalls ③
# ...

```

- ① The `tags` root-level field for describing tags. It is an array of objects each having at least a `name` field, and optionally a `description` and/or `externalDocs` field.
- ② The name of our tag, `Reviews`. This is case sensitive and operations that want to add this tag should match it exactly.
- ③ The description of our tag!

### 8.4.3 Adding the rest of the tags

Now that we've successfully added a tag to the `GET /reviews` operation *AND* we added a description for the `Reviews` tag. We can go ahead and add tags to the remaining operations within the definition. As well as the description for the `Users` tag.

Go ahead and add the following tags to the following operations...

**Table 8.7 Organize into Tags**

Operation	Tag	Added?
<code>GET /reviews</code>	<code>Reviews</code>	Yes
<code>POST /reviews</code>	<code>Reviews</code>	No
<code>GET /reviews/{reviewId}</code>	<code>Reviews</code>	No
<code>POST /users</code>	<code>Users</code>	No
<code>POST /tokens</code>	<code>Users</code>	No

And add the following description to the `Users` tag...

**Table 8.8 Tag descriptions**

Tag	Description
<code>Reviews</code>	Reviews of your favourite/worst farm stalls
<code>Users</code>	Users and authentication

At the end of this exercise, SwaggerEditor should render it into the following...

The screenshot shows the SwaggerEditor interface with two main sections: `Reviews` and `Users`.

- Reviews Section:**
  - `GET /reviews` (blue button)
  - `POST /reviews` (green button, locked)
  - `GET /reviews/{reviewId}` (blue button)
- Users Section:**
  - `POST /users` (green button)
  - `POST /tokens` (green button)

Annotations on the left side of the interface indicate the tags assigned to each operation:

- A red line labeled "tags: - Reviews" points to the `POST /reviews` operation in the `Reviews` section.
- A red line labeled "tags: - User" points to the `POST /users` operation in the `Users` section.

**Figure 8.10 SwaggerEditor with tags and tag descriptions**

## 8.5 Hosting our API documentation using Netlify.com and SwaggerUI

API documentation is only as good as it is reachable. If Bob the developer cannot reach the API documentation then he won't know how to use it!

In this section we're going to host a SwaggerUI webpage using our OpenAPI definition, and we're going to make it publicly reachable — So that we can show it off and get feedback on it!

### SIDE BAR    **Swagger-what ?**

So far we've dealt with one of the Swagger tools, namely SwaggerEditor. What we may not have realized is that inside SwaggerEditor there is another tool... SwaggerUI!

It forms the right-hand panel of SwaggerEditor and can be used on its own to render HTML documentation, based on an OpenAPI definition.

Historically SwaggerUI was found embedded in API Servers, so that there would be a console for developers to play with the API by making API calls. The embedded SwaggerUI would use relative URLs to talk to the Server that hosted it.

Nowadays you'll find SwaggerUI in all sorts of places as it can be hosted on its own. The project is a collection of HTML/JS/CSS files that dynamically build up a webpage, rendering the OpenAPI definition.

After we have a working SwaggerUI instance we're going to share it online using a static file server. We've chosen one that suits our needs (it's free, it looks cool and is quick to set up). For hosting we're going to use Netlify.com.

### 8.5.1 Preparing SwaggerUI with our definition

Right! Let's get our own version of SwaggerUI set up!

What we'll need is...

- An HTML page for rendering SwaggerUI
- An OpenAPI definition (we'll be using the FarmStall API definition, of course).

## Listing 8.8 SwaggerUI boilerplate

```

<!DOCTYPE html> ①
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" type="text/css" href="//unpkg.com/swagger-ui-dist@3/swagger-ui.css"> ②

  <title>FarmStall API v1</title>

<body>

  <div id="farmstall-docs" /> ③

  <script src="//unpkg.com/swagger-ui-dist@3/swagger-ui-bundle.js"></script> ④
  <script>

    window.onload = function() { ⑤

      const ui = SwaggerUIBundle({ ⑥
        url: "openapi.yaml", ⑦
        dom_id: "#farmstall-docs", ⑧
        deepLinking: true, ⑨
      })
    }

  </script>

</body>
</html>

```

- ① Some general HTML boilerplate.
- ② We link to the SwaggerUI CSS (using Unpkg.com).
- ③ The DOM element that SwaggerUI will inject HTML into.
- ④ The SwaggerUI javascript bundle/file (also Unpkg.com).
- ⑤ An onload hook (ie: called when the browser has finished loading).
- ⑥ Initialize a SwaggerUI instance.
- ⑦ URL to the OpenAPI definition (can be relative).
- ⑧ The DOM element to inject HTML into.
- ⑨ An extra feature to enable links to individual operations (used in the **Auth Section** rich text).

Whoa... *slightly scary* but not that scary. We've just created all that we need to host a SwaggerUI instance.

The only thing missing from this is the OpenAPI definition we've been working on.

Go ahead and create a folder where you're store the `index.html` and `openapi.yaml` files.

The `openapi.yaml` is the definition we've been working on throughout this chapter.

You can grab a copy of the HTML from here [github.com/ponelat/farmstall/tree/master/part01/farmstall-api-docs](https://github.com/ponelat/farmstall/tree/master/part01/farmstall-api-docs) (you'll also find a copy of the openapi.yaml in there too, if you need it).

*Be sure to spice up the rich text a little bit to make it your own!*

If you have a static file server lying around you can demo it yourself to see if something is broke, else let's get busy hosting this onto the Internet!

**NOTE**

Something I find I often need is a way to quickly host some files from my laptop or a remote server. Installing a file server is *okay* but sometimes I just want to test something quickly.

I've found that using python (which comes installed by default in most 'nix systems, like macOS and Linux) is quick and simple.

If its python 2: `python -m 'SimpleHTTPServer' 8080`

If its python 3: `python3 -m 'http.server' 8080`

Both will do the same thing and expose the directory you ran it from on port 8080. Its a nifty trick!

### 8.5.2 Hosting on Netlify.com

In this section we're going to host our folder on a publicly accessible URL.

We'll need to

- Create a Netlify.com account.
- Upload our folder.
- Optional: Drink a cup of coffee to take a break.

Given that we have a folder with `index.html` and `openapi.yaml` we're ready to go!

*Now there are a million different ways to host static files on the Internet but I choose Netlify.com only because it is simple to set up and looks like it may survive for a while longer. It is also darn useful to know a service that is easy to host static websites on, knowing a method to serve static files (personally I make use of GitHub pages) has proven very useful for me.*

**IMPORTANT** The Netlify.com steps and access may change at any time after this book has been published. While care was taken to choose a service that would last and remain similar, it's also true that the Internet changes faster than a speeding bullet. So mileage may vary on the Netlify.com steps.

Alternatives to Netlify.com include:

- Hosting on a S3 bucket.
- Using GitHub Pages (part of the paid plan at time of writing).
- GitLab Pages.
- Dropbox.
- Google Drive.

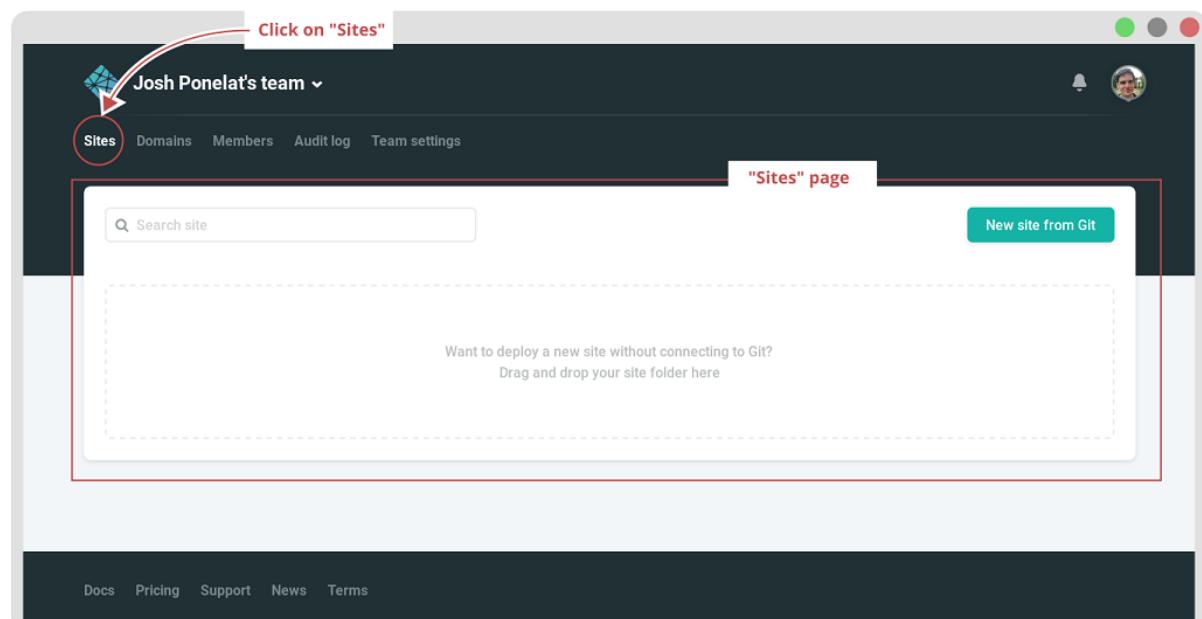
Netlify.com has a free plan that allows us to host a static folder and to even connect a domain name to it (the domain connecting is beyond the scope of this chapter, but here is how [www.netlify.com/docs/custom-domains/](https://www.netlify.com/docs/custom-domains/))

First things first we need a (free) account with Netlify.com in order to create a static website. Go ahead and create a new account with Netlify.com.

*It accepts signing up with GitHub which I personally use.*

Link to site... [app.netlify.com/](https://app.netlify.com/)

After signing up you can click on the "Sites" link in the navigation bar. From there we'll be able to upload a folder which will be our static site.

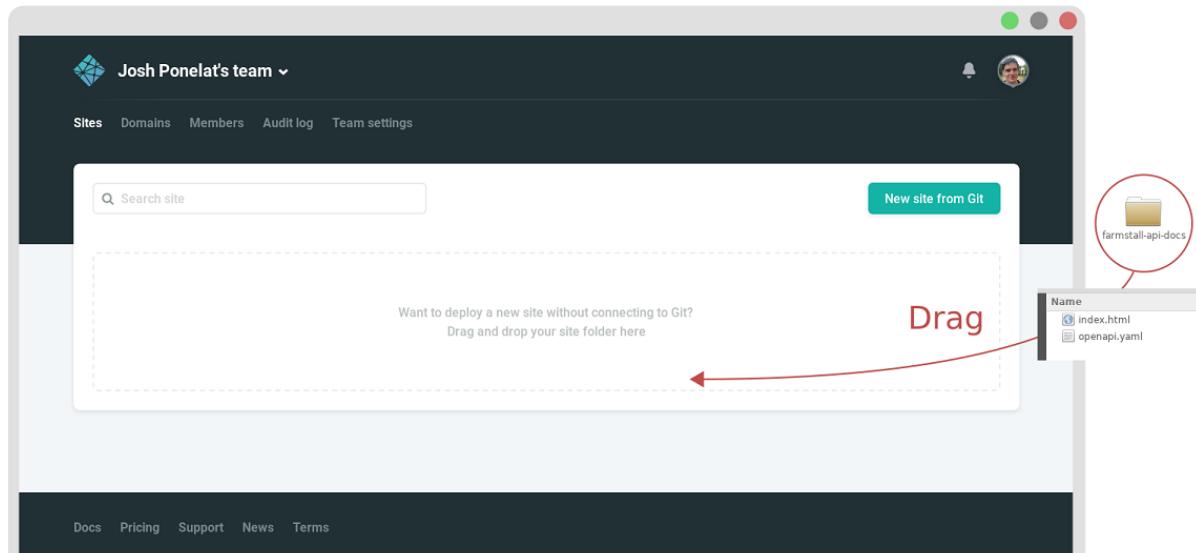


**Figure 8.11** Netlify.com with Sites open

Next we need to upload our folder that contains: - index.html (with SwaggerUI boilerplate). - openapi.yaml (our API definition of FarmStall v1)

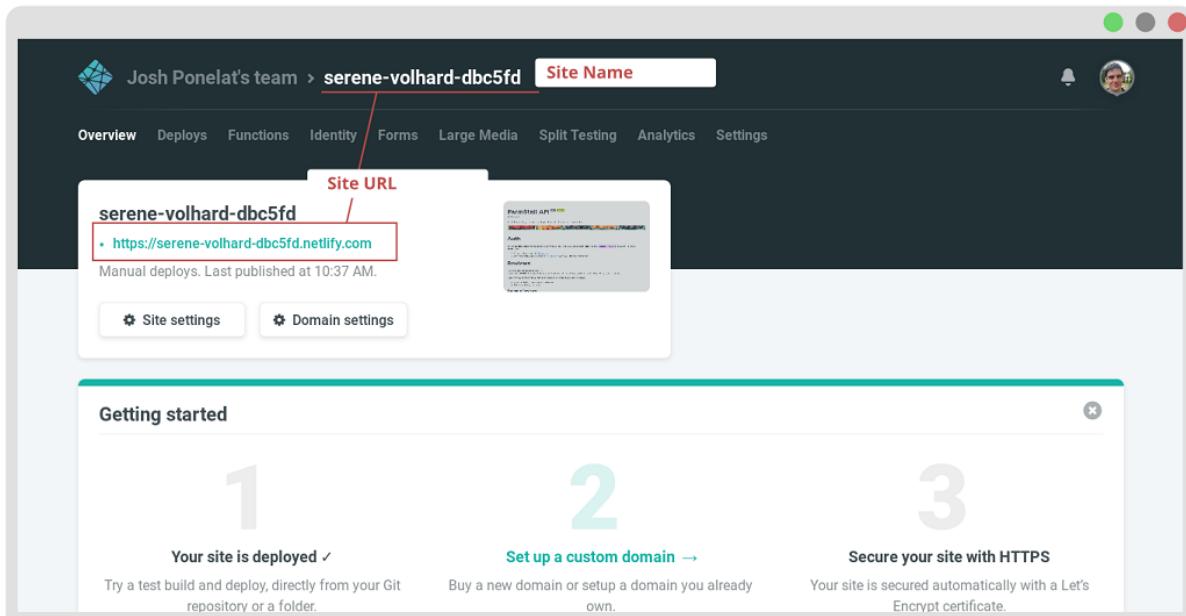
*Again, you can grab copies of both over here: [github.com/ponelat/farmstall/tree/master/part01/farmstall-api-docs](https://github.com/ponelat/farmstall/tree/master/part01/farmstall-api-docs)*

Time to upload our site...



**Figure 8.12 Netlify.com upload folder**

After a few seconds Netlify.com will give us a randomly generated site name, and our site will be hosted on a URL based on that site name.



**Figure 8.13 Netlify.com site successfully uploaded**

All that remains is to visit our site, see if it looks good, and then share it amongst our eager consumers (well friends and colleagues for now!).

The screenshot shows the FarmStall API v1 documentation as it appears in a browser. The URL is <https://serene-volhard-dbc5fd.netlify.com>. The page has a header with the API name and version, and a banner featuring various fruits. Below the banner, there are sections for **Auth** and **Reviews**. The **Reviews** section contains sample reviews and their ratings. At the bottom, there is a detailed API description for the **/reviews** endpoint, showing methods like **GET /reviews**, **POST /reviews**, and **GET /reviews/{reviewId}**. There are also sections for **Users** and **Auth** with their respective endpoints.

**Figure 8.14** SwaggerUI Hosted

## 8.6 Notes on Part One

That concludes part one of our adventure into OpenAPI and Swagger.

In the previous chapters we learned how to describe an existing API using OpenAPI and

SwaggerEditor. We looked at the basic building blocks of an API definition and finally ended up hosting our own SwaggerUI instance online, including some rich text!

### 8.6.1 How can you apply this?

Describing an API gives you the leverage to use tooling to help you manage that API. In this part of the book we described some of the basics of a contrived API, but one that includes a lot of the common patterns found in RESTful design. From the more simple CRUD-like methods to authentication and authorization.

The very first thing we're able to do with a freshly described API is to host API documentation (using SwaggerUI or other tools), which in itself is useful enough, but that is only the beginning.

### 8.6.2 Next part

In the next part of the book, we'll be looking at the design phase of APIs and how we can incorporate OpenAPI into that critical stage. Designing an API from scratch and more importantly incrementally adding/changing it as one would in a real world environment.

Exciting times ahead!

## 8.7 Summary

- APIs require good metadata to be really useful to consumers. Add details such as contact info (under `info.contact`), licensing info (under `info.license`) and if necessary a link to any auxiliary information that might exist (under `externalDocs` on the root level).
- Markdown can be used in the `description` fields. OpenAPI 3.x officially supports CommonMark as the syntax. Markdown is used to add life to text with common semantics such as headings, bold, italic, inline-code, numbered lists, unordered lists, block-quotes, code-blocks, links, images and horizontal rules. And when those won't suffice, you can defer to using raw HTML as well which some (not all) tools support.
- Tags can be used to organize/group operations together. Adding a tag to an operation can be done by adding a string to the `paths.{path}.{method}.tags` array. Tags can also be associated with descriptions. To add a description to a tag, it will need to be added to the root level `tags` field, under `tags[0].name` and `tags[0].description`.

# Designing a web application



## This chapter covers

- Goals, scenario and plan for the second part of this book
- Creating a domain model for the scenario
- Adding functionality to the domain model with user stories

In the first part of the book, we went through the basics of APIs, how to use them, and how to formally describe them with OpenAPI. We also worked with Swagger Editor to document an existing API - the FarmStall API - using OpenAPI. Now, in the second part of this book, we'll design a new API for a web application from scratch.

Going through an API design process and further through the API lifecycle is not just about OpenAPI and tools. It's also about people and processes. There are always new requirements and unforeseen circumstances that require handling. We tried making this second part of the book as close to reality as possible. That's why we go through the process with a fictional company.

It starts with a founder envisioning an idea and assembling a team. Together they create a plan to realize the project that starts with a domain model and user stories, continues with API design and ends with software implementation and integration. We will explain these methods as we move along the process.

## 9.1 The PetSitter idea

Meet José. He is the owner of a small web development shop. Even though he's earning good money designing custom websites and web applications, he thinks a lot about developing his business and starting his own product. And he's already got a business idea.

José and his wife are dog lovers. However, both are working full-time and don't want to leave their dog alone. Sometimes José brings it to the office, but that's not always an option either.

And even then, someone needs to take it for a walk when José is as busy as company owners tend to get. Finding someone to look after their dog is a chore that José and his wife would like to simplify. Why isn't there an app for that?!

After mulling over the idea for a bit, José takes out a notebook and scribbles down the initial set of requirements for the functionality of the app:

- Sign up: As dog owner or dog walker
- Dog owners: can post jobs
- Dog walkers: can apply to posted jobs

Although José only thought about dogs, he decides to use the more generic "PetSitter" (instead of "DogSitter") as the working title for his project. With that title and a list of functional requirements, José feels he is ready to get started.

## 9.2 PetSitter project kickoff

To get from a business or project idea to a working product requires execution. José has a business background and is not a developer himself, thus he needs to build a team to implement an application. Luckily he can draw from the pool of his employees. Assembling a team, however, is not enough. Every project also needs a plan so that every member of the team knows what they need to do. Let's join José in building his team and outlining their plan.

### 9.2.1 Additional requirements

While thinking about the resources he has at his disposal to actualize his plan, José adds the following notes:

- Build web app with in-house team - two people
- Mobile app - work with other development agency (later!)
- Chance to experiment with new technology
- Release first working prototype as soon as possible

Unlike the functional requirements he wrote down before that directly relate to the application's functionality, these are non-functional requirements. That is an umbrella term that covers various attributes of the product itself as well as constraints around the development process.

As we proceed, we will always check back whether our plan matches the requirements.

**Table 9.1 Requirements Checklist**

Type	Requirement	In plan?
Functional	Sign up: As dog owner or dog walker	
Functional	Dog owners: can post jobs	
Functional	Dog walkers: can apply to posted jobs	
Non-functional	Build web app with in-house team - two people	
Non-functional	Mobile app - work with other development agency (later!)	
Non-functional	Chance to experiment with new technology	
Non-functional	Release first working prototype as soon as possible	

## 9.2.2 Team structure

José goes through the list of his employees and looks at their skills and the kinds of projects that they're involved with at the moments. Then, he schedules a meeting with two of them, Nidhi and Max. Both developers have worked with José for a while and have shown their aptitude for learning and solving problems in unique ways.

As both agree to join the project, we have a three person team now. Being the initiator, José acts as the project lead. The roles for the developers are not defined yet. We will do that next.

In their first meeting, José presents the plan based on his notes about the functional as well as the non-functional requirements. Nidhi tells him that, if they want to expand into the mobile realm later, they should work on a clear separation between backend and frontend. "That way", she says, "we can have a backend with an API that different clients, such as our web application and then later the mobile application, can use!" "Great", says Max enthusiastically, "then we can build an SPA, a single-page-application. I've experimented with React lately, and I think we can use that here!"

The three of them keep talking, and everyone seems hooked on the project. "However, José", Nidhi adds, "remember that I have a few clients to support. We should try and work independently and asynchronously as much as possible. We can't always meet to sync up." Max nods in agreement, "Same here."

In this discussion, the developers suggest an architecture in which backend and frontend are two separately developed components, and Max already has a technology suggestion for the frontend. Before we move on, let's first agree on what frontend and backend means in the context of a web application:

- The frontend is everything that happens on the user's computer in their browser. The frontend is made with HTML, using JavaScript for interactivity. Max talked about React, which is a JavaScript framework for creating web application frontends.
- The backend is whatever happens on the server of the web application's provider. Backends can use many different programming languages and frameworks and typically use a database to persist data.

The setup and the developers' interests and availability naturally lead to a team structure with one frontend developer and one backend developer:

- Nidhi implements the backend
- Max implements the frontend

Josés first non-functional requirement about being able to build the application with two developers is met. We look more closely at the technology and the process for building each part later in this book.

### 9.2.3 API-driven architecture

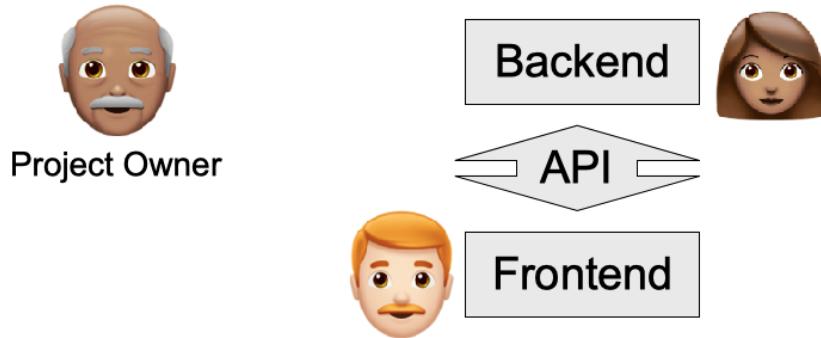
Josés eyes light up after hearing the word API out of Nidhis mouth. His team has integrated a few APIs into client projects, for example, for sending SMS notifications or integrating e-commerce data or for marketing automation. To date, however, they have not built their own. "We could release this API later so that people can build stuff", he suggests, "maybe some smart device that lets my pet sitter in automatically? Or something for my voice assistant?"

With everything that's possible with an API-driven architecture, his second requirement about being able to work with an external agency for building a mobile app is easily fulfilled. And so is the third requirement of experimenting with new technology, as it's the first time for Josés company to build an API of their own.

**Table 9.2 Requirements Checklist**

Type	Requirement	In plan?
Functional	Sign up: As dog owner or dog walker	
Functional	Dog owners: can post jobs	
Functional	Dog walkers: can apply to posted jobs	
Non-functional	Build web app with in-house team - two people	OK
Non-functional	Mobile app - work with other development agency (later!)	OK
Non-functional	Chance to experiment with new technology	OK
Non-functional	Release first working prototype as soon as possible	

Traditional web applications run backend code that dynamically generates the HTML for the frontend. In an API-driven architecture, however, the frontend generates HTML with client-side JavaScript code based on the API responses from the backend, which typically are in JSON format. That way, the backend is disconnected from the presentation logic on the client.



**Figure 9.1 Architecture and Development Team**

#### 9.2.4 The plan

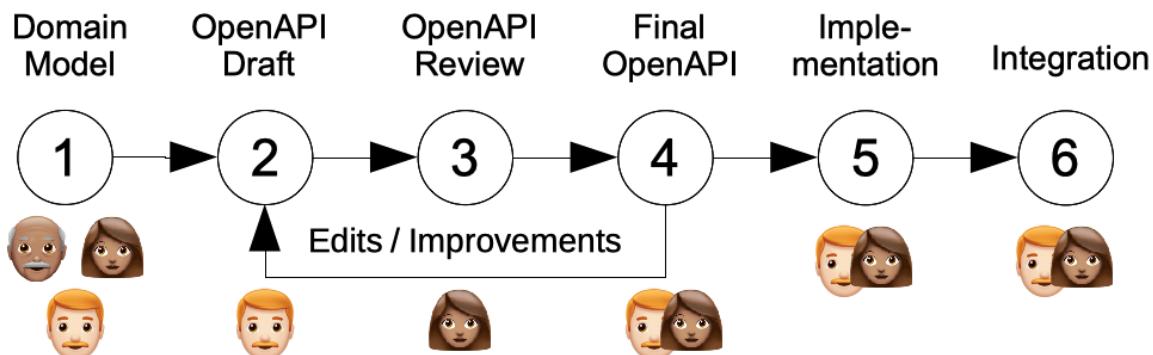
In our setup with its API-driven architecture, the backend and the frontend developer can work autonomously, but it requires them to agree on the API beforehand. As you know from reading this book, you can use OpenAPI to create a formal description of an API. And, as you will learn as we proceed, there are tools that help in the process of building an application based on said OpenAPI description.

We have a team now and we have the basic architecture of the software. What's missing is a plan. The purpose of this plan is to go from idea to implementation. As there are two parts of the implementation that each developer tackles individually, the plan needs to include the immediate step of designing the API. As the basis of the API design, the team needs to create a domain model. We'll get to that in a bit.

Putting everything together, the team writes down the following actionable steps:

1. In another whiteboard session, they jointly create a domain model.
2. Max creates the first draft for their API design.
3. Nidhi reviews that draft.
4. Both agree on finalizing the specification, or make edits and review again as necessary.
5. Both work independently on their parts of the implementation.
6. After completion, they integrate their code into one application.

We will follow this plan and walk through all steps in the given order in the current and the upcoming chapters of this book. For the current chapter, we'll focus on the domain model.

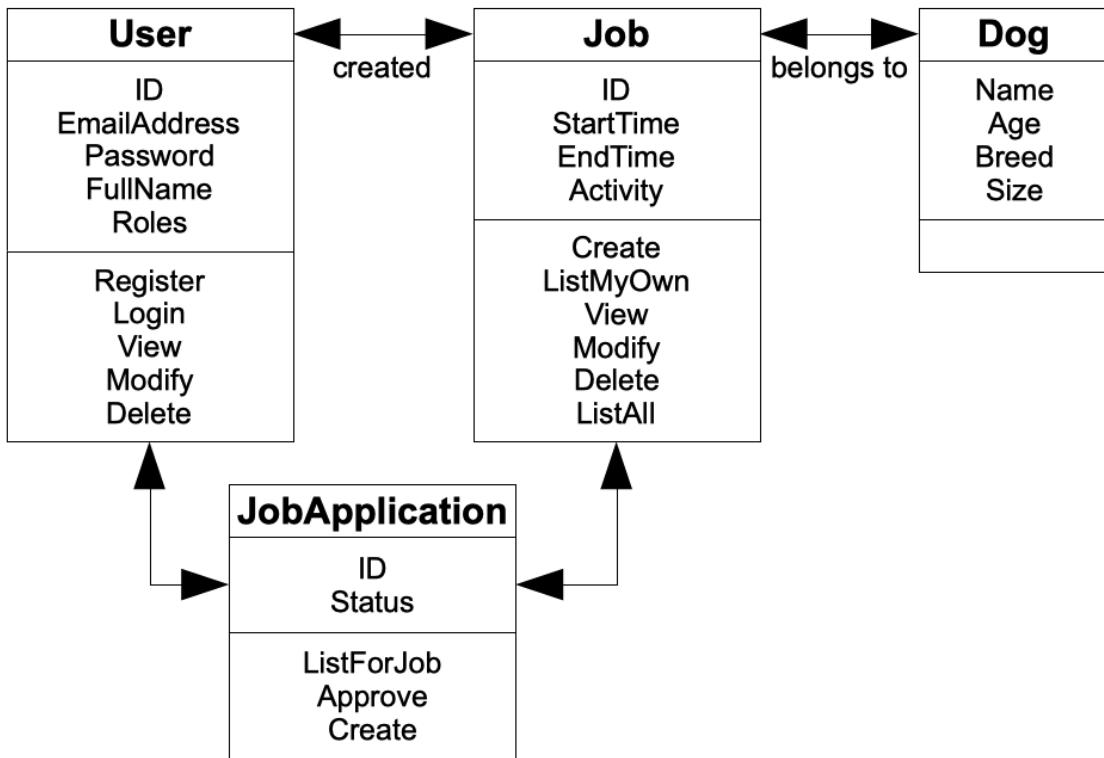


## Figure 9.2 Action Plan

### 9.3 Domain Modeling and APIs

Domain modeling is the process of taking our problem domain and creating a description that can be implemented in computer software. In the upcoming sections, we'll first look at domain modeling in general. Then, we discuss the specifics that we need to consider if we want to create a domain model that works well with an API. As third step, we'll look back at the FarmStall API from the first part of this book, which had a domain model even if we did not explicitly describe it as such.

Afterwards, we're ready to create the model of our new PetSitter application. Here is a sneak peek of what it will look like at the end of the chapter.



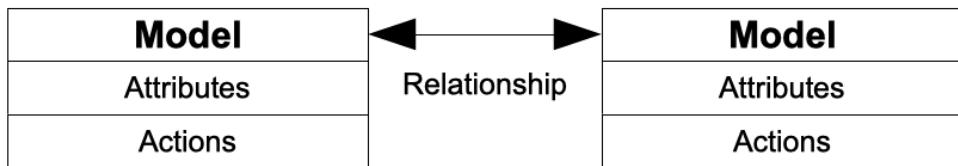
**Figure 9.3 PetSitter Full Domain Model**

### 9.3.1 General Introduction

To create a domain model, we map concepts from the real world onto an abstract representation. A concept is a type of object (or subject) that has attributes and relationships to other objects as well as behavior and/or functionality, which we'll call actions from now on. Attributes are data that describes the object, such as a name, and actions are things that the object can do or what a user can do with the object.

A domain model is not yet dependent on a specific technical implementation and we can express it in different ways. In this chapter, we use a textual representation in the form of bulleted lists and a visual representations through figures that are loosely based on UML (Unified Modeling Language) class diagrams. In this form of visualization, each model is a box with three areas. The upper contains the name, the middle contains the attributes and the bottom contains the functionality. Arrows between these boxes symbolize relationships between the respective models.

You've seen this visualization already in figure 9.3. The diagram contains the attributes, actions, and relationships that we'll identify as we go through this chapter. For a more generalized example of the visualization, refer to figure 9.4.



**Figure 9.4 General Domain Model**

**SIDE BAR**

**Other applications of models**

Models appear everywhere in computing, albeit in different forms. In Object-Oriented Programming (OOP), for example, they appear as classes. In relational databases (such as MySQL or PostgreSQL), they appear as tables. There are always minor differences, but the general idea remains the same. While it may be helpful for you to make a connection between existing implementations and what you're about to learn about domain modeling for APIs, it is not a prerequisite to understanding the next sections.

### 9.3.2 Domain modeling for APIs

Creating a domain model for an API is a crucial task. A class or relational model is mostly an implementation detail that is relevant for those dealing with the inner workings of a component. An API, however, is more akin to the view layer of an application. It is a clearly defined boundary between parts of a system. That includes the possibility to act as an abstraction layer and hide underlying complexity. An API designer should always look at the API from the perspective of the client and not the server.

To take this client-side perspective, Josés team did something that can be considered a good practice when designing APIs for web applications: they put Max, the frontend developer, in charge of the first draft. He's not the one building the API but the one integrating it.

**SIDE BAR** **A word on autogenerated API domain models**

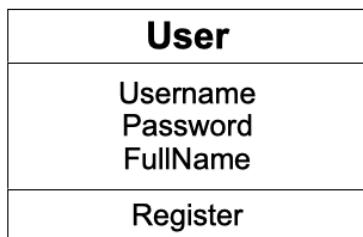
If you have previous experience with medium-sized or larger database-driven web applications and the frameworks used to build them, you may be aware that you need to create different representations of your domain. You have classes in your application layer and tables in your database. And, depending on how the application is built, there is either a manual translation between them or an automated system called an Object-Relational-Mapper (ORM) in place. The API can be considered a third layer with its own domain model. You may be tempted to look for ways to avoid doing API design for an existing application and automate the connection to the other layers. Be careful! I will explain later in this book why this is dangerous territory. At this moment, however, we're starting from scratch anyway.

### 9.3.3 Looking back on FarmStall

In the first part of this book, we learned about OpenAPI using the FarmStall API as a basic example. What is its domain model? As mentioned before, we never explicitly talked about a domain model. However, we can deduce the concepts of the problem domain by looking at the API description.

The two concepts present in the basic version of the FarmStall API are *users* and *reviews*. Let's have a look at these concepts and think about them in terms of attributes, relationships and behavior or functionality.

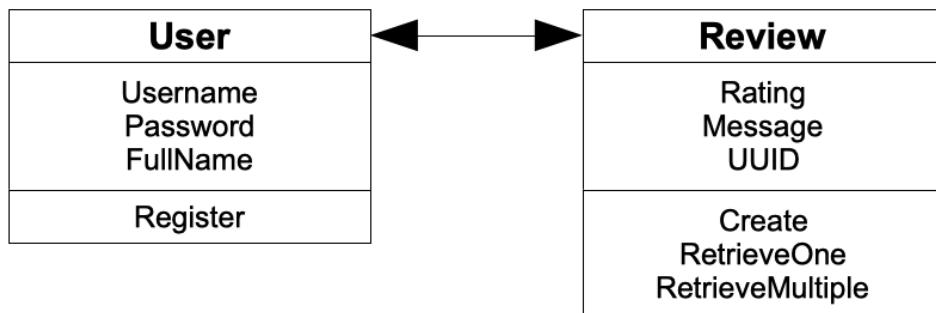
Users have three attributes: a username, a password, and a full name. They also have the ability to register.



**Figure 9.5 FarmStall User Model**

Reviews have at least three attributes: a rating, a message, and a UUID. In case they are not anonymous, they also have a user ID. In the OpenAPI description, that is a fourth attribute. Due to the fact that it is a reference to another concept, in a domain model we would not include it as an attribute but instead describe a relationship between the user and the review concepts.

Reviews can be created and retrieved. For retrieval, it is possible to get all reviews, optionally filtered by rating or a single review based on its ID.



**Figure 9.6 FarmStall User and Review Models**

Without consciously knowing it, you already created a representation of this domain model in OpenAPI, using JSON Schema for attributes and API paths for functionality. Just now we took this API description and created a domain model from it.

In the next chapter, we reverse this process. We take a domain model and transform it into OpenAPI. For that, however, we leave our beloved FarmStall API behind and look at our new PetSitter use case.

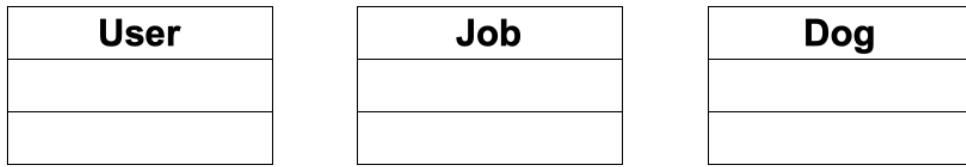
## 9.4 A domain model for PetSitter

José, Nidhi, and Max meet for the second time. José reminds them of his fourth and final non-functional requirement where he said that he wants to get a working prototype out as soon as possible. This working prototype should be usable and provide value to the user but not contain any non-essential functionality. By focussing on the essentials of the application, we can create a simpler domain model, API and implementation.

### 9.4.1 Concepts in the model

As the first step, the team lists all concepts that their domain model will likely contain:

- Pet owners and pet sitters use the application, so we need a *user* model.
- As pet owners post jobs and pet sitters apply to them, we probably have a *job* model.
- The jobs are about dogs, so we may need a *dog* model, too.



**Figure 9.7 Pettersitter Initial Domain Model**

We will now look at the three models we mentioned to list their attributes and relationships.

#### 9.4.2 The user model

As mentioned before in the general introduction and when talking about the FarmStall API, a user appears in almost every domain model. It doesn't mean, however, that this model always looks the same. The attributes, actions, and connections may change significantly depending on the use case. In PetSitter, we already talked about two types of users, pet owners and pet sitters. Our model needs to consider that by including a role attribute.

Apart from the two roles already mentioned, we probably have administrators who moderate the whole marketplace. It's always helpful to include this role to think about administrative duties that happen in the application even if they are not part of a regular user's featureset.

The team collects the following attributes on their whiteboard:

- Email address
- Password
- Full name
- User's Role: pet owner, pet sitter, or admin

#### SIDE BAR What about phone number, address etc.?

One might argue about the need for a phone number and whether or not an email address is sufficient. From the perspective of a real-world PetSitter application, having an easy way to call one another makes sense. Also, for billing purposes a full address may be required. As book authors, however, we don't want you to provide personal data in sample applications unnecessarily, thus we are not including any personal data except for name and email in our domain model.

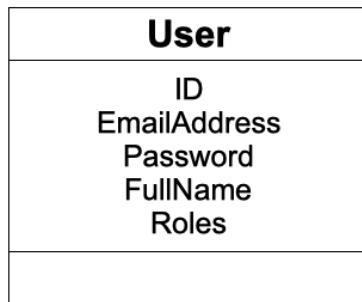
The role leads to a bit of discussion in our team, centered around the following question: Does every user have a single role, or can they have multiple roles?

José believes that a person either *has* a pet or wants to look after *other* pets. Max agrees that

cases where a person might want to do both are uncommon. "However", he argues, "in those rare cases having to register twice for the same application is annoying."

A question like this might seem trivial at first; nevertheless, it not just changes the user experience, it also requires different representations inside the API. And they are a potentially breaking change. That is another reminder how important it is to get your domain model right. The team eventually decides to support multiple roles.

From her perspective as a backend developer, Nidhi makes another suggestion: "An email address can change. A user identifier shouldn't have to. We could add an ID attribute." The team agrees and adds "ID" to their list.



**Figure 9.8 PetSitter User Model**

#### 9.4.3 *The job and dog models*

José asks his developers to brainstorm: "If I asked you to look after my dog, and imagine this is the first time and you haven't met it yet, what is it that you would want to know?" It helps to ask questions like this during domain modeling that force us to look at the model from the perspective of a new application user.

He also reminds the team that they should keep things simple and, even though the name is PetSitter, they can limit the model to dogs for now. Seeing the potential complexity of a more generic pet model, the team agrees. As book authors, we're also happy with that decision since we don't want to dive too deep into sophisticated domain modeling right now at the expense of other aspects of API design.

Nidhi and Max write down their thoughts, compare notes, and present the following joint list to José:

- When is the job, and how long does it take?
- What do you want me to do? Go for a walk, look after the dog at home, or something else?
- Who is the dog? Name, age, breed, etc.?

The first two questions lead to an attribute list for jobs:

- Start time
- End time
- Activity

Instead of start and end time, another possibility would be to include start time and duration. Both versions convey the same information. Also, similar to the user model, we'll add an ID so we can uniquely identify every job posted on PetSitter.

Job
ID
StartTime
EndTime
Activity

**Figure 9.9 PetSitter Job Model**

The third question leads to an attribute list for dogs:

- Name
- Age (in years)
- Breed
- Size (in case people are not familiar with the breed)

Dog
ID
Name
Age
Breed
Size

**Figure 9.10 PetSitter Dog Model**

At this point, we have three concepts in our domain model, but we only looked at their attributes. We now leave the domain model in its current state and complete it later after looking at the application through the lens of user stories.

## 9.5 User stories for PetSitter

For the completion of our domain model and to start implementing the PetSitter application, we need to discuss the connections between the models and the actions that they can take or that can be taken upon them. The team decides to write *user stories* for that. We first introduce user stories as an instrument to describe application functionality, then collect the stories for PetSitter, and finally merge the results with the domain model.

### 9.5.1 What are user stories?

User stories are an informal project management method for requirements analysis during software development. Each user story is written from the perspective of the user of a software product and describes one activity that they perform within the software in order to accomplish something.

Commonly, user stories are written with a template like the following:

*As a <role> I can <capability>, so that <receive benefit>.*

That template includes a role, which makes it work well with applications where users have different roles with different capabilities. The "so that" clause is optional and provides background information on the purpose of the user story.

For user stories that depend on other stories, we can use this template:

*Given <prerequisite>, I can <capability>.*

### 9.5.2 Collecting user stories

We have already seen that user stories support roles and that there are three roles in PetSitter, pet owner, pet sitter, and admin. It makes sense to look at each of them in separation, and, with a multi-person team, we can split the work of brainstorming and writing down user stories.

In the PetSitter team, José takes on the view of a pet owner, and Max puts himself in the shoes of a potential pet sitter. Meanwhile, Nidhi investigates the admin role. To keep the stories short, we present them in separate lists and drop the "as a <role>" prefix. The following lists act as an overview. We'll look closer at each user story afterwards as we map them to actions and relationships for the domain model.

Here are José's results for the pet owner role:

- **I can register a new account and choose my role, so that I can log in.**
- I can log in to my account, so that I can use the marketplace.
- **I can post a job on PetSitter, including a description of one of my dogs, so that pet sitters can apply.**

- I can see a list of jobs I have posted.
- Given that I have posted a job, I can view and modify its details.
- Given that I have posted a job, I can delete it.
- Given that I have posted a job, I can see the pet sitters that applied.
- Given that I have found a suitable candidate, I can approve them.
- I can modify my account details.
- I can delete my account.

Here are Max's results for the pet sitter role:

- **I can register a new account and choose my role, so that I can log in.**
- I can log in to my account, so that I can use the marketplace.
- I can view a list of pets that need looking after.
- **Given that I have found a job, I can apply to it.**
- I can modify my account details.
- I can delete my account.

Here are Nidhi's results for the administrator role:

- I can log in to my account, so that I can access the admin functionality.
- I can modify my account details.
- I can modify other user's account details.
- I can edit jobs that other users have posted.
- I can delete users.

Four of the user stories (printed in bold text) directly correspond to the functional requirements that José wrote down initially, so it seems his team is on the right track to build the application he wants. We can check them off the requirements checklist.

**Table 9.3 Requirements Checklist**

Type	Requirement	In plan?
Functional	Sign up: As dog owner or dog walker	OK
Functional	Dog owners: can post jobs	OK
Functional	Dog walkers: can apply to posted jobs	OK
Non-functional	Build web app with in-house team - two people	OK
Non-functional	Mobile app - work with other development agency (later!)	OK
Non-functional	Chance to experiment with new technology	OK
Non-functional	Release first working prototype as soon as possible	OK

Having collected all stories, we can now investigate them and update our models as needed.

### 9.5.3 Mapping user stories

Previously, we created three models, *user*, *job* and *dog*. They both already have a set of attributes. To find out about functionality or relationships, we should walk through the user stories and see which of the models they affect. If they affect only one of the models, we can add an action to that model. If they affect multiple models, we also have to look at it from a relationship perspective.

To start, we look at those user stories that appear in the same or a similar fashion for multiple roles:

#### **I CAN REGISTER A NEW ACCOUNT AND CHOOSE MY ROLE.**

This user story appears for both pet owners and pet sitters. Registration is a prerequisite to using the application and is independent of any jobs. For the user model, we can derive the action "Register" from it.

#### **I CAN LOG IN TO MY ACCOUNT.**

This user story appears for all three roles and is also independent of any jobs. We can assume having some "Login" action in our application's user model.

#### **I CAN MODIFY MY ACCOUNT DETAILS.**

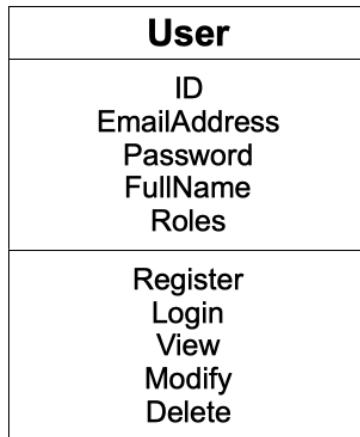
This user story also appears for all three roles and adds a "Modify" action to the user model. Although not explicitly mentioned here, we can safely assume that a user needs to retrieve and see their details first before making any changes. Therefore we can also add a "View" action to the user model.

#### **I CAN DELETE MY ACCOUNT.**

This user story appears for both pet owners and pet sitters and adds a "Delete" action to the user model.

So far, we have identified some actions for the user model, but we haven't touched jobs, dogs, or any relationships yet. Here are the actions:

- User: Register
- User: Login
- User: View
- User: Modify
- User: Delete



**Figure 9.11 PetSitter User Model with Actions**

Now, let's have a look at the list of user stories for pet owners:

**I CAN POST A JOB ON PETSITTER, INCLUDING A DESCRIPTION OF ONE OF MY DOGS.**

This story calls for a "Create" action related to the jobs model. It also includes the dog model, so let's take a closer look at that.

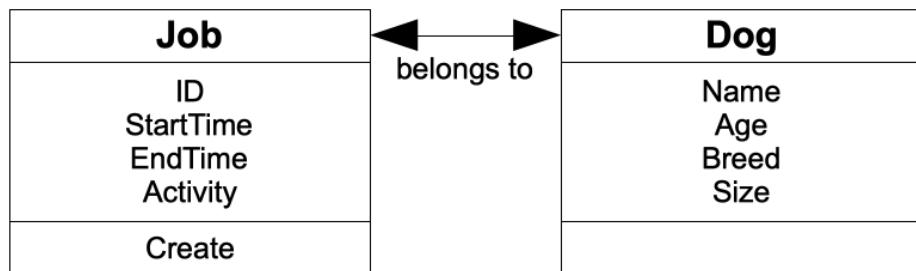
According to the user story, posting a job and including the dog that the job is about is a single step. There is no preceding user story for adding a dog to the PetSitter application, which might also lead to the need for user stories for listing, editing and deleting dogs. That is a design choice. José says that it keeps the application simple and the developers wholeheartedly agree as it is less actions to implement.

What about relationships? Well, due to the inclusion of the dog in every job posting, there is a strong connection between the dog and the job models. It is a one-to-one mapping, which means that for every job there is exactly one dog, and every dog is assigned to exactly one job. As a result of this, we can drop the ID from the *dog* concept as it is no longer required because we can identify each dog by the job it belongs to.

Of course, there might be scenarios where jobs ask the pet sitter to look after multiple dogs. They are, however, not covered by this user story which explicitly mentioned "one of my dogs" and, as we said before, we want to keep the model simple. At the same time it is very likely for one pet owner creating multiple jobs for the same dog over the course of time. As the dog's description is included in the job, though, it would be a different dog model even if it is the same dog in the real world.

There are two lessons to be learnt here. One is that there is no perfect mapping between an instance of a concept in reality and an instance of the same concept in the domain model. The

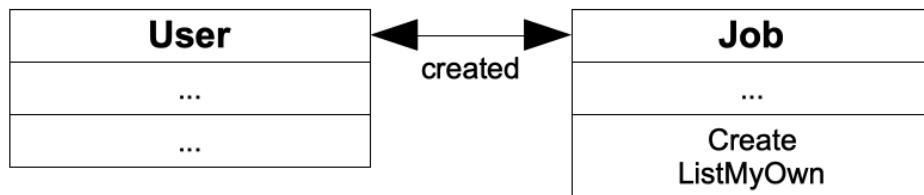
second lesson is that the way we write our user stories and, hence, how we want users to interact with our applications, can throw assumptions off the rails.



**Figure 9.12 PetSitter Job and Dog Models**

### I CAN SEE A LIST OF JOBS I HAVE POSTED.

From this story, we can assume a "List my own" action for jobs, which also requires knowing the user that created them. Since that connection is needed so users can list their jobs, we can now draw a relationship between the user and job and call it "user created job".



**Figure 9.13 PetSitter User and Job Create Relationship**

### GIVEN THAT I HAVE POSTED A JOB, I CAN VIEW AND MODIFY ITS DETAILS.

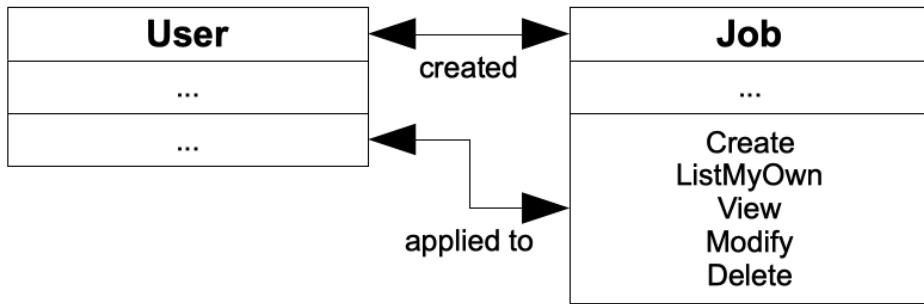
This user story adds a "View" and a "Modify" action to the job model. It doesn't tell us anything new about relationships.

### GIVEN THAT I HAVE POSTED A JOB, I CAN DELETE IT.

This user story is similar to the previous one and adds a "Delete" action to the job model.

### GIVEN THAT I HAVE POSTED A JOB, I CAN SEE THE PET SITTERS THAT APPLIED.

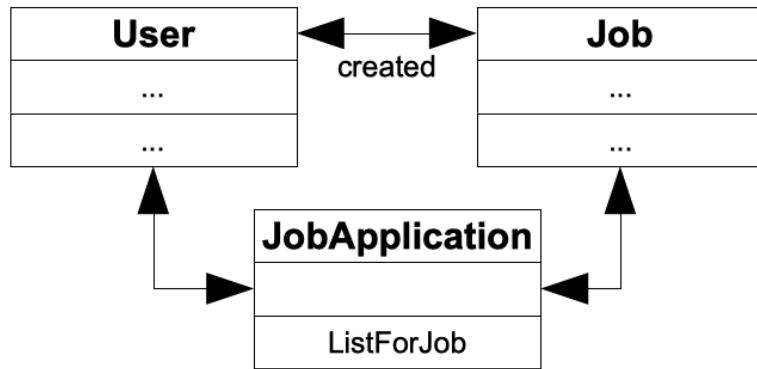
Users with the pet sitter role can apply to jobs. The application process itself is another user story that we'll look at when we go through the user stories for pet sitters. The current user story takes the perspective of the pet owner who wants to see these pet sitters. To support it, we could draw a second relationship between the user and the job models and call this a "user applied to" relationship.



**Figure 9.14 PetSitter User and Job Application Relationship (first approach)**

What could be a proper name for an action that corresponds to this user story, and to which model should it belong? It could be "List applications". It probably doesn't belong to *dogs*, but is it for *users* or *jobs*? Somehow it is about both, and we also introduced a new noun - "application" - in the action name. Maybe we have to revise our domain model? If we can't name an action without a new proper noun, it's an indicator that we need new concepts in the domain model.

So, we can create a model named *job application* and connect it to both *user* and *job*. In this way, we can have a "List for Job" action for the new model. This action also includes a noun but that's okay since it's a noun that already exists as a concept.



**Figure 9.15 PetSitter User and Job Application Relationship (improved approach)**

### GIVEN THAT I HAVE FOUND A SUITABLE CANDIDATE, I CAN APPROVE THEM.

A candidate for a job is a user that applied to that job, or, in other words, someone who created an application. We add an action to the *job application* model, and we call it "Approve".

In the initial stage where we created the attributes for our domain model, we didn't yet have the *job application*. However, our domain model should be the result of the "Approve" action. Therefore the team decides to add a "Status" attribute, which could indicate *applying* or *accepted*. Also, in consistency with the other models, the job application gets an "ID" attribute.

JobApplication	
ID	Status
ListForJob	Approve

**Figure 9.16 PetSitter Job Application**

Great, we've completed all user stories for pet owners, and we have updated our domain model with relationships. Before we move on to the user stories for pet sitters, let's recollect all the actions we have identified so far:

- Job: Create
- Job: List my own
- Job: View
- Job: Modify
- Job: Delete
- Job Application: List for Job
- Job Application: Approve

Now, let's look at the pet sitter user stories:

### **I CAN VIEW A LIST OF PETS THAT NEED LOOKING AFTER.**

As we've established in our domain model, pets, or dogs, are created and listed as part of the jobs. Thus the list that the pet sitter can view is not a list of pets but rather a list of jobs. We can add an action to the job model and call it "List all". This user story does not require any changes to the relationships.

### **GIVEN THAT I HAVE FOUND A JOB, I CAN APPLY TO IT.**

This user story establishes the *job application* that connects a user and a job, so we can add a "Create" action to our new concept.

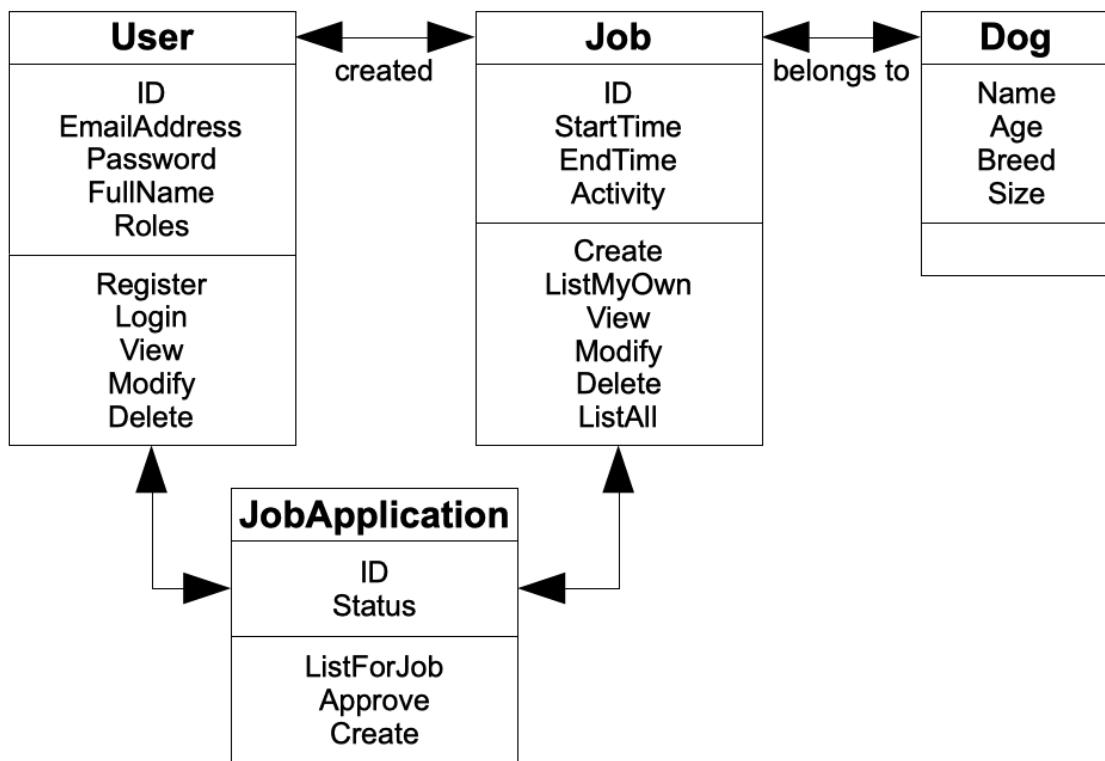
Awesome, we got through all of the pet sitter user stories now. They work with our domain model and don't require any new or modified relationships. We can add the following actions to our collection:

- Job: List all
- Job Application: Create

Last but not least, let's look at the user stories for the administrator:

- I can modify other user profiles.
- I can edit jobs that other users have posted.
- I can delete users.

One thing that these stories have in common is that we already defined actions like "View", "Modify", and "Delete" for users and jobs. The only difference is that regular users can only execute these actions for themselves or jobs they have created and the administrator. We have to consider these user stories when implementing permissions in the backend, but they do not result in changes to our domain model.



**Figure 9.17 PetSitter Full Domain Model**

Great, it seems we're done with this step. José, Nidhi and Max leave the meeting with a photograph of the whiteboard containing the full domain model. According to the plan, it's now Max' responsibility to turn it into the first version of the OpenAPI description.

## 9.6 Summary

- PetSitter is an application that connects busy dog owners with job-seekers who want to take care of them. It is the scenario that we work on throughout the second part of this book.
- We have a plan that involves a team with two developers, one for backend and one for frontend. First we will design an API for the application, then build then two parts, and finally integrate both. We cover this iterative process in the coming chapters.
- Domain modeling is the process of creating a representation of concepts in a problem domain. It is the first step for building an API and starts before writing the OpenAPI description. PetSitter's domain model includes a user, a job, a dog, and a job application concept.
- User stories can help with domain modeling, especially with defining actions for the models and relationships between different models. As the result of writing and analyzing user stories, we have updated the domain model with four relationships and a list of actions for users, jobs, and job applications. The complete model forms the basis of the OpenAPI work.

/= **Designing APIs with Swagger and OpenAPI** /:chapter: 10 /:sectnums: /:figure-caption:  
 Figure 9. /:listing-caption: Listing 9. /:table-caption: Table 9. /:sectnumoffset: 1 /:leveloffset: 1  
 /:source-highlighter: pygments /:pygments-style: emacs /:icons: font

# 10

## *Creating an API design using OpenAPI*

### This chapter covers

- Creating reusable schemas in OpenAPI
- Converting the PetSitter domain model into schemas
- Designing an API following the CRUD approach
- Creating paths and operations for the PetSitter API

In the previous chapter, we got to know José and his team who are building the PetSitter application. We accompanied them through their initial meeting in which they created an action plan for building the application. We also joined their domain modeling whiteboard session in which they prepared a high-level domain model.

The domain modeling session was the first item on their action plan, which leaves us with the following steps:

1. Max, the frontend developer, creates the first draft for their API design.
2. Nidhi, the backend developer, reviews that draft.
3. Both agree on finalizing the specification, or make edits and review again as necessary.
4. Both work independently on their parts of the implementation.
5. After completion, they integrate their code into one application.

In this chapter, we'll go through the first of the remaining steps together with Max.

### 10.1 The problem

In the previous chapter, we created a domain model. That model is an informal, high-level representation of the concepts underlying the PetSitter application. In the implementation stage of the project, we will develop a frontend and a backend part, connected with an API. Hence, we now have to bridge the gap between these two, and we do this with the formal description of the API using OpenAPI.

At the end of this chapter, we should have an OpenAPI file that satisfies the following three objectives:

- It is a valid representation of the domain model, i.e., it fulfills the requirements for the project.
- A backend developer can use it to create an implementation of the API.
- A frontend developer can write code to integrate the API.

### 10.1.1 Converting a domain model into OpenAPI

In a domain model, we assign attributes, relationships and functionality to various concepts. To understand the conversion of each of these to OpenAPI, let's take a look back at the FarmStall API. We already showed a domain model for FarmStall in the previous chapter where we identified users and reviews as the main concepts. We are revisiting this model now and take a closer look at the OpenAPI description to see how to do a mapping between the two.

To retrieve a list of reviews in the FarmStall API, users can make an API call to the `GET /reviews` operation. In chapter 5, we used JSON Schema to describe the response. As a reminder, here is a listing of the schema part of the OpenAPI description for this response:

#### **Listing 10.1 GET /reviews response schema**

```
type: array
items:
  type: object
  properties:
    uuid:
      type: string
      pattern: '^[0-9a-fA-F\\-]{36}$'
    message:
      type: string
    rating:
      type: integer
      minimum: 1
      maximum: 5
    userId:
      type: string
      pattern: '^[0-9a-fA-F\\-]{36}$'
      nullable: true
```

The schema describes the properties of an object. In this case, `uuid`, `message`, `rating` and `userId`. In the previous chapter, we showed a domain model for reviews in which UUID, message and rating were attributes and the user ID turns into a relationship between a review and a user model.

Generally speaking, schemas in OpenAPI definitions are representations of the attributes and relationships for concepts in a domain model. We'll stick to using the term "model" or "concept" for the high-level domain model representation and the word "schema" for its technical

implementation as a data structure. In other documentation in the context of domain modeling with OpenAPI, however, you may also see the words "model" and "schema" used interchangeably.

If you look at the HTTP method `GET` and the URL `/reviews`, you can read it as "Get Reviews". This is a "Get" action taken on the *reviews* concept in the domain model. The functionality or behavioral parts of the domain model are represented through the API operations in OpenAPI. We will look at operations later in this chapter and focus on schemas first.

### 10.1.2 Ensuring reusability

In the definition of the FarmStall API's `GET /reviews` operation in chapter 5, we provided the schema as part of the operation. That is called an "inline schema".

Let's look at another functionality of the FarmStall API: adding reviews. To do so, users can make an API call to the `POST /reviews` operation. In chapter 6, we created this operation with a request body. Again, as we described the data structure as part of the request itself. In other words, we provided an inline schema. As a reminder, here is a listing of the schema part of the OpenAPI description for this request:

#### **Listing 10.2 POST /reviews request schema**

```
type: object
properties:
  message:
    type: string
    example: An awesome time for the whole family.
  rating:
    type: integer
    minimum: 1
    maximum: 5
    example: 5
```

If we do a comparison of both listings, we notice some duplication:

- Both have a `message` property with a `string` type.
- Both have a `rating` property with an `integer` type, a minimum constraint of `1`, and a maximum constraint of `5`.

Now, assume that you wanted to change the rating scale so that users cannot rate between `1` and `5` but between `1` and `10`.

We've already identified two places where you would have to make a change. However, for the sake of brevity, we only included two inline schemas here. The `POST /reviews` endpoint returns a response that echoes the review back, resulting in a third inline schema. Additionally, further down in chapter 6, there is the `GET /review/{reviewId}` operation into which we copied and pasted the same response format.

In total, there are four places to make a change. Changing something in four places in a file doesn't seem like an impossible burden for the developer. It still introduces a margin for errors. Imagine what could go wrong if the change was made in only some places, for example in `POST /reviews` but not in `GET /reviews`. A developer integrating the `GET /reviews` would be under the assumption that there's a maximum of 5, so they might design their API client with this expectation. For example, they build a visual user interface with five stars. At the same time, another developer would send reviews with a rating of ten. Those could not be displayed in the first developer's application.

Consistency is key for API design. Hence, having a way to define a schema only once and the using it throughout the OpenAPI description, would be really helpful. Apart from the practical advantages, it also provides a closer mapping between the domain model and its implementation because a single schema in OpenAPI represents exactly one concept from the domain model. And that is what we're about to do.

## 10.2 Creating the schemas

In this section, we will create reusable schemas and you'll learn where they are located in an OpenAPI file. To get there, however, we have to create a new OpenAPI file first.

### 10.2.1 Starting an OpenAPI file with schemas

To start the new OpenAPI definition for PetSitter, use <https://editor.swagger.io>, the tool you already got to know in chapter 4. After opening the website, clear out the editor, so you can start writing on a blank slate.

As a reminder, for a new OpenAPI file, you need to do the following:

- Specify the version of OpenAPI you're using.
- Add an `info` section with `title` and `version`.
- Add an empty `paths` object if you do not define any operations yet, because otherwise you will get a syntax error.

Your first file should look like this:

#### Listing 10.3 Minimal PetSitter OpenAPI file

```
openapi: 3.0.0
info:
  title: PetSitter API
  version: "0.1"
paths: {}
```

There is another top-level element for OpenAPI files, `components`. The idea behind `components` is that it is a container in which you can define elements of your API that do not belong to a specific path. You can add references to components in various places throughout your API description.

In chapter 7 you have already seen this element where we used it to create a security scheme definition for the API. Now, we use `components` and its sub-element `schemas` to define reusable schemas for the API. After adding these container elements, your OpenAPI file should look like this:

#### **Listing 10.4 PetSitter OpenAPI with schemas**

```
openapi: 3.0.0
info:
  title: PetSitter API
  version: "0.1"
paths: {}
components:
  schemas: {}
```

The OpenAPI file is now ready for schema descriptions of our *user*, *job*, *dog*, and *job application* concepts from the domain model.

#### **10.2.2 Referencing common schemas**

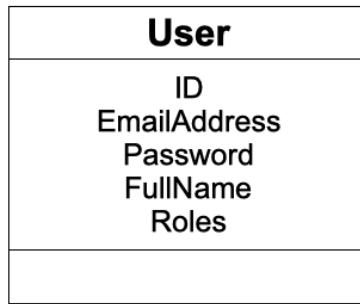
Once we have created common schemas in the `components` section of our OpenAPI file, we use the `$ref` keyword to add references to it. Those references can be used in requests, responses, and even other schemas. The value for the `$ref` keyword is a JSON pointer that describes where we can find the schema in the hierarchical structure of our OpenAPI file. The JSON pointer starts with the hash symbol (#), followed by the path `/components/schemas/`, and ends with the name that we used for our schema:

```
$ref: '#/components/schemas/User'
```

#### **10.2.3 The User schema**

According to the results of the domain model discussions from the previous chapter, a *user* has the following attributes:

- ID
- Email Address
- Password
- Full Name
- User's Roles (i.e., whether they have a pet and want to provide jobs or whether they're looking for a pet-sitting job, or both)



**Figure 10.1 PetSitter User Model**

We have to create an `object` schema with multiple fields, where each field, or property, represents one of these attributes.

When converting the attributes, we should follow the naming convention for JSON objects that says that all properties are lowercase and without spaces. We also have to add a `type` to each of them. Considering these two requirements, we can create a table of properties.

**Table 10.1 The User fields and their types**

Field	Type
<code>id</code>	<code>integer</code>
<code>email</code>	<code>string</code>
<code>password</code>	<code>string</code>
<code>full_name</code>	<code>string</code>
<code>roles</code>	<code>array (of strings)</code>

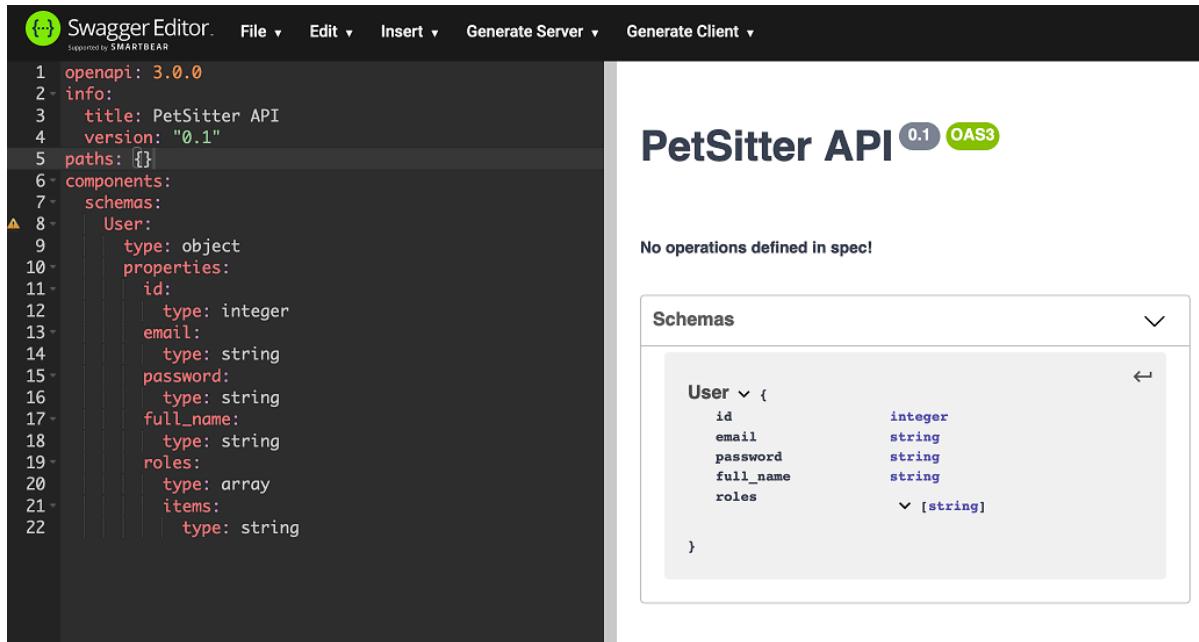
The `ID` is an `integer`. `Email`, `password` and `full_name` have the `string` type, which is a sensible default unless we can be sure that they only contain numeric or boolean values. The `roles` field is an `array`, because users can have multiple roles. Each role is a `string` itself, so we can set the `array`'s `items` type keyword to `string`.

Now, let's add our schema to the OpenAPI description. To do so, you need to provide the name of your schema as the YAML key under `schemas` and the description below it. Unlike property names, which are lowercase, schema names typically start with an uppercase letter. Hence, we create schema with the name `User`. Here is the OpenAPI file with our first schema:

## Listing 10.5 PetSitter OpenAPI with User schema

```
openapi: 3.0.0
#...
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        email:
          type: string
        password:
          type: string
        full_name:
          type: string
        roles:
          type: array
          items:
            type: string
```

When adding this code to Swagger Editor, you can see a new section called "Schemas" appear in the right column of the editor. Inside the section, you can expand the "User" model and see the properties you have defined.



The screenshot shows the Swagger Editor interface. On the left, the OpenAPI specification is displayed in a code editor:

```
1 openapi: 3.0.0
2 info:
3   title: PetSitter API
4   version: "0.1"
5 paths: {}
6 components:
7   schemas:
8     User:
9       type: object
10      properties:
11        id:
12          type: integer
13        email:
14          type: string
15        password:
16          type: string
17        full_name:
18          type: string
19        roles:
20          type: array
21          items:
22            type: string
```

On the right, the "Schemas" section is expanded, showing the "User" schema with its properties:

```
Schema: User
User {
  id: integer
  email: string
  password: string
  full_name: string
  roles: [string]
}
```

Figure 10.2 Swagger Editor with User Model

### 10.2.4 The Job schema

In the PetSitter domain model, a *job* has the following attributes:

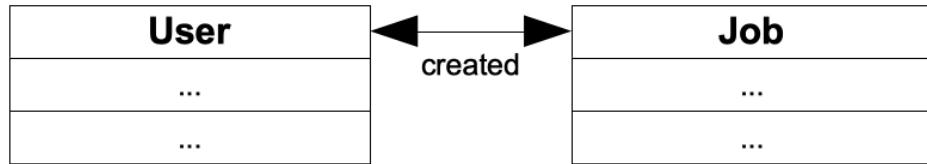
- ID
- Start time
- End time

- Activity

Job
ID
StartTime
EndTime
Activity

**Figure 10.3 PetSitter Job Model**

Similar to what we did in the User schema, we also make the ID an `integer` and use `string` for everything else. Relationships in domain models also lead to properties in the schema. We saw this in the FarmStall API where there was a `user_id` field in a review. Therefore, we also have to look at the relationships. So far, we have User and Job schemas, so we can only look at the relationship between them, "user created job".



**Figure 10.4 PetSitter User and Job Create Relationship**

To reference the user who created the job, we can add a field that includes the ID of the user. A common naming approach for these fields is to use the name of the target schema, followed by an underscore and then the name of a field on the target schema that contains the unique identifier, e.g., `id`. With this approach, the property name would be `user_id`. We can also be more specific and include a description of the relationship into the name, which is especially helpful in case we have more than one relationship between the same two schemes. With that said, let's call it `creator_user_id` then.

**Table 10.2 The Job fields and their types**

Field	Type
<code>id</code>	<code>integer</code>
<code>creator_user_id</code>	<code>integer</code>
<code>start_time</code>	<code>string</code>
<code>end_time</code>	<code>string</code>
<code>activity</code>	<code>string</code>

Here is the OpenAPI file after adding our second schema:

### **Listing 10.6 PetSitter OpenAPI with Job schema**

```
openapi: 3.0.0
#...
components:
  schemas:
    User:
      #...
    Job:
      type: object
      properties:
        id:
          type: integer
        creator_user_id:
          type: integer
        start_time:
          type: string
        end_time:
          type: string
        activity:
          type: string
```

#### **10.2.5 The Dog schema**

When creating the dog model in the previous chapter, the team listed the following attributes:

- Name
- Age (in years)
- Breed
- Size (in case people are not familiar with the breed)

We can use an `integer` for the age, as it is a number, and `string` for everything else. Just as before, we list the attributes in a table first.

**Table 10.3 The Dog fields and their types**

Field	Type
name	string
age	integer
breed	string
size	string

Then, we add our third schema to the OpenAPI file:

## Listing 10.7 PetSitter OpenAPI with Dog schema

```
openapi: 3.0.0
#...
components:
  schemas:
    User:
      #...
    Job:
      #...
    Dog:
      type: object
      properties:
        name:
          type: string
        age:
          type: integer
        breed:
          type: string
        size:
          type: string
```

Now, what about the "dog belongs to job" relationship? As we realized while processing the user stories, this is a one-to-one mapping, and pet owners create dogs as part of the jobs they post. It follows that we have to somehow include the Dog schema into the Job schema. We can do that with a reference: the Job schema gets an additional property called `dog` with a `$ref` pointer to the new schema. That way, the dog's description becomes part of the job, just as intended. Here is how that looks like in OpenAPI:

## Listing 10.8 PetSitter OpenAPI with Job schema, referencing Dog

```
openapi: 3.0.0
#...
components:
  schemas:
    User:
      #...
    Job:
      type: object
      properties:
        id:
          type: integer
        creator_user_id:
          type: integer
        start_time:
          type: string
        end_time:
          type: string
        activity:
          type: string
        dog:
          $ref: '#/components/schemas/Dog'
    Dog:
      #...
```

### 10.2.6 The Job Application schema

Our fourth and last schema is the Job Application, which has the following attributes:

- ID
- Status

The ID can be an `integer` again and status can be a `string`. We also have relationships to the Job and User schema, as every job application is created by a user and dedicated to one specific job. To support them, we can add a `user_id` and a `job_id` field with `integer` types (because `id` on Job and User is an `integer`).

**Table 10.4 The JobApplication fields and their types**

Field	Type
<code>id</code>	<code>integer</code>
<code>status</code>	<code>string</code>
<code>user_id</code>	<code>integer</code>
<code>job_id</code>	<code>integer</code>

Awesome, now let's update the OpenAPI file with the final schema:

**Listing 10.9 PetSitter OpenAPI with JobApplication schema**

```
openapi: 3.0.0
#...
components:
  schemas:
    User:
      #...
    Job:
      #...
    Dog:
      #...
    JobApplication:
      type: object
      properties:
        id:
          type: integer
        status:
          type: string
        user_id:
          type: integer
        job_id:
          type: integer
```

Great, we now have four complete schemas in our OpenAPI file. Good work, Max! Let's continue with the actions in our domain model. Before we can look at them, however, it's time for some more theory as we learn about CRUD.

### 10.3 The CRUD approach to API operations

The abbreviation CRUD stands for Create-Retrieve-Update-Delete. Originally, CRUD comes from the world of database management systems. It describes the essential operations that can be executed upon a certain piece of data.

CRUD as an API design paradigm fits in nicely with some of the concepts of REST which

you've heard about in chapter 1. URLs represent resources and the different HTTP methods (or verbs) represent the operations. That, in turn, leads to a certain approach in designing the URL paths and the operations available on them. You have already seen the approach in the FarmStall API as well as some of the external examples throughout part one of this book. Before we can reproduce it for PetSitter, however, let's take another look at it and work out the specifics.

We can typically make a distinction between two kinds of URL paths in an API:

- Resource endpoints
- Collection endpoints

Later in this chapter, you'll learn about a third type of endpoint, but let's focus on these two first. Also, let's agree on a definition for the term "resource". Resources are individual instances of a concept in the domain model. We have, for example, *user* and *job* models. Therefore, every specific user is a resource, and so is every specific job.

The best practice for the path to an individual resource is to use the pluralized name of the model, followed by a slash and a unique identifier for the instance. For example, if there's a user with an ID of 123, you can access it as `/users/123`. If you think of the URL as a directory structure (which, for static websites, it actually is!), you can imagine a folder called *users* which contains one file for each user.

We call the URL to an individual resource a resource endpoint. On a resource endpoint, you can use HTTP GET to retrieve the resource, PUT or PATCH to update it and DELETE to remove it.

Accessing individual resources, however, is not sufficient. Often you have to retrieve a list of resources of the same type, i.e., instances of the same concept in the domain model. For this purpose, you use the pluralized name of the model without suffixing it with an ID. To follow the filesystem analogy, you open a folder instead of just a specific file.

We call the URL to a resource listing a collection endpoint. On a collection endpoint, you can use HTTP GET to retrieve all resources of the same type. You may argue, of course, that it's not practical to retrieve all resources if there are thousands or millions of them. Thus, there are concepts like pagination and filters. You saw filters in chapter 2 where the collection endpoint GET `/reviews` of the FarmStall API allows a `maxRating` parameter to only return reviews with a certain rating value. As far as pagination goes, we will discuss it later in this book.

It's also common to use the collection endpoint URL combined with the HTTP POST method to create a new element for which the ID is not yet known, as it is assigned by the server.

**Table 10.5 CRUD Operations, Methods and Paths**

Operation	Method	Typical path
Create	POST	Collection endpoint ( <code>/{{schema}s}</code> )
Read	GET	Both collection and resource endpoints
Update	PUT / PATCH	Resource endpoint ( <code>/{{schema}s/{{id}}}</code> )
Delete	DELETE	Resource endpoint ( <code>/{{schema}s/{{id}}}</code> )

### 10.3.1 Defining API requests and responses

In an interaction with an API, the operation defines the URL, the HTTP method, and, optionally, request parameters and a response body that the client sends to the server. The server then sends back a response, which contains a HTTP status code and a response body. In chapter 5 we already introduced HTTP status codes so you can refer to that chapter for more information about them.

The CRUD approach to designing an API includes some rules for requests and responses. We'll cover the theory in this section and then follow up demonstrating how it looks in practice as we add operations to the PetSitter OpenAPI description.

## REQUESTS

For "Read" and "Delete", there is no request body. If necessary, input for "Read" operations, such as filter criteria, typically goes in query parameters.

For "Create" and "Update", you need to send the JSON representation of the resource, i.e., a JSON structure following the schema, as the request body.

## RESPONSES

For "Read" on resource endpoints, the response is a JSON object following the schema of the resource. A successful API call gets a status code 200 ("OK"). If the requested resource does not exist, the API should return a 404 ("Not Found").

For "Read" on collection endpoints, the response is a collection object that contains an array of resource objects, optionally accompanied by additional fields with meta data. The field that contains the array of items is often called `items` or the name of the resource. Let's look at an example of a response structure.

## Listing 10.10 Collection Endpoint Response Example

```
#...
responses:
  '200':
    description: A list of items
    content:
      application/json:
        schema:
          type: object
          properties:
            items:
              type: array
              items:
                $ref: '#/components/schemas/Item'
```

### SIDE BAR Why not return a top-level array?

We could simplify the structure above by not having an `object` with an `items` field and instead just make the whole response an `array`. There are two reasons why this is discouraged. One is that there is a security vulnerability in some older browsers that allows a circumvention of CORS (cross-origin-resource-sharing) restrictions when the top-level element is an `array`. The second reason is that we may need some additional fields with meta data, especially when we use pagination. An example of such meta data could be the count of items available but not returned in the current API response. Even if we have no apparent use for them, we should follow the best practice of never returning top-level arrays in the APIs we design.

For collection endpoints, every API call should return status code 200, even if the collection is empty.

### SIDE BAR Why not use 404 for empty collections?

An API call to a resource endpoint receives a 200 status if the resource exist and 404 if it doesn't. A collection endpoint always returns 200, even if the collection is empty. The reason is that the collection itself still exists even if there are no resources in it. If you think of it as a bucket, it makes sense, because an empty bucket is still a bucket and you can grab it and do something with it.

For "Update", the resource endpoints responds with a JSON object following the schema of the resource. That results in symmetry of request and response and also consistency between "Read" and "Update" as the resource endpoints give the same response independent of the HTTP verb. For continued symmetry, update requests also return a 200 status code.

For "Delete", the response from the resource endpoint is typically empty. This is not consistent with the other operations on the resource endpoint. However, it makes sense as after a "Delete",

the resource no longer exists. A successful deletion request returns a 204 ("No Content") status code because 200 expects a response body.

For "Create", which is a POST request on the collection endpoint, the response body is typically empty. Instead, the response contains a `Location` header that points to the resource endpoint for the newly created resource. A successful creation request returns a 201 ("Created") status code.

**Table 10.6 CRUD Responses**

Operation	Status Code	Response Body
Create	201	Empty, with <code>Location</code> header
Read	200	Resource or collection object
Update	200	Resource
Delete	204	Empty

### 10.3.2 From user stories to CRUD design

You might now conclude that API design is about following the mechanical process of taking your models, adding collection and resource endpoints for each of them, and specifying all CRUD operations, i.e., GET and POST for collection endpoints and GET, PUT and DELETE for resource endpoints. And indeed, the API that results from this process might be what API designers call a well-designed, consistent API. Unfortunately, however, it may not be the API that your consumers, e.g. your web application, needs.

In the previous chapter, Josés team wrote user stories to cover the requirements of the PetSitter application. As API designers, you should look at those stories and ask yourself:

- Does this user story match with one or more CRUD operations? If so, make sure you include these in your API design.
- Does the story require a different kind of operation? If so, is there a way I can include this in my API that still feels "right" from a resource-oriented (CRUD/RESTful) perspective?
- Are there CRUD operations that none of my user stories need? If so, you can and should leave them out of the API design.

## 10.4 API operations for PetSitter

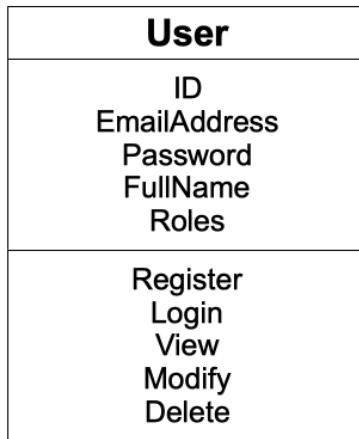
With our toolbox prepared, we can now move on to implement the respective operations for the PetSitter API. There are no actions on the *dog* concept, but we still have to walk through the actions for *user*, *job*, and *job application*.

### 10.4.1 User operations

Reviewing our PetSitter user model, we find the following actions:

- Register
- Login

- View
- Modify
- Delete



**Figure 10.5 PetSitter User Model with Actions**

Let's take a closer look at each action and see how it matches the CRUD operations.

## REGISTER

Registration is the initial action that a user takes to create their representation in the application, i.e., their account. Therefore we can map this action to "Create" and add a POST operation on the `/users` path, which is the collection endpoint for the User resources, to our OpenAPI description. The request for that operation includes the User schema as a component reference with `$ref`, and the response has a 201 status code and a `Location` header.

### Listing 10.11 PetSitter OpenAPI Register User

```

openapi: 3.0.0
#...
paths:
  /users:
    post:
      summary: Register User
      responses:
        '201':
          description: Created
          headers:
            Location:
              schema:
                type: string
        requestBody:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
#...
  
```

## LOGIN

A user logs in to an application when they want to start using it. In the login process, users authenticate themselves and the application ensures that they are authorized for access. We already learned about authentication in APIs in chapter 7 of this book, so we might assume that this action is not represented as an operation but rather relates to the security section of the OpenAPI description. Max decides to skip this action for now and work on security later.

## VIEW

We added this action from the user story about modifying user details under the assumption that the user needs to retrieve their profile before they can modify it. In the CRUD model, this would be a "Read" on a single resource. In the API, we can add a GET operation on the resource endpoint for users, `GET /users/{id}`.

## MODIFY

After viewing a profile, the user can modify it. In other words, they can do an "Update". Again, this would be on a single resource, so we can add a PUT operation on the resource endpoint for users, `PUT /users/{id}`.

## DELETE

Users can delete themselves. This can be a DELETE operation on the resource endpoint for users, `DELETE /users/{id}`.

In an OpenAPI file, we specify paths and then all operations below them. For the three actions that use the resource endpoint for users, we need a common path parameter for the ID. As a reminder, we introduced path parameters in chapter 6. Only the "Modify" operation needs a request body. Here is a listing of these three operations in the OpenAPI file. As you look at the code, take note of the references of the User schema, especially how we have placed a reference to the same schema for request and response bodies for the PUT operation:

## Listing 10.12 PetSitter OpenAPI User Operations

```

openapi: 3.0.0
#...
paths:
  /users:
    #...
  /users/{id}:
    parameters:
      - schema:
          type: integer
          name: id
          in: path
          required: true
    get:
      summary: View User
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
    put:
      summary: Modify User
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
    delete:
      summary: Delete User
      responses:
        '204':
          description: No Content

```

### 10.4.2 Job operations

Looking at the *job* model from the previous chapter, we have collected the following actions:

- Create
- List my own
- View
- Modify
- Delete
- List all

Job
ID StartTime EndTime Activity
Create ListMyOwn View Modify Delete ListAll

**Figure 10.6 PetSitter Job Model with Actions**

## CREATE

Creating a job is the action that generates a new resource. We can add a POST operation on the `/jobs` path, the collection endpoint for the Job resources, to our OpenAPI description. It follows a very similar design to the "Register" action for the User schema.

### Listing 10.13 PetSitter OpenAPI Create Job

```
openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    post:
      summary: Create Job
      responses:
        '201':
          description: Created
          headers:
            Location:
              schema:
                type: string
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Job'
#...
```

## LIST MY OWN

To list jobs, we would use the Jobs collection endpoint, `/jobs`, with a GET operation. If we do that for "List my own", however, we will clash with another action for the Job model, "List all". As you learned about collection endpoints, they are typically used to list all instances of a resource, so this operation fits "List all" more than it does "List my own". What should we do, then?

We have already learned that we can use query parameters to implement filter criteria and use collection endpoints for searches. One option for the "List my own" action would be to use a filter parameter on the collection endpoint, so we can use `GET /jobs?user_id=` to fetch the user's jobs. There is another alternative, however, so it's time we talk about the third type of endpoint in CRUD APIs that I promised.

Directory structures, which we used as an analogy for paths in URLs, can be nested. You can create a folder within another folder. Now, imagine that you have a folder for users and not just a file for each user, but also a subfolder for the user into which you can save other files related to that user. In PetSitter, you can think of each pet owner having a folder with all the jobs they have ever posted. In our CRUD terminology, we call that approach a subresource collection endpoint. Its general structure is `/{{schema}s/{{id}}/{{subschema}s}`. For our specific case, the path is `/users/{{id}}/jobs`. These endpoints use CRUD methods in the same way as top-level collection endpoints, which means we use the `GET` verb.

For the response format, we follow the collection structure we introduced earlier. To do so, we create a `collection` object with an `items` field that is an `array`. All the items in that array are instances of our `Job` schema, which we link here with the `$ref` keyword.

### **Listing 10.14 PetSitter OpenAPI List User's Jobs**

```
openapi: 3.0.0
#...
paths:
  /users:
    #...
    /users/{id}:
      #...
      /users/{id}/jobs:
        parameters:
          - schema:
              type: integer
              name: id
              in: path
              required: true
        get:
          summary: List Jobs For User
          responses:
            '200':
              description: OK
              content:
                application/json:
                  schema:
                    type: object
                    properties:
                      items:
                        type: array
                        items:
                          $ref: '#/components/schemas/Job'
#...
```

**SIDE BAR** **Shouldn't we use a subresource collection endpoint for "Create" as well?**

From the analogy we used, if `/users/{id}/jobs` is the resource location for all jobs created by user, shouldn't we refactor our OpenAPI definition and use `POST /users/{id}/jobs` to create a new job instead of the shorter `POST /jobs` we used before? That is a valid concern. In general, however, when designing APIs with the CRUD approach, we try to keep the use of subresources to a minimum and always use the shortest path we can get away with. The only reason we're using the subresource here is that there is a clash with another action in a current user story. There's no similar clash or ambiguity for the "Create" action.

**VIEW**

Users can get details for a single job. That is a "Read" on the resource, so we can add a GET operation on the resource endpoint for jobs, `GET /jobs/{id}`.

**MODIFY**

Pet sitters can perform an "Update" on the jobs they posted, so we can add a PUT operation on the respective resource endpoint, `PUT /jobs/{id}`.

The last two operations did not have anything peculiar to them, they are very basic CRUD operations and they also follow the same format as the similar actions we have for users. Here is a listing of the two operations in the OpenAPI file:

## Listing 10.15 PetSitter OpenAPI Job Operations

```

openapi: 3.0.0
#...
paths:
  #...
  /jobs/{id}:
    parameters:
      - schema:
          type: integer
          name: id
          in: path
          required: true
    get:
      summary: View Job
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Job'
    put:
      summary: Modify Job
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Job'
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Job'

```

### LIST ALL

We already mentioned the "List all" action. And, as we used `GET /users/{id}/jobs` for "List my own", we're now free to use the collection endpoint for "List all", so we can add `GET /jobs` to our OpenAPI file:

## Listing 10.16 PetSitter OpenAPI List All Jobs

```
openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    post:
      #...
    get:
      summary: List All Jobs
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  items:
                    type: array
                    items:
                      $ref: '#/components/schemas/Job'
#...
```

As you may imagine, if PetSitter takes off, there could be a lot of jobs in its database. Adding parameters for pagination and filtering is a good idea, but we want to keep things simple for now, so we'll revisit that in a later chapter.

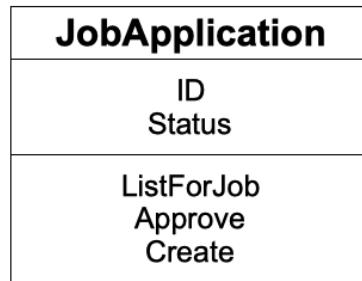
### SIDE BAR    What about "Delete" for Jobs?

Don't we have a "Delete" action in our domain model that is missing from the OpenAPI file? Yes! It appears you are paying more attention than frontend developer Max who accidentally skipped over it. Good that we have a review cycle. We will eventually notice and add the respective operation to our OpenAPI later in this book.

### 10.4.3 Job Application operations

In the last model that we created for PetSitter, we have the following actions:

- List for Job
- Approve
- Create



**Figure 10.7 PetSitter Job Application Model with Actions**

## LIST FOR JOB

When we look at this action, we may find that it is similar to the "List my own" action we have for jobs. We're working with resources from one schema that have a relationship with a resource from another schema. In "List my own", we had the jobs for a user. In "List for Job", we have the job applications for a job. Therefore, it is another case for a subresource collection endpoint. In our directory analogy, we can think of each job having a folder for all the applications it receives. That path is `/jobs/{id}/job-applications`. And, since this is a "Read" operation, we use the GET verb.

### Listing 10.17 PetSitter OpenAPI List Applications For Job

```

openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    #...
    /jobs/{id}/job-applications:
      parameters:
        - schema:
            type: integer
            name: id
            in: path
            required: true
      get:
        summary: List Applications For Job
        responses:
          '200':
            description: OK
            content:
              application/json:
                schema:
                  type: object
                  properties:
                    items:
                      type: array
                      items:
                        $ref: '#/components/schemas/JobApplication'
#...
  
```

## APPROVE

Pet owners approve applications to their jobs. The Job Application schema contains a `status` field that can be used to indicate whether the application is pending, accepted or denied. Therefore, the "Approve" action is a more specific version of a "Modify" action that changes a Job Application resource. To follow the CRUD approach, we can create the more generic operation with a `PUT` method on the `/job-applications/{id}` path. To perform the "Approve" action, the pet owner sends a request following the Job Application schema with the `status` field set to `approved`.

If there was another action like "Deny", which the PetSitter application will likely have down the line, but which we have not included in the current domain model, the same operation can be used. The lesson is that not every action in a domain model maps to exactly one operation, sometimes an operation can cover multiple actions that differentiate through the request parameters.

### Listing 10.18 PetSitter OpenAPI Approve Application For Job

```
openapi: 3.0.0
#...
paths:
  #...
  /job-applications/{id}:
    parameters:
      - schema:
          type: integer
          name: id
          in: path
          required: true
    put:
      summary: Modify Job Application
      requestBody:
        description: Update the application details
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/JobApplication'

      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/JobApplication'
    #...
```

## CREATE

Pet sitters apply to jobs by creating job applications. A "Create" requires a `POST` method on a collection endpoint. In this case, there are two options for the API design:

- We can use the collection endpoint for job applications, `/job-applications`. The

information about the job that the user applies to is included in the request body through the `job_id` field in the Job Application schema.

- We can use the subresource collection endpoint for applications for a specific job, `/jobs/{id}/job-applications`, which is the same that we used for the "List for Job" action. Users can then omit the `job_id` field from the request body because it is redundant information.

There are arguments in favor of both approaches, so there is no true right or wrong here. Our API designer Max decides to use the second option. His argument is that we do not use the `/job-applications` endpoint so far and by using `/jobs/{id}/job-applications` the "Create (for job)" action pairs well with the "List for Job", leading to more consistency within the API design.

### Listing 10.19 PetSitter OpenAPI Create Job Applications

```
openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    #...
    /jobs/{id}/job-applications:
      parameters:
        - schema:
            type: integer
            name: id
            in: path
            required: true
      get:
        #...
      post:
        summary: Create Job Application
        responses:
          '201':
            description: Created
            headers:
              Location:
                schema:
                  type: string
        requestBody:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/JobApplication'

  #...
```

Awesome, we made it through the whole domain model and created an OpenAPI file! The API definition that we have is a representation of that domain model. It should be possible to develop a frontend and a backend that communicate with each other through this API and, together, form the PetSitter application that fulfills all the requirements laid out in the user stories from the previous chapter. At the same time, there is probably still room for enhancements of the OpenAPI definition. And, if you have ever worked in a real world software development project, you have surely experienced that the initial design is never perfect and requires changes due to issues that arise within the development lifecycle.

Based on the plan we presented in chapter 9, frontend developer Max sends his OpenAPI file to backend developer Nidhi for feedback before implementation. In the next chapter, we introduce a workflow to handle changes to the API and look at a specific change of the API definition.

## 10.5 Summary

- Each concept from the domain model has an equivalent schema in the OpenAPI file. The attributes from the model become properties with a specific datatype. Relationships between concepts usually show up as a property in one schema pointing to the ID of the other schema. For example, a job points to its creator with the `creator_user_id` field. It is also possible to include one schema in another, which we did for dog and job.
- OpenAPI provides the `components` sections where we can define common, reusable schemas. In that way, we don't have duplicate schemas in different places. Instead, we can add references with the `$ref` keyword. For example, we can use the same `Job` schema for creating, viewing, listing, and modifying jobs.
- The CRUD approach provides common pattern for expressing the actions in the domain models as operations in an API. The four actions, Create-Retrieve-Update-Delete, map to HTTP methods. Each instance of a concept is called a "resource" and there are resource endpoints for each resource and collection endpoints for retrieving multiple resources with the same schema. There are also subresource endpoints for listing resources related to another resource.
- As we go through actions in the domain model, we strive to map them to CRUD and add the respective paths and operations to the OpenAPI file. Some actions correspond directly, such as "View" with "Read", or "Modify" with "Update". Sometimes we need to express actions differently, though. For example, we modeled "Approve Job Application" as an "Update" on the `Job Application` resource with a specific value for its `status` field.

# *Building a change workflow around API Design First*



## This chapter covers

- Identifying the critical issues around an API Design First approach.
- Setting up a workflow to solve those critical issues using GitHub.
- Walking through an example change to the API Definition.

Having defined an API definition the next logical step is to start building it. When we do so we'll be missing something critical that will cause us pain down the road. We have to consider how changes to the API definition will be communicated when we're not all in the same room. Changes that are the result of issues found during implementation or because of evolving business requirements. Before we start implementing the code, we should take the time to set up a change workflow so that we'll be able to adapt confidently as changes arise.

Building on from the Action plan found in chapter 9, we're currently within the draft/review cycle. Iterating on the API definition until we've concluded its design, so that the next step of implementing it, can begin.

Describing the API ahead of building it, ie: an API Design First approach—while hugely beneficial, comes with trade-offs that we need to be aware of and have an answer to.

We'll be looking at these trade-offs found in the API Design First approach, specifically around changes to the API definition and how to keep everyone on the same page.

At the end of this chapter we'll have put together a change workflow based on GitHub. To highlight the workflow, we'll be walking through a practical example in the PetSitter API.

**CAUTION** This change workflow is in place for changes that occur before a release of the application. This matters because we're not bumping versions or communicating these changes to the public. This will serve as the base for when we need to consider public and breaking changes to our API.

## 11.1 The problem

Let us consider what happens when a stakeholder discovers an issue with the API design. Any issue that could naturally affect one or more stakeholders, needs to be addressed—ideally, not by shouting across a room (if you're lucky enough to be in the same room!). Or when there are dependencies between stakeholders, and so one or more become blocked by the other which leads to wasting time—while they wait to get unblocked.

*We may have not use the word "stakeholder" before, in this context it refers to people who have a role to play in the project. This includes developers, architects, product managers, UX designers, etc. José has two developers (frontend and backend) and himself as the project's lead, making three stakeholders. It's really just a nice way of saying "people who are involved".*

Developing software (and so APIs) is fiendishly becoming more "artistic" when it comes to figuring out how to manage these implementation details, with new ideologies popping up all the time. Most problems and their solutions will stem from the organization's structure<sup>1</sup>. If you are an independent developer working completely on your own, then your problems will be significantly different from an organization that has thousands of developers and an active intern program. Workflows are like beverage choices: quite personal both to the situation and to the people involved. Despite that, we the authors, are strong believers that showing a concrete—if biased—workflow is far superior over the alternative and dreaded words... *"it depends"*, yuck! This doesn't mean you're off the hook, you'll still need to consider the nuances of your precise situation and tweak the workflow to suit, but we'll be here to help.

In API design and in its implementation there are some common points of interest across these disparate organizational structures. Specifically when more than one person is involved in the design and/or implementation of the API.

Firstly there is a need to find the source of truth for the API, the one place to always find the latest version of the API Definition. There is a need to make changes to the design of the API, when issues inevitably arise during implementation. Followed by the need to communicate those changes to other stakeholders and getting consensus on integrating those changes. And finally there is a need to know what has changed since you, the stakeholder, last viewed the API definition.

**NOTE**

These concerns are not really specific to API design and are more general software design issues. When we look at API design we do notice that by its very definition—Application Programming Interface—that it is an interface. Which to me really highlights the human boundary that exists. The boundary of a producer and a consumer of that piece of work. This is where the pains of developing software really start to make themselves known. But before I digress further, we are still focused on the API design. We'll leave the other interfaces (like UI, code libraries, etc) for another day.

When designing an API from scratch (and later adding to it) you'll naturally discover these concerns. When summed up, they look like this...

Challenges of implementing an API (with design first approach)...

- View the latest API definition
- Suggest and accept changes
- Compare changes to working copy



**Figure 11.1 Challenges**

What all these concerns need is a system with a rhythm to get stakeholders pushing towards the same goal.

In this chapter we're going to put together a workflow that is built around making changes to the API definition. Too much theory makes Jack a dull boy (or was it not enough playing?) and so in the last part of this chapter we'll walk through making a change to the PetSitter API, seeing how each of our three stakeholders are impacted by that change.

At the end of this chapter we'll have established that workflow using GitHub—which addresses the critical issue of communicating changes, found in an API Design First approach.

**NOTE**

As authors we need to narrow our focus to a single audience, so when you're wondering what size team this is focused for (despite the shared points of interest) you can wonder no more: In our heads we're thinking of a team with separate people doing the frontend and backend pieces of the application, as well as separate people coming up with the business requirements. This is José's team, including Nidhi and Max. If this doesn't match your team structure, you can think of these as "roles" that can be split up further (more fine grained roles and/or teams) or combined (a single person with multiple responsibilities). The net effect should be the same. This is about setting up a workflow to implement an API over time.

## 11.2 Communicating and reacting to change

At the heart of our workflow will be this idea that changes to the design are inevitable. The ones we're specifically interested in are those that arise during implementation, things we could not foresee at the time we designed the API. Although there are other valid reasons why a design should change after it has begun to be implemented (missing business requirements, as an example).

There is a balance to be made on how much time we spend on the design part and when we start to implement it. Too little on the design side and we'll end up making expensive changes that need to be reverted. *Too much* time on the design side and we risk waiting too long to implement. We cannot solve all design issues at design time, but they are the cheapest way to solve them.

So we can assume that there will be changes—how do we build a workflow around that?

Stakeholders will need to be able to highlight an issue in the design, suggest a change and get consensus on that change. As well as react to changes made by other stakeholders. Where these changes are based around the source of truth for the design (an important point!). Without a central source of truth we could be working on older, invalid features!

These are the important points that our workflow needs in order to be effective, let's expand on them a little more...

- **A single source of truth for the API definition.** Everyone needs to know what the latest and agreed upon version of the definition is. A lot of confusion occurs when there are multiple documents lying around (via emails, slack channels, PRs, etc) and that causes real delays. Find a single place in this world where your API definition lives and agree that it speaks the truth and is the whole truth.
- **A way to suggest a change and a way to agree to that change.** There are multiple stakeholders and each will have their own priorities and needs. Each needs to be able to suggest a change to the API definition and there needs to be an authority (or quorum, etc)

that accepts that change.

- **A way of viewing changes since you last saw the API definition.** As a stakeholder you may have an older idea of what the API was before it got changed and so you need a way to compare and see the differences between the API version you remember and the current API. If you're a developer this will give you requirements for the API changes you need to implement.

Let's stick them into a table so that we can address them and find solutions...

**Table 11.1 List of requirements for the workflow of API Design changes.**

Workflow issue	Solved by
A single source of truth for the API definition	?
A way to suggest a change and a way to agree to that change	?
A way of viewing changes since you last saw the API definition	?

As a software developer you probably already have a workflow that satisfies these needs in your codebase. All of the above can be reasonably solved by a version control system, like git. And as our first workflow for API design, that's exactly what we'll use. More specifically we'll make use of GitHub and its pull request feature, although you could use your favourite version control system instead and you should be just fine.

We've discussed the points of interest in our changes-are-at-the-heart-of-everything workflow, now let's see how we can use GitHub to solve for them.

**NOTE**

Is GitHub enough for API Design First approach? It certainly can be, there are more API-specific tools out there in the wild that offer more features for API design and definition management. While it's tempting to cover them here in this book, we've made an effort to keep it as agnostic as possible and focus on the root problem statements involved in OpenAPI and its uses. Since these tools will also change to adapt to the markets, but the problem statements will remain true for longer. Tools such as SwaggerHub, Stoplight, Apiary, etc.

### 11.3 GitHub as our workflow engine

GitHub is a great service for managing code, it's built on the awesome `git` version control system. There are two reasons we'd want to use GitHub (or BitBucket, GitLab, etc) instead of just `git`.

1. We want an obvious place where new/existing stakeholders can grab the latest version of the API definition.
2. We want a central place to suggest, review and merge changes.

This doesn't mean you can't use `git` on its own to achieve these goals. What it does mean is that you'd need to figure out another way of centralizing the system or at least making it accessible to

all stakeholders, which is exactly what GitHub does us.

We have three concerns we're interested in our API workflow, let's begin to solve them with GitHub...

### 11.3.1 A single source of truth for the API definition

This is an easy one, since it's a core part of GitHub. GitHub uses branches and we can pick one that will act as our "latest and greatest" version. Later on we can declare a feature branch be source of truth as we develop specific features, but ultimately merging them back into the "latest and greatest" version. For simplicity's sake we won't dive into what our branch name *could* be, we'll just say that the `main` branch holds the latest version of the API definition.

### 11.3.2 A way to suggest a change and agree on it

To suggest a change in our workflow system we'll say that you can modify any part of the API definition within your own git branch and use that to suggest the changes into the `main` git branch. If you have an issue, but no ideas on how to change the API - then you can simply raise a GitHub issue instead.

Agreeing on changes is a little harder. There are a few ways we can do it...

### 11.3.3 Agreeing on a change

1. Have a single authority in charge of the API and that person/team will sign-off every API change.
2. All stakeholders need to agree. This is good for small teams, but doesn't scale well for larger teams.
3. Accept votes on a change. Depending on the size of the team, perhaps just two votes are necessary to accept an API change.

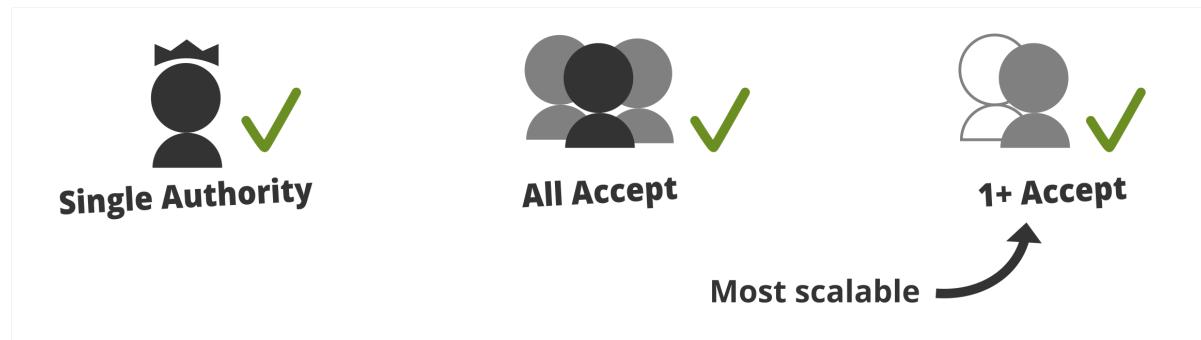


Figure 11.2 Different strategies for agreeing on changes

You'll need to decide which option (or alternative) suits your organization structure. An important point to stress is that autonomy equates to speed and more eyes on a subject reduces mistakes. Strike that balance! For José and his team we're going to say that at least one other stakeholder needs to agree. This keeps progress moving.

Regardless of who we give authority to—the mechanism is the same in GitHub, the pull request. These will be our steps in accepting a change...

## GITHUB PULL REQUESTS

1. Create a pull request from your branch to the `main` branch to show the suggested change.
2. Add reviewers to the pull request, each Reviewer can *Approve* the pull request.
3. Merging the pull request will accept the change and update the `main` branch.

For those stakeholders that weren't part of the conversation, they'll need a way of reviewing all the changes made since *they* last saw the API definition. This leads us to our next workflow item, a way of viewing changes.

### 11.3.4 A way of viewing changes (based on an older version)

This is the trickiest one so far. We're going to use a boring approach (apologies!) and rely on git's text diffing feature to show the differences between two API definitions.

I would have loved to show a specialized tool that compares OpenAPI definitions but alas none are stable enough to put into print. And while such a tool would be awesome and useful, we've found that diffing text files works well enough in practice.

Each stakeholder will need to know what version of the API definition they last saw, in order to later compare it to the latest version. In our GitHub workflow system we have two (perhaps many more) ways of achieving this.

We can either rely on checking for changes before doing a `git pull`(on the same branch), but this is a little risky. In git there are good reasons to encourage pulling as often as possible for unrelated reasons. So relying on stakeholders to compare API definitions before pulling is untenable, mostly because there is no explicit declaration of what API definition you're currently working on.

Alternatively we can make it explicit by creating a branch for each stakeholder and then instead of pulling they can *merge* in changes from the source-of-truth branch (ie: `main`). They'll only merge when they are ready and at that point in time they can compare to see if there are any relevant changes for them to react to.

So how can they view changes? Because they have their own branch, at any point they can use GitHub to compare their branch to the `main` branch—and see what changes have been made. As an added bonus, they can make changes to their branch as a way of suggesting changes to the main branch. This will work well for our purposes.

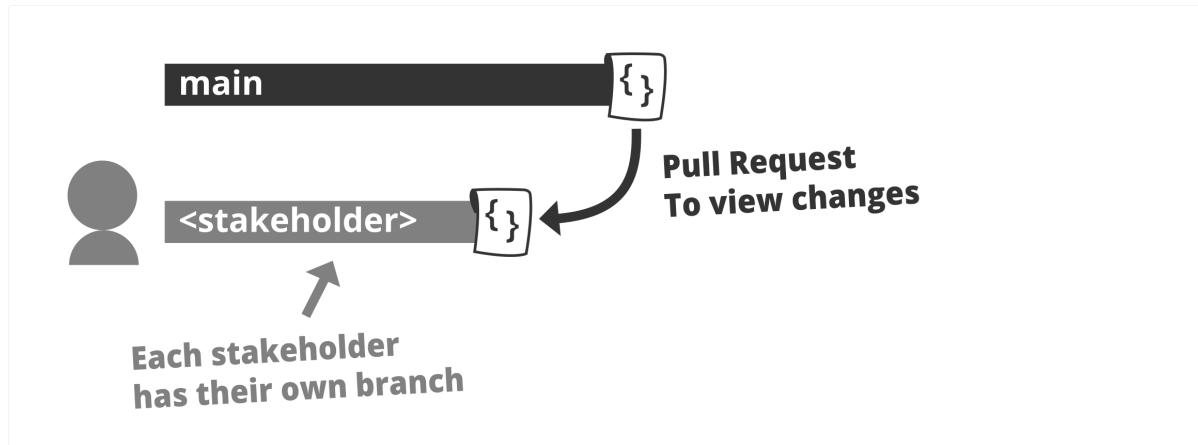


Figure 11.3 How to compare change from each stakeholders' perspective

## 11.4 Tying the GitHub workflow together.

Now that we've addressed each point in isolation, we need to tie them together into something coherent. We can list out steps for common actions (like suggesting a change) and show a friendly diagram that gives us a mental model of what's going on.

The first step is to set up our workflow and set up GitHub.

### 11.4.1 Setting up GitHub and the source of truth

We'll start with a new GitHub repository (aka "repo"). You don't have to follow along as we don't really want you to get lost in the weeds of code. But we're mentioning this here as a way of making it complete.

#### SETTING UP AN API REPOSITORY

Set up a GitHub repo, such that the following is true... 1. There is a `main` branch, set as the default branch. 2. There is a branch for each of the stakeholders, `frontend`, `backend` and `business`. 3. The `main` branch contains the file `petsitter.0.1.oas.yml` which has the contents of the PetSitter API we've been describing so far.

Here is an official GitHub guide which covers the above, <https://guides.github.com/activities/hello-world/>. In it you should see how to create a new GitHub repo, branches and your first commit. If you're already familiar with these concepts, feel free to merely gloss over it.

## NAMING CONVENTIONS.

Here is a naming convention that makes sense: `petsitter.0.1.oas.yml`, for the following reasons: 1. You obviously need to name the API, here it's named after the product, PetSitter. 2. Your API will have versions, pre-release there shouldn't be any version changes as the API should be considered fluid. After a release it becomes increasingly important and semantic versioning<sup>2</sup> is the best standard we have today. 3. Adding major/minor versions shows if there are added features (minor) or if the API has breaking changes (major). 4. Adding `.oas` gives a nice indicator of what type of document this is without looking inside! 5. It's a YAML file, so we'll need a `.yml` or `.yaml` extension.

For the version, we've explicitly left out the `patch` version (according to Semantic Versioning) since `patch` versions do not impact the semantics of the API, and we wouldn't want a file for each new tweak we make—unless it affects the semantics. If it is an additive change (ie: new operation added) we can bump the `minor` version, if it's a breaking change (ie: removal of a parameter) then we can bump the `major` version. The versioning of the API definition will be covered later, after we launch it, as it'll become increasingly important then.

### 11.4.2 Steps in our GitHub workflow

Each stakeholder needs a way of doing the following...

#### VIEWING THE LATEST API DEFINITION.

Visit the GitHub repo and view the `main` branch. In it will be the latest and greatest API definition.

#### SUGGESTING A NEW CHANGE.

1. Create a new branch or update your own branch with the change.
2. Create a pull request against the `main` branch. Add a description to the pull request showing the motivation behind the change.
3. Add your fellow stakeholders as reviewers.

#### REVIEWING AND ACCEPTING A CHANGE.

1. When you get notified of a pending pull request.
2. View the changes and add your feedback.
3. If you're satisfied then approve the pull request.
4. The authority (or the owner of the pull request) can then merge the change after enough reviewers have approved.

## COMPARING CHANGES TO THEIR WORKING COPY.

1. When the `main` branch has new changes in it.
2. Compare that to your own branch to see the changes since your last merge. Using Git(Hub) diffs.
3. Note down any tasks that you need to do based on the changes (if any).
4. Merge in `main` branch into your own branch to keep it up to date.

Revisiting our table from earlier we can fill in the blanks...

**Table 11.2 GitHub workflow solutions**

Workflow issue	Solved by
A single source of truth for the API definition	Nominate a branch, ie: <code>main</code>
A way to suggest a change and a way to agree to that change	Use pull requests
A way of viewing changes since you last saw the API definition	Each stakeholder maintains their own branch

Now that we've described how GitHub works let's get back into the world of API definitions! Given that we have a GitHub workflow (on paper at least) it's time to kick the tires and test out a design change to our API definition, PetSitter.

## 11.5 Practical look at the workflow

José's team have decided to adopt a more formal workflow for making design changes to their API definition. This makes sense, as even though their team is small—they still cannot meet for every small change to the API design and yet they need to be aware of each change. An awareness that won't impact their other responsibilities.

In some cases it'll be impossible to create a meeting yet the process must still continue. A formal workflow has become a must for them. Let's see how a single change makes its way into the design with our new GitHub workflow.

Nidhi (our favourite backend engineer) was contemplating backend designs for the application when she discovered that, while the Domain Model said there should be a way to delete jobs, the API definition that Max had shared was missing this operation. A simple oversight on his part. She knew this would be needed so she wanted to suggest this as a small change to the design. Here is her process...

### 11.5.1 Creating and suggesting `DELETE /jobs/{id}`

The change itself is quite small and Nidhi has no problem writing it directly into the definition on her branch. She doesn't anticipate that this will need a design session, nor does she need to do any research into it. This is a small change.

She whips out Swagger Editor (her favourite tool for this job, although any will do) and copies the definition into it. Adds her changes and commits the change into *her branch* on GitHub, ie: backend branch.

Her change...

### Listing 11.1 Added DELETE /job/{id} change

```
/jobs/{id}:
# ...
  delete:
    summary: Delete Job
    responses:
      '204':
        description: No Content
```

After committing her change she can now create a pull request to suggest this change to her colleagues. She adds in a simple motivational message behind her changes, and adds in the reviewers.

In GitHub that looks like the following...

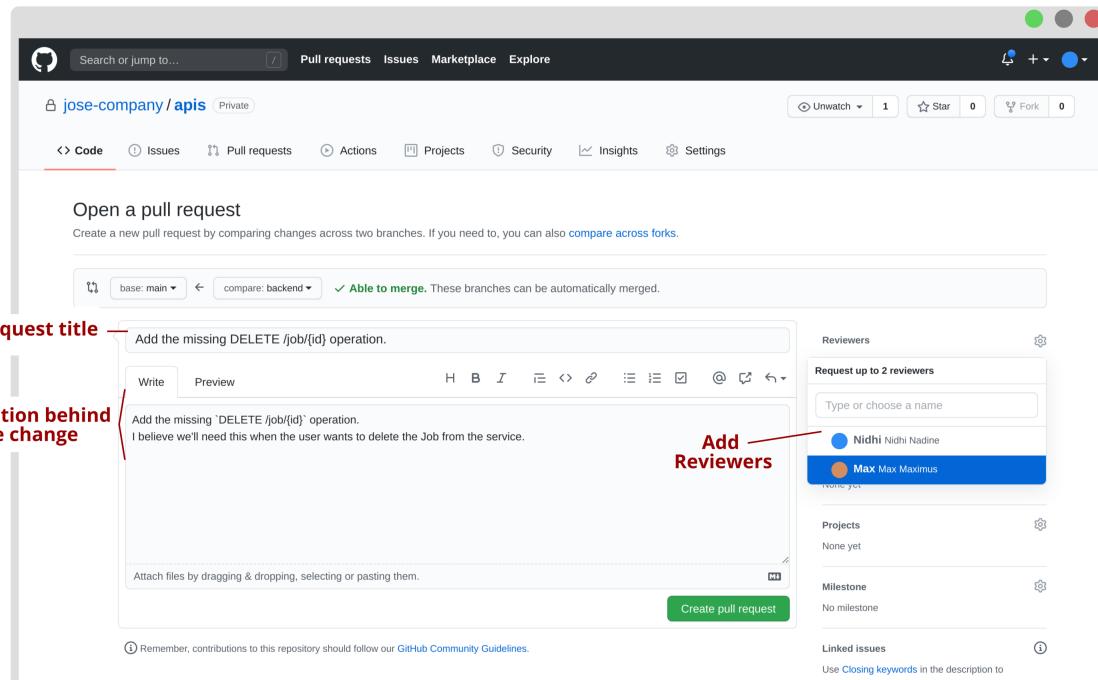


Figure 11.4 Creating a new pull request in GitHub

### 11.5.2 Reviewing and accepting changes

José is on holiday, which leaves Max as the only other stakeholder that can approve the change. After Nidhi creates the pull request, and depending on how Max has set up his notifications, he'll get notified almost immediately.

As soon as Max is ready he can look over and supply his feedback on the pull request. He's looking at two things...

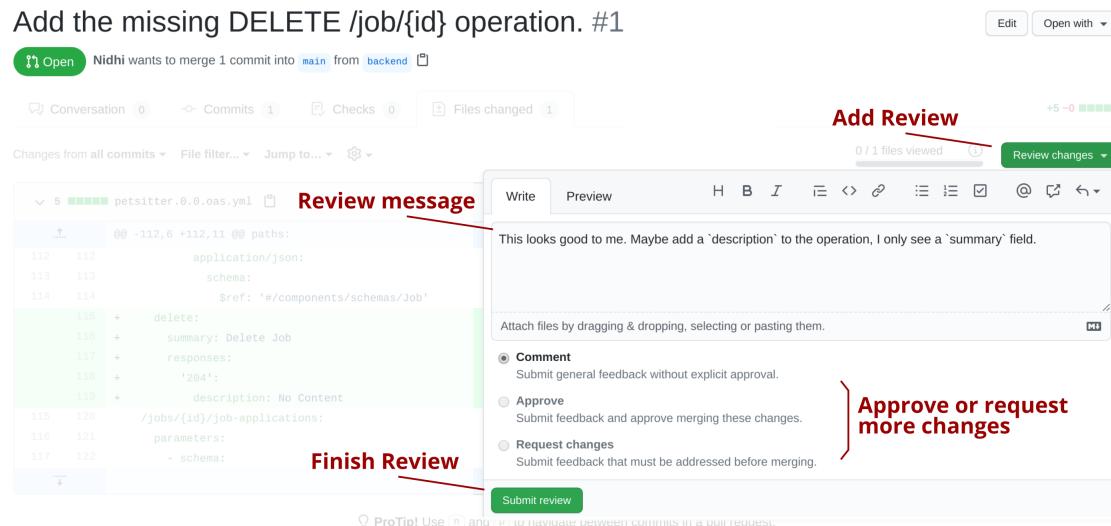
1. Is the motivation behind the change sound? Does this require research?
2. Does the change itself make sense?

Max thinks to himself, shucks, this is an oversight—I simply forgot to add it in. He doesn't need to research the change, since it's self-evident. Looking at the pull request, Max is clearly able to see the single change to the API definition and it looks consistent with the other DELETE operations. He's now satisfied with the pull request.

He mentally checks off the two questions as such...

1. The motivation makes sense, they will need to be able to delete Jobs in the future.
2. The change is valid and matches the motivation, a single operation was added and appears valid.

He has no problem approving the change, although he can't resist throwing in a little comment about something that caught his eye!



**Figure 11.5 Reviewing a pull request**

Nidhi notices that the description is missing too, and adds a new commit. Which will automatically update the pull request after she pushed the commit. The change can now be merged!

### 11.5.3 Comparing older branches to the latest

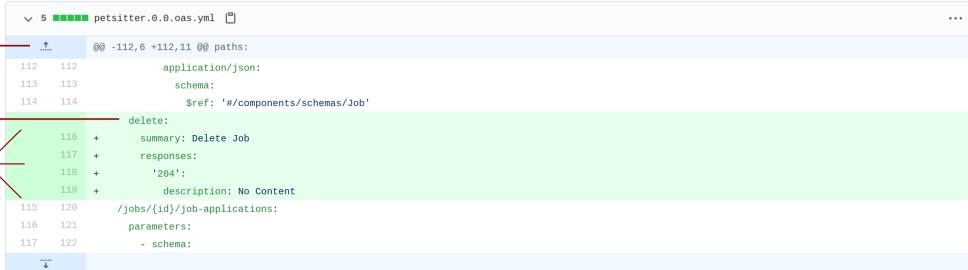
Time passes and José returns from his (well deserved) holiday. He now has the pressing need of catching up on the API design that's taken place. He has several options depending on what sort of information he's looking for. For instance, if he's just interested in what the latest API definition looks like today, he can go and visit the `main` branch and scan over it. Nothing further. But if he's looking to confirm *exactly* what's changed to see if any detail will impact him, he can compare using his branch (the `business` branch).

In this method he's interested in answer the following

1. What are the exact differences between when *I* last saw the API definition and the latest today
2. Do any of those changes impact my interest in the API? For a business stakeholder, it might be whether this will cost more money or allow more features in the future.

Instead of creating a pull request between '`<some-branch>`' and `main`, José is going to create a pull request between `main` and his branch `backend` (ie: the reverse, so that `main` will be merged into `backend`). This will allow him to consider the changes and when that's done to finally update his branch to the latest, by merging the pull request himself.

Given that Nidhi only added a single operation, the diff would look like this...



```

View more
New Operation
The diff

5 5 petsitter.0.0.oas.yml
  00 -112,6 +112,11 @0 paths:
  112 112
  113 113
  114 114
  application/json:
  schema:
    $ref: '#/components/schemas/Job'
  delete:
    summary: Delete Job
    responses:
      '204':
        description: No Content
  /jobs/{id}/job-applications:
  parameters:
    - schema:

```

Figure 11.6 GitHub diff of the API changes.

José sees only one change and it doesn't affect him. He can merge in the branch and carry on with tasks of the day, confident in what's happened to the API design in his absence.

### 11.5.4 What we've done

What we've accomplished so far is devised a workflow for API Design changes based on three critical points and using GitHub as a concrete way of implementing those design changes. There are more API specific workflow tools that we encourage you to consider, that reduce a lot of the manual effort—but of course they target different audiences and you'll need to consider how important those features of API design are, to your organization.

Hopefully the GitHub solution is enough to get you going, where you can discover the way in

which API design can be both incremental, asynchronous and generally smooth.

## 11.6 Summary

- API Design First is an approach to solving design issues as cheaply as possible, but there are critical points of interest that make the process run smoothly. How to find the source of truth for an API. How to make/suggest changes, review them and get them accepted. And how to compare changes made, from your perspective (ie: relative to when you last viewed the API design).
- GitHub can be used to manage the API Design workflow and we created a simple approach to solving the three critical issues of API Design First approach. The source of truth is a branch called `main`. Use pull requests to suggest, review and accept changes. Create a separate branch for each stakeholder and use pull requests to explicitly update them, each time you're interested in viewing exact changes and dealing with them.
- There are several ways to accept changes made to the design. Depending on the size of your team you'll either nominate someone a single authority and only they can accept changes. More often you'll allow changes to be accepted if one or more reviewers accept the change. The last approach is the most scalable.

# 12

## *Implementing frontend code and reacting to changes*

### **This chapter covers**

- Building the frontend against a mock server (Prism) based on OpenAPI.
- Identifying design issues found during implementation.
- Using OpenAPI examples to verify that API changes make sense.

One part of building components separate from each other, as in this case where we have a developer for the backend and a developer for the frontend. Is that they have a dependency on each other. The frontend will need to make API calls to the backend. Normally that would mean that the backend needs to be built first.

In this chapter, we're going to look at how to build the frontend without having the backend implemented. This will free us up to start straight away, instead of waiting for the backend. It'll also allow us to catch design issues sooner, while it's still cheap to build them into the backend.

To build the frontend however, we have several options to consider, mostly around the question: Where should we mock? We can do the following...

1. Mock the data on the view layer
2. Mock the data in central data store (think redux, mobx, rjxs, etc)
3. Set up a mock API server

The first is the quickest, but messiest. We'd need to be careful of where we've added mock data and when to remove it. It should only be used for very short lived tests.

Mocking data in the central data store is better since we have one place where it'll be mocked. This will allow us to also toggle it on/off in code when it comes time to integrating with the backend. The downside is that we still have mock code written in our source files.

The last option is to keep the frontend code clean by not polluting it at all with mock data and instead set up a mock server. This does have tradeoffs, but is the cleanest approach and requires the least (almost no) code changes when it comes time to switch to using the real backend.

One of those tradeoffs is that we can't write logic in our mock server, it'll only serve up data that we tell it to.

## 12.1 The Problem

To build the frontend without a backend, we're going to look at how to use a mock server. In particular one called Prism that will allow the frontend implementation to start. We'll also deal with the inevitable challenge of handling an API design change that was missed during its initial design. By updating our local API definition and using examples to test different scenarios and edge cases.

By the end of this we'll be suggesting a design change to our stakeholders based on needed requirements that we've verified will solve our problem.

At the core of this, we're going to look at a single change that is missing from the existing OpenAPI definition. One which we, as authors, discovered during the implementation of the demo site. Yes, we even make mistakes while designing toy APIs!

To get to where we need to go we're going to look at the following...

- Setting up a mock server
- Learning how to build against a mock server
- Identifying a missing operation
- Design a possible solution
- Verify that solution works for our use case

## 12.2 Setting up Prism

Prism is an open source mocking server that reads in an OpenAPI definition and serves up responses that fit the shape of the data. In other words it serves up what you've described in the OpenAPI definition. At least enough to adhere to the contract. It is simple and gaining in support.

### 12.2.1 Installing Prism

Prism is a Node.js command line (cli) tool, so you'll need to install Node.js (and npmjs which usually comes bundled with it) to use it.

## REQUIREMENTS

- Node.js v12+ <https://nodejs.org/en/download/>
- npmjs (comes bundled with Node.js)

### Listing 12.1 Install prism cli tool

```
npm install --global @stoplight/prism-cli
```

You should now be able to run `prism --help`, which should print out usage information. If that didn't work try restarting your terminal.

### 12.2.2 Verifying that prism works

To test out prism we'll need an OpenAPI definition, go ahead and grab the latest pet-sitter API from here: <https://app.swaggerhub.com/apis/ponelat/pet-sitter/ch12-start>. Save that file locally and call it `openapi.yaml`

Now you can spin up a prism server with the following...

### Listing 12.2 Spinning up prism

```
prism mock -p 8080 ./openapi.yaml
```

Voila! It'll stick around for as long as you need. When you're done with it you can `ctrl-c` (or `Cmd-c`) to exit it. As I'm sure you've guessed, the mock server will be running on port 8080.

**NOTE**

If it fails to start up, check the documentation to see if prism requires a newer version of Node.js or some other dependency (in case it changes after this book goes to print), <https://github.com/stoplightio/prism>.

To see if it's correctly serving up responses you can use your favourite API client to test. Postman from chapter 2 is useful, or you can simply use `curl` as follows

```
curl http://localhost:8080/jobs
```

You should see a JSON response similar to the following...

### Listing 12.3 200 Response from <http://localhost:8080/jobs>

```
{
  "items": [
    {
      "id": 0,
      "creator_user_id": 0,
      "start_time": "string",
      "end_time": "string",
      "activity": "string",
      "dog": {
        "name": "string",
        "age": 0,
        "breed": "string",
        "size": "string"
      }
    }
  ]
}
```

Now that we have a serviceable API we can hand this over to the frontend team!

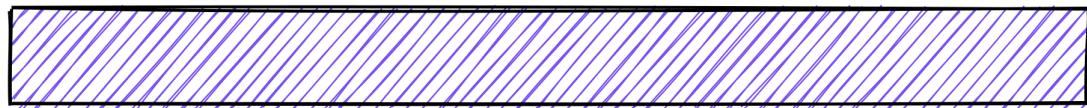
## 12.3 Building a frontend based on a mock server

To work with Prism you'll need to accept that you're dealing with static or canned responses, not a real API server. By real I mean that it doesn't have any logic. Logic such as storing data and responding with correctly formed numbers (eg: there will only be one item in an array). Instead we'll just get data that is in the correct shape. This gives us a crude, but as we'll see, workable API.

The focus should be handling the shape of the data and creating the pipelines from that API into your frontend app. That will include setting up your HTTP library, state management, view layer, etc. The mock API will work sufficiently to support a happy path<sup>3</sup> and it lets the team focus on what matters... building the frontend. For triggering errors, simulating little or large amounts of data—we'll need to get creative.

Let's take a look at building the following page of PetSitter API, we have the following UI design to work from (courtesy of José)...

# PetSitter



Job ID	Dog	Duration	Date	Activity	
#123	Fido	1 day	2020-01	walk	<span>APPLY</span>
#234	Rex	2 days	2020-03	sit	<span>APPLY</span>
#345	Blossom	1 day	2020-03	walk	<span>APPLY</span>

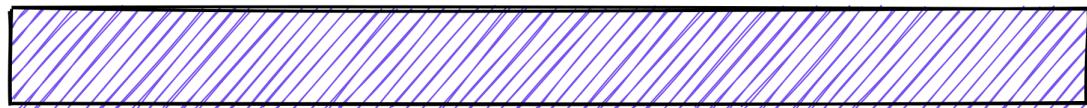
Figure 12.1 List of Jobs page in [petsitter.designapis.com](https://petsitter.designapis.com)

We can see that for each Job we have a row with some data and a button.

To populate that page, we need to fetch the list of Jobs from the API, ie: `GET /jobs`. Our frontend will naturally expect a 200 response with the data in the correct shape. And thankfully that's exactly what prism will deliver.

After pointing our code to use the mock server we may get something like the following...

# PetSitter



Job ID	Dog	Duration	Date	Activity	
string	string	string	string	string	<b>APPLY</b>

**Figure 12.2 List of Jobs page in [petsitter.designapis.com](https://petsitter.designapis.com) with boring data**

That looks... bare.

We can see that there is only one row of data and the contents of the fields are just `string`! If the frontend team got this far, it's certainly an achievement (the API is wired correctly, the page renders correctly, etc). But we're unable to test anything of substance. It would be nice if we could see more data and more realistic data at that!

By getting a little creative we can test more than "the frontend is wired correctly to the API". To test some of those things (we won't be able to test all the things a real API offers, we'll need to make some concessions given that it is a static server), we can make use of examples within the OpenAPI definition.

**NOTE**

A topic for another day is virtualization and how to fully simulate the business logic and request/responses of an API. This is an area that we believe could use some love and attention. While there are some products out there (ReadyAPI, Postman) none have captured our hearts to solve this problem. They often require more energy than it's worth to simulate an API.

Links to look up:

- ReadyAPI/ServiceV  
<https://smartbear.com/product/ready-api/servicev/overview/>
- Postman Mock Server <https://www.postman.com/features/mock-api/>

### 12.3.1 Add multiple examples into your OpenAPI definition

In chapter 6 we bumped into the `example` field for adding a bit of colour to our schemas, showing what real-world data can look like. In addition to `example` there is also `examples` (plural), which was introduced later into the OpenAPI specification for the obvious purpose of allowing *multiple* examples in a request/response/parameter/etc. Later in this chapter we're going to be making use of multiple examples in our mocking server, so let's learn a bit about how we can describe them.

The `examples` field can be found under the Media Type Object (ie: <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md#media-type-object>) which is found in `requestBodies` and `responses`. The shape of the `examples` field looks like this...

#### Listing 12.4 Shape of examples field

```
openapi: 3.0.0
#...
paths:
  /:
    get:
      responses:
        '200':
          content:
            application/json: ①
              schema:
                type: object
                properties:
                  name:
                    type: string
              examples: ②
                john-doe: ③
                  summary: Using John Doe as example data. ④
                  value: ⑤
                    name: John Doe
                some-other-example: #... ⑥
```

① The media type, `examples` comes under this.

- ② The `examples` field.
- ③ The name of the example, anything reasonable string.
- ④ The summary describing the particular example, optional.
- ⑤ The `value` field, which holds the example data.
- ⑥ We can describe as many examples as we need.

Tools (such as SwaggerUI) can show the example which often illustrates the shape of the data more readily than the schema rules. An example that would look like the following...

#### **Listing 12.5 Example in JSON**

```
{ "name": "John Doe" }
```

Now that we're armed with the ability to describe multiple examples, let's move on to using them in our mock server.

#### **12.3.2 Using examples in Prism**

Prism will generate a simple mock response for us based on the schema, using values like `string` for strings and `0` for numbers. Which is *okay* but limits how much we can test out our frontend. We can use examples to more clearly showcase data that might come from a real production server.

Let's add the following example into our definition to match the data in our UI design...

## Listing 12.6 First example for our mock server

```
openapi: 3.0.0
paths:
  #...
  /jobs:
    get:
      #...
      responses:
        '200':
          content:
            application/json:
              schema: #...
              examples:
                with-some-data:
                  summary: With some data
                  value:
                    items:
                      - id: 123
                        creator_user_id: 345
                        start_time: 2020-06-01T00:00:00Z
                        end_time: 2020-06-02T00:00:00Z
                        dog:
                          name: Fido
                          age: 3
                          breed: Doberman
                          size: medium
                          activity: walk
                      - id: 234
                        creator_user_id: 345
                        start_time: 2020-06-01T00:00:00Z
                        end_time: 2020-06-03T00:00:00Z
                        dog:
                          name: Rex
                          age: 2
                          breed: Rottweiler
                          size: large
                          activity: sit
                      - id: 234
                        creator_user_id: 345
                        start_time: 2020-06-01T00:00:00Z
                        end_time: 2020-06-02T00:00:00Z
                        dog:
                          name: Blossom
                          age: 2
                          breed: Rottweiler
                          size: large
                          activity: walk
```

Phew! That's a fair amount of data. When we execute a request against the mock server (if prism was running while you made these changes, then it'll automatically restart itself... nifty!), then we should see a response like the following (using `curl http://localhost:8080/jobs`)...

## Listing 12.7 Response after adding in an example

```
{"items": [{"id":123,"creator_user_id":345,"start_time":"2020-06-01T00:00:00Z","end_time":"2020-06-02T00:
```

It's not pretty, with the JSON having no whitespace, but we can see the data matches our example! Prism will pick the first example it finds instead of using it's own generated data. This is the key to us exploring our API, via examples!

## 12.4 Identifying a missing API operation

Max has the following UI mockup (courtesy again of José) that he is about to build...



Figure 12.3 UI Mockup of Job Applications page.

He sees that he needs a way of fetching all the Job Applications for the logged-in user. But after looking back at the API design (ie: the OpenAPI definition), he cannot find an operation that will do that for him. Even though implementation has begun he fortunately has a way of suggesting a new change (see chapter 11). Let's see how he'll achieve that...

Here is what Max is going to do:

- First, Max needs to clearly state what he needs for his task, a way of fetching all the Job Applications for the logged-in user.
- Then he needs to do his due diligence and make sure that he really does need a new operation.
- Max can then design the new operation.
- Before suggesting it, he'll need to test it out and make sure it really satisfies his need.
- Finally Max can confidently suggest the new API change and get it merged into the agreed upon design.

Max writes down what he needs: "A way to fetch the Job Applications for the logged-in user".

With that he set's about getting what he needs.

### 12.4.1 Due diligence for getting Job Applications for the logged-in user

That UI mockup shows all the Job Applications for the logged-in user. How could he fetch them?

- He could download the Jobs that belong to the user, with `GET /users/{id}/jobs`, then fetch their Job Applications with `GET /jobs/{id}/job-applications`. But that wouldn't fetch all the Job Applications, since most of them will be for Jobs that *don't belong* to the user.
- He could fetch *all the Jobs*, then fetch *all their Job Applications* and filter them by `userId` in memory... but that's a little ludicrous, it clearly won't scale well when the data becomes a little larger.

Neither of these seems viable. So Max feels a new operation is necessary.

### 12.4.2 Design the new operation

To fetch all the Job Applications that are specific to the logged-in user, Max can think of two different approaches...

1. `GET /job-applications?userId={userId}`
2. `GET /users/{userId}/job-applications`

The first is quite adequate, allowing us to add more query parameters to filter out the job applications based on other criteria. But we are already using the second pattern in `/jobs/{id}/job-applications` and having consistent patterns in our API is valuable (ie: the principle of least surprise). So Max decides to follow the existing pattern and go with `GET /users/{id}/job-applications`.

Starting from our API definition, we can add in the following sketch of an operation (ie: not a complete description of one). In addition to the operation, we'll make space for the example that we'll flesh out, in order to test whether the operation satisfies our needs...

### Listing 12.8 Adding a new operation for getting a user's job applications

```

openapi: 3.0.0
# ...
paths:
  /users/{id}/job-applications:
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: number
    get:
      description: Get a list of Job Applications for the given user.
      responses:
        '200':
          application/json:
            schema:
              type: object ①
            examples: ②
              two-items:
                summary: Two Job Applications
                value: # ... ③
              empty:
                summary: Zero Job Applications
                value: # ... ④
              many:
                summary: A lot of Job Applications
                value: # ... ⑤

```

- ① We're going to simply say it's an object, which will be valid. We'll flesh out the details later.
- ② Where we're going to add in our examples to test out the frontend.
- ③ We'll put an example of two Job Applications in here.
- ④ An example of zero Job Applications.
- ⑤ An example of many Job Applications.

Let's get some examples going.

#### 12.4.3 Example: Two Job Applications

### Listing 12.9 Basic happy case

```

# ...
examples:
  two-items:
    summary: Two Job Applications
    value:
      items: ①
        - id: 123 ②
          user_id: 123 ②
          job_id: 123 ②
          status: PENDING ③
        - id: 123
          user_id: 123
          job_id: 123
          status: COMPLETE

```

- ① Following the pattern we've established, we'll use an object to wrap "items", which will contain the list of Job Applications.
- ② Not going to bother with more realistic ID values, we don't know what they'll look like.
- ③ We've thrown in a Status to match the UI, the values are unknown so we'll start by guessing some.

This is the happy case, where we can see what our page would look like when it's populated with data.

#### 12.4.4 Example: No data

##### Listing 12.10 No Job Applications

```
# ...
examples:
empty:
  summary: Zero Job Applications
  value:
    items: []
```

#### 12.4.5 Example: Lots of data

Using <https://www.json-generator.com/> and <https://www.json2yaml.com/> we generated a bunch of data and stuck it in here: <https://gist.github.com/ponelat/7d9c0b296038f0875b5397759a2c6170>. The following is just the first few lines. We've used randomized values for all the `job_id` and `user_id`, but not for `id` (since that needs to be unique).

##### Listing 12.11 No Job Applications

```
# ...
examples:
many:
  summary: Many Job Applications
  value:
    items:
      - id: 0
        user_id: 358
        job_id: 4012
        status: COMPLETE
      - id: 1
        user_id: 3089
        job_id: 3902
        status: PENDING
      - id: 2
        user_id: 4040
        job_id: 5269
        status: PENDING
      - id: 3
        user_id: 5636
        job_id: 8420
        status: PENDING
    # ... total of 40 items...
```

## 12.5 Choosing which mock data response to get from prism

Using prism with a single mock example described works perfectly for us. But we want to be able to describe multiple examples and to choose which of those examples to use. To help us out, prism supports a simple method of sending options via the request header "Prefer". The two options we can focus on are `code` and `example`. With those two we'll be able to chose which response(`code`) and which example data prism will return.

Given the following API definition...

### Listing 12.12 Tiny API definition to showcase prism + prefer

```
# ./tiny.yaml
openapi: 3.0.0
info:
  title: Tiny API
  version: "1.0.0"
paths:
  /:
    get:
      description: Simple get
      responses:
        '200':
          description: Created
          content:
            application/json:
              examples:
                one:
                  value:
                    one: 1
                two:
                  value:
                    two: 2
            schema:
              type: object
        '404':
          description: Created
          content:
            application/json:
              examples:
                error:
                  value:
                    code: e404
```

And running prism in mock mode with `prism mock -p 8080 ./tiny.yaml`, the following curl requests will yield their respective responses.

### Listing 12.13 Prefer: code=404

```
curl -H "Prefer: code=404" http://localhost:8080/ ①
# Gives us...
{"code": "e404"}
```

① Note: `-H` is the curl flag for a request header.

**Listing 12.14 Prefer: code=200,example=one**

```
curl -H "Prefer: code=200,example=one" http://localhost:8080/
# Gives us...
{"one": 1}
```

**Listing 12.15 Prism prefer: code=200,example=two**

```
curl -H "Prefer: code=200,example=two" http://localhost:8080/
# Gives us...
{"two": 2}
```

This is a simple, but powerful enough for us to control which responses we want to get from within our code. In our frontend code we can set these headers during testing and remove them for when the code gets shipped into production.

For more info on response examples in prism:  
<https://meta.stoplight.io/docs/prism/docs/guides/01-mocking.md#response-examples>

## 12.6 Formalising and suggesting the change

After we've gone though our tests and sketches, we're ready to suggest a change to the API given that we're happy it will solve our problem. Let's tidy up our definition file and create a pull request in GitHub to suggest this change to the other stakeholders.

In our examples we used the following shape...

**Listing 12.16 Shape of list of Job Applications**

```
items:
- id: 123
  user_id: 123
  job_id: 123
  status: PENDING
- #...
```

Which is an object containing `items` and the list of Job Applications. Now we've already defined Job Applications in our schema, so we're going to use that and update our operation to the following...

### Listing 12.17 New operation ready for a pull request

```
openapi: 3.0.0
# ...
paths:
  /users/{userId}/applications:
    parameters:
      - name: userId
        in: path
        required: true
        schema:
          type: number
    get:
      operationId: getUserJobApplications
      summary: Job Applications associated with User
      description: Get a list of Job Applications for a given user
      responses:
        '200':
          application/json:
            schema:
              type: object
              properties:
                items: ①
                  type: array
                  items: ②
                    $ref: '#/components/schemas/JobApplication' ③
            examples:
              #...
```

- ① This is the property name not the OpenAPI keyword of the same name.
- ② This is the OAS keyword that describes the schema of the array items.
- ③ Reference our existing schema for Job Applications.

And that will do it! We can suggest a change that we're confident will now work, given that we've had the opportunity to test out different responses from within our frontend based on suggested design.

- Finding design issues during implementation is near inevitable. What's important is that there is a way to adapt to these issues, verify new solutions and fold those solutions back into the API design.

## 12.7 Summary

- Building a frontend based on a mock server is a great way to start implementation without having to wait for the backend to be built as both of those concerns can use the API contract (OpenAPI definition) as the guideline for what will be expected.
- Of the three types of mocking in the frontend that we touched upon, mocking in the view layer, mocking in the state layer and using a mock server. The last allows us to completely decouple our mocking concerns from our frontend code.
- Using the generated data from a mock server is limited, but can be greatly improved upon by using OpenAPI examples. To the point where certain scenarios can be tested by creating different examples (eg: no data, lots of data, typical data, etc). Error cases can also be tested in this fashion.
- We can use mock servers to test out new API changes that we're considering before suggesting them to other stakeholders, this expedites the process significantly as it allows us to bring solutions, not just questions to our colleagues and interested parties.
- Prism allows choosing which response and which example to return by looking at the `Prefer` header in the incoming request. `Prefer: code=404` will return the 404 response (if it is defined), `Prefer: example=one` will return the example named `one`. These options can be combined, eg: `Prefer: code=200,example=one`.

# *Building a Backend with Node.js and Swagger Codegen*

13

## This chapter covers

- Generating backend code with Swagger Codegen
- Optimizing an OpenAPI definition for code generation
- Designing a Mongoose/MongoDB database based on the domain model
- Implementing a basic API operation in Node.js

José's PetSitter team has created an OpenAPI definition for the API that connects the frontend with the backend. They have now reached the stage of the project where both developers are confident that the API is solid enough to start working on the implementation. In this chapter, we'll join Nidhi, the backend developer, as she builds a backend that exposes the API that Max designed in chapter 10 and that she herself reviewed in chapter 11. While we'll touch upon backend functionality like database persistence, our primary focus will be the process of going from API to code.

Our chosen backend technology is Node.js. Node.js is the server-side version of JavaScript. Of course there are a lot of other programming languages to choose from, such as Python, Ruby, Go (which we used for the FarmStall API in part 1), or PHP. We picked JavaScript because we expect most developers to have at least a basic grasp of the language syntax, even if it's just from the client-side.

We'll talk about the problem of keeping API definition and backend in sync and then introduce Swagger Codegen. We'll feed the results from chapter 10 into Codegen, evaluate the backend architecture, make some adjustments to the OpenAPI, regenerate the code, and then finally extend it to bring the backend to life. In the process, we'll also look into testing our API with curl and Postman.

In José's team, Nidhi and Max work independently on frontend and backend, hence the development we described in the previous and in this chapter happens in parallel. It also means Nidhi starts from the OpenAPI file after chapter 11 which does not yet include the additional endpoint introduced in chapter 12.

## 13.1 The problem

Every piece of software that is out there in the world is an attempt at building a solution for a business problem (unless, of course, a developer wrote code just because they were bored or wanted to show their skills). Sometimes, however, a developer goes astray. They're writing code and their code executes and does something, but it's not what the business needs. I'm not talking about bugs, I'm talking about a program that doesn't match the requirements laid out in the user stories.

Take the example of a very basic script that calculates prices with taxes. The business requirement was that the user enters a net price and the software adds sales tax and shows the final price. The developer, however, built a software, that takes the final price and splits it into net price and sales tax. The software works and calculates the right prices but it doesn't provide the calculations that the business really needs.

In PetSitter, we started by looking at *jobs* and *dogs* as two concepts in the domain model, but while writing user stories we realized that dogs always appear as part of a job description and we don't need specific actions to create, view, modify, or delete dog resources. Our backend developer should not create these operations, but they should make sure that every job sent to the API contains a dog.

Our goal is to build a backend for a product that follows the API Design First approach. Therefore, we need to find a path that takes us from the OpenAPI definition to running backend. And, ideally, that path is one that actively helps us stay on track and build the right API operations.

## 13.2 Introducing Swagger Codegen

Swagger is a set of open source tools for working with OpenAPI definitions. We've covered Swagger UI in chapters 1 and 8, and we've continuously used Swagger Editor throughout this book to write and validate our OpenAPI specifications. Let's have a look at another tool in this belt, Swagger Codegen, which we'll just call Codegen from now on.

The name is an abbreviation for "code generation" and it's a descriptive name that tells us what the tool does. It takes an OpenAPI file as its input and then generates code in various programming languages as its output. There are two primary features in Codegen, *client* code generation and *server* code generation.

### 13.2.1 Client code generation

Client code generation is, in fact, SDK (software development kit) generation. In the context of APIs, an SDK is a library that wraps an API so that developers creating an application that integrates the API don't have to build their API calls as HTTP requests. Instead, developers can call an SDK method that almost looks like a method in the standard library of their programming language, and the SDK converts that into an HTTP request. Codegen automates writing the code that does this conversion. You give it your OpenAPI file and it will provide you with a complete library to integrate in your application for interacting with the API.

In PetSitter, frontend developer Max wanted to use an autogenerated SDK for the PetSitter API while building his frontend. However, he was unable to find a suitable template. While Swagger Codegen has support for a lot of languages, most will at best be 80% of what the programmer really wants. This is a reality of code generation and customization will become more important as the project grows. Max will need to look at how to create his own template in the future, in order to catch API changes directly in code.

### 13.2.2 Server code generation

We can describe server code generation as server *stub* generation. In programming, a stub is an incomplete method. It already has the interface of the final method but it doesn't yet perform the full functionality. Instead, it returns "mock" or "dummy" data. For example, imagine a method for getting the details of a job application in PetSitter. The method takes an ID as its parameter, makes a database request to fetch job application details, checks whether the user is allowed to view the application (e.g, PetSitter only allows the creator of the job and the pet sitter who applied to it view the application), and finally converts the format of the details and returns it. A stub would take the ID but not consult the database and, instead, return an example of how a job application looks like. You can also think of stubs as fill-in-the-blanks for developers.

Unlike client code generation, server code generation with Codegen doesn't generate a library but rather a draft version of the structure of a server implementation with lots of blanks to fill. A starting point for developers, or a template, if you will. PetSitter backend developer Nidhi decides that it is a good way for her to start coding her part.

### 13.2.3 Swagger Generator

Codegen is an open source software tool that you can download from GitHub and run locally on the command line or integrate into a process. If you build a super secret stealth API, we recommend doing that. There is, however, an easier approach to get started. Smartbear Software, the maintainers of the Swagger toolchain, offer a hosted version - Swagger Generator - that you can access via an API at <https://generator.swagger.io/>. And, to make it even more accessible to developers, Swagger Editor integrates Swagger Generator. Thanks to that integration, you can directly trigger generating the backend in Swagger Editor and download a ZIP file with your generated code in your browser.

## 13.3 The backend architecture

Our first step is to take the OpenAPI file we built in the previous chapter, just throw it at Codegen, and then have a look at what we get and how we can work with that. We'll walk through the generated code to learn the architecture of the backend that Codegen prepared for us.

### 13.3.1 Generating the backend

To start, follow these steps:

- Open your OpenAPI file in Swagger Editor.
- Click on **Generate Server** in the menu bar. Swagger Editor shows you the backend technologies for which it can build code for you.
- Click on **nodejs-server**. Within seconds, your browser prompts you to download a ZIP file.
- Save the file on your drive.
- Extract the ZIP file into a directory.
- Open the directory in a code editor or IDE, such as Visual Studio Code<sup>4</sup>.

**NOTE**

We recommend using Visual Studio Code, or a similar code editor. Of course you could use vim or Notepad, too, but a multi-file code editor or IDE provides a better overview over both the directory structure of the generated backend.

### 13.3.2 Investigating the architecture

When you open the generated code directory, you'll find the following subdirectories and files:

- The `.swagger-codegen` directory contains a file `VERSION` indicating Codegen version that created the project.
- The `api` directory contains a file `openapi.yaml`. If you open the file, you can see that this is the OpenAPI file that you used as input, but with some modifications. We'll take a look at these modifications in a moment.
- The `controllers` directory contains a file `Default.js`. Inside the file, you'll see a list of functions with names based on the paths in the API definition. For example, `GET /jobs` has become `jobsGET`. And in each function, there's a block of code that calls a

function with the same name on a `Default` object, which, as you can see in the head of the file, comes from the `service` directory.

- The `service` directory contains a file `DefaultService.js`. It contains the functions that the controller functions in `Default.js` call. So, for each API operation, there is a controller function and a service function, and both have the same name. We'll explain this architecture further down. Each of the service functions contains some code that defines mock responses for the respective API operation. These are Javascript objects based on the schemas in our OpenAPI definition. For example, in `jobsGET`, there is an object that follows the `Job` schema.
- The `utils` directory contains a file `writer.js` that defines helper functions for generating API responses.
- The `index.js` file is the entrypoint of the application. You can see in the code that it references the `controllers` directory, the `api/openapi.yaml` file and a library called `oas3-tools`.
- The `package.json` file is the configuration of a Node.js application. It contains a few third-party dependencies. The most important of it is the `oas3-tools` mentioned before, which is a library that includes a set of helper functions for OpenAPI. Broadly speaking, it helps Node.js "understand" OpenAPI.

**NOTE**

**It's possible that future versions of Codegen follow a slightly different structure or create some additional files. Don't worry if the output you're getting does not exactly look like ours, and try to investigate whatever looks similar.**

From the quick glance at the generated code we have seen that the backend uses an application structure with controllers and services. It is a common architectural pattern that you can see in a lot of web application frameworks:

- Client-side requests arrive at the entrypoint of the application, which is `index.js` in this case. The entrypoint dispatches the request to controllers.
- Every path and method on the server, or every API operation, has its own controller function that handles requests and generates responses. For the generated code, controllers are functions in files in the `controllers` directory.
- The functionality of the application resides in services that the controllers can call as needed. For the generated code, services are functions in files in the `service` directory.

If you worked with other frameworks such as Express in Node.js or, for example, Laravel and Symfony in the PHP world, you have probably seen that you need a definition of routes that map paths and methods to a controller function. There is no such mapping here, but there is a reference to `openapi.yaml` in `index.js`. So, in fact, the OpenAPI file not only drives the code generation process, it becomes part of the running application itself and is responsible for the mapping. Let's find out how that works.

### 13.3.3 OpenAPI changes

Open `openapi.yaml` in the `api` directory and have a look at the first path definition. There are two new keywords that we should pay attention to:

- There is an `operationId` for each operation. For the first (with `summary` set to "Register User") it is `usersPOST`, a name that Codegen selected by combining the path (`/users`) with the HTTP method (`POST`). If you look at `controllers/Default.js` and `service/DefaultService.js`, you'll notice that this is the function name for the controller that handles the API operation.
- There is also `x-swagger-router-controller` and it is set to `Default`. As you've seen before, `Default` is the name for the controller and the service that contains the implementation of this method.

The combination of these two keywords, `x-swagger-router-controller` and `operationId`, connects the definition in OpenAPI with the implementation that is behind the respective operation.

We didn't have these keywords in our original OpenAPI file, so Codegen put everything inside the `Default` controller and generated an `operationId` before building the backend. If we don't like that, we can refactor it. At this point, we would have to make the changes in both the OpenAPI file and in the controllers and services, so we'll do something else. We'll update our OpenAPI file in Swagger Editor and regenerate the backend.

Another change that you can see is that Codegen added the `servers` element to your OpenAPI file. It adds a single server with the relative `url` value `/`.

## 13.4 Updating OpenAPI for the backend

As we've learned in the previous section, Codegen creates code based on the OpenAPI definition, filling in some defaults for OpenAPI keywords that we didn't use. We can leverage these keywords to assist Codegen in creating a better backend architecture and, in the process, create a more well-rounded OpenAPI file for other parts of the API lifecycle as well.

### 13.4.1 Adding operation IDs

An operation ID is a unique identifier for an operation. Hence, you cannot use the same ID more than once in your API definition. Apart from defining its uniqueness, the OpenAPI specification also says that tools and libraries may use this identifier for internal purposes and therefore recommends that names should follow common programming naming conventions.

Codegen uses operation IDs for function names in the `Node.js` code it generates, so we should provide operation IDs that sound like function names. It means we use only alphanumeric characters, no spaces, and start with a lower case letter. To connect multiple words, we could use either snake case (i.e., `register_user`) or camel case (i.e., `registerUser`). The latter decision

is up to the developer's preference, but should be held consistently throughout the API and, ideally, through all APIs that a developer or a company creates. Nidhi decides to use camel case for PetSitter.

**NOTE**

Of course, every programming language has slightly different conventions for function names. As we're working with Node.js we can look at JavaScript conventions, but we should keep in mind that the same OpenAPI file could drive frontends and backends in different languages.

To create good operation IDs that look and feel consistent, you can extend the basic conventions we already mentioned with an additional set of rules for generating the names. Here are the rules that we use:

- Start the operation ID with the action name from the domain model, followed by the schema name - singular for resource endpoints and plural for collection endpoints.
- Suffix the name for resource endpoints with `WithId` to have a reference the path parameter.
- For subresource endpoints, use the following structure: action name, subresource name, `For`, parent resource name (leave out `WithId` to shorten the name).

The table shows the operation IDs that we should create for all the operations in our PetSitter API, based on the rules we defined.

**Table 13.1 List of Operation IDs**

Method	Path	Operation ID
POST	/users	registerUser
GET	/users/{id}	viewUserWithId
PUT	/users/{id}	modifyUserWithId
DELETE	/users/{id}	deleteUserWithId
POST	/jobs	createJob
GET	/jobs	listAllJobs
GET	/jobs/{id}	viewJobWithId
PUT	/jobs/{id}	modifyJobWithId
DELETE	/jobs/{id}	deleteJobWithId
GET	/jobs/{id}/job-applications	viewApplicationsForJob
POST	/jobs/{id}/job-applications	createJobApplication
GET	/users/{id}/jobs	listJobsForUser
PUT	/job-applications/{id}	modifyJobApplicationWithId

You can specify your operation IDs by adding an attribute with the `operationId` keyword to your operation's definition, as shown in the following example for `registerUser`:

### Listing 13.1 PetSitter OpenAPI Register User, with ID

```
openapi: 3.0.0
#...
paths:
  /users:
    post:
      summary: Register User
      operationId: registerUser
      responses:
        '201':
          description: Created
          headers:
            Location:
          schema:
            type: string
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
#...
```

#### 13.4.2 Tagging API operations

When you design a software architecture, you should be careful to not put too much in a single file and rather break down your code into more manageable units. That way you have a clearer structure and better overview over the way your code is organized. Unless specified otherwise, Codegen puts all controllers and services in a single file, aptly named `Default`. We've seen that there's a `x-swagger-router-controller` keyword and, as you may guess from the name's `x`-prefix, it is not part of the OpenAPI standard and rather something that is specific to Swagger. While we *could* use that keyword to structure our API, let's look at a more standard way to do this - tags.

Tags are an OpenAPI feature that we previously used in chapter 8 to document the FarmStall API. Tools and libraries can use tags in different ways. In the Swagger toolchain, tags have the following effect:

- As we saw in chapter 8, Swagger UI shows subheaders in your API documentation that split the list of operations, i.e., your API reference, in multiple parts. It makes it easier for the reader to see which operations belong together and understand the purpose of the API in terms of these groups. If you assign multiple tags to the same operation, it appears multiple times in the UI.
- Codegen uses tags to create controllers and services. Every individual API operation has one designated controller file which contains the function with its implementation, so the router knows which function to call for a certain route and method. Unlike documentation, it makes no sense to duplicate code. Codegen only uses the first tag, and, to avoid ambiguity, it still creates the `x-swagger-router-controller` attribute. Hence, tags can be changed, e.g., for documentation purposes, without refactoring the code.

Our general recommendation is to use tags early in the API design process for both code generation and documentation. If, at any point, tags need to change, you can still make a decision

about either refactoring your code to follow the new tags or rely on `x-swagger-router-controller` to maintain the old structure.

## CHOOSING TAGS

To choose appropriate tags for your API definition, you can look at your domain model and the URLs for your operations. In many cases you can use the concepts in your domain model, or the first segment of the URL path, as tags. When applying this approach to PetSitter, we would, for example, have a tag named "Users" that includes all the operations for users which, thanks to our design approach from chapter 10, have a path starting with `/users`. For a good overview in Swagger UI and manageable code files in Codegen, you should aim to have around four to eight operations under each tag.

The PetSitter domain model has four concepts, *user*, *job*, *dog* and *job application*. Let's look at their actions and the respective API operations again:

- There are no specific actions for dogs, so we have no dog-related operations in our API and don't need a "Dog" tag.
- We have five user-specific actions (four operations in the API, as there is no representation of "Login"), so including a dedicated "Users" tag makes sense.
- We also have six job-specific actions and operations, so we should add a "Jobs" tag.
- Finally, we have just three operations related to job applications. One of them requires the `/job-application` prefix whereas the others are specific to a user or a job, so they use a subresource endpoint under `/users` or `/jobs`. If we consider these factors, we may not need a "JobApplication" tag but can distribute those under "Users" or "Jobs" instead.

We add our tag definitions for "Users" and "Jobs" in our OpenAPI file like this:

### Listing 13.2 PetSitter OpenAPI Tags

```
openapi: 3.0.0
info:
  title: PetSitter API
  version: "0.1"
tags:
  - name: Users
    description: User-related operations
  - name: Jobs
    description: Job-related operations
paths:
#...
components:
#...
```

## ASSIGNING TAGS

We decided we want to group our operations under the "Users" and "Jobs" tags. All operations on paths starting with `/users` go under the "Users" tag whereas operations on paths starting with `/jobs` and `/job-applications` go under the "Jobs" tag. You can refer to the table for a full list.

**Table 13.2 List of Operation IDs**

Method	Path	Tags
<b>POST</b>	/users	<b>Users</b>
<b>GET</b>	/users/{id}	<b>Users</b>
<b>PUT</b>	/users/{id}	<b>Users</b>
<b>DELETE</b>	/users/{id}	<b>Users</b>
<b>POST</b>	/jobs	<b>Jobs</b>
<b>GET</b>	/jobs	<b>Jobs</b>
<b>GET</b>	/jobs/{id}	<b>Jobs</b>
<b>PUT</b>	/jobs/{id}	<b>Jobs</b>
<b>DELETE</b>	/jobs/{id}	<b>Jobs</b>
<b>GET</b>	/jobs/{id}/job-applications	<b>Jobs</b>
<b>POST</b>	/jobs/{id}/job-applications	<b>Jobs</b>
<b>GET</b>	/users/{id}/jobs	<b>Users</b>
<b>PUT</b>	/job-applications/{id}	<b>Jobs</b>

The following example shows the assignment of the "Users" tag to the `registerUser` operation:

### Listing 13.3 PetSitter OpenAPI Register User, tagged

```
openapi: 3.0.0
#...
paths:
  /users:
    post:
      tags:
        - Users
      summary: Register User
      operationId: registerUser
      responses:
        '201':
          description: Created
          headers:
            Location:
              schema:
                type: string
        requestBody:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
#...
```

#### 13.4.3 Regenerating the backend stubs

After adding the tags and operation IDs in Swagger Editor, it's time to generate the backend again. Remember, you can build your backend directly from Swagger Editor by clicking on **Generate Server** in the menu bar, selecting **nodejs-server** and downloading the generated ZIP file in your browser.

If you open the new backend directory in your code editor or IDE, you'll spot the following

differences:

- The `controllers` directory contains two files, `Jobs.js` and `Users.js`, named after your tags. Likewise, the `service` directory contains two files, `JobsService.js` and `UsersService.js`.
- The files contain controller and service functions for the respective operations that you tagged. These functions now match the operation IDs. For example, `jobsPOST()` has become `createJob()`.

Now that feels much better to work with, doesn't it? Let's try and run the backend.

## 13.5 Running and testing the backend

Our next step is to run the backend, play around and see what Codegen prepared for us. As a prerequisite, you need to have Node.js and its package manager, `npm`, installed on your computer. If you tried out prism in chapter 12 you are already good to go, as it required the same dependencies.

To run the backend, open the console or terminal and change the working directory to the one containing the generated Node.js project from Codegen. Then, type the following command:

```
npm start
```

If, by any chance, you see `npm ERR! Invalid version: "0.1"`, edit the `package.json` file and change the version from `0.1` to `0.1.0`. Node.js requires semantic versioning but, as we explained in chapter 11, we want to skip the patch version for OpenAPI. After changing the file, run the command again.

Once the command runs successfully, you see a list of operations in your console. You also see a URL that hosts an instance of Swagger UI that is part of the generated project. Open that URL<sup>5</sup> in your browser and you'll see your API operations, just as you did in Swagger Editor. Let's try one:

- Click on the "View User" operation to expand it.
- Click on **Try it out**.
- Enter some random value for ID.
- Click **Execute**.

You see the response body which contains dummy data but follows the User schema. To assure yourself that this was a real API call served from your backend, you can also copy the `curl` command shown in Swagger UI and run it in a new console tab or window. You should get the same output.

### 13.5.1 Testing with Postman

We can also try out the API in a different tool that we got to know in chapter 6, the Postman application. To save some work building API requests in Postman, we can import the OpenAPI file:

- Start Postman.
- Click the **Import** button.
- Select your OpenAPI file. Use the one from the `api` directory of the generated backend, not your original file.
- Under *Import as*, you should see *Collection* and *API* selected. Click on it and remove the checkbox in front of *API* so that only *Collection* is selected. We are not using Postman's full API management capabilities so having a collection of the methods is sufficient.
- If you see the option *Folder organization*, change it from *Paths* to *Tags* to have your API operations organized in the same way as in Swagger UI.
- Click **Import** again to confirm.

You can now go to the *Collections* tab, open *PetSitter API* and browse all your API operations. Before we can try them, we have to tell Postman where to find the backend. We've previously seen that Codegen put a relative URL `(/)` in the `servers` element, which is not sufficient for an external tool like Postman. Follow these steps:

- Click the three dots that appear when you hover over the collection name.
- Click **Edit**.
- Go to the *Variables* tab.
- For the `baseUrl` variable, change both initial and current value from `/` to `http://localhost:8080`.
- Click **Update** to save changes.

Now, finally, we can make an API request with Postman. Try an operation with a GET method first, for example "List All Jobs". Open the operation and click the **Send** button in the request tab. If everything is OK, you should get a sample JSON structure similar to the one you got in Swagger UI and curl.

### 13.5.2 Testing input validation

So far we tested only GET operations, so why not turn things up a notch and try a POST operation. You can do this with Swagger UI or Postman, as you wish. Make a request to the "Register User" operation, `POST /users`, and provide the following request body:

```
{
  "full_name": "John Doe",
  "roles": "PetSitter",
  "email": "john.doe@designapis.com"
}
```

Now, something's not quite right here. While designing our API, we made the `roles` attribute an

array so that users can have different roles. In other words, the user resource is invalid. And here's something awesome: Your Codegen-generated backend code includes input validation, so it should catch that mistake. And indeed, when you send the request, you'll get a response with a 400 ("Bad request") status code and the message "request.body.roles should be array".

Let's fix the request body, so that we get a 200 ("OK") response back:

```
{
  "full_name": "John Doe",
  "roles": [ "PetSitter" ],
  "email": "john.doe@designapis.com"
}
```

**NOTE**

According to our definition, we should get 201 ("Created") instead of 200 ("OK"). This is one of the few things that Codegen doesn't yet do, so we have to do this manually. We won't cover this part of the code in this book, but you can look at the source code over at <https://designapis.com/>.

If you look in your service or controller files, you will not see any autogenerated input validation code from Codegen. Instead, the `oas3-tools` library, which is responsible for mapping request paths to operations based on your OpenAPI file, also takes care of input validation. Less custom code for routine work such as input validation keeps your implementation clearer and more organized, and it saves development time. Yet another bonus for API Design First!

So far, your OpenAPI file only specifies datatypes like `integer`, `string`, and `array`. To take even more advantage of input validation you can add additional constraints in your API definition. We'll discuss some of them in chapter 20.

### 13.5.3 Output validation with prism

Input validation is an extremely important feature because any input towards a software system is never trustworthy as it is outside the developer's influence. For an API, however, output validation is equally important. The OpenAPI definition is a contract that specifies inputs and outputs, and API consumers want to rely on the fact that the data that the API returns fulfills the contract. Sadly, Codegen or `oas3-tools` do not provide this automatically.

One approach towards output validation during development that doesn't require any code changes is using `prism`, the tool you already got to know in chapter 12 as a mocking tool. Frontend developer Max used it to test his frontend against an API that fulfills the contract without having any access to a running backend yet. Apart from using it for mocking purposes, you can also run `prism` in a proxy mode where the tool sits in between the client and the server.

You can run `prism` in proxy mode with the following command:

```
prism proxy -p 8081 api/openapi.yaml http://localhost:8080/
```

**NOTE**

**Both backend and proxy need to run simultaneously. As these tools take hold of your command line, you need to open a new window or tab.**

The first parameter (`-p`) indicates the port for the proxy, the second parameter points to the location of the OpenAPI file and the third parameter specifies the URL for the running API backend (`prism` ignores the `OpenAPI servers` element). When we tested the API with curl and Postman, we sent our requests directly to <http://localhost:8080/>. Instead of that, we now set up `prism` to connect to the API on this URL and need to change the `baseUrl` in Postman or the URLs we call with curl to <http://localhost:8081/>.

Let's try getting a user from the backend, both directly and through the proxy:

```
curl http://localhost:8080/users/test ①
curl http://localhost:8081/users/test ②
```

- ① Direct backend request.
- ② Proxy request.

Both commands should return a valid response. For the second, you should see a message similar to the following in the command line where the proxy runs:

```
[PROXY]  info      Forwarding "get" request to http://localhost:8080/users/test...
```

To confirm that output validation works, let's try and break the backend! Open `service/UsersService.js`, go to the implementation of `viewUserWithId()` and change the example. For example, you could change `"full_name" : "full_name"` into `"full_name" : [ "full_name" ]`, i.e., turning a string into an array. After modifying the source code, you have to stop and restart the backend to load the latest change. Then, executing the proxy request still works, but you'll be warned by `prism`:

```
[PROXY]  info      Forwarding "get" request to http://localhost:8080/users/test...
[VALIDATOR]  error      Violation: response.body.full_name should be string
```

Awesome, output validation works! While continuing to develop the backend and replacing mock data with actual application we can always make test requests through the proxy and observe validation errors.

Enough playing with mock data, time to build a real backend.

## 13.6 Database persistence with Mongoose

Few application backends can do a lot of useful stuff without a persistence layer that stores application data permanently. That persistence layer typically is a database, either a relational database or a document-oriented NoSQL database. In this section we explain the persistence technology we use and also look at the domain model for the database and how it related to the schemas in the API.

As our chosen backend technology is Node.js, we'll go ahead with MongoDB<sup>6</sup>, which is very often used in combination with that programming language. MongoDB is a document-oriented NoSQL database with good support within the Node.js ecosystem. There is also a library called Mongoose<sup>7</sup> that streamlines the integration between database and backend code by enabling a developer to create and interact with models, so we'll use that as well.

### 13.6.1 Another API modification

When Max created the first draft of the OpenAPI file, he assumed that all IDs are numeric and set their `type` field to `integer`. It was a reasonable assumption from someone who used to work with relational databases where auto-incrementing row numbers are the standard. MongoDB, however, uses a different, unordered approach to organize the content of the database. Because of that, MongoDB assigns longer, random, alphanumeric object IDs to the documents. That is not a problem itself, but it makes the implementation incompatible with the `type` we used in the OpenAPI file. And, as we learned, the `oas3-tools` library validates whether inputs match the schema, so we can't just ignore that, and it would defeat the purpose of the OpenAPI file as a contract and a single source of truth.

Hence, we need to update the OpenAPI file and change our schemas:

- In the User schema, change the `type` field for the `id` property from `integer` to `string`.
- In the Job schema, set the `type` for both `id` and `creator_user_id` to `string`.
- In the JobApplication schema, set the `type` for `id`, `user_id`, and `job_id` to `string`.
- In the operations that have a path parameter for the ID, change the `type` from `integer` to `string`.

As an example, here is the updated JobApplication schema:

### Listing 13.4 Updated PetSitter JobApplication schema

```
JobApplication:
  type: object
  properties:
    id:
      type: string
    status:
      type: string
    user_id:
      type: string
    job_id:
      type: string
```

We have to make the change both in the original OpenAPI file, using a pull request as described in chapter 11, and in the `openapi.yaml` file that is part of the backend project. The modification affects input validation but does not change the structure of the generated code, so we do not have to regenerate the backemd.

### 13.6.2 Getting ready to use MongoDB

To use MongoDB, you have different options. The traditional approach would be to download the database, or, more specifically, its *community server* edition from <https://www.mongodb.com/> and run it locally.

#### TIP

#### Alternatives to installing MongoDB directly

MongoDB offers a hosted service, including a free trial plan, so you also could sign up for a cloud-based database and connect to it from your development machine instead of installing it locally. However, we have not tried this, so you're on your own with this approach.

If you have previously worked with Docker<sup>8</sup> and have it running locally already, we recommend running MongoDB in a container. You can use the following single command that downloads a MongoDB image from Docker hub (if not yet installed) and starts a container:

```
docker run --name petsitter-db -d -p 27017:27017 mongo:latest
```

### 13.6.3 Configuring Mongoose in the project

You can add Mongoose to your project using `npm`, with the following command:

```
npm install mongoose --save
```

You also have to load the library and initialize the database connection in `index.js`. To do so, add the following lines in the top part of the file:

### Listing 13.5 Mongoose initialization code

```
const databaseUrl = 'mongodb://127.0.0.1/petsitter_db'; ①
const mongoose = require('mongoose');

mongoose.connect(databaseUrl, {
  useNewUrlParser: true,
  useUnifiedTopology: true
});
```

- ① The database URL for a local MongoDB instance.

#### 13.6.4 Creating models

Before we create our database models, let's recap what we did when we started designing the PetSitter application:

- Our first step was creating a domain model. In the domain model we identified the different concepts for the application, *users*, *jobs*, *dogs*, and *job applications*.
- Then, we converted the domain model into reusable schemas, `User`, `Job`, `Dog`, and `JobApplication`, and we created API endpoints for resources that are based on the schemas.

We created the domain model based on the outside view of the application and with the purpose of designing an API. However, we also need an internal domain model to represent the resources inside the database. It is crucial to understand that the external domain model for the API and the internal domain model do not have to be the same. For some applications, especially complex ones, the differences can be vast. Still, when following API Design First, the external domain model provides a good first draft for the database model. For a reasonably small web application like PetSitter we can work on the assumption that all concepts from the external model appear in the internal model, too. Some of their attributes or their datatypes can be different, though.

MongoDB stores data in documents, and every document belongs to a collection. To represent a domain model in MongoDB, we can use the following approach:

- For every schema, there is a collection.
- For each resource, i.e., each instance of a schema, there is a document in the respective collection.

And, in fact, Mongoose follows this approach already. We create Mongoose models from the schemas we have and Mongoose creates a MongoDB collection for each model. So, what are the models we have to create?

- A `User` model based on the `User` schema.
- A `Job` model based on the `Job` schema. As we've established that dogs are always part of a specific job, we can integrate the `Dog` schema into this model.
- A `JobApplication` model based on the `JobApplication` schema.

Let's create a new directory in our project and call it `models`, so we have a place for storage, and then move on to create the models.

## USER MODEL

In the `models` directory, create a new file called `User.js`. Here is the content for the new file, along with explanations:

### Listing 13.6 User model code

```
'use strict';

const mongoose = require('mongoose'); ①
const Schema = mongoose.Schema; ②

exports.User = new Schema({ ③
  email: String, ④
  password: String,
  full_name: String,
  created_at: Date, ⑤
  updated_at: Date,
  roles: [ String ] ⑥
});
```

- ① Imports Mongoose library.
- ② Creates local alias for Schema.
- ③ Defines User as export so it can be used in other files.
- ④ Field definition with String datatype.
- ⑤ Field definition with Date datatype, that didn't exist in OpenAPI.
- ⑥ Field definition with array-of-String datatype.

#### NOTE

You might be wondering why we're talking about Mongoose models if we create them with the `Schema` keyword. In Mongoose terminology, every model has an underlying schema, so we define a schema and then later when using it we import it and turn it into a model. You'll see that import code later.

If you look at the definition, you'll see that they mostly have the same field names and datatypes, but there are also the following differences between the User schema in OpenAPI and the Mongoose User model:

- There is no `id` field in the Mongoose model. The reason for that is that MongoDB implicitly adds a field called `_id` to each document so we don't have to define it.
- We added two additional fields, `created_at` and `updated_at`, which we didn't have in the OpenAPI schema. These fields help us to observe changes in the database over time. We *may* add them to the API later, but, as mentioned before, there is no need to have the same model in the database and in the API. Unlike OpenAPI, Mongoose has an explicit `Date` datatype<sup>9</sup>.

At some point in our application, we need a mapping between the internal and external model. In other words, we need to generate the API response format from the internal representation. The respective logic could be located in different places of the code. Nidhi decides to add it directly to the model files. There are good reasons against this approach because it creates a strong coupling between the internal model layer and the view layer (in this case, the API) of the application, but for a new application with a scope like PetSitter where the internal and external model bear a lot of similarity, it is a pragmatic solution.

Mongoose allows developers to add custom functions to the models they create, by attaching them to the `methods` field. Let's create a function called `toResultFormat()` which returns the external format:

### Listing 13.7 User model `toResultFormat`

```
// ...

exports.User.methods.toResultFormat = function() { ①
  return {
    id : this._id, ②
    email : this.email, ③
    full_name : this.full_name,
    roles : this.roles
  };
};
```

- ① Function definition on `User.methods`.
- ② Different field names `id` and `_id`.
- ③ Same field names for others.

The function ignores the `created_at` and `updated_at` fields that we added for internal use in the database.

## JOB MODEL

Similar to the User model, the Job model goes in a `Job.js` file in the `models` directory. As mentioned before, we're integrating the Dog schema in this model. Here is the definition:

### Listing 13.8 Job model code

```
// ...

exports.Job = new Schema({
  creator_user_id: Schema.ObjectId, ①
  starts_at: Date,
  ends_at: Date,
  activity: String,
  created_at: Date,
  updated_at: Date,
  dog: { ②
    name: String,
    age: Number,
    breed: String,
    size: String
  }
});
```

- ① Reference to another document.
- ② The inline Dog schema.

There are two interesting new things that we didn't see in the User model. One is the use of `Schema.ObjectId` as a datatype to indicate that a field references another document in the database. In our case, that would be an instance of the User model. The other is the inclusion of an inline schema by nesting the definition.

Following the same approach we had in the User model, we also define a `toResultFormat()` function that converts from the internal to the external format:

### Listing 13.9 Job model toResultFormat

```
// ...

exports.Job.methods.toResultFormat = function() {
  return {
    id: this._id,
    creator_user_id: this.creator_user_id,
    start_time: this.starts_at, ①
    end_time: this.ends_at,
    activity: this.activity,
    dog: this.dog
  };
};
```

- ① Different field names.

## JOBAPPLICATION MODEL

Last, but not least, we create a `JobApplication.js` file for the JobApplication model. There's nothing new to see there, so we'll just leave you the code without comment for sake of completeness.

### Listing 13.10 JobApplication model code

```
// ...

exports.JobApplication = new Schema({
  created_at: Date,
  updated_at: Date,
  user_id: Schema.ObjectId,
  job_id: Schema.ObjectId,
  status: String
});

exports.JobApplication.methods.toResultFormat = function() {
  return {
    id: this._id,
    user_id: this.user_id,
    job_id: this.job_id,
    status: this.status
  };
};
```

## 13.7 Implementing API methods

So far we've optimized our OpenAPI file, generated stubs for controllers and services, and created a persistence layer with a database model that looks similar to our external model. As we said in the introduction, we want this chapter to focus on going from OpenAPI to backend code with the help of Codegen but not necessarily walk through the full application. Hence, we'll just show you one of the API operations, `viewJobWithId()`, to give you an idea about how to implement any operation. You have access to the full source code of the application on the website that accompanies the book.

You can find the `viewJobWithId()` function in the controller `Jobs.js` and the service `JobsService.js`. Let's have a look at the controller code first. Here is what Codegen prepared for us:

### Listing 13.11 viewJobWithId controller code

```
module.exports.viewJobWithId = function viewJobWithId (req, res, next, id) {
  Jobs.viewJobWithId(id) ①
    .then(function (response) { ②
      utils.writeJson(res, response); ③
    })
    .catch(function (response) { ④
      utils.writeJson(res, response); ⑤
    });
};
```

- ① Call service function.
- ② If successful ...
- ③ ... write JSON response.
- ④ If an error occurred ..
- ⑤ ... write JSON response, too.

The controller does nothing more but call the service function, pass the ID and forward the response as a JSON object. But that is sufficient for now, so we do not have to make any changes to the controller code. Because the output is passed through from the service, we have to ensure that our service function returns a structure that matches the response format that we defined for the API endpoint. Also, the code structure with `then()` and `catch()` indicates the use of promises in JavaScript. The service function must return a promise which can either resolve or reject. With that said, let's have a look at the matching service code that Codegen created:

### Listing 13.12 `viewJobWithId` generated service code

```
exports.viewJobWithId = function(id) {
  return new Promise(function(resolve, reject) { ①
    var examples = {};
    examples['application/json'] = { ②
      "creator_user_id" : 6,
      "start_time" : "start_time",
      "activity" : "activity",
      "end_time" : "end_time",
      "id" : 0,
      "dog" : {
        "size" : "size",
        "name" : "name",
        "age" : 1,
        "breed" : "breed"
      }
    };
    if (Object.keys(examples).length > 0) {
      resolve(examples[Object.keys(examples)[0]]); ③
    } else {
      resolve();
    }
  });
}
```

- ① Create promise.
- ② Prepare mock response.
- ③ Resolve promise with mock response.

We don't want to return "mock" data anymore but rather return a real job from our database. Before we can write database logic, we need to load the database Jobs model, by including the following lines in the head of our services file.

### Listing 13.13 Including Mongoose Job model in a service

```
const mongoose = require('mongoose');
const JobModel = mongoose.model('Job', require('../models/Job').Job);
```

Then, we can update the `viewJobWithId()` function:

### Listing 13.14 viewJobWithId updated service code

```
exports.viewJobWithId = function(id) {
  return new Promise(function(resolve, reject) {
    JobModel.findById(id) ①
      .then(function(job) { ②
        resolve(job.toResultFormat()); ③
      });
  });
}
```

- ① Find document with ID.
- ② If successful ...
- ③ ... resolve promise with job in result format.

The code shown above works, but we can improve it by replacing the Promise-related code with modern JavaScript `async/await` syntax, which makes it more readable. Here is the improved function:

### Listing 13.15 viewJobWithId improved service code

```
exports.viewJobWithId = async function(id) {
  let job = await JobModel.findById(id); ①
  return job.toResultFormat(); ②
}
```

- ① Find document with ID.
- ② Return job in result format.

That is much shorter, isn't it? And that's all we have to do to have a working API operation, at least for successful requests. As of now, we have not considered failures in our code. For instance, we have not defined the behavior for invalid inputs such as non-existing IDs. If we run the application like this, it would throw exceptions. We will build error handling later in chapter 19.

## 13.8 Summary

- Swagger Codegen takes an OpenAPI definition and turns it into client-side or server-side code in different languages. In the case of server-side code generation, the generated code is a full application with a framework based on controllers and services. It contains stubs with mock data so that it runs out of the box. You need to fill the gaps with the business logic of the application, such as retrieving data from a database.
- To support the way Codegen organizes the code in the backend, you should tag your operations and provide the `operationId` attribute. Tags group code into different controllers and services, and every `operationId` becomes a function name. The major concepts in your domain models are good candidates for the tags you should define.
- The generated code includes input validation based on the schemas in your OpenAPI definition and rejects input that doesn't conform. You don't need to write custom code for input validation, but it is important to get the schemas right during API design. There is no output validation but you can proxy your requests through prism to get notified if your custom code violates your schemas.
- The persistence layer of an application, such as a database, requires an internal data model. This model is often similar to the domain model and contains the same concepts, but some attributes can be different. Wherever they deviate you need custom mapping code to convert objects from the internal model to the external model.

# *Integrating and releasing the web application*

# 14

## This chapter covers

- Adding minimum viable authentication
- Managing code and definition repositories
- Serving backend and frontend with a single server and base URL

Our PetSitter team has been busy. The two developers created an API definition and coded a first version of their respective parts. During development, they encountered various issues with the API design and resolved them through a change process. Now they've reached an exciting point in the project: José wants to start testing the application to see what Max and Nidhi have been doing. To be able to check the workflows for the different roles, he also wants to provide access to a small demonstration instance to additional test users, both inside and outside the company.

The two developers have to run their backend and frontend together for the first time. As they used API Design First they made sure that all changes to the OpenAPI definition are communicated. Those modifications we described in chapters 12 and 13 happened in parallel. Now they have all been merged and we have a common, stable OpenAPI file in our `main` branch - the "latest and greatest" version! This contract should ensure that both components work together and our team is eager to see that promise fulfilled.

As Nidhi and Max are about to plan their next steps, however, they realize that there are a few unsolved problems with PetSitter. Most glaringly, they had skipped over one feature in the domain model, which they believe they need for the release: authentication. Also, they had created a GitHub repository to establish a collaborative process for the OpenAPI and each developer had created a separate repository for the implementation of their part (which we didn't cover in the book, but this is what they would've done naturally), so there are now three

individual repositories. To facilitate integration and deployment, they want to reconsider this structure and evaluate if the repositories can be consolidated. Finally, they need to discuss setting up the demonstration instance, and, as a prerequisite for that, how to make sure that the URLs used for frontend and backend are compatible.

This chapter differs from the previous chapters insofar that it covers multiple smaller topics. We will describe the problem statement for each topic in the first section and then, in the same order, dedicate another section for each topic outlining the respective solution.

## 14.1 The problems

In this chapter, we'll tackle a chain of three remaining technical questions that the PetSitter team has to answer before they can set up their first demonstration instance:

- How can we identify a user and implement the "Login" action with minimal effort that is sufficient for demonstration purposes?
- How do we organize and maintain our code and API definition to be ready for deployment?
- How can we serve the application, frontend and backend as a whole, from a server that José and the other beta testers can interact with?

Within this problem section, we'll explain why they're important to answer before launch and what they entail. The remainder of the chapter covers the solutions.

### 14.1.1 Authentication

The domain model that the PetSitter team designed in chapter 9 included a "Login" action in the User concept. When Max converted the domain model to an API design in chapter 10, he skipped over the "Login" action under the assumption that there is no API operation that corresponds to the action.

Of course, every registered pet sitter and pet owner in the PetSitter application should prove that they are allowed to access their user accounts before being able to post jobs, apply to jobs, or do anything else in the system. Without some sort of security, users could impersonate each other, potentially leading to fraud within the marketplace. And even when we're just testing and have no real users yet, we can expect problems or confusion. Therefore, launching without proper authentication is not an option.

If we look closer, we can identify the following requirements:

- The "Register" action for users (`POST /register`) should not require authentication so that it is available to everyone (as the user doesn't have any account credentials yet).
- Every other action (apart from a "Login" action - which we don't yet have) involves an authenticated user, therefore we must require authentication to access them.
- Due to the CRUD structure, many API calls require the user's ID, as in `GET`

`/users/{id}`. The ID is an arbitrary identifier that the server, or, more specifically, the database, assigns to its users. On the frontend, however, users typically use their email address and their password to log in and don't know their internal ID.

To fulfill the requirements, we should do the following:

- Create an OpenAPI security scheme that describes a security credential.
- Add a "Login" action to our domain model and, hence, an operation to our OpenAPI paths that allows users to turn their email address and password into the ID and the security credential.
- Specify that the security scheme, i.e., the credentials, is necessary for every path except the "Register" action and the "Login" action.

We will look at a solution later in this chapter, after listing and analyzing all the problems. Our next step is to decide how to organize the code.

### 14.1.2 Organizing code

So far, Max and Nidhi have collaborated on the OpenAPI file using GitHub and the workflow we described in chapter 11. Each of them also set up a repository for their component's code files individually, so there's a total of three repositories. To decide whether we should keep the three repositories, let's take a step back and look at the options we have:

- Keep the existing structure.
- Create a shared Git repository for the implementation of both components.
- Consolidate code and API definition in the same repository.

To get one thing out of the way first: due to an API that acts as a clear boundary between frontend and backend and the API Design First approach it shouldn't be necessary to access the code of another component. You could deploy each of them separately, exchange the URLs, and the resulting application should work, without any developer ever seeing another's source code. So, why do we even consider moving both components into the same repository?

The reason for the PetSitter team to share the code is because they want to collaborate on setting up a single test server that hosts both components together. Ideally, the server should be portable enough so that it can be downloaded and run as a local test instance or uploaded in one step to company infrastructure. It's also important that compatible versions of the code are deployed together, as we haven't covered versioning yet. Having both components in one place might be helpful.

We will look at the three options and choose one further down in this chapter. Before we do that, however, let's move on to our third and final problem, serving both components together.

### 14.1.3 Serving both components

As of now, the application exists as two separate components, frontend and backend. Both components expose themselves over the HTTP protocol, so let's quickly revisit their URL structure. In general, a URL can be split into two parts. The first part, often called base URL, is the common prefix shared by all URLs in a component. The second part is a specific route to address an API operation, web page, or file.

An API backend has a base URL, which can be the root URL for a hostname (e.g., <https://example.com/>) or have a path prefix (e.g., <https://example.com/api/v1/>). With OpenAPI, the base URL is set in the `servers` array. The full URL for an API endpoint is the base URL combined with the path specified for the API operation (e.g., `/users`).

Being a web application, the frontend uses URL paths in two different ways:

- URL paths pointing to real file paths for static assets, such as images, external stylesheets, and JavaScript files. For example, if the base URL is <https://example.com/> and the application contains a file `images/logo.png`, the browser can retrieve it from <https://example.com/images/logo.png>.
- Paths for different pages in a web application which contain dynamic content but are not individual files. The web application has a router component that handles these URLs. In a traditional web application, the router is part of the server-side code that dynamically generates the HTML pages. For an SPA, a single-page-application, the server returns the same file - usually named `index.html` - for all URLs, and a client-side router handles page generation. The latter case applies to PetSitter, as it is an SPA that communicates with an API.

As we mentioned in the previous section, our developer team wants a portable and flexible solution that ideally serves both components together, so they can set up a single test server. As a result, the base URL for frontend and backend might be the same. Therefore, we have to think about our URL design for the two components in combination, so that there are no ambiguities. For example, the `/jobs/{id}` path could mean both the "View Job" API operation and the page that displays the job details to the user. In order to avoid such clashes, there are a few options we could think of:

- Use a different hostname or port for API and frontend.
- Use a path prefix for the API.
- Use a path prefix for the frontend.
- Employ some other URL design rules that result in URLs that can be clearly associated with one of the components.

Now that we've got an overview over the problems we have to tackle, let's get into solving them. As mentioned before, we'll go through the solutions in the same order in which we looked at the problem statements. That means, we'll start with authentication before moving on to the repositories and finishing with the URL and server setup.

## 14.2 Implementing authentication

To recap the problem statement, we need to implement authentication for most API operations in the PetSitter API to make sure that only authorized users can take action inside the application. We already identified a three-step process to do so:

1. Create an OpenAPI security scheme.
2. Add a "Login" action to our domain model.
3. Assign the security credential to every operation that needs it.

Let's walk through those steps, one by one. As a reminder, we previously covered authentication in chapter 7, so you can always refer back to that chapter if necessary.

### 14.2.1 Creating a security scheme

A security scheme describes a certain method of authentication that API consumers can use when accessing the API. An OpenAPI definition can contain multiple security schemes. There are four types for security schemes. In chapter 7 we used the `apiKey` type, which is very flexible. It describes a single parameter that API consumers should add to their requests. To further describe a security scheme for an API key, we can use the following attributes:

- Using `in`, we can specify the kind of the parameter, which is either `query`, `header`, or `cookie`.
- Using `name`, we specify the name of the parameter, e.g., the respective header, query or cookie name.

As we don't want to introduce additional complexity or a new authentication strategy at this point, we'll stick to the same approach we used in chapter 7:

- We set `in` to `header` to use an HTTP header.
- We set `name` to `Authorization`, which is the standard HTTP header name for this purpose.

#### Listing 14.1 PetSitter OpenAPI Security Scheme

```
openapi: 3.0.0
...
components:
  schemas:
    ...
  securitySchemes:
    SessionToken: ①
      type: apiKey
      in: header
      name: Authorization ②
```

- ① Identifier for the schema.
- ② The standard "Authorization" HTTP header.

We chose the identifier `SessionToken` for the scheme to emphasize the fact that the user interacts with a web application, where the word "session" is commonly used to describe the interactions of authenticated users in a certain time period.

### 14.2.2 Adding a Login action

Now that we created the `SessionToken` scheme, we have to figure out how the user can get such a token. In the domain model, we called this the "Login" action. So, how do we map this action into the CRUD structure of our API operations?

When designing an API, we try to map actions from the domain model into one of the CRUD verbs, create, read, update, or delete, as they map well to the HTTP methods. Sometimes this relation is somewhat obvious, such as matching a "View" action with the read verb. At other times, we have to think a little out of the box, for example, when we turned the "Approve" action for job applications into a more generic "Modify" action which nicely maps to the update verb.

The "Login" action neither creates, reads, updates, or deletes a user, nor any other concept within our domain model. Well, you could maybe argue that it reads a user in order to compare whether email address and password matches, but it is clearly distinct from the "View" action for users, which reveals all user details when given an ID.

Sometimes we have to move away from the CRUD paradigm and design operations in a different way. Some API designers mix in non-CRUD paths in their API design, such as `POST /users/login`, while others use a specific prefix to separate these actions, as in, for example, `POST /users/actions/login`. Before doing so, however, we should investigate whether we can achieve a more elegant solution by extending our domain model.

We called our scheme `SessionToken` as it identifies the session of a user interacting with the API. The session is a concept of the web application, so what if we made it a part of the domain model as well?

The Session concept is connected to a user. By logging in, a session starts for a specific user. In other words, we *create* a session. Did you notice? With our new concept in the domain model, CRUD comes naturally, and hence we can apply the rules from chapter 10 for turning CRUD actions into HTTP methods and API paths.

As we did with all other concepts in the domain model, let's first look at the attributes we need and turn those into an OpenAPI schema. When we discussed the requirements, we found that we have to get two pieces of information for a usable session:

- The ID of the user, so we can make API requests to endpoints that require this ID. Following the naming conventions, we should call it `user_id`.
- The credential that we use in our security scheme, i.e., the value of the `Authorization` header. To state the purpose of this field but still keep it short, we decide to call it

```
auth_header.
```

Both attributes are strings. You can see an overview of the session attributes in table [14.1](#).

**Table 14.1 The Session fields and their types**

Field	Type	Description
user_id	string	Identifier for the user.
auth_header	string	The content of the authentication header.

When we created the first version of the OpenAPI definition, we added all the concepts from our domain model as reusable schemas in the `components` section. To emphasize that we think of the session as another concept in our domain model, we create a Session schema in the same way, and add it to our OpenAPI definition:

### **Listing 14.2 PetSitter OpenAPI with Session schema**

```
openapi: 3.0.0
#...
components:
  schemas:
    User:
      #...
    Job:
      #...
    Dog:
      #...
    JobApplication:
      #...
    Session:
      title: Session
      type: object
      properties:
        user_id:
          type: string
        auth_header:
          type: string
  securitySchemes:
    #...
```

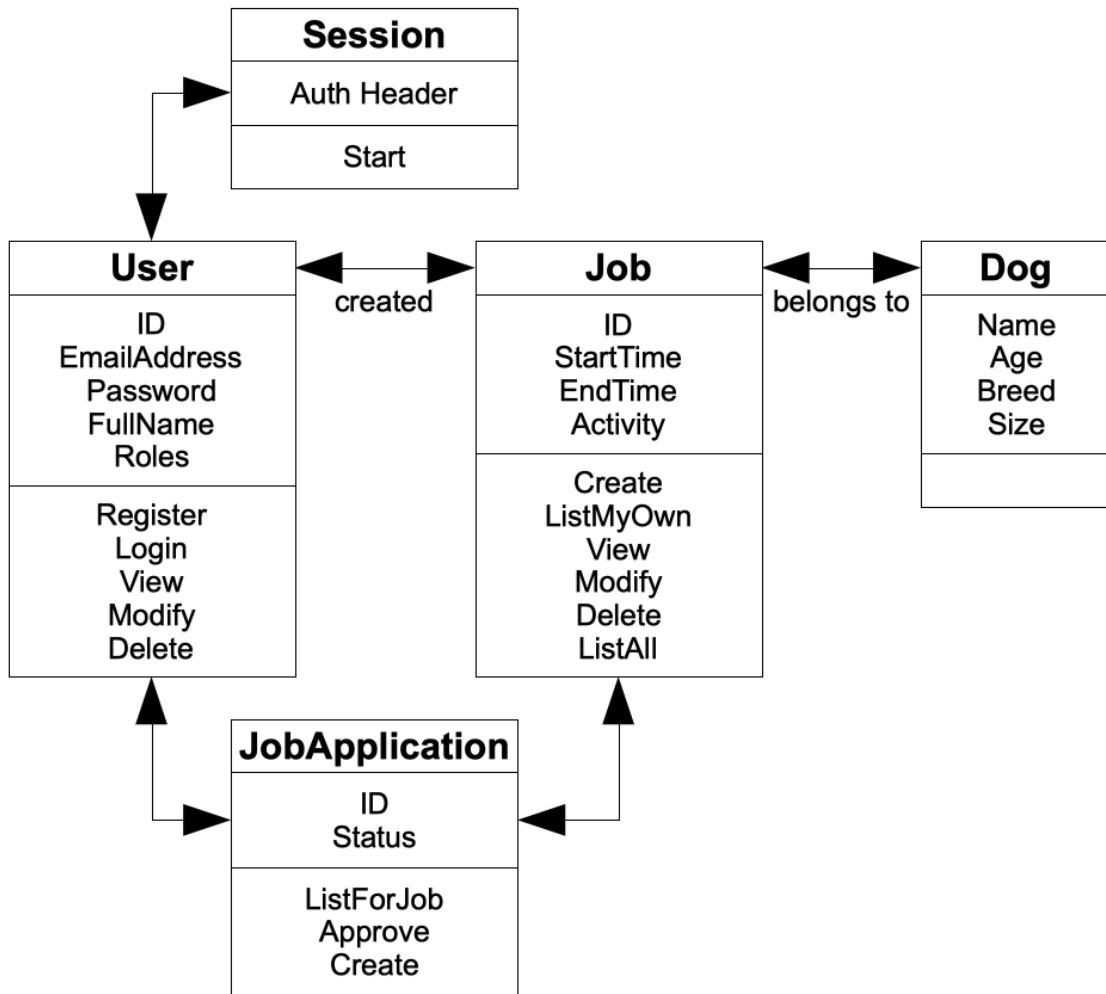


Figure 14.1 Updated PetSitter Domain Model with Session

Now that we've got a **Session** schema, it's time to add another operation. Following the CRUD mapping approach from chapter 10, the path `/sessions` is a collection endpoint for sessions, and resources are created with a `POST` request on the collection endpoint. Hence, our "Login", now known as "Start Session" action, is `POST /sessions`. Before adding this operation to the OpenAPI, we should consider its input and output, i.e., request and response schemas. As the response schema, we can use the **Session** schema we just created. For the request, we need `email` and `password` fields, which we find in the **User** schema.

At first glance, using the **User** schema as the request body feels about right. Users send the required information about themselves and get a session in return. However, the operation wants a specific subset of the **User** schema, different from the fields required for `POST /register` which already has the **User** schema as its request body. Fields like `full_name` or `roles` have no relevance for starting a session. We decide against using the **User** schema and choose a naive approach of designing an inline schema with `email` and `password` fields inside the operation.

As we learned in chapter 13, there are some additional fields that we should add to requests, so let's do that and add an `operationId` called `startSession`. What about tags? It makes no sense to invent a new tag for a single operation. There's also no connection to jobs. Therefore, we can group "Start Session" with the other user-related operations under the "Users" tag.

With all the information we gathered and the choices we've made, we can finally add our new operation to the OpenAPI definition for PetSitter:

### Listing 14.3 PetSitter OpenAPI Start Session

```
openapi: 3.0.0
#...
paths:
  #...
  /sessions:
    post:
      tags:
        - Users
      summary: Start Session (Login)
      operationId: startSession
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Session'
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                email:
                  type: string
                password:
                  type: string
#...
```

#### 14.2.3 Defining operation security

As mentioned before, we need authentication and authorization for every API operation in the PetSitter API, except for "Register User" and "Start Session". In chapter 7 we showed you the `security` attribute that enables the assignment of one or more security schemes to each operation. Now, before you go and add this attribute to the twelve operations in PetSitter that require authentication, let's look at an alternative: global security.

A global security declaration is an OpenAPI feature that allows us to make a statement like this: unless the operation's definition says something else, use the following security scheme everywhere. With twelve operations having a security requirement and two operations not having security requirements, it makes sense to define our security scheme as the default behavior and marking the two exceptions, instead of going the other way. Also, as it is likely that we'll add more functionality for logged-in users, and thus declare additional API operations that need authorization, we may forget adding the `security` attribute, potentially leaving these endpoints

unprotected. By making authorized access the default, we avoid this problem. Now, how do we do that?

First of all, a global security declaration looks like an operation-specific security declaration. It even uses the same keyword `security`, but the declaration sits at the highest level in the YAML hierarchy of the OpenAPI definition instead. We can add it to the end of the file:

#### Listing 14.4 PetSitter OpenAPI Global Security

```
openapi: 3.0.0
#...
security: ①
  - SessionToken: [] ②
```

- ① Global security keyword.
- ② Identifier for the schema and empty options array.

If we want to remove the security requirements from an operation, we have to use the `security` keyword in a slightly different way. Instead of adding a list of requirements below it, we have to explicitly set its value to an empty array (`[]`). Let's do this for the two operations that need it:

#### Listing 14.5 PetSitter OpenAPI No Security Operations

```
openapi: 3.0.0
#...
paths:
  /users:
    post:
      #...
      security: []
  #...
  /sessions:
    post:
      #...
      security: []
#...
```

The OAS tools that we got from Swagger Codegen in the backend understand these security declarations and automatically return a 401 ("Unauthorized") response when they detect a request without correctly applied security (e.g., not containing the credentials) for one of these operations. However, OAS tools does not itself validate API keys, tokens or passwords, it just checks for its presence according to the API definition, the rest is up to the backend developer. We won't cover the implementation in this chapter, but you can look at the source code for the PetSitter backend as well as frontend to see how we used authentication.

With the "Start Session" action and all security configuration integrated in the API definition, we can consider the application feature-complete for its first launch. We still need to bring the components together, in code and in deployment, so let's move on to tackle the question of code repositories.

## 14.3 Managing repositories

When stating the problem, we identified three different options for the code repositories. Before looking at each of them in more detail, we'd like to introduce two terms software teams often use when they talk about organizing code and other assets, the "monorepo" and the "multirepo" approach. In the monorepo approach, everything goes into a single repository, whereas in a multirepo approach, individual system components get their own repository. With that said, let's explore our options:

### 14.3.1 Keep the existing structure

The advantage of the existing structure, which is an obvious multirepo approach, is that different components are isolated from each other, and it is possible to provide limited access to them. As we mentioned earlier, it's not strictly necessary to have access to the source code of other components when all communication happens over APIs. The approach works well when companies want to protect their codebase because they can provide access to different repositories on a need-to-know basis, or even if they just want to harden the boundaries between components, something that's in line with API Design First.

On the other hand, as we established earlier we want to facilitate sharing the code for joint deployment. Therefore, when using a separate repository for each component, we need to access both to set up a server.

### 14.3.2 Create a shared Git repository for the implementation of both components

In a single repository for both components (not calling it monorepo right now because there's still the other repository for the OpenAPI definition), it's easier to track changes within the whole project. Every developer has a complete overview and a full copy of the codebase to run, test, and inspect. The downside is that every developer has to deal with the full content of the repository even if they are just working with a part of it.

As of now, the PetSitter team and application are not very large and complex, so the downside does not really apply here.

### 14.3.3 Combine code and API definition in a repository

Keeping your OpenAPI file together with the code in your monorepo ensures that everybody has the latest version and no developer misses an update. The discovery of new and updated OpenAPI files is effortless. However, there is the following caveat: If you do API Design First, your API definition always runs ahead of the code, because you first design new schemas and operations and only later implement them.

Following the process we introduced in chapter 11, changes happen in stakeholder branches in

the OpenAPI definition Git repository. As a reminder, there are three branches, `frontend`, `backend`, and `business`. We expect stakeholders to propose changes to the API definition in the branch associated with their role and send a pull request. Another stakeholder accepts the pull request and merges the changes into `main`. Hence, the `main` branch always contains an approved revision of the API design. It doesn't mean, however, that this revision correlates with the state of the implementation! Our process also entails developers merging changes back from `main` to *their* branch in order to see the changes and defining the tasks they need to work on to update their code to the "latest and greatest" version of the API definition.

The branch structure of our API definition repository, which we just described, was designed specifically for this process. For example, José as the product owner is a stakeholder in the API, but he is not a developer. He might collaborate on API design through the `business` branch, but not on the source code. We might have to consider restructuring this. Of course, we could use the `frontend` and `backend` branches for each developer and let them work there and merge their code into the `main` branch. Then, however, using branches for additional purposes, such as specific feature branches, complicates things. On the API design side, we may later have additional stakeholders that need access to the API (e.g. technical writers working on API documentation) but should *not* access the source code.

Another issue is the use of Swagger Codegen and the OAS tools which, as we realized in chapter 13, use the OpenAPI file directly in the implementation. Codegen, however, does its own modifications to this file. We should therefore not use this file directly to make manual design adjustments but only do so after the API definition has changed, got approval, and the change was also implemented. Hence, we'd need at least two OpenAPI files in the repository, one that the team maintains and one that Codegen maintains. Having them both in the monorepo might irritate developers and lead to modifications of the "wrong" file.

If you feel confused now ... well, so are we! Without going on into further details for these issues, our team decides that the difficulties in aligning the API change process with implementation workflows are not justified and they'll keep code and definition separate.

#### 14.3.4 Making the choice and refactoring

Nidhi and Max selected the second option. In line with their focus on simplicity first and being lean and agile, they decide to put `backend` and `frontend` in the same Git repository so they can use that to deploy their application, while keeping the existing API definition repository untouched. They create a new repository, copy their files into it, and delete the older, personal repositories. As their codebase still contains two fully separate components, each of them goes in a different directory:

- `frontend` for the frontend code
- `backend` for the backend code

Awesome, the authentication and code questions are settled. Let's move forward to the next step.

## 14.4 Setting up an integrated webserver

In the previous section, the PetSitter team created a new repository. Having frontend and backend side by side, we can now integrate them. We will first solve the URL design problem and then configure the application to serve both frontend and backend.

### 14.4.1 URL design

Earlier in this chapter we talked about the problem of potential URL clashes. Let's look at the different options we have to solve this problem:

#### USE A DIFFERENT HOSTNAME OR PORT FOR API AND FRONTEND

If we use one hostname for the web application and one for the API, for example, <https://www.example.com/> for the application and <https://api.example.com/> for the API, there will be no URL clashes. If we think about our production environment, an approach like this has its advantages, because we can use different hosting infrastructure for both components. However, it isn't a portable approach: if we run an application on a developer's machine we typically have to use `localhost` unless we want to set up hostnames locally. If we still want to separate backend and frontend, we can run two webservers on different TCP ports. This can lead to complications if one of the ports is unavailable, for example because another application blocks it on a shared server.

There's another issue: For security reasons, browsers have limitations regarding the kind of API requests that a Javascript-based frontend can make. By default, you can only make requests to the same hostname. There is a feature called Cross-Origin Resource Sharing (CORS)<sup>10</sup> that we could configure in our API to overcome this restriction, but we don't want to cover it in the context of this book, hence we rule this option out.

#### USE A PREFIX FOR THE API

An API prefix is a specific path, for example, `/api`, that we reserve for the API. Following this approach, the frontend developer has to refrain from using a particular prefix for their files and routes. As long as they do that, there will be no clashes. To follow up on the example from the introduction to the problem, when we use `/api/jobs/{id}` for the "View Job" API endpoint the frontend developer is free to use `/jobs/{id}` for the respective page. The approach also works with any hostname.

## USE A PREFIX FOR THE FRONTEND

In the same manner, we could use a prefix like `/app` for the frontend. In our example, the "View Job" page in the application would reside at `/app/jobs/{id}`. The only issue we have is that end users accessing the application's frontend will typically try the root path `(/)` on the hostname first, so we have to assign that to the frontend as well.

## DESIGN DIFFERENT URLs FOR BOTH COMPONENTS

All URLs required for the backend are defined in the OpenAPI file, which (thanks to API Design First) exists before frontend implementation. There is nothing similar for the frontend, but we can ask the frontend developer to choose URLs that don't clash with those in the OpenAPI definition. That works well in the beginning, but as we proceed we also have to consider the frontend when updating the OpenAPI definition, so this could cause problems down the line.

After investigating these options, it seems that prefixes are the best way to have a clean separation between frontend and backend that still works independently of the hostname and port used to serve the application. Nidhi and Max decide to strictly use the `/api` prefix for the backend and limit the frontend mainly to `/app` routes, with the exception of the initial URL that remains at the root `(/)` and static files.

### 14.4.2 Server setup

Now that we know how we want our URLs to look like, we still have to figure out a way to implement the chosen approach in a way that is not overly complicated. We'll look at two options here:

- Setting up a reverse proxy for both components.
- Integrate the frontend as a static part of the backend server.

Let's evaluate the first approach. A reverse proxy is a web server that accepts incoming requests and forwards them to different servers behind it. It is possible to configure many web servers, such as Apache or nginx, as reverse proxies with specific configuration rules. For example, we could configure nginx to forward every request whose URL starts with `/api` to our backend's Node.js server, and either serve the static frontend pages itself or forward to another static web server.

Production setups often rely on reverse proxies for scalability and security. By the way, a specific type of reverse proxy for APIs is called an API gateway. However, while the approach is great in production, it can be difficult to set up during development. The test server needs to run multiple processes and have all integrations configured correctly. You can't provide a single application that developers or testers can run locally on their machine. Remember, we are in the stage where our PetSitter team wants to release a working prototype as soon as possible. Therefore, let's evaluate the second option.

The Node.js backend serves the different routes as defined in the `paths` section of the OpenAPI definition. We probably don't want to add our frontend routes to the OpenAPI file, but maybe we can instruct Node.js to serve some additional routes? As it turns out, we can! The Node.js application created by Swagger Codegen uses the Express web server internally, so we can use Express functionality for that. The backend serves the frontend, so users can access the whole web application by running the backend server. That works well as a pragmatic solution during development and testing.

## SETTING THE PREFIX

The backend server mounts all the paths directly to the root of the hostname. For example, when we run our backend on <http://localhost:8080/>, the `/users` path has the absolute URL <http://localhost:8080/users>. It is, however, not the URL we want, because we decided to add an `/api` prefix. The desired URL would be <http://localhost:8080/api/users>. How can we configure this?

The `oas3-tools` library looks at the `servers` element in the OpenAPI file to find a prefix. If you open the `api/openapi.yaml` file, you'll see a single server with its `url` set to just a slash `()`:

```
servers:
- url: /
```

We can change that URL to the desired prefix:

```
servers:
- url: /api/
```

After making this change and restarting the backend, Node.js will serve our API under the prefix. Awesome, we're one step closer to our test server!

## CONFIGURING EXPRESS

As we mentioned before, an SPA often has two kinds of URL paths, and that applies to the PetSitter frontend as well. The first kind points to real static files, the second kind are the application pages that all map to the entry file of the application, `index.html`.

Supporting the first kind doesn't take much effort. Express has the `express.static` built-in Middleware function that automatically adds all static files in a given directory as routes to the API. We can use the middleware on the root path for the hostname `()` so that both the starting page and the static files are served and the routes defined by the API still work. Also, we can create a rule for every path starting with `/app` and map it to the `index.html` file using Express' `sendFile()` function. Note that, unlike for the API, this setup did not magically move the routes under `/app`, but Max had already implemented their frontend that way.

The following code listing shows the two rules in the context of the app initialization code in the `index.js` file.

### Listing 14.6 PetSitter Express Static Configuration

```
// ...

const expressAppConfig = oas3Tools.expressAppConfig(path.join(__dirname, 'api/openapi.yaml'),
options);
expressAppConfig.addValidator();
const app = expressAppConfig.getApp();

// Beginning of Frontend integration rules

app.use('/', ①
  express.static(path.join(__dirname, '../frontend/build'))); ②

app.get(/\/app\/?.*/ , (req, res, next) => { ③
  res.sendFile(path.join(__dirname, '../frontend/build/index.html')) ④
});

// End of Frontend integration rules

mongoose.connect(databaseUrl, {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

// ...
```

- ① On the root path ...
- ② ... include the static frontend.
- ③ For virtual routes starting with /app ...
- ④ ... always load the main HTML.

After adding the code and restarting the API server with `npm start`, opening <http://localhost:8080/> will serve the frontend. You can confirm that Swagger UI still exists at <http://localhost:8080/docs/> and now provides the API at `/api`.

## 14.5 Summary

- No API should go live without authentication! For a web application with authorized users, only "Registration" and "Login" actions are available to anonymous users. Through the `global security` keyword, we can ensure that authentication is required for every API operation. For the few actions that require none, we can then explicitly disable security. The "Login" action creates a session that includes the user's ID and a credential. To remain consistent with the CRUD approach, we can make these sessions a part of the domain model (and rename "Login" to "Start Session" to make that explicit).
- We can use Git and GitHub to maintain implementation code and API definitions, and there are different repository setups, each with various advantages and disadvantages. For PetSitter, we have decided to use a single repository for the application. The OpenAPI definition, however, lives in a different repository, to facilitate collaboration with non-developers and allow it to clearly represent the agreed-upon latest version of the definition, not the current implementation.
- When deploying a web application with a frontend and an API, we have to consider that both components require certain URLs. It's possible to deploy API and frontend on different ports and hostnames, but these options impede portability as they require specific infrastructure to support them. The chosen approach chosen for PetSitter is an `/api` prefix in front of the API operations. For demonstration purposes, the Express server set up by Swagger Codegen can serve both API and frontend to users.

# *The API Design First approach*

15

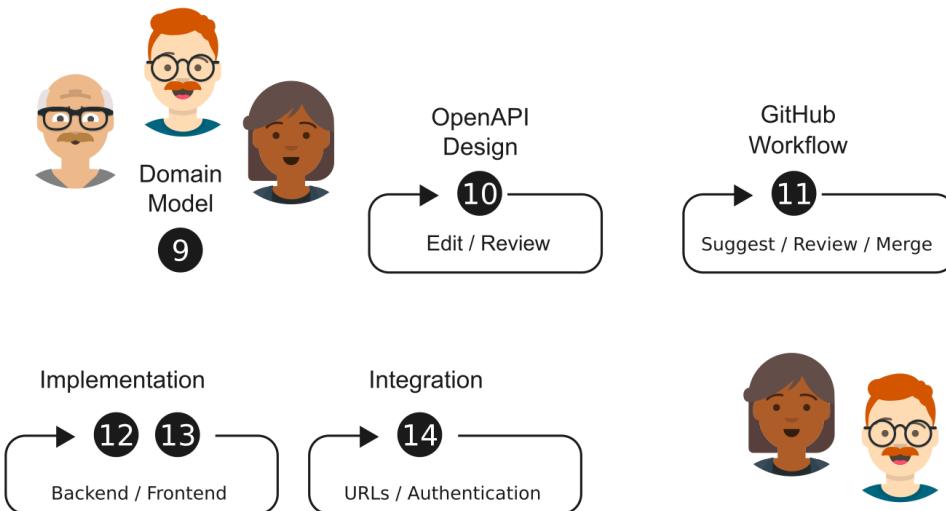
## **This chapter covers**

- Going over the material we covered in part two
- Highlighting why certain decisions were made in José's API Design First approach
- Touching up when a design is complete

The PetSitter project was accomplished by using an API Design First approach where we used the design as the central driving force to move the project forward. In this chapter we are going to revisit the decisions that helped us get there and set a standard for designing APIs in the future.

In part two we went over the following chapters...

- 9 Designing a web application
- 10 Creating an API design using OpenAPI
- 11 Building a change workflow around API Design First
- 12 Implementing frontend code and reacting to changes.
- 13 Building a Backend with Node.js and Swagger Codegen
- 14 Integrating and releasing the web application



**Figure 15.1 Steps in an API Design First approach**

Each chapter covered a part of the API Design First approach. From designing the domain models to tying together the implemented code. They were all in aide of using the design to push the project forward.

The benefits of an API Design First approach are great, they highlight issues earlier when they are the cheapest to fix and they act as common ground for different stakeholders to gather over and discuss. We will see more and more Design First (not just for APIs) approaches in our industry, we're certain of that. Although they are not without their challenges. There is more time spent designing which is time not spent building. And there is the technical challenge of keeping your design and your code in sync with each other.

OpenAPI and other types of contract documents exist to help with these challenges. They act as the manifestation of a design, data that can be written, queried and turned into code and processes—in a practical way. In this section of the book we hope to have shown how OpenAPI can be used as your next API Design First approach in building HTTP APIs. So we wanted to take the time to go over what we accomplished and highlight the salient points again.

Let's take a look back at the start.

## 15.1 Choosing the constraints

Before the project can be designed, it needs people to build it. Contrary to the name, constraints are not limitations that inhibit, they are the foundations that we can rely on. Remove all constraints and we'd be lost trying to figure out how to move forward.

José is one businessman who has an idea for a product. He has the desire for, but not the skills, required to build this project. So he draws from his pool of developers two people. Max, a

frontend developer and Nidhi a backend developer. Given the size of the project and the target audience (they're not looking for funding...just yet) then this is a well sized team. One could argue that a single person working full time, with a fullstack skillset could also build this project, but José is wise for his years. He knows that having everything inside one person's mind adds unnecessary risk to the project, particularly when the project requires more people in the future. This is no science and teams will always vary in size, roles and skillsets. What matters to us is that this is the team for the PetSitter API.

The team that you choose to build the project will set the stage for how that project will progress. Once your constraints have been laid out, trust in them and build forward. There will be a time to reconsider those constraints, but it cannot be at every decision point.

Now that a team has been established, the implementation of it can be thought out. Separating the backend from the frontend was as good step, one with challenges, but it gives us the opportunity to talk about APIs themselves. APIs are the boundaries between people more than they are between code. An API is how we ask for and communicate the needs to and from other people. By making the API explicit we are making that understanding explicit, we are incentivising documentation and communication between the team members.

Without an explicit API, the project will grow organically which can easily build itself tightly coupled and into a corner. This could end up costing a bundle to get out of that corner. A big picture is a useful thing and the explicit contract is just that.

José's team is set. Together with Max and Nidhi they'll be able to see this project off. Let's explore the next phase, designing the domain model.

## 15.2 Creating the first design

Having set up the constraints the next step is to design with them in mind. The goal of this design is to enable José's two developers to start building as soon as possible and also independently.

In chapter 9 the idea of PetSitter was broken down into a design made up of user stories and models. This paved the way to create an OpenAPI definition later.

The steps taken to establish the design of the system were the following... - Write down the idea of the project, what problem it's looking to solve. - Create user stories, small scenarios that describe how users would interact with the application and what their outcomes would be. - Build domain models, the resources in our application along with their attributes and interactions.

This process ended with an established domain model, which is where the idea of the PetSitter project starts to become more real and something that can be improved and built upon. By real we mean it is less open to interpretation—less ambiguous.

From this domain model the API definition shall evolve!

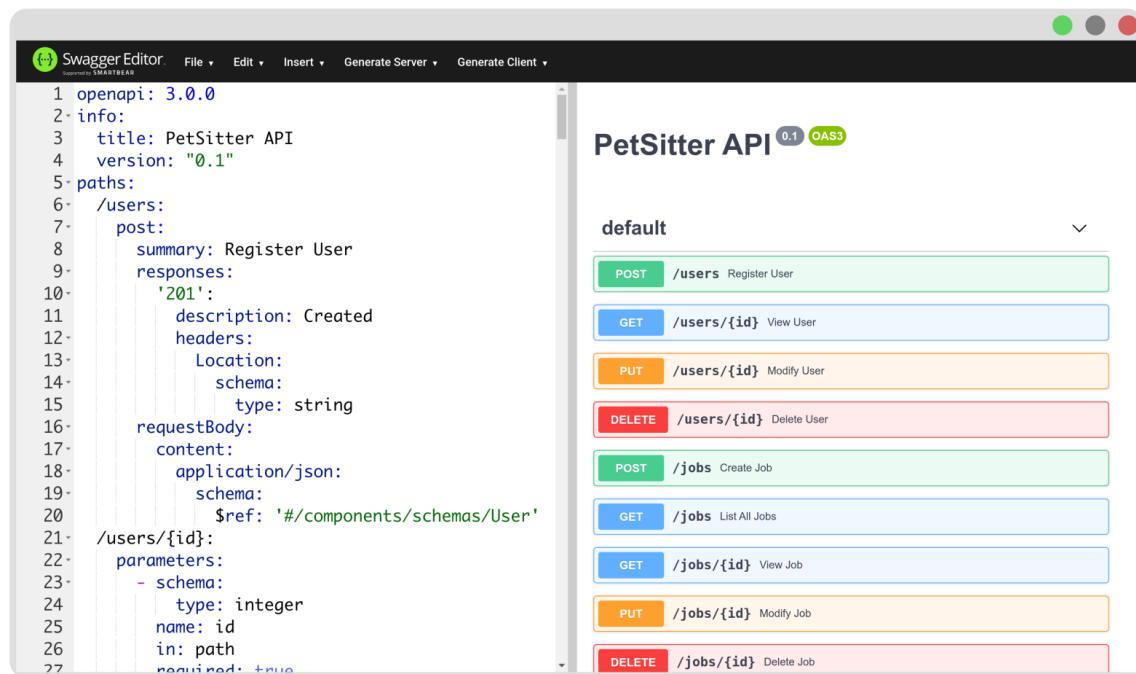
## 15.3 Creating the OpenAPI definition

Having done the hard work of domain modelling the designer is in the comfortable position of taking a pause from designing and instead has a task to do. They need to create the initial OpenAPI definition based on the domain model.

They would go through the following steps...

- Create the schemas from the domain model, mapping attributes to properties and giving them types. Taking note about relationships between models.
- Describe the operations with basic CRUD in mind, the endpoints that match the behaviours of the domain model.

It would end up with the initial API design draft that meets the criteria of the project. When the definition file is shared it enables the gathering of feedback and ironing out of the edge cases that can be identified at this stage. The team may feel the tingling of a little excitement with an API designed they're able to start moving on to building and start bringing the project to life!



```

1 openapi: 3.0.0
2 info:
3   title: PetSitter API
4   version: "0.1"
5 paths:
6   /users:
7     post:
8       summary: Register User
9       responses:
10      '201':
11        description: Created
12        headers:
13          Location:
14            schema:
15              type: string
16        requestBody:
17          content:
18            application/json:
19              schema:
20                $ref: '#/components/schemas/User'
21   /users/{id}:
22     parameters:
23       - schema:
24         type: integer
25         name: id
26         in: path
27         required: true

```

**PetSitter API** 0.1 OAS3

**default**

- POST** /users Register User
- GET** /users/{id} View User
- PUT** /users/{id} Modify User
- DELETE** /users/{id} Delete User
- POST** /jobs Create Job
- GET** /jobs List All Jobs
- GET** /jobs/{id} View Job
- PUT** /jobs/{id} Modify Job
- DELETE** /jobs/{id} Delete Job

Figure 15.2 Image of the API definition as seen in Swagger Editor

## 15.4 Keeping the code and design in sync

Designing is an iterative process. It needs feedback from people and will face challenges as soon as it's introduced into the wild. Keeping the design clean and ready for change is the primary focus of chapter 11 and is the foundation of any API Design First strategy.

Once a design has been made, the challenge becomes ensuring that the web application gets built the way the design intended. Almost as important is that your design will need to change, during and after your implementation—something the team will need consider from the start.

Keeping the design and the implementation in sync is tough but fortunately there are tool and tricks that can help.

### 15.4.1 Keeping the design in sync with code

Creating changes to the design needs structure. The following were the points of focus in chapter 11...

- View the latest API definition
- Suggest and accept changes
- Compare changes to working copy

This is a process and less of a specific tool. A series of requirements that will allow the team to update the design as smoothly as possible. With a structure in place the team will be confident in suggesting changes and knowing where the source of truth lies. This is a critical structure to set in place, *before* jumping into implementation so that the nuances here don't further complicate the challenges of design changes made during that implementation. Think of this as kitchen prep, before the dinner service when the kitchen gets hot and mad!

Using GitHub as a change tracker the team is able to get the source of truth (ie: `main` branch), they're able to make and suggest changes (using branches) and they're able to compare changes made to the source of truth or by other stakeholders (each stakeholder keeps their own branch).

This method should act as the base of how you might enforce your own way of keeping the design in sync with your team and stakeholders.

### 15.4.2 Keeping the code in sync with the design

With a design that is ready to be implemented, anyone can be forgiven if they're itching to start building it. As may become evident after taking a few designs and building them out, there will undoubtedly be small (possibly large) issues with the design and the team will need to update the design to address those. The challenge now becomes how to feed that new design *back into the implementation*. After all, a bunch of the design may already be built things *will* get messy without a structure in place to address it.

In chapter 12 we looked at how to build the frontend against a mock server and to go as far as testing the design changes before they were suggested to a wider audience. Using the OpenAPI `field examples` and executing requests with a `Prefer` header gave a way of controlling what responses and response bodies were served by the mock server. This allowed implementers to

test design changes and make sure they work before suggesting the design changes. It also gave a way to test out design changes made by others, the mechanism would be the same.

Then in chapter 13 we looked at Swagger Codegen to generate code that generated the initial server implementation<sup>11</sup>. Using it to build servers gives the team a head start and helps find design flaws even faster. But more importantly, making it part of the build pipeline (ie: list of steps that happens each time you build your code) gives an extra edge, as it allows design changes to automagically<sup>12</sup> be reflected in the code. Ensuring that changes to the design be reflected in code and setting that up does depend a lot on the coding language and the maturity of the codegen modules. In this book we used Node.js because of the ubiquity of JavaScript. However typed languages will benefit even more from codegen as you can leverage that type system to tell you exactly how the API has changed in code.

There are some tricks when working with codegen, such as adding in the OpenAPI field, `operationId`, to give the generated code methods/functions better names, which also makes changing paths/methods easier in the future.

Chapter 13 also saw us looking at how to create database models, which should be separate from the API models. Why you may ask? It might be tempting to see the two as the same and often when a project starts, they are the same. However if you do not keep them separate from the start, the API and the database will become tightly coupled. As the project grows the differences will start to show themselves and making large changes to address them may be very costly, limiting the agility of the project significantly.

It happens all too often as projects grow up and become more mature. Do not be tempted to assume your database is your API—don’t do it, we can sense you want to :D.

## 15.5 Dealing with the real world

We now come to chapter 14, integrating the components together and getting ready to ship it into production. As we discover all too often, getting our project out into the world has all sorts of weird challenges. The API was designed, built and looked just lovely, but even so when it comes time to combine different components together (frontend and backend) ... the team will run into challenges.

The first was figuring out what base URLs to use for both the application and the API. Our focus in this book was the API, so opted for the less conventional approach of putting a prefix on the *application* URLs and not the API. We wanted our API to be the star of the show.

This decision left the OpenAPI definition as-is, since we didn’t make our API take a backseat. Outside of this, you may well consider using nicer URLs for the web pages, particularly when you want them to be linked-to and shared.

The real world wasn't done testing the project just yet. The team had neglected authentication, a security measure that is obviously important. In the application they opted for the tried-and-test API key approach. Which allowed them to share that approach with both the API consumers (who they look forward to seeing in the future) and the web application users.

One would think that the API design and its implementation might be finished after it's deployed and running, but if you take away only one nugget of information from this section of the book, it is this... change is inevitable and planning gives you control over it.

## 15.6 When a design is complete

Throughout this chapter we've talked about designing and also the importance of getting the design out there sooner to catch those design flaws that only tend to appear out in the wild.

A design is complete when it expresses the whole thought although not necessarily with all the details. For the PetSitter application that thought was allowing someone with a pet to connect to someone looking for a job as a pet sitter. When the design allowed for that to occur it can be considered complete and ready to move onto the next stage.

Hold on! What about the details? What about user logging in, or deleting jobs, or a number of other small actions that may not fit that "whole thought". These details *are important* but there will be other details too, others that will only be discovered during implementation (or worse, during deployment).

We know that as the project progresses, design issues become harder to change. So when do we design more and when to start implementing? We make sure that our design must be *at least* complete. So that it *at least* describes the original thought completely. After we've reached that point, any design work will certainly be valuable the more we iterate over it, but we will get diminishing returns. If we think about it, it makes a little sense. If you spend one day designing the API you may get it to be complete. If you spend another day, you'll discover a bunch of edge cases and critical issues and perhaps even resolve them. Fast forward to two months spent on one simple design and you may not even find a spelling mistake. Having spent two months, as soon as you try to build it, a whole new class of design issues will show themselves. While you were busy fixing typos in the design those issues were waiting for you!

We consider a design to be complete when it expresses a complete thought. And the sooner we can get that design to the implementation phase, the sooner we can discover and address those hidden issues. This whole process is about improving a design not about designed everything up front. For this reason we encourage investing in ways to make changes to your design as frictionless as possible, during all stages of that process.

Happy designing!

## 15.7 Summary

- API Design First enables us to think through what we want to achieve and OpenAPI is that platform when designing HTTP-based APIs.
- The API Design First steps to building an API from scratch are: Break down your idea into user stories, model your domain, describe your API, establish how you can change your design throughout, implement your code using the tools available and always be prepared for more change!
- Domain models are the building blocks from which you can develop your API. The models, attributes and interactions will map cleanly to OpenAPI and HTTP's endpoints, schemas and methods.
- You can keep your frontend implementation in sync with your design by using mock servers to test against. And you can keep your backend code in sync by generating chunks of that code from your API design. We know changes will occur so it's important to think about keeping things in sync.
- Integrating your components, such as frontend and backend may have issues. Be prepared to consider those. The two we encountered were the base path of the API compared to the Application and the other was a missing authentication mechanism.
- A design is complete, when it expresses at least a complete thought or the entire intent. This does not mean it needs to have every detail, so long as it does what you want it to do. Spending more time on the design will catch more issues, but you will experience diminishing returns. Getting your design into the wild should be a high priority.

# 16

## *Designing the next API iteration*

### This chapter covers

- Planning the next development sprint with the PetSitter team
- Reviewing and updating the user stories with new functional requirements
- Aspects of developer experience that we'll cover in upcoming chapters

In the second part of the book, we met José and his PetSitter team and joined them as they created a web application from scratch. Their journey started at a whiteboard draft of a domain model, continued with an initial API design phase, went through the implementation with various API changes along the way, and finally ended with publishing the first prototype. In the third part of this book, we continue our journey together with the PetSitter team and application, as they take the next baby steps towards domination of the global pet sitting industry.

Let's imagine the demonstration of the software went incredibly well and the application worked as designed. Overeager José immediately decided to start using it in production and posted some jobs where he needed someone to take care of his dog. Then, he recruited a few of his previous pet sitters as beta testers for the application, who willingly applied to the pet sitting jobs through the new system. Nidhi and Max didn't want to stop his enthusiasm and made sure that the server kept running smoothly.

While word spreads about the new PetSitter application and feedback begins to trickle in, José and his team start thinking about the next steps. They want to implement requested changes, of course, to make their users happy. However, they also know about the future milestones that they want to achieve with PetSitter and that they prepared for with their API-driven architecture - a mobile application and an eventual API release. They should make progress towards that as well.

In this chapter, we'll learn about sprints as a way to understand phases of software development.

We'll review the previous sprint and plan the next sprint, which includes both new functionalities and API improvements to make the API ready for future requirements. For the new functionalities, we'll review our user stories to prepare for the necessary changes to the domain model, the API definition, and the implementation. For API improvements, we'll learn about developer experience and highlight a few aspects that we will tackle in upcoming chapters.

## 16.1 Reviewing the first development sprint

The word "sprint" indicates a block of time in a software development project. In the beginning, the team defines their goals to define the scope for the sprint. Then, everybody gets to work. At the end, the team reviews their work and the process before starting the next sprint. You may recognize this structure from part two of the book. Similarly, part three can be considered another sprint, and follows roughly the same structure. It highlights the fact the software development and API design are circular activities.

**NOTE**

You may know the term "sprint" from agile methods like Scrum. The PetSitter team is not following Scrum, but we think it may still be helpful to apply this as a rough framework to understand how the development progresses. In Scrum, a sprint is typically one to four weeks long. The initial event is called sprint planning. At the end of a sprint, two events take place: the sprint review in which the team demonstrates their work (chapter 14), and a sprint retrospective in which they reflects about the work process (chapter 15) and refines it if necessary.

At the beginning of the first sprint when José first decided to bring his PetSitter project idea to life, he collected some initial functional and non-functional requirements. To prepare and start the next sprint, he wants to check both lists to see how much the team already achieved and what can and should be done next. As a reminder, these were his original functional requirements that described the features of the app:

- Sign up: As dog owner or dog walker
- Dog owners: can post jobs
- Dog walkers: can apply to posted jobs

José had purposefully started with a short list of requirements that his team was able to implement in a single sprint. The current version of PetSitter already supports the full set of functional requirements and José can check them all off.

Apart from the functional requirements, José wrote down non-functional requirements for the product and the development process:

- Build web app with in-house team - two people
- Mobile app - work with other development agency (later!)

- Chance to experiment with new technology
- Release first working prototype as soon as possible

In the first development sprint, the developers Nidhi and Max built and released an app that already works for their boss and many other people. José can happily say that his requirements of being able to build the app with an in-house team of two people and releasing a working prototype as soon as possible was fulfilled. It was an inaugural API Design First project where his company used OpenAPI throughout the whole lifecycle, which was a great chance to experiment with this new (for his company, at least) technology. Hence, he can check off three out of four items. The remaining open item on the list is the mobile app.

## 16.2 Planning the next sprint

From the review of the requirements, José sees multiple goals for the next sprint. All current functional requirements have been met, but, as he talked to all his friends and colleagues and other people who tested PetSitter, José got a variety of different suggestions for the new features. On the other hand, he wants to get ready to build the mobile app as well.

The mobile application is an important milestone, as José knows that many people access services primarily on their phones instead of desktops and laptops. Especially when they are regular users, a native mobile app can perform better than a website and have a deeper integration into the mobile phone and its user's life, for example through push notifications. Being present in the app stores of the mobile operating system vendors (such as Apple and Google) provides additional visibility.

José doesn't have in-house mobile developers and he doesn't want to recruit any just yet, considering his primary line of business is custom websites and PetSitter is still an experiment. As he expects the outsourced development project will take some time as well, he wants to be ready as soon as possible. He needs to find an agency and prepare a contract. Once that's done, he needs to quickly provide the external team a functional backend and well-documented API so they can build that app efficiently.

The term "API" did not appear on the list of non-functional requirements as José wrote the requirements without thinking about the technical parts of the implementation. The list together with the initial discussion with his developers during the kickoff meeting provided the motivation for the team's decision to build an API-driven application with clear separation of backend and frontend. With that architectural choice, the team prepared for sharing the API with a third-party contractor to build a mobile app on top of the same backend. That same choice put another potential milestone on the horizon: sharing the API with additional parties and eventually opening up to the wider public.

At this point, it may be too early to release the API, however it makes sense to keep that goal in mind and continuously improve the API. We'll go back to that later in the chapter when we talk

about developer experience. We can think of the private API release to the mobile app development contractors as a stepping stone of API releases. First, the API is purely for internal use. Then, you share with a limited number of external developers. Finally, you open the floodgates.

So, Josés primary goal for the development sprint, which is what we cover in part three of this book, is to get the API ready for the limited release, in which the API will have exactly one external consumer: the mobile developer. We'll discuss what it means for the API to be "ready" later in this chapter. And, if the team can do first steps towards a wider API release (to more developers or the general public) in the process, even better. We'll get back to the missing steps for a public API in chapter 22.

Again, at the same time, José wants to address feedback from the first release. Obviously, he would like to make all users happy, but he also knows from past projects with his web development company that you have to maintain focus and not jump at every client request immediately. His plan of action is to look at all his notes from the conversations with beta testers and potential users and put them into an order with the same or similar ideas grouped together. On that basis, he sorts all ideas in descending order of priority, where priority is the number of people who made the suggestion.

**NOTE**

We're aware that strictly following the masses is probably not the best approach to a great product development strategy, but for our fictional pet example (pun intended!) it's probably fine.

With his intentions set for the sprint, José goes and talks to Nidhi and Max again to get their opinions and agree on the final plan.

### 16.3 Preparing for new features

Similarly to the kickoff session at the beginning, José, Nidhi, and Max come together for another planning session. At the start of this kickoff meeting for the next sprint, José presents the first three ideas in his prioritized list of suggestions made by PetSitter users:

- The top missing feature was raised by a few dog owners who have multiple dogs. A lot of times they need someone to take care of all these dogs, but the application's frontend allowed them to enter the details like name, age, and breed for only a single dog as part of a job description. As a result, they had to either resort to mentioning only one dog or creating multiple jobs for what was essentially a single job.
- The second most requested feature came from pet owners who don't just have dogs but also other pets, like cats. Since they also sometimes need someone to take care of the cat (or the hamster), they would appreciate if they could select other species of animals, not just dog breeds, when creating new jobs.
- The third issue was something that was on Josés mind already, but also raised by some of the forward-thinking testers with a technical background. As of now, the frontend shows

a long list of jobs available in the system. What happens if that list gets very long? How will the list be organized, will there be multiple result pages, or are there filter and search features?

Hearing about the first two requirements, Max reminds José that the team had already expanded the terminology of the initial functional requirements during the creation of the domain model to be broader than dogs. They could now be formulated like this:

- Sign up: As pet owner or pet sitter
- Pet owners: can post jobs
- Pet sitter: can apply to posted jobs

"I see", says José, "but does that mean we can easily support these new feature requests?" "No", Max replies, "we have used the broader terms in our User schema, but we still just have a Dog schema! We have to update our user stories and the domain model."

For the third requirement, Nidhi suggests implementing both filters and pagination. "That way", she says, "we can have users select the types of jobs they want to see, and we make sure there's always a limited numbers of posts in the API and the frontend. That helps with scalability, too." José nods. He wants to start looking at the domain model and the user stories right away so that the team can estimate the complexity of the changes. He reminds them that his main goal at the moment is to prepare the API for release and implementing the feature comes second. "Well", Max said, "these could be small but still breaking API changes, so it makes sense to tackle them now!" A breaking change in an API, to quickly define the term, is anything that requires consumers of that API to change their integration because otherwise it would no longer work. We'll talk more about handling breaking changes later in this chapter and also in chapter 21.

As a summary, we can describe the new functional requirements like this:

- Pet owners: can post jobs containing one or more pets of various types
- Pet sitters: can browse jobs based on certain criteria, divided into pages

Now, these are just two items instead of three. How come!? Well, what we did is combine two requirements into one and, through that combination, we go further than the user feedback. While users asked for multiple types of pets and multiple dogs per job, it's probably safe to say that adding more than one pet to a job is not just useful for dogs but all pet types supported in the system.

### 16.3.1 Reviewing the domain model

As you know from chapter 9, a domain model is an abstract representation of concepts from the real world that play a role in our software. For each concept, we can describe attributes, functionality, and relationships to other concepts. The domain model created at the beginning of part two acted as the foundation for the schemas in our API design that resulted in a working implementation at the end of the part. The model had four concepts, *user*, *job*, *dog*, and *job application*. In chapter 14, we added a *session* concept solely to help with the "Login" action for users.

In our previous sprint, i.e., the second part of the book, we created the domain model in two passes. In the first pass, we only looked at the concepts and their attributes. Then, in the second pass, we wrote the user stories and, based on them, added the actions and relationships to the domain model. This section can be considered the first pass of domain model reviews, before looking at the user stories in more detail.

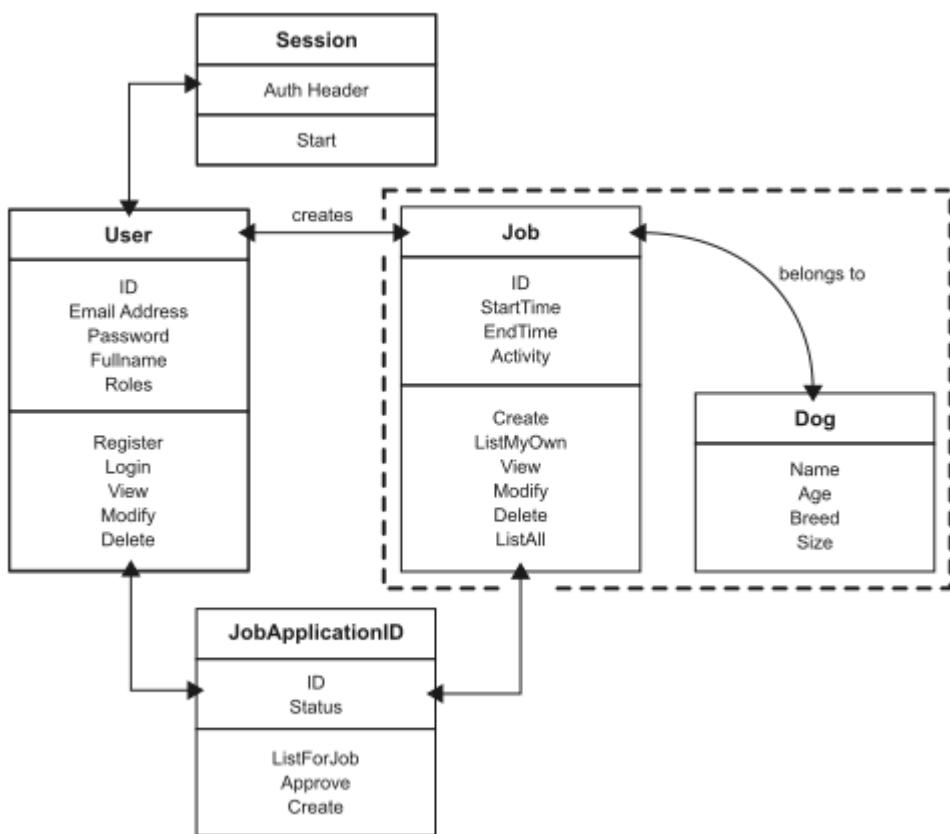


Figure 16.1 PetSitter Current Domain Model

The new requirements don't fundamentally change the way PetSitter works. There are still users with different roles. Pet owners create jobs and pet sitters apply to them. The updated domain model still needs *user*, *job*, *job application*, and *session* concepts. What about the *dog*, though?

Two options appear in the minds of our PetSitter team:

- Replace the *dog* with a *pet* concept.
- Add each type of pet as its own concept, i.e., have *dog*, *cat*, etc.

In figure 16.1, we've highlighted the segment of the domain model that may need changes. In fact, for the entire part three of this book, we are only working with this segment. It includes the *dog* as well the only concept directly connected to it, the *job*, as we can assume that whatever supersedes *dog* will similarly be tied to *job*.

In the original design, the team made the design choice to have dogs tightly connected to jobs in a one-to-one mapping and not maintain them separately in the system, as that would require additional user stories (which turn into actions, and those turn into API operations) to create and manage dogs individually. With pet owners creating jobs with multiple dogs or other pets, it may be worth revisiting the design choice in the future, but as we don't have the requirement to do that now, we won't.

Since we're already talking about it, let's stick with the *job* concept for a moment. We can expect that jobs will have the same attributes in the new version. The activities could be different for different pets (e.g., you wouldn't take most cats for a walk), but we had not tried to represent different activities in our domain model and thought of this as more of a "free-text" job description field. Adding a structured datatype with an enumeration of various activities is something we could do in future but the implications for the domain model would be too heavy for the sprint. So, again, we won't change the attributes of the *job* for now.

At this point, the PetSitter team is unsure which option they should choose for the *dog*, or whether there's other ways they haven't thought of. Hence, they decide to postpone this discussion for now. Here's a sneak preview for our dear readers: yes, there are other options. They require us to learn additional concepts about domain modelling, though. We'll dedicate chapter 17 to this problem and its solution.

### 16.3.2 Reviewing user stories

In chapter 9, the team collected user stories that cover the potential interactions that users with different roles have in the PetSitter application. Now we need to determine whether they need to be updated, or if we need additional user stories. Those modifications may then lead to additional domain model changes as well. Our first step is to reduce the full list of user stories into a smaller list of relevant stories to investigate. So, here's the long list:

- I can register a new account and choose my role, so that I can log in.
- I can log in to my account, so that I can use the marketplace.
- I can modify my account details.
- I can delete my account.
- As a pet owner, I can post a job on PetSitter, including a description of one of my dogs,

so that pet sitters can apply.

- As a pet owner, I can see a list of jobs I have posted.
- As a pet owner, given that I have posted a job, I can view and modify its details.
- As a pet owner, given that I have posted a job, I can delete it.
- As a pet owner, given that I have posted a job, I can see the pet sitters that applied.
- As a pet owner, given that I have found a suitable candidate, I can approve them.
- As a pet sitter, I can view a list of pets that need looking after.
- As a pet sitter, given that I have found a job, I can apply to it.
- As an administrator, I can modify other user's account details.
- As an administrator, I can edit jobs that other users have posted.
- As an administrator, I can delete users.

We already stated in the previous section that the way PetSitter works at a high level doesn't change with the new features and the domain model contains more or less the same concepts, except for the *dog*, although we haven't decided yet how to change it. However, we had the realization that whatever supersedes it will similarly be tied to *job*.

In a first attempt to reduce the user stories to consider, the team looks at the highlighted concepts in figure 16.1. The user stories including the words "job" or "dog" seem to be relevant. All the others can be discarded, as they don't touch any of the new concepts. Here's our new list, which is already much shorter:

- As a pet owner, I can post a job on PetSitter, including a description of one of my dogs, so that pet sitters can apply.
- As a pet owner, I can see a list of jobs I have posted.
- As a pet owner, given that I have posted a job, I can view and modify its details.
- As a pet owner, given that I have posted a job, I can delete it.
- As a pet owner, given that I have posted a job, I can see the pet sitters that applied.
- As a pet sitter, I can view a list of pets that need looking after.
- As a pet sitter, given that I have found a job, I can apply to it.
- As an administrator, I can edit jobs that other users have posted.

You may be wondering about the one user story in that list neither containing the word "job" nor "dog". Nidhi added it because she noticed the word "pets" which appears nowhere else. Also, she vaguely remembers something peculiar about it. Bear with me, we'll get to it in a minute.

Again, in the current domain model, *dog* has no actions. The API design from chapter 10 has no operations for dogs either. With the way the team designed and implemented PetSitter, dogs are a passive part of job descriptions. There is a reference to the Dog schema from the Job schema, nothing else.

When users interact with PetSitter, they create, view, edit, and delete jobs. These jobs currently tag a dog along. With the new feature implemented, they tag one or more pets along. That fact doesn't change with our choice of representation, which, as mentioned before, is postponed to

chapter 17. Hence, we can safely say that none of the user stories about jobs will change, and we can eliminate those that only mention "job" from the review list.

Suddenly, the list of user stories that need a review only contains those that mention the word "dog", as well as the one mentioning "pets". In other words, we shrunk the list to just two user stories:

- As a pet owner, I can post a job on PetSitter, including a description of one of my dogs, so that pet sitters can apply.
- As a pet sitter, I can view a list of pets that need looking after.

For comparison, these are our two new functional requirements:

- Pet owners: can post jobs containing one or more pets of various types
- Pet sitters: can browse jobs based on certain criteria, divided into pages

These two lists look very similar, don't they?! Each of the new functional requirements appears to affect exactly one user story. Of course, that is a coincidence and not a general rule. We're just lucky. Let's investigate each of the user stories. We'll first rename it to accurately reflect the new requirements. Then, based on the new wording, we'll look at the implications for our domain model.

### **AS A PET OWNER, I CAN POST A JOB ON PETSITTER, INCLUDING A DESCRIPTION OF ONE OF MY DOGS, SO THAT PET SITTERS CAN APPLY.**

In the title of this user story, the crucial segment is "one of my dogs". It doesn't fit the new requirement of supporting multiple pets, which don't have to be dogs. We can change the wording of the user story like this: "I can post a job on PetSitter, including a description of one or multiple of my pets."

The domain model reflects the original version of the user story in the form of the "Create" action on the *job* model. The action name and the concept it belongs to is no different with the new requirements (which only affect the job-dog relationship, our task for chapter 17), so we won't have to make any changes to the *job*.

### **AS A PET SITTER, I CAN VIEW A LIST OF PETS THAT NEED LOOKING AFTER.**

This user story doesn't contain the word "job", but it contains the word "pets", which wasn't a part of the original domain model. However, as the team realized in chapter 9 when they converted the user stories into actions, that was just sloppy wording. Since pets (at that point, exclusively dogs) belong to jobs, what pet sitters are viewing are, in fact, jobs. Let's first rephrase the user story to use the same terminology as the domain model: "As a pet sitter, I can view a list of available jobs."

Based on the user story, we added the "List all" action to the job model. However, with the new

requirement for browsing jobs based on certain criteria, a "List all" doesn't cut it anymore. As we discussed earlier in this chapter, a long list of jobs makes it overwhelming for pet sitters to find suitable jobs and doesn't scale beyond a certain number of jobs, so we decided to give them the opportunity to search specific jobs through filters and view them separated into different pages through pagination.

Let's imagine a user journey in the new app. Instead of seeing a single page listing all the jobs, pet sitters will see a variety of search criteria. They could filter for jobs with certain types of pets, in a range of dates, or for specific types of activities like dog-walking, just to give some examples. After selecting their filters, they'll see jobs that match their search criteria. If there's a larger number of jobs, they will only see the first, let's say ten, jobs. At the end of the list, they can change to the next page.

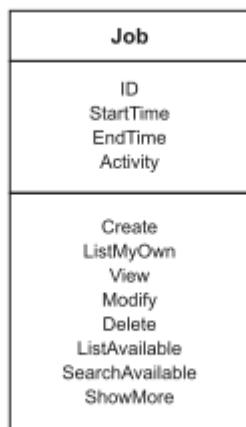
Looking at the user journey, there are two activities. Let's formulate them as user stories:

- As a pet sitter, I can specify the kinds of jobs I'm looking for and browse jobs with these criteria.
- As a pet sitter, given that I'm seeing a list of jobs and there are more jobs available, I can navigate to the next page to see more jobs.

Representing these two new user stories, we add two new actions to the *job*:

- Search available
- Show more

We can continue letting pet sitters view the list without any filters, too, but we shouldn't call it "List all" because that list is also subject to pagination. Let's call it "List available" instead, to get rid of the word "all". You can see the updated job model with the three actions (one renamed, two added) in figure [16.2](#).



**Figure 16.2 PetSitter Job Model with Actions**

While we mentioned some examples, we haven't decided on filter criteria yet. We'll do that when we look at the OpenAPI changes to support them in chapter 18.

Alright, let's recap the functional changes. We've reviewed our domain model and our user stories to reflect the new functional requirements brought in by user feedback and will finish that process with OpenAPI changes in chapters 17 and 18. We changed one user story to support multiple pets and added two new user stories to describe filters and pagination. The *job* model already received new actions and the *dog* model will still undergo changes or replacement. The PetSitter team is happy and concludes the first part of their sprint planning. Let's join them as they move on to the next topic of the sprint.

## 16.4 Improving developer experience

Apart from the new features we've discussed until now in this chapter, the PetSitter teams plans to be ready for the mobile app development at the end of the sprint and also take necessary steps towards an eventual release of the API.

José had decided to contract another development agency that has the skills and knowledge to build a mobile app for PetSitter. The external mobile developer will be in a similar position as frontend developer Max. He or she will interact with the backend that Nidhi implements through the same PetSitter API. However, compared to Max, they will be at a disadvantage. As they were not involved in the designing process, they will only see the end product without knowing what went into it.

The PetSitter team chose the API Design First approach to let Max and Nidhi work independently so they don't have to disturb each other, but reaching out and discussing issues with each other was always an option for them. For an external person, things are different, because they are further removed from the original development.

Every company wants happy customers. Customer satisfaction depends on a variety of factors, such as the purchase process or customer service, but it obviously starts with the product itself. None of the second-order experiences of interactions with a product's vendor matter as much as the customer recognizing that the product itself has high quality and using it is delightful. We can think of every API as a product as well, even if starts with only a single internal customer.

Whenever we're talking about developers integrating components from other developers such as APIs, we should talk about "developer experience". So, what does that term mean? For comparison, "user experience" describes what a general user experiences while interacting with a piece of software (or another product, such as a hardware panel). In a similar fashion, the term "developer experience", often shortened as DX, describes the personal experience of a developer interacting with an API, SDK, library, or other developer tool.

You may argue that an API connects machines and handles communication between software

components, so it must primarily fulfill that job by being efficient and performing well. Of course, technical performance matters, but APIs are for machines *and* humans. The customer is a human and their first impression of the API while understanding its purpose and implementing the integration counts. Hence, the emphasis on developer experience.

Developer experience also helps with organizational scalability. The more developers consume an API, the more it pays off to invest in good DX. If bad DX leads to problems for one engineer integrating an API, a support person can help them. But if that person needs to support everyone they will be overwhelmed with requests. Max can walk over to Nidhi's desk or ping her on the company's internal chat. The mobile app contractor can possibly also schedule a call with Nidhi if it's necessary. We can't expect her to provide support to hundreds of API consumers, though. With good developer experience, those API consumers are able to solve most of their problems without additional help from the API provider. It's still the early days for PetSitter, but if they can improve their DX now it will pay off later.

In José's case where his company provides an API to an external consumer that he's contracted to build a mobile app, any delays due to bad DX don't just result in wasted time and frustrations. Considering a software developer's hourly rate, it can quickly become really expensive. José thinks of his bank account and the money he's already spending on debugging, fearing high bills from the contractors. He asks his team: "What can we do to drive developer experience forward during this sprint?" Max and Nidhi start thinking.

For APIs, developer experience comes from two primary sources, API design and API documentation. Especially for public APIs there are additional aspects of developer experience related to the second-order experiences I mentioned above, such as the responsiveness of the support team or an active developer community forum. For now and for this book, we want to focus on the API design aspects. You can always write additional documentation after you've built an API and you can continuously enhance your developer support by establishing a developer relations team, but with API Design First you need to think of the API design aspects of DX from the beginning.

We will look at a few aspects of DX now, expand on them in further chapters within the third part of the book, and touch on the remaining DX issues when we talk about taking the API public in chapter 22. You can think of these areas as those that Nidhi and Max want to focus on during the sprint. They agree that in its current state the API is not ready for the mobile developer yet, but after the DX improvements it will be.

### 16.4.1 Consistency

The most important aspect of DX is that developers can discover patterns within the API where similar API endpoints have similar design. In chapters 9 and 10 we outlined a formal approach to API design that leverages domain modeling and creating endpoints based on specific CRUD design rules. By following that approach in the initial design but also for updating the API with new features, as we discussed in this chapter, Josés team already scores well in this regard.

### 16.4.2 Error handling

While discussing with the team, Max says, "I remember there was a situation where I was struggling with the API and couldn't understand what was wrong. In the end I realized it was my mistake, but it took me some time to figure out." It's an all too common situation for software developers to be in his place, seeing things that don't work and trying to debug to find the underlying cause, which can either be a mistake they made or a bug in another component. One aspect for great developer experience is that, in case of errors, developers receive as much relevant information as possible that helps them debugging.

For APIs, that kind of information primarily comes in the form of error messages that the API returns when either the input is invalid or something else went wrong outside the influence of the developer integrating the API. These messages should be clear, actionable, and, you may have guessed it, consistent.

As Max described the situation, Nidhi replies, "While developing my backend I haven't looked into error handling yet. I just used what Swagger Codegen provided. I can overhaul the error messages to make it easier for you and future API consumers."

We're dedicating the entire chapter 19 to error handling, where we'll introduce a standard error format and explain best practices for error messages.

### 16.4.3 Input validation

When we created the OpenAPI description with the PetSitter team in chapter 10, we created schemas with attributes and data types for those attributes. Then, in chapter 13, we looked at input validation and output validation for API requests and responses using those schemas. Now, how is this relevant for developer experience?

Through schema validation, we can guarantee that the API follows its contract, the API definition. If the API follows the contract, which we can ensure through validation, the (frontend) developer consuming the API can rely on the contract as well. They can be confident that the API acts exactly as it should and as it is documented, avoiding unexpected surprises that result in bad DX.

There are two relevant principles of software engineering that seem to be at odds, though. The

first one is Postel's Law, also known as the robustness principle. It states that any system should strictly follow the specified contract when it comes to the output it generates. At the same time, it should be liberal in the input that it expects from others. Applied to APIs, it means that, for example, a field defined as `string` should always be a string in the API response, but the API may decide to accept input as `integer` as well.

The second principle is Hyrum's Law, which is the observation that any observed or accepted behavior of the system will be the de-facto contract and someone will probably depend on it. To follow our previous example, if the API's contract only says that a field must be a `string` but accepts `integer` as well in practice, there'll eventually be an API consumer relying on that behavior. Which means that if the implementation in the API changes in the future and no longer accepts the `integer` value, it would be a breaking change for said API consumer. The breaking change would go unnoticed because the OpenAPI definition hasn't changed and any API Design First change process wouldn't catch that. Hence, it may be better to strictly follow the contract in the beginning when you're still able to enforce it, and schema validation for inputs helps with that.

So far, we only talked about data type validations, such as enforcing an input to be `string`, `integer`, or `array`. However, JSON Schema can do much more. We can specify required and optional fields, and we can also provide ranges for numbers and enumerations of allowed string values. In chapter 20 we'll thoroughly look at these and more.

#### 16.4.4 Versioning vs. evolvability

Changes in APIs are inevitable, as a product grows and expands its set of features. We need to distinguish breaking and non-breaking changes. In a nutshell, a non-breaking change is a change that doesn't require users of the software to change their behavior. The most common example are bugfixes and new features that do not touch the existing system. Breaking changes, on the other end, require users to adapt to new patterns of interactions.

Thinking about UI design and human users of a system, the line between a breaking and a non-breaking change can be blurry. When a UI element gets a new color or a slightly edited caption (e.g., "Store" instead of "Save") or two buttons switch places, human users are typically quick to notice and adapt to the new system. For APIs, it's different. API consumers and API providers must follow the contract, the OpenAPI definition, to the letter, quite literally. If the contract changes, integrations may break. When we consider developer experience, we see that change management is even more important than it is for user experience.

For PetSitter, frontend and backend are deployed together, at least for now. If we introduce a new API with breaking changes, both components need to speak the new contract at deployment

time. Whether we only deploy at the end of the sprint or regularly as the development happens, we must only deploy compatible versions. Breaking changes are still relevant as Max and Nidhi test their components against each other during the sprint.

Still, not every change to the API contract is a breaking change. We have to identify which of the API design changes are breaking and which aren't, how we can deal with them, and what are the implications for the development process. There is a choice between releasing new API versions or trying to evolve our API in a way that avoids breaking changes. We will investigate these options in detail in chapter 21.

## 16.5 Summary

- The PetSitter application goes into its next development sprint. The objective of this sprint and the third part of this book is to include feature requests from initial users and improve the API's developer experience in preparation of additional API consumers.
- There are two new functional requirements. The first is the support for different pets and multiple pets per job. The second is the inclusion of filters and pagination for finding jobs. To support these requirements, we reviewed all user stories, changed one of them and added two new ones. To support different pets, we need modify the concepts in the domain model, which we'll do in chapter 17. We already added two actions to the *job* concept in the domain model to support filters and pagination, a topic which we'll discuss further in chapter 18.
- Developer experience (DX) describes the user experience of a developer working with an API or another developer tool. Good DX helps solving problems with an API integration faster or avoids them from occurring in the first place. The more developers consume an API, the more it pays off to invest in DX. The main ingredients are the API design, API documentation, and developer support. For this book, the focus is on API design (obviously).
- To improve the developer experience prior to sharing the API with the mobile developers building on top of it, the PetSitter team wants to look at error handling, input validation, and versioning. Each of these topics has its own chapter in this book. We'll cover error handling in chapter 19, input validation in chapter 20, and versioning (or how to avoid it) in chapter 21.

# 17

## *Versioning an API and handling breaking changes*

### This chapter covers

- Updating the domain model with new concepts
- Using subtypes and inheritance in the domain model
- Creating composite JSON schemas from the domain model

In the last chapter, José and his team identified new requirements from the initial tests and planned a new development sprint with the goal of sharing the API with an external team for mobile app development as well as working towards making the API ready for wider release. One functional requirement to tackle as part of the sprint is the support of multiple pets per job and also pets other than dogs. The requirement implied changes to the domain model.

By going through the user stories, the PetSitter team realized that many parts of the domain model and, thus, the API description, can remain the same. The affected segment of the model is the *dog* concept and its relationship with the *job*, which we'll now update.

In this chapter, we consider different approaches for changing the *dog* concept to represent different pets in the domain model and the OpenAPI definition. We look at cats and dogs for now, but the goal is finding a system that is extensible for additional species. We learn about polymorphism in domain models and the OpenAPI composition keywords and apply them. At the end, by doing that, we'll have an updated domain model and OpenAPI description that help implementing the new functional requirements.

To show you where we're going, figure 17.1 contains the final domain model that we'll have built at the end. We've highlighted the segment that contains the focus area of the domain model that we're working on during part three of this book.

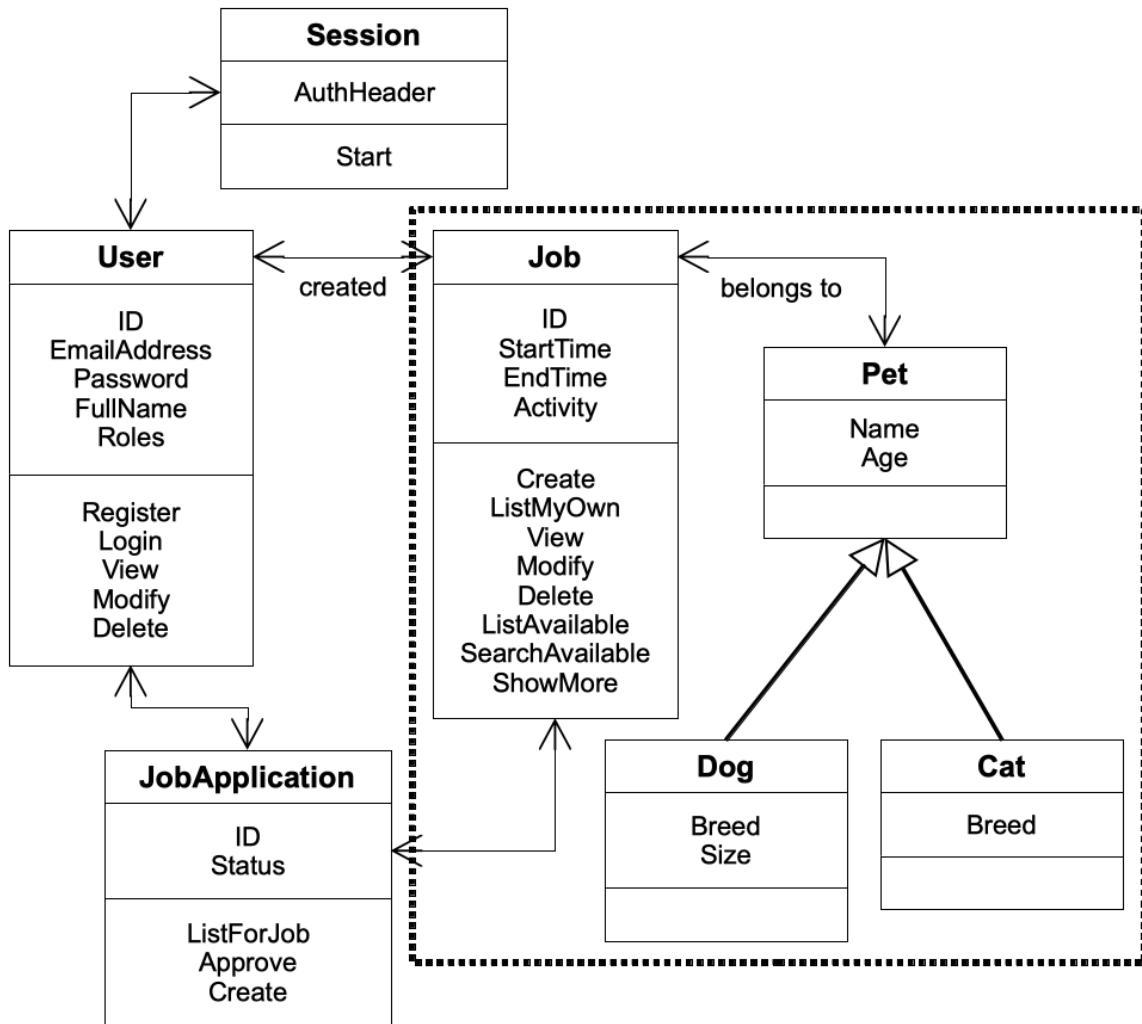
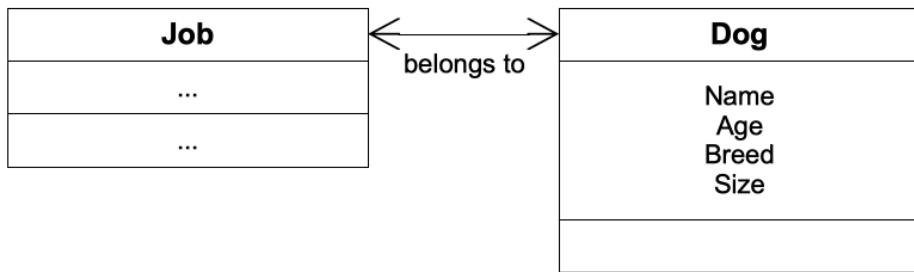


Figure 17.1 PetSitter Final Domain Model

## 17.1 The problem

Our objective is to find a solution to represent other pets in the domain model and, subsequently, in the JSON schemas in our OpenAPI description. The focus is on dogs and cats for now, but we should also consider support for other types, either now or in the future. In this chapter, we'll explore different approaches.

Before we can find the solution, let's quickly have a look at the *dog* concept in the current domain model. It has four attributes - Name, Age, Breed, and Size, - no actions, and an associative relationship with the *job* concept, as depicted in figure 17.2.

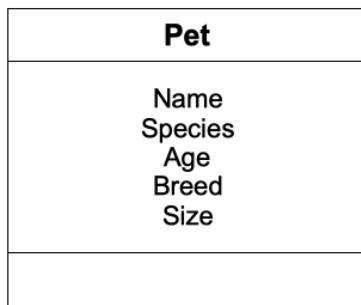


**Figure 17.2 PetSitter Job and Dog Model**

As José eventually wants to support any pet in PetSitter, we could drop the *dog* concept and replace it with a more generalized *pet* concept. This first idea is in line with the way we formulated the new requirement and it seems very straightforward, because when we just modify or swap out one concept, the whole domain model doesn't undergo drastic changes. That is likely to apply for the implementation as well, won't it? Let's try it.

The basis for the new *pet* concept is the existing *dog*. If we only used the existing attributes, however, there is no way to distinguish the species of the animal. Granted, we could use the breed attribute for that, but, even without diving too deep into biology here, it's safe to say there's a huge gap between a different species and different breeds of the same species. To be more specific and reflect the real world in our domain model, we can add a dedicated attribute for the species. Now, let's look at the full attribute list taken from the existing dog model with the new extension:

- Name
- Species (e.g., dog)
- Age (in years)
- Breed
- Size



**Figure 17.3 Proposed PetSitter Pet Model**

To describe a dog now, we would use the pet model, write "Dog" in to the species attribute, and use all other attribute as we previously did. Done! Let's describe a cat next. First, we write "Cat"

in the species attribute. Quite obviously, cats have a name and an age. There are also different cat breeds, but way less than dog breeds, around 40 to 70 depending on who you ask, compared to over 450 dog breeds. The difference does not really matter if the "Breed" attribute is a free-text field, but if we wanted to build some kind of advanced taxonomy of breeds at some point in the future, it would very likely look quite different for dogs and cats. Related to this, dog breeds come in an enormous variety of sizes, whereas even different cat breeds have roughly the same size. The "Size" attribute would not be required or, again, if we want to give it a specific data type, it would be very different.

And that's just cats and dogs. You can easily imagine other types of animals that require a different set of attributes. Some species bring out very picky eaters, so those need a food preference attribute, whereas others will just eat anything you throw at them. Does that mean we have to add that attribute to our *pet* model even if we often don't need it? If you let your imagination run wild and add all the attributes you could think of for various pets, the number of attributes would blow up, and most of them would be irrelevant. The point we're trying to make is that trying to put all possible attributes into the single *pet* concept seems less than optimal. So, do we have other options?

We can, of course, add every type of pet as its own concept in the domain model. To stick with two species for now, let's say we have a *dog* model and a *cat* model. If we do that, we can modify each concept separately and adjust it with our growing understanding of different pets and their important attributes we need within the PetSitter context. Suppose we do that, we end up with a new domain model with five concepts: *user*, *job*, *job application*, *cat*, and *dog*. In figure 17.4, we show this domain model in which we only added the "Size" attribute for dogs but not for cats.

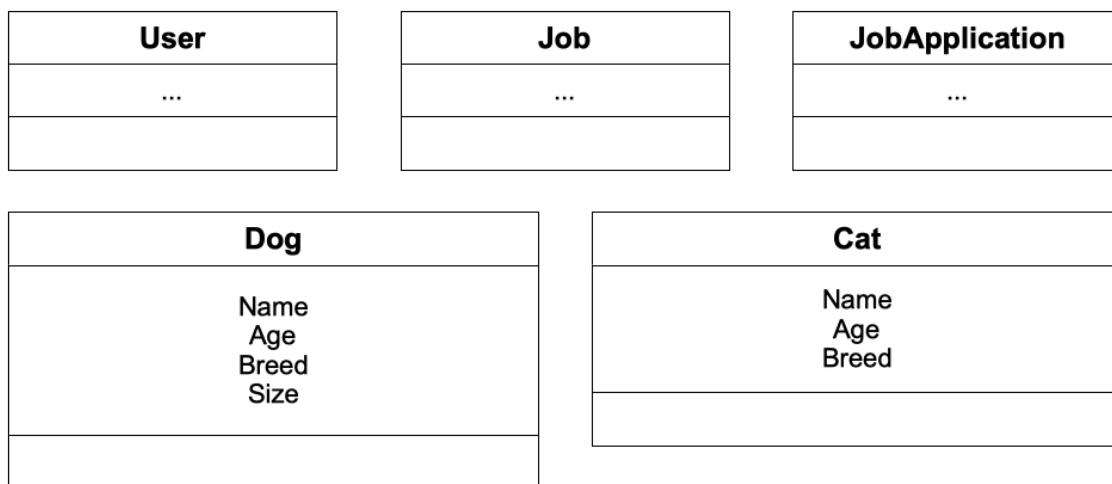


Figure 17.4 Full PetSitter Model with Cat and Dog

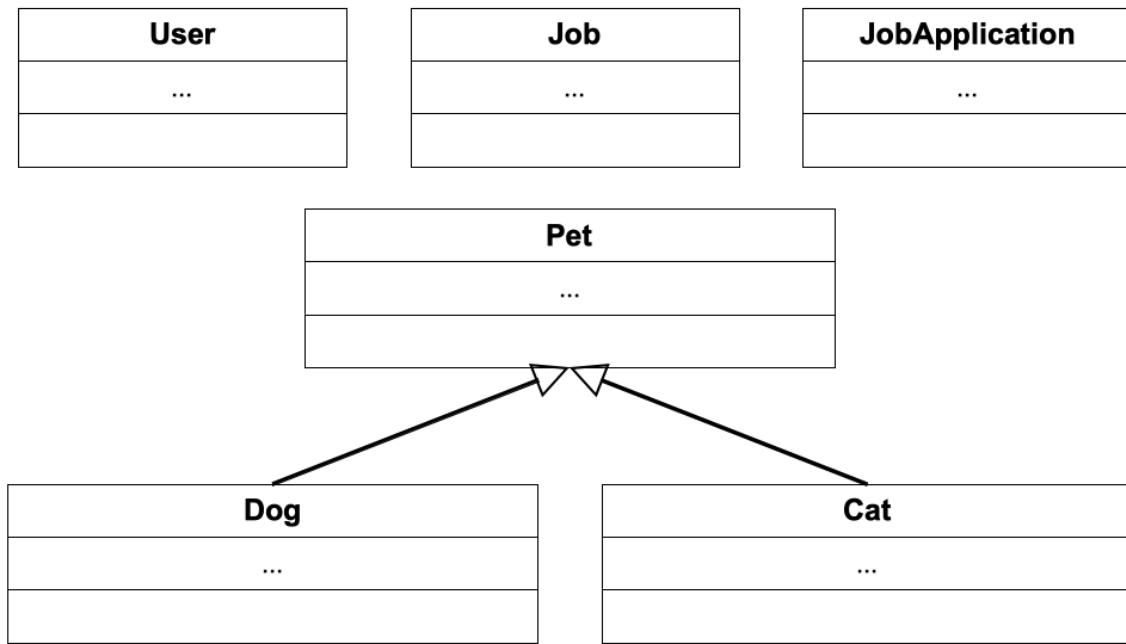
Looking at the domain model with five concepts, we lost one piece of information compared to the *pet* model we suggested before. From our requirements we can assume that *cat* will be used similarly to *dog*, as both are pets and the PetSitter application treats them as such. The information may become apparent as we draw similar relationships for *cat*, *dog*, and any other pet we add, i.e., a relationship from *job* to *cat* in the same way as the relationship between *job* and *dog*, as jobs can have both dogs and cats.

Looking at the terms "pet", "dog", and "cat" and their usage in PetSitter, we can say that every dog and every cat can also be considered to be a pet. It would be useful to express that we have a general and more specific concepts in our domain model, i.e., that every *cat* and *dog* is also a *pet*. The problem remains: how can we put that information into our domain model and the API definition?!

## 17.2 Polymorphism and inheritance in domain models

Let's talk about polymorphism or, more specifically, subtype polymorphism, also known as just subtyping. You may already know these concepts from object-oriented programming. However, even if you're unaware of the concepts, don't let those fancy words scare you! We'll explain what they mean, of course, and how they help us with understanding *pet*, *dog*, and *cat* in our domain model.

A subtype is a concept that is more specific than another. If you look from the other direction, a supertype is a concept that is more generic than another. You can think of *pet* as the supertype and *dog* and *cat* as subtypes. Every *dog* is a *pet*, but not every *pet* is a *dog*. It might, for example, be a *cat* instead. The implication, that the name polymorphism alludes to, is that you can have a reference to the supertype and substitute it with any subtype. We'll see later how that is beneficial when we look at relationships, but it's already becoming clear that using subtyping might be a great idea to make our understanding of cats and dogs as pets explicit. If we apply it, we end up with a new domain model with six concepts: *user*, *job*, *job application*, *pet*, *cat*, and *dog*, with the added information that *cat* and *dog* are specific subtypes of the generic *pet* concept.



**Figure 17.5 Full PetSitter Model with Subtyping**

In figure 17.5, you can see the updated domain model with *pet* as the supertype and *cat* and *dog* as its subtypes. Following the style and conventions of UML (Unified Modeling Language) class diagrams, an arrow with an empty triangle on the end of the supertype indicates a subtyping relationship, which is different from what we previously discussed, namely associative relationships between different concepts (as in, the *dog* belongs to the *job*).

What about the attributes, which we've conveniently left out of figure 17.5? Before we discuss them, let's throw another subtyping-related term into the mix: inheritance. With subtyping, we say that a subtype inherits from its supertype, which means that subtypes have all attributes and behavior of their supertype. Of course, they can add their own attributes and behavior, too. For our domain model that includes inheritance, it means the following:

- For attributes that every pet has, it is sufficient to add them to the *pet* concept. We do not need to explicitly add them to *cat* or *dog*.
- For attributes that only cats or dogs have, we can add them to *cat* or *dog*, respectively. Of course, that also applies for any type of animal that PetSitter supports in the future.

Based on our initial thoughts about the attributes of cats and dogs, we'll make the following attributes common attributes of the pet concept:

- Name
- Age

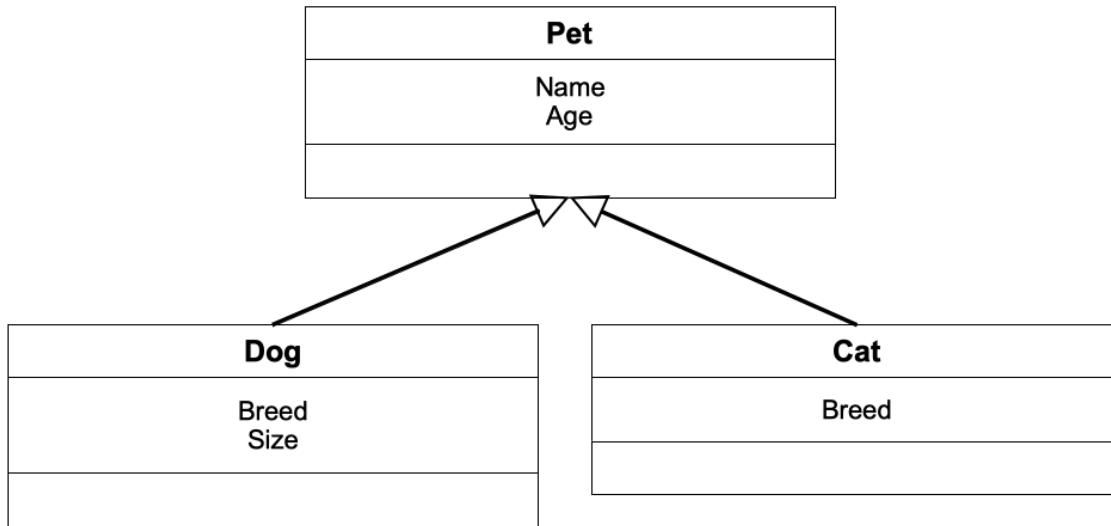
Then, we add the following attributes to dogs:

- Breed
- Size

Finally, we add the following attribute to cats:

- Breed

Why didn't we make breed a common attribute? Well, while these attributes have the same name, we already mentioned that we may represent them differently in the system later on. Also, there's always a chance that we add a type of animal later which is its own concept in the domain model and doesn't have or need the separation into different breeds.



**Figure 17.6 PetSitter Pet Model with Subtypes Dog and Cat**

Alright, we have supertypes and subtypes in our domain model now. Due to the inherent idea of replaceability in polymorphism, we can draw a relationship from the *job* concept to our new *pet* concept and immediately know that jobs can have dogs and cats, without having to draw a relationship from *job* to *dog* and *cat*. Our full domain model with all concepts, even those we didn't touch in this chapter, was already shown in figure 17.1. The one thing that the diagram doesn't show you yet is that a job can now have multiple pets instead of just one, the second part of the functional requirement we wanted to tackle here. However, to be fair, the old diagram didn't show that there could only be one dog, either. We used a very high level visualization without cardinalities that indicate how many instances of one concept are associated with another. Although we don't see it here, we obviously have to keep it in mind as we redesign the schemas in the OpenAPI description later in this chapter.

Now that we have a full updated domain model, the next step will be to express the modifications in the OpenAPI definition for the PetSitter API.

## 17.3 Updating the schemas

In chapter 5, you learned about inline schemas as a way to describe the inputs and outputs of your API operations. In chapter 10, you learned about reusable schemas to express common data structures and add them to multiple API requests or responses using the `$ref` keyword. By following the API design approach, the concepts in the domain model became the common schemas in the API definition, and those became the request and response structures in the API operations.

As a little reminder, let's have a look at the existing Dog schema, and how it appears in the OpenAPI definition. Schemas for domain model concepts typically have the type `object` and use the `properties` keyword to list all the properties (i.e., attributes) that the object can have:

### Listing 17.1 Current PetSitter OpenAPI Dog schema

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Dog:
      type: object
      properties:
        name:
          type: string
        age:
          type: integer
        breed:
          type: string
        size:
          type: string
```

We'll scrap this schema from the OpenAPI and start creating the new Pet, Dog, and Cat schemas based on our updated domain model concepts next. But wait! Since we're removing the existing Dog schema, we should look at any references to that schema, by searching for `$ref: '#/components/schemas/Dog'`. We find one such reference in the Job schema, expressing our associative relationship between *job* and *dog* in the domain model:

## Listing 17.2 Current PetSitter OpenAPI Job schema

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Job:
      type: object
      properties:
        id:
          type: integer
        creator_user_id:
          type: integer
        start_time:
          type: string
        end_time:
          type: string
        activity:
          type: string
        dog:
          $ref: '#/components/schemas/Dog'
```

It's likely that we have to modify the Job schema, considering that it has a property named `dog` and we want to support other pet types and also multiple pets per job. We'll skip the Job schema for now and come back to it later. For now, let's focus on the Pet, Dog, and Cat schemas.

### 17.3.1 The Pet schema

According to our new domain model, we need to have two attributes on that Pet schema - Name and Age -, which are a subset of the original dog attributes.

**Table 17.1 The Pet fields and their types**

Field	Type
name	string
age	integer

Here it is in OpenAPI:

## Listing 17.3 PetSitter OpenAPI Pet schema

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Pet:
      type: object
      properties:
        name:
          type: string
        age:
          type: integer
```

### 17.3.2 The Dog schema

The dog schema has the remainder of the original dog attributes, Breed and Size:

**Table 17.2 The Dog fields and their types**

Field	Type
breed	string
size	string

#### Listing 17.4 PetSitter OpenAPI Dog schema

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Dog:
      type: object
      properties:
        breed:
          type: string
        size:
          type: string
```

### 17.3.3 The Cat schema

The cat schema has only the Breed attribute:

**Table 17.3 The Cat fields and their types**

Field	Type
breed	string

#### Listing 17.5 PetSitter OpenAPI Cat schema

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Cat:
      type: object
      properties:
        breed:
          type: string
```

Up until now, we only used the elements of OpenAPI that we learned about throughout the second part of this book. However, the inheritance relationship between Pet and Dog as well as Pet and Cat is missing from the schemas. Expressing it will be our next task, and we can't do it with the material we know. Hence, it's time to learn new OpenAPI keywords.

## 17.4 Polymorphism and inheritance in OpenAPI

There are four relevant OpenAPI keywords for inheritance, also known as composition keywords. They are `allOf`, `oneOf`, `anyOf`, and `not`. Let's look at their definitions:

- `allOf`: Indicates that a schema is a composition of multiple other schemas, which means that it has all the attributes from those schemas. If we checked a JSON object against a schema with `allOf`, it would only be valid if it passed the check against all schemas.
- `oneOf`: Indicates that a schema is one of multiple alternative schemas, which means that it has the attributes of exactly one of those schemas. If we check a JSON object against a schema with `oneOf`, it would only be valid if it passed the check against just one of those schemas, not multiple.
- `anyOf`: Indicates that a schema is a combination of multiple schemas, which means it can have attributes from any of those schemas. If we checked a JSON object against a schema with `anyOf`, it would be valid as soon as it passed the check against at least one of the others.
- `not`: Indicates that a schema is not another schema. If we checked a JSON object against a schema with `not`, it would only be valid if it failed the check against the other schema.

It's possible to draw analogies that help in understanding the composition keywords `allOf`, `oneOf`, and `anyOf`, and the difference between them. Let's look at them through the lens of set theory, as visualized in figure 17.7, and through logic operators:

- `allOf`: Indicates an *intersection* of multiple schemas, or a logical AND.
- `anyOf`: Indicates a *union* of multiple schemas, or a logical OR.
- `oneOf`: Indicates a *disjunctive union* (also called *symmetric difference*) of multiple schemas, or a logical XOR.

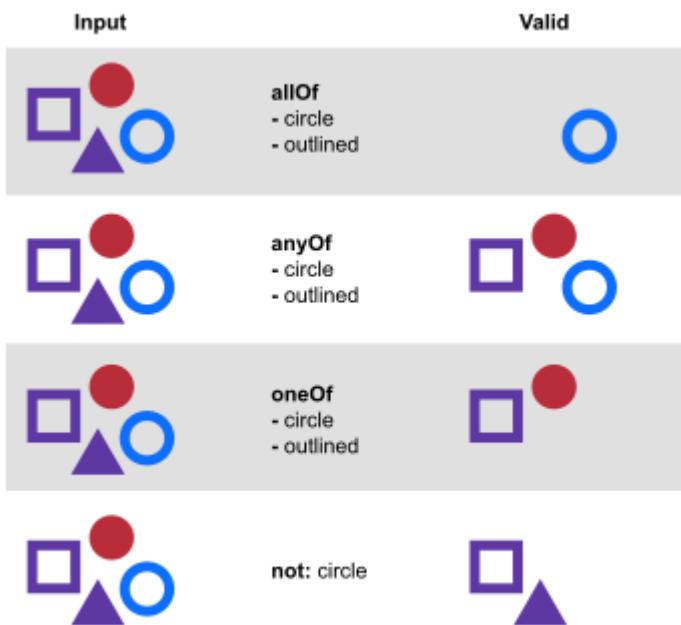


Figure 17.7 OpenAPI Composition Keywords

Inside the YAML code of our OpenAPI description, the keywords are used as first-level elements inside a schema and point to an array of other schemas (except for `not`, which points to a single schema). These other schemas can be both references within in our OpenAPI file or inline schemas. Here is how that looks in a hypothetical example:

### Listing 17.6 OpenAPI Composition Example

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Schema1:
      anyOf: ①
        - $ref: '#/components/schemas/Schema2' ②
        - type: object ③
          properties:
            #...
```

- ① Keyword inside `Schema1`.
- ② Schema reference.
- ③ Inline schema.

Now that we have a first idea of these keywords, let's get back to PetSitter. The fact that we want to express is that a pet can either be a dog or a cat, or the inverse fact that dogs and cats are also pets. Which of the keywords could help us?

- With `allOf`, we could say that a dog is a composition of all the pet properties and the specific dog properties, which would be correct.
- With `oneOf`, we could say that a pet should either be represented by the cat schema or the dog schema, which is the also correct and indicates the reverse of the previous statement.
- With `anyOf`, we could say something along the lines that pets have any combination of cat and dog properties, which doesn't make sense (unless you are a scientist working on genetic modification of pets).
- With `not`, we could say, for example, that a dog is not a cat and vice versa. While that is technically true, it doesn't help us express the pet-to-dog and pet-to-cat relationships.

Considering that we can use two different keywords and we have two opposite ways of expressing the fact we want to include in our schemas, we may have different options of designing our schemas. Let's take a look at two different approaches:

- Composition inside the Dog/Cat schemas
- Composition inside the Pet schema

The first option attempts to include the generic Pet attributes into Dog (and Cat, but we'll only walk through the Dog example). The second option includes the choice of either Dog or Cat attributes into the Pet schema. Let's start looking at the first.

### 17.4.1 Composition inside the Dog/Cat schemas

Using `allof`, we can convert our Dog schema into a composite schema, which includes its own properties (as an inline schema) and a reference to the Pet schema we created earlier. Here's how that looks like:

#### Listing 17.7 PetSitter OpenAPI Dog schema, first option

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Dog:
      allof: ①
        - $ref: '#/components/schemas/Pet' ②
        - type: object ③
          properties:
            breed:
              type: string
            size:
              type: string
```

- ① Keyword `allof` in Dog.
- ② Reference to the Pet schema.
- ③ Inline schema with Dog properties.

We can do the same with the Cat schema, of course. For every additional pet that we want to support, we can create a schema and have it inherit the Pet attributes by making it a composition of the more generic Pet schema and the respective custom attributes. Great! So far, the Pet schema itself remains unmodified, as its sole purpose is to provide the common superclass attributes.

## REFERENCING FROM JOBS

Now, remember how we skipped the Job schema earlier. This is a good time to come back to it. As a reminder, we want to support various types of pets and also multiple pets per job. We can change the property name from `dog` to `pets` to express both. Since we want to support multiple pets, the property cannot be a reference to a single type but it has to be an array. As you may remember, for `array` we can specify the type of individual entries in the array using the `items` keyword. And we have a Pet schema now, so we can reference it. Seems solid! Here's our updated Job schema in OpenAPI:

## Listing 17.8 PetSitter OpenAPI Job schema

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Job:
      type: object
      properties:
        id:
          type: integer
        creator_user_id:
          type: integer
        start_time:
          type: string
        end_time:
          type: string
        activity:
          type: string
        pets: ①
          type: array ②
          items:
            $ref: '#/components/schemas/Pet' ③
```

- ① New property name pets instead of dog.
- ② Array to support multiple pets.
- ③ Changed reference from Dog to Pet.

Does that mean we're done? Sadly, no! Our first approach only expresses one direction of our original statement about the relationship between the generic concept *pet* and specific concepts like *cat* and *dog*. The Dog and Cat schemas have a reference to Pet, but Pet doesn't "know" its subclasses. Due to this, we have to mention all types of pets here. We can do that with the `oneOf` keyword inside `items` and then placing a reference to each pet, like this:

## Listing 17.9 PetSitter OpenAPI Job schema, first option with all pets

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Job:
      type: object
      properties:
        #...
        pets: ①
          type: array ②
          items:
            oneOf: ③
              - $ref: '#/components/schemas/Dog' ④
              - $ref: '#/components/schemas/Cat' ⑤
```

- ① New property name pets instead of dog.
- ② Array to support multiple pets.

- ③ Indicates that items can be one of multiple schemas.
- ④ Reference to Dog schema.
- ⑤ Reference to Cat schema.

With the approach we've chosen, it will be necessary to edit the Job schema every time we add a new pet. That doesn't seem an optimal representation of the domain model where the inheritance between *pet* and its subtypes keeps everything inside these concepts alone and *job* has only a single associative relationship with *pet*. Maybe we should try approaching it from a different angle?!

### 17.4.2 Composition inside the Pet schema

Let's revisit the reference from the Job schema to the Pet schema that we created a little earlier. Here's the relevant part again:

#### Listing 17.10 PetSitter OpenAPI Job schema (excerpt)

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Job:
      type: object
      properties:
        # ...
        pets:
          type: array
          items:
            $ref: '#/components/schemas/Pet'
```

It looks like this is a great way to make a reference from the Job schema to the Pet schema, just as it exists in the domain model, so is there a way to design the Pet, Cat and Dog schemas to make this work? Let's try!

Before looking at our composition keywords, we had set up the Pet, Dog, and Cat schemas with their respective properties. Then, for the first option, we decided to change the Dog and Cat schemas to include a reference to the Pet schema to include the common properties. We can do it in reverse, and reference the Dog and Cat schemas from the Pet schema instead. It means that the Dog and Cat schemas remain unchanged. Instead, we design the Pet schema to express that the following two must apply to each pet:

- The pet should have common pet attributes.
- The pet can additionally have the attributes of exactly one type of pet (e.g. cat or dog).

To put this in OpenAPI, we need to formulate it as a nested composition. The outer composition is an `allOf` of two things:

- An inline `object` with the common properties for all pets.
- A `oneOf` with references to all schemas (the inner composition).

Here's how that looks if we put it into OpenAPI:

#### Listing 17.11 PetSitter OpenAPI Pet schema, second option

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Pet:
      allOf: ①
        - type: object ②
          properties:
            name:
              type: string
            age:
              type: integer
        - oneOf: ③
          - $ref: '#/components/schemas/Cat' ④
          - $ref: '#/components/schemas/Dog' ⑤
```

- ① Outer composition with `allOf`.
- ② Inline `object` with common pet properties.
- ③ Inner composition with `oneOf`.
- ④ Reference to Dog schema.
- ⑤ Reference to Cat schema.

Both options that we discussed would work for PetSitter. The second option is closer to the domain model since there are no changes within Job when we add new pets, and we have all the complex composition logic in a single place, encapsulated in the Pet schema. On the other hand, the first option doesn't need nested composition, so it has a slightly easier structure. We led you through two different options to teach you the composition keywords in a different context and highlight that there are different possibilities to express something similar. Our PetSitter team decides to follow through with the second option, though.

## 17.5 Adding discriminators in OpenAPI

Time for a quiz! Take a look at the following JSON object and tell me whether Fluffy is a cat or a dog:

```
{
  "name": "Fluffy",
  "age" : 5,
  "breed" : "Border Collie",
  "size" : "50 cm"
}
```

You probably guessed correctly that Fluffy is a dog. How did you find this out? Maybe you

heard of Border Collies before and have the common knowledge that they are a dog breed. Fair enough, but let's assume you're a machine that knows nothing about dogs in the real world and only knows about the domain model and the JSON schemas. Within these constraints, you could still say that Fluffy is a dog because the JSON object has a `size` attribute and we only specified that property for dogs.

Now, imagine you're a programmer who's tasked with building an algorithm that tells you the species of the pet just by looking at the incoming JSON object and the OpenAPI description. You would have to check all properties against all schemas to make the distinction. That seems overly complicated, doesn't it?

Remember that our first idea in the problem section was adding a "Species" attribute to the *pet* concept. Maybe that wasn't such a bad idea after all? We can add the property to our Pet schema to have a clear indicator:

#### Listing 17.12 PetSitter OpenAPI Pet schema, species property

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Pet:
      allOf:
        - type: object
          properties:
            name:
              type: string
            species: ①
              type: string
            age:
              type: integer
        - oneOf:
          - $ref: '#/components/schemas/Cat'
          - $ref: '#/components/schemas/Dog'
```

- ① New property added.

Okay, this seems better, but OpenAPI knows nothing about the semantics of the `species` property. Hence, it would technically be valid to create the following object:

```
{
  "name" : "Fluffy",
  "species" : "Cat",
  "age" : 5,
  "breed" : "Border Collie",
  "size" : "50 cm"
}
```

Luckily the OpenAPI authors added an additional keyword to the specification that helps in this type of situation. It's called `discriminator` and its purpose is to define a property whose value

indicates a schema to select. At the time of writing, discriminators are not widely used yet and some OpenAPI tools may not support them, but we expect support to increase so it's still useful to add them.

You need to add the `discriminator` keyword next to the `oneof` keyword on the same level in the YAML file. The prerequisite for it is that all the entries inside that `oneof` are references and not inline schemas, because they need to have a name that identifies them, and inline schemas don't have those. Inside, the `discriminator` definition contains values for two other keywords, `propertyName` and `mapping`:

- With `propertyName` you specify the name of the property that points to the respective schema. It must be a `string` property that exists in each of the schemas. We'll show you shortly how that is done.
- The `mapping` describes which value of that property corresponds to which referenced schema. For example, it can connect the string "Cat" with the Cat schema. There is an implicit mapping that automatically connects the string with a schema whose name is identical to the value of the property in a JSON object (e.g., "Cat" with the Cat schema, "Dog" with the Dog schema), but we recommend making things explicit.

There's another important caveat to consider with the nested structure that we designed. We just added the `species` attributes to the inline schema with the common pet attributes. The `discriminator` however, doesn't belong to the outer `allOf`, it belongs to the inner `oneOf`. And, according to the OpenAPI specification, the property used as `discriminator` must exist individually in every schema that you reference in the `oneOf`. Hence, we can no longer have the `species` attribute at the point where we added it but have to move it into the Cat and Dog schemas. The necessity to add it to each schema separately is the price we pay for the ability to use the `discriminator` keyword.

Let's take a look at the updated Pet schema:

### Listing 17.13 PetSitter OpenAPI Pet schema with discriminator

```

openapi: 3.0.0
#...
components:
  schemas:
    #...
    Pet:
      allOf:
        - type: object
          properties: ①
            name:
              type: string
            age:
              type: integer
        - oneOf:
            - $ref: '#/components/schemas/Cat'
            - $ref: '#/components/schemas/Dog'
          discriminator: ②
            propertyName: species ③
          mapping: ④
            Cat: '#/components/schemas/Cat'
            Dog: '#/components/schemas/Dog'

```

- ① Species removed from inline schema properties.
- ② Added discriminator next to oneOf.
- ③ The property name is species.
- ④ Mapping Cat/Dog strings to respective schemas.

As mentioned before, we have to add the `species` property to each of the schemas between which we discriminate, or, in other words, those listed in the `oneOf` that discriminator belongs to. On top of that, we have to make the property required. So far, we haven't designated any property in a JSON object as required, so let's quickly introduce the `required` keyword before moving on.

By default, properties JSON objects are optional. To make them mandatory, you can use the `required` keyword to list a number of properties that a JSON object must have in order to be valid against the schema. At this point, we'll add the `required` keyword to Cat and Dog and make only the `species` attribute required. We'll get back to a more thorough discussion of the `required` keyword, its implications, and other places to use it in our OpenAPI description, in chapter 20.

Here's the updated Cat and Dog schemas:

### Listing 17.14 PetSitter OpenAPI Dog and Cat schemas, with species

```
openapi: 3.0.0
#...
components:
  schemas:
    #...
    Dog:
      type: object
      properties:
        species: ①
          type: string
        breed:
          type: string
        size:
          type: string
      required:
        - species ③
    Cat:
      type: object
      properties:
        species: ④
          type: string
        breed:
          type: string
      required:
        - species ⑤
        - species ⑥
```

- ① New species property for dogs.
- ② List of required properties for dogs.
- ③ The property `species` is required.
- ④ New species property for cats.
- ⑤ List of required properties for cats.
- ⑥ The property `species` is required.

Okay, let's recap. We've created new Pet, Cat, and Dog schemas to replace the previous Dog schema. Our new Pet schema contains the common attributes for all pets and references the list of pet types, currently Cat and Dog. Each pet type has species-specific properties as well as the `species` property that can be used to clearly distinguish between them. On the Pet schema, we made this `species` property a `discriminator` to assist in finding the right schema for validation.

The new domain model also provides a blueprint for adding new types of pets. You can create a new domain model concept for the pet with custom attributes and make it a subtype of `pet`. In the OpenAPI file, you create a JSON schema for that pet type with the same attributes as properties. Then, you can update the Pet schema and add a reference to the new schema in the `oneOf` segment as well as the `mapping` for the `discriminator`. No other parts of the domain model or the API description require changes. Sounds good! Our work in the domain model and the OpenAPI description is done.

Our PetSitter team can run through the API changes and then, as before, Nidhi can code the backend and Max can code the frontend. We won't cover implementation in this book, though. Apart from this new feature, we have additional changes in this development sprint, which we'll go through in the next chapters.

## 17.6 Summary

- With polymorphism it is possible to add a generic concept, the supertype, to a domain model, and then describe various more specific concepts as subtypes. Every subtype inherits the attributes from the supertype and can add their own. PetSitter's updated domain model includes *pet* as the supertype and *dog* and *cat* as specific subtypes.
- We also changed the relationship between *job* and *dog* in the domain model to be between *job* and *pet*. Thanks to the polymorphism, the *pet* can be replaced with any subtype, so we don't have to draw arrows indicating relationships from *job* to *dog*, *cat*, or any other subtype we may add later.
- The OpenAPI specification includes the composition keywords `oneOf`, `anyOf`, `allOf`, and `not`. They can be used individually or in combination to express polymorphism, and, more generally, describe complex schemas as well as relationships between different schemas.
- Instead of the existing Dog schema, we added new Pet, Dog, and Cat schemas to our OpenAPI definition. We looked at two different alternatives for using composition keywords to implement our domain model. Eventually, we decided to make the Pet schema a composition of the common pet properties and include references to the Cat and Dog schemas for specific attributes. With the `discriminator` keyword, we designated the `species` property as an indicator of the subtype.

# 18

## *Scaling collection endpoints with filters and pagination*

### This chapter covers

- Designing filters, pagination, and sorting for APIs
- Enhancing the PetSitter OpenAPI definition with these features

As the PetSitter application grows, there will eventually be a lot of jobs posted in the system at the same time. It means that pet sitters will have a hard time going through all the job postings to find those they are interested in. Also, the API response for listing the jobs can get quite large. The PetSitter team realized this during their sprint planning in chapter 16. At that time, they decided to implement filters and pagination to solve the issue. While discussing these we're also adding a third, related topic: sorting.

Before we start looking at API design, though, let's make sure we're on the same page regarding the meanings of the terms filters, pagination, and sorting:

- **Filters** are a way to add search criteria that identify a subset from a collection of resources. If the API consumer sends the filters with their API request, the API includes only matching resources into its response.
- **Pagination** divides the collection into chunks. Any API call returns only the first chunk in the response. When the API consumers wants to see additional resources, they have to send another request and ask for the next chunk of data.
- **Sorting** specifies the order in which the API lists resources in a collection. Their default order may not be what the API consumer needs, so the API can give them the option to change it.

You can see these definitions visualized in figure 18.1 as well. The three options should not be mutually exclusive, so our API design must allow paginating and sorting both filtered and non-filtered lists.

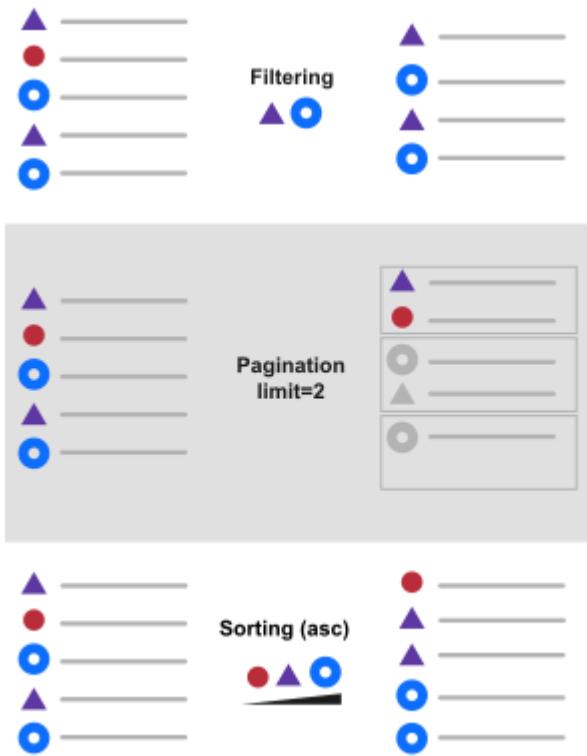


Figure 18.1 Filters, Pagination, and Sorting

In this chapter, we'll look at filters, pagination, and sorting in general as well as their specific implementation in PetSitter. Our approach is to switch between theory and practice throughout. It means we'll introduce each of the three topics and talk about the different possibilities and options we have when we include it in our API design. Then, we'll look at PetSitter, select appropriate options, enhance our API design with parameters covering the respective topic, and finally include them in the OpenAPI description. Then, we'll move on to the next topic.

## 18.1 The problem

Many well-designed APIs follow the CRUD style, and we're focusing on this type of API design in this book. There are other API design paradigms, for example, query-based APIs - you may have heard of GraphQL. For those types of APIs, we have native support for queries, so filters and pagination come somewhat naturally. For RESTful/CRUD-style APIs, we have to include them in our API design. The question is: how do we do that? Following the CRUD concepts from chapter 10, we have the following conditions:

- With filters, pagination, and sorting, we're looking at lists, or collections, of resources. Hence, we are dealing with collection endpoints, not individual resource endpoints.
- No resources are created, updated, or deleted in the process. We're just retrieving data. Therefore, we should use the GET method. Apart from the semantic meaning, using the GET method gives us the benefit of HTTP caching.

So far, we've used collection endpoints to list all resources of a specific type. For example, in PetSitter, we had a "List all" action for the *job* model and turned it into the `GET /jobs` operation. We also had "List my own" action, which we transformed into "List for user" and made it the subresource collection operation `GET /users/{id}/jobs`.

In the updated domain model, we renamed "List all" to "List available" to clarify that it won't return all jobs at once but only the first page. We also added two actions, "Search available" and "Show more". At least so far, every action we listed in the domain model corresponded to an operation in the API. Considering that, we may naively extend the design with additional endpoints:

- `GET /jobs/search` or `GET /search-jobs`?
- `GET /jobs/more` or `GET /more-jobs`?

Adding additional endpoints produces potential namespace clashes, either between the action keyword "search"/"more" and resource identifiers, or with a schema called "search-jobs". These may be theoretical concerns and not applicable in PetSitter, but there's another good reason against adding additional endpoints. If we wanted to fulfill the requirement of combining filters, pagination, and sorting in this way, we couldn't do so, or we would require even more endpoints, such as `GET /jobs/search/more` - or is it `GET /jobs/more/search`?! This path leads us nowhere, it seems.

For filters, we can picture the "List available" action as a special case of the "Search available" action. The unfiltered list of available jobs is a superset of any filtered list. In other words, it's the list where the filters are no filters. We can use the same collection endpoint - the standard collection endpoint for a resource (`/{{schema}s}`) - for both, and add the filters as optional parameters.

Any type of collection can be paginated. By default, we can assume the collection endpoint returns the first page of results. We can then similarly add parameters to indicate that we want to dig deeper into the results.

Finally, for sorting, we can also use parameters for any type of collection endpoint. If those parameters are absent, we apply the default order for resources.

After deciding to not add any new endpoints and use parameters instead, we must decide on the type of parameter. Sending our filter, pagination, or sort parameters in the request body is not an option because GET requests, which we want to use to adhere to CRUD semantics, do not have a request body (if you see some that do, run ...). That leaves two options:

- Query parameters
- HTTP headers

Throughout the course of this book, you've encountered both types of parameters already. While

it's technically possible to use both, you may already be aware that they come with certain RESTful semantics that you've seen in our previous usage of query parameters and HTTP headers. Let's make those explicit now:

- Query parameters are a part of the URL and, thus, an input to the API request. In other words, they help in identifying the resource or resources to access.
- HTTP headers are for meta information that does not identify the resource but adds additional information to the API request, such as authorization.

With these semantics, query parameters are the obvious choice. So, to recap, to implement filters, pagination, and sorting for resource collections in an API, we don't need additional endpoints. Instead, we extend our collection endpoints with parameters. Now, the crucial part is how we design those parameters to be intuitive and consistent.

There are some standards and frameworks for designing APIs that go beyond the basic RESTful principles and CRUD conventions we introduced in this book. Two well-known specifications are JSON:API<sup>10</sup> and OData<sup>11</sup>. They cover a lot of ground but some of their conventions are too heavy for simpler APIs. We will not discuss them in detail but may occasionally refer to them in this chapter. Other sources for the suggestions on parameter design come from famous APIs such as Stripe's and the internal API styleguides from companies who publish theirs.

The OpenAPI specification in its current iteration doesn't prescribe how to design filter, pagination, and sort parameters. It also doesn't let you express the semantics of query parameters, just their syntactical format. For example, you could say that there's a parameter called `sort` that accepts (among others) the string `start_time`, but there's no way to indicate in a machine-readable way that this `start_time` corresponds to the `start_time` property in your response schema. Therefore you have to add human-readable descriptions and additional documentation to explain your developers what the parameters mean. With that said, let's dive into the first area, filters.

## 18.2 Designing filters

We encountered the first filter parameter very early in this book. In chapter 2, we introduced the Farmstall API as a way to get ratings for various farm stalls. In that API, to get a list of public reviews one can use `GET /reviews`. Also, one can filter reviews by their rating using the query parameter `maxRating`. The Farmstall API design already follows some of the standard conventions we introduced in the problem section above: it uses the GET method and a query parameter. And, the parameter remains optional, so we use the same collection endpoint for both filtered and unfiltered requests.

The API design also raises some questions. The schema for reviews contains a numeric `rating` field. Then, there is a query parameter called `maxRating`, which appears related to the `rating` field, and defines an upper bound for ratings. Is that an arbitrary choice, or is there a pattern or

convention that the API introduces? For example, could you use `minRating` to define a lower bound. What if you wanted to get reviews with a specific rating number? Let's take a step back and see how we can best design filters.

First of all, let's consider two types of filters. The primary type of filter we're looking at is a *selection* filter, which means it selects a subset of resources from a collection. In the Farmstall API, that would be all reviews with specific ratings. There's another type of filter, called a *projection* filter, that selects a subset of fields shown for each resource - it doesn't affect which resources the API returns. In other words, a projection allows API consumers to say that they're only interested in some fields for the resources in the collection. For example, imagine you have an API for an online shop with a Customer schema. The Customer schema most likely contains the full postal address of the customer, so that you can ship goods to them. However, an email marketing software as an API consumer would not be interested in those. It just needs name and email address.

### 18.2.1 Projection filters

Projection filters are not as common as selection filters and they are only useful if you have heavy schemas with a lot of fields. Large schemas, however, could be an indication that the domain model concepts are too big and you should break them down into smaller, more specific concepts. We're including projection filters in this chapter for the sake of completeness, but won't implement them in PetSitter.

A common convention for handling projections that many APIs in the wild use as well, is a query parameter called `fields`. The value for the parameter is a comma-separated list of properties from the schema. For example, imagine the Customer schema having `id`, `name`, `email`, and `address` fields. To receive all customers but only their names and emails, the API consumer could request `GET /customers?fields=name,email`. That's all there is to say about projection filters, so let's move back to selection filters, which are slightly more elaborate.

**NOTE**

You may sometimes have fields in your schema that not all of your API consumers can see due to permissions checks. That's also a kind of projection, though unrelated to a filter parameter. While implementing your API, make sure nobody can use projections to access fields they shouldn't see.

### 18.2.2 Selection filters

All selection filters look at a specific field, or property, of a resource. Only resources matching certain conditions for that field should be included in the response. The filter either defines a single acceptable value, or a range of acceptable values. We can formulate ranges in different ways:

- For any data type: an enumeration of acceptable values.
- For numbers: exact matches (`=`), less than (`<`), greater than (`>`), less or equal (`<=`), greater or equal (`>=`), between.
- For strings of text: exact matches, case-insensitive matches, text starting with, text ending with, text containing.

Of course, not every API needs every option from this list. And you don't need to provide the same options everywhere. Remember, we're dealing with CRUD-style APIs, and it's your API design and not a generic query language. Providing too many options can be as detrimental for developer experience, due to the increased complexity, as is not providing enough. In any case, it's important to keep the options in mind, especially considering an evolvable API design. The challenge is mapping these different ranges to query parameters, which are simple key-value pairs.

Many APIs follow a naive and straightforward approach where they give the query parameter the same name as the field itself. To return to the Farmstall example, that would mean that `GET /reviews?rating=3` fetches all reviews where `rating` is *exactly* 3. If most requests ask for exact matches and not ranges, this is the easiest to understand and implement for the API.

When following this approach, there's one thing you have to keep in mind: you're now mixing query parameters having names that correspond to fields from the schema with parameters referencing other things. For example, if you have a `fields` query parameter for projections as mentioned above, you couldn't do filtering on a schema property named `fields`. A better option would be putting a prefix in the query parameter, for example calling it `filter:fields` or `filter[fields]` instead of just `fields`. On the other hand, such collisions are rare. Even popular APIs like Twilio or Stripe mostly use the simple parameter design. We won't show a filter prefix in our examples, but you can always add one if you prefer.

Let's look at the example of Stripe a little closer, because they have an interesting way to support ranges. Assuming there's a field `created` which indicates when a resource was created, they accept the following query parameters:

- `created` for an exact match
- `created.gt` for a greater match
- `created.gte` for a greater-or-equal match
- `created.lt` for a lesser match
- `created.lte` for a lesser-or-equal match

Using suffixes like `gt` and `gte` is an explicit way to provide ranges without ambiguity. With `maxRating` and `minRating`, a developer may wonder whether that's an inclusive or an exclusive minimum or maximum. The suffixes are also applicable to all sorts of fields with ranges. They are not very human-readable, though. We can probably do better. For dates, for example, we could use the following:

- `created_before` for dates before the given value, i.e., a lesser match
- `created_after` for dates after the given value, i.e., greater match

The before/after terminology isn't necessarily the best choice for every data type, so it's more difficult to be consistent here. Another approach we've seen in some APIs is to move the range to the right-hand side, in other words, in the value part of the query parameter. Let's look at an example for that, which might be a redesign of the FarmStall API:

- `rating=eq:5` or `rating=5` for an exact match
- `rating=gt:3` or `rating>3` for a greater match
- `rating=lt:3` or `rating=<3` for a lesser match

There's an advantage to putting the range indicator into the left-hand side, or the key part. Very often an API consumer wants to provide two conditions to define an upper and a lower bound for the filter parameter. For example, all ratings between 2 and 4, or all dates from last week Monday to Friday. Compare the following three approaches:

1. `rating.gte=2&rating.lte=4`
2. `rating=gte:2&rating=lte:4`
3. `rating=gte:2,lte:4`

The first option obviously indicates two filter criteria, and we'd recommend this approach over the others. The second seems wrong since it uses the same key twice. It is technically allowed by the HTTP specification to use the same query parameter more than once, and it's not uncommon, but it doesn't follow our intuitive sense of how key-value pairs work. Finally, the third one uses a complex comma-separated value format that requires additional explanations.

Another potential filter input is an enumeration. With an enumeration, the API consumer provides a set of specific values that they want to accept. For these, it's best practice to use comma-separated values. For example, the following query would return all reviews with ratings 1, 3, and 5:

```
rating=1,3,5
```

With so many different options, it can be difficult for you to make a choice, and it's equally hard for us to give you actionable advice. The most important thing is that you're internally consistent, which means that you must use the same syntactical structure for every parameter and endpoint. Once your API consumers have identified patterns, they will expect them everywhere.

### 18.2.3 Handling nested schemas

Let's assume you have the following data structure in your resources:

```
{
  "name": "John Doe",
  "email": "johndoe@example.com",
```

```

"address": {
  "country": "US",
  "zip_code": "12345",
  "city": "Boomtown"
}
}

```

Using the query parameters `name` and `email` to filter by name and email respectively seems straightforward. However, what about filtering by address? Since the address is a complex data structure with multiple fields, you may want to support each field individually. There are a few naming options to refer to the nested structure. For example, the query parameter for filtering by country could be one of the following:

- `country`
- `address_country`
- `address.country`
- `address[country]`

The first option drops the name `address` entirely, which may lead to namespace clashes. Just imagine you have multiple addresses, such as `billing_address` and `shipping_address`. Therefore we'd strongly advise against it. We'd also advise against using the underscore ("`_`") as it clashes with snake case naming conventions for fields. There's nothing that indicates whether `address_country` is a single or a nested field name.

Choosing either of the other options can be a matter of taste or native support in your implementation framework. The dot notation is common in Javascript and many other object-oriented languages to access nested data structures. On the other hand, for example, PHP natively parses the square bracket style into arrays. Yet again, you're not shipping your backend, you're designing an API that everybody can talk to. Still, the advantage is that it allows you to use the dot notation for other things, such as a suffix for range indicators (e.g, `created.gte`).

Also, starting with specification version 3.0, OpenAPI also provides a very compelling reason for the square bracket style. They support `object` schemas for parameters and the `deepObject` serialization style. The OpenAPI support might turn this into a de-facto standard. We will demonstrate it later in this chapter as we create the parameters for PetSitter. With square bracket style, suffixes should be part of the inner field name and not appear at the end. For example, the parameter should be `address[zip_code.gte]`, not `address[zip_code].gte`.

We should also look at another nested format where we have an array of items inside the resource schema, such as the following contact format with multiple addresses:

```

{
  "name": "John Doe",
  "email": "johndoe@example.com",
  "addresses": [
    {
      "country": "US",
      "zip_code": "12345",
      "city": "Boomtown"
    }
  ]
}

```

```

        "city": "Boomtown"
    },
    {
        "country": "US",
        "zip_code": "54321",
        "city": "Complexity City"
    },
]
}

```

In this case, we can either add some indicator, like `[]`, to point to the array, or we can simplify and ignore the array structure and just use the addresses framework. We'd consider the following all good options for a country filter:

- `addresses.country`
- `addresses[][country]`
- `addresses[country]`

#### 18.2.4 Query languages

Finally, there are APIs that use a single parameter that accepts some sort of query language. For example, the aforementioned API framework OData suggests a parameter named `$filter` for all API calls with filters. The value of this parameter can be a more or less advanced query. A basic query could be, for example:

```
$filter=name eq 'José'
```

Other APIs have such complex filter languages that they break the maximum URL lengths that some clients and servers enforce. Those APIs need to use POST for queries, which violates HTTP semantics and prevents caching. We advise against these constructs and advocate for the simplest query parameter design that you can get away with while still supporting your major use cases.

#### 18.2.5 Special conventions

Before we finish this section, there are two more conventions that we'd like to include. The first brings us back to chapter 10 where we designed the "List my own" endpoint for jobs in PetSitter. At the time, we briefly considered implementing a filter and support `GET /jobs?user_id={id}` before settling on the subresource collection endpoint `GET /users/{id}/jobs`. Whenever a field refers to another schema, as `user_id` does in the Job schema, you should preferably use subresource collection endpoints instead of filters, as that leads to a nicer URL design that reflects the relationships in your domain model. In some cases, however, it may be necessary to support both, especially when there are different fields that API consumers may want to combine. And, obviously subresource collection endpoints don't support ranges. We'll look at an example in the next section as we design filters for PetSitter.

The other convention is the parameter `q` (= question). Sometimes you want to support full-text

search in your API, covering multiple fields. For example, when you have a schema in which first name and last name are two separate fields, you may want to offer search over both. In these cases, support the `q` parameter to search through all fields.

## 18.3 Filters for PetSitter

As we've mentioned in the problem section, we implement filters, pagination, and sorting by adding query parameters to collection endpoints. While reviewing the domain model in chapter 16, the PetSitter team discussed the "List all" action for the `job` model. They renamed it to "List available" and added "Search available" and "Show more" actions to indicate that they want filters and pagination at this point. There are, however, other API endpoints that return collections of resources. Hence, the PetSitter team looks at their existing API design to see if there are other endpoints that could benefit from filtering.

**Table 18.1 PetSitter API operations returning collections**

Schema	Action	API Operation
Job	List available	GET /jobs
Job	List for user	GET /users/{id}/jobs
Job Application	List for job	GET /jobs/{id}/job-applications

As you can see in table 18.1, there are three endpoints. "List available" is the only "root" collection endpoint. The other two are subresource collection endpoints, so they already have a filter built in. Nidhi and Max decide to start with the first, most important endpoint they already identified, and focus solely on that endpoint for the current sprint. They don't think that a single user will create so many jobs and a single job will have so many applications that it's impossible to look at them without filters. As book authors, we agree with their assessment.

### 18.3.1 Finding filter fields

For potential pet sitters, the primary use case when interacting with PetSitter is finding and applying for jobs. Finding jobs in the first place is the crucial part, which is why the "List available" action received a "Search available" counterpart. To determine the fields that could be potential filters, let's have a look at the Job schema again:

## Listing 18.1 PetSitter Job schema

```
Job:
  type: object
  properties:
    id:
      type: integer
    creator_user_id:
      type: integer
    start_time:
      type: string
    end_time:
      type: string
    activity:
      type: string
    pets:
      type: array
      items:
        $ref: '#/components/schemas/Pet'
```

As you remember, the pet or pets that the job is about, is or are an essential part of the job description. That means we can add filters covering the Pet schema as well. Let's have a look at that schema:

## Listing 18.2 PetSitter Pet schema

```
Pet:
  allOf:
    - type: object
      properties:
        name:
          type: string
        age:
          type: integer
    - oneOf:
        - $ref: '#/components/schemas/Cat'
        - $ref: '#/components/schemas/Dog'
  discriminator:
    propertyName: species
    mapping:
      Cat: '#/components/schemas/Cat'
      Dog: '#/components/schemas/Dog'
```

We could go even deeper and look at the Cat and Dog schemas, but we'll stop here. The reason is that the composition adds additional complexity and each pet type has different fields. We'd have to document them all as query parameters for the same endpoint (`GET /jobs`), which not only creates a long list of fields but also breaks the separation of pet species that we intended.

Before we walk through all the fields in the Job and Pet schemas, we will set two general conventions. First, we will not add a general prefix like `filter` in front of the query parameter names. Second, we will use the square bracket style `([])` to access nested fields and ignore array structures. It will allow us defining parameters as objects, too. Also, here's a reminder that we only add selection filters, no projection filters. The PetSitter team believes the current schemas are small enough to include the resources into all responses without reducing their size.

**NOTE** It's a good practice to write down conventions in a style guide that you share with everyone who's collaboration on the API design, for example, in the API definition repository.

Let's get started now and make cases for or against specifying a filter based on the fields:

## ID

IDs are internal identifiers that rarely make sense to users. We already have the resource endpoint `GET /jobs/{id}` to access one job with a specific `id`, which means we don't need a filter for that specific case. And there's no apparent use case for listing multiple selected jobs or a range of identifiers. Hence, we'll not add a filter for `id`.

## CREATOR\_USER\_ID

We already have the subresource collection endpoint `GET /users/{id}/jobs` to list all jobs for a specific user. The current approach to permissions for this endpoint says that it's only for listing one's own jobs. If we wanted pet sitters to look at all the jobs for a specific pet owner, we could grant permissions on that endpoint instead of adding a filter. Hence, no query parameter for `creator_user_id` either.

## START\_TIME

Searching by date is a likely use case. For example, a pet sitter may be free to look after pets only on specific dates or during specific times. Therefore we should add a filter parameter for `start_time`, which needs to support ranges with both upper and lower bounds.

Based on the considerations earlier in this chapter, we should add two query parameters: one for the lower and one for the upper bound. The following two options seem good choices:

- `start_time.lt / start_time.gt` (also maybe `lte/gte` variants)
- `start_time_before / start_time_after`

We'll pick one after gathering all the filters.

## END\_TIME

It may seem that `start_time` is enough for a date filter, but there are certainly use cases where API consumers may want to search for `end_time`, for example if a pet sitter has another appointment and needs to set a boundary when the job ends. Adding the parameter is straightforward and should be analogous to `start_time`, so we'll pick one of the following later:

- `end_time.lt / end_time.gt` (also maybe `lte/gte` variants)
- `end_time_before / end_time_after`

## ACTIVITY

Pet sitters may be interested in specific activities, such as taking dogs for a walk. Therefore, adding a filter for `activity` is useful. In the current version of the PetSitter application, it is a free-text field, which means that pet owners enter some text here instead of selecting from a predefined range of activities.

Due to the free-text nature, we should allow full-text search. For example, if a potential pet sitter enters "walk" as a filter, it should find all of the following activities:

- "walk"
- "walking"
- "dog-walking"
- "take my dog for a walk"

On the other hand, we don't need ranges. There's no semantic value in finding activities that are alphabetically close to "walk". Hence, we can do with a single filter parameter called `activity`.

## PETS[NAME]

Searching by a pet's name doesn't make sense if a pet sitter is looking for jobs in general. If they're searching for a specific pet, the pet's name may not be unique enough. In the latter case, searching by its owner is a better approach, but we've already decided to not add a filter for it in the current sprint and probably extend permissions instead.

## PETS[AGE]

A pet's age may be a useful filter. For example, a pet sitter may specifically want an older pet with the expectation that those are tamer and easier to handle. Providing both an upper and lower bound is also useful, so we need two parameters. Let's collect the options:

- `pets[age.gt]` / `pets[age.lt]`
- `pets[age_below]` / `pets[age_above]`

## PETS[SPECIES]

The `species` attribute isn't part of the Pet schema itself, but as the `discriminator` it is present in all the specific pet schemas, so we can include it in our scope. And filtering by a pet's species could be the most important filter, considering that the PetSitter app project originally started as a dog-walking app. There may be users who are only interested in dogs. On the other hand, some people may be allergic to cats and dogs, but would love to take care of someone's fish tank.

There is a well-defined set of species in the OpenAPI definition, so we don't need full-text search. We also don't need search with alphabetical upper and lower bounds. An enumeration,

however, could be useful. We'll add a query parameter `pets[species]` and allow ranges such as `pets[species]=Cat,Dog`.

### 18.3.2 Adding filters to OpenAPI

In the previous section, we collected the following filters:

- `start_time.lt/.gt` and `start_time_before/_after`
- `end_time.lt/.gt` and `end_time_before/_after`
- `activity`
- `pets[age.gt]/pets[age.lt]` and `pets[age_below]/pets[age_above]`
- `pets[species]`

We still need to finalize the naming for the parameters with boundaries, so let's do that. There are three filter fields, two are dates and one is a number. For all of them, there are prepositions like *before*, *after*, *above*, and *below* that can be attached to create a natural language, human-readable name. They are, however, not indicative of whether they define an inclusive or exclusive maximum or minimum respectively. Suffixes like `lt`, `gt`, `lte`, and `gte` are more specific. We could change the human-readable versions into specific parameters and use, for example, `start_time_at_or_before`. These versions would be quite long, though. After some back and forth discussion, the PetSitter team decides to use the human-readable versions and make them inclusive boundaries without lengthening the parameter name. Instead, they will describe the behavior in the API documentation. Eventually, we'll end up with the following parameters for the PetSitter API:

- `start_time_before`
- `start_time_after`
- `end_time_before`
- `end_time_after`
- `activity`
- `pets[age_below]`
- `pets[age_above]`
- `pets[species]`

Now, how do we add those to our OpenAPI definition? We saw our first query parameter back in chapter 3 in the FarmStall API. Since that was a while back, let's quickly recap the general query parameter format:

### Listing 18.3 Query Parameters in OpenAPI

```
openapi: 3.0.0
#...
paths:
  /resources:
    get:
      description: Description of operation
      parameters: ①
        - name: parameterName
          in: query ②
          description: Description of parameter ③
          required: false ④
          schema: ⑤
            type: number
```

- ① This keyword indicates the parameters for an operation.
- ② The `in` keyword defines the type of parameter, such as `query`.
- ③ Human-readable description of the parameter.
- ④ Indicate whether or not a parameter is required.
- ⑤ The schema describes the data type and constraints.

As we're reusing a collection endpoint that API consumers can call without parameters as well, we want all filter parameters to be optional. In OpenAPI, that's the default behavior for parameters if the `required` keyword is missing, so we can omit `required: false` from our parameter definitions.

Also, as we mentioned earlier in this chapter, OpenAPI has limited capabilities to describe the semantics of query parameter. The fact emphasizes the need for the `description` field where we can explain API consumers how to use the parameter.

Now, let's have a look at the current definition of the "List All Jobs" operation:

### Listing 18.4 PetSitter List All Jobs

```
openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    post:
      #...
    get:
      summary: List All Jobs
      operationId: listAllJobs
    #...
```

As this operation covers two actions in our domain model now, we should change the `summary` and the `operationId` to something more inclusive. Also, we have to add the `parameters` keyword:

## Listing 18.5 PetSitter List/Search Available Jobs

```
openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    post:
      #...
    get:
      summary: List/Search Available Jobs
      operationId: listOrSearchAvailableJobs
      parameters:
        #... ①
      #...
```

- ① This is the place to add filter, pagination, and sorting parameters.

That looks better already. Now let's move on and add our parameters:

### START\_TIME AND END\_TIME

We decided to allow upper and lower bounds for both start and end times. Therefore we have to add four parameters in total, which look quite similar. Their `schema` should have `type: string`. We will add another property, `format: date-time`, to specify that the string has the format of a timestamp. We haven't covered this particular use for the `format` keyword yet, but we'll get back to it in chapter 20 when we improve our schemas. The following listing shows all four parameters and the user-friendly descriptions we chose for them:

## Listing 18.6 PetSitter Job Search Date and Time Filters

```
- name: start_time_before
  in: query
  description: Search jobs starting before this date and time.
  schema:
    type: string
    format: date-time
- name: start_time_after
  in: query
  description: Search jobs starting after this date and time.
  schema:
    type: string
    format: date-time
- name: end_time_before
  in: query
  description: Search jobs ending before this date and time.
  schema:
    type: string
    format: date-time
- name: end_time_after
  in: query
  description: Search jobs ending after this date and time.
  schema:
    type: string
    format: date-time
```

## ACTIVITY

For the activity filter, we need a single query parameter with `string` data type. We will not add further constraints, but we'll add a description that explains how the search works. That's the simplest kind of filter parameter an API could have (from an API design viewpoint):

### Listing 18.7 PetSitter Job Search Activity Filter

```
- name: activity
  in: query
  description: Performs a full-text search for the phrase entered in job activities.
  schema:
    type: string
```

## PETS[AGE] AND PETS[SPECIES]

We'll tackle all the filters that belong to the Pet schema in one step, as we want to demonstrate adding a parameter with an `object` schema. You can create these parameters in the same way as parameters with `string` data types. We can create an `object` with three properties as an inline schema:

### Listing 18.8 PetSitter Job Search Pets Filter

```
- name: pets
  in: query
  description: Searches for pets matching specific criteria.
  schema:
    type: object
    properties:
      age_below:
        type: integer
        description: Return only pets with this age or younger.
      age_above:
        type: integer
        description: Return only pets with this age or older.
      species:
        type: string
        description: Return only pets with this species. Provide multiple species as
        comma-separated values.
```

Apart from adding `type: object`, however, you also have to specify how you want the object serialized into key value pairs. For this purpose, there's the `style` keyword. By default, OpenAPI assumes the `form` style. This style flattens the object, which means that the query parameter is called, for example, `age_below`. If we want to make sure the object converts its properties to, for example, `pets[age_below]`, we have to add `style: deepObject` to it:

### Listing 18.9 PetSitter Job Search Pets Filter (with style)

```

- name: pets
  in: query
  description: Searches for pets matching specific criteria.
  style: deepObject
  schema:
    type: object
    properties:
      #...
  
```

#### 18.3.3 Making a request

Here's an example for a request that combines some of the filters we've added, imagining a potential petsitter who would like to look after a cat some time in July 2022:

```

curl -H "Authorization: {Auth}" "https://petsitter.designapis.com/jobs \
?start_time_after=2022-07-01T00:00:00+00:00 \
&end_time_before=2022-07-31T00:00:00+00:00&pets[species]=Cat"
  
```

The backend receives these parameters, parses them and somehow (depending on the implementation) turns them into a database query. And that is how the result may look like:

```

{
  "items" : [
    {
      "start_time" : "2022-07-02T10:00:00+00",
      "end_time" : "2022-07-04T19:00:00+00",
      "pets" : [
        {
          //...
          "species" : "Cat"
        }
      ],
      //...
    },
    //...
  ]
}
  
```

Let's take a breath, and then move on to the next topic: pagination.

## 18.4 Designing pagination

Pagination is the practice of dividing a long list of results into pages. A lot of times, you can take the word "page" quite literally. You've probably seen websites where you see the first page of results and, in the bottom, there's a list of numbers indicating all the result pages and you can skip to the next and the previous page. On other websites and apps, you may have seen the practice of "infinite scrolling". The first results are displayed and, as you scroll down, the website or app loads additional lists and adds them to the bottom of the results. Both are different user interface approaches, but the underlying API design is almost the same. I'm saying almost, because, as you'll see later in this section, different approaches to user interfaces correspond better to different pagination API designs.

When it comes to pagination, we have to consider the two sides of an API call. On the one hand, we need query parameters to indicate the results that we want to retrieve. On the other hand, we need to extend the response to give some indication where we are in the dataset and if further pages are available.

Generally speaking, we can differentiate two different approaches to pagination, which we'll call *offset-based* pagination (including its close cousin *page-based* pagination) and *cursor-based* pagination. Let's look at them, one by one.

### 18.4.1 Offset-based and page-based pagination

APIs with offset-based or page-based pagination (which we'll both describe as offset-based in this section) accept two common query parameters for their collection endpoints:

- The first parameter indicates the maximum number of results to return. Typically this is called `limit` or `per_page`. It's common to make the parameter optional and set a default value in its absence. There should also be a maximum limit that the API is willing to serve in a single request.
- The second parameter indicates either the number of results (offset-based) or the number of pages (page-based), starting from the beginning, to skip before returning any. In the former case it's generally called `offset`, in the second case it's often called `page`.

**NOTE**

OData requires naming offset parameters `$top` and `$skip`. JSON:API doesn't have any requirements, but reserves the parameter name `page` and suggests using it as a prefix for any pagination inputs. For example, `page[offset]` and `page[limit]`.

Let's look at an example to make this clearer. Assume you have 40 potential results in a collection. Calling the collection endpoint with `limit=20` (or `per_page=20`, depending on the name used in the API) would return a first page with resources 1 to 20. To get resources 21 to 40, you'd call the API with `limit=20&offset=20` for resource offsets, or `limit=20&page=2` for page-based pagination.

If the terms "limit" and "offset" sound strangely familiar, you may have some experience with relational databases. In Structured Query Language (SQL), there are the same keywords. The API requests from the previous paragraph might internally map to queries like `SELECT * FROM collection LIMIT 20 OFFSET 20`. This equivalence has two advantages. First, many developers are already familiar with offset-based pagination from SQL. Second, it's easy to implement the APIs, especially when there's a relational database management system in the background. However, it's time for a word of caution! As we previously mentioned, your API design should not reflect the internal implementation but the needs of the customer. It's okay to

go the opposite route and implement your first backend close to the API design, as we did ourselves in chapter 13, but your backend may evolve fast while your API should remain consistent.

A specific attribute of offset-based pagination is that it always looks at the full set of results and calculates the offset from the beginning. To understand why this sometimes leads to unwanted behavior, imagine a collection endpoint that returns a set of results starting with the latest entry. Think of a blog with multiple posts where you always see the newest post first. Now, let's look at the following interactions of multiple clients:

1. A client asks for the 10 latest blog posts, and the API returns them.
2. Another client (the author) publishes a new post. That means that all older items shift down. The 10 latest blog posts are now different; the page contains a new post and nine older posts, the tenth post moved onto the second page.
3. The first client wants to get the 10 next blog posts, so it asks for an offset of 10. The API calculates that offset with the new collection, so the first post on the second page is the original tenth post that the client has already seen.

Looking back at the user interface options we mentioned earlier, a repeated post on the second page might not be too bad. In an infinite scrolling interface, however, the duplicate entry would immediately look out of place. Also, imagine deleting an item instead of adding an item. That would shift items upward instead of downward and may have the implication that the client never sees some of the results. You have to weigh in these disadvantages with the familiarity and ease-of-implementation that are the advantages of this pagination style.

Before moving on to cursor-based pagination, let's look at response formats. From the beginning of our API design process, we have always recommended to wrap the result array for a collection endpoint into an object with a property called `items`, arguing that there may be other properties necessary for pagination. What are those, though?

For offset-based pagination, it's helpful to return the total result count, for example, in a property aptly named `count` or `total_results`. For page-based pagination, the API can alternatively return the number of pages, e.g., `page_count`. These values help with determining how many pages are available, and they are most useful if you expect API consumers to display all available pages in the navigation. An API consumer can use the offset/limit values from its previous request in combination with the counter to determine whether it can fetch additional results:

```
{
  "items": [
    {
      // result item
    }
  ],
  "total_results" : 20
}
```

### 18.4.2 Cursor-based pagination

Similar to offset-based pagination, cursor pagination supports two query parameters for the collection endpoints:

- Again, the first parameter indicates the maximum number of results to return (e.g. `limit` or `per_page`). The same conventions we introduced for offset-based pagination, namely a default value and a maximum value for this parameter, apply as well.
- The second parameter is the cursor that identifies the page. The cursor comes from the previous request.

To explain how cursors work, let's look at an example. In fact, the blog post example from the previous section is a great one. Imagine there are 30 posts in the blog and they have numeric IDs from 1 to 30. An API client makes a request with `limit=10`. As the blog shows the latest posts first, it would reveal posts 30 to 21. Then, it returns the cursor "ic20":

```
{
  "items": [
    {
      // item 30
    },
    ...
    {
      // item 21
    }
  ],
  "cursor" : "ic20"
}
```

To get the next posts, the API client makes a request with `limit=10&cursor=ic20` and receives those with IDs from 20 to 11, as well as cursor "ic10". The last page omits the cursor or sets it to `null` to indicate that there are no more results.

In the example above, it's possible to deduce the meaning of the cursor "ic20" (or "ic10"): the cursor is the next post identifier that comes after the current page. The API backend knows that it should only look at posts with ID 20 or smaller (i.e., older). As a general rule, however, API clients don't need to understand how the cursor works. They can simply treat it as an opaque identifier for "the next page". To illustrate this: that ours is the next item ID prefixed with "ic" (which stands for "item cursor") is just a randomly made-up convention, we may have as well called it "foobar20", simply "20", or "nextafter21". In the backend, you can think of it as a pointer to a specific position or a row in the database from which you continue delivering results.

#### NOTE

OData has a so-called "next link" in each API response, that describes how to make the request for more data. JSON:API suggests providing these links for `next`, `previous`, `first`, and `last` to allow API clients to move between pages. Such links can be helpful even when using offset-based pagination. Giving not just a cursor but a whole link is related to the idea of Hypermedia, which we briefly mentioned in chapter 1.

The example shows an advantage of cursor-based pagination over offset-based pagination. If the blog author publishes a new post with ID 31, the second page still starts with the next oldest post that the user hasn't read (i.e., ID 20), instead of an offset of 10 posts from the beginning (i.e., ID 21). Since deleting a post doesn't change the IDs of the existing posts, it wouldn't affect the results either. Thanks to these attributes, cursor-based pagination works very well with infinite scrolling user interfaces.

There's a downside to cursor-based pagination, however: it isn't designed for skipping over pages. You can just move from one page to the next and maybe back, but even if there were a maximum count of results in the response and you could deduce the number of pages, you couldn't jump to one of the later pages immediately. This makes it hard or even impossible to use with traditional pagination designs where jumping to different result pages is common.

As you've seen in this section, there are a lot of things to consider when designing pagination. The style may be affected by the type of user interface, so UI designers should be involved. The defaults and maximum limits for a single page depend on the backend, the database, and operations, because the API shouldn't let consumers request so much data that it violates internal constraints or request timeouts. Hence, the whole team should be involved in the API design, which again drives home the importance of API Design First and using a single source of truth that's accessible to the whole team.

**SIDE BAR**
**What are keyset and seek pagination?**

Keyset and seek pagination can be considered "lazy" variants of cursor-based pagination. The API backend only supports a `limit` parameter and asks the API consumer to figure out how to find the next page through filter parameters. For example, the API client would look at the ID or the date last item on the first page and use filter parameters like `created_before`, `since` or `before_id` to get the next page. To provide better developer experience, we recommend an explicit `cursor` or `next` attribute in the result so that API consumers don't have to figure things out and only need to pass the parameter.

## 18.5 Pagination for PetSitter

We're back in Josés office now. He left the decision to his developers to choose a pagination approach. As he doesn't feel strongly about the type of user interface or the need to jump to specific pages, he removes a crucial argument for offset- or page-based pagination. Max is willing to work with any approach and believes that the external mobile developers and eventual public API consumers would probably accept both, too.

Even though the backend architecture shouldn't necessarily prescribe the API design, Nidhi

starts some research into implementing pagination with MongoDB, the database management system she used to implement the PetSitter backend (see chapter 13). As it's important to remain consistent even in the long run, choosing a pagination approach that is difficult to implement or doesn't scale well with larger sets of data, will be a problem. If the team decides on a different pagination approach later, every API consumer needs to adapt their code.

As Nidhi looks into MongoDB, she finds out that it supports the `limit()` and `skip()` operations in its interface. As `skip()` is an offset, just by another name, implementing offset-based pagination is easy. However, she also learns that `skip()` can become slow and inefficient for larger datasets, due to the way that MongoDB works internally. That may not be a problem now, but it may eventually become one, so if the team decides to use offset-based pagination they may have to solve this later, for example by adding some sort of index system on top of the database. Building a cursor-based pagination that uses MongoDB's `ObjectId` is apparently more efficient.

It's also possible to combine different styles of pagination. For example, you could support an `offset` query parameter but also provide a `cursor` as part of the response. While this provides the most flexibility for the API consumer, it also makes the API design more complex. That, in turn, negatively affects the developer experience. The PetSitter team doesn't want that either.

### 18.5.1 Adding pagination to OpenAPI

Eventually, Nidhi and Max decide to use cursor-based pagination for PetSitter. To recap, for cursor-based pagination we need at least the following:

- A `limit` parameter, so that the API consumer can decide how much data they want. This parameter always has an `integer` data type. The API provider also decides on a default and a maximum value for the parameter. For PetSitter, backend developer Nidhi cannot say yet how much load her system can withstand, so she recommends being conservative and set small values for these. Eventually, the developers agree on a default of 20 and a maximum of 100.
- A `cursor` parameter that the API consumer can pass with every request, except the first, to decide from which point they want additional items. In general, this should be a `string`.
- A `cursor` attribute in the response which indicates that more items are available and how to get them. As the cursor in the response is the input for the next request, it needs the same data type: `string`. The team decides that the value should be `null` if there are no further results, so they add `nullable: true`.

With the initial decisions made, it's time to add the parameters. There are already some parameters - our filters - defined for the API operation `GET /jobs`, so we can now extend this list with two additional entries:

## Listing 18.10 PetSitter List/Search Available Jobs with Pagination

```

openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    post:
      #...
    get:
      tags:
        - Jobs
      summary: List/Search Available Jobs
      operationId: listOrSearchAvailableJobs
      parameters:
        #... ①
        - name: limit ②
          in: query
          description: The maximum number of results to return.
          schema:
            type: integer
            default: 20
            maximum: 100
        - name: cursor ③
          in: query
          description: Use the cursor from the response to access more results.
          schema:
            type: string

```

- ① Filter parameters omitted in this listing.
- ② Limit parameter with constraints.
- ③ Cursor parameter.

We also have to touch the `responses` part of the same API operation in order to add the `cursor` to the inline schema object next to the `items` array. Here's how that looks like:

## Listing 18.11 PetSitter List/Search Available Jobs with Pagination

```
openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    post:
      #...
    get:
      #... ①
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema: ②
              type: object
              properties:
                items: ③
                type: array
                items:
                  $ref: '#/components/schemas/Job'
                cursor: ④
                type: string
                description: Cursor for the next result page.
                nullable: true
```

- ① Omitted request configuration in this listing.
- ② Inline response schema to modify.
- ③ Existing items array.
- ④ New cursor parameter.

Awesome, we've included pagination in our API design for the `GET /jobs` endpoint, which covers the "Show more" function in the *job* concept of our domain model. As we haven't had anything planned for other endpoints, and with the same reasoning that made the team decide that filters aren't necessary for other endpoints, that's all we're doing for the sprint (and this book).

### 18.5.2 Extending our request example

As we've mentioned before, API consumers can combine filters and pagination. Let's take the example from the previous section and add a limit parameter:

```
curl -H "Authorization: {Auth}" "https://petsitter.designapis.com/jobs \
?start_time_after=2022-07-01T00:00:00+00:00 \
&end_time_before=2022-07-31T00:00:00+00:00&pets[species]=Cat&limit=10"
```

This time, the result also includes a cursor:

```
{
  "items" : [
    //...
  ],
```

```

  "cursor" : "507f1f77bcf86cd799439011"
}

```

With that cursor, the API consumer can make their subsequent request:

```

curl -H "Authorization: {Auth}" "https://petsitter.designapis.com/jobs \
?start_time_after=2022-07-01T00:00:00+00:00 \
&end_time_before=2022-07-31T00:00:00+00:00&pets[species]=Cat \
&limit=10&cursor=507f1f77bcf86cd799439011"

```

Last but not least, let's discuss sorting.

## 18.6 Designing sorting

Just like filters and pagination, sorting helps API consumers get the data that they need in the most efficient manner. Every collection requires an order in which the API returns the resources. The `items` element that we used for collection endpoints while designing the PetSitter API is an `array`, which is, by definition, an ordered list. Of course, there are data structures like `sets` that don't have an order or where the order doesn't matter, but they aren't relevant for collection endpoints in CRUD APIs. In other words, there's always a default sort order. Even if for some odd reason the underlying database wouldn't have one, you'd probably define it in your API. The matter that we want to discuss in this section is whether and how client can instruct the API to change that order.

In general, if we want to give instructions for sorting, we need to specify two inputs:

- The field or property that we want to use as the sorting key.
- The desired direction for sorting.

Let's unpack those. In CRUD APIs, the schemas for resources are typically compound data structures with the `object` type. There is no inherent order for these structures, so we need to choose a property with a simple data type like `string` or `number`. For example, if we take the PetSitter User schema, we could use the `full_name` property to sort users by their name. The direction for sorting is generally specified as either *ascending* or *descending*. What each direction means depends on the data type. For numbers, the meaning is obvious, and for strings it generally refers to alphabetical order. For date and time fields, the direction is either from oldest to newest (= ascending) or from newest to oldest (= descending).

Let's go back to the blog example from the introduction to pagination. The expected behavior is that the user wants to see the newest posts first. Hence, it makes sense that a descending order on a field indicating the creation date is the default. For simplicity, the blog might also give an incrementing ID to each post, so returning posts with a descending ID order would have the same effect. However, imagine an API client that wants to find the oldest post. If there's only a small number of posts, enough to fit in one page, they could just retrieve them in one API call and pick the last item. If there's more content, however, they would have to go through all result

pages until they reach the last page. By adding a parameter to indicate that they want to get the oldest posts first, they could achieve the same thing in a single API call. The example illustrates that sorting is most useful when combined with pagination, especially cursor-based pagination where it's impossible to skip ahead.

### 18.6.1 Single-field sorting

As with filters and pagination, we have to face the challenge of mapping the two inputs for our sorting algorithm in a key-value pair. The most common solution we've seen in the wild is a single query parameter, typically named `sort` or `sort_by`. As its value, the API expects the field name, suffixed or prefixed with an indicator for the direction. Here are some examples of how the query might look like:

- `sort_by=name:asc` / `sort_by=name:desc`
- `sort=+created_at` / `sort=-created_at`

Another option is to separate field and direction into two different parameters, removing the need for a separator character (like `:`):

- `sort_by=name&order_by=asc` / `sort_by=name&order_by=desc`

The version with two parameters requires you to think about the behavior when the API consumer provides just one parameter but not the other. You could choose to reject requests where one parameter is missing, but you could also decide to make one direction, e.g., ascending, the default if there's just `sort_by` but no `order_by` in the request.

### 18.6.2 Multi-field sorting

There may also be a requirement to use multiple sort parameters. For instance, if you have a database of contacts, you may want to sort them by the city they live in, but for those that live in the city you also want to sort them alphabetically by name. If you need to support this kind of behavior, you require another separator character (like `,`). Then, your API could look like this, for example:

- `sort_by=city:asc,name:asc`

When there are multiple sorting keys, separating them and the order in different query parameters makes no sense, so that option is taken off the table.

**NOTE**

**OData and JSON:API also use a single parameter. OData calls it `$orderby` and JSON:API uses `sort`. Both specifications support sorting with multiple fields (comma-separated) as well. OData requires the "Asc" or "Desc" suffix for field names. JSON:API makes ascending the default and asks for a minus ("`-`") as the prefix if the order should be descending.**

As you can see, sorting parameter design can get quite involved. Before you make choices for the APIs you design, there are a few things you should keep in mind. As usual, API design starts with customer requirements and you should only add to your API what your customers need. There's no need to overcomplicate things and hurt your developer experience without having a strong use case. Also, as we've mentioned earlier there's no native way in OpenAPI to document complex strings like `city:asc, name:asc` (except maybe regular expressions, but they only cover syntax and not semantics), which means you may have to rely on prose in the description to explain your format to developers. As a result of it being prose, there are no tools to assist developers.

### 18.6.3 Consistency throughout parameter types

Consistency, as usual, is the crucial part of parameter design. It means that not only should your sorting parameters look the same for every endpoint, they should also feel consistent with other parameters, such as filters. To understand what that means, have a look at the following API call parameter strings and think about how they "feel", before reading on:

1. `created_before=2020-07-01&sort_by=author&order_by=desc`
2. `created=lt:2020-07-01&sort=author:desc`
3. `created=<2020-07-01&sort=-desc`
4. `created=<2020-07-01&sort_by=author&order_by=desc`
5. `created.lt=2020-07-01&sort_by=-author`
6. `created=lt:2020-07-01&sort_by=-author`
7. `created=<2020-07-01&sort_by=author:desc`
8. `created=lt:2020-07-01&sort_by=author.desc`

The first three parameter strings each follow a certain style:

1. Long, nicely human-readable keys, avoiding special characters.
2. Putting all details in the value part with keywords (`lt`, `desc`) and a consistent separator (`:`).
3. Similarly putting details in the value part, using single special characters as prefixes (`<`", `-`).

For the other five, there is no specific style; naming conventions, special characters, and separators are used in inconsistent ways. We won't discuss them in detail, but we believe that either the first three - no matter which style you prefer - provide a more joyful developer experience.

And again, at the risk of sounding like a broken record, your API design should not be a direct reflection of your backend and your database, but you must obviously support the capabilities. As with filters, you may need indexes in your database to efficiently support queries with sorting, and you generally don't want to index your database on all fields. And once you've

added a capability to your API, it's impossible to remove it without breaking at least one integration, because someone will rely on it. So it's probably better to err on the side of offering fewer capabilities, such as only supporting some designated fields for sorting. You can always add them later.

## 18.7 Sorting for PetSitter

For sorting, the PetSitter team needs to identify the properties that can be used as sorting keys and decide on a format for the sort parameter. As with the previous sections, we'll only look at the `GET /jobs` endpoint for this sprint.

### 18.7.1 Finding sorting fields

All fields from the Job and Pet schema are potentially relevant. Nidhi and Max decide to go through the fields in the same way they did for filters:

#### **ID**

As PetSitter doesn't use auto-incrementing numeric IDs, to the user, IDs are arbitrary strings. Even if they have an alphabetic order in the underlying database, this is an implementation detail that could change. If the API consumer was interested in newly added jobs, it would make more sense to filter on a field like `created_time`, but there is no such field in the Job schema. As long as that's the case, there will be no sorting for IDs.

#### **CREATOR\_USER\_ID**

Similar to `id`, there isn't necessarily semantic meaning to these identifiers. Even if it were, there is no use case for sorting on users.

#### **START\_TIME AND END\_TIME**

Pet sitters may look for jobs starting soon or plan in advance. In combination with a filter for `start_time`, using this as a sort criteria makes sense, so we'll allow it. For consistency with the filters which we created for `end_time` in the same way, we should also allow sorting by end time.

#### **ACTIVITY**

As we explained earlier, this is a free text field. While it makes sense to search, the alphabetic order isn't relevant. Therefore, no sorting on `activity`.

#### **PETS[NAME], PETS[AGES], AND PETS[SPECIES]**

The PetSitter team is unsure about the use cases for these. To keep things simpler for now, we will not add sorting on any Pet field.

## 18.7.2 Designing the sort parameter

The decision for the sort parameter design is its name, whether to use one or two parameters, how to identify the direction, and whether to allow multiple sorting keys in one request. As of now, we have only decided to allow sorting on `start_time` and `end_time`. The combination of these two doesn't feel useful, so it's likely enough to allow one parameter. It can always be extended later. Still, since that's a potential future use case, we'll use a single sort parameter because two parameters (one for the key and one for the direction) don't work well when we support this use case later. The PetSitter team decides to name the parameter `sort` - no frills! They also decide to use `:asc` and `:desc` as the suffixes.

## 18.7.3 Adding pagination to OpenAPI

To integrate sorting into the OpenAPI definition, it's just a single parameter to add to the existing parameter list for the `GET /jobs` operation. The parameter is a `string`, and we'll not give any constraints and instead rely on the `description` to explain how it works:

### Listing 18.12 PetSitter List/Search Available Jobs with Sorting

```
openapi: 3.0.0
#...
paths:
  #...
  /jobs:
    post:
      #...
    get:
      tags:
        - Jobs
      summary: List/Search Available Jobs
      operationId: listOrSearchAvailableJobs
      parameters:
        #...
        - name: sort
          in: query
          description: |
            Indicate the sorting key and direction for the results.
            Use the field name, suffixed with ":asc" for ascending
            or ":desc" for descending order.
            Valid fields: start_time, end_time
      schema:
        type: string
```

## 18.7.4 The final request example

Once again, let's extend our sample request from the filter and pagination sections. Imagine our cat sitter primarily wants to find jobs ending late in July, so he decides to add sorting for `end_time` in descending order:

```
curl -H "Authorization: {Auth}" "https://petsitter.designapis.com/jobs \
?start_time_after=2022-07-01T00:00:00+00:00 \
&end_time_before=2022-07-31T00:00:00+00:00&pets[species]=Cat&limit=10 \
&sort=end_time:desc"
```

This time, the API would return jobs ending late in the month first:

```
{
  "items" : [
    {
      "start_time" : "2022-07-20T10:00:00+00",
      "end_time" : "2022-07-30T22:00:00+00",
      //...
    },
    //...
  ],
  "cursor" : "addedfeed0000000000000000"
}
```

## 18.8 Summary

- Filters, pagination, and sorting give API consumers the ability to control the results that the API returns, how many resources are included in the collection per request, and how to sort them. In CRUD-style APIs, they should be optional query parameters for collection endpoints. This way, API consumers can add any combination of these three features to their API calls.
- Not every endpoint needs these parameters, and it's not necessary to allow filtering and sorting for every field. As part of the API design process, API providers should choose the parameters that they believe their API consumers need and that they can continually support, even when the backend of the API changes. For PetSitter, we added filters, pagination, and sorting for the `GET /jobs` endpoint, using a subset of fields from the Job and Pet schemas.
- Filters are typically query parameters named after fields. Wherever it doesn't make sense to only filter for explicit values, it's necessary to add suffixes to the field names or use a specific value syntax to allow upper and lower bounds.
- Pagination can either be offset-based or cursor-based. Both options have their advantages and disadvantages. For PetSitter, we chose cursor-based.
- Sorting typically requires a single parameter, in which the field to sort by is suffixed by the direction - ascending or descending.
- There are a lot of ways to design the parameters and there isn't a single right or wrong answer when it comes to parameter naming, ranges etc.. For great developer experience, the crucial aspect of a parameter design that follows a recognizable overall style and therefore feels internally consistent.

# Supporting the unhappy path: error handling with problem+json

19

## This chapter covers

- Finding and categorizing API errors
- OAS tools error handling
- The problem+json format
- Adding error responses to the PetSitter OpenAPI definition

As we designed and implemented the PetSitter API, we've mostly looked at the happy path, which is when everything works according to the plan and things are "200 OK". Obviously we want the interactions between our API and its users to be on this path as often as possible, but we cannot always guarantee that. In this chapter, we'll look at the ways things can go wrong and how to handle the situation.

The OpenAPI definition for an API is a contract that both sides, client and server, have to follow. If you look outside the field of technology and into contracts as legal documents, you'll notice that they don't just describe the happy path. In fact, most of the times the greater part of the legalese in the document describes all the potential problems and how to mitigate them. It's when things go wrong that contracts are the most relevant. Error handling is equally important.

The same process that developers use to collaborate on the happy path, which includes designing schemas and API operations, can and should also guide their approach towards error handling. Each developer can bring their perspective to the table (and so can non-developers involved in the API design process).

In this chapter, we'll look at why error handling is crucial and the negative effects of *not* having proper handling. Then, we attempt to categorize types of errors and look at the API operations in our PetSitter API to find out which errors could occur for each of them. We also talk about the requirements for useful error responses. As we get some error handling from the OAS tools in the PetSitter backend thanks to Swagger Codegen, we'll discuss that format and see if it fulfills

our requirements. For additional error handling we'll introduce the problem+json format. The goal we reach at the end of the chapter is having the documentation of the error responses as part of the PetSitter OpenAPI definition. Afterwards, we'll conclude with some advice on implementing error handling.

## 19.1 The problem

It is a fact of life that things can go wrong. We cannot always avoid failures, but we should make sure that we notice them, recover and find ways to fix them. The rule applies to technology in general and the world of APIs in particular as well.

Let's look at a specific example in PetSitter. The first thing a user needs to do to use the software is register an account. How does this look like from a user's point of view, ignoring the implementation details and the API for a moment? They go to the PetSitter website, enter their details into a registration form, and submit that form. The system informs them that they registered successfully and can log in. That is the happy path.

It's also possible that registering an account fails. Let's collect a few reasons for why we may have entered an unhappy path:

- The user did not fill all required inputs.
- The user entered some invalid information, for example a malformed email address.
- The user entered an email address that already exists in the system and cannot register again.
- There's a bug in the frontend code.
- There's a bug in the backend code.
- The backend is temporarily unavailable, for example because of a redeployment or due to maintenance work.
- The database crashed.
- A router in the datacenter crashed.
- The Internet connection of the user stopped working, for example because they moved out of WiFi coverage.

Wow, so many unhappy paths! I'm sure you could make this list even longer (try it, if you want). However, don't let this discourage you. Some of these are fairly quick to fix by the user, for example they can complete the form or fix a typo in their email address in seconds and retry. It would be helpful for them, though, if they knew which field is missing and invalid.

If a user registers with an existing email address, they should see a message indicating that they cannot register. Maybe the user just forgot that they registered an account for PetSitter before? We can point them in the right direction, for example, by asking them to log in to their existing account.

There are other problems that the user cannot fix, such as when there are bugs in the code or the

server infrastructure has problems. In that case, however, the user is less frustrated when the application can tell them it's not their fault and that they should wait and try again later.

In any case, the most negative user experience would be if they saw a message along the lines of "Something went wrong" (regardless of *which* unhappy path they're on), enter an unexpected state inside the application, get a blank screen or, even worse, leave with the impression that everything went well.

The first user of an application is the developer who creates it. In a web application like PetSitter where a backend and a frontend developer work separately to build their parts, the frontend developer is both the first user of their frontend and the first consumer of the backend developer's API. If something goes wrong while they're testing the application, they ultimately have to ask themselves:

- Did I do something wrong as a user?
- Is there a bug within my code that I have to fix?
- Is there a bug in the backend API, i.e., does it behave differently from the mock server I used before, and I have to report that to the backend developer?

Without error handling, it is difficult for the frontend developer to answer this question, so they get stuck and waste additional time debugging. To solve this problem and get the developer (or any API consumer) unstuck, error handling is already essential during development and shouldn't be an afterthought. As the first step towards the solution, we will try and categorize the potential issues to tackle error handling strategically.

## 19.2 Error categories

If we look at APIs, we can have three types of errors:

- The user made a request that the API doesn't understand or cannot fulfill. For example, a call to an undefined API operation, a request for a non-existing resource, invalid authentication, or a request body that doesn't conform to the schema that the operation requires. We call these *client errors*.
- The user made a perfectly valid API request but something is wrong with the API itself or its underlying infrastructure. For example, a lack of server-side resources like memory, the unavailability of a dependency like the database, or a bug in the (server-side) code. We call these *server errors*.
- The transmission of the API request or API response between client and server failed. We call these *network errors*.

If you want, you can look at the list of issues from the previous section and try to group them in these three categories. While network errors are important to keep in mind, they are outside the scope of APIs so we won't cover them in this chapter. We'll also mostly gloss over server errors, because their cause is usually a bug in the code or faulty infrastructure - things unrelated to API design. Both these types can also indiscriminately affect every API operation. As there's

sometimes no clear distinction between server errors and network errors, don't get caught in their differences - you can put them in the same bucket, if you want. For the remainder of the chapter, our focus is on client errors.

### 19.2.1 Finding unhappy paths

If we want to understand which API operations can cause which client errors, one way to go about it would be to look at each of them individually and ask yourself: what could go wrong? Our plan is to collect various client errors and then establish a system of categorization.

#### POST /REGISTER

- Malformed input (e.g., no or invalid JSON), missing fields or invalid data types
- User already exists

#### GET /USERS/{ID}

- User does not exist
- User is not allowed to access another user

#### PUT /USERS/{ID}

- Malformed input (e.g., no or invalid JSON), missing fields or invalid data types
- User does not exist
- User is not allowed to access another user

#### DELETE /USERS/{ID}

- User does not exist
- User is not allowed to access another user

#### POST /JOBS

- Malformed input (e.g., no or invalid JSON), missing fields or invalid data types
- User is not allowed to create a job because they don't have the "pet owner" role

#### GET /JOBS

- Invalid input for the query parameters we added in chapter 18 (to support filters, pagination, and sorting)

**GET /JOBS/{ID}**

- Job does not exist

**PUT /JOBS/{ID}**

- Malformed input (e.g., no or invalid JSON), missing fields or invalid data types
- Job does not exist
- User is not allowed to modify the job (because they didn't create the job and are not an admin)

**DELETE /JOBS/{ID}**

- Job does not exist
- User is not allowed to delete the job (because they didn't create the job and are not an admin)

**GET /JOBS/{ID}/JOB-APPLICATIONS**

- Job does not exist
- User is not allowed to view job applications

Is it an error if the job has no applications? In chapter 10, we stated the following: "For collection endpoints, every API call should return status code 200, even if the collection is empty." Therefore no, we wouldn't consider that an error, and return a collection with no items as our success response.

**POST /JOBS/{ID}/JOB-APPLICATIONS**

- Malformed input (e.g., no or invalid JSON), missing fields or invalid data types
- Job does not exist
- User is not allowed to apply to the job because they don't have the "pet sitter" role or it is a job they posted themselves

**GET /USERS/{ID}/JOBS**

- User does not exist
- User is not allowed to access another user

As mentioned before, an empty collection is not an error.

## PUT /JOB-APPLICATIONS/{ID}

- Malformed input (e.g., no or invalid JSON), missing fields or invalid data types
- Job application does not exist
- User is not allowed to modify the job application (because it's not theirs and they are not an admin)

## POST /SESSIONS

- Malformed input (e.g., no or invalid JSON), missing fields or invalid data types
- Invalid credentials

### 19.2.2 Common error patterns

Looking at the list in the previous section, we can identify four major groups of client errors:

- Structurally invalid inputs, such as missing fields or malformed data
- Semantically invalid inputs, such as a user trying to register when they've already registered
- Requests for resources that don't exist
- Permission issues (wrong user role or missing access grants)

When we decrease the number of groups from four to three by subsuming both invalid input types into one, each group corresponds to a common HTTP status code from the client error range: 400 ("Bad request") for invalid input, 404 ("Not found") for non-existing resources, and 403 ("Forbidden") for permission problems.

We can also recognize patterns concerning the type of API operations where these errors occur. Invalid input errors can happen in every write operation that requires a request body, i.e., operations that use the POST, PUT or PATCH method. They also occur in GET requests that support query parameters. Non-existing resources are common errors for individual resource endpoints with a path parameter identifying a specific resource, and they affect every operation (GET, PUT, DELETE). They also appear in subresource collection endpoints if the original resource doesn't exist, but not if the collection is empty. Finally, permission issues can arise for every resource, collection, and endpoint, depending on the business logic for the permission system of the application. We've summarized all error types in table [19.1](#). Later in this chapter, we'll look at the error codes for each method.

**Table 19.1 Common API client errors**

Status	Description	Occurrence
400	Invalid input	POST/PUT as well as GET with query parameters
403	Access forbidden	Any endpoint with permission-related business logic
404	Resource not found	Individual resource endpoints and subresource collection endpoints

## 19.3 Requirements for error responses

Whenever an API fails gracefully, it should return a useful error response to the API consumer. Our next step is to look at the requirements for designing error responses that support developers when they get stuck integrating an API, so they can get unstuck and return to the happy path.

First of all, an error response should be clearly distinguishable from a successful response. HTTP status codes help make this distinction. Successful responses (including redirects) have status codes ranging between 200 and 399 whereas errors have status codes ranging between 400 and 599.

Next, both success and error responses should have the same data serialization format. Most APIs, including all we created or discussed in this book, use JSON for responses. Using the same format reduces the effort on behalf of the consumer to understand different formats, they can run every response through a JSON parser and then work with the result. Also, malformed JSON can be treated as an unexpected error in the same way as a network error.

Finally, the data structure, i.e., the JSON Schema, should be similar for all error responses. Let's look at two failing requests to a fictitious API with responses that are a good example of how *not* to do it. The first is a non-existing resource:

### Listing 19.1 Request/response example for bad error handling 1

```
curl "https://example.com/api/resources/nonExisting"
{
  "error": "Path /resources/nonExisting not found."
}
```

And the next is an invalid input:

### Listing 19.2 Request/response example for bad error handling 2

```
curl -d "email=test@example.com" "https://example.com/api/resources"
{
  "code": "invalid_field",
  "field": "email"
}
```

Having a common and consistent structure helps the API consumer because they can reuse more of their error handling code. In example [19.3](#) there's a field called `error` and in example [19.4](#) it's called `code`. What should the developer look for?! Also, are these error messages complete or what else should the data structure contain?

In any case, an error response should have a field with a human-readable error message ([19.3](#) has it, [19.4](#) doesn't), describing the error in an understandable way. Optionally, the error message could come in a short version, e.g., a single sentence, and a longer description with explanations

how to fix it. Having a human-readable error message has the following advantages:

- The developer consuming the API can immediately understand what's happening, even if they don't understand the rest of the error response or the HTTP status code.
- In many cases, the default behavior for error handling - at least for *client errors* which are likely caused by user input - can be to take this verbatim message and show it to the end user, so they can also understand what's wrong.

Having a human-readable message is, however, not sufficient for a great error response, due to the fact that things that are easy to understand for humans are often rather difficult to understand for machines. Now, which "machines" could be interested in understanding errors?

- The client-side code, if it doesn't just want to relay the error message but implement some additional error handling. For example, if a field is missing, it can highlight it in the UI by adding a red-colored frame or underlining the input.
- An API gateway or proxy that stands between the client and the server, or an API testing or monitoring system that wants to create and analyze logfiles to indicate how often which errors occur in an API.

There are various ways to create specific groups of errors that require different ways to handle them or which are helpful to distinguish in logfiles. HTTP status codes are a helpful first step but they are not granular enough. For example, a 404 ("Not Found") could mean both that a certain path doesn't exist in the API or that a resource was not found.

Going back to the example from the problem section, we could consider the following questions to which the error response could provide a machine-readable response:

- What's the overall type of error? The 400 ("Bad Request") status code indicates that *something* is wrong with the client input. Specific types could be "Missing field", "Invalid syntax for field", "Duplicate data", etc. (a common `code` like "invalid\_field" in [19.4](#) works well).
- Which field is the error related to? The answer could be "email", "name", etc., which allows the client-side code to point the user to the input for that field (the `field` property in [19.4](#) seems useful).
- What exactly is wrong with the field? The answer could be "too long", "too short", "missing an @", "already existing in database", "on a blocklist" etc.

If we want to answer these questions without applying natural language processing to the human-readable error message, we should accompany that message with a set of structured data in a consistent format. What should be an appropriate schema, though?

For this chapter, we'll look at two formats:

- The error format that's built into the OAS tools, for the error handling we get for free from using Codegen.
- An open standard called "Problem Details for HTTP APIs", specified in RFC 7807 (we'll call this *problem+json*).

If you design and implement an API from scratch and you have enough resources, the gold standard to aim for would obviously be a *single* error format. That way, all errors have a consistent schema. For PetSitter, however, we'll ignore our own advice and use *two* formats. The PetSitter team can justify this as a pragmatic decision that allows them to reuse the error handling from Codegen and the OAS tools, which we discussed in chapter 13, but also follow a well-defined standard for their custom error messages. From our perspective as authors of this book we believe there's educational value in teaching you both these formats as well. Let's have a look at each of them next.

## 19.4 The OAS tools / Swagger Codegen format

We used Codegen to autogenerate a backend implementation from our OpenAPI definition. As we did that, we learned that we get some functionality from this process for free, such as input validation. On top of that, we can expect any web application framework to handle non-existing paths or unsupported API operations. Back then, we did not focus on the format of these error messages, but now that we're talking about error handling, we should analyze how they created their error messages and whether those fulfill the requirements we outlined in the previous section. If you've created or tested the autogenerated backend code from chapter 13, you have likely seen some of these error responses already.

**NOTE** There is a possibility that newer versions of Codegen or the OAS tools implement a different format.

Let's look at operation-related errors first. Try the following:

- Make a request to a non-existing route, such as `GET /pets`
- Make a request to an existing route but with an unsupported verb, such as `DELETE /users`

### Listing 19.3 PetSitter response to `GET /pets` - HTTP/1.1 404 Not Found

```
{
  "message": "not found",
  "errors": [
    {
      "path": "/pets",
      "message": "not found"
    }
  ]
}
```

#### Listing 19.4 PetSitter response to DELETE /users - HTTP/1.1 405 Method Not Allowed

```
{
  "message": "DELETE method not allowed",
  "errors": [
    {
      "path": "/users",
      "message": "DELETE method not allowed"
    }
  ]
}
```

If you look at these responses, they tick a lot of boxes:

- They contain the HTTP status code we expect.
- They are in JSON, just as the successful responses.
- Both have a consistent schema. In this case, it contains `message` and `errors` fields.
- There is a human-readable error message. It's always in the `message` field.

Moving forward, let's look at an input validation error. We'll reuse the sample from chapter 13 in which we called the "Register User" action located at `POST /users` but send a string instead of the expected array for the `roles` field:

```
{
  "full_name": "John Doe",
  "roles": "PetSitter",
  "email": "john.doe@designapis.com"
}
```

#### Listing 19.5 PetSitter response to invalid POST /users - HTTP/1.1 400 Bad Request

```
{
  "message": "request.body.roles should be array",
  "errors": [
    {
      "path": ".body.roles", ①
      "message": "should be array", ②
      "errorCode": "type.openapi.validation" ③
    }
  ]
}
```

- ① The JSONPath pointing to the faulty request field.
- ② A short and specific error description related to the field.
- ③ A code that identifies this as a validation error.

As expected, we get a 400 response. On the surface, this looks roughly the same as the two other error responses. An interesting aspect is the `path` field, which now doesn't refer to a URL but to a position within a JSON request body. The syntax is based upon the JSONPath<sup>12</sup> standard.

Also, there is a `message` field with a human-readable error description, and there's an `errors`

array. The item inside `errors` provides an `errorCode` field with a specific type identifier called `type.openapi.validation` so that clients can immediately understand that this is a validation error and connect all validation errors, regardless of path and message, with the same error handling implementation. A point worthy of criticism is that the `errorCode` was absent from the `errors` array for our 404 and 405 ("Method not allowed") response, so there's no full consistency here. Still, the format fulfills enough of our expectations.

We summarized the fields we've observed in these error messages in table [19.2](#), and the schema for each error in the array in table [19.3](#).

**Table 19.2 JSON Schema for OAS tools errors**

Field	Type	Description
<code>message</code>	<code>string</code>	Human-readable error message.
<code>errors</code>	<code>array</code>	List of errors (see subschema).

**Table 19.3 JSON Schema for OAS tools errors, subschema**

Field	Type	Description
<code>path</code>	<code>string</code>	For input validation errors, identifies where in the JSON request body the error occurred.
<code>message</code>	<code>string</code>	Human-readable error message.
<code>errorCode</code>	<code>string</code>	Code indicating error type.

Now we got one of the schemas, which is for the errors handled by the OAS tools. As mentioned before, we'll not use the same for our custom error handling, so let's move on to the other format.

## 19.5 The problem+json format

The "Problem Details for HTTP APIs" specification, published in RFC 7807, exists as a minimal but extensible, standardized approach towards error responses. It has an XML and JSON serialization, though we'll only consider the JSON version. The specification suggests setting the `Content-Type` to `application/problem+json` instead of `application/json` for error responses as an additional indicator that it's an error response and it's following a specified standard. Therefore, we'll call the format *problem+json*. In chapter 21, you'll learn more about custom content types like that.

As with all API-related best practices and open standards, using them helps with API design consistency not just within the scope of a single API but also among multiple APIs in an organization or even across various API providers. We recommend using this format for error responses whenever possible.

The problem serialization is an object with various defined properties, which we've listed in table [19.4](#).

**Table 19.4 JSON Schema for Problems**

Field	Type	Description
<code>type</code>	<code>string</code>	A URI describing the type of the error.
<code>title</code>	<code>string</code>	A short, human-readable title for the error.
<code>status</code>	<code>integer</code>	The HTTP status code.
<code>detail</code>	<code>string</code>	A human-readable longer explanation of the error.
<code>instance</code>	<code>string</code>	A URI identifying the occurrence of the problem.

There are a few interesting things to note about the schema:

- It uses URIs for the `type` and `instance` fields. One advantage of URIs is that they are globally unique identifiers. The other advantage is that they can be "dereferenced", which is a fancy way of saying that you can put them in a browser and retrieve a page with information. For example, you could create a URI that identifies a certain problem and, at the same time, links to an API documentation page on your website or developer portal that has additional information about the type of error. The format we discussed in the previous section had a similar notion of a unique type identifier with the `errorCode` field, but it used string tokens like `type.openapi.validation` instead of URIs.
- The human-readable part of the error comes in two parts, a short `title` and a longer `details` field. The format we discussed in the previous section only had a single `message` field for this purpose.
- The JSON structure contains a copy of the HTTP status code. In most cases the information is redundant because that code should be identical between the HTTP header and the JSON body. In other cases, however, if someone in between the API and its consumer tampers with the HTTP protocol, you can retrieve the code from the body.

Developers can extend the format with additional fields to provide machine-readable information that have type-specific semantics. The specification for the `problem+json` format does not specify further details.

By being in JSON, just as the successful responses, being applicable consistently for different types of errors, and containing a human-readable error message, they tick all boxes laid out in our requirements. Let's look at an example for a failing request that we've implemented in PetSitter:

```
curl -H "Authorization: {Auth}" "https://petsitter.designapis.com/jobs/nonExistingJobId"
```

And that's how the problem response body looks for this:

```
{
  "type": "https://petsitter.designapis.com/problem/not-found",
  "status": 404,
  "title": "Job not found."
}
```

Now, that we know our two schemas, it's time to include them in our OpenAPI definition.

## 19.6 Adding error responses to OpenAPI

When you design API operations, you add them under the `paths` attribute in your OpenAPI file with the URL path and the HTTP method. For each API operation, there is a `responses` field. So far in this book, we have always added a single entry in `responses`, typically named `200` for the `200` ("OK") status code. At some points we also used `201` ("Created") and `204` ("No Content") status codes, which still belong to the success range. Then, within that response, we added the `content` field, which either included an inline schema or a reference (using `$ref`) to a common, reusable schema from the `components` section of the OpenAPI file.

To support an unhappy path, or, more precisely, different unhappy paths, we can add any number of additional responses as part of the `responses` object. Considering that the OpenAPI specification uses the HTTP status codes as keys in that object, we can describe exactly one response for every status code. On top of that, we can use the `default` key to describe a default response, implying that API responses with an HTTP status code other than those included in the API definition should follow the default format.

Of course, we could blindly add a `default` response for errors to every method and call it a day. Technically it covers everything and the contract would be valid. Would the contract be useful, though? If we get into the shoes of the frontend developer, or an API consumer in general, it wouldn't be. They would know that something can go wrong (I bet they already knew that), but they have no idea what problems could occur and which status codes they can expect for each method. If we want to help the API consumer in writing specific error handling code, we should tell them more about the unhappy paths. Also, even if we don't care that much about developer experience, specific error documentation in OpenAPI also helps the API provider uphold their side of the contract.

As we've mentioned before, we want consistent error messages throughout our API. Therefore, it doesn't make a lot of sense to use an inline schema and copy it into every operation. Instead, we should create common schemas.

### SIDE BAR

#### Common schemas in external files

We are using `$ref` to reference components within the same OpenAPI file. It is also possible to make references to external files or URLs. For example, if you are in an organization with multiple APIs and want to reuse various schemas, such as, but not limited to, error response formats, it makes sense to keep the common elements in separate files. As we're working with a single API, we'll stick to a common schema within the same file for now.

Another thing worth mentioning is that the structure of an OpenAPI definition allows us to provide error responses in the `responses` field for all API operations that exist. However, the

specification doesn't provide a way to describe errors for non-existing paths and methods. Which means, that 404 (if it means the *operation* was not found, not that a *resource* was not found) and 405 errors from Codegen that we looked at earlier don't find their way to the OpenAPI file. However, they are no less relevant and we should make them consistent with other responses, wherever possible.

Our next steps are to add the two formats we introduced earlier as common schemas, and then add error responses to our paths.

### 19.6.1 Creating error schemas

All schemas we created so far were the result of the intricate API design process that our PetSitter team had to go through in chapters 9 and 10 (and updated in chapters 16 and 17). For error handling, we don't have to design schemas, since we can reuse what's already out there. For the problem+json schema, we can use the official specification as our definition. For the errors thrown by the OAS tools we can create the schema from what we observed. Earlier in this chapter, we listed all fields for these schemas (in tables [19.2](#) and [19.4](#)) already. Now let's add them to our OpenAPI file:

## Listing 19.6 PetSitter OpenAPI with OAS Error and Problem schemas

```

openapi: 3.0.0
#...
components:
  schemas:
    OASError: ①
      type: object
      properties:
        message:
          type: string
          description: Human-readable error message
        errors:
          type: array
          items:
            type: object
            properties:
              path:
                type: string
                description: For input validation errors, identifies where in the JSON request
                body the error occurred
              message:
                type: string
                description: Human-readable error message
              errorCode:
                type: string
                description: Code indicating error type
    Problem: ②
      type: object
      properties:
        type:
          type: string
          description: URI indicating error type
        title:
          type: string
          description: Human-readable error title
        status:
          type: integer
          description: HTTP status code
        detail:
          type: string
          description: Human-readable error details
        instance:
          type: string
          description: URI indicating error instance

```

- ① Common schema for the OAS tools error format.
- ② Common schema for the problem+json format.

Now that we have the error schemas that we can reference - OASError and Problem - we should look at the operations and add the necessary error responses.

## 19.6.2 Adding errors to operations

We've already gone through the operations earlier in this chapter and answered the question: what could go wrong? Based on that question, we categorized the types of client errors and the types of API operations where they typically occur. We also learned that we can rely on the OAS tools for some of our error handling, mainly input validation, whereas we need to write custom code for others. Finally, we know now that the errors that OAS tools handle need to reference the OASError schema and our custom errors reference the Problem schema. Therefore, we should be good with the following rules:

- All POST and PUT operations as well as the parameter-heavy endpoint for job searches rely on input validation and could therefore potentially throw a 400 error with the OASError schema.
- For operations that contain an `{id}` placeholder, we may find the user trying to request a resource that doesn't exist. As we handle this in custom code, there might be 404 errors with the Problem schema.
- Similarly, the user may not be allowed to access the resource based on permission business logic. Therefore, the same resource endpoints may also throw 403 errors with the Problem schema.
- Every API operation that needs authorization could cause a 401 error with the OASError schema.

When we apply the rules, we get the error responses shown in table [19.5](#).

**Table 19.5 Operations with errors**

Operation	400	401	403	404
POST /users	Yes	No	No	No
GET /users/{id}	No	Yes	Yes	Yes
PUT /users/{id}	Yes	Yes	Yes	Yes
DELETE /users/{id}	No	Yes	Yes	Yes
POST /jobs	Yes	Yes	Yes	No
GET /jobs	Yes	Yes	No	No
GET /jobs/{id}	No	Yes	No	Yes
PUT /jobs/{id}	Yes	Yes	Yes	Yes
DELETE /jobs/{id}	No	Yes	Yes	Yes
GET /jobs/{id}/job-applications	No	Yes	Yes	Yes
POST /jobs/{id}/job-applications	Yes	Yes	Yes	Yes
GET /users/{id}/jobs	No	Yes	Yes	Yes
PUT /job-applications/{id}	No	Yes	Yes	Yes
POST /sessions	Yes	Yes	No	No

Based on the operations and the errors we identified, we can add them to our OpenAPI file. We will not include all updated operations here, but you can get the OpenAPI file from the book's website or repository. Anyway, we'll show you one example - the `POST /jobs` method - to

illustrate how it works:

### Listing 19.7 PetSitter POST /jobs with errors

```
openapi: 3.0.0
#...
paths:
#...
/jobs:
post:
#...
summary: Create Job
operationId: createJob
responses:
'201': ①
  description: Created
  headers:
    Location:
    schema:
      type: string
'400': ②
  description: Bad request
  content:
    application/json:
    schema:
      $ref: '#/components/schemas/OASError' ③
'401': ④
  description: Unauthorized
  content:
    application/json:
    schema:
      $ref: '#/components/schemas/OASError' ⑤
'403': ⑥
  description: Forbidden
  content:
    application/problem+json: ⑦
    schema:
      $ref: '#/components/schemas/Problem' ⑧
requestBody:
  content:
    application/json:
    schema:
      $ref: '#/components/schemas/Job'
```

- ① The success response we created earlier.
- ② A 400 error ...
- ③ ... using the OASError schema - from framework input validation.
- ④ A 401 error ...
- ⑤ ... using the OASError schema - from framework operation security check.
- ⑥ A 403 error ...
- ⑦ ... with the application/problem+json content type ...
- ⑧ ... and using the Problem schema - from PetSitter business logic.

Once we've added the errors to all operations, we have a thorough documentation of the unhappy paths in the PetSitter API. Not just PetSitter frontend developer Max but also the external contractor developing the PetSitter mobile app and eventual public API users now know what

they can expect.

## 19.7 Error handling guidance

We now know how to find and categorize errors, and how to document them in our OpenAPI definition. Let's now look at some things that frontend and backend developers need to consider when it comes to dealing with errors during their implementation.

### 19.7.1 Frontend Developer

First of all, a frontend developer integrating an API should be aware that things can go wrong, and that things can ungracefully fail, even if the backend developers have done their homework. They can and should expect the API to return the errors specified in the OpenAPI definition, but they must implement some generic error handling as well to fall back upon if the error handling code they wrote doesn't cover the API response.

For client errors, there can be two different sources:

- The client-side code, which may contain bugs. Fixing those is obviously the responsibility of the frontend developer.
- The end user interacting with the application may provide invalid inputs. Sometimes the client-side can detect these problems, other times it has to rely on the API.

Let's look at one example of an invalid input error. The `POST /register` endpoint expects, among others, an `email` field. The client-side HTML code contains an input box for an email address. The client-side Javascript code reads the user's input and sends it to the API. Here are some things that might happen:

- Due to a misunderstanding or a typo in the code (maybe Max developed this on a Friday afternoon, exhausted from his week), the client-side code sends the email address in a field named `e-mail`. The server indicates as a 400 ("Bad request") error that the input is invalid as the `email` field is missing. This type of error is relatively easy to spot because it prevents the happy path, too. The frontend developer must fix this issue.
- The user enters an empty or invalid email address and submits the registration. The client-side code sends this to the API and the API responds that the input is invalid because it contains a malformed value for the `email` field. It's the frontend developer's responsibility to relay that information to the user so that they know what they did wrong and can rectify the issue. For this type of error, the frontend developer can also do additional client-side validations to spot invalid inputs, so that the UI can provide immediate feedback to the user without waiting for the API to respond. We'll briefly touch upon how OpenAPI can help with these validations in chapter 20.
- The user enters an email address that already exists. The API responds that the input is invalid because the email address is already registered. The frontend developer must properly relay that information to the user so they can either switch from registration to login or change the email address. For this type of error, the frontend developer can do nothing else because there is no way that this problem could be recognized on the client-side.

For server errors and network errors, the situation is different. The fault is clearly outside of the client scope, so neither the frontend developer nor the end user can do anything to fix the issue. However, the frontend developer should make sure that their client-side code relays the information that an error has occurred to the user<sup>13</sup>. They need to decide, however, much information to show to a user. Since server errors can be transient failures, an option to retry the request is helpful.

### 19.7.2 Backend Developer

For the backend developer, it's an important realization that error handling code can exist in various places throughout the codebase:

- If the framework has built-in functionality for things like input validation, such as our code from chapter 13 generated with Swagger Codegen does, the error handling happens outside the usual developer's codebase and in a library or dependency.
- When something fails unexpectedly, the code throws an exception or low-level error. If there's no exception handling, the framework will apply or its own error handling or fail in unexpected ways.
- The developer wrote conditional code, such as if-else-blocks, which may indicate success or failures. This code can be adjacent to the code that generates the API response, in which case they can similarly generate the error response. Other times, the code is hidden deeper in the architecture, for example in a service class. In this case, it's their responsibility to carry the error forward to the code that generates the response.

To summarize, the backend developer has two responsibilities:

- Whenever they check for success or error conditions themselves, they also need to generate error responses. If the code that determines whether a request is successful or not is spread throughout the codebase, a common approach could be to throw a custom exception and then centralize error handling in exception handling code (a "catch" block).
- Wherever error handling is outside their control, they should learn about their framework's default behavior and if and where they can change it. If they can't change it, they should document it for their API consumers - which is what we did with the OAS tools format in PetSitter.

The question if it's the backend developer's sole responsibility to write error handlers or whether they get support from their framework, does not have a general answer, because different frameworks vary in the depth of functionality. Generally speaking, a machine-readable API description can help with validation, so a framework that understands OpenAPI can handle some client errors but not all.

As an example, take the PetSitter backend that uses OAS tools. It can automatically detect whether the user provided an `email` field and whether that field contains an email address (although we still have to teach it that trick, which we'll do in chapter 20). If any of these checks fails, the Node.js code automatically returns a 400 error. However, since we wired up the database in our service code, we have to check there if a user with that email address already

exists and throw the 400 error ourselves. We also need to handle all 403 and 404 errors ourselves, as they're related to our database code and business logic.

When adding authentication to the API in chapter 14, we learned that the OAS tools automatically check for the presence of the `Authorization` header if we define operation security in OpenAPI. It is another thing that the framework takes care of, and it returns a 401 ("Unauthorized") response if a necessary authentication parameter is absent.

## 19.8 Summary

- Error handling is an essential part of the API definition. Useful error responses and their documentation helps API consumers such as frontend developers understand and solve problems as well as relay problems related to user input to the end user of an application integrating the API.
- It's possible to roughly categorize errors in client errors, server errors and network errors. Network errors are outside the scope of APIs, but API consumers must recognize their occurrence. Server errors can happen for every API call due to problems with the server-side code or the infrastructure. Client errors are related to client-side bugs or user input. They can be different for each API operation and they can be further categorized into invalid inputs (400), permission errors (403), and requests for non-existing resources (404).
- The OAS tools provide input validation with a custom error format. We analyzed the format and documented it in the OpenAPI definition for PetSitter. Then, we added a reference to this schema as a 400 response to all operations with potential input errors.
- For errors not handled by the OAS tools, we looked at an open standard for describing errors called `problem+json` and added its specification to the OpenAPI definition for PetSitter. Then, we added a reference to this schema as a 404 response to all resource endpoints and a 403 response to all operations that may require specific user permissions (e.g., roles).

# Improving input validation with advanced JSON Schema

20

## This chapter covers

- Changing an API and its impact
- Supporting multiple versions of APIs/schemas
- Avoiding breaking changes

José noticed that the change of moving from `dog` to `pets` had a knock-on affect with Max the frontend developer. If Nidhi had released the backend change before Max was ready, a large portion of the PetSitter website would have stopped working. But, because the team communicate often this was hardly a problem, and they could easily coordinate how to roll those changes out. But what will happen when the API is made public? How can those breaking changes be avoided? And if they'd made these changes after outsourcing the mobile team, how would that be handled?

In the team's case, they have a small and easily managed API between the frontend and backend as well as between José's team and the mobile developers, but as it grows to include more consumers (and more developers) things will get harder to manage. After they release the API to the public they won't have that control anymore and will need to be considerate when making changes.

The team decided to look at this as an example of a breaking change and explore what sorts of actions they should take in future and if there is anything they can do ahead of time to prepare for that eventuality.

In this chapter we're going to look at breaking changes and the different ways to handle them. Ultimately we want to avoid breaking changes entirely (it just makes for a better world), so a portion at the end will be a few tips on how to do that.

## 20.1 The problem

In chapter 17 we described the feature of adding multiple pets into the OpenAPI definition by adding the Pet schema. That on its own isn't a breaking change, but modifying the Job schema to use `pets` instead of `dog` was.

The goal is to introduce this change without breaking the consumers of the PetSitter API.

It's also worth keeping in mind the cost for the developers to accommodate this change. Specifically how to avoid adding code that isn't directly related to the core business (of pet sitting in this case), ie: extraneous infrastructure code.

At the end of this chapter the team will describe the dog-to-pets change with the least impact on consumers of the API as well as the development team.

The problem we'll focus on is taking the API from...

### **Listing 20.1 Describing the Job schema with dog**

```
openapi: 3.0.0
# ...
components:
  schemas:
    Job:
      type: object
      properties:
        #...
        dog:
          $ref: '#/components/schemas/Dog'
```

To now also include the following, but without breaking API consumers...

### **Listing 20.2 Describing the Job schema with pets**

```
openapi: 3.0.0
# ...
components:
  schemas:
    Job:
      type: object
      properties:
        #...
        pets:
          type: array
          items:
            $ref: '#/components/schemas/Pet'
```

## 20.2 What is a breaking change?

We define a breaking change as anything that requires consumers to do something. If they don't update their integration then they'll experience degraded service, bad end-user experience or quite commonly—downtime.<sup>13</sup>

Breaking changes, in other words, break consumer integrations. In the world of APIs and in particularly JSON and XML APIs, a breaking change is usually one of the following changes...

- Removing a field from a response
- Changing a field in a request or response
- Adding a required field to a request

Things that aren't typically breaking changes are...

- Adding a new field to a response
- Adding an optional field to a request
- Removing a field from a request

There may be nuances and some consumers may be a little more finicky, but the above generally hold true for the most common types of integrations.

The best way to think about breaking changes is to imagine the consumer what they'll need to do if you make a change. Are you asking the consumer to do something more? Or are you taking away something the consumer might be relying upon. Building this intuition takes time, but those are the fundamentals.

Let's go release a breaking change...

## 20.3 Releasing a breaking change

For breaking changes it falls to the API designer to figure out how to release and describe those changes. So that it will impact consumers the least as well as cost the least to maintain in the future. The API designer should consider both the consumers of the API and the development team.

Nidhi has been put in charge of releasing this API change, she'll be both the designer of the API and it's implementer. She'll take a look at the following approaches...

1. Coordinated breaking change
2. Multiple API Versions
3. Media types for schema versions
4. Add and deprecate

### 20.3.1 Coordinated breaking change

Internal APIs aren't released to the public and typically the producers have direct communication with all the teams and consumers involved. For those APIs, changing the API without backwards support may be the cleanest approach. However it is not free, as the hidden cost is coordinating that change.

Some cases will be easily coordinated, such as when the consumer rarely uses the API (a consumer that generates a weekly report). Some will have regular release cadences and so coordination would around be fitting into that schedule. And some cases will need special attention to coordinate. The challenge is when both the consumer and the producer (ie: server) need to be released at the same time. Because of the downtime that might incur, organizations have been known to do these types of deployments at midnight to minimize that impact.

For public APIs we have the general assumption that somewhere a consumer depends on our service and will have downtime if the API has a breaking change. This can be better supported with metrics to further identify how many consumers would be impacted.

Nidhi considers the impact of the change first, to see how large it actually is, changing from this...

### **Listing 20.3 OpenAPI with dog field**

```
Job:
  type: object
  properties:
    #...
    dog:
      $ref: '#/components/schemas/Dog'
```

To this...

### **Listing 20.4 OpenAPI with pets field**

```
Job:
  type: object
  properties:
    #...
    pets:
      type: array
      items:
        $ref: '#/components/schemas/Pet'
```

Would impact all operations that references the Job schema, which so far include...

- GET /jobs
- POST /jobs
- GET /jobs/{id}
- PUT /jobs/{id}
- DELETE /jobs/{id}
- GET /users/{id}/jobs

That's quite a few operations!

If the consumer's code base is well structured the impact may not be so large but without looking at their systems, it's only a guess. The size of the API change is a rough estimate at impact.

A Coordinated Change approach works best when you're able to asses the impact on consumers and that impact is less than the cost of coordinating the change.

### 20.3.2 Multiple API Versions

Nidhi now considers the purist's approach.

"This is a breaking change, so it must mean a new version of the API", she thinks out loud. What does she mean by that? She wants to communicate to consumers that when the version of the API changes, it means that it's incompatible with the previous version. So consumers of version 1 will need to change their code before they can use version 2.

However unlike code libs (such as those from maven, npm or Github) which can easily store multiple versions of the libs, hosting multiple versions of an API is far more costly. Both in resources, code setup and maintenance.

Let's run through this example to see how we can host multiple versions of an API...

- Using different base paths, eg: /v1/..., /v2/... etc.
- Using a query parameter ?version=1, ?version=2, etc.
- Using a header Version=1, Version=2, etc.

Both versioning with query parameters and headers are quite similar, in that they can be for the entire API (as in all operations) or they can be used for individual operations. We'll take a look at that next, but for now let's look at the changing the base path as that will always affect the entire API.

The first option of changing the URL can often be seen in wild (although rarely does the version go beyond number three). Because of changing the base path, all operations will have a different URL. Making consumers choose which version of the entire API they need.

To describe that in OpenAPI there would be two separate OpenAPI definitions, one for each version.

Version one would have the original API definition, from before the feature...

## Listing 20.5 Version one

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: Petsitter API
servers:
  - url: https://petsitter.designapis.com/
paths:
  # ...
components:
  schemas:
    Job:
      type: object
      properties:
        #...
        dog:
          $ref: '#/components/schemas/Dog'
```

And version two of the API would include a new base URL as well as the breaking change(s)...

## Listing 20.6 Version two

```
openapi: 3.0.0
info:
  version: 2.0.0
  title: Petsitter API
servers:
  - url: https://petsitter.designapis.com/v2 ①
paths:
  # ...
components:
  schemas:
    Job:
      type: object
      properties:
        #...
        pets: ②
          type: array
          items:
            $ref: '#/components/schemas/Pet'
```

- ① The new base URL with v2.
- ② The breaking change.

As you can see this is already starting to be a lot of work and we haven't considered how the development team (**cough** Nidhi herself!) will handle this. With all of these changes what is gained and what is lost?

The pro of this approach is that the existing API remains exactly as-is, with no changes to it. The con is the cost involved. The development team have to effectively manage two separate APIs... the documentation has to include both versions and possibly a migration guide to get from version one to version two. These are non-trivial costs.

The only time we'd recommend this approach is if the new version of the API is so radically

different, that it is practically a new API *and* you still want this new API to fall under the original name/brand/domain. For other cases there are cheaper approaches. We can start by looking at versioning individual operations instead of the entire API

## VERSION STRINGS

Before we continue, let's take a quick look at version strings to clear that up. We've been mixing and matching different strings to represent a version. 2.0.0, v2, 2, to name a few. Which is better and why? In chapter 11 we made a reference to Semantic Versioning which is the dominant standard for a version string (there are others, but not really worth mentioning for APIs).

Semantic Versioning (or semver for short, see: [semver.org/](http://semver.org/)) communicates three core things. Breaking changes (Major changes), added features (Minor changes) and bug/security fixes (Patch changes). Using three numbers, in the following format...

<Major>.<Minor>.<Patch>

Eg: 2.3.0 has 2 for Major, 3 for Minor and 0 for Patch.

To compare which version is newer we look first at which has the higher Major number, if they're the same then which has the higher Minor and finally if those are also the same, which has the higher patch.

Some examples:

- 2.0.0 is greater than 1.99.99
- 3.1.10 is greater than 3.0.18

Changes to the Major number indicate that there is a breaking change, so caution should be used when upgrading/migrating to that version. Changes to the Minor number indicate new features but no breaking changes. And finally a change in the patch number indicates bug or security fixes.

Versions are linear, so when the Major number changes, it resets the Minor and Patch numbers, as a breaking change encompasses added features and bug fixes. Example...

*If the current version is 1.2.3 and we want to communicate a breaking change we'd bump the Major number and end up with 2.0.0 (not 2.2.3).*

Should we use semver for everything? Well one place that it doesn't work well is in your URLs, eg: `api.example.com/2.0.0/foos` when it changes to `api.example.com/2.1.0/foos` then consumers will need to change their code to use it, even though the version communicates no breaking change which is kind of ironic! So instead of using semver in the URLs, it is a better

idea to use the Major version only, eg: `api.example.com/2/foos` which allows the URL to remain the same for all Minor and Patch version changes.

One last point to make. Noticed how that URL didn't look like a versioned URL? Numbers might be confused for IDs or other parameters, so a final flourish is to prefix it with `v` (for version of course). Making the URL look like it's got a version in there, ie: `api.example.com/v2/foos`. Much better!

Let's jump back into versioning APIs...

### 20.3.3 Using Media Types to version operations

Instead of creating entirely new APIs each time there is a breaking change, one could instead create new operations only and version them accordingly. When Nidhi looked at using query parameters or headers to version the API she clearly saw that they could also be used to version the operations only, since they aren't necessarily API-wide.

A query parameter is a quick way to version, so changing the original into...

## Listing 20.7 Version the operation with a query parameter

```

openapi: 3.0.0
# ...
paths:
  /jobs:
    parameters: ①
      - name: version
        in: query
        schema: ②
          type: number
          default: 1
          enum: [1,2]
    get:
      summary: List All Jobs
      responses:
        '200':
          description: |
            The response will depend on the the `version` parameter. ③
          content:
            application/json:
              schema:
                oneOf: ④
                  - $ref: '#/components/schemas/Job' ⑤
                  - $ref: '#/components/schemas/Job2' ⑥

components:
  schemas:
    Job:
      type: object
      properties:
        #...
        dog:
          $ref: '#/components/schemas/Dog'
    Job2:
      type: object
      properties:
        #...
        pets:
          type: array
          items:
            $ref: '#/components/schemas/Pet'

```

- ① We add a parameter to our path (affects all operations under it).
- ② We can prescribe a default value and limit the options to only 1 and 2.
- ③ We need to somehow tell the consumer how this works.
- ④ We have to say it's either Job or Job2, and rely on the description to tell the consumer that it's based on the `version` parameter.
- ⑤ Our original schema.
- ⑥ Our new schema

With this approach we have a way to version the operation by using the query parameter, `version`, but we still need to tell the consumers of this novel approach. There is a better (read: more semantic) way of doing this, one which can use OpenAPI's structure and HTTP semantics to better communicate this. Using custom media types.

A media type is a way for consumers to ask for the content to be in a specific format. We've seen it used for XML (ie: `application/xml`) and commonly JSON (ie: `application/json`) but we can use a nifty trick to request (or send) data in even more specific terms, say a specific version of a schema.

To do that we can use our own media type. An example of a custom media type is `application/vnd.some.cool.example+json`. Anything between the `vnd.` and `+json` is for us to use. To version our schema we could create one like this... `application/vnd.petsitter.v1+json` Where we specify us, petsitter, and a version—`v1`.

**NOTE**

A media type has the following structure...

**Listing 20.8 Structure of media types**

```
type "/" [tree "."] subtype [ "+" suffix]* [ ";" parameter]
```

By using the `vnd.` tree we're able to use our own media types in a standardized way. Other trees include standard (no prefix), personal/vanity prs. and unregistered x... The vendor tree is the one designated for this purpose of custom data types for public consumption. We're also using the `+json` suffix, so that coding libraries can more easily parse the data. Without that (eg: `application/vnd.example.cats`), most code libraries would need to register the media type and associate it with a base format. But by adding a suffix, most serializers will know how to parse it, eg: treating `application/vnd.example.cats+json` as `application/json`. There are a few standard suffixes, the most common being `+xml` and `+json`.

With our custom media types, let's version our new change of dog-to-pets with...

## Listing 20.9 Versioning the change with custom media types

```

openapi: 3.0.0
info:
  # ...
paths:
  /jobs:
    post:
      # ...
      requestBody: ①
      content:
        application/json: ②
        schema:
          $ref: '#/components/schemas/Job'
        application/vnd.petsitter.v2+json: ③
        schema:
          $ref: '#/components/schemas/Job2' ④
    get:
      summary: List All Jobs
      responses: ①
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  items:
                    type: array
                    items:
                      $ref: '#/components/schemas/Job'
            application/vnd.petsitter.v2+json:
              schema:
                type: object
                properties:
                  items:
                    type: array
                    items:
                      $ref: '#/components/schemas/Job2'

      # ... Other changed operations left out ⑤
      Job:
        type: object
        properties:
          #...
          dog:
            $ref: '#/components/schemas/Dog'

      Job2: ④
        type: object
        properties:
          # ...
          pets:
            type: array
            items:
              $ref: '#/components/schemas/Pet'

```

- ① Schemas in both requests and responses can have different media types.
- ② The older version with the standard media type.
- ③ The new version with the custom media type.
- ④ The new Job schema, titled Job2.
- ⑤ Each operation would need to be updated to include both media types!

That's a fair amount of changes to the OpenAPI definition, considering we still left out all the other operations affected. Despite the amount of work done in the definition, it may mean less work for the consumer. They can decide which versions of the operations they want and if they do nothing, they'll continue to use the older versions.

Something that also needs to be addressed in this instance... what does v2 actually mean? Is it the entire API or just the operation? For example if we update another operation do we move to v3 across everything or only update that operation? Do we update the operation to v2 so that it is only the operation that gets updated?

These are not well defined areas. Versioning by schema is an easier plan for consumers to manage, but it can still be a challenge for producers to manage, despite the support we see in OpenAPI for it. How teams version their operations would need to be defined and documented.

This leaves the last option we'll explore, and we intended to leave the best for last. Never break the API...

## 20.4 Add and deprecate

Managing multiple versions of the API or even of its operations is a huge overhead and should rarely be the first choice when dealing with a breaking change. Instead consider the simpler (in almost every way) approach of adding the new feature and deprecating the older feature—which turns it from a breaking change, into a non-breaking change.

Let's see what the PetSitter team would do if they wanted to keep their consumers happy by not breaking the API...

### Listing 20.10 Adding pets to the schema

```
components:
  schemas:
    Job:
      type: object
      properties:
        #...
        dog: ①
          $ref: '#/components/schemas/Dog'
        pets: ②
          type: array
          items:
            $ref: '#/components/schemas/Pet'
```

- ① There is the old feature, as-is.
- ② And there is the new feature added on top of that.

That is it. Well, mostly.

The only thing left is to tell the consumer how to use this schema. Since there is now a bit of ambiguity. We will need to add a description...

### Listing 20.11 Adding a description to Job schema

```
components:
  schemas:
    Job:
      type: object
      properties:
        #...
        dog: ①
        allOf: ①
        - $ref: '#/components/schemas/Dog'
        - description: |
          This is deprecated, prefer to use `pets`.
          If both exist, `dog` will be ignored and `pets` will be used.
      pets:
        type: array
        items:
          $ref: '#/components/schemas/Pet'
```

- ① We need to use `allOf` to combine a description with our `$ref`.
- ② The new behaviour around `dog`, that still allows older consumers to use it.

We can do more though, we can indicate to tools (and documentation builders) with the `deprecated` property. It'll better highlight and discourage that field from being used...

### Listing 20.12 Adding the deprecated field

```
components:
  schemas:
    Job:
      type: object
      properties:
        #...
        dog:
          allOf:
          - $ref: '#/components/schemas/Dog'
          - description: |
            This is deprecated, prefer to use `pets`.
            If both exist, `dog` will be ignored and `pets` will be used.
          deprecated: true ①
      pets:
        type: array
        items:
          $ref: '#/components/schemas/Pet'
```

- ① The `deprecated` flag, deprecating this field.

This is the approach to take for adding the new feature of multiple pets without breaking older consumers.

It does beg the question of when this not possible, or an even better one... what can be done to encourage these types of changes in the future?

### 20.4.1 Objects for the win

The biggest thing to do is to use objects (the JSON kind, ie: maps) wherever you can. Objects are awesome, you can always add more fields without needing to remove older ones, it allows the types of changes that aren't breaking.

As a simple example, always wrap your arrays inside an object.

Here is an example...

#### Listing 20.13 Example array response

```
[ "one", "two", "three"] ①
```

- ① A naked array, not wrapped in an object

When we want to add pagination we'd have to break the API to make that change like so...

#### Listing 20.14 Changing an array to a paginated list

```
{
  ①
  "total": 3,
  "items": [ "one", "two", "three" ] ②
}
```

- ① Our root schema has changed from an array to an object
- ② Our original array is still seen here, nested under `items`

Instead, even if we may never need pagination, it helps for us to wrap arrays inside objects and so give us the opportunity later on to add more data to the schema.

#### Listing 20.15 Example array wrapped in object

```
{
  "items": [ "one", "two", "three" ] ①
}
```

- ① Wrapped in an object for future proofing

## 20.5 Summary

- Changing APIs are inevitable but the way in which they change can keep or lose consumers.
- A breaking change is one that requires consumers to do work or else their integration will begin to fail. Avoiding them is the best approach.
- A coordinated breaking change is one where all the stakeholders agree on a time and plan for the breaking change to occur. This can work well for internal APIs but up to a limit before the coordination takes too much effort compared to introducing the breaking change.
- OpenAPI supports multiple versions of an API by using multiple definition files, typically this would involve changing the URL to use a different base path, ie: v2.
- One can use custom media types to version the entire API or with a bit of finesse, only the individual operations. OpenAPI supports declaring different media types with different schemas in both the `requestBody` and `response`s.
- Only ever adding fields to schemas, is a great strategy for avoiding breaking changes. This approach relies on using objects wherever possible, avoiding unwrapped arrays or other primitives.
- Schemas in OpenAPI can have the `deprecated` flag, which indicates that the schema (or sub-schema) should be avoided as it'll be removed one day.

# *Beyond the application: what's next for our API?*

21

## This chapter covers

- Going through the pre-release API checklist
- Touching upon topics not covered in this book
- Releasing the PetSitter API

After the success of the mobile product, José is now considering releasing the API to the general public. He wants to allow others to build on this platform and give the platform more exposure to different audiences and market places.

As excited as José and the team are, they should consider what they'll release and how they'll release it, before jumping in.

## 21.1 Pros and cons of a public API

There are several challenges around releasing an API to the general public. The chief problem is surely that consumers cannot change their code immediately after the API has a breaking change. Add to that there doesn't exist a direct communication channel with them and we're left a pickle jar and no way of opening it. Topics like security, testing and monitoring are also worthy of consideration to keep the API healthy and engaging. With all these challenges, it may be a little wearisome to release! But there are benefits too, increased usage of the platform... new and innovative use cases not considered before. We, as authors of an API book, are naturally a little biased but even so—releasing an API is generally good.

Looking at the pros and cons list of what goes into a public API.

### Pros

- Access to more people using the service
- Allows for more innovative uses of the service
- Once invested in, people won't quickly change their APIs

## Cons

- Security surface area increases
- Changing the API has increased overheads

For the majority of services that want to release their API to the public, it comes as a trade-off. You'll be giving up full control of the end user's experience with your platform while at the same time greatly increasing the reach that your platform will have. In most cases (from our personal experience) releasing an API has had a positive impact on a service.

José has decided that his core value is in the people who use his system, the more people who use it (ie: engagement) the more valuable it becomes. By opening up the API he allows more engagement while at the risk of others creating better "portals" to his platform. It's a risk he needs to balance and account for.

**NOTE**

**Example: Bob could build a website that combines PetSitter with Airbnb, which may be a better experience for users. This is a potential risk for José's business which may not offer all those features. But in practice this can easily be made to benefit both parties**

We'll look at what we've covered in this book so far for releasing an API and what we *haven't* covered that is still very much worth exploring.

## 21.2 The problem

José wants to make sure he's aware of all the bits and bobs that he and the team need to cover before releasing the Petsitter API. He's not sure if the API is in a good state to release, he's not sure if he should release the entire API or just a part of it. In short he's not sure.

In this chapter the problem we're tackling is getting the confidence to release the PetSitter API. We'll tackle it by doing due diligence and looking at the different aspects of an API release. We'll start by drawing up a checklist to go through and then consider the different items in that checklist.

We'll end up with a completed checklist and José will have the confidence to decide when and how to release his API.

What should we consider when releasing an API?

### 21.2.1 The checklist

To get the checklist, start with the end goal and work your way backwards.

José tries to imagine the ideal situation. The API is released and is well received by all, he knows this because he has metrics showing good growth. Developers are reaching out via email less often as the error messages are clear and show how to solve the problem. He and his team are confident in making changes as they know what is considered a breaking change or not as well as what to do if there is a breaking change. There isn't much fear from nefarious folks trying to attack the system as their security is up to scratch. And new users are excited to integrate with the API as they know what's available now and what will be available in the future.

José sure knows how to dream big!

Let's break down the dream into some areas worth investigating some more.

Rough checklist...

- Set up metric collection
- Get secure
- Get an API change strategy
- Provide documentation / communication
- Provide an API roadmap

This is an excellent start, although some things that were more implicit in that dream and things that may not be obvious when thinking about the API. For instance, it may be obvious, but if the API isn't working... it's probably not a good idea to release it to the public! We'll add the following to the list...

- Get your API working
- Get your API consistent
- Set up API monitoring
- Determine what to release

At the ends of this chapter we'll fill out the following table...

**Table 21.1 The pre-release API checklist**

Item	How to deal with it?
Get the API working	
Documentation	
Build an API roadmap	
Establish communication channels	
Set up metrics / monitoring	
Improve security	

**NOTE** Reminds me of a fun little game to play. Take popular stories and tell them in reverse, for example:

*Godzilla in reverse is about a benevolent lizard that rebuilds Tokyo then moon walks back into the ocean.*

**Use this game with caution!**

## 21.2.2 Get your API working

In this section we want to talk about the functional side of the API and make sure it's ready for a more general release. Typically this involves testing both manually and with automation. Testing is how we know our API works.

We'll only gleam over testing in this section as it is a much larger topic and unfortunately outside the scope of this book. The outcome of testing is confidence that the system works as you expect it to. We'll take a look at two different styles of testing and how they work together to form a testing strategy.

The two styles are unit testing and end-to-end (e2e) testing.

### UNIT TESTING YOUR API

Unit testing aims to limit your test space to one thing or "unit". In software we think of units as functions, in APIs we can consider the operation (ie: path + method) as our unit under test. How you define a unit is still up to you and we may be abusing the term here to refer to a single operation but it works well for us.

Testing your one operation may require you to isolate that operation to effectively test it. Consider your database and making other 3rd party API requests, could those potentially interfere with your testing?

Unit testing your API can take the following approach:

- Test by putting in a request and asserting that the response is correct
- Test as many edge cases as you have an appetite for
- Mock or stub out as much as you care to, to best isolate your function.

Ideally these unit tests should be fast to run and not depend on any external network calls.

Every programming language will have different libraries to help with API testing, as well as different frameworks for running unit tests—so we won't go into too much detail around that.

For our Node.js APIs we use a library called supertest that is based on an HTTP client called superagent. It works well enough for unit testing those API calls. Here is an example to help illustrate what we mean.

### Listing 21.1 Example unit test in Node.js using supertest

```
const app = require('../our-expressjs-server.js')
const request = require('superagent')

describe('POST /foos', function() {
  it('should create a new Foo', function(done) {
    app.mockDb(true)
    request(app)
      .post('/foos')
      .send({ foo: 'A Foo' })
      // Expect status code 201
      .expect(201)
      // Expect a JSON response body
      .expect('Content-Type', /json/)
      // Expect the response body to have fooid
      .expect({ fooid: 'abc' }, () => {
        // Should also assert the DB was updated
        expect(app.mockDb.foos).toBe([{foo: 'A Foo'}])
        // Finish the test
        done()
      })
    })
    // ...
  })
})
```

The above is loosely written and only meant to convey the gist of how unit tests can be written for operations. We hope it captures the idea of it.

Link to supertest: [github.com/visionmedia/supertest](https://github.com/visionmedia/supertest)

### 21.2.3 End to end testing

With unit tests, we are testing that the code works the way we expect it to for each operation. But what we don't really capture is if it works when all the units are composed together. For that we need to do end-to-end testing. This is testing the entire application much like our consumers would. Both for the web interface and for the public API.

The first and most straightforward way to test the API is to sit down and use it.

An important part of end-to-end testing is use the API as your users would, as close as possible. Fortunately HTTP APIs are incredibly standardized and most (if not all) HTTP clients have the same behaviour. That means you can test with any HTTP client. Sit down, grab your favourite API client (curl, Postman, swagger-ui, etc) and make some requests against the API.

José isn't a that technical that he's comfortable working with many of the features of an API client, but with the help of his team he can use one to see what data is provided and returned—which is enough to think through some edge cases that his consumers might

encounter.

With this manual, exploratory testing, the team are more able to empathize with the consumers... but running through the same flows over and over again won't scale well and so automation is needed. Particularly around areas that are well established. These automation suites are a curated tool. One that the team will need to maintain and cultivate for it to continue to yield value.

We're going to show you a little tool that is surprisingly useful in testing small to medium sized APIs after which we'll link to other tools that can handle larger, more complicated flows.

## STREST

The first flow José considers is to create a new job and then fetch the list of available jobs. His goal is to see what the consumer would see and to put himself in the consumers shoes. After create a job, he'll expect that job to be in the list returned. For that we'll need to assert that the list contains an the automatically generated ID. The tool to help us is strest.

Strest is an open source tool that consumes a YAML file which describes a series of API requests to make, and then asserts that the responses are correct. What makes it even more useful is the ability to chain requests together. Which suits our purposes perfectly, we can make one request and capture the generated ID and use it in the assertion of the next request.

Link to strest: [github.com/eykrehbein/strest](https://github.com/eykrehbein/strest)

Here is a simple example:

## Listing 21.2 Simple strest example

```

# petsitter.strest.yaml
version: 2
requests:
  createSession: #... ①
  # ... other example requests left out
  createJob: ②
    request:
      url: '<$ Env("URL") $>/jobs'
      method: POST
      headers:
        - name: Authorization
          value: |
            <$ createSession.content.auth_header $> ③
      postData:
        mimeType: application/json
        text:
          activities: [walk]
          description: A friendly pooch
          ends_at: 2021-01-01T00:00:00
          starts_at: 2021-02-01T00:00:00
    validate:
      - jsonpath: status ④
        expect: 201

  getJobFromLocation:
    request:
      url:
        [CA] '<$ Env("URL") $><$ createJob.headers.location $>' ⑤
        method: GET
        headers:
          - name: Authorization
            value: <$ createSession.content.auth_header $>
    validate:
      - jsonpath: content.description ⑥
        expect: A friendly pooch

```

- ① Left out createSession for brevity
- ② The name of our example request
- ③ Using the response body of a previous example request
- ④ Validating that the status code was 201
- ⑤ Using a header from a previous request
- ⑥ Validating the response body

We can run the above example as follows:

### Listing 21.3 Running the strest example

```
$> URL=http://petsitter.designapis.com/api strest ./petsitter.strest.yml

[ Strest ] Found 1 test file(s)
[ Strest ] Schema validation: 1 of 1 file(s) passed

Testing createSession succeeded (5.69s)
Testing createJob succeeded (5.657s)
Testing getJobFromLocation succeeded (5.644s)

[ Strest ] Done in 17.012s
```

And naturally if it fails we can see some output based on that

### Listing 21.4 Failed output

```
$> URL=http://petsitter.designapis.com/api strest ./petsitter.strest.yml

[ Strest ] Found 1 test file(s)
[ Strest ] Schema validation: 1 of 1 file(s) passed

Testing createSession succeeded (5.748s)
Testing createJob failed (5.692s)
[Validation] The JSON response value should be 201 but was 400

Request: "curl 'http://petsitter.designapis.com/api/jobs'
[CA] -H 'accept: application/json' ...
Response:
{
  "status": 400, ❶
  "statusText": "Bad Request",
  "headers": {
    "x-powered-by": "Express",
    "content-type": "application/json; charset=utf-8",
    "content-length": "205",
    "etag": "W/\"cd-MzLFwusgBhz314tewJyoekBJWvI\"",
    "date": "Fri, 04 Jun 2021 09:20:41 GMT",
    "connection": "close"
  },
  "content": {
    "message": "request.body should have required property 'activities'",
    "errors": [
      {
        "path": ".body.activities",
        "message": "should have required property 'activities'",
        "errorCode": "required.openapi.validation"
      }
    ]
  }
}
[ Strest ] Failed before finishing all requests
```

- ❶ If the response gave us 400 instead of 201 it will cause the test run to fail.

Overall we've had a good experience with Strest, particularly how easy it is to get started testing your API end-to-end.

Link to the full stest suite for PetSitter API For the first part of the book we wrote a simple test suite to know that the farmstall.ponelat.com API continued to work when we made changes. You

can go check out the `.strest.yaml` file over here...  
[github.com/designapis/petsitter/blob/master/.strest.yml](https://github.com/designapis/petsitter/blob/master/.strest.yml)

## POSTMAN'S NEWMAN AND READYAPI

Building on the same idea of end-to-end testing by executing requests and asserting on the response, there are more "production grade" tools to do what strest did in the previous section.

Most notably are...

- Postman's newman: [github.com/postmanlabs/newman](https://github.com/postmanlabs/newman)
- ReadyAPI's TestEngine: [github.com/SmartBear/testengine-cli](https://github.com/SmartBear/testengine-cli)

Both follow a similar pattern.

Using a GUI tool (Postman or ReadyAPI) you can test API by simply making requests, but you can expand on that by writing scripts (newman uses JavaScript and ReadyAPI uses Groovy/JavaScript) to compose together a suite of tests that can be executed from the command line. Which is critical for automating those test runs.

Both have pros and cons. Both are used by many companies to automate their end-to-end testing and build on from the patterns discussed in the strest section with added scripting abilities. This space of end-to-end API testing is still nascent and we look forward to even more powerful ways to verify a working API!

## A WORKING API

With some testing in place José will feel confident that the API does as he expects (as far as he and the team have tested it).

But just because it's tested doesn't mean that anyone will know how to use the API? That's where documentation becomes important. To grossly misuse a famous quotation... if a tree falls in the woods with no-one to hear it, does it make a sound? <sup>14</sup> If an API exists but no-one knows how to use it, does it really exist at all?

An API needs documentation in some form or fashion. The better the documentation the quicker consumers will be able to use it and more effectively too.

Let's talk about docs...

**NOTE**

On an unrelated note... One of our favourite Quality Assurance (QA) jokes goes something like this:

A QA walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders an asdfasdf.

A customer walks in and asks where the bathroom is. The bar bursts into flames and completely explodes.

Lesson? Users often do their own thing, it's worth watching how they use your API and for what purpose!

## 21.3 Documentation

Documentation is the entry point for your consumers to use your API. From the docs, your consumers should learn the following key ideas:

- Can I do what I want? (ie: list all jobs in PetSitter API)
- How can I gain access (URL + authentication/authorization)
- What is the request that I want and what are its details?

This is where OpenAPI shines. It has the nitty-gritty details of the operations captured. That standard allows consumers can read it in many different ways. Either using the documentation provided by the service, using documentation tools like swagger-ui/redocly or something bespoke to the consumer!

As we've seen so far, describing your API gives you 80% of what consumers need to get going. The remaining 20% is really the human touch and the pieces of an API that are more complex.

For that we recommend you consider the following:

- From your API docs, how soon can a consumer start using your API? How much do they need to read, how many pages do they need to visit?
- Can they play with your API? Some APIs are serious (those involving money, launching of rockets, etc) is there a sandbox for users?
- How is your API versioned? Are there different versions available and if so how can you know which to use and how?
- How can they reach out for help? A Contact Us form, a discord/slack group or even an email address will help.

An area that OpenAPI comes up short in, is describing related operations. For example, if you need to make several requests in a particular order to achieve a single goal. For those cases, use the `markdown description` field in your OpenAPI definitions. Or create dedicated pages where consumers can more clearly see those flows.

Consumers will want to explore your API docs. To see what is available and to see if it satisfies

their requirements. Much like the home page of a website, the API docs home page should inspire users to engage the service. It can inspire them by being simple, and making it clear to them what is possible. It is as important to be easy to get started with and comprehensive enough to show that they can solve their problems.

Here is an example showing a good API introduction...

### Listing 21.5 Example of API Introduction

```
PetSitter API ①

Hosted at https://petsitter.designapis.com ②

PetSitter connects Pet owners with Pet sitters using Jobs. ③
A Job describes a Pet and time period for when the pet needs attention.
A Job can be applied for with a Job Application, which can then be approved
or rejected by the owner.

The PetSitter API requires an API token. ④
To get one, signup (over here)[petsitter.designapis.com/app/signup]
then (visit here)[petsitter.designapis.com/app/token] to create a token.
You can then add the header: `Authorization: Bearer <token>` to requests.

PetSitter has only one version at present. ⑤

See list of operations... ⑥
...<swagger-ui>...
```

- ① The name of the API or service.
- ② Where is this API hosted? Multiple servers can be expressed to.
- ③ What is the highlevel description of this service?
- ④ Is there authentication? If so how can I get it.
- ⑤ Are there multiple versions of this API in the wild?
- ⑥ And of course the details of the operations!

#### 21.3.1 Get your API consistent

In previous chapters, we looked at ensuring our API was consistent. We did this for a number of reasons and the one we'll discuss now is about releasing the API.

Before an API is released we have full control and can make changes whenever we choose. As soon as the API is released we have to consider the impact those changes will have on existing consumers. As such it pays to make the API as consistent as possible, not only as good practice, but also to prevent having to "fix" the API after it's released.

Let's consider pagination.

### Listing 21.6 A response that contains a list of items

```
# ...
/foos:
  get:
    description: A list of Foos
    responses
      200:
        application/json:
          schema:
            type: array ①
            items:
              type: object
              properties:
                foo:
                  type: string
                  example: A Foo
```

- ① Note that this response returns an array.

In the above example we have a response that returns a flat array. This might be perfectly adequate if the number of items is stable and low. But what happens in the future if the number of items increases, to the point where we need to use pagination? In that case we'd likely have to make a breaking change. If instead we put in pagination now, before we release we're saving ourselves that hassle.

It is said in software development, that pre-optimization is the root of all evil. And by adding pagination before we need it, we're definitely dabbling in this dark art. So what do we do? How do we balance the idea of optimizing for the future to avoid future pain and keeping our API lean and mean for the present problems it is trying to solve?

Our approach to this conundrum is be consistent. If we have a response that returns a list of items. **ALWAYS** make it paginated. This consistency does two things, one it removes the need to consider each response and ask the same question of, does this need to be paginated. It also gives us the excuse we need to optimize now for a future that might or might not ever happen. The list of Foos may never need to be paginated.

Consistency does another good deed for us and for our consumers. It makes our API predictable. To quote yet another software development koan, aim for the approach of least surprise.

Within reason, strive to make your API consistent *before* you release so as to reduce the amount of future pain. There may be some outliers, requests or responses that you think may need extra complexity to address possible future pains. For those individual operations, you should consider that extra complication carefully. For the operations that are similar (responses that return a list of item), simply opt in to be consistent. It's generally cheaper.

With the schema's consistent there are other ways to be consistent too, and that is in error reporting...

### 21.3.2 Validation and error reporting

Errors are an important communication channel with your users. If users do something wrong, it's your opportunity to help them fix it by giving good error messages. If they're missing a token, tell them they are unauthenticated. If they hit the wrong endpoint, give them some alternatives. Errors can be highly effective at getting your consumers comfortable and confident with your API. As per chapter 19, we learned about how to include both user readable and machine readable errors.

With the errors in place, José will try to break the system and read the errors himself. After each error, José would ask "Am I able to fix this by reading the error?".

Validation play a big role in error messages, there are often small issues with the request that validation can catch. If so, don't be shy in detailing exactly why the request failed. The only exception would be security as sometimes it's unwise to share information.

### 21.3.3 An API Roadmap and exposure index

From the start José has put the API first in the PetSitter service. Using it to decouple the front- and backends from each other. He also used it as way of outsourcing the mobile app development, allowing that team to use the API effectively instead of more complicated options.

This was informally the API's roadmap. The plan José and his team had for the architecture of the system.

Now that José is releasing the API to the public, he is faced with more options. He doesn't have to release the entire PetSitter API right away. Instead he considers what sort of actions he'd like to initially expose to his users. Enough to give his users real value while still allowing him and the team to get comfortable with a public API. The smaller the API is, the easier it is to manage.

José grabs a piece of paper and scribbled down the following endpoints...

#### **Listing 21.7 List of endpoints to consider**

```
/users
/users/{id}
/jobs
/jobs/{id}
/jobs/{id}/job-applications
/users/{id}/jobs
/job-applications/{id}
/sessions
```

From the list José can see a dividing line between the endpoints. Those that are more user specific (such as job-applications and sessions) and those that are more general (such as all jobs).

The smallest sub-set of the API José could release that still delivers value would be to show all

jobs available, as well as the details of an individual job.

### **Listing 21.8 Smallest API to release**

```
GET /jobs
GET /jobs/{id}
```

This doesn't require consumers to have an API token, nor an identity. This information is public and can get gotten anonymously. There is value in exposing this to consumers. But José is interested in more...

The next piece of value to add to users would be to create jobs and job applications—as a user. As well as manage those existing ones and see all jobs of the current user.

This would include the following operations...

### **Listing 21.9 List of user based endpoints**

```
POST /jobs
POST /jobs/{id}/job-applications
GET /jobs/{id}
GET /jobs/{id}/job-applications
GET /users/{id}/job-applications
PUT /jobs/{id}
PUT /jobs/{id}/job-applications
```

Which would all require an API token to authenticate the user creating jobs and job applications. This is closer to what José imagines the initial public API release to be. But then he starts thinking ahead... Such as when users would like to use 3rd party applications that integrate seamlessly with the PetSitterAPI. Or possibly (although unlikely) when enterprises wish to manage all their users under one admin.

For the seamless integration, scoped tokens would help limit the access to resources. Such as a token that can only read Bob's job-applications and cannot read other job-applications nor create new job-applications for Bob. This token is more easily shared than the current token that allows everything Bob is allowed to do. These scopes set the foundation for OAuth which further reduces the friction for end users (not the consumers) to integrate 3rd party applications with PetSitter. We'll call this "Granular scopes".

Where other vendors can build systems on top of the PetSitter API and gain access to manage user's jobs and job-applications. If OAuth was added to, it would allow end users to simply click a button and allow the vendor access. With that level of integration, PetSitter API would surely become a staple of everyone's lives!

### **Listing 21.10 Granularly scoped tokens**

```
POST /tokens {
  scopes: [jobs:read, jobs:write, profile:read, etc] ①
}
```

- ① This is only a doodle and not currently part of PetSitter's API

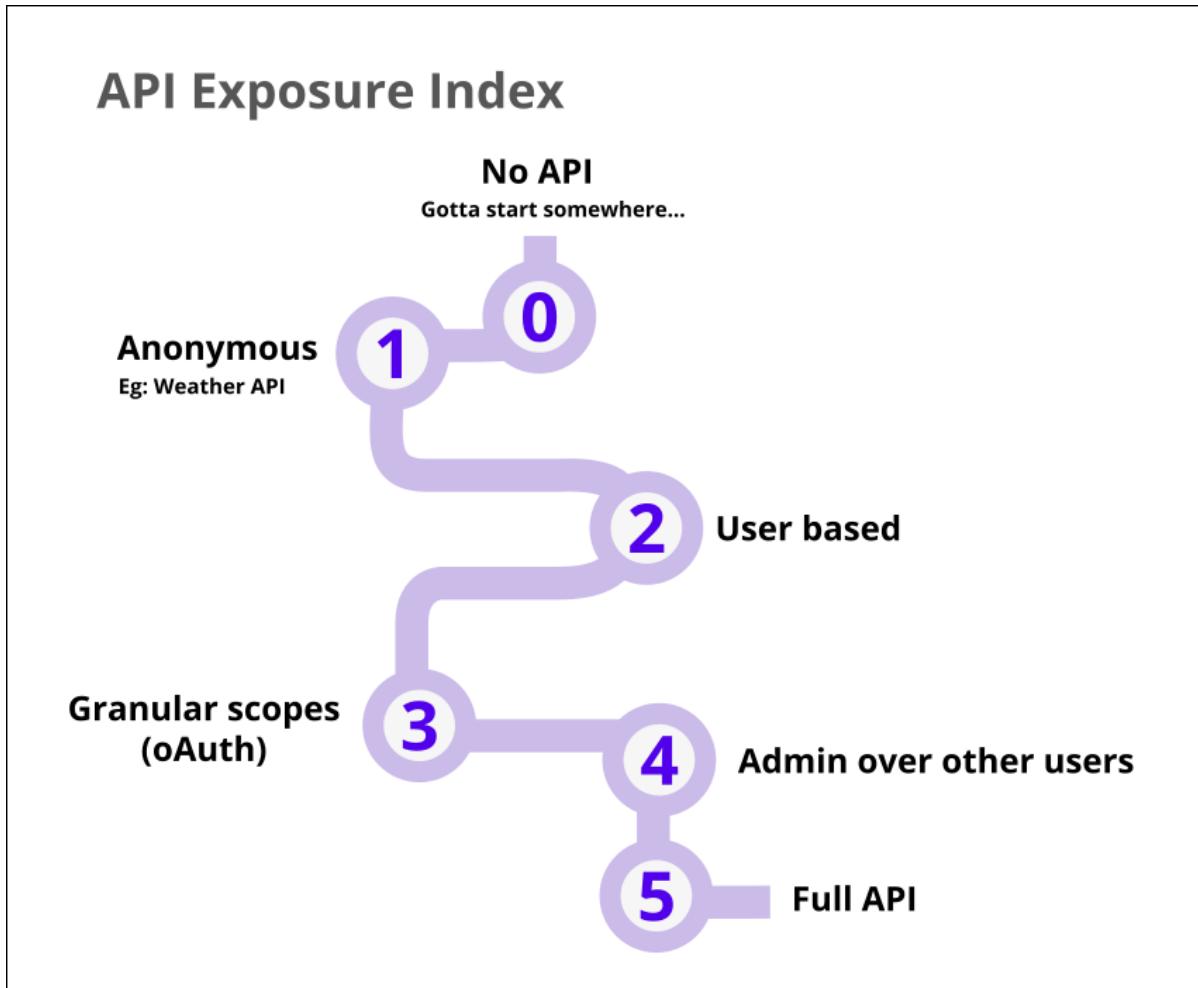
And finally, thinking of larger organizations that wish to manage users collectively. A thing unlikely needed for PetSitter but still worth considering. Those operations would include exposing...

**Listing 21.11 User management side of the API**

```
GET /users
POST /users
PUT /users/{id}
```

This would involve changes to the API and service to allow for some higher entity, such as an organization or company. That entity would have full control of its own users. Viewing all, adding new and making changes to them.

Capturing this progression of exposing more and more of the API, we've put together this diagram. A rough guideline in how much of an API to expose.



**Figure 21.1 Diagram of API integration index**

The key takeaway is that you (or José) don't have to release the entire API all up front. Instead it's better to consider what value you can expose initially and get feedback on that first. It's cheaper and it keeps the rest of your internal API agile while you gain the experience and confidence of having released a public API.

José has decided to release an authenticated API (API Exposure Index #3) and will consider further exposure later on.

### 21.3.4 Get a change strategy

José knows his API will eventually change, hopefully for the better.

And so, from the previous chapter, he had decided on an evolutionary change strategy. Add and deprecate. *Should* a breaking change need to occur it'll be communicated as best as they can, deprecated and removed. All without adding any new versions to the API. He and the team will strive to build their API in a way that doesn't require breaking changes, but there is confidence for both him, the team and the consumers in knowing how a breaking change will be rolled out.

Because of this decision, he's going to be extra careful in what he releases—keeping feature releases as small as possible to minimize the commitment of the API.

### 21.3.5 Security

The security of your API can be tough, like fashion... it's never finished. Ensuring a few basics will get you going, but it's important to be diligent and proactive in your security stance. While the risk of a breach may be reasonably low (it does depend on the circumstances) but the cost of one is most certainly high.

A few pointers:

- Does your API have access control? Prevent unauthorized users from doing things they shouldn't?
- Denial of service attacks (DDoS) can be mitigated with a Web Application Firewall (WAF) or some other protection at the edge of your system.
- Keep your development and staging environments secure, there have been recent attacks for several large breaches because of insecure dev/stage environments.
- Avoid leaking information in your error responses (see previous note on 404 vs 403 status codes). And definitely avoid stack traces in your errors, as they tell a lot about the type of code you're running.

To combat weak security and increase awareness there exists the wonderful Open Web Application Security Project often abbreviated to OWASP. It has resources to help establish stronger security across the web.

To keep things simple, they maintain the Top Ten list of security issues to consider, which we

highly recommend everyone take a look at.

[owasp.org/www-project-top-ten/](https://owasp.org/www-project-top-ten/)

They have set the standard and we should all aim to have an answer to each item in the list, for our benefit as well as others!

#### **IMPORTANT 404 vs 403**

When to hide information for security reasons? Attackers are good at exploiting systems by testing those systems. Poking and prodding them to discover information. In an age of automation we have to be careful about what information we expose.

Consider these resources... 1. /foos/bobs-thing 2. /foos/franks-thing 3. /foos/does-not-exist

If any user requests /foos/does-not-exist the service will return 404 Not Found.

If Bob has access to /foos/bobs-thing but does NOT have access to /foos/franks-thing. What status code should the service respond with when Bob requests /foos/franks-thing?

If you responded with 403 (or 401) you wouldn't be wrong, but there is a problem with responding with those status codes. A 403/401 response code in this case reveals information. It reveals that /foos/franks-thing exists if we compare that to the 404 Not Found we get if we request /foos/does-not-exist.

In this case we should return a 404 Not Found, so as to protect the information that /foos/franks-thing exists from prying eyes.

Take another example.. If Bob is allowed to read /foos/bobs-thing with a GET, but is NOT allowed to update it. Then what status code should we respond with when Bob tries to PUT /foos/bobs-thing?

Here a 403 Forbidden is perfectly acceptable as there is no information leak. Bob is allowed to know about the resource, and the status code does not leak any further information. It is in fact helpful, as Bob will know that he cannot update the resource and perhaps should seek some higher authority.

### **21.3.6 Monitoring your API**

Knowing your API is alive and well isn't a given. During working hours and an active testing/development team you may discover rather quickly if your API isn't responding to requests, but over the weekend you may not discover until many hours or even days after the outage.

To respond to that we, as API providers, often set up some sort of monitoring to make sure that we're notified as soon as our API goes down and we generally have a call list of who should be called and what they're responsible for.

Keeping your API alive is generally a good thing. Know when it goes down and respond quickly. API monitoring tools basically ping your website at one or more endpoints and will send you an email or message as soon as the website stops responding, or responds with an error.

A few of them include...

- PagerDuty.com
- AlertSite.com
- And bunch of others: [geekflare.com/api-monitoring-tools/](http://geekflare.com/api-monitoring-tools/)

**NOTE**

PagerDuty has the best ringtones for when something bad happens, go list at: [community.pagerduty.com/forum/t/pagerduty-ringtones/1536](http://community.pagerduty.com/forum/t/pagerduty-ringtones/1536)

### 21.3.7 Set up metric collection

How will you know if you've succeeded? Or if you're failing? Metrics, metrics is always the answer.

Setting up API metrics is one of those topics we'd love to talk about, but it's outside the scope of this book. Instead we'll point you to the following resources. In our opinion there are a few metrics we're particularly fond of...

1. Unique users.
2. User engagement.
3. Churn rate (those no longer using the API).
4. Acquisition rate (new users).

We look at a monthly period to keep things simple, so monthly view of unique users, churn rate, etc.

S o m e                    f o l l o w                    u p                    r e a d i n g :                    \*

[www.moesif.com/blog/technical/api-metrics/API-Metrics-That-Every-Platform-Team-Should-be-Tracking](http://www.moesif.com/blog/technical/api-metrics/API-Metrics-That-Every-Platform-Team-Should-be-Tracking)

\* [stackify.com/top-api-performance-metrics-every-development-team-should-use/](http://stackify.com/top-api-performance-metrics-every-development-team-should-use/)

Using these (and similar) metrics allows you to track success in a data-oriented way.

For the technical team there are other important metrics that'll improve the quality of the API...

\* CPU usage \* Memory usage \* Mean response time \* Error rate (400's and 500's).

Together you can own your API completely and see the impact of the decisions you make in the future. Go get some metrics!

## 21.4 Release your API

Let's recap our checklist...

**Table 21.2 The pre-release API checklist**

Item	How to deal with it?
Get the API working	Test it and get it consistent.
Documentation	Both reference and additional docs.
Build an API roadmap	Slowly expose your API and provide value.
Establish communication channels	Versioning and other.
Set up metrics / monitoring	Both business and technical monitoring.
Improve security	Use OWASP and be continuously vigilant.

With our checklist complete all that remains is to release the API.

José is as excited as one can imagine. The hard work of consideration and preparation are now complete and a clearer picture of what a successful API is formed. Him and the team have admirably done their due diligence and completed out the checklist.

As they embark on the journey of a public API, we the authors wish you the same joy and success!

## 21.5 Summary

- After deciding to release an API, it's time to run through the pre-release checklist. Going over these now will save you some pain down the way. Before releasing ensure that: Your API is complete and consistent, has correct input validation, has human and machine readable errors, has a version and breaking change strategy and is secure. Also consider how your API will be monitored and what metrics you want to collect.
- Ensure your API works with testing. Decide on what types of testing you'll require from unit testing to e2e testing. Consider the different e2e tooling available to see if they'll suit your process, such as Postman's newman or ReadyAPI.
- Ensure your API is documented (yay for OpenAPI!) and include a human touch, a metric of success can be the amount of time it takes from first reading to the first requests a consumer can make.
- Ensure your API is consistent with common patterns such as pagination. Consider your error messages as they should both be human and machine readable to guide your consumers.
- Consider the API Exposure index when you release your API. How much of your API do you really need to release immediately versus the value added to your consumers. Releasing a smaller API is easier to manage and adapt, while releasing a larger API usually gives more value.

## Notes ch1 - ch10

1. [www.linuxfoundation.org/](http://www.linuxfoundation.org/)
2. [www.ics.uci.edu/~fielding/pubs/dissertation/top.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)

If your name is Ron, you may find fault with this statement. An edge case (of no consequence), is that JSON technically allows for duplicate keys, whereas YAML does not. No JSON parser, I know, implements this. Nor does it make sense to implement it. Ergo, YAML is a superset of JSON. See:

3. [yaml.org/spec/1.2/spec.html#id2759572](http://yaml.org/spec/1.2/spec.html#id2759572)

There are still parts of the specification that are ambiguous, like how deeply you can nest arrays and other

4. odd ball issues, but on the whole it did a very good job.

5. [yaml.org/spec/1.2/spec.html](http://yaml.org/spec/1.2/spec.html).

6. Yeah, Bob Ross rocks!

7. [json-schema.org/](http://json-schema.org/)

Idempotent: Can be applied multiple times without changing the result beyond the first execution, ie:

8. executing it five times has the same effect as executing it once. [en.wikipedia.org/wiki/Idempotence](http://en.wikipedia.org/wiki/Idempotence)

9. The Happy Path is for when things are all valid and good

10. [jsonapi.org/](http://jsonapi.org/)

11. [www.odata.org/](http://www.odata.org/)

12. [goessner.net/articles/JsonPath/](http://goessner.net/articles/JsonPath/)

13. Non-essential functionality, for example logging or analytics, is exempt. These can fail without affecting the user, so there is no need to inform them.

## Notes ch11 - ch21

1. Conway's Law
2. <https://semver.org/>
3. The steps a user takes which don't include any errors or failures.
4. <https://code.visualstudio.com/>
5. It should be: <http://localhost:8080/docs/>
6. <https://www.mongodb.com/>
7. <https://mongoosejs.com/>
8. <https://www.docker.com/>
9. There is a way to express dates in OpenAPI, but it's not a data type. More on that in chapter 20.

If you're curious how CORS works, we recommend starting at Mozilla's documentation:

10. <https://developer.mozilla.org/de/docs/Web/HTTP/CORS>

11. Generating code from an API definition has tooooons of applications, the prospect of it is endlessly exciting!
12. Can't resist using the word
13. Downtime is when a service stops working for a time period
14. [en.wikipedia.org/wiki/If\\_a\\_tree\\_falls\\_in\\_a\\_forest](https://en.wikipedia.org/wiki/If_a_tree_falls_in_a_forest)